

Spieleprogrammierung mit Cocoa und OpenGL

Wolfgang Reminder

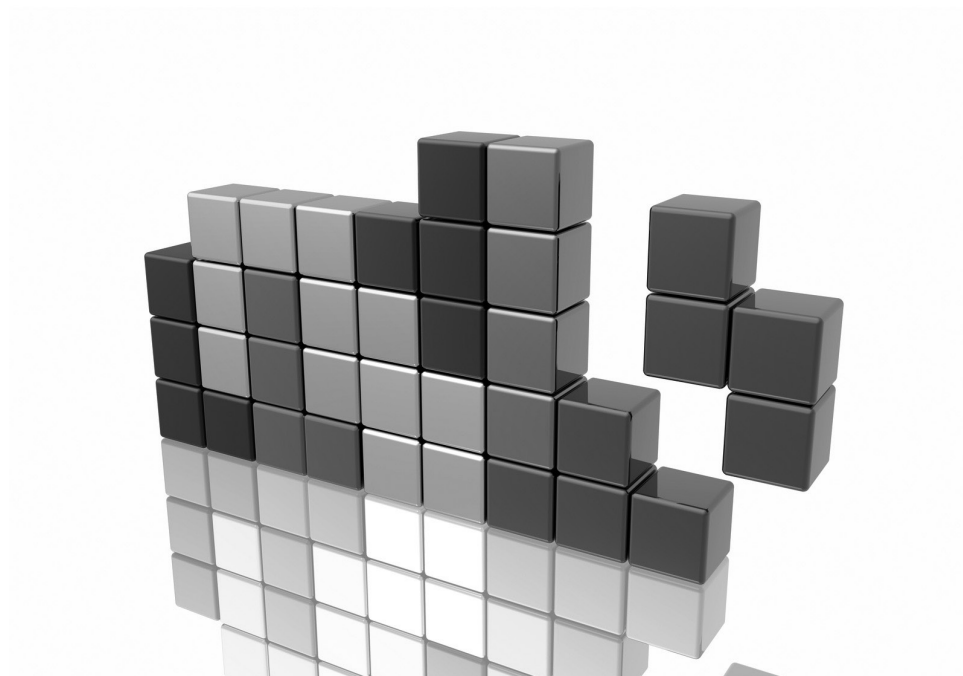



SMART
BOOKS

Wolfgang Reminder



Spieleprogrammierung mit Cocoa und OpenGL



Spieleprogrammierung mit Cocoa und OpenGL

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Copyright © 2009 SmartBooks Publishing AG

ISBN 13: 978-3-908497-83-7

Lektorat:	Jeremias Radke
Korrekturat:	Dr. Anja Stiller-Reimpell
Layout:	Peter Murr
Satz:	Susanne Streicher
Covergestaltung:	Johanna Voss, Florstadt
Druck und Bindung:	Himmer AG, Augsburg
Coverfoto:	istockphoto 4366128 und istockphoto 6897767
Illustrationen:	fotolia, tetris game © Dmitry Sunagatov

Umwelthinweis:

Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt. Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

Trotz sorgfältigem Lektorat schleichen sich manchmal Fehler ein. Autoren und Verlag sind Ihnen dankbar für Anregungen und Hinweise!

Smart Books Publishing AG	Sonnenhof 3, CH-8808 Pfäffikon SZ,
http://www.smartbooks.ch	E-Mail: info@smartbooks.ch
Aus der Schweiz:	Tel. 055 420 11 29, Fax 055 420 11 31
Aus Deutschland und Österreich:	Tel. 0041 55 420 11 29, Fax 0041 55 420 11 31

Alle Rechte vorbehalten. Die Verwendung der Texte und Bilder, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und strafbar. Das gilt insbesondere für die Vervielfältigung, Übersetzung, die Verwendung in Kursunterlagen oder elektronischen Systemen. Der Verlag übernimmt keine Haftung für Folgen, die auf unvollständige oder fehlerhafte Angaben in diesem Buch oder auf die Verwendung der mitgelieferten Software zurückzuführen sind. Nahezu alle in diesem Buch behandelten Hard- und Software-Bezeichnungen sind zugleich eingetragene Warenzeichen oder sollten als solche behandelt werden.

Besuchen Sie uns im Internet!

www.smartbooks.ch

www.smartbooks.de

Übersicht

Kapitel 1	Über dieses Buch	13
Kapitel 2	Mathematik	53
Kapitel 3	Zeichnen in OpenGL	81
Kapitel 4	Virtuelle Kameras und Projektionen	117
Kapitel 5	Farben, Materialien und das Licht	133
Kapitel 6	Alpha-Blending	159
Kapitel 7	Texturierung	173
Kapitel 8	Rendervorgang beschleunigen	205
Kapitel 9	Textausgabe in OpenGL	235
Kapitel 10	Spezialeffekte	249
Kapitel 11	3D-Models	287
Kapitel 12	Shader	309
Kapitel 13	Sound-Entwicklung mit OpenAL	381
Kapitel 14	Kollisionserkennung	403
Kapitel 15	Das Spiel ScrambleX	433
Index		457

Inhaltsverzeichnis

Kapitel 1	Über dieses Buch	13
	Über dieses Buch	14
	Was möchte dieses Buch?	14
	Was brauchen Sie?	15
	Die benötigten APIs	15
	OpenAL	15
	OpenGL	16
	OpenGL am Mac	16
	Der Framebuffer	18
	OpenGL intern	19
	Bibliotheken	19
	Datentypen	20
	Funktionsnamen	21
	OpenGL-Erweiterungen	22
	Ihre erste Anwendung	23
	OpenGL-Anwendung – der einfache Weg	23
	OpenGL-Anwendung mit einem NSView	39
	Shared Context	44
	Fullscreen-Anwendung	45
	Zusätzliche Informationen	52
Kapitel 2	Mathematik	53
	Skalare, Punkte und Vektoren	55
	Vektor-Grundlagen	57
	Berechnung: Vektorlänge	57
	Berechnung: Einheitsvektor (normierter Vektor)	57
	Berechnung: Vektor-Rechenoperationen	58
	Berechnung: Multiplikation mit einem Skalar	59
	Berechnung: Punktprodukt (Dotproduct, Innerproduct, Skalarprodukt)	59
	Berechnung: Kreuzprodukt (Crossproduct)	60
	Matrizen-Grundlagen	62
	Matrizen in OpenGL	65
	Matrizen verwenden	66
	Reihenfolge der Transformationen	70
	Eigene Matrizen	72
	Matrize laden	72
	Matrize multiplizieren	73
	Matrix-Stapel	73
	Schlussbemerkung	80
	Zusätzliche Informationen	80
Kapitel 3	Zeichnen in OpenGL	81
	Zeichnen in OpenGL	82
	Punkte	83

Punktgröße	84
Linien	86
Linienstärke	87
Linienmuster	87
Line_Strip und Line_Loop	90
Dreiecke	92
Zeichenrichtung	93
Backface-Culling	94
Zeichenmodi	94
Triangle-Fan	98
Triangle-Strip	98
Vierecke	102
Polygone	103
Tiefenpuffer (Z-Buffer, Depth-Buffer)	104
Asteroids	107
Timebased versus Framebased	109
FPS	111
Bounding-Box	115
Zusätzliche Informationen	116

Kapitel 4 Virtuelle Kameras und Projektionen 117

Virtuelle Kameras und Projektionen	118
Modelview	119
Viewport	119
Projektion	119
Orthogonale Projektion (Parallelprojektion)	120
Perspektivische Projektion (Zentralprojektion)	122
Die virtuelle Kamera	125
Rotationen und der Gimbal Lock	132
Zusätzliche Informationen	132

Kapitel 5 Farben, Materialien und das Licht 133

Farben, Materialien und das Licht	134
Farben verwenden	135
Shading	136
Smooth-Shading	136
Flat-Shading	137
Licht	137
Die verschiedenen Lichtarten	138
Lichtmodel	139
Globales Licht	140
Betrachter-Position	140
Beidseitige Beleuchtung der Polygone	140
Separate Berechnung des Glanzanteils	140
Licht-Abschwächung	141
Materialien	141
Normale	142
Glanzeffekte	147
Spotlicht	150

	Lichtposition.....	150
	Color-Tracking	153
	Zusätzliche Informationen.....	158
Kapitel 6	Alpha-Blending	159
	Alpha-Blending	160
	Wie funktioniert das Blending?	160
	Blending einschalten	161
	Blendfunktionen.....	161
	Polygone sortieren	163
	Transparente Objekte	164
	Reflektionen.....	166
	Wie funktioniert's?	166
	Antialiasing	169
	Verhaltensregeln (Hints)	170
	Zusammenfassung	172
	Zusätzliche Informationen.....	172
Kapitel 7	Texturierung	173
	Texturierung.....	174
	Texturen laden	176
	Textur löschen	183
	Textur-Größe	183
	Textur-Umgebung	184
	Texturen »wickeln«	186
	Texturen filtern.....	187
	Mip-Maps	191
	Secondary Color	192
	Anisotropes Filtern.....	193
	Textur-Transformation	195
	Alpha-Masking.....	195
	Multi-Texturing.....	198
	Testen, ob das System Multi-Texturing unterstützt	198
	Textur-Einheit aktivieren.....	199
	Textur-Koordinaten festlegen	200
	Zusammenfassung	204
	Zusätzliche Informationen.....	204
Kapitel 8	Rendervorgang beschleunigen	205
	Rendervorgang beschleunigen	206
	Display-Lists.....	206
	Display-Lists erstellen	206
	Display-Lists mit Daten füttern.....	207
	Display-Lists ausführen	208
	Display-Lists löschen.....	209
	Vertex-Arrays.....	211
	Vertex-Arrays benutzen.....	212
	Der Stride-Parameter.....	217
	Indexierte Vertex-Arrays	217

Die Königsklasse Vertex-Buffer-Objects (VBOs)	220
VBOs rendern	222
VBOs löschen	224
Indexierte VBOs.....	224
VBOs ändern.....	229
Statische und dynamische Daten mischen.....	230
VBOs mit Offset	230
Zusammenfassung	233
Kapitel 9 Textausgabe in OpenGL	235
Textausgabe in OpenGL.....	236
Font laden.....	237
Bitmaps.....	243
Text ausgeben	244
Zusätzliche Informationen.....	248
Kapitel 10 Spezialeffekte	249
Spezialeffekte	250
Billboards.....	250
Billboards erstellen	251
Beispiel-Billboards.....	253
Partikel	261
Partikel-Systeme.....	263
Partikel Feuer-Effekt	265
Shockwave-Effekt	269
Das Pentagonram.....	274
Point-Sprites	276
Nebel.....	279
Nebel-Parameter	279
Volumetrischer Nebel.....	283
Zusätzliche Informationen.....	286
Kapitel 11 3D-Models	287
3D-Models.....	288
3D-Formate.....	288
Das Wavefront-Format.....	288
Das obj-Format intern.....	289
Wavefront-Model laden.....	291
Schritt 1	292
Schritt 2	294
Schritt 3	297
Zusätzliche Informationen.....	308
Kapitel 12 Shader	309
Shader.....	310
Was sind Shader.....	311
Warum Shader benutzen.....	313
Voraussetzungen	313
Handhabung der Shader.....	314

Shader-Objekte.....	315
Programm-Objekte	316
Zusammenfassung	318
Shader-Hilfsklassen	319
CFXShaderManager	319
CFXShaderObject	324
Ein erster Versuch	325
GLSL-Grundlagen.....	329
Vektoren	330
Matrizen	330
Typenqualifizierer	331
Built-In Variablen	333
Beispiel 2, der Farbverlauf.....	338
Bursting Mesh.....	343
Material und Beleuchtung	351
Ambientes Licht.....	351
Glanz-Anteil.....	353
Per-Pixel-Beleuchtung.....	355
Texturierung.....	358
Textur-Koordinaten	358
Texturen.....	359
Textur-Transformation	360
Multi-Texturing.....	365
Texturen kombinieren.....	367
Texture-Combiners	368
Mehr Multi-Texturing.....	369
Texturen mit Material und Licht kombinieren.....	371
Alpha-Masking.....	373
Alpha-Masking ohne Alpha-Kanal	375
Nebel	375
Nebelberechnung.....	376
Per-Pixel-Nebel.....	377
Entwicklungsumgebung	379
Zusätzliche Informationen.....	379
Auflösung Textur-Quiz	380

Kapitel 13 Sound-Entwicklung mit OpenAL	381

Sound-Entwicklung mit OpenAL	382
Soundausgabe am Mac.....	382
OpenAL	382
ALUT (OpenAL Utility Kit)	383
OpenAL einbinden.....	383
OpenAL initialisieren.....	383
OpenAL Fehlerbehandlung.....	384
OpenAL beenden	389
Mehrere Sounds.....	392
Dopplereffekt	392
OpenAL abfragen.....	393

OpenAL Extensions	394
CFXOpenAL.....	396
Alternativer Wav-Loader.....	398
Musik mit Quicktime abspielen.....	398
Zusätzliche Informationen.....	402

Kapitel 14 Kollisionserkennung 403

Kollisionserkennung.....	404
Bounding Box.....	404
AABB.....	405
OBB	406
Kollisionstest.....	406
Hitbox.....	408
Beispiel AABB-AABB-Kollision	408
Bounding-Sphere.....	411
Sphere-Sphere-Kollision	412
Sphere-AABB-Kollision	412
SAT	415
Frustum Culling.....	417
Frustum extrahieren	418
Punkt im Frustum.....	420
AABB im Frustum	421
Sphere in Frustum.....	422
FPS-Counter.....	424
Occlusion Queries	425
Erzeugen und Löschen von Queries	426
Queries nutzen	427
Zusätzliche Informationen.....	432

Kapitel 15 Das Spiel ScrambleX 433

Das Spiel ScrambleX.....	434
Bestandsaufnahme.....	435
Höhlenmesh aufteilen	437
Höhle rendern	439
Kollision mit der Höhle.....	440
Spielobjekte.....	443
Der Endgegner.....	444
Die Models.....	446
Das Level	449
Das HUD.....	450
Die Texturen.....	451
Die Partikelsysteme.....	452
Die Shader	453
Der Rest	455
Zusätzliche Informationen.....	456

Index 457

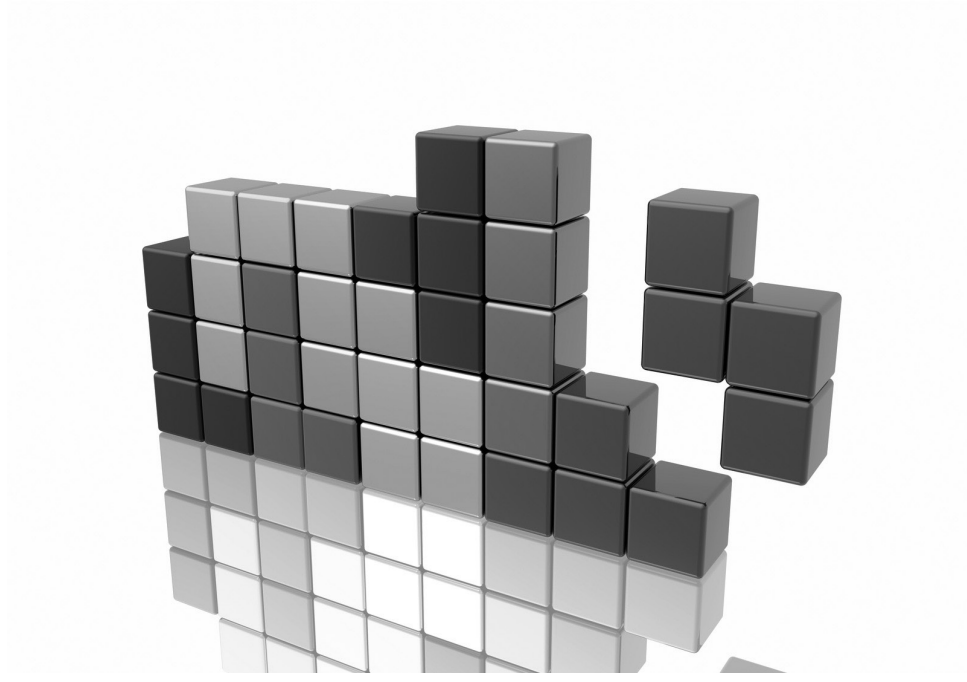
Danke

Ich möchte mich zu allererst bei meiner Frau Andrea bedanken, die mich überhaupt dazu ermutigt hat, dieses Buch zu schreiben und dafür das sie mir in der ganzen Zeit den Rücken frei gehalten hat. Bei meinem Sohn Tim, der mehr als einmal hören musste »ich hab jetzt keine Zeit, später vielleicht«. Weiterhin bedanke ich mich bei Christian Klönz (<http://klonsemann.de/>) für seine 3D-Models und die guten Ideen zum Leveldesign des Spiels, die Zusammenarbeit hat wirklich sehr viel Spaß gemacht. Danke auch an Amin für die »Connection« zum Verlag.

Grüße gehen an Kay Löhmann vom OS-X-Entwicklerforum (www.osxentwicklerforum.de) und natürlich an alle, die dort für jede Menge Diskussionsstoff sorgen, Chris Hauser, Thomas Bierdorf und alle anderen Mitstreiter von der Macoun in Frankfurt (wir sehen uns dann). Grüße auch an Frank Scholl und Manfred Kress (die »CocoaHeads« Mannheim, sucht ihresgleichen).

Last but not least grüße ich alle, die mich die letzten Wochen kontaktiert haben und fragten »Wann kommt das Buch denn nun endlich raus«. Nun ist es soweit. Das Warten hat ein Ende!

Kronau im Juni 2009



Über dieses Buch

Über dieses Buch

Mit diesem Buch möchte ich Ihnen zeigen, wie Sie mit Hilfe von Cocoa, OpenGL und OpenAL Spiele für den Mac entwickeln können.

Es gibt dutzende von Büchern, die sich mit dem Thema Spielprogrammierung beschäftigen, diese sind aber zum Großteil für den PC und DirectX geschrieben. Der Grund dafür dürfte wohl jedem bekannt sein, wobei ich mir sicher bin, dass sich dieses »Phänomen« in den kommenden Jahren ändern wird. Seitdem Apple den »Switch« auf Intel gemacht hat, gibt es immer mehr Benutzer, die mit einem Mac liebäugeln, wodurch es meiner Meinung nach nur noch eine Frage der Zeit ist, bis sich der Mac als Spieleplattform durchgesetzt hat.

Was auf den ersten Blick vielleicht etwas »unüblich« wirkt, ist die Tatsache, dass ich auf Cocoa und nicht, wie die meisten anderen Entwickler, auf C/C++ in Verbindung mit Carbon gesetzt habe.

Geht es in der Spielentwicklung nicht darum, die maximale Geschwindigkeit aus einem System zu holen?

Nun ja, sieht man die Performance, die aktuelle Prozessoren (respektive der Grafikkarten) bieten, ist die Verwendung der Sprache meiner Meinung nach eher zweitrangig. Oft entsteht nämlich eine schlechte Performance nicht durch die benutzte Programmiersprache, sondern dadurch, wie man sie nutzt. Ich denke, solange es Java-Portierungen von Quake gibt, müssen wir uns um die Geschwindigkeit von Cocoa keine Sorgen machen.

Weiterhin spricht für Cocoa, dass es für fast alle anstehenden »Probleme« schon vorgefertigte Klassen gibt, die man auch für die Spielentwicklung sehr gut verwenden kann.

Was möchte dieses Buch?

Wie schon erwähnt, ist das Ziel des Buches, Ihnen die Welt der Spielentwicklung am Mac zu zeigen.

Gerade das Thema OpenGL bereitet vielleicht dem einen oder anderen Leser »Bauchschmerzen«, galt es doch jahrelang als »Heiliger Gral« der 3D-Programmierung, über den man so gut wie keine Informationen fand.

TIPP

Jeff Molofee mit seiner Seite <http://nehe.gamedev.net/> war wohl einer der ersten, der so manch einem Entwickler (einschließlich mir) das Tor zu OpenGL geöffnet hat.

Ich verspreche Ihnen, Sie werden erstaunt sein, wie einfach die Grundlagen von OpenGL sind.

Der Weg vom einfachen Dreieck bis zu einem kompletten Spiel ist zwar ein wenig steinig, wenn man aber die Grundzüge verstanden hat (und ich bin überzeugt, dass Sie sie verstehen werden, wenn Sie das Buch durchgearbeitet haben), dann ist der Rest »nur noch« Fleißarbeit.

Was brauchen Sie?

Um das Buch erfolgreich bis zum Ende durcharbeiten zu können, ist es wichtig, dass Sie über folgende Vorkenntnisse bzw. Fähigkeiten verfügen:

- Solide Kenntnisse in der Programmierung mit Objective-C / Cocoa sind unabdingbar. Hilfreich wären zudem Kenntnisse in der C-Programmierung, da davon auch Gebrauch gemacht wird.
- Grundkenntnisse in der Mathematik
- Räumliches (3D) Vorstellungsvermögen wäre hilfreich.
- Durchhaltevermögen
- Und ganz wichtig: eine gehörige Portion Neugier

Die benötigten APIs

Bevor wir uns an die Arbeit machen, verschaffen wir uns noch einen kurzen Überblick über die beiden APIs OpenAL und OpenGL, um zu wissen, wofür beide zuständig sind. Wie Sie feststellen werden, liegt das Hauptaugenmerk auf OpenGL, da diese Bibliothek um ein Vielfaches komplexer ist. Aber keine Sorge, OpenAL wird nicht zu kurz kommen.

OpenAL

OpenAL (Open Audio Library) ist eine plattformunabhängige 3D-Audio-API, welche hauptsächlich für die Spieleentwicklung erarbeitet wurde. Die API kann man als Ergänzung zu OpenGL betrachten, weshalb auch die Handhabung sehr ähnlich ist. Da OpenAL auf dem Mac schon als Framework enthalten ist, müssen keine weiteren Bibliotheken nachinstalliert werden, was uns natürlich sehr entgegenkommt.

Wenn man bedenkt, dass es OpenAL erst seit dem Jahr 2000 gibt (die Version, welche 1998 von Loki Software entwickelt wurde, war wohl nicht das Wahre), hat sich die API doch recht schnell verbreitet. Inzwischen nutzen sogar kommerzielle Entwickler OpenAL. Die wohl bekannteste Engine, die auf OpenAL setzt, dürfte die Unreal-Engine von Epic sein, aber auch so bekannte Titel wie Doom3, Quake4 oder S.T.A.L.K.E.R. nutzen OpenAL zur Soundausgabe.

OpenGL

OpenGL ist eine plattformunabhängige Bibliothek zur Entwicklung von 3D-Grafikprogrammen, die ursprünglich von SGI im Jahre 1992 veröffentlicht wurde. Seit diesem Zeitpunkt wurde sie ständig durch das ARB weiterentwickelt und liegt im Moment in der Version 2.1 vor. Diese Version beinhaltet zurzeit mehr als 200 Funktionen für die 3D-Programmierung.

TIPP

Das ARB (Architecture Review Board) ist ein Zusammenschluss mehrerer namhafter Firmen wie z. B. SGI, NVidia, Apple, 3DLabs, Intel, usw., die den Standard von OpenGL festlegen. Microsoft war einst Mitbegründer des ARBs und hat es 2003 verlassen, was wohl daran lag, dass sie ihre eigene 3D-Schnittstelle vorantreiben wollen.

Im Gegensatz zu Direct3D (Microsofts Gegenstück zu OpenGL) ist OpenGL eine rein prozedurale API und arbeitet in Form eines Zustandsautomaten (State-Machine). Dies bedeutet, dass man einen Zustand (z. B. die Beleuchtung) explizit ein- bzw. ausschalten muss. Diese Zustände beeinflussen dann die nachfolgende Ausgabe auf dem Bildschirm. Diese State-Machine kann gerade am Anfang ein großer Frustfaktor sein, weil man mitunter nicht immer gleich dahinter kommt, weshalb die Ausgabe nicht so aussieht, wie man es erwartet hat. Auch hier gilt: Mit ein wenig Übung klappt das schon.

OpenGL am Mac

Was ist so besonders an OpenGL am Mac?

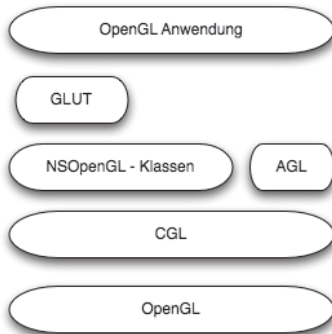
Während ein Windows-PC theoretisch auch ohne OpenGL auskommen kann, ist das bei Mac OS X nicht der Fall, da OpenGL hier ein elementarer Bestandteil des Betriebssystems ist. Eine weitere Besonderheit von OpenGL am Mac ist die Möglichkeit, wie man OpenGL-basierte Anwendungen schreiben kann.

Apple bietet nicht weniger als 3 Schnittstellen für die 3D-Programmierung, dies dürfte wohl einzigartig sein.

AUFGEPASST

Apple spricht in der Dokumentation zwar von 3 Möglichkeiten, auf OpenGL zuzugreifen, verweist aber gleichzeitig auf GLUT und X11, womit es dann sogar 5 »Wege nach Rom« wären.

Schauen wir uns zunächst mal an, wie diese aufgebaut sind.



Das Schichtenmodell für OpenGL-Anwendungen

CGL

Auf der untersten Ebene, über der Treiberschicht, befindet sich CGL (Core OpenGL) und stellt somit die Basis für AGL und die NSOpenGL-Klassen dar.

CGL bietet im Allgemeinen die höchste Flexibilität, wie man eine OpenGL-Anwendung erstellt, leider aber auch die »komplizierteste«.

GRUNDLAGEN

Die größte Einschränkung von CGL ist aber, dass man damit keine fensterbasierte Anwendungen schreiben kann. Eine CGL-Anwendung kann lediglich eine Vollbild- bzw. Offscreen-Ausgabe erstellen.

AGL

AGL (Apple GL) ist das Carbon-Interface am Mac. Wir werden uns aber nicht weiter mit AGL beschäftigen, da der Fokus in diesem Buch ja auf Cocoa liegt.

Cocoa / NSOpenGL-Klassen

Cocoa stellt uns mehrere Klassen bereit, die für das Arbeiten mit OpenGL wichtig sind.

Nachfolgend die 3 wichtigsten:

- **NSOpenGLView:** Bietet einen einfachen Zugriff auf OpenGL und kann entsprechend im Interface Builder erstellt werden. Das NSOpenGLView ist eine Unterklasse von NSView und beinhaltet gleichzeitig ein NSOpenGLContext und ein NSOpenGLPixelFormat-Objekt. Anders als ein »gewöhnlicher« NSView, kann ein NSOpenGLView keine Subviews enthalten.
- **NSOpenGLContext:** Die Verknüpfung zwischen den OpenGL-Befehlen, die verarbeitet werden sollen, und der grafischen Ausgabe auf dem Bildschirm.
- **NSOpenGLPixelFormat:** Beschreibt das Format unseres Framebuffers, den wir erstellen wollen.

Der Framebuffer

Der Framebuffer ist eine Zusammenfassung aus vier verschiedenen Puffern, die wie ein zweidimensionales Array aufgebaut sind und Daten pro Pixel speichern. Jeder dieser Puffer hat dabei eine ganz bestimmte Aufgabe.

Der Color-Buffer

Darin wird die Farbe für jedes Pixel gespeichert. Wenn man ein Pixelformat mit Double-Buffering anlegt, gibt es zwei dieser Color-Buffer.

Der Depth-Buffer (Z-Buffer)

Speichert die Tiefeninformation der Pixel im 3D-Raum. Dies ist wichtig, damit OpenGL später »weiß«, welches Pixel vor bzw. hinter einem anderen Pixel liegt und somit gerendert bzw. nicht gerendert wird.

Der Stencil-Buffer

Ist ein Maskierungspuffer, mit dem man bestimmte Bereiche aus dem Renderprozess ausschließen kann.

Der Akkumulation-Buffer

Wird zum Beispiel für das Szenen-Antialiasing benutzt, oft aber auch für Motion-Blur-Effekte.

Keine Sorge, wenn Sie im Moment noch nicht allzu viel mit den »Fachbegriffen« anfangen können.

Am Ende des Buches wissen Sie Bescheid.

Wie gesagt, bietet der *NSOpenGLView* die einfachste Möglichkeit, eine OpenGL-Anwendung zu erstellen. Wenn man aber flexibler sein will, besteht natürlich die Möglichkeit, ein *NSView* abzuleiten und mit den beiden anderen oben genannten Klassen eine Verbindung zu OpenGL herzustellen.

Wir werden uns im weiteren Verlauf die verschiedenen Möglichkeiten noch genauer anschauen. Der Vollständigkeit halber sei hier nochmals auf GLUT und X11 hingewiesen, mit denen man auch OpenGL-Anwendungen erstellen kann. Diese beiden werden wir uns aber nicht weiter anschauen.

Aus Sicht der Spieleprogrammierung sollte man vielleicht noch SDL erwähnen. Dies ist auch eine plattformunabhängige Bibliothek, die speziell dafür entwickelt wurde, um Spiele zu erstellen. Entsprechend einfach ist die Handhabung von SDL, da diese sich um Dinge wie die grafische Ausgabe (OpenGL), Benutzer-Eingaben (auch mit Gamepads und Joysticks) usw. kümmert.

TIPP

SDL (Simple Direct Layer <http://www.libsdl.org/>) wurde von Sam Lantinga zwischen 1999 und 2001 entwickelt. Zu dieser Zeit arbeitete Lantinga bei Loki Software, die Spiele nach Linux portieren. SDL diente als Grundlage für z. B. Civilization: Call to Power und Descent.

Schauen Sie sich die Bibliothek ruhig einmal an, Sie werden erstaunt sein, welch eine große Fangemeinde SDL hat.

OpenGL intern

Nun, da wir nun wissen, welche Möglichkeiten es gibt, am Mac eine OpenGL-Anwendung zu schreiben, schauen wir uns OpenGL mal ein wenig genauer an.

Bibliotheken

Die OpenGL-Funktionen werden über Bibliotheken angesprochen, die in der Regel mit Xcode schon fertig installiert werden, wodurch man sofort damit beginnen kann, OpenGL-Anwendungen zu schreiben.

GL

Dies ist die Standardbibliothek (Graphic Library), welche einen Großteil der wichtigsten Funktionen beinhaltet, die Sie bei der 3D-Programmierung brauchen werden. Funktionen aus dieser Bibliothek beginnen mit dem Präfix `gl_`.

GLU

Diese Bibliothek (OpenGL Utility Library) ist eine Erweiterung zur GL-Bibliothek. Funktionen dieser Bibliothek sind häufig eine Zusammenfassung von verschiedenen GL-Funktionen und erleichtern dadurch dem Entwickler die Arbeit

mit OpenGL. Ein klassisches Beispiel ist die Funktion `gluLookAt( . . . )`, welche eine virtuelle Kamera erstellt. Diese Funktionen beginnen mit dem Präfix `glu_`.

GLUT

Das OpenGL Utility Toolkit erleichtert das Erstellen von Tastatur- und Mauseingaben. Weiterhin kümmert sich diese Bibliothek um das Erstellen von Fenstern und Menüs. Ich möchte aber hier gleich von der Verwendung von GLUT abraten, da sie seit einiger Zeit nicht mehr weiterentwickelt wird. Für das Entwickeln von plattformunabhängigen Anwendungen ist FreeGLUT eine gute Alternative zu GLUT. Anders als GL und GLU gehört GLUT nicht zu den Standard-Bibliotheken von OpenGL.

Datentypen

OpenGL arbeitet intern mit einem Satz eigener Datentypen. Dies hat auch einen guten Grund: Nehmen wir z. B. eine *int*-Variable, diese ist je nach Betriebssystem und Compiler einmal 16 Bit und einmal 32 Bit groß.

Damit wir uns keine Gedanken darüber machen müssen, welche Größe ein Datentyp hat, bietet uns OpenGL seine eigenen Datentypen an. Gerade wenn es darum geht, Anwendungen auf andere Systeme zu portieren, sollte man diese unbedingt verwenden.

Nachfolgend eine Auflistung der OpenGL-Datentypen:

OpenGL Typ	Intern	C-Typ
GLbyte	8-bit Integer	signed char
GLshort	16-bit Integer	short
GLint, GLsizei	32-bit Integer	long
GLfloat, GLclampf	32-bit Float	float
GLdouble, GLclampd	64-bit Float	double
GLubyte, GLboolean	8-bit unsigned integer	unsigned char
GLushort	16-bit unsigned integer	unsigned short
GLuint, GLenum, GLbitfield	32-bit unsigned integer	unsigned long
GLchar	8-bit character	char
GLsizeiptr, GLintptr	Native pointer	ptrdiff_t

Benutzt werden die Datentypen genauso, wie Sie es z. B. von C gewohnt sind.

Hier ein Beispiel eines Arrays von 10 *GLfloat*-Variablen:

```
GLfloat vertices[10];
```

Dies mag am Anfang ein wenig befremdend wirken, aber man gewöhnt sich sehr schnell daran.

Funktionsnamen

Wie Sie weiter oben ja schon erfahren haben, folgen die meisten OpenGL-Funktionen einer Namenskonvention, welche dem Entwickler sofort zeigt, aus welcher Bibliothek sie stammen.

Darüber hinaus sieht man auch meistens sofort, wie viele Argumente bzw. welchen Datentyp diese Funktionen benötigen.

Sehr oft setzen sich die Funktionen wie folgt zusammen:

<Bibliothek Präfix><Befehl><Anzahl der Argumente><Argument Typ>

Hier ein Beispiel für einen Befehl, der eine weiße Farbe definiert:

```
glColor3f(1.0, 1.0, 1.0);
```

GRUNDLAGEN

OpenGL erwartet die Farbangaben mittels 3 Float-Werten für RGB bzw. 4 Float-Werten für RGBA, diese reichen von 0.0–1.0.

- **gl** leitet das Präfix für die Bibliothek ein (hier die GL-Bibliothek)
- **Color** wäre der Befehl an sich
- **3** beschreibt die Anzahl der Argumente (1.0, 1.0, 1.0)
- **f** den Datentyp (hier *GLfloat*).

Tatsächlich ist es so, dass es sehr oft identische Befehle mit unterschiedlichen Argumenten und Datentypen gibt:

`glColor3i` (erwartet drei Argumente vom Typ *GLint*)

`glColor3fv` (erwartet einen Array von 3 *GLfloat*-Werten)

`glColor4f` (erwartet 4 *GLfloat*-Werte, wobei der letzte Wert den Alphaanteil der Farbe beschreibt)

OpenGL-Erweiterungen

Wie schon weiter oben erwähnt, ist das ARB für die Erweiterungen von OpenGL verantwortlich. Zur Freude für uns Entwickler gibt es eine ganze Reihe von Erweiterungen, die dadurch OpenGL immer »mächtiger« machen.

Erweiterungen, die mit dem Postfix ARB beginnen, sind von allen Mitgliedern des ARB bestätigt worden und sind somit am weitesten verbreitet.

Eine dieser Erweiterungen sieht z. B. so aus:

```
glGenBuffersARB(1, &_vboIdIndicies);
```

Daneben gibt es auch solche, die mit EXT beginnen und die immerhin von einigen Mitgliedern des ARBs befürwortet wurden, sowie herstellerspezifische (NV, APPLE, SGI), die manchmal auch von anderen Herstellern unterstützt werden.

In dem Moment, in dem eine Erweiterung zum Kern von OpenGL wird, entfällt das Postfix.

TIPP

Die Erweiterungen (Extensions) sind in der Datei »`glext.h`« deklariert.

Ihre erste Anwendung

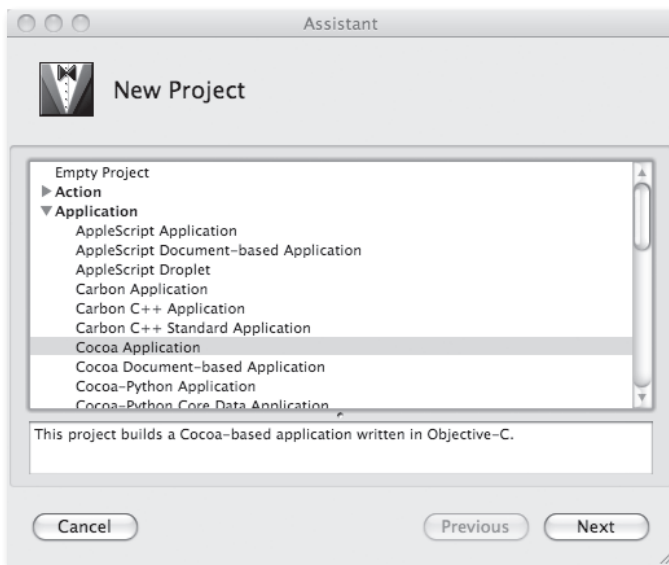
Nach diesen ganzen theoretischen Ausführungen wird es Zeit, mal etwas Praktisches auf den Bildschirm zu bringen. Bekanntermaßen lernt man am besten durch die praktische Anwendung.

Unsere erste OpenGL-Anwendung wird noch ein wenig »blass« werden, aber Sie lernen dabei schon einige sehr wichtige Dinge. Sie werden erstaunt sein, mit wie wenig Code wir auskommen werden.

OpenGL-Anwendung – der einfache Weg

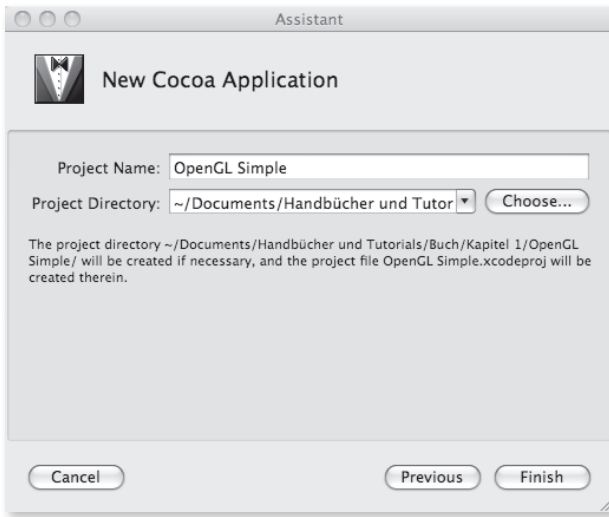
Ich gehe zwar davon aus, dass Sie mit Xcode vertraut sind, möchte aber trotzdem das Erstellen einer Anwendung hier nochmals Schritt für Schritt zeigen. Sie finden das Beispielprojekt im Ordner »Kapitel 1/OpenGL Simple«.

1. Starten Sie bitte Xcode und wählen Sie im ersten Schritt als Projekttyp »Cocoa Application«.



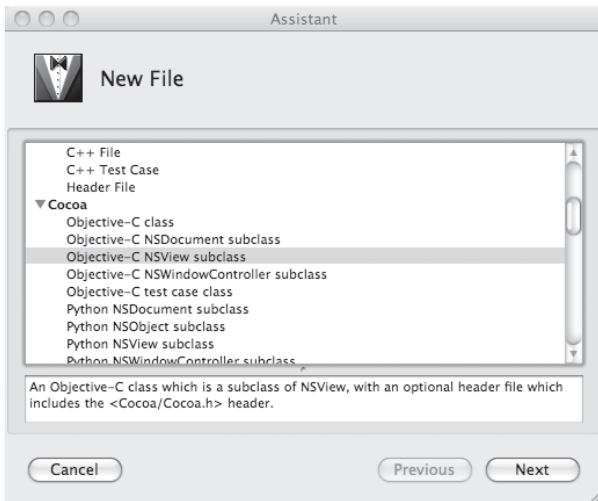
Unsere erste Anwendung

2. Wählen Sie bitte einen Namen (»OpenGL Simple«) und einen Speicherort (~ / Documents /) für Ihre Anwendung.



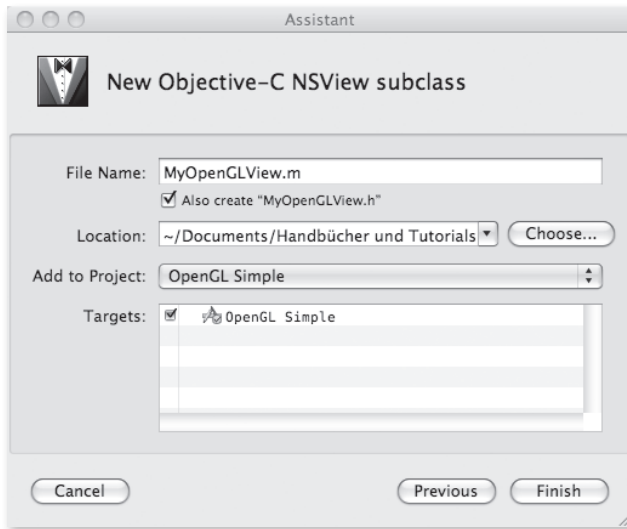
Der Name für unser Projekt

3. Zum Schluss klicken Sie auf *Finish*, um die Anwendung zu erstellen.
4. Wir werden jetzt für den Anfang die OpenGL-Ausgabe mit Hilfe eines *NSOpenGLView* wählen, da diese wie gesagt am einfachsten ist. Dazu erstellen wir uns zunächst eine Klasse, die von *NSOpenGLView* abgeleitet wird.
5. Dazu wählen Sie bitte unter *File* den Menüpunkt *New File*, wählen anschließend *Cocoa | Objective-C NSView subclass* und klicken dann wieder auf *Next*.



Wir erstellen eine Subclass von NSView.

6. Geben Sie dann noch einen Namen für Ihre Klasse ein (z. B. *MyOpenGLView*) und beenden den Schritt mit einem Klick auf *Finish*. Auf der linken Seite im Xcode-Fenster sehen Sie nun, dass die beiden Dateien Ihrem Projekt hinzugefügt wurden.



Noch ein Name für unseren View, und dann war's das schon.

7. Als Nächstes öffnen Sie bitte die Datei »MyOpenGLView.h« und ändern die Zeile

```
@interface MyOpenGLView : NSView {
```

in

```
@interface MyOpenGLView : NSOpenGLView {
```

8. des Weiteren fügen Sie bitte oben unter der Zeile

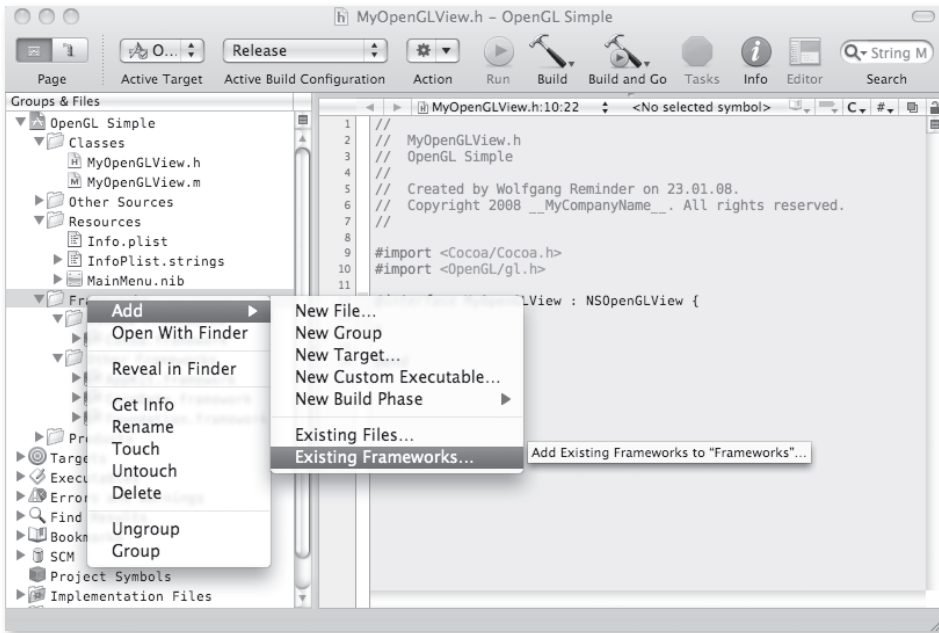
```
#import <Cocoa/Cocoa.h>
```

die Zeile

```
#import <OpenGL/glu.h>
```

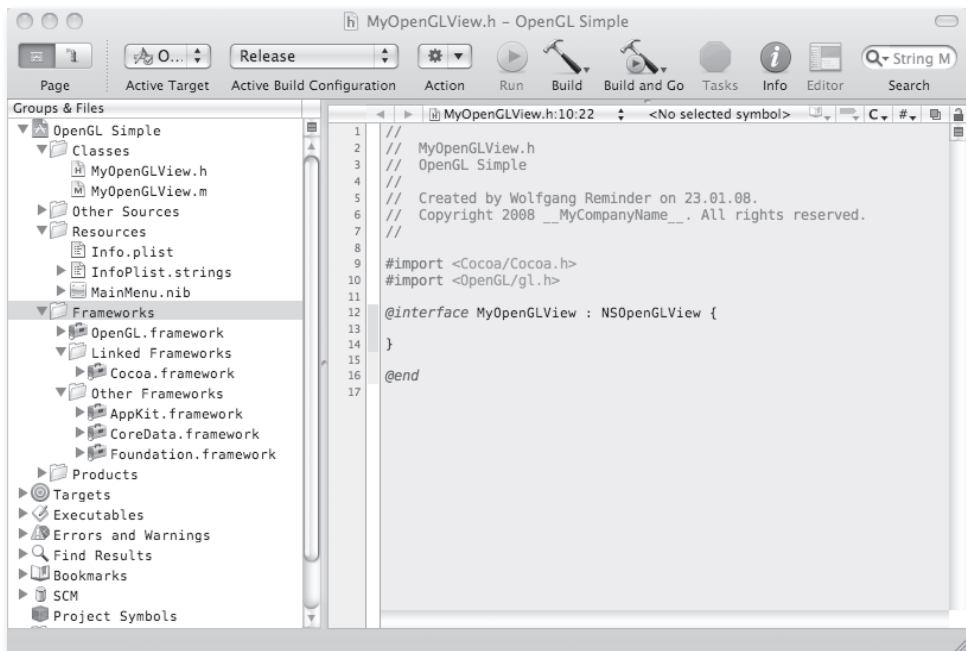
ein.

9. Zum Schluss müssen wir noch das Framework für OpenGL unserem Projekt hinzufügen. Dazu klicken Sie bitte mit der rechten Maustaste im Xcode-Fenster auf den Ordner »Frameworks« und wählen dann den Eintrag *Add | Existing Frameworks...*



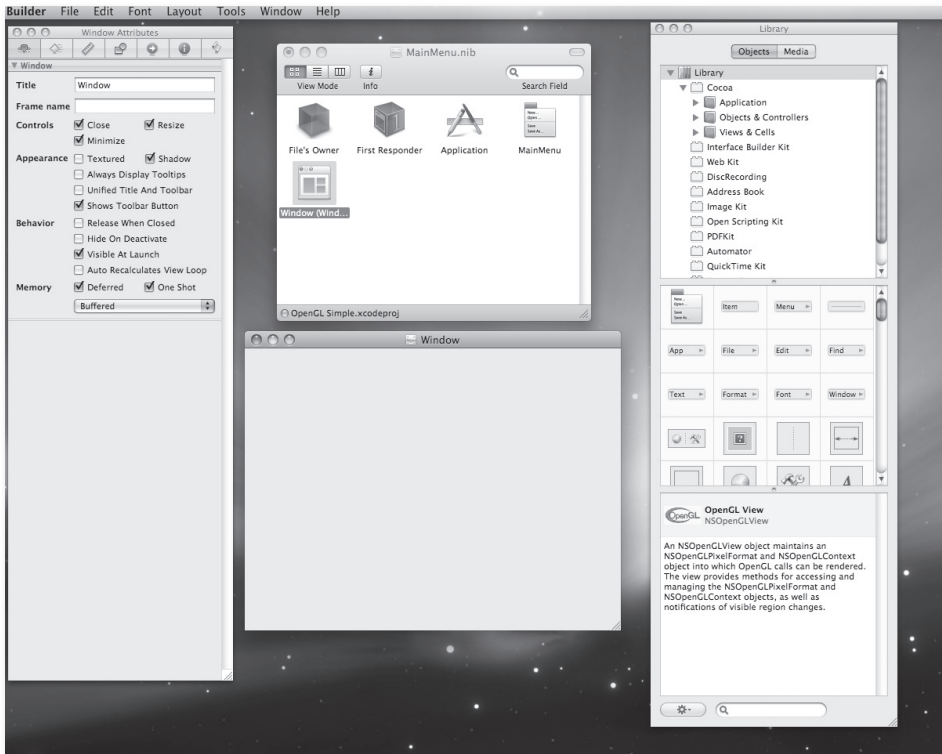
Wir müssen das OpenGL-Framework dem Projekt hinzufügen.

10. Im folgenden Dialog wählen Sie bitte die Datei »OpenGL.framework« aus (diese befindet sich im Ordner /Systems/Library/Frameworks/) und bestätigen die Auswahl mit *Add*.
11. Danach kommt noch ein Dialog, den Sie bitte auch mit *Add* quittieren.
12. Wenn alles geklappt hat, sollte nun Ihr Fenster in etwa so aussehen:



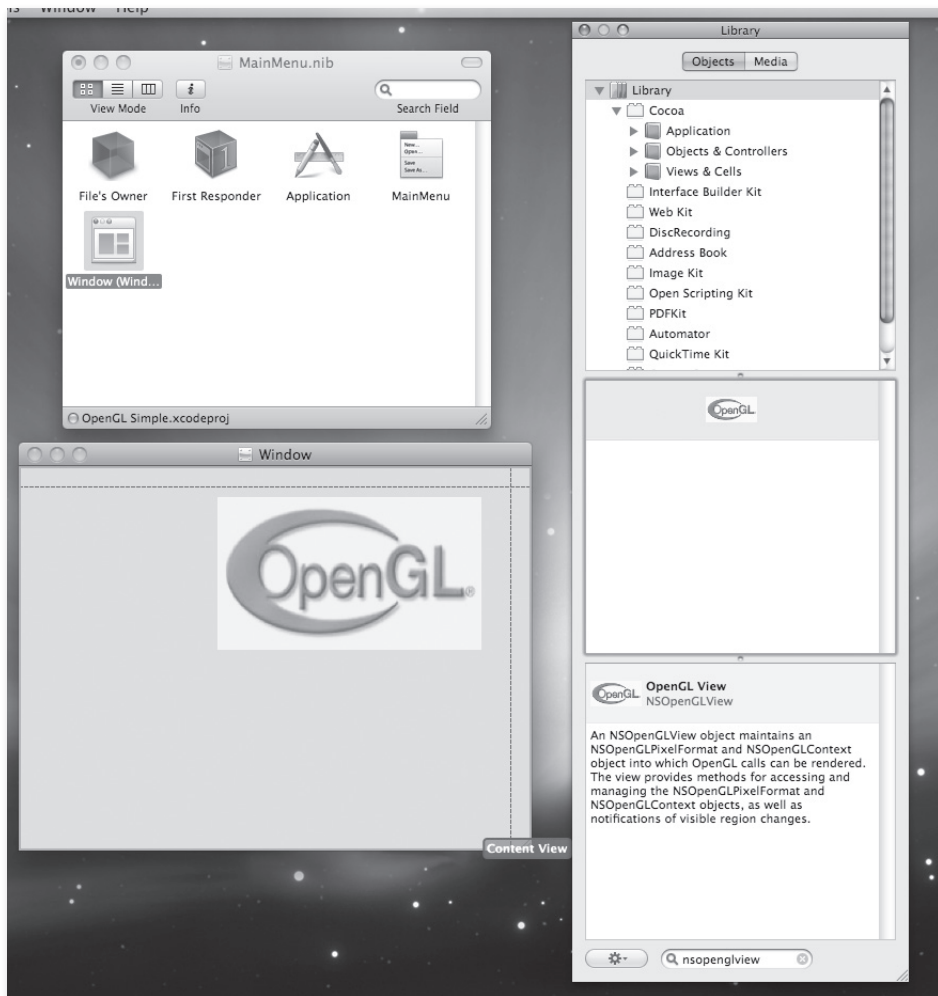
Unser fertiger Arbeitsbereich

13. Doppelklicken Sie nun bitte auf der linken Seite im Xcode-Fenster im Ordner Resources auf den Eintrag »MainMenu.nib«. Dies öffnet dann den Interface Builder, wo wir nun das Layout für unsere erste Anwendung erstellen wollen. Sollten Sie das Hauptfenster (*Window*) für unsere Anwendung nicht sehen, doppelklicken Sie im »MainMenu.nib«-Fenster auf das Symbol mit der Beschriftung *Window*, womit sich dann das Hauptfenster öffnen sollte.



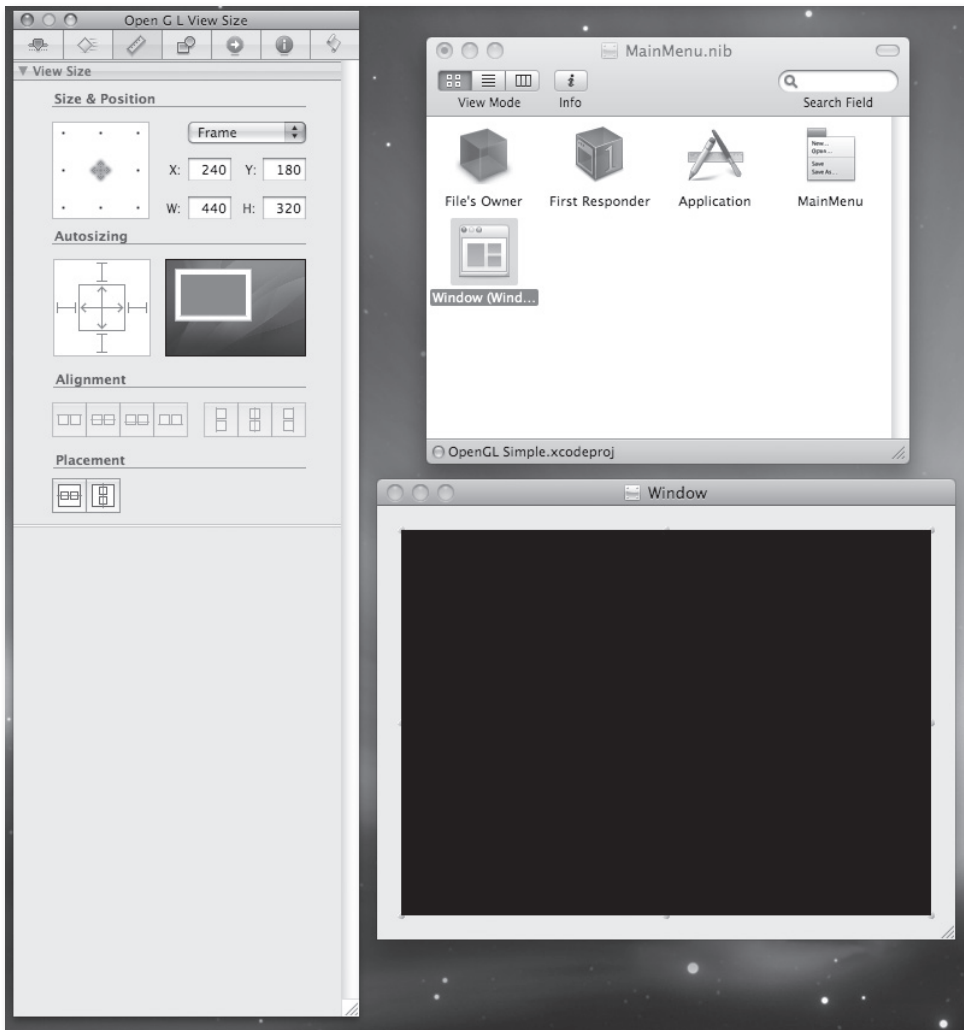
Lego für Entwickler: Im Interface Builder erstellen wir unsere Oberfläche.

14. Auf der rechten Seite im Library-Fenster tippen Sie bitte unten in das Feld mit der Beschriftung »Filter« das Wort *NSOpenGLView* ein. Daraufhin erscheint nun in der Mitte des Library-Fensters das *NSOpenGLView*, das Sie nun bitte auf Ihr Hauptfenster (*Window*) ziehen.



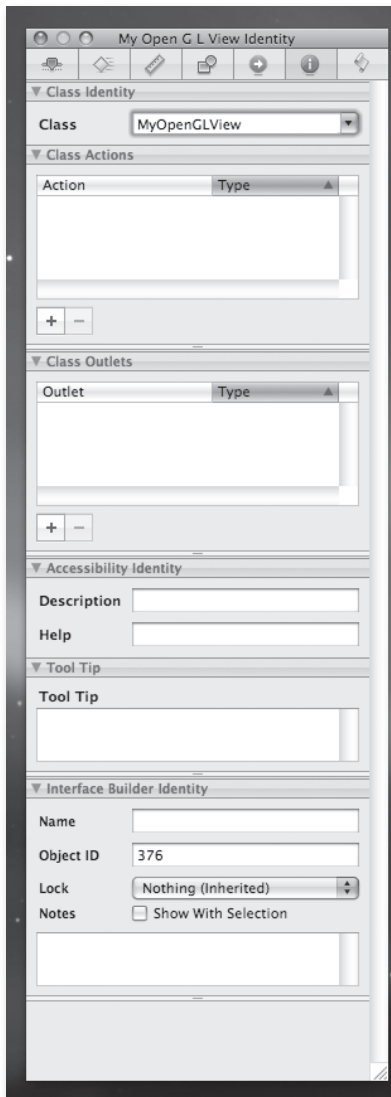
Noch schnell ein OpenGL-View in das Fenster gezogen.

15. Nun platzieren Sie bitte das *NSOpenGLView* entsprechend im Fenster und übernehmen dann bitte die Einstellungen für das Autosizing (zu finden im Inspector bei dem Reiter mit dem gelben Lineal), so wie es in der folgenden Abbildung zu sehen ist. Dadurch erreichen wir, dass unser View immer entsprechend zur Fenstergröße auch seine eigene Größe verändert.



Mit dem Autoresizing passen wir die Größe des Views an.

16. Zum Schluss müssen wir noch unserem `NSOpenGLView` mitteilen, dass es eine Subklasse von unserer erstellten Klasse ist. Dazu geben Sie bitte im Inspector in dem Reiter mit dem blauen Ausrufezeichen (Identity Pane) unter *Class* den Namen ein, den Sie für Ihre Klasse vorhin gewählt haben. Bei mir wäre das `MyOpenGLView`.



*Jedes Kind braucht einen Namen. Unser View heißt **MyOpenGLView**.*

17. Speichern Sie nun Ihre Arbeit im Interface Builder und schließen Sie ihn.
18. Als Nächstes wollen wir uns anschauen, was wir noch an Code eingeben müssen. Wechseln Sie nun bitte wieder zu Xcode, öffnen Sie die Datei »MyOpenGLView.m« und geben Sie dort bitte folgenden Code ein.

AUFGEPASST

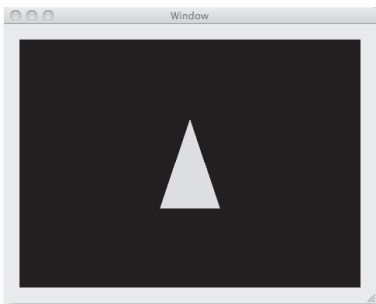
Sollte die *initWithFrame*-Methode noch im Code stehen, löschen Sie diese bitte.

```

- (void) prepareOpenGL
{
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
}
- (void)reshape
{
    NSRect rect = [self bounds];
    glViewport( 0, 0, (GLsizei)rect.size.width, (GLsizei)rect.size.height);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, rect.size.width / rect.size.height, 1.0, 5.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
}
-(void) drawRect: (NSRect) bounds
{
    glClear(GL_COLOR_BUFFER_BIT );
    glColor3f(1.0f, 0.85f, 0.35f);
    glBegin(GL_TRIANGLES);
    {
        glVertex3f( -0.2, -0.3, -2.0);
        glVertex3f( 0.2, -0.3, -2.0);
        glVertex3f( 0.0, 0.3, -2.0);
    }
    glEnd();
    glFlush();
}

```

19. Nachdem Sie nun alles eingegeben haben, übersetzen (Menü *Build* | *Build and Go*) und starten Sie bitte das Programm.
20. Wenn alles ohne Fehler funktioniert hat, sollten Sie nun eine Ausgabe haben, die in etwa so aussieht:



Das kann sich sehen lassen. »Hello World« in Gelb.

Glückwunsch, Sie haben soeben den ersten Schritt in die Spieleprogrammierung gemeistert. Keine Angst, wenn Sie den Code, den Sie eingegeben haben, noch nicht bzw. nur teilweise verstehen, ich werde Ihnen alles erklären.

Zuerst fällt auf, dass wir mit einer Handvoll Code auskommen, um ein einfaches Dreieck auf den Bildschirm zu rendern, und das alles in nur 3 Methoden (von denen 2 sogar optional sind).

TIPP

prepareOpenGL könnte hier sogar komplett entfallen. Wenn wir den Code aus der *reshape*-Methode in die *drawRect* schreiben würden, könnte man sogar auf diese Methode verzichten.

Wenn Sie schon auf anderen Systemen OpenGL-Anwendungen entwickelt haben, werden Sie sich wahrscheinlich wundern, warum nur so wenig Code nötig ist. Das Geheimnis dahinter verbirgt sich, wie schon weiter oben erwähnt, in der Benutzung der *NSOpenGLView*-Klasse, die uns den Großteil der Arbeit abnimmt. Bevor wir uns andere Möglichkeiten anschauen, mit denen man noch OpenGL-Anwendungen erstellen kann, schauen wir uns den Code einmal genauer an.

Zuerst kommt die Methode *prepareOpenGL*.

Man sollte diese Methode benutzen, um OpenGL-spezifische Einstellungen zu vorzunehmen.

GRUNDLAGEN

prepareOpenGL wird genau ein Mal beim Start aufgerufen.

Das hat folgenden Grund: In dem Moment, in dem diese Methode aufgerufen wird, ist sichergestellt, dass ein OpenGL-Kontext erstellt und aktiviert wurde. Erst dann können wir beginnen, OpenGL-Code zu schreiben.

Probieren Sie einmal Folgendes: Fügen Sie Ihrem Code in der Datei »MyOpenGLView.m« folgende Zeile hinzu:

```
- (void)awakeFromNib
{
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
}
- (void) prepareOpenGL
```

```
{  
    //glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );  
}
```

und kommentieren Sie bitte die Anweisung innerhalb der *prepareOpenGL* aus.

Übersetzen und starten Sie nochmals das Programm.

Es passiert anscheinend nichts bzw. das Programm startet nicht mehr. Was ist passiert? *awakeFromNib* wird doch aufgerufen, oder nicht? Ja, das stimmt, aber in dem Moment, in dem *awakeFromNib* ausgeführt wird, gibt es anscheinend noch keinen OpenGL-Kontext, und somit kann die Anweisung, die ja offensichtlich eine OpenGL-Funktion ist, nicht ausgeführt werden.

AUFGEPASST

Alles, was mit der OpenGL-Initialisierung zu tun hat, kommt in die *prepareOpenGL* Methode.

Also löschen Sie bitte wieder die *awakeFromNib* aus Ihrem Code und nehmen die Kommentare vor `glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );` in der *prepareOpenGL* wieder heraus.

Als Nächstes kommt die *reshape*-Anweisung, in der jede Menge »Seltsames« steht.

Der Inhalt der Methode ist im Moment noch nicht weiter wichtig. Wichtig ist allein die Tatsache, dass diese Methode immer dann aufgerufen wird, wenn sich die Fenstergröße ändert.

Zum Schluss kommt die Methode an die Reihe, die uns unser Ergebnis (eben das Dreieck) auf den Bildschirm »zaubert«. Wie Sie wahrscheinlich wissen, ist die *drawRect*-Methode nicht OpenGL-spezifisch, sondern eine Methode des *NSView* (was ja die Superklasse ist). Diese wird nun aufgerufen, wenn es etwas zu zeichnen gibt.

Nun lüften wir noch das Geheimnis, was denn da nun in der *drawRect*-Methode passiert.

Zuerst sagen wir, dass wir den Bildschirm (unser View) löschen wollen, danach definieren wir mit *glColor3f* eine Zeichenfarbe, die Werte darin stehen für die RGB-Komponenten einer Farbe und reichen von 0.0 bis 1.0.

Eine rote Farbe würde dann wie folgt aussehen:

```
glColor3f(1.0, 0.0, 0.0);
```

Grün wäre demnach:

```
glColor3f(0.0, 1.0, 0.0);
```

Und blau entsprechend:

```
glColor3f(0.0, 0.0, 1.0);
```

Probieren Sie es ruhig aus, um ein wenig das Gefühl dafür zu bekommen.

Wie schon oben erwähnt, gibt es in OpenGL Funktionen, die im Prinzip identisch sind, sich aber vom Datentyp bzw. der Anzahl der Parameter unterscheiden.

POWER

Wenn Sie in der Datei »gl.h« nachschauen, werden Sie feststellen, dass es mehr als 30 verschiedene Versionen von `glColor` gibt.

Sie gelangen am schnellsten zu der Definition von `glColor(...)`, indem Sie die Funktion im Quelltext, bei gedrückter Apfel-Taste, doppelklicken.

So, aber nun weiter in unserem Beispiel: Als Nächstes wenden wir uns dem Abschnitt zu, in dem wir unser Dreieck definieren. Wir leiten das Ganze mit dem Befehl `glBegin` ein, gefolgt vom Typus, der die Geometrie beschreibt (in unserem Fall `GL_TRIANGLES`).

TIPP

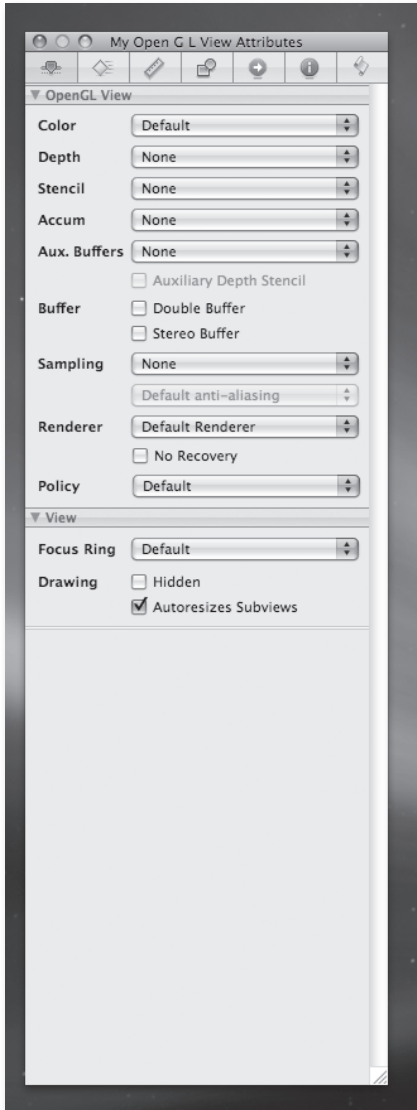
Die geschweiften Klammern nach `glBegin(...)` sind optional und dienen nur der Übersichtlichkeit.

Danach beschreiben wir die Eckpunkte über die Funktion `glVertex3f`, die sich aus jeweils einer X|Y|Z-Position zusammensetzen. Wir beenden unser Dreieck mit der `glEnd`-Funktion, die OpenGL mitteilt, dass keine weiteren Eckpunkte folgen.

Ganz zum Schluss folgt ein `glFlush()`, diese Funktion teilt OpenGL mit, dass alles so schnell wie möglich auf den Schirm erscheinen soll.

So, das war doch gar nicht so schlimm bis jetzt. Ich werde im weiteren Verlauf noch genauer auf die einzelnen Funktionen eingehen.

Gehen wir aber noch mal einen Schritt zurück. Wir öffnen nochmals den Interface Builder und wählen unser *NSOpenGLView* aus. Im ersten Reiter des Inspectors (mit dem kleinen blauen Schieberegler, das Attributes Pane) sehen wir jede Menge Einstellungen, die wir an unserem View vornehmen können.



Die Einstellungen zum *NSOpenGLView*

Diese Einstellungen wollen wir uns nochmals kurz anschauen. Wir gehen sie nacheinander durch.

Color: Hier stellen wir das Farbformat ein.

Depth: Das ist das Format für den Tiefenpuffer (dazu später mehr).

Stencil: Hier wird das Format für den Stencil-Buffer eingestellt. Wie gesagt lassen sich über diesen bestimmte Teile aus dem Renderprozess ausschließen. Sehr oft wird dieser auch für Schatteneffekte benutzt.

Accum: Auch das ist ein Puffer, der oft verwendet wird, um Motion-Blur-Effekte zu erzielen.

Aux. Buffer: Er speichert temporär ein Bild, bevor es in den Accum-Puffer wandert.

Buffer:

- Double Buffer besagt, dass wir Double-Buffering verwenden möchten (was wir später auch tun werden). Das Double-Buffering ist eine oft benutzte Technik, in der im Hintergrund in einen unsichtbaren Puffer gerendert wird. Nachdem der Vorgang abgeschlossen ist, wird der vordere, sichtbare Puffer mit dem hinteren Puffer getauscht. Durch diese Technik wird ein Flackern des Bildes verhindert, welches z. B. bei Animationen auftreten kann.
- Stereo Buffer wird, wie schon der Name sagt, für Stereo-Rendering benutzt. Eine Technik, mit welcher man für jedes Auge ein eigenes Bild berechnet, um es dann mit einer speziellen Brille (Shutter-Glasses) zu betrachten. Hierbei muss man bedenken, dass 2 Framebuffer benötigt werden, die entsprechend den doppelten Speicher auf der Grafikkarte benötigen.

Sampling / Antialiasing: Das ist, wie man vermuten kann, für die Kantenglättung einer Szene zuständig. Wenn man diese einschaltet, hat man mitunter ein besseres Renderergebnis, was sich aber leider auch negativ auf die Performance niederschlägt.

Renderer: Besagt, über welchen Renderer die grafische Ausgabe erfolgen soll. Das hört sich vielleicht seltsam an, könnte man doch meinen, dass die Grafikkarte bzw. nur deren Treiber dafür verantwortlich ist.

Es ist aber so, dass es oft noch einen Software-Renderer gibt, dieser ist zwar nicht hardwarebeschleunigt (daher sein Name), dafür kann man ihn aber wunderbar

dafür benutzen, um Funktionen zu testen, die von der Grafikkarte nicht unterstützt werden.

POWER

Würde man sich den Software-Renderer als Hardware vorstellen, wäre es die Grafikkarte, die am meisten Funktionen unterstützt.

Manchmal wird der Software-Renderer auch »Floating Point Software Renderer« oder einfach nur »Floating Renderer« genannt.

No Recovery: Besagt, dass, wenn es z. B. zu Problemen mit dem hardwarebeschleunigten Renderer kommen sollte (vielleicht zu wenig RAM), OpenGL nicht automatisch auf einen anderen Renderer umschalten soll.

Policy: Hier muss man ein wenig ausholen. Wie man weiß, sind nicht alle Grafikkarten gleichermaßen leistungsfähig. Als Entwickler stößt man sehr schnell an ein Problem, wenn man Grafiksoftware entwickeln möchte, die aber leider nur auf den topaktuellen Grafikkarten funktioniert.

Aus diesem Grund gibt es diese sogenannten Policies.

Nehmen wir als Beispiel ein Pixelformat, das so aussieht (Pseudocode):

```
ColorSize: 24-Bit  
DepthSize: 16-Bit  
MinimumPolicy
```

was dann so viel bedeuten würde wie: Die Grafikkarte (bzw. das System) muss mindestens diese Voraussetzungen erfüllen in Bezug auf ColorSize und DepthSize.

Wenn Sie nochmals im Interface Builder (im Inspector), im Drop-Down Feld für *Policy*, nachschauen, sehen Sie die Werte, die Sie vergeben können. Wenn Sie *maximum* wählen, ist das so zu verstehen, dass mindestens das Pixelformat, das Sie angegeben haben, verfügbar sein muss, aber das größtmögliche Format verwendet werden soll.

Minimum und *Maximum* sind nur für den Color-, Depth- und den Accumulation-Buffer verfügbar.

Closest ist nur für den ColorBuffer verfügbar.

Wir werden uns das jetzt in einem praktischen Beispiel mal genauer anschauen.

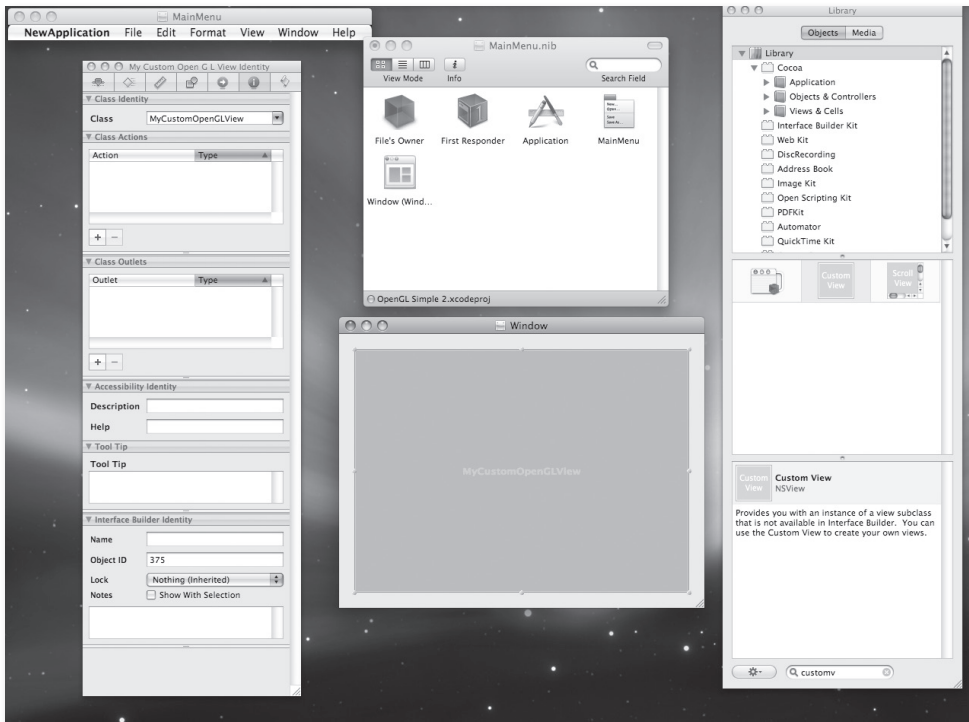
OpenGL-Anwendung mit einem *NSView*

Wie schon mehrfach angedeutet, führen mehrere Wege nach Rom, wenn es darum geht, mit OpenGL zu rendern. Wir werden uns nun die zweite Möglichkeit anschauen, nämlich diejenige, wie man aus einem »normalen« View eine OpenGL-Anwendung machen kann. Dies hat einen enormen Vorteil im Vergleich zu unserem ersten Beispiel. Dadurch kann man nämlich den OpenGL-Kontext mehrfach nutzen und z. B. in zwei Fenstern denselben Content ausgeben.

Auf an die Arbeit, im Prinzip unterscheiden sich die Schritte nur geringfügig von denen im obigen Beispiel, weshalb ich diese nur nochmals kurz erläutern werde.

Sie finden das fertige Beispiel im Ordner »Kapitel 1/OpenGL Simple 2«.

21. *Xcode File | new Project | Cocoa Application*
22. Neue Unterklasse von *NSView* erstellen (Xcode | File | Cocoa | Objective-C *NSView* subclass)
23. OpenGL-Header einfügen `<OpenGL/glu.h>`
24. OpenGL-Framework importieren
25. »MainMenu.nib« doppelklicken (Interface Builder starten)
26. So, bis hierher. Anders als vorhin ziehen Sie nun bitte nicht das *NSOpenGLView* in das Fenster, sondern ein *NSCustomView*.
27. Das View wieder positionieren und das Autosizing wie gehabt einstellen.
28. Als Class geben Sie bitte wieder den Namen ein, den Sie vorhin gewählt haben. Bei mir wäre das *MyCustomOpenGLView*.



Wieder im Interface Builder, diesmal mit einem **NSCustomView**

Wenn Sie alle Einstellungen im Interface Builder getätigt haben, können Sie diesen wieder schließen (speichern bitte nicht vergessen).

Zurück in Xcode öffnen Sie bitte die »CustomView.h«-Datei und geben folgenden Code ein:

```
#import <Cocoa/Cocoa.h>
#import <OpenGL/glu.h>
@interface MyCustomOpenGLView : NSView
{
    NSOpenGLContext    *_context;
    NSOpenGLPixelFormat *_pixelFormat;
}
```

Ergänzen Sie nun bitte noch die Datei »CustomView.m« um den Code, der fett formatiert ist:

```
#import "MyCustomOpenGLView.h"
```

```

@implementation MyCustomOpenGLView
- (id)initWithFrame:(NSRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        NSOpenGLPixelFormatAttribute attributes[] = {
            NSOpenGLPFAWindow,
            NSOpenGLPFAAccelerated,
            NSOpenGLPFAColorSize, 24,
            NSOpenGLPFAMinimumPolicy,
            0
        };
        _pixelFormat = [[NSOpenGLPixelFormat alloc]
            initWithAttributes:attributes];
        if(!_pixelFormat)
        {
            NSLog(@"Fehler beim Anlegen des Pixel Formats");
        }
    }
    _context = nil;
    return self;
}
- (void) dealloc
{
    [_pixelFormat release];
    [_context release];
    [super dealloc];
}
- (NSOpenGLContext*)openGLContext
{
    if(!_context)
    {
        _context = [[NSOpenGLContext alloc] initWithFormat:_pixel-
            Format shareContext:nil];
        if(!_context)
        {
            NSLog(@"Fehler beim Anlegen des OpenGL Context");
        }
    }
    return (_context);
}

```

```

-(void)lockFocus
{
    [super lockFocus];
    NSOpenGLContext * c = [self openGLContext];
    if([c view] != self)
    {
        [c setView:self];
    }
    [c makeCurrentContext];
}
- (void) prepareOpenGL
{
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );
}
-(void) drawRect: (NSRect) bounds
{
    [_context update];
    glViewport( 0, 0, (GLsizei)bounds.size.width, (GLsizei)
    bounds.size.height);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 45.0, bounds.size.width / bounds.size.height,
    1.0, 5.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 0.85f, 0.35f);
    glBegin(GL_TRIANGLES);
    {
        glVertex3f( -0.2, -0.3, -2.0);
        glVertex3f( 0.2, -0.3, -2.0);
        glVertex3f( 0.0, 0.3, -2.0);
    }
    glEnd();
    glFlush();
}

```

Bitte übersetzen und starten (Menü *Build* | *Build and Go*) Sie nun die Anwendung. Wenn alles geklappt hat, sollten Sie die gleiche Ausgabe wie vorhin haben.

Wie man unschwer erkennen kann, ist es nun doch schon mehr Code, der aber den gleichen Zweck erfüllt. Schauen wir uns nun den Code genauer an. Wie schon ein-

mal erwähnt, ist ein *NSOpenGLView* im Prinzip nichts anderes als eine »Kombination« aus einem *NSView*, *NSOpenGLPixelFormat* und *NSOpenGLContext*, weshalb wir die Objekte für den Kontext und das Pixelformat in der Header-Datei mit

```
@interface MyCustomOpenGLView : NSView
{
    NSOpenGLContext    *_context;
    NSOpenGLPixelFormat *_pixelFormat;
}
```

bekannt machen müssen.

In der *initWithFrame*-Methode erzeugen wir unser Pixelformat mit den Attributen:

<i>NSOpenGLPFAWindow</i>	Fenstermodus
<i>NSOpenGLPFAAccelerated</i>	wir möchten einen hardwarebeschleunigten Renderer
<i>NSOpenGLPFAColorSize</i>	Farbtiefe
<i>NSOpenGLPFAMinimumPolicy</i>	unsere minimale Anforderung

Danach erzeugen wir einen Kontext mittels

```
-(NSOpenGLContext*)openGLContext
{
    if(!_context)
    {
        _context = [[NSOpenGLContext alloc] initWithFormat:_pixel-
            Format shareContext:nil];
        if(!_context)
        {
            NSLog(@"Fehler beim Anlegen des OpenGL Context");
        }
    }
    return (_context);
}
```

Dann wird das View für das Zeichnen vorbereitet und unser Kontext aktiviert:

```
-(void)lockFocus
```

```
{
    [super lockFocus];
    NSOpenGLContext * c = [self openGLContext];
    if([c view] != self)
    {
        [c setView:self];
    }
    [c makeCurrentContext];
}
```

Die *prepareOpenGL* ist identisch geblieben. Geändert hat sich nur noch die *drawRect*-Methode. Da ein *NSView* nicht über eine *reshape*-Methode verfügt, müssen wir den Code, der vorher darin stand, nun in die *drawRect*-Methode verlagern.

Wie gesagt, was der Code im Einzelnen macht, besprechen wir später ausführlich.

Wichtig ist noch der Aufruf von

```
[_context update];
```

was dem Kontext mitteilen soll, dass sich unter Umständen die Größe unseres Views geändert hat und er deshalb aktualisiert werden muss.

TIPP

Kommentieren Sie einmal die Zeile aus und starten Sie das Programm noch einmal. Verändern Sie bitte die Größe des Fensters und beobachten Sie dann, was passiert.

Wie Sie sehen, wird der Inhalt nun nicht mehr richtig aktualisiert.

Shared Context

Es besteht auch die Möglichkeit, dass mehrere *NSView*'s, sich einen Kontext teilen.

Ich möchte hier jetzt nicht näher auf die unzähligen Möglichkeiten, die man damit hat, eingehen. Ich habe ein Beispielprojekt erstellt, um zu zeigen, wie ein solcher »Shared Kontext« aussehen kann. Sie finden es im Ordner »Kapitel 1/OpenGL Shared Context«.



Beide Views greifen auf den gleichen Kontext zu.

Fullscreen-Anwendung

Die letzte Möglichkeit, die ich noch zeigen möchte, beschreibt, wie man eine Fullscreen-Anwendung (»Kapitel 1/Fullscreen«) erstellen kann, was für die Spieleentwicklung wichtig ist, weil man in der Regel ja nichts vom Desktop sehen will.

Das *NSView* selbst bietet über die Methode

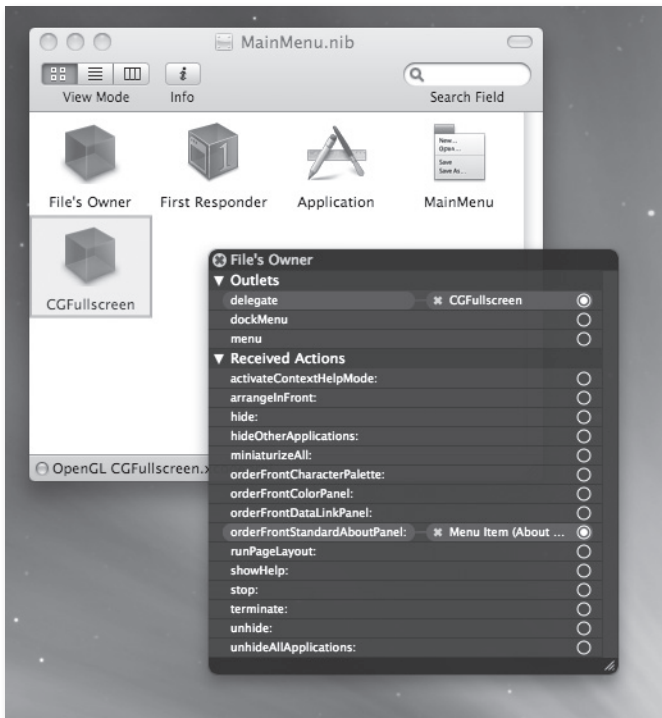
```
- (BOOL)enterFullscreenMode:(NSScreen *)screen withOptions:
(NSDictionary *)options
```

eine entsprechende Möglichkeit, um in einen Fullscreen-Modus umzuschalten.

Wenn wir allerdings ohne ein *NSView* auskommen möchten, müssen wir uns eine andere Möglichkeit suchen. Apple bietet uns gleich mehrere dieser Möglichkeiten. Eine davon möchte ich Ihnen hier zeigen. Zuerst erstellen Sie bitte ein neues Projekt, wie nachfolgend nochmals beschrieben.

1. Xcode File | New Project | Cocoa Application
2. OpenGL-Header einfügen `<OpenGL/glu.h>`
3. OpenGL-Framework importieren
4. Dem Project eine neue Objective-C Klasse hinzufügen (z. B. Fullscreen)

5. Instanzieren Sie nun bitte die neue Klasse im Interface Builder, indem Sie aus der Library des Interface Builders ein *NSObject* (blauer Würfel) in das *MainMenu.nib*-Fenster ziehen und als Klassen-Namen den Namen eingeben, welchen Sie Ihrer Klasse gegeben haben (bei mir Fullscreen).
6. Zum Schluss benötigen wir noch ein *delegate*-Outlet von *File's Owner* zu der neuen Klasse.



Jemand muss die Arbeit machen. Unsere Klasse wird zum *delegate*-Knecht.

Speichern Sie nun bitte Ihre Arbeit im Interface Builder und beenden Sie diesen wieder.

Zurück in Xcode vervollständigen Sie nun bitte Ihre Header-Datei um folgenden Inhalt:

```
#import <Cocoa/Cocoa.h>
#import <OpenGL/glu.h>
@interface Fullscreen : NSObject {
    BOOL          _stayInFullscreenMode;
    NSOpenGLContext *_context;
```

```
        GLsizei    _screenWidth;  
        GLsizei    _screenHeight;  
    }  
    - (BOOL)captureDisplay;  
    - (void)enterMainLoop;  
@end
```

Die Datei »Fullscreen.m« ergänzen Sie bitte mit diesem Code:

```
#import "Fullscreen.h"  
@implementation Fullscreen  
- (void) applicationDidFinishLaunching: (NSNotification *) note;  
{  
    _screenWidth = 1024;  
    _screenHeight = 768;  
    if([self captureDisplay])  
    {  
        [self enterMainLoop];  
    }  
    [_context clearDrawable];  
    [_context release];  
    CGReleaseAllDisplays();  
    [[NSApplication sharedApplication] terminate:self];  
}  
- (BOOL)captureDisplay  
{  
    CGDisplayErr err;  
    //Bildschirm sichern  
    err = CGDisplayCapture(kCGDirectMainDisplay);  
    if(err != CGDisplayNoErr)  
    {  
        NSLog(@"Fehler: CGDisplayCapture");  
        return NO;  
    }  
    //Modus holen  
    CFDictionaryRef newMode = CGDisplayBestModeForParameters  
    (kCGDirectMainDisplay, 24, _screenWidth, _screenHeight,  
    NULL);  
    if(err != CGDisplayNoErr)  
    {  
        NSLog(@"Fehler: CGDisplayBestModeForParameters");  
        return NO;  
    }  
}
```

```

    }
    //In Modus umschalten
    err = CGDisplaySwitchToMode(kCGDirectMainDisplay, newMode);
    if(err != CGDisplayNoErr)
    {
        NSLog(@"Fehler: CGDisplaySwitchToMode");
        return NO;
    }
    NSOpenGLPixelFormatAttribute attr[] = {
        NSOpenGLPFADoubleBuffer,
        NSOpenGLPFAAccelerated,
        NSOpenGLPFAColorSize, 24,
        NSOpenGLPFADepthSize, 24,
        NSOpenGLPFAFullScreen,
        NSOpenGLPFAScreenMask,CGDisplayIDToOpenGLDisplay-
        Mask(kCGDirectMainDisplay),
    0 };
    //PixelFormat erstellen
    NSOpenGLPixelFormat *pixelFormat = [[NSOpenGLPixelFormat
    alloc] initWithAttributes:attr];
    //Context erstellen
    _context = [[NSOpenGLContext alloc] initWithFormat:
    pixelFormat shareContext: nil];
    [pixelFormat release];
    //Context setzten
    [_context makeCurrentContext];
    [_context setFullScreen];
    return YES;
}
- (void)enterMainLoop
{
    glClearColor( 0.2f, 0.2f, 0.2f, 1.0f );
    glViewport( 0, 0, _screenWidth, _screenHeight);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    float ratio = (float)_screenWidth / (float)_screenHeight;
    gluPerspective( 45.0, ratio, 1.0, 5.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    _stayInFullScreenMode = YES;
    while ( _stayInFullScreenMode )
    {

```

```

NSAutoreleasePool *pool = [[NSAutoreleasePool alloc]
init];
NSEvent *event;
event = [NSApp nextEventMatchingMask:NSAnyEventMask
          untilDate:[NSDate distantPast]
          inMode:NSDefaultRunLoopMode
          dequeue:YES];
NSEventType type = [event type];
if(type == NSKeyDown)
{
    if([event keyCode] == 53) // ESC
        _stayInFullscreenMode = NO;
}
glClear(GL_COLOR_BUFFER_BIT);
glLoadIdentity();
glColor3f(1.0f, 0.85f, 0.35f);
glBegin(GL_TRIANGLES);
{
    glVertex3f( -0.2, -0.3, -2.0);
    glVertex3f(  0.2, -0.3, -2.0);
    glVertex3f(  0.0,  0.3, -2.0);
}
glEnd();
[_context flushBuffer];
[pool release];
}
@end

```

Wenn Sie nun das Programm kompilieren und starten (Menü *Build | Build and Go*), sollten Sie die gewohnte Ausgabe haben, diesmal im Fullscreen-Modus und mit einem grauen Hintergrund.

Das Programm verlassen Sie mit der *ESC*-Taste.

Schauen wir uns nun den Code an:

Da wir unsere Klasse als *delegate* von *File's Owner* definiert haben, bekommen wir auch die Nachricht

```
- (void) applicationDidFinishLaunching: (NSNotification *) note;
```

Darin definieren wir die beiden Variablen, welche unsere Bildschirmauflösung festlegen.

```
_screenWidth = 1024;  
_screenHeight = 768;
```

Das wirklich Wichtige passiert nun in der Methode:

```
-(BOOL)captureDisplay
```

Hier sichern wir zuerst unseren Hauptbildschirm, so dass kein anderes Programm mehr Zugriff darauf hat. Sollte ein anderes Programm dasselbe vor uns gemacht haben, bekommen wir einen Fehler.

GRUNDLAGEN

Es kann immer nur eine Anwendung den exklusiven Zugriff auf eine Full-screen-Anwendung haben.

Danach holen wir uns über

```
CFDictionaryRef newMode = CGDisplayBestModeForParameters  
(kCGDirectMainDisplay, 24, _screenWidth, _screenHeight, NULL);
```

den Modus, der am besten zu unseren Einstellungen passen würde.

Das bedeutet aber auch, dass wir nicht unbedingt das bekommen, was wir anfordern. Möchten wir ein etwas exotisches Format wie z. B. 900x800 Pixel, werden wir vermutlich einen 1024x768-Modus zurückbekommen, weil unser Monitor bzw. die Grafikkarte dieses Format nicht unterstützt.

Sollte das geklappt haben, schalten wir in den neuen Modus um.

Zum Schluss erstellen wir noch unser Pixelformat und den OpenGL-Kontext, was nichts Neues mehr ist.

Schauen wir uns noch die

```
-(void)enterMainLoop
```

genauer an. Wir erstellen uns zuerst einen eigenen *NSAutoreleasePool* – dies ist nötig, da wir uns in einer Endlosschleife befinden – und fangen dann das *NSKey-*

Down-Ereignis ab. Dort reagieren wir auf die *ESC*-Taste, die uns wieder aus der Schleife bringt.

Im Rendercode musste nun

```
glFlush();
```

dem Aufruf von

```
[_context flushBuffer];
```

weichen.

Das hängt damit zusammen, dass wir nun ein Pixelformat mit einem Double-Buffer erstellt haben und der Aufruf von

```
[_context flushBuffer];
```

schon ein *glFlush* enthält.

Wie gesagt wird beim Double-Buffering der Kontext in einen unsichtbaren Puffer im Hintergrund geschrieben, und mit *flushBuffer* werden dann der Front-Buffer und der Back-Buffer einfach getauscht.

GRUNDLAGEN

Das Tauschen der beiden Puffer ist recht simpel: Es werden einfach die jeweiligen Zeiger auf die Puffer miteinander vertauscht.

Ich möchte nochmals darauf hinweisen, dass wir die OpenGL-spezifischen Dinge aus diesem Kapitel noch ausführlicher kennen lernen werden. Also keine Sorge, wenn Sie noch nicht alles verstanden haben.

Im nächsten Kapitel werden wir uns einmal anschauen, was wir an mathematischen Kenntnissen benötigen, um einen ersten Einstieg in die 3D-Programmierung zu bekommen.

Zusätzliche Informationen

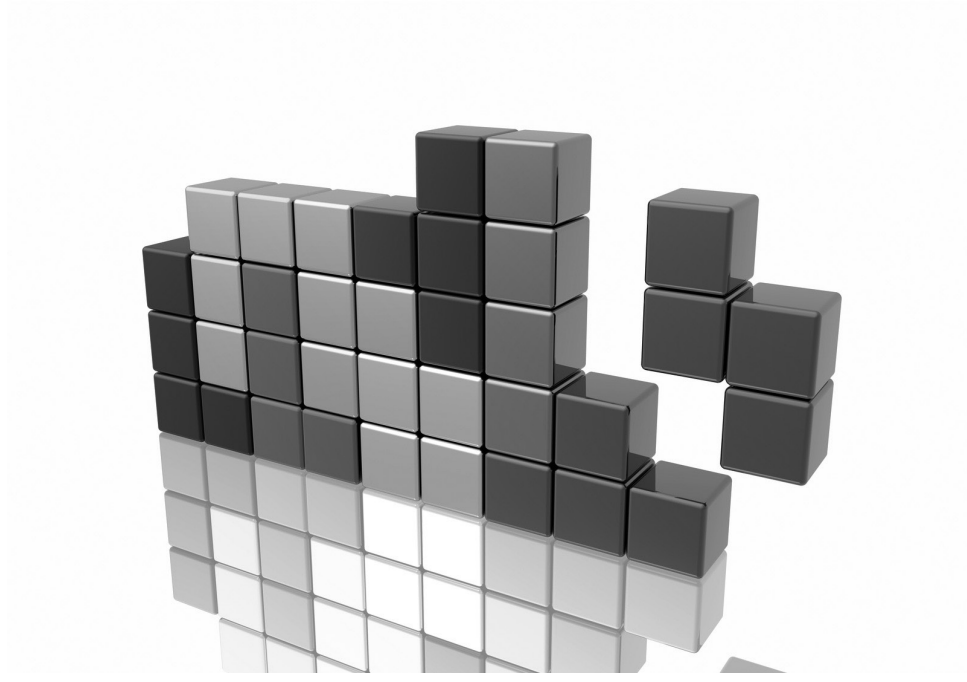
OpenGL Programming Guide for Mac OS X
In der Hilfe zu Xcode enthalten

<http://www.opengl.org/>
Heimat von OpenGL. Hier gibt es auch komplette Bücher zu OpenGL im HTML-Format.

<http://nehe.gamedev.net/>
Der Klassiker unter den OpenGL-Seiten von Jeff Mollofee

<http://www.openal.org/>
Webseite zu OpenAL

<http://www.libsdl.org/>
Webseite von SDL



Mathematik

In der Spielprogrammierung geht so gut wie gar nichts ohne Mathematik, weshalb wir uns in diesem Kapitel die Grundlagen, die wir benötigen, erarbeiten werden.

Wir werden auch gleich damit anfangen, uns einige Wrapper-Klassen zu erstellen, die wir für unser Spiel, welches wir zum Ende des Buchs erstellen wollen, brauchen werden. Sie finden die Klassen im Ordner »CFX«.

TIPP

Der Ausdruck Wrapper-Klassen ist wohl besser bekannt als 3D-Engine. Ich verzichte aber gerne auf diese Definition, da es zu einem Modewort der Spielentwicklung geworden ist. Im Internet kursieren hunderte solcher »3D-Engines«, die es gerade so schaffen, einen simplen Würfel auf den Schirm zu bringen.

Ich verspreche Ihnen, dass ich es so kurz wie nötig halten werde, da das Thema doch recht trocken ist. Aber ganz ohne geht es dann leider auch nicht.

In der 3D-Computergrafik benötigen wir eine Möglichkeit, die Geometrie, die wir rendern möchten, auch zu beschreiben. Egal, wie komplex diese auch immer ist, man definiert sie immer nach dem gleichen Schema.

Skalare, Punkte und Vektoren

Schauen wir uns zuerst einmal an, was wir brauchen, um einen geometrischen Körper zu erstellen.

Skalar:

Skalare sind einfach Zahlenwerte wie z. B. 0, 1, 2, 3, -3, -5.

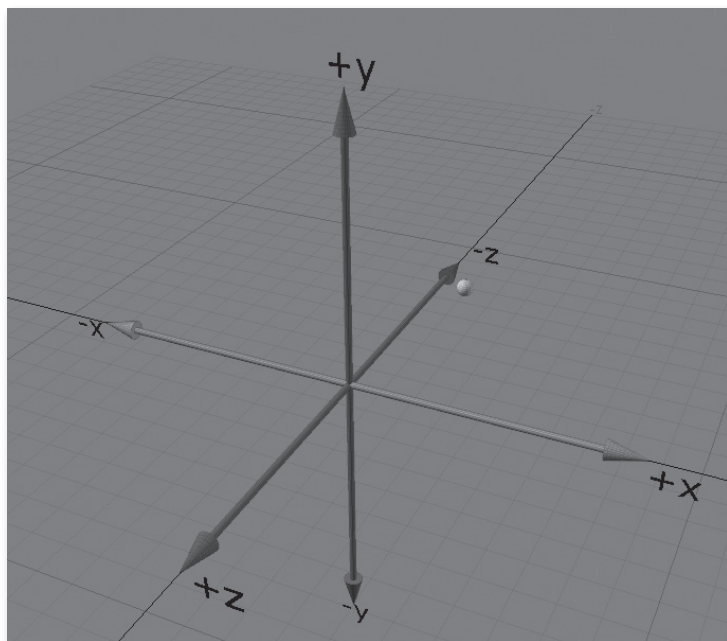
Wenn man es mehr wissenschaftlich betrachten möchte, ist ein Skalar eine physikalische Größe, die nur einen Zahlenwert besitzt, aber keine Richtung hat.

Ich persönlich mag die erste Definition lieber.

Punkte:

Ein Punkt definiert zuerst einmal nur eine Position im 3D-Raum. Dieser setzt sich aus 3 Skalaren (der X|Y|Z-Koordinate) zusammen, die den Abstand zum Ursprung (0,0,0) des globalen Koordinatensystems beschreiben.

Was sich jetzt vielleicht kompliziert anhört, lässt sich in einem Bild sehr einfach darstellen.



Ein einzelner Punkt im 3D-Raum

Der Ursprung ($x=0, y=0, z=0$) ist dort, wo sich alle 3 Achsen schneiden. Die Kugel befindet sich demnach bei den Koordinaten:

$x= 1.0$

$y= 1.0$

$z= -1.0$

TIPP

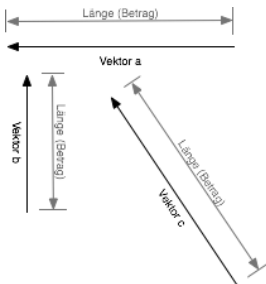
In diesem Zusammenhang möchte ich darauf hinweisen, dass man in der 3D-Grafik nicht von Punkten spricht, sondern von einem Vertex bzw. Vertices.

Vektoren:

Eines der wichtigsten Elemente der 3D-Programmierung ist der Vektor.

Man kann ihn sich als einen Pfeil vorstellen, der im Raum »liegt«. Das bedeutet, dass ein Vektor einen Ursprung und entsprechend eine Länge (Betrag) besitzt.

Schauen wir uns zunächst mal an, wie so ein Vektor aussehen kann.



3 Vektoren mit unterschiedlicher Richtung und Länge

Alle 3 dieser Vektoren haben einen Ursprung und zeigen in eine beliebige Richtung.

Allgemein wird ein Vektor mit einem Pfeil $\vec{a}$ darüber notiert. Manchmal aber auch nur fett geschrieben »**a**« beide Schreibweisen sind korrekt und würden den Vektor **a** beschreiben.

Ortsvektor: Ist ein Vektor, der im Ursprung des Koordinatensystems beginnt und in eine beliebige Richtung zeigt.

Beispiel (Pseudo):

```
Vector v1 = 1.0, 2.0, 3.0; // x,y,z
```

Dieser Vektor zeigt nun vom Ursprung ($x=0, y=0, z=0$) nach ($x=1, y=2, z=3$)

Richtungsvektor: Ist ein sogenannter freier Vektor, der durch 2 beliebige Koordinatenpaare angegeben wird. Man erhält diesen Vektor indem man die Differenz aus dem Endpunkt und dem Anfangspunkt bildet.

AUFGEPASST

Vorsicht: Die Reihenfolge ist unbedingt einzuhalten, damit der Vektor in die richtige Richtung zeigt!

Vektor-Grundlagen

Berechnung: Vektorlänge

Hat man einen Vektor, möchte man unter Umständen auch seine Länge wissen.

TIPP

Die Vektorlänge ist z. B. dafür nützlich, wenn man wissen möchte, wie weit ein Objekt (z. B. der Gegner) vom Spieler entfernt ist.

Diese wird mit 2 vertikalen Strichen notiert $|\mathbf{a}|$. Wie man an der Formel sehen kann, handelt es sich um den Satz des Pythagoras, mit dem Unterschied, dass wir ja einen 3-Dimensionalen Raum haben und entsprechend die Z-Komponente mit dazu nehmen müssen.

Formel: $|\mathbf{a}| = \sqrt{x^2 + y^2 + z^2}$

Beispiel:

$$x = 1.0$$

$$y = 2.0$$

$$z = 3.0$$

```
float length = sqrt((1.0*1.0 + 2.0*2.0 + 3.0*3.0));
```

length wäre dann: 3.741657

Berechnung: Einheitsvektor (normierter Vektor)

Unter einem Einheitsvektor versteht man einen Vektor mit der Länge von 1.

Das interessante an einem Einheitsvektor ist, das er, nachdem er »normalisiert« wurde, immer noch in die gleiche Richtung zeigt, aber eben nur noch die Länge von 1 hat.

TIPP

Gerade im Umgang mit der Beleuchtung von 3D-Objekten spielt dieser Vektor eine besondere Rolle, was wir im Kapitel 5 noch sehen werden.

Um einen Einheitsvektor zu erhalten, dividiert man ihn einfach durch seine Länge.

Formel: $n = a / |a|$

Beispiel:

Vektor $a = (x|y|z)$

Länge: $|a| = \sqrt{(x^2 + y^2 + z^2)}$

Einheitsvektor:

$a = (x|y|z) / (|a|)$

$a = (x|y|z) / (\sqrt{(x^2 + y^2 + z^2)})$

Berechnung: Vektor-Rechenoperationen

Mit Vektoren kann man genauso rechnen wie mit Skalaren.

Nachfolgend eine Vektor-Addition. Bei der Subtraktion ändert man einfach den Operator:

Formel: $a + b$

Beispiel:

Vektor $v1$

Vektor $v2$

Vektor $v3 = (v1.x + v2.x, v1.y + v2.y, v1.z + v2.z)$



Eine Addition von 2 Vektoren: Das Ergebnis ist ein dritter Vektor.

Berechnung: Multiplikation mit einem Skalar

Vektoren können auch mit einem Skalar multipliziert werden. Dabei ändert sich seine Richtung nicht, sondern nur seine Länge.

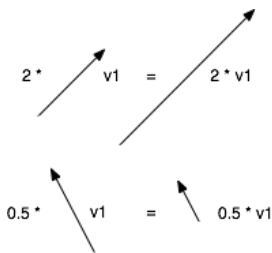
Formel: $s * v1$

Beispiel:

$v1.x * s;$

$v1.y * s;$

$v1.z * s;$



Vektoren kann man auch mit Skalaren multiplizieren.

POWER

Wenn Sie einen Vektor mit -1 multiplizieren, erhalten Sie den gleichen Vektor, der aber in die entgegengesetzte Richtung zeigt.

Berechnung: Punktprodukt (Dotproduct, Innerproduct, Skalarprodukt)

Das Punktprodukt oder besser bekannt als das »Dotproduct« zieht sich durch die ganze Palette der 3D-Programmierung. Man braucht es, um den Winkel zwischen 2 Vektoren zu ermitteln.

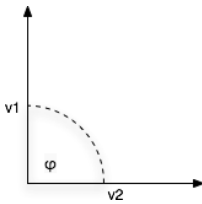
Das Punktprodukt wird mit einem ».« notiert.

Formel:

$$v1.v2 = |v1| * |v2| * \cos(\varphi)$$

Beispiel:

```
float tmp = (v1.x*v2.x) + (v1.y*v2.y) + (v1.z*v2.z);
float dot = acosf (tmp)
```



Mit dem Punktprodukt kann man den Winkel zwischen 2 Vektoren berechnen.

GRUNDLAGEN

Wie man am obigen Beispiel unschwer erkennen kann, liefert das Punktprodukt eine Zahl und keinen Vektor!

Hier noch ein paar wichtige Eigenschaften des Punktproduktes, die man sich merken sollte:

- $A \cdot B = 0$ Wenn der Winkel zwischen den beiden Vektoren 90 Grad ist.
- $A \cdot B < 0$ Wenn der Winkel zwischen den beiden Vektoren > 90 Grad ist.
- $A \cdot B > 0$ Wenn der Winkel zwischen den beiden Vektoren < 90 Grad ist.

Berechnung: Kreuzprodukt (Crossproduct)

Genau wie das Punktprodukt ist das Kreuzprodukt ein wichtiger Teil in der 3D-Programmierung, wir benötigen es z. B. für die Beleuchtung oder die Kollisionserkennung.

TIPP

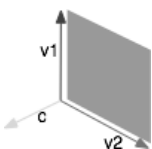
Wenn man sich nun 2 Vektoren vorstellt, die ein Parallelogramm aufspannen, ist das Kreuzprodukt ein Vektor, der senkrecht darauf steht.

Formel: $v1 \times v2$

Beispiel:

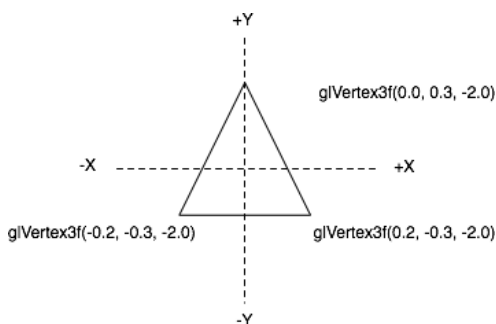
Die Elemente der beiden Vektoren werden nun überkreuz multipliziert.

$$c = (v1.y \cdot v2.z - v1.z \cdot v2.y, v1.z \cdot v2.x - v1.x \cdot v2.z, v1.x \cdot v2.y - v1.y \cdot v2.x)$$



Das Kreuzprodukt

So, mit dem ganzen Wissen über Skalare, Vertices und Vektoren können wir nun auch das Geheimnis lüften, wie wir das Dreieck aus den vorherigen Beispielen auf den Schirm gebracht haben.



So haben wir unser Dreieck definiert.

Der erste Vertex ist links unten, der zweite rechts unten und der dritte in der Mitte oben.

Wie Sie sehen, ist das Dreieck um den globalen Ursprung definiert. Wir geben in der *glVertex*-Funktion einfach die einzelnen XYZ-Koordinaten an, um dieses Dreieck zu definieren.

TIPP

Sie sollten es sich angewöhnen, die Vertices immer in der gleichen Reihenfolge zu definieren. Dies ist zwar kein Muss, erleichtert aber die Lesbarkeit ungemein.

So viel zu den Vektoren. Es gibt mit Sicherheit noch sehr viele Dinge, die man mit Vektoren machen kann. Aber wie gesagt, möchte ich keine komplizierte Abhandlung darüber halten. Für den Anfang genügen uns erst mal diese Informationen. Im Laufe des Buches werden Sie genügend Möglichkeiten bekommen, das hier Besprochene zu verwenden bzw. zu vertiefen.

Die Datei »CFXVector.h« beinhaltet die wichtigsten Operationen mit den Vektoren. Weiterhin gibt es noch die Datei »CFXMath.h«, welche einige mathematische Hilfsroutinen beinhaltet.

Im nächsten Abschnitt werden wir sehen, wie man das Dreieck (natürlich auch alle anderen geometrischen Formen) z. B. rotieren kann.

Matrizen-Grundlagen

Nicht weniger wichtig für uns sind Matrizen. Diese sind nichts anderes als eine Tabelle aus Zahlen (so eine Art zweidimensionales Array). Wie bei einer Tabelle üblich, werden diese Zahlen in Spalten und Zeilen gespeichert. Eine bestimmte Art von Matrize haben wir schon kennengelernt, nämlich die mit einer Spalte ($n=1$), welche ein Vektor ist. Eine Matrize kann folgendermaßen aussehen:

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \quad \text{Eine } 3 \times 3\text{-Matrize}$$

Der Zugriff auf die Werte erfolgt über die Spalten und Zeilen.

Der Wert aus »m11« wäre demnach bei (1,1) und nicht, wie man vielleicht vermuten möchte, bei (2,2). Wie gesagt, eine Matrize ist wie ein Array definiert, dort fangen wir ja bei 0 an zu zählen.

OpenGL arbeitet mit einer sogenannten homogenen Matrize, die aus 4×4 Elementen besteht und somit eine Gesamttransformation speichern kann. Bevor Sie jetzt Kopfschmerzen bekommen, hier mal eine Abbildung dazu:

$$\begin{pmatrix} R_x & R_x & R_x & T_x \\ R_y & R_y & R_y & T_y \\ R_z & R_z & R_z & T_z \\ u & v & w & \end{pmatrix} \quad \text{Eine homogene } 4 \times 4\text{-Matrix, die eine Gesamttransformation speichern kann}$$

Diese Matrize ist folgendermaßen aufgebaut:

R speichert die Informationen, wenn wir ein Objekt rotieren möchten. **T** beinhaltet hingegen die Informationen für eine Verschiebung. Man kann zudem auch sofort erkennen, wo die entsprechenden Koordinaten gespeichert sind:

- **x** erste Reihe
- **y** zweite Reihe
- **z** dritte Reihe

Die Werte $u \ v \ w$ stehen für die Projektionstransformation, dazu aber später mehr.

So, da wir nun wissen, wie Matrizen aussehen, schauen wir uns mal an, wie diese funktionieren.

Schauen wir uns zunächst die wohl wichtigste Matrize an, die »Einheitsmatrize«:

$$E = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Die Einheitsmatrix}$$

Die Einheitsmatrize (Identitymatrix) dient als Initialisierungsmatrize und wird in OpenGL mittels `glLoadIdentity()` aufgerufen.

GRUNDLAGEN

Wenn wir `LoadIdentity()` aufrufen, wird unsere aktuell gesetzte Matrize durch eine Einheitsmatrize ersetzt.

Sie ist völlig neutral aufgebaut und wird normalerweise immer ganz am Anfang der Zeichenroutine aufgerufen. Das ist insofern wichtig, da alle anderen Matrizen-Operationen (Rotation, Verschiebung, Skalierung) unmittelbar aufeinander aufbauen. Wir sehen das gleich noch weiter unten.

POWER

Die Operationen Verschieben, Rotieren und Skalieren werden üblicherweise unter dem Sammelbegriff »Transformation« zusammengefasst.

Verschiebung:

Wenn wir nun einen Punkt im 3D-Raum verschieben möchten, multiplizieren wir seine

XYZ-Koordinaten mit den entsprechenden Elementen für die Verschiebung (also die »T«-Reihe) in der Matrize:

$$MT * p = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}$$

Wir verschieben einen Punkt, indem wir ihn mit der Matrix multiplizieren.

Hier wird der Punkt »p« durch die Matrize »MT« verschoben. Zusammenfassend sieht die Verschiebung dann folgendermaßen aus:

$$x' = x + dx$$

$$y' = y + dy$$

$$z' = z + dz$$

Rotation:

Natürlich können wir auch Objekte im Raum drehen. Schauen wir uns dazu zunächst die Rotation um die X-Achse an:

$$m_{RX} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Eine Rotation um die X-Achse}$$

Wie man sehen kann, werden bei einer Rotation um die X-Achse die Werte für X (oberste Reihe) nicht verändert.

Analog dazu hier eine Rotation um die Y-Achse:

$$m_{RY} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Rotieren um die Y-Achse}$$

Und zum Schluss noch die Z-Achse:

$$m_{RZ} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{Wir können natürlich auch um die Z-Achse rotieren.}$$

Auch hierzu nochmals eine Zusammenfassung. Wenn wir einen Vektor um die Z-Achse rotieren möchten, sähe das so aus:

$$x' = (x \cdot \cos \theta) - (y \cdot \sin \theta)$$

$$y' = (x \cdot \sin \theta) + (y \cdot \cos \theta)$$

$$z' = z$$

Sie sehen, der Z-Wert bleibt unverändert, was auch klar sein dürfte. Stellen Sie sich vor, Sie stecken einen Bleistift durch ein Blatt Papier, welches Sie vor sich halten: Wenn Sie nun das Papier drehen, bleibt der Bleistift (Z-Achse) unverändert.

Skalierung:

Wenn wir ein Objekt in seiner Größe verändern möchten, dann tun wir das über eine Skalierungsmatrize, die wie folgt aussieht:

$$m_S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Elemente der Skalierung sind diagonal angeordnet.

Bei einer Skalierung werden die einzelnen Komponenten eines Vektors mit den entsprechenden Komponenten (s_x , s_y , s_z) der Matrize multipliziert.

$$x' = x * s_x$$

$$y' = y * s_y$$

$$z' = z * s_z$$

So, erst mal geschafft. Ich kann Sie beruhigen, Sie müssen das jetzt nicht alles auswendig lernen, es geht lediglich darum, dass Sie den mathematischen Hintergrund einmal gesehen haben.

Erfreulicherweise nimmt uns OpenGL in Bezug auf Matrizen eine Menge Arbeit ab, so dass wir uns entspannt zurücklehnen können.

Wir werden anschließend zu diesem Thema eine praktische Übung machen, und ich verspreche Ihnen, Sie werden erstaunt sein, wie einfach das funktioniert.

Matrizen in OpenGL

Schauen wir uns aber zuvor noch an, welche Matrizen es in OpenGL überhaupt gibt und wofür sie zuständig sind.

GL_MODELVIEW: Die Modelviewmatrix

Wenn wir anfangen, Vertexdaten an OpenGL zu senden, sollte diese Matrize aktiv sein.

Diese Matrize wirkt sich direkt auf unsere Vertexdaten aus, die wir z. B. mit *glVertex* übergeben. Des Weiteren beeinflusst sie die normalen, die mit *glNormal* gesetzt werden.

GL_PROJECTION: Die Projektionsmatrix

Diese Matrice wirkt sich darauf aus, **wie** unsere Objekte später auf den Bildschirm projiziert werden. Man kann sie sich wie eine virtuelle Kamera vorstellen. Wie das funktioniert, werden wir uns in einem späteren Kapitel nochmals genauer anschauen. Wichtig ist im Moment nur, dass Sie wissen, dass es eine Projektionsmatrix gibt.

GL_TEXTURE: Die Texturmatrix

Wenn diese Matrice aktiv ist, betreffen die Transformationen, die wir vornehmen, nicht die Geometrie selbst, sondern die Texturen der Geometrie.

Auch das Thema Texturen kommt später noch ausführlich an die Reihe. Es genügt auch hier erst einmal, dass Sie wissen, dass es eine solche Matrice gibt.

Matrizen verwenden

Da wir jetzt wissen, welche Matrizen es gibt, brauchen wir auch eine Möglichkeit, diese zu aktivieren, um mit ihnen zu arbeiten.

In OpenGL tun wir das über den Befehl

```
glMatrixMode( );
```

und geben als Parameter die Matrice an, die wir gerne aktivieren möchten. Als Beispiel aktivieren wir mal die Modelviewmatrix:

```
glMatrixMode( GL_MODELVIEW );
```

TIPP

Wir haben ja zu Anfang gehört, dass OpenGL wie eine State-Machine arbeitet, folglich werden dann alle Manipulationen an jener Matrice gemacht, welche zuletzt aktiviert wurde, bitte nicht vergessen!

So, wie schon weiter oben erwähnt, kann man ganz interessante Dinge mit Matrizen machen.

Verschieben

Wenn wir z. B. ein Objekt im Raum verschieben möchten, machen wir das mit der OpenGL-Funktion:

```
glTranslatef (GLfloat x, GLfloat y, GLfloat z);
```

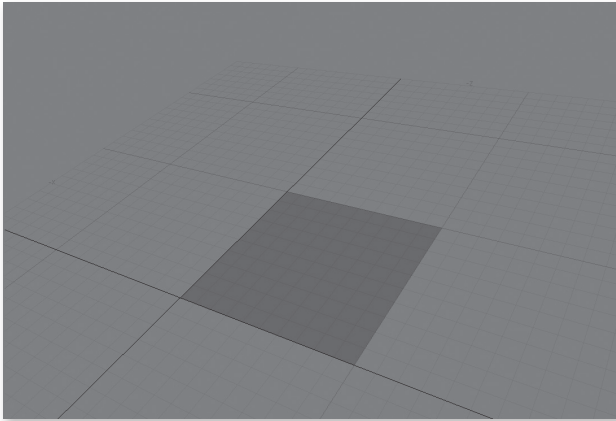
Diese Funktion multipliziert die aktuell gesetzte Matrice (in der Regel die Modelviewmatrix) mit einer Verschiebungsmatrice, die wir in Form eines Vektors übergeben.

Wenn wir also schreiben:

```
glTranslatef (1.0, 0.0, -3.0);
```

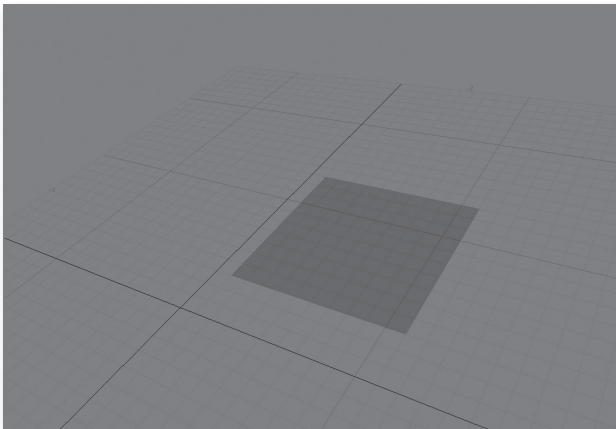
dann werden **alle** Vertexdaten, die anschließend folgen, um jeweils 1.0 Einheit nach rechts
0.0 Einheiten nach oben (also keine Verschiebung) und
-3.0 Einheiten nach hinten (in dem Raum hinein) verschoben.

Vor dem Verschieben:



Die Geometrie vor einer Verschiebung

Nach der Verschiebung:



*Die Geometrie, nachdem wir sie mit **glTranslatef(...)** verschoben haben.*

Schauen wir uns den kompletten Code an:

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glTranslatef(1.0, 0.0, -3.0);  
DrawBluePlane(); //Pseudo
```

Zuerst aktivieren wir die Modelviewmatrix, weil sich die Verschiebung auf unser Objekt beziehen soll. Danach rufen wir die Funktion *glLoadIdentity()*; auf, diese überschreibt die Modelviewmatrix mit einer Einheitsmatrize (welche ja neutral aufgebaut ist).

GRUNDLAGEN

Die Funktion *glLoadIdentity()*; wird in der Regel ganz am Anfang des Rendervorgangs (nachdem die Modelviewmatrix gesetzt wurde) aufgerufen. Dadurch haben wir eine »saubere« Matrize, auf welcher alle folgenden Transformationen basieren.

Dann erfolgt die eigentliche Verschiebung mittels *glTranslatef(...)*, und zum Schluss wird die Geometrie ausgegeben.

Wie Sie sehen, ist der Ablauf sehr intuitiv. Allein die Funktionsnamen verraten uns schon, was hier alles passiert.

Rotieren

Das Rotieren funktioniert ähnlich und wird in OpenGL mittels

```
glRotatef (GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

gemacht.

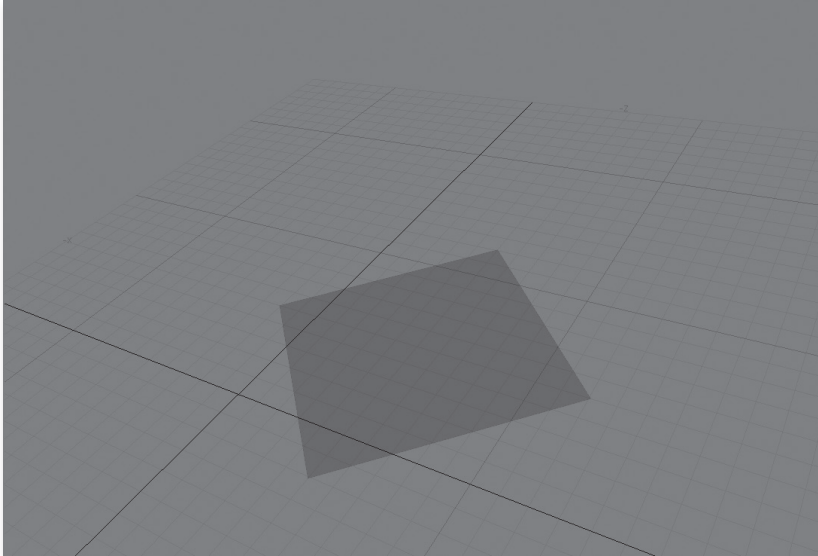
Die Funktion erwartet als ersten Parameter den Winkel in Grad, den wir rotieren möchten, und als weitere Parameter die Rotationsachse.

TIPP

Negative Gradangaben drehen im Uhrzeigersinn, positive entsprechend entgegengesetzt.


```
glRotatef (45.0, 0.0, 1.0, 0.0);
```

dreht die Geometrie um 45 Grad gegen den Uhrzeiger auf der Y-Achse.



Eine Rotation von 45 Grad gegen den Uhrzeigersinn

GRUNDLAGEN

Eine Rotation wird immer um den lokalen Ursprung der Geometrie gemacht.

Skalieren

So, und nun noch die Skalierung die in OpenGL mittels

```
glScalef (GLfloat x, GLfloat y, GLfloat z);
```

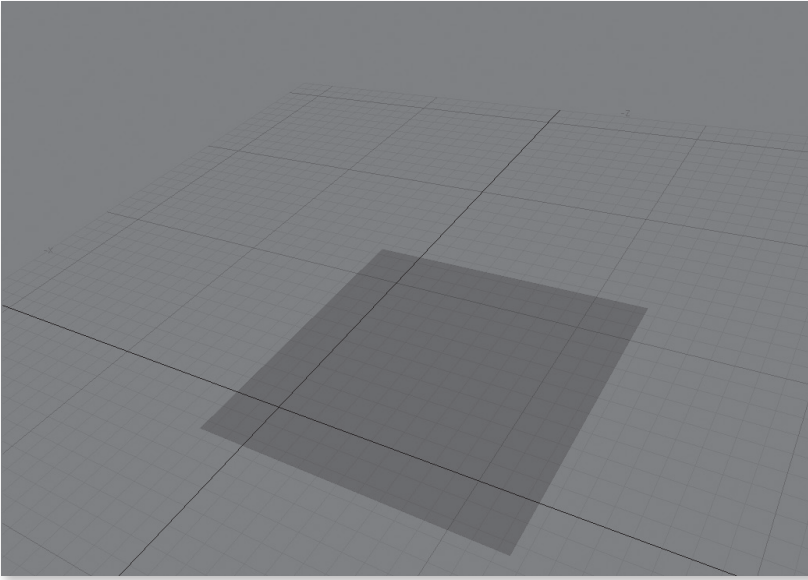
gemacht wird.

Wir geben hier die Parameter der Skalierung für die einzelnen Achsen an:

Hier eine Skalierung um 150 Prozent auf der X und Z-Achse:

```
glScalef (1.5, 1.0, 1.5 );
```

Die Y-Achse bleibt dabei unverändert. Werte kleiner als 1.0 verkleinern ein Objekt entsprechend.



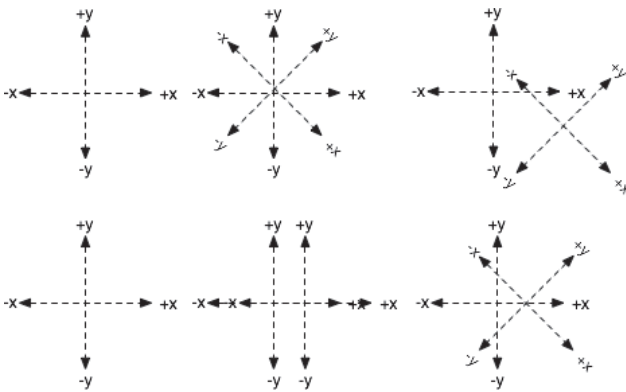
Eine Skalierung auf der XZ-Achse

Reihenfolge der Transformationen

Nun ist aber die Reihenfolge der einzelnen Transformationen sehr wichtig.

Schauen wir uns dazu folgende Grafik an:

Im oberen Bild wird zuerst rotiert und dann verschoben und im unteren Bild genau anders herum.



Oben wird zuerst rotiert und dann verschoben, unten genau anders herum.

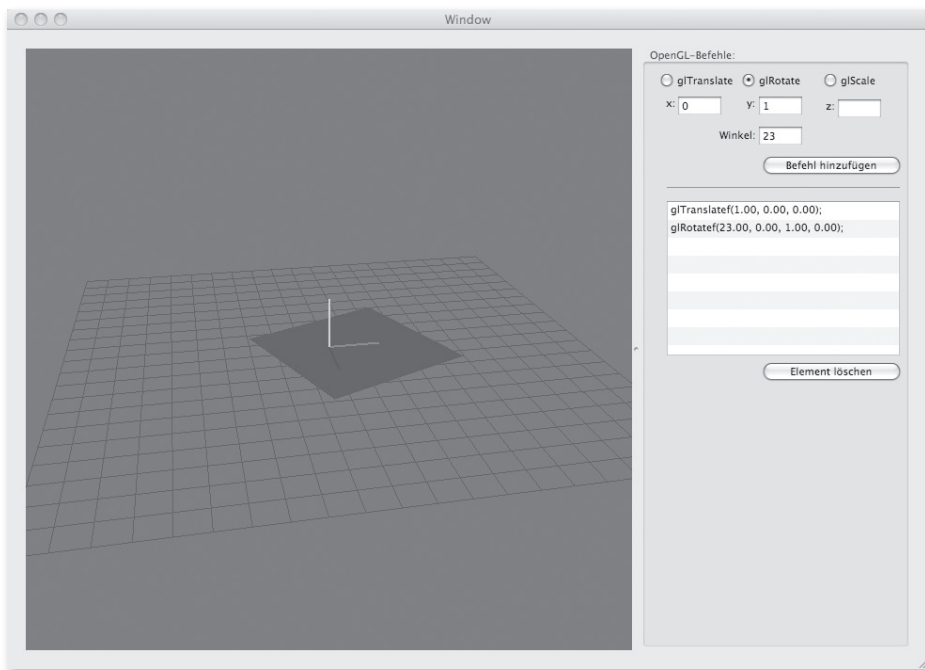
Man erkennt sofort den Unterschied. Aber warum ist das so?

Bei einer Rotation wird das Objekt um seinen eigenen **lokalen** Ursprung rotiert, danach ist entsprechend sein lokales Koordinatensystem auch rotiert. Wenn wir uns den oberen Teil der Grafik nochmals anschauen, wird es klarer.

Wir machen zuerst eine 45-Grad-Rotation um die Z-Achse, anschließend zeigt die lokale X-Achse nach schräg rechts unten (dort, wo +X steht).

Wenn wir jetzt das Objekt auf der X-Achse verschieben, befindet es sich entsprechend dort, wo die X-Achse hinzeigt (schräg rechts unten), und nicht wie gewohnt rechts neben dem Ursprung (so wie in dem unteren Teil der Grafik).

Damit Sie ein wenig das Gefühl dafür bekommen, was diese 3 Befehle machen, habe ich ein kleines Testprogramm dazu geschrieben, Sie finden es im Ordner »Kapitel 2/Transformations«.



Testprogramm zu den Transformationen

Auf der linken Seite des Fensters sehen Sie eine Livevorschau der Szene. Der Raster hat einen Abstand von einer Einheit und erstreckt sich um jeweils 10 Einheiten um den globalen Ursprung der Szene.

Zur besseren Orientierung habe ich die 3 Achsen (rot, grün, blau) mit eingebaut, die sich genau am Ursprung befinden (0, 0, 0).

Das blaue Quadrat in der Mitte ist das Objekt, das Sie manipulieren können.

Dazu finden Sie auf der rechten Seite eine Art Menü, mit dem Sie die 3 OpenGL-Funktionen eingeben können, in der Tabelle unten sehen Sie dann die Reihenfolge, in der die Befehle abgearbeitet werden.

Der Quellcode hat einige Funktionen, die Sie bis jetzt noch nicht kennengelernt haben. Stören Sie sich bitte nicht daran, wir besprechen diese noch ausführlich.

Eigene Matrizen

OpenGL bietet auf auch die Möglichkeit, eigene Matrizen zu verwenden. Dies kann z. B. dann sinnvoll sein, wenn Sie ein Objekt anhand einer Rotation eines anderen Objektes ausrichten müssen und dadurch die Rotationsmatrize speichern müssen. Oder aber, wenn Sie bestimmte Effekte an einem Objekt erzielen wollen, die Sie mit den Standard-Funktionen von OpenGL nicht realisieren können.

Matrize laden

Um überhaupt mit eigenen Matrizen arbeiten zu können, müssen Sie diese zunächst einmal laden. Dies geschieht mit Hilfe der Funktion

```
glLoadMatrixf (const GLfloat *matrix);
```

Dabei wird die aktuell gesetzte Matrize durch die geladene Matrize ersetzt.

Wenn Sie z. B. eine eigene Einheitsmatrize laden möchten, könnten Sie folgenden Code verwenden:

```
GLfloat identityMatrix[16] ={  
1.0, 0.0, 0.0, 0.0,  
0.0, 1.0, 0.0, 0.0,  
0.0, 0.0, 1.0, 0.0,  
0.0, 0.0, 0.0, 1.0};  
glLoadMatrixf(identityMatrix);
```

Matrize multiplizieren

Eine aktuell gesetzte Matrize kann man mit der Funktion

```
glMultMatrixf(const GLfloat *matrix);
```

multiplizieren.

Auch hierzu nochmals ein kleines Beispiel:

```
float translation[16] = {  
1.0, 0.0, 0.0, 0.0,  
0.0, 1.0, 0.0, 0.0,  
0.0, 0.0, 1.0, 0.0,  
0.0, 0.0, -10.0, 1.0};  
glMultMatrixf(translation);
```

Diese Anweisung würde alle nachfolgenden Vertices um 10 Einheiten auf der Z-Achse nach hinten verschieben.

TIPP

Noch ein Tipp zur Verwendung eigener Matrizen: Wann immer es geht, sollten Sie die eingebauten Matriz-Funktionen von OpenGL nutzen, da diese in der Regel hardwarebeschleunigt verarbeitet werden.

Matrix-Stapel

Wie wir eben bei den Matrizen gelernt haben, beeinflusst eine Veränderung der Modelviewmatrix die komplette nachfolgende Geometrie.

Wenn wir z. B. 2 Objekte unabhängig voneinander verschieben wollen (beide um eine Einheit nach rechts), könnte das so aussehen:

```
glMatrixMode( GL_MODELVIEW );  
glLoadIdentity();  
glTranslatef( 1.0, 0.0, 0.0 );  
renderGeometrie();  
glTranslatef( 1.0, 0.0, 0.0 );  
renderAndereGeometrie();
```

Doch was auf den ersten Blick logisch aussieht, entpuppt sich leider als »Fehler«. Das zweite Objekt wird dabei nämlich um zwei und nicht, wie man vielleicht vermuten könnte, um eine Einheit verschoben. Aber warum ist das so?

Schauen wir uns kurz Schritt für Schritt an, was da passiert:

Zuerst aktivieren wir die Modelviewmatrix und ersetzen sie durch die Einheitsmatrix.

Also sieht sie im Moment so aus (beachten Sie bitte die letzte Spalte, die ja für eine Verschiebung zuständig ist):

```
1.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0
0.0 0.0 1.0 0.0
0.0 0.0 0.0 1.0
```

Dann kommt unsere erste Transformation um 1 Einheit nach rechts:

```
1.0 0.0 0.0 1.0
0.0 1.0 0.0 0.0
0.0 0.0 1.0 0.0
0.0 0.0 0.0 1.0
```

Zum Schluss erfolgt noch die zweite Transformation, also sieht die Modelviewmatrix so aus:

```
1.0 0.0 0.0 2.0
0.0 1.0 0.0 0.0
0.0 0.0 1.0 0.0
0.0 0.0 0.0 1.0
```

TIPP

Sie können den Inhalt einer Matrice jederzeit über den Befehl `glGetFloatv (GLenum pname, GLfloat *params);` abfragen.

Die zweite Verschiebung ist also abhängig von der ersten.

Das ist aber nicht das, was wir möchten. Wir wollen ja beide Objekte **unabhängig** voneinander verschieben. Man könnte nun auf die Idee kommen und zwischen- durch einfach `glLoadIdentity();` aufrufen.

Hier mal ein Codeausschnitt, wie das aussehen könnte:

```
glMatrixMode( GL_MODELVIEW );  
glLoadIdentity();  
glTranslatef( 1.0, 0.0, 0.0 );  
renderGeometrie();  
glLoadIdentity();  
glTranslatef( 1.0, 0.0, 0.0 );  
renderAndereGeometrie();
```

Nun, das würde so auch funktionieren, wäre aber höchst ineffizient. OpenGL bietet hier eine bessere Möglichkeit, diese nennt sich »Matrix-Stapel«.

OpenGL kennt folgende 3 Matrix-Stapel:

- Modelviewmatrix-Stapel
- Projektionsmatrix-Stapel
- Texturmatrix-Stapel

Wie der Name schon vermuten lässt, kann man sich diese wie einen Stapel mit Containern vorstellen, die jeweils eine 4x4-Matrize speichern können.

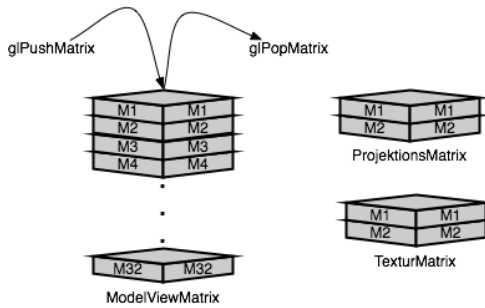
Zu Beginn eines OpenGL-Programmes ist die Stapelhöhe immer 1 hoch. Man legt etwas auf diesen Stapel mit dem Befehl

```
glPushMatrix();
```

was so viel bedeutet wie: »Speichere meine aktuell gesetzte Matrize«. Man könnte auch einfach sagen: »Hey OpenGL, merk dir mal kurz, was in der Matrize steht«.

```
glPopMatrix();
```

macht genau das Gegenteil, nämlich entfernt die Matrize wieder vom Stapel.



Matrizen-Stapel unter OpenGL

Leider kann man aber nicht beliebig viele Elemente auf diese Stapel legen. Die maximale Anzahl, die solch ein Stapel aufnehmen kann, lässt sich mit folgenden Funktionen ermitteln:

```
glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, &depth);
glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, &depth);
glGetIntegerv(GL_MAX_TEXTURE_STACK_DEPTH, &depth);
```

Schauen wir uns nochmals das Beispiel von eben an, diesmal aber mit der Stapel-technik und dem Ergebnis, das beide Objekte jetzt um eine Einheit verschoben werden:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glPushMatrix();
glTranslatef( 1.0, 0.0, 0.0 );
rendereGeometrie();
glPopMatrix();
glPushMatrix();
glTranslatef( 1.0, 0.0, 0.0 );
rendereAndereGeometrie();
glPopMatrix();
```

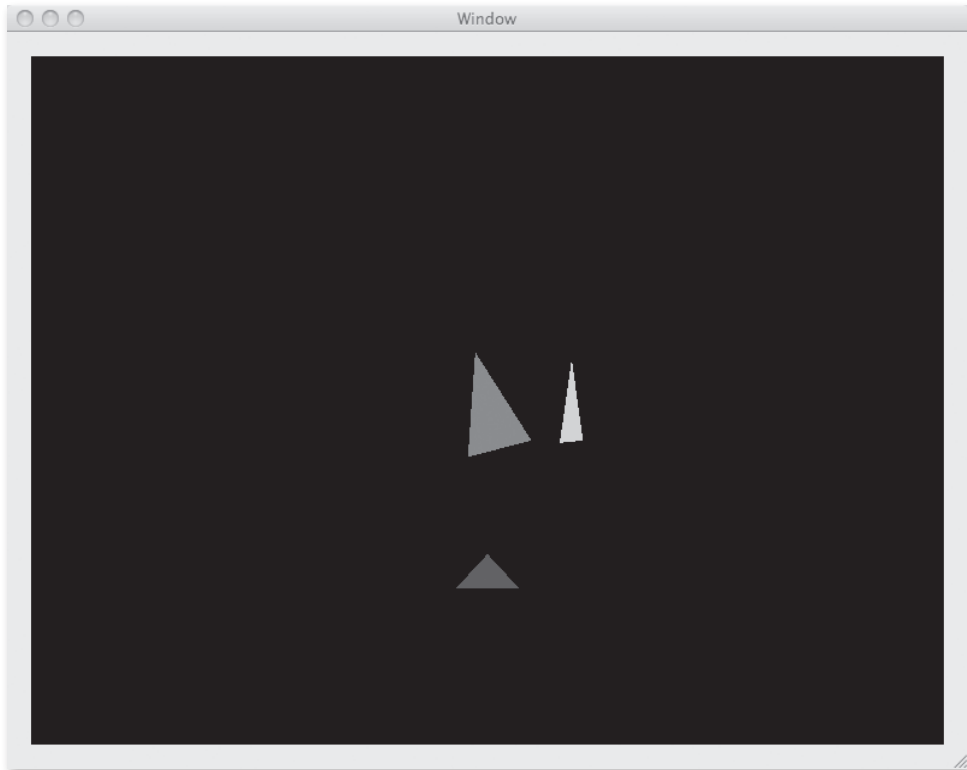
Zu Beginn setzen wir die Modelviewmatrix und laden die Einheitsmatrix, soweit nichts Neues. Danach legen (Push) wir diese Matriz auf den Stapel (OpenGL merkt sich nun das, was darin steht) und wir machen unsere Transformation, anschließend wird die Matriz wieder vom Stapel herunter genommen (Pop).

Bei der zweiten Geometrie machen wir es ebenso.

Durch das Push und Pop haben wir also eine »saubere« Matriz, mit der wir unsere Geometrie transformieren können. Es sieht also für die nachfolgende Transforma-

tion so aus, als hätte die erste zwischen *PushMatrix()* und *PopMatrix()* nie stattgefunden. Das war genau das, was wir wollten.

Da man es an einem praktischen Beispiel natürlich viel besser sieht, habe ich dazu ein kleines Testprogramm geschrieben, »Kapitel 2/MatrixStacks«.



Das Testprogramm *MatrixStacks*

Darin sind 3 Dreiecke (keine Angst, wir werden bald auch andere Sachen auf den Bildschirm zaubern) zu sehen, die sich alle unterschiedlich drehen. Schauen wir uns aber zuerst den Code dazu an.

In der *prepareOpenGL*-Methode habe ich einen Timer eingerichtet, der in einem bestimmten Intervall unserem View sagt, dass es neu gezeichnet werden soll. Ohne diesen Timer wäre die Animation nicht zu sehen, da unser View ja nur dann neu gezeichnet wird, wenn die *reshape*-Methode aufgerufen wird.

Danach habe ich noch die 3 Funktionen zum Abfragen der Stapelhöhe für die einzelnen Matrizen eingebaut, die einfach per *NSLog(...)* ausgegeben werden.

Dies hat keinen Einfluss auf unser Programm, es soll nur zeigen, wie man die Stapelhöhe abfragen kann.

Die folgenden 3 Funktionen

```
glEnable( GL_DEPTH_TEST );  
glDepthFunc( GL_LEQUAL );  
glClearDepth( 1.0f );
```

besprechen wir im nächsten Kapitel.

Nun aber zu dem, was wirklich auf dem Schirm passiert. Gehen wir alle Dreiecke Schritt für Schritt durch.

Zuerst wird eine rote Farbe erzeugt und mit *PushMatrix()* die Modelviewmatrix auf den Stapel gelegt, dann erfolgt eine Rotation um die Z-Achse und anschließend definieren wir ein Dreieck. Zum Schluss holen wir die gespeicherte Matrix wieder vom Stapel herunter.

```
glColor3f(1.0f, 0,0);  
glPushMatrix();  
glRotatef(_rot2, 0.0f, 0.0f, 1.0f);  
glBegin(GL_TRIANGLES);  
{  
    glVertex3f( -0.2, -0.3, 0.0);  
    glVertex3f(  0.2, -0.3, 0.0);  
    glVertex3f(  0.0,  0.3, 0.0);  
}  
glEnd();  
glPopMatrix();
```

Bei dem grünen Dreieck passiert fast dasselbe, mit dem Unterschied, dass hier zuerst eine Rotation erfolgt und dann eine Verschiebung. Wie wir wissen, erreichen wir dadurch eine Rotation, die **nicht** um den lokalen Ursprung des Dreiecks geht. Also kreist dieses Dreieck um das rote von oben.

```
glColor3f(0,1,0);  
glPushMatrix();  
glRotatef(_rot1, 0.0f, 1.0f, 0.0f);  
glTranslatef(1.2f, 0.0f, 0.0f);  
glBegin(GL_TRIANGLES);  
{
```

```
        glVertex3f( -0.2, -0.3, 0.0);  
        glVertex3f(  0.2, -0.3, 0.0);  
        glVertex3f(  0.0,  0.3, 0.0);  
    }  
    glEnd();  
    glPopMatrix();
```

Zum Schluss noch das blaue Dreieck:

Hier machen wir zuerst eine Verschiebung, dann eine Rotation und danach wieder eine Verschiebung.

```
glColor3f(0,0,1);  
glPushMatrix();  
glTranslatef(0.0f, 0.0f, -0.6f);  
glRotatef(_rot1, 1.0f, 0.0f, 0.0f);  
glTranslatef(0.0f, 0.0f, 1.2f);  
glBegin(GL_TRIANGLES);  
{  
    glVertex3f( -0.2, -0.3, 0.0);  
    glVertex3f(  0.2, -0.3, 0.0);  
    glVertex3f(  0.0,  0.3, 0.0);  
}  
glEnd();  
glPopMatrix();
```

Am besten, Sie »spielen« ein wenig mit dem Programm herum, um zu sehen, was passiert.

AUFGEPASST

Hier noch eine kleine Quizfrage:

Sie möchten 1 Dreieck auf dem Schirm ausgeben. Dieses Dreieck soll gleichzeitig skaliert, verschoben und rotiert werden. Hat es Sinn, hier mit der Stapeltechnik zu arbeiten?

Richtig, natürlich nicht, da sich die Transformation sowieso nur auf das eine Dreieck beziehen würde.

Abschließend hier noch eine Befehlsübersicht zu den Matrizen bzw. Stapeln:

Befehl	Beschreibung
<code>glLoadIdentity</code>	lädt die Einheitsmatrix
<code>glMatrixMode</code>	Matrixmodus setzen
<code>glPushMatrix</code>	erhöht den Matrixstapel
<code>glPopMatrix</code>	verringert ihn wieder
<code>glPushAttribute</code>	erhöht den Attributstapel
<code>glPopAttribute</code>	verringert ihn wieder
<code>glPushName</code>	erhöht den Namenstapel
<code>glPopName</code>	verringert ihn wieder

Die Befehle ab *PopMatrix()* werden wir im weiteren Verlauf noch kennenlernen, sie dienen hier nur der Vollständigkeit.

Schlussbemerkung

Ich hoffe, das Thema war nicht allzu »trocken« für Sie. Wie Sie gesehen haben, kommen wir zu Anfang mit recht einfachen Mathematikkenntnissen zurecht.

Keine Bange, wenn Sie vielleicht das eine oder andere noch nicht ganz verinnerlicht haben, in den nächsten Kapiteln machen wir jede Menge Übungen, in denen wir das hier Erlernte wiederholen werden.

Ab dem nächsten Kapitel werden wir richtig Gas geben.

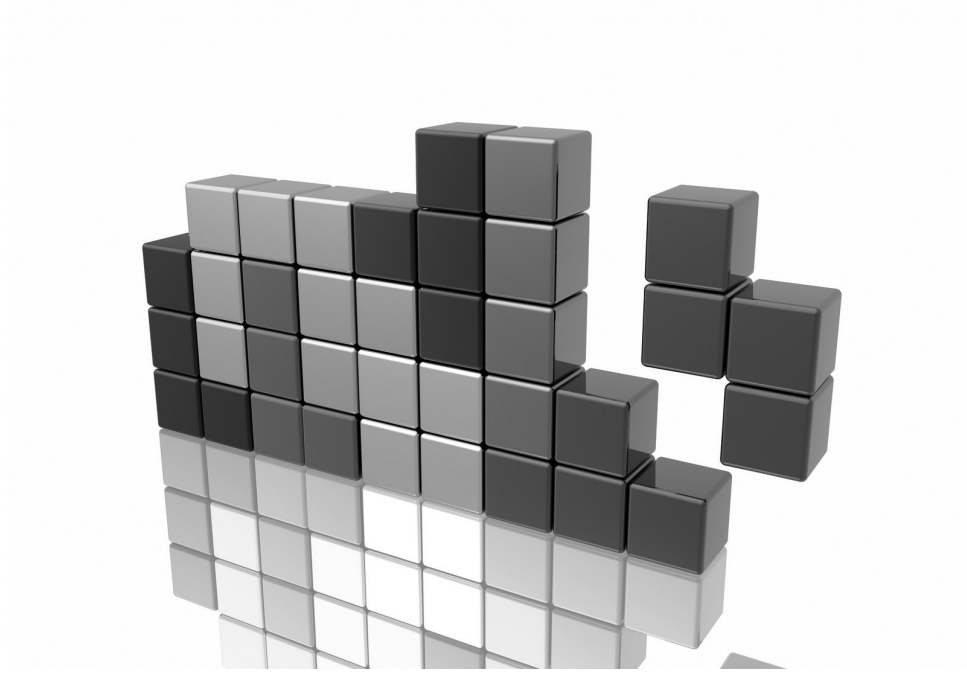
Auf an die Arbeit.

Zusätzliche Informationen

<http://de.wikipedia.org/wiki/Vektor>
Sehr gute Erklärung zu Vektoren

[http://de.wikipedia.org/wiki/Matrix_\(Mathematik\)](http://de.wikipedia.org/wiki/Matrix_(Mathematik))
Hier nochmals für Matrizen

<http://gamemath.com/>
Das Buch für die 3D-Mathematik



Zeichnen in OpenGL

Wie man Dreiecke auf den Bildschirm zaubert, haben Sie nun ja schon gesehen. Nun wird es Zeit, mal etwas anderes zu zeichnen.

Wenn wir Objekte in unserer Szene beschreiben, dann sprechen wir in der 3D-Programmierung von »Primitiven«. Da eine Szene durchaus aus mehreren 1000 dieser Primitiven bestehen kann, ist es zuerst einmal wichtig zu wissen, wie man diese am effektivsten erstellt, bzw. was ihre Eigenschaften sind.

OpenGL selbst kennt 10 verschiedene Arten von Primitiven. Bevor wir nun aber eine davon auf dem Schirm sehen, müssen wir das Primitiv zuerst einmal definieren. Dazu haben wir in den vergangenen Kapiteln den Befehl *glBegin(...)* schon kennengelernt.

Über diesen Befehl haben wir OpenGL mitgeteilt, dass wir nun etwas auf den Schirm zeichnen wollen. Was genau wir zeichnen möchten, erwartet die Funktion *glBegin(...)* als Parameter. Dieser kann einer der folgenden in der Tabelle sein:

Parameter	Beschreibung
GL_POINTS	einzelne Punkte
GL_LINES	nicht verbundene Linien
GL_LINE_STRIP	eine Serie von verbundenen Linien
GL_LINE_LOOP	geschlossene Serie von verbundenen Linien
GL_TRIANGLES	einzelnes Dreieck
GL_TRIANGLE_STRIP	eine Serie von verbundenen Dreiecken
GL_TRIANGLE_FAN	ein Satz Dreiecke, welche einen gemeinsamen Mittelpunkt haben
GL_QUADS	ein einzelnes Viereck
GL_QUAD_STRIP	eine Serie von verbundenen Vierecken
GL_POLYGON	ein Polygon mit einer variablen Anzahl an Eckpunkten

Nachdem wir nun mit *glBegin(...)* unsere Zeichenroutine eingeleitet haben, müssen wir sie mit *glEnd()* wieder abschließen.

AUFGEPASST

Sollten Sie `glEnd()` vergessen, werden Sie wahrscheinlich gar nichts bzw. nicht das, was Sie erwarten, auf dem Bildschirm sehen.

Soweit mal die grundsätzliche Vorgehensweise, wir schauen uns nachfolgend an, was OpenGL uns in dieser Hinsicht zu bieten hat.

Punkte

Beginnen wir mit dem wohl einfachsten, dem Punkt.

Um einen Punkt auf den Bildschirm zu zeichnen, leiten wir unsere Routine ein mit

```
glBegin(GL_POINTS);
```

damit teilen wir OpenGL mit, dass wir einen Punkt zeichnen möchten. Danach folgt die Koordinatenangabe (X|Y|Z)

```
glVertex3f(1.0, 0.0, 0.0);
```

und zum Schluss noch der Befehl

```
glEnd();
```

Wie Sie vielleicht bemerkt haben, steht als Parameter der *glBegin(...)-Funktion* `GL_POINTS`, was darauf hindeutet, dass man anscheinend auch mehrere Punkte zeichnen kann. Genau so ist es, wenn wir mehrere Punkte auf einmal zeichnen möchten, dann tun wir das innerhalb **eines** *glBegin(...)* / *glEnd()*-Blocks und erstellen **nicht** für jeden Punkt einen einzelnen Block.

Das würde zwar auch funktionieren, ist aber sehr ineffektiv.

Hier ein Beispiel, welches 2 Punkte zeichnet:

```
glBegin(GL_POINTS);  
    glVertex3f(1.0, 0.0, 0.0);  
    glVertex3f(2.0, 1.5, -10.2);  
glEnd();
```

TIPP

Um den Code ein wenig zu strukturieren, hat es sich bewährt, entweder die Befehle innerhalb eines *glBegin(...)* / *glEnd()-Blocks* einzurücken oder sie in geschweifte Klammern zu fassen.

Wie in der Einleitung schon erwähnt, gibt es identische OpenGL-Funktionen, die sich nur in der Art und Anzahl des Datentyps unterscheiden. Das ist bei *glVertex(...)* nicht anders, ich habe mehr als 20 verschiedene Versionen zu *glVertex(...)* gezählt, folgende Funktion erwartet z. B. ein Array von 2 Float-Werten:

```
glVertex2fv();
```

Punktgröße

Wenn man Punkte zeichnet, möchte man unter Umständen auch die Punktgröße ändern, dies geschieht über die Funktion

```
glPointSize(GLfloat size);
```

Als Parameter übergibt man die neue Größe (Standardwert ist 1.0).

TIPP

Nicht alle Punktgrößen werden unterstützt.

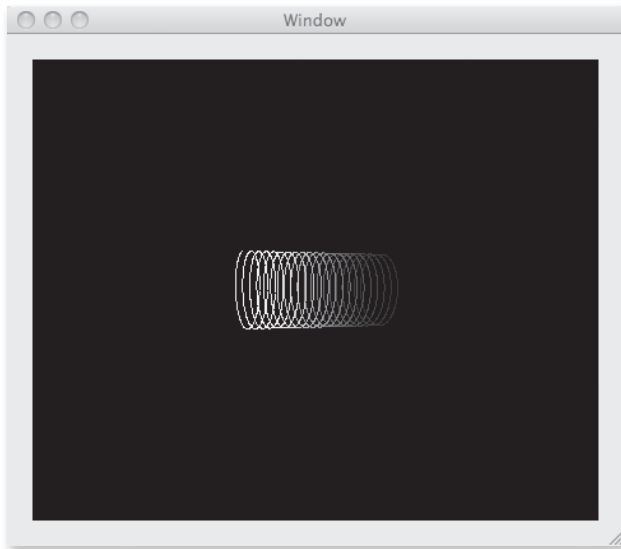
```
GLfloat sizes[2];  
glGetFloat(GL_POINT_SIZE_RANGE, sizes);
```

liefert einen Bereich, der angibt, welche Größen erlaubt sind.

```
GLfloat step;  
glGetFloat(GL_POINT_SIZE_GRANULARITY, &step);
```

liefert die Schrittweite, mit welcher man die Größe ändern kann.

Achtung! Die Punktgröße wird definiert, bevor der *glBegin(...)-Block* beginnt.



Testprogramm »Points«, welches eine Spirale zeichnet

Das Testprogramm »Kapitel 2/Points« zeichnet eine Spirale mit Hilfe von einzelnen Punkten.

Hier noch mal der relevante Code:

```
glTranslatef(0.0, 0.0, -15.0);
glRotatef(_rotation, 0.0f, 1.0f, 0.0f);
glBegin(GL_POINTS);
while(z<2.0)
{
    r = 0.5 * (1 + sin(z));
    g = 0.5 * (1 + sin(z));
    b = 0.5 * (1 + cos(z));
    glColor3f(r, g, b);
    glVertex3f(sin(_angle), cos(_angle), z);
    _angle=_angle+(3.1415*0.01f);
    z=z+0.001f;
}
glEnd();
_rotation+=2.0f;
if(_rotation > 360.0)
    _rotation = 0.0;
glFlush();
```

Zuerst wird die Geometrie um 15 Einheiten auf der Z-Achse in den Raum hinein verschoben und anschließend auf der Y-Achse rotiert. Auch hier kommt wieder ein Timer zum Einsatz, den Sie ja schon aus dem vergangenen Kapitel kennen.

Danach wird die eigentliche Zeichenroutine mit

```
glBegin(GL_POINTS);
```

eingeleitet und mit Hilfe der beiden Winkelfunktionen (*sin / cos*) eine Spirale definiert.

Zum Schluss wird noch die Rotation inkrementiert und mittels

```
glFlush();
```

die Ausgabe erzwungen.

TIPP

Wie Sie sehen, ist es auch möglich, innerhalb des Zeichenblocks eine andere Farbe zu definieren, in meinem Beispiel entsteht dadurch ein Farbverlauf.

Linien

Das Zeichnen von Linien funktioniert im Prinzip genauso wie das Zeichnen von Punkten.

```
glBegin(GL_LINES);  
    glVertex3f(1.0, 0.0, 0.0);  
    glVertex3f(10.0, 0.0, 0.0);  
glEnd();
```

zeichnet eine Linie von $X = 1.0$ nach $X = 10.0$.

Auch hier gilt: Sie können so viele Linien, wie Sie möchten, in dem Block definieren. Wenn Sie eine ungültige Zahl an Definitionen angeben, wird der Befehl einfach ignoriert.

```
glBegin(GL_LINES);  
    glVertex3f(1.0, 0.0, 0.0);  
    glVertex3f(10.0, 0.0, 0.0);  
    glVertex3f(11.0, 1.0, 2.0);  
glEnd();
```

Der letzte *glVertex-Befehl* wird hier einfach verworfen, da ein einzelner Punkt natürlich keine Linie ergibt.

Linienstärke

Wie bei den Punkten, so ist es auch hier möglich, die Linienstärke zu verändern. Die Vorgehensweise ist identisch mit der, die schon oben beschrieben wurde. Einzige Parameter der Funktion *glGetFloat(...)* sind entsprechend anders.

```
glGetFloat(GL_LINE_WIDTH_RANGE, sizes);  
glGetFloat(GL_LINE_WIDTH_GRANULARITY, &step);
```

Linienmuster

Linien haben zudem noch die Besonderheit, dass man ein Muster erstellen kann, mit welchem die Linien dann gezeichnet werden.

TIPP

Das Erstellen eines Linienmusters wird üblicherweise als »Stippling« bezeichnet.

Um das Stippling zu nutzen, wird es zunächst über

```
glEnable(GL_LINE_STIPPLE);
```

eingeschaltet. Danach wird mit der Funktion

```
glLineStipple (GLint factor, GLushort pattern);
```

das Muster festgelegt.

Der *pattern-Parameter* erwartet einen 16-bit Hex-Wert, der das Muster definiert, und *factor* legt den Multiplikator für jedes Bit im Muster fest.

Hier ein kleines Beispiel:

```
glLineStipple (3, 0xAAAA);  
glBegin (GL_LINES);  
glVertex3f (0.0, 0.0, 0.0);  
glVertex3f (5.0, 0.0, 0.0);  
glEnd ();
```

Binär geschrieben wäre das Muster

`0000100010001000`

jede 1 würde demnach gezeichnet und jede 0 entsprechend nicht.

AUFGEPASST

Das Muster wird immer von hinten nach vorne gelesen.

Also könnte man es so interpretieren:

3 Bits aus
1 Bit an
3 Bits aus

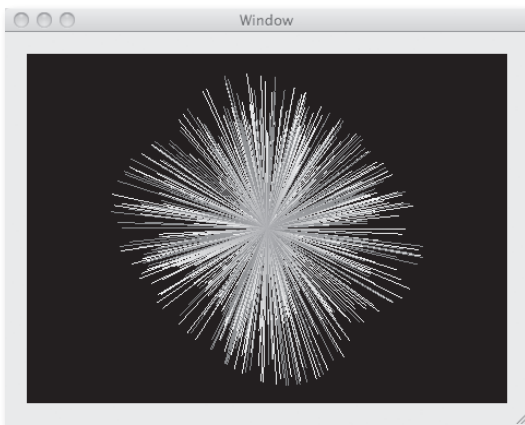
usw.

Wie gesagt, dient der *factor-Parameter* als Skalierung, in unserem Beispiel hat er den Wert 3, wonach unser Pattern dann so ausgegeben würde:

9 Bits aus
3 Bits an
9 Bits aus

usw.

Auch zu den Linien habe ich ein kleines Testprogramm »Kapitel 3/Lines« geschrieben.



Das Lines-Testprogramm

Zu Demonstrationszwecken habe ich einmal die beiden Dateien »CFXVector.h« und »CFXMath.h« verwendet, die wir im Kapitel über Vektoren und Matrizen erstellt haben.

Hier der relevante Code, der 360 Linien zeichnet und anschließend auf 2 Achsen rotiert.

```
typedef struct _Line
{
    CFXVector v1;
    CFXVector v2;
    float rotation;
    float speed;
}myLine;
@interface MyOpenGLView : NSOpenGLView {
    myLine _lines[360];
}
@end
@implementation MyOpenGLView
- (void) prepareOpenGL
{
    int i;
    float length = 1.5f;
    float x;
    float y;
    float z;
    for(i=0; i<360; i++)
    {
        x = length * cos(i);
        y = length * sin(i);
        z = length * cos(i);
        _lines[i].v1 = makeVector(-x,y,z);
        _lines[i].v2 = makeVector(x,-y,-z);
        _lines[i].rotation = randomFloat(1.0, 359.0);
        _lines[i].speed = randomFloat(0.1, 0.4);
    }
}
-(void) drawRect: (NSRect) bounds
{
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -6.0);
```

```

int i;
for(i=0; i<360; i++)
{
    glPushMatrix();
    glRotatef(-_lines[i].rotation, 1.0, 1.0, 0.0);
    glBegin (GL_LINES);
        glColor3f (1.0, 1.0, 0.0);
        CFXVector v1 = _lines[i].v1;
        glVertex3f (v1.x, v1.y, v1.z);
        glColor3f (1.0, 0.0, 0.0);
        CFXVector v2 = _lines[i].v2;
        glVertex3f (v2.x, v2.y, v2.z);
    glEnd ();
    glPopMatrix();
    _lines[i].rotation+=_lines[i].speed;
    if(_lines[i].rotation>360.0)
        _lines[i].rotation = randomFloat(1.0, 359.0);
}
glFlush();
}

```

Wie Sie sehen, habe ich zuerst die Struktur *myLine* definiert, die später jeweils eine Linie darstellen soll.

In der *prepareOpenGL-Methode*, werden dann wieder, über die beiden Winkel-funktionen, die Positionen der Linien berechnet.

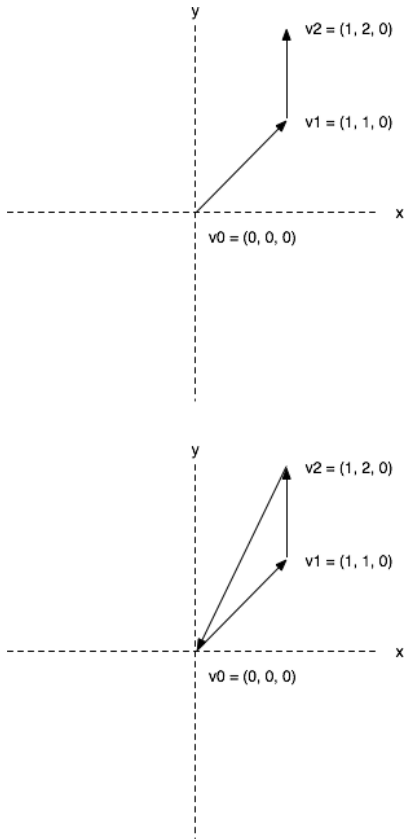
Anschließend wird dann jeweils eine Zufallszahl (*randomFloat()* ist in »CFXMath.h« definiert) für die Rotation und eine für die Rotationsgeschwindigkeit erzeugt.

In der Zeichenroutine wird die Position der Linie ausgelesen und der *glVertex(...)*-Funktion zugewiesen. Den Rest der Methode kennen Sie ja bereits aus dem »Points«-Beispiel.

Line_Strip und Line_Loop

Um verbundene Linien zu zeichnen, nutzt man in der Regel die beiden Primitiv-typen *GL_LINE_STRIP* und *GL_LINE_LOOP*.

GL_LINE_STRIP und *GL_LINE_LOOP* werden im Prinzip fast genauso benutzt wie *GL_LINES*, weshalb ich den Unterschied nur nochmals kurz an einer Grafik verdeutlichen möchte.



GL_LINE_STRIP (oben) und *GL_LINE_LOOP* (unten)

Die 3 Vertices (v_0 , v_1 , v_2) ergeben mit *GL_LINE_STRIP* zwei Linien und mit *GL_LINE_LOOP* ein Dreieck.

Wie man sieht, wird mit *GL_LINE_LOOP* die Geometrie automatisch vom letzten zum ersten Punkt geschlossen.

Mit denen bis jetzt gezeigten Methoden kann man eigentlich schon jede 3D-Geometrie erstellen. Was Ihnen aber vielleicht aufgefallen ist, ist die Tatsache, dass wir bis jetzt keine farbigen Flächen erstellen konnten.

Je nachdem, wie man die Linien anordnet, sieht das später zwar aus wie eine Fläche, ist aber keine.

Was uns direkt zu den »echten« 3D-Primitiven bringt. Den wohl wichtigsten Typ haben wir ja schon kennengelernt.

Dreiecke

Das Dreieck spielt in der 3D-Programmierung eine sehr wichtige Rolle, weil man damit jede beliebige Form erstellen kann. Tatsächlich ist es so, dass Flächen, die **nicht** aus Dreiecken zusammengesetzt wurden, von OpenGL automatisch trianguliert werden.

GRUNDLAGEN

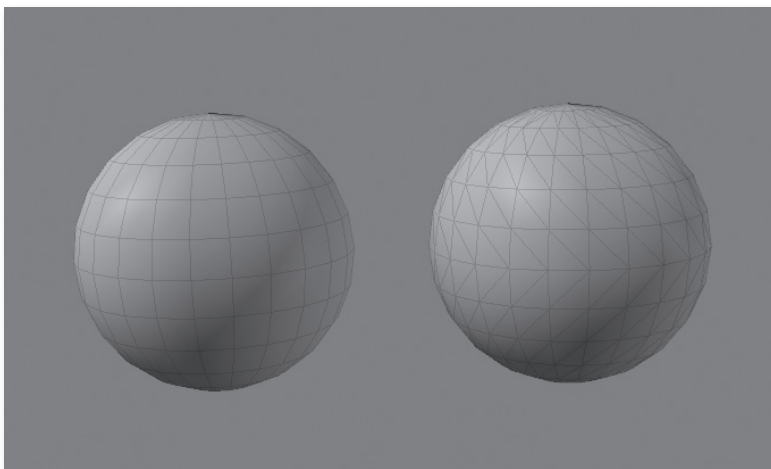
Unter dem Begriff »Triangulation« versteht man das Aufteilen einer Fläche in Dreiecke.

Das Triangulieren ist bei komplexen Objekten sehr rechenintensiv, weshalb man sich folgenden Grundsatz merken sollte:

AUFGEPASST

Wann immer es geht, sollte man ein Objekt aus Dreiecken zusammensetzen, damit OpenGL nicht triangulieren muss.

Nachfolgend eine Abbildung einer Kugel. Auf der linken Seite bevor sie und rechts daneben nachdem sie trianguliert wurde.



Triangulierung einer Kugel

Auch bei der Definition von Dreiecken können Sie beliebig viele Punkte innerhalb eines `glBegin(...)` / `glEnd()`-Blocks definieren.

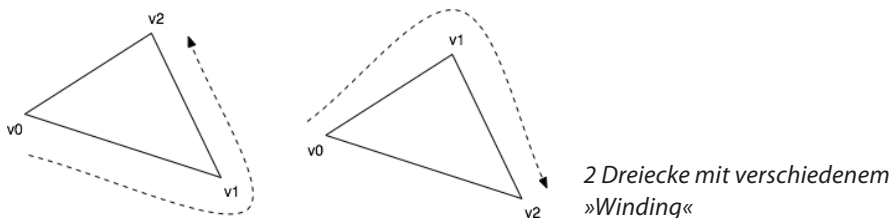
```
glBegin(GL_TRIANGLES);
    glVertex3f(0.0, 0.0, 0.0); //v0
    glVertex3f(1.0, 0.0, 0.0); //v1
    glVertex3f(0.5, 1.0, 0.0); //v2
    glVertex3f(2.0, 0.0, 0.0); //v3
    glVertex3f(3.0, 0.0, 0.0); //v4
    glVertex3f(2.5, 1.5, 0.0); //v5
glEnd();
```

Das obige Beispiel würde 2 Dreiecke zeichnen. Da keine Zeichenfarbe definiert ist, werden die Dreiecke weiß (Standard) gezeichnet **und** gefüllt.

Zeichenrichtung

Ein ganz wichtiger Aspekt beim Erstellen von Polygonen ist, in welcher Richtung man die Vertices angibt. Man spricht dabei vom »Winding«.

Schauen wir uns zunächst folgende Grafik an:



Auf der linken Seite werden die Vertices entgegen dem Uhrzeigersinn (Counter-clockwise Winding, CCW) angegeben, rechts daneben im Uhrzeigersinn (Clockwise, CW).

Wo ist da der Unterschied?

Ein Polygon besteht zunächst einmal aus einer Vorder- und Rückseite. Diese beiden Seiten werden von OpenGL auch unterschiedlich behandelt. Beispielsweise kann man für beide Seiten eines Polygons verschiedene Materialeigenschaften vergeben oder aber eine Seite komplett aus dem Renderprozess ausschließen (Backface-Culling).

OpenGL entscheidet nun anhand des Windings, was bei einer Fläche vorne und was hinten ist. Standardmäßig ist alles, was gegen den Uhrzeigersinn angegeben wurde, vorne und im Uhrzeigersinn entsprechend hinten.

GRUNDLAGEN

Die Zeichenrichtung wird mit dem Befehl

```
glFrontFace(GL_CW);
```

geändert. Das bedeutet, alle Polygone, die CW definiert werden, sind jetzt vorne.

```
glFrontFace(GL_CCW);
```

stellt den Standard wieder her.

Backface-Culling

Wie oben erwähnt, werden mit Hilfe dieser Technik nicht sichtbare Polygone aus dem Renderprozess ausgeschlossen. Zu diesen nicht sichtbaren Polygonen gehören z. B. bei geschlossenen Objekten die Innenseiten oder bei offenen Objekten die Rückseiten der Flächen. Durch diese Vorgehensweise erhöhen wir die Performance unserer Anwendung, da wir ja weniger Polygone zeichnen müssen.

Das Culling können wir über folgende 2 Funktionen steuern:

- `glEnable(GL_CULL_FACE);` aktiviert das Backface-Culling
- `glCullFace(GL_FRONT / GL_BACK);` legt fest, welche Seiten nicht gerendert werden sollen.

POWER

Grundsätzlich gilt:

Wann immer es möglich ist, sollte das Backface-Culling eingeschaltet sein!

Zeichenmodi

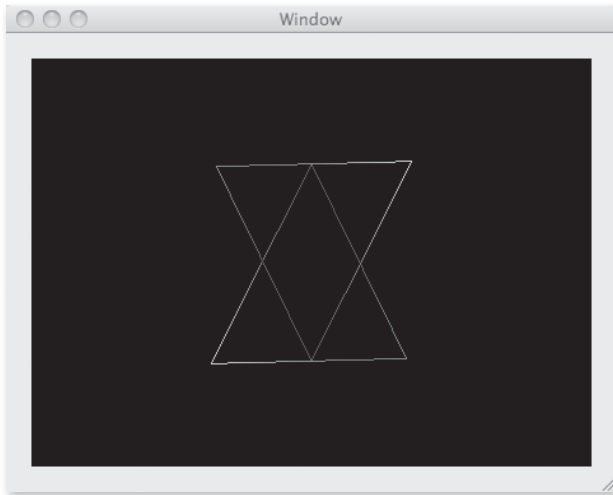
Bevor wir zu unserem nächsten Beispiel kommen, schauen wir uns noch die Zeichenmodi an, die OpenGL kennt.

- `GL_POINT`: Die Eckpunkte des Polygons werden als Punkt ausgegeben (im Prinzip dasselbe wie beim Zeichnen von Punkten).

- `GL_LINE`: Die Polygone werden als Linien ausgegeben.
- `GL_FILL`: Die Polygone werden »gefüllt« ausgegeben (Standard).

Diese Werte können für beide Seiten einer Fläche separat angegeben werden.

Das Testprogramm »Kapitel 3/Polys« rendert 2 Dreiecke (ich verspreche Ihnen, wir werden auch noch was anderes erstellen).



2 Dreiecke, die in verschiedenen Modi angezeigt werden

Werfen wir einen Blick auf den Code:

```
glClear(GL_COLOR_BUFFER_BIT);  
glLoadIdentity();  
glTranslatef(0, -1.0, -5);  
glPushMatrix();  
glRotatef(_rotation, 0.0, 1.0, 0.0);  
glBegin(GL_TRIANGLES);  
glColor3f(1,0,0);  
glVertex3f(-1.0, 0.0, 0.0);  
glColor3f(0,1,0);  
glVertex3f(1.0, 0.0, 0.0);  
glColor3f(0,0,1);  
glVertex3f(0.0, 2.0, 0.0);  
glEnd();  
glPopMatrix();
```

Soweit nichts Neues, es wird lediglich ein Dreieck definiert und mit einem Farbverlauf über alle 3 Vertices gefüllt.

```
glPushMatrix();
glTranslatef(0.0, 1.0, 0.0);
glRotatef(180, 0.0, 0.0, 1.0);
glTranslatef(0.0, -1.0, 0.0);
glRotatef(_rotation, 0.0, 1.0, 0.0);
glBegin(GL_TRIANGLES);
glColor3f(1,0,0);
glVertex3f(-1.0, 0.0, 0.0);
    glColor3f(0,1,0);
    glVertex3f(1.0, 0.0, 0.0);
    glColor3f(0,0,1);
    glVertex3f(0.0, 2.0, 0.0);
glEnd();
glPopMatrix();
```

Das zweite Dreieck sieht schon ein wenig interessanter aus. Wenn man genau hinschaut, sieht man, dass es nicht um den Ursprung definiert wurde, es »sitzt« auf der Y-Achse und erstreckt sich über 2 Einheiten nach oben.

Da es aber um den Ursprung gedreht werden soll, wird es zunächst um eine Einheit nach unten verschoben, dann um 180 Grad auf der Z-Achse gedreht (damit es auf dem Kopf steht) und schließlich wieder um eine Einheit nach oben zurück verschoben.

Anschließend wird eine Rotation um die Y-Achse gemacht, so wie bei dem ersten Dreieck.

Jetzt aber der wirklich wichtige Code:

In dieser Methode wird das *KeyUp-Event* abgefangen, darin werten wir diverse Tasten aus.

Die Taste »1« zeichnet unsere Dreiecke gefüllt, die »2« zeichnet sie als Linien und die »3« als Punkte, so wie oben beschrieben.

Mit der Taste »4« sagen wir, dass die Vorderseite gefüllt und die Rückseite der Dreiecke als Linien dargestellt werden sollen.

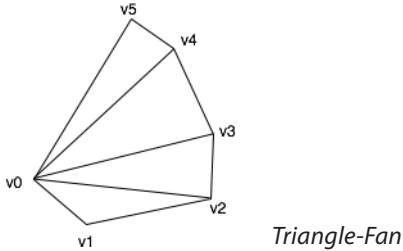
Ganz interessant ist die Leertaste »49«: Hier fragen wir zuerst, ob das Backface-Culling eingeschaltet ist, wenn ja, schalten wir es aus und sehen dann beide Seiten

der Dreiecke, wenn nein, wird es eingeschaltet, und die Seite, die hinten ist, wird nicht gezeichnet.

```
- (void)keyUp:(NSEvent *)theEvent
{
    //NSLog(@"%d", [theEvent keyCode]);
    switch([theEvent keyCode])
    {
        //Polygonemode gefuelllt (Standard)
        case 18://1
        {
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        }break;
        //Polygonemode Linien
        case 19://2
        {
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        }break;
        //Polygonemode Punkte
        case 20://3
        {
            glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
        }break;
        // Vorderseite gefuelllt
        // Rueckseite Linien
        case 21://4
        {
            glPolygonMode(GL_FRONT, GL_FILL);
            glPolygonMode(GL_BACK, GL_LINE);
        }break;
        // Bakfaceculling an/aus
        case 49://Space
        {
            GLboolean c;
            glGetBooleanv(GL_CULL_FACE, &c);
            if(c)
                glDisable(GL_CULL_FACE);
            else
                glEnable(GL_CULL_FACE);
        }break;
    }
}
```

Triangle-Fan

Mit dieser Methode lassen sich fächerähnliche Objekte erstellen.



Die Dreiecke eines Triangle-Fans teilen sich einen Mittelpunkt. Auffallend dabei ist die Anzahl der Vertices: Es werden lediglich 6 Eckpunkte benötigt, um diese Geometrie zu beschreiben. Hätten wir stattdessen »normale« Dreiecke genommen, hätten wir 12 Eckpunkte definieren müssen!

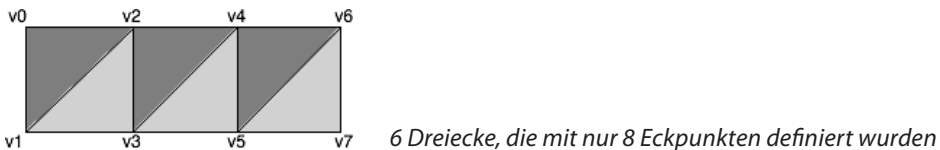
Sie sehen also, wie wichtig die Auswahl des richtigen Primitiv-Typs bei der Erstellung von Objekten ist.

Triangle-Strip

Die wohl schnellste Methode, um Flächen zu rendern, sind Triangle-Strips. Diese haben durch ihre Anordnung 2 Besonderheiten:

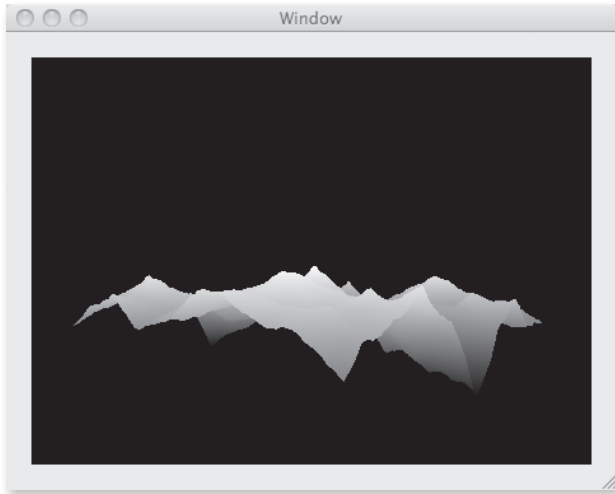
- Dreiecke, die nacheinander definiert wurden, haben eine gemeinsame Kante.
- Dreiecke, die nacheinander definiert wurden, benutzen 2 gemeinsame Eckpunkte (Vertices).

Die folgende Abbildung soll das nochmals verdeutlichen:



Wie man sieht, sind nur 8 Eckpunkte nötig, um die Fläche zu beschreiben. Zu den Triangle-Strips habe ich einmal ein etwas »anspruchsvolleres« Beispiel erstellt, das Sie im Ordner »Kapitel 3/Terrain« finden.

Es sind zwar wieder »nur« Dreiecke, aber wie Sie sehen, kommt es drauf an, was man aus diesen macht.



Ein Terrain, das mit Hilfe von **Triangle-Strips** gerendert wurde

Wie Sie in der Abbildung sehen können, wird mit Hilfe von *Triangle-Strips* ein Terrain gerendert.

Da doch einiges an Code dazugekommen ist, werden wir uns mal anschauen, was da alles passiert.

Wir definieren zuerst eine Struktur, die Informationen für das Terrain aufnimmt. Diese sieht folgendermaßen aus:

```
typedef struct _MapData
{
    unsigned char* data;
    int size;
}MapData;
```

data speichert die Höheninformationen, die aus einem Graustufenbild gelesen werden.

Die Werte in diesem Bild liegen zwischen 0 (schwarz) bis 255 (weiß). Das bedeutet, je heller ein Pixel ist, desto höher wird später das Terrain an dieser Stelle.

size ist die Größe des Bildes, welche durch 2 teilbar sein muss (im Beispiel 128 x 128 Pixel).

Dieses Bild laden wir mit folgender Methode:

```
-(BOOL)loadMap:(NSString*)fileName size:(int)size
{
    FILE* file
    file= fopen( [fileName cStringUsingEncoding:NSUTF8String-
    Encoding], "rb" );
    if( file==NULL )
    {
        return NO;
    }
    _map.data= malloc(size*size * sizeof(unsigned char));
    if(_map.data == NULL)
        return NO;
    fread( _map.data, 1, size*size, file );
    fclose( file );
    _mapSize = size;
    return YES;
}
```

Wie Sie sehen, wird der Dateiname übergeben und mit der C-Funktion *fread(...)* das Bild eingelesen. Die Variable *_map.data* speichert dann die Höheninformationen. Dass ich hier keine Cocoa-Methoden zum Einlesen genommen habe, hat keinen besonderen Grund, es ist lediglich eine Gewohnheit von mir.

Die Methode

```
-(unsigned char)heightAtPosition:(int)x zPosition:(int)z
{
    return ( _map.data[( z*_mapSize )+x] );
}
```

liefert die Höhe (also den Farbwert) innerhalb des Bildes an der angegebenen Position.

```
-(float)scaledHeightAtPosition:(int)x zPosition:(int)z
{
    return ( ( float ) ( _map.data[( z*_mapSize )+x] )*_heightScale );
}
```


Liefert auch die Höhe, allerdings wird diese mit dem Faktor *heightScale* multipliziert, worüber man steuern kann, wie »spitzig« das Terrain wird.

Nun zum wichtigsten Teil des Codes, dem Rendervorgang. Die Verschiebung und Rotation kennen Sie ja schon, weshalb ich sie hier einmal außen vor lasse.

```
if(_loaded)
{
    unsigned char color;
    int z;
    int x;
    for( z=0; z<_mapSize-1; z++ )
    {
        glBegin( GL_TRIANGLE_STRIP );
        for( x=0; x<_mapSize-1; x++ )
        {
            =color=[self heightAtPosition:x zPosition:z];
            glColor3ub( color, color, color );
            float y = [self scaledHeightAtPosition:x zPosition:z];
            glVertex3f( (float)x, y, (float)z );
            color=[self heightAtPosition:x zPosition:z+1];
            glColor3ub( color, color, color );
            y = [self scaledHeightAtPosition:x zPosition:z+1];
            glVertex3f( (float)x, y, (float)z+1 );
        }
        glEnd( );
    }
}
```

Zuerst wird geprüft, ob das Bild geladen wurde (*_loaded*), wenn ja, werden über 2 ineinander verschachtelte Schleifen die Eckpunkte für unsere Triangle-Strips berechnet.

Die Farbe holen wir uns über die oben genannte Methode und setzen sie mit *glColor3ub(...)*.

Die Funktion *glColor3ub(...)* erwartet die Farbwerte zwischen 0 und 255 (*GLubyte*), das bietet sich an, da unser Graustufenbild die Werte in diesem Format gespeichert hat.

Mit der Taste »w« können Sie zwischen den beiden Zeichenmodi *GL_LINE* und *GL_FILL* umschalten.

Mit der Leertaste wird das Backface-Culling ein- bzw. ausgeschaltet.

TIPP

Im Wireframe-Modus (*GL_LINE*) kann man sehr schön die Anordnung der *Triangle Strips* erkennen.

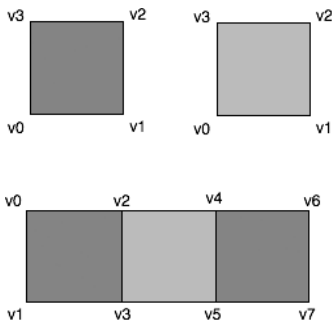
Die beiden Zeilen

```
glEnable( GL_DEPTH_TEST );
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

werden wir später abhandeln. Diese dienen der korrekten Darstellung des Terrains.

Vierecke

Vierecke, die mit *GL_QUADS* oder *GL_QUAD_STRIP* definiert werden, unterscheiden sich nur in der Anzahl der Vertices von den beiden Modi *GL_TRIANGLE* bzw. *GL_TRIANGLE_STRIP*, weshalb ich sie hier nur nochmals an einer Grafik verdeutlichen möchte.



Vierecke mit *GL_QUADS* oben, *GL_QUAD_STRIP* unten

Bei *GL_QUADS* besteht jedes Viereck aus 4 Eckpunkten, die nur zu diesem einen Viereck gehören.

Bei *GL_QUAD_STRIP* haben nacheinander definierte Vierecke eine gemeinsame Kante und zwei gemeinsame Eckpunkte.

Polygone

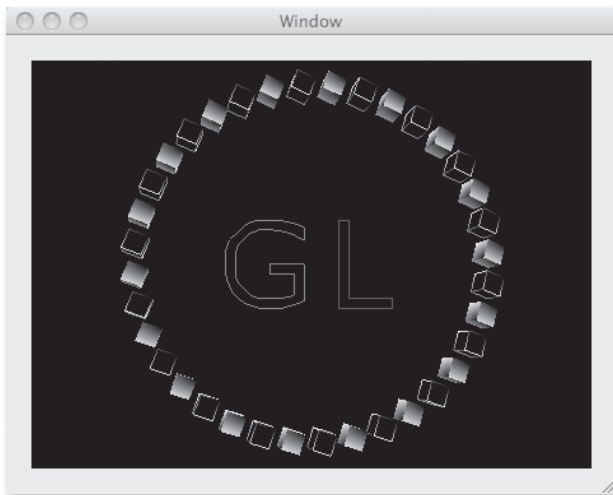
Mit dem Zeichenmodus `GL_POLYGON` lassen sich Flächen mit einer beliebigen Anzahl an Vertices zeichnen. Innerhalb eines `glBegin(...)` / `glEnd()`-Blocks kann immer nur **ein** Polygon definiert werden.

Der letzte Eckpunkt, der definiert wurde, wird automatisch mit dem ersten verbunden.

AUFGEPASST

Wenn Sie mehrere Polygone zeichnen möchten, müssen Sie für jedes einen separaten `glBegin(...)` / `glEnd()`-Block definieren.

Zu den Vierecken und Polygonen habe ich nochmals ein kleines Programm »Kapitel 3/Quads« geschrieben.



Würfel, die mit `GL_QUADS` definiert wurden

Der Code dürfte bis auf die Zeile

```
glEnable( GL_DEPTH_TEST );
```

selbsterklärend sein, weshalb wir uns den Depth-Test nun genauer anschauen werden.

Tiefenpuffer (Z-Buffer, Depth-Buffer)

Anders als der Farbpuffer, der die Farbinformationen der Fragmente (Pixel) speichert, nimmt der Tiefenpuffer deren Entfernung (Z-Koordinate) zum Betrachter auf. Anhand dieser Z-Koordinate entscheidet OpenGL dann, ob ein Fragment gezeichnet werden muss oder nicht, was im Allgemeinen als Depth-Test bezeichnet wird.

Um diesen Depth-Test durchführen zu können, muss man sich zunächst einen Z-Buffer erstellen, der die Tiefeninformationen speichert.

Das funktioniert im Interface-Builder (NSOpenGLView) über den Reiter

Attribute (Depth)

oder aber, wenn man eine Vollbildanwendung schreiben möchte, über den Parameter

`NSOpenGLPFADepthSize, 24`

beim Setzen der Pixel-Attribute.

GRUNDLAGEN

Je höher der Wert des Z-Buffers liegt, desto genauer arbeitet er.

Nachdem der Puffer erstellt wurde, wird mittels

```
glEnable( GL_DEPTH_TEST );
```

der Tiefentest aktiviert.

Danach folgt die Vergleichsfunktion. Anhand des Wertes dieser Funktion wird entschieden, ob ein eingehendes Fragment den Tiefentest besteht oder nicht. Oder einfacher ausgedrückt: Es wird geprüft, ob ein Pixel gezeichnet werden soll oder nicht.

Diese Vergleichsfunktion wird mittels

```
glDepthFunc (GLenum func);
```

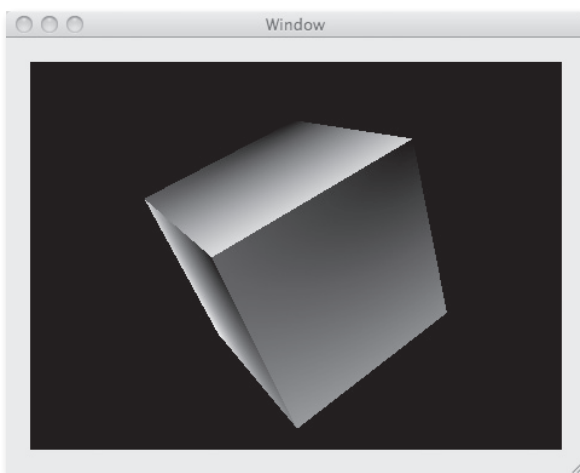
festlegt, wobei folgende Parameter für *func* verwendet werden können:

Parameter	Beschreibung
GL_NEVER	Eingehende Fragmente bestehen niemals den Test.
GL_LESS	Eingehende Fragmente bestehen den Test, wenn sie einen geringeren Tiefenwert haben. (Standard)
GL_EQUAL	Eingehende Fragmente bestehen den Test, wenn sie einen gleich großen Tiefenwert haben.
GL_LEQUAL	Eingehende Fragmente bestehen den Test, wenn sie einen kleineren oder gleich großen Tiefenwert haben.
GL_GREATER	Eingehende Fragmente bestehen den Test, wenn sie einen größeren Tiefenwert haben.
GL_NOTEQUAL	Eingehende Fragmente bestehen den Test, wenn sie einen anderen Tiefenwert haben.
GL_GEQUAL	Eingehende Fragmente bestehen den Test, wenn sie einen größeren oder gleich großen Tiefenwert haben.
GL_ALWAYS	Eingehende Fragmente bestehen immer den Test.

Der Standardwert ist *GL_LESS*. Das bedeutet, dass ein Fragment nur dann gezeichnet wird, wenn seine Z-Koordinate kleiner (also näher am Betrachter) ist als jene, die schon im Z-Buffer steht.

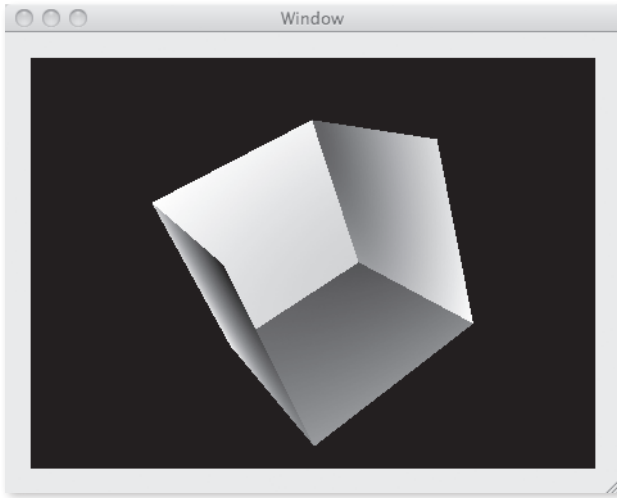
Nachfolgend 2 Abbildungen, welche den Depth-Test »in Aktion« zeigen.

Szene mit aktivem Depth-Test



Korrekt dargestellter Würfel mit aktivem Depth-Test

Gleiche Szene ohne Depth-Test. Wie man sieht, werden die obere und die vordere Fläche einfach »überzeichnet«.



Weil der Depth-Test deaktiviert ist, wird die Szene falsch dargestellt.

Der Z-Buffer wird bei jedem Renderdurchlauf gelöscht, was man in der Regel zusammen mit dem Farbpuffer mittels

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

macht.

Man kann auch beide separat löschen, was aber nicht üblich ist.
Mit der Funktion

```
glDepthMask(GL_FALSE);
```

kann man OpenGL daran hindern, Werte in den Tiefenpuffer zu schreiben, dies ist z. B. bei Partikelsystemen sinnvoll (wie wir in Kapitel 10 sehen werden).

Wie Sie sehen, ist die Verwendung des Depth-Test ziemlich einfach, hier nochmals eine Zusammenfassung:

1. Z-Buffer aktivieren (View oder über Pixelformat-Parameter)
2. Depth-Test einschalten `glEnable( GL_DEPTH_TEST );`
3. Vergleichsfunktion definieren `glDepthFunc (GLenum func);`

```
4. Z-BufferlöschenglClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Zum Schluss noch ein Tipp:

TIPP

Bei einfachen 2D-Anwendungen benötigen Sie natürlich keinen Z-Buffer, da es ja keine Tiefeninformationen gibt, die man speichern muss.

Aber darauf wären Sie bestimmt auch selbst gekommen.

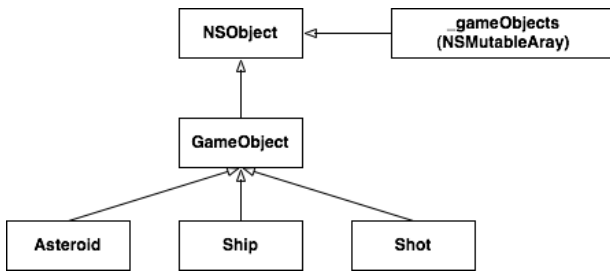
Asteroids

So, nachdem wir nun alle Primitive durchgearbeitet haben, könnte man doch auf die Idee kommen, etwas »Vernünftiges« damit zu machen. Haben Sie Lust auf ein kleines Spielchen? Klar doch, sonst würden Sie das Buch ja nicht lesen.



Kleine Übung zu den OpenGL-Primitiven, ein Asteroid-Clone

Wie man unschwer erkennen kann, habe ich einem kleinen Asteroid-Clone geschrieben, der viel von dem beinhaltet, was wir inzwischen gelernt haben. Ich glaube, den »Inhalt« des Spiels muss man nicht weiter erklären, schauen wir uns deshalb zunächst einmal an, wie das Spiel aufgebaut ist.



Objekte des Asteroids-Clones

An der Grafik kann man erkennen, dass es zunächst 3 Arten von Spielobjekten gibt:

- *Asteroid* sind die »Brocken«, die über den Bildschirm fliegen.
- *Ship* der Spieler
- *Shot* der Schuss des Spielers

Alle 3 leiten sich von *GameObject* ab, welche eine Subklasse von *NSObject* ist. Durch diese Struktur ist es möglich, die Spielobjekte in einem *NSMutableArray* (*_gameObjects*) zu verwalten.

GameObject beinhaltet die Eigenschaften (inkl. Getter und Setter) der Spielobjekte sowie die zwei Methoden

```
-(void)update:(float)deltaTime;
```

(aktualisiert die Position des Objekts und prüft auch gleichzeitig, ob es außerhalb des Bildschirms liegt)

und

```
-(void)render;
```

(zeichnet das Objekt auf den Bildschirm).

Schauen wir uns nun im Detail an, was in der Datei »Asteroids.m« alles passiert:

Zu Anfang wird eine Fullscreen-Anwendung erstellt, diese ist identisch mit der aus dem ersten Kapitel, weshalb wir sie nicht nochmals durchsprechen müssen.

Gehen wir direkt in die *enterMainLoop*-Methode:

```
_gameObjects = [[NSMutableArray alloc] init];
int i;
```



```

for(i=0; i<5; i++)
{
    CFXVector p = makeEmptyVector();
    p.x = randomFloat(-5.0, 5.0);
    p.y = randomFloat(-3.0, 3.0);
    [self createAsteroid:0.8f onPosition:p];
}
_playerShip = [[Ship alloc]init];
[_playerShip setSize:0.2f];
[_gameObjects addObject:_playerShip];

```

Zunächst wird das Array `_gameObjects` erstellt, welches unsere Objekte verwaltet, danach werden fünf Asteroiden erstellt.

Wie Sie sehen, verwende ich auch hier den `CFXVector`, welcher die Position und die Geschwindigkeit für die Objekte speichert.

Danach wird der Spieler erstellt und auch in `_gameObjects` eingefügt.

AUFGEPASST

Da wir den Spieler direkt ansprechen (wegen der Steuerung), darf er, nachdem er in das Array eingehängt wurde, nicht gelöscht werden.

```

_leftArrowPressed = NO;
_rightArrowPressed = NO;
_upArrowPressed = NO;

```

Darüber prüfen wir, ob eine Taste gedrückt wurde.

Durch diese Vorgehensweise erhalten wir eine »unterbrechungsfreie« Abfrage der Tasten.

Würden wir den Code, der ausgeführt werden soll, direkt in das *KeyDown*-Ereignis einbauen, bekämen wir diese »Pause«, die auftritt, wenn man eine Taste gedrückt hält, und das wollen wir natürlich nicht.

Timebased versus Framebased

Weiter geht's in der *while*-Schleife, dort dürfte dieser Block sofort auffallen:

```

//TimeDelta berechnen
_start = [NSDate timeIntervalSinceReferenceDate];

```

```
_deltaTime = _end-_start;  
_deltaTime =-_deltaTime;  
_end = _start;
```

Wofür ist das gut? Hier muss man ein wenig ausholen. Schauen wir uns dazu folgenden Code an:

```
CFXVector position = [object position];  
position.x+=10.0f;  
[object setzeNeuePosition:position];
```

Jedes Mal, wenn das Programm an diese Stelle kommt, wird die X-Position des Objekts um 10 Einheiten erhöht.

Nun ist es ja so, dass nicht alle Computer gleich schnell sind, weshalb ein 3Ghz-Rechner diese Zeile öfter ausführen würde als z. B. ein 1.5Ghz-Rechner.

Machen wir dazu ein kurzes Rechenbeispiel:

Der 3Ghz-Rechner schafft 30FPS (FPS = Frames Per Seconds).

```
position.x+= 30*10.0f;
```

Das sind 300 Pixel in der Sekunde.

Der 1.5Ghz-Rechner bringt es auf 15FPS.

```
position.x+= 15*10.0f;
```

Das ergibt 150 Pixel in der Sekunde.

POWER

Man spricht hier von »Frame-based-Motion«, was so viel bedeutet wie: »die Bewegung der Objekte ist von den erzielten FPS abhängig.«

Das ist natürlich ein Problem, wir wollen nämlich, dass die Position auf allen Rechnern gleichermaßen geändert wird. Die Lösung für dieses Problem ist die »Zeit«.

Wie Sie wissen, ist die Zeitmessung auf jedem Rechner gleich, egal, wie schnell oder langsam er ist. Aus diesem Grund nehmen wir diese in unsere Berechnung mit auf. Dazu holen wir uns bei jedem Renderdurchlauf die Zeit mittels

```
_start = [NSDate timeIntervalSinceReferenceDate];
```

und ermitteln danach, wie lange der Rechner gebraucht hat, um wieder an diese Stelle zu gelangen.

Dann errechnen wir eine Differenz, welche wir in *_deltaTime* speichern.

Genau diese Differenz fließt dann in unsere Berechnung für die Bewegung mit ein:

```
position.x+= 10.0f*_deltaTime;
```

Wir multiplizieren also den Wert mit der verstrichenen Zeit.

Dazu nochmals eine kleine Rechnung:

1.5Ghz-Rechner:

```
_deltaTime = 2.0f  
position.x+= 10.0f*_deltaTime;  
position.x+= 20.0f;
```

Da der 3Ghz-Rechner aber doppelt so schnell ist (theoretisch natürlich), schafft er in der gleichen Zeit 2 Durchläufe, wonach unsere Rechnung dann wieder stimmt.

3.0Ghz-Rechner:

```
_deltaTime = 1.0f  
position.x+= 10.0f*_deltaTime; //1. Durchlauf  
position.x+= 10.0f*_deltaTime; //2.Durchlauf  
position.x+= 20.0f;
```

POWER

Diese Form von Bewegung ist die oben beschriebene »Time-based-Motion«, sie ist heute die wohl am meisten genutzte Methode.

FPS

Im Allgemeinen kann man sagen, dass ein Spiel mit mindestens 24 FPS laufen sollte, damit eine weiche Animation möglich ist. Wir werden später noch einen FPS-Counter bauen, um etwaige Performance- Probleme aufzudecken.

FPS als Verkaufsargument

Glauben wir den Hardwareherstellern, brauchen wir ständig die topaktuellen Grafikkarten (am besten noch im SLI-Verbund), damit wir unsere Lieblingstitel mit 570 FPS anstatt 450 FPS bewundern können.

Interessanterweise folgen immer wieder zig Benutzer den Aufrufen und tauschen ihre 3 Monate alte Grafikkarte gegen das brandneue 1000-€-Model aus.

Zum Glück betrifft dieses Phänomen eher den PC-Markt, die »Obstfreunde« unter den Spielern können sich da noch beruhigt zurücklehnen, wobei ich mir sicher bin, dass sich das in naher Zukunft auch noch ändern wird.

So, nun aber weiter im Code.

Anschließend folgt die Abfrage der Tasten, die dann jeweils in einer *switch*-Anweisung ausgewertet werden. Die für uns wichtigen Ereignisse sind:

- Die ESC-Taste bringt uns aus dem Programm.
- Die Space-Taste feuert einen Schuss ab.
- Die linke / rechte Pfeiltaste rotiert den Spieler.
- Die Pfeil-nach-oben-Taste bewegt den Spieler.

Danach folgt der Block, in dem alle Objekte bewegt und gerendert werden:

```
// Objekte updaten und rendern
NSEnumerator *enumerator = [_gameObjects objectEnumerator];
GameObject *object;
while ((object = [enumerator nextObject]))
{
    [object update:_deltaTime];
    [object render];
}
```

Zum Schluss schauen wir uns noch die letzte Methode an, in der die Kollisionsprüfung aller Objekte stattfindet. Das alles passiert in den zwei ineinander verschachtelten Schleifen.

Wir prüfen zuerst mit

```
if([object isDead])
    continue;
```

ob das Objekt noch »lebt«, also sichtbar ist. Wenn ja, steigen wir in die zweite Schleife ein und prüfen beim zweiten Objekt auch, ob es sichtbar ist:

```
for(z=0; z<[_gameObjects count]; z++)
{
    GameObject * object2 = [_gameObjects objectAtIndex:z];
    if([object2 isDead])
        continue;
```

Danach folgt der wichtigste Teil, die Kollisionsprüfung. Diese schauen wir uns gleich weiter unten genauer an.

Sollte also eine Kollision zwischen 2 Objekten stattgefunden haben, müssen wir zuerst prüfen, um was für Objekte es sich gehandelt hat, was wir mit folgender Zeile tun:

```
if( ([object isKindOfClass:[Shot class]] || [object isKindOfClass:
Class:[Ship class]]) && [object2 isKindOfClass:[Asteroid class]])
{
```

War es eine Kollision zwischen dem Spieler und einem Asteroiden, dann wird

- der Asteroid gelöscht, und es werden 2 neue, kleinere Asteroiden erstellt,
- dem Spieler wird ein Leben abgezogen. Sollte er keine Leben mehr haben, wird er ebenfalls gelöscht, und das Spiel ist zu Ende

```
//AnhandderGroessebestimmenobderAsteroidnochmalsgeteiltwerden soll
// bei einer Groesse von 0.1 ist Schluss
// ansonsten werden 2 kleinere Asteroiden erstellt
if([object2 size]> 0.1f)
{
    int i;
    for(i=0; i<2; i++)
    {
        [selfcreateAsteroid:[object2size]*0.5fonPosition:[object2position]];
    }
}
// Der Spieler darf nicht geloescht werden
if([object isKindOfClass:[Ship class]] == NO)
{
    [object setIsDead:YES];
}
```

```

else
{
    // Leben abziehen
    _playerLeft--;
    if(_playerLeft<=0)
    {
        // Kein Leben mehr da ? Spieler loeschen / Game Over
        [object setIsDead:YES];
        _gameOver = YES;
    }
}

```

Zu guter Letzt wird noch geprüft, welche Objekte »tot« sind, diese werden dann aus dem Array gelöscht;

```

// Tote Objekte loeschen
int i;
for(i=0; i<[_gameObjects count]; i++)
{
    GameObject *o = [_gameObjects objectAtIndex:i];
    if([o isDead])
        [_gameObjects removeObject:o];
}

```

Das war dann schon alles, einen Z-Buffer habe ich natürlich nicht eingebaut, da es ja ein 2D-Game ist.

Was natürlich noch fehlt, ist die Ausgabe von Punkten, Leben usw. Da wir aber noch keine Textausgabe gemacht haben, wollte ich dem nicht vorgreifen.

POWER

Fleißige Leser können ja mal probieren, die Textausgabe mit einfachen Linien zu erstellen.

So, nun aber nochmals zur Kollisionsabfrage, diese ist so wichtig, dass ich sie hier separat behandeln möchte.

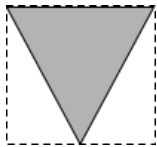
Bounding-Box

Eine der einfachsten Kollisionsabfragen ist die Bounding-Box-Kollisionsabfrage, diese nutzen wir auch in unserem Spiel.

POWER

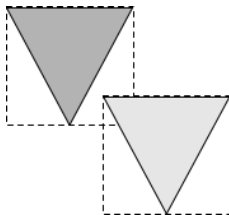
Strenggenommen handelt es sich um eine sogenannte Axis-Aligned-Bounding-Box-Kollisionsabfrage oder kurz AABB.

Die Idee dahinter ist folgende: Man erstellt zunächst um jedes Spielobjekt eine unsichtbare Box, die es vollständig umschließt.



Bounding-Box um ein Spielobjekt

Möchte man nun wissen, ob 2 Objekte miteinander kollidiert sind, prüft man einfach, ob sich die Kanten der beiden Bounding-Boxen überschneiden haben. Das hat den Vorteil, dass dieser Test recht schnell in der Ausführungsgeschwindigkeit und obendrein noch einfach zu implementieren ist.



Kollision von zwei Bounding-Boxen

Wie Sie sehen, haben wir zwar eine Kollision der beiden Bounding-Boxen, aber nicht der eigentlichen Spielobjekte. Wir könnten nun, nachdem wir wissen, dass sich die beiden Bounding-Boxen überschneiden haben, einen weiteren Test machen, der die einzelnen Linien der Dreiecke testet, das würde aber hier zu weit gehen.

TIPP

Tatsächlich ist es in der Praxis oft so, dass zunächst auf eine BBox-Kollision geprüft und anschließend nochmals genauer »nachgeschaut« wird.

Sehr oft sieht man diese kleine Unzulänglichkeit überhaupt nicht, weshalb man sich einen ausführlichen Test sparen kann. Gerade bei schnellen Action-Spielen ist das oft so, man bemerkt es nur, wenn man gezielt darauf achtet.

Schauen wir uns noch den Code zu der BBox-Kollisionsprüfung an:

```
// Bounding Box Kollision
static inline int collide (float left1, float left2, float right1,
float right2, float top1, float top2, float bottom1, float bottom2)
{
    if( left1 > right2 ) return 0;
    if( right1 < left2 ) return 0;
    if( top1 < bottom2 ) return 0;
    if( bottom1 > top2 ) return 0;
    return 1;
}
```

Wir übergeben der Funktion lediglich die Eckpunkte der beiden Bounding-Boxen, danach wird einfach geprüft, ob sich deren Kanten (bzw. Eckpunkte) überschneiden.

Wir werden später im Kapitel 14 noch einmal auf das Thema Kollisionsprüfung zurückkommen, weshalb ich hier das Thema zunächst einmal abschließen möchte.

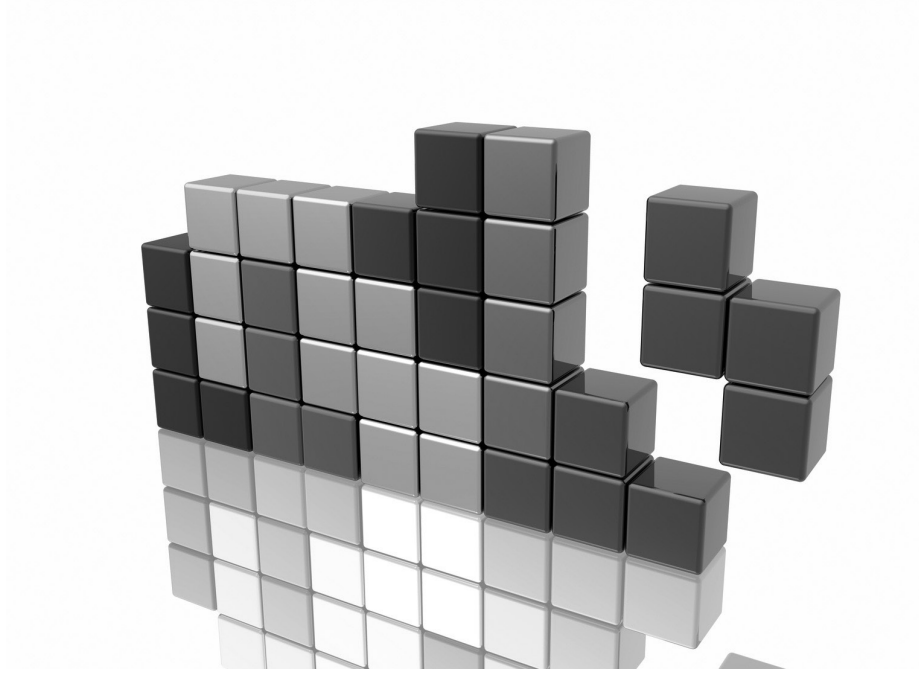
Dieses Kapitel war doch schon recht umfangreich.

Hier haben Sie schon viel über die Funktionsweise von OpenGL gelernt. Sie wissen nun, wie man Primitive zeichnen kann und was ein Depth-Test ist. Des Weiteren haben wir gesehen, wie man mit relativ wenig Mitteln ein kleines Spiel erstellen kann. Im nächsten Kapitel lernen wir etwas über »virtuelle Kameras« und Projektionen.

Zusätzliche Informationen

<http://www.glprogramming.com/red/chapter02.html>

Kapitel aus dem »RedBook ver 1.1«, was viele als das OpenGL-Buch betrachten.



Virtuelle Kameras und Projektionen

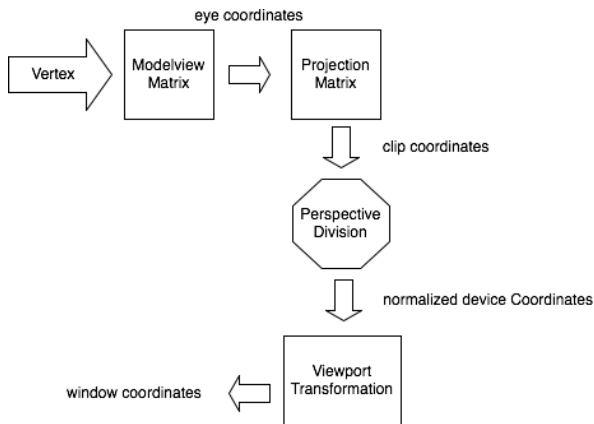
Als wir in den vergangenen Beispielen unsere Objekte zeichnen, haben wir uns noch keinerlei Gedanken darüber gemacht, wie diese letztendlich auf dem Bildschirm ausgegeben werden. Weshalb wir uns in diesem Kapitel einmal näher damit beschäftigen wollen.

Wenn wir eine 3D-Szene rendern, durchlaufen die Vertexdaten mehrere Matrizen. Eine der wichtigsten haben wir bereits kennengelernt: die Modelviewmatrix.

Diese wird, wie wir gelernt haben, immer dann aktiviert, wenn wir Vertexdaten definieren und transformieren möchten.

Dies ist aber nur die halbe Miete bis zur Ausgabe auf dem Bildschirm, tatsächlich passiert nämlich noch jede Menge mehr mit diesen Daten.

Zunächst einmal ein grafischer Überblick darüber, welche Transformationen unsere Daten durchlaufen.



Die Vertex-Transformations-Pipeline

- ModelView bringt unser Objekt an seine Position in der Szene (Eye Coordinates).
- Projection beschreibt, wie und was wir später von unserem Objekt sehen werden (Clip Coordinates). Hier wird die eigentliche Projektion von 3D nach 2D gemacht.
- Viewport nimmt die Anpassung der Ausgabe an die Fensterkoordinaten vor. Weiterhin wird hier das Höhen-Seitenverhältnis eingestellt.

Modelview

Zunächst einmal hat jedes Objekt (Model) sein eigenes Koordinatensystem (Objekt Koordinaten).

Nach der Transformation durch die Modelviewmatrix erhält man dann die sogenannten Eye Coordinates.

Diese heißen so, weil die Objektkoordinaten nun im globalen Koordinatensystem liegen.

Nach dieser Transformation wird der Ursprung dieses Koordinatensystems zum Betrachter (View) gesetzt. Da dieser Vorgang in einem Rutsch geschieht, nennt man sie »Modelviewmatrix«.

Viewport

Bevor wir unsere Daten senden, müssen wir zunächst einen Viewport festlegen.

Dies haben wir in den vergangenen Beispielen folgendermaßen gemacht:

```
CGRect rect = [self bounds];  
glViewport( 0, 0, (GLsizei)rect.size.width, (GLsizei)rect.size.  
height);
```

GRUNDLAGEN

Über die Funktion *glViewport* beschreiben wir die Größe unseres Betrachtungsfensters.

In unserem Fall wäre dies die komplette Größe des *View*.

Projektion

Danach muss eine Projektionsart festgelegt werden.

Dazu aktivieren wir zunächst die Projektionsmatrix und ersetzen sie durch eine Einheitsmatrix.

```
glMatrixMode( GL_PROJECTION );  
glLoadIdentity();
```

Dann erfolgt die Definition des sogenannten Sichtbereichs.

Diesen Sichtbereich kann man sich am besten als eine Art Box vorstellen, durch die wir die Szene betrachten.

OpenGL kennt 2 Arten dieser »Boxen«, nämlich die orthogonale und die perspektivische.

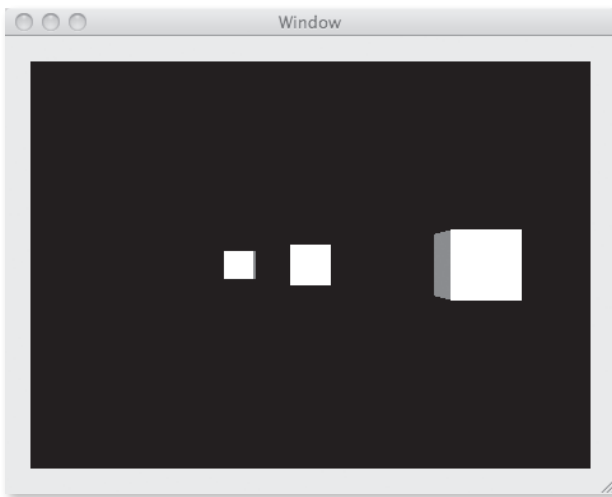
Orthogonale Projektion (Parallelprojektion)

Diese Art von Projektion wird üblicherweise im CAD-Bereich genutzt. Dabei erfolgt keine Größenanpassung der Geometrie entlang der Z-Achse. Das bedeutet, dass Objekte, die im Raum liegen, genauso groß gezeichnet werden wie Objekte, die näher beim Betrachter sind.

TIPP

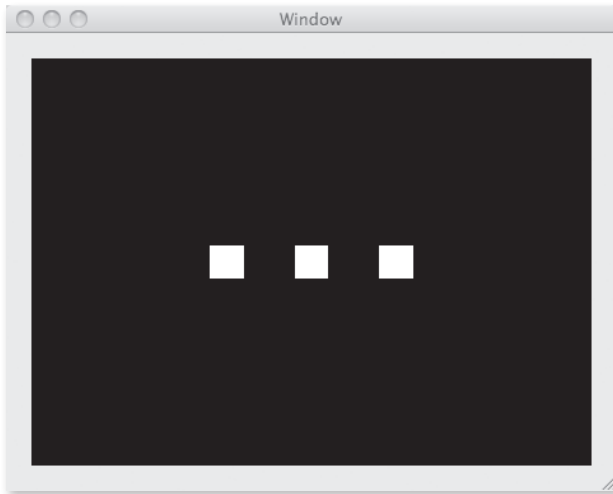
In der Spiele-Entwicklung nutzt man diesen »Ortho-Mode« sehr häufig für die Textausgabe bzw. das Rendern eines HUD (Head Up Display) oder aber für reine 2D-Spiele.

Schauen wir uns dazu folgende zwei Abbildungen an:



Drei Würfel mit einer perspektivischen Projektion

Und hier dasselbe Programm mit einer orthogonalen Projektion:



Dieselben drei Würfel, diesmal mit einer orthogonalen Projektion

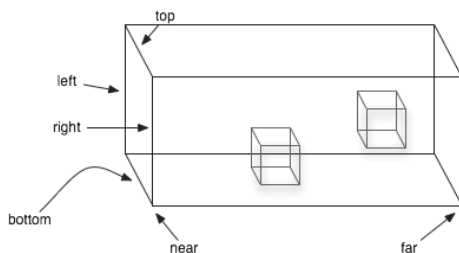
Auf der oberen der beiden Abbildungen kann man erkennen, dass die Objekte auf der Z-Achse verteilt liegen, des Weiteren kann man auch zwei Seitenflächen der Würfel sehen.

Eine orthografische Projektion wird in OpenGL mittels der Funktion

```
glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);
```

gesetzt.

Über die Parameter der Funktion beschreiben wir die Positionen der Schnittflächen (Clip-Planes) der »Box«, durch die wir die Szene betrachten.



Orthogonale Projektion

Der Code könnte dann z. B. so aussehen:

```
glOrtho(-1.0, 1.0, -1.0, 1.0, 5.0, 100.0);
```

Alle Objekte, die sich nun innerhalb dieser »Box« befinden, sind für den Betrachter sichtbar.

GRUNDLAGEN

Der Vollständigkeit halber möchte ich noch auf die Funktion `gluOrtho2D` hinweisen, mit der Sie auch eine orthogonale Projektion erstellen können. Diese Funktion entspricht dem `glOrtho(...)`-Aufruf mit den Parametern `(-1, 1)` für *zNear* und *zFar*.

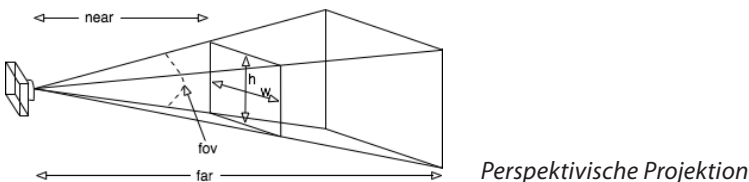
Wie Sie am Präfix erkennen können, handelt es sich um eine Funktion aus der GL-Utility-Bibliothek.

Soweit zur orthogonalen Ansicht. Kommen wir nun zur zweiten Art der Projektion.

Perspektivische Projektion (Zentralprojektion)

Diese Art der Projektion wird bei Spielen wohl am meisten verwendet. Sie verleiht einer Szene bei weitem mehr »Realität« als eine orthogonale Projektion.

Hier wird das Betrachtungssichtfeld (Frustum) nicht wie oben durch eine »Box« definiert, sondern durch einen Pyramidenstumpf.



OpenGL bietet uns hier 2 Möglichkeiten, dieses Frustum zu definieren: einmal über die Funktion

```
glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);
```

und einmal über

```
gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

Wir werden die zweite Variante nutzen, da sie meiner Meinung nach einfacher zu handhaben ist.

Schauen wir uns nun der Reihe nach die Parameter zu *gluPerspective* an.

- **fov** ist der Winkel des Blickfeldes (entlang der Y-Achse) in Grad.
- **aspect** ist das Verhältnis von Höhe zu Breite des Blickfelds (entlang der X-Achse). In der Regel entsprechen die Höhe und Breite der Viewports.
- **near** stellt die Entfernung der nahen Z-Schnittfläche zum Betrachter (positiver Wert) dar.
- **Far** ist die Entfernung der fernen Z-Schnittfläche zum Betrachter (positiver Wert).

Ein Beispiel:

```
gluPerspective( 45.0, rect.size.width / rect.size.height, 1.0,  
70.0 );
```

wäre also ein Sichtfeld mit einem Winkel von 45 Grad, welches bei 1.0 beginnen und bei 70.0 enden würde.

Achtung, die beiden Werte für *near* und *far* stehen nicht für die Z-Koordinate, sondern für den Abstand der beiden Schnittflächen relativ zum Betrachter (0,0,0).

AUFGEPASST

Das Verhältnis von *near* zu *far* sollte nicht größer sein als unbedingt nötig, denn je größer der Abstand, desto ungenauer arbeitet der Tiefenpuffer. Es hat auch keinen Sinn, den Wert für *far* auf 1500.0 zu setzen, wenn sich das Objekt bei z: 150.0 befindet (die verbleibenden 1350 Einheiten wären verschenkter »Platz«). Außerdem sind zu große Werte schlecht für die Performance.

Ich habe auch hierzu wieder ein kleines Beispielprojekt zum Thema erstellt, Sie finden es im Ordner »Kapitel 4/Projections«.

Hier nochmals der wichtigste Teil des Codes:

```
-(void)changeProjection  
{  
    NSRect rect = [self bounds];
```

```
glViewport( 0, 0, (GLsizei)rect.size.width, (GLsizei)rect.size.height);
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
if(_perspective)
{
    gluPerspective( 45.0, rect.size.width / rect.size.height,
        1.0, 50.0 );
}
else
{
    GLfloat range = 6.0f;
    int w = rect.size.width;
    int h = rect.size.height;
    if (w <= h)
        glOrtho (-range, range, -range*h/w, range*h/w, 1.0, 50.0);
    else
        glOrtho (-range*w/h, range*w/h, -range, range, 1.0, 50.0);
}
glMatrixMode( GL_MODELVIEW );
[self setNeedsDisplay:YES];
}
```

Im Programm können Sie über die Leertaste zwischen den beiden Projektionen umschalten. Die Funktion

gluPerspective(...)

kennen Sie nun bereits. Schauen wir uns noch den Code zu *glOrtho(...)* genauer an.

Zunächst wird über *range* eine Variable definiert, welche die Größe für die Werte links, rechts, oben und unten speichert. Danach wird das Bildverhältnis ermittelt und entsprechend in die Berechnungen für diese vier Werte mit eingebaut.

TIPP

Ohne diese Größenanpassung würden die Würfel verzerrt auf dem Bildschirm dargestellt werden.

Zum Schluss wird wieder die Modelviewmatrix aktiviert, und dem View wird mitgeteilt, dass es neu gezeichnet werden soll.

Hier nochmals ein kleines Beispiel, in welchem die Ausgabe immer genau so groß ist, wie das View selbst:

```
- (void)reshape
{
    NSRect rect = [self bounds];
    glViewport( 0, 0, (GLsizei)rect.size.width, (GLsizei)rect.size.
height);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho(0, (GLsizei)rect.size.width, 0, (GLsizei)rect.size.height,
-1, 1);
    glMatrixMode( GL_MODELVIEW );
}
```

Die virtuelle Kamera

So, nun haben wir die beiden Projektionen durchgesprochen, bleibt noch die Kamera.

Stellen wir uns folgendes Szenario vor: Der Spieler steht in einer 3D-Landschaft, in der es jede Menge Bäume, Häuser, »Gegner« und andere Objekte gibt.

Nun stehen wir vor dem Problem, dass wir nicht nur stur in eine Richtung schauen möchten, sondern auch gerne mal das sehen würden, was sich hinter uns befindet. Man könnte das Problem nun lösen, indem man einfach alle Objekte so lange auf der Y-Achse rotiert (sprich, die Transformationen an der Modelviewmatrix vornimmt), bis man sie direkt vor sich hat.

Dies würde auch funktionieren. Bei einigen hundert oder sogar tausend Objekten kann man sich allerdings vorstellen, welchen Rechenaufwand das mit sich brächte.

Hier muss eine andere Lösung her, so eine Art freibewegliche Kamera.

OpenGL bietet selbst keine Möglichkeit, solch eine »Kamera« zu erstellen. Abhilfe schafft wieder einmal die GL-Utility-Bibliothek, welche über die Funktion *gluLookAt(...)* eine »bewegliche Kamera« bereitstellt.

Diese Kamera wird folgendermaßen erstellt:

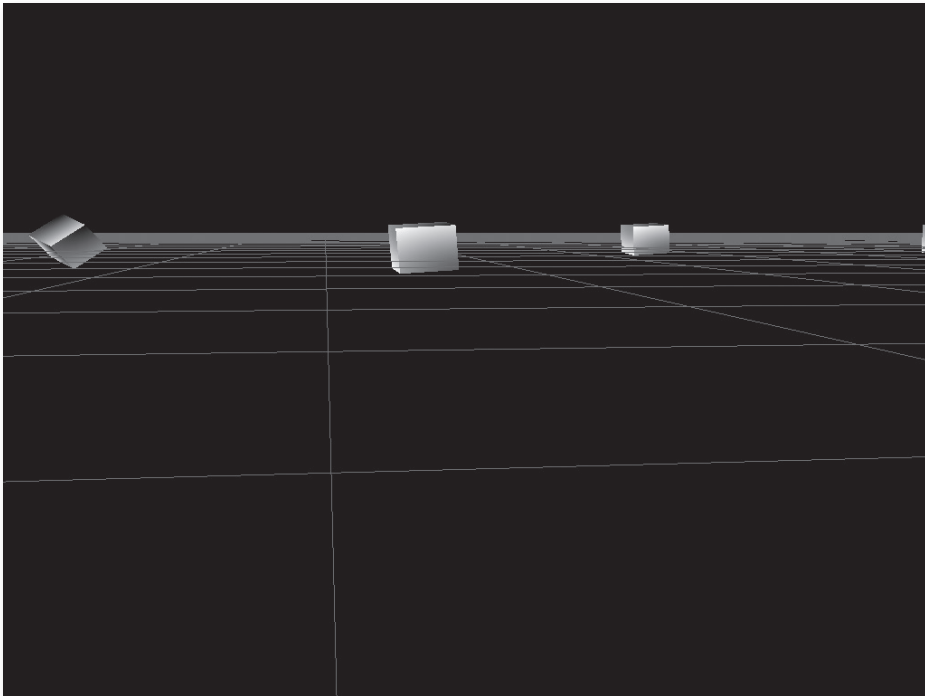
```
gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

Die ersten 3 Werte stehen für die X|Y|Z-Position der Kamera (0.0, 0.0, 10.0).

Danach folgt die Position (X|Y|Z), zu der die Kamera »schauen« soll. In unserem Fall wäre das der Koordinatenursprung bei (0.0, 0.0, 0.0).

Und als Letztes müssen wir der Funktion noch sagen, was »oben« ist. Diese 3 Werte stehen wieder für X|Y|Z. Über den Wert 1.0 sagen wir, dass »oben« entlang der positiven Y-Achse ist.

Da nun all diese Werte variabel sind, kann man mit der *gluLookAt(...)*-Funktion wunderbar eine bewegliche Kamera erstellen, was ich mit dem Projekt »Kapitel 4/Camera« auch gemacht habe.



*Eine freibewegliche Kamera, die über **gluLookAt(...)** realisiert wurde*

Darin wird eine Szene beschrieben, in der sich der »Spieler« mittels Pfeiltasten bewegen kann.

Weiterhin kann man mit der Maus die Blickrichtung ändern, so wie man es aus einem FPS (nein, nicht Frames-Per-Second, sondern First-Person-Shooter) gewohnt ist.

Schauen wir uns die wichtigsten Teile des Codes an:

Zunächst wird eine Vollbildanwendung erstellt so wie in den vorhergehenden Beispielen auch.

Bei Anwendungen, die im Vollbildmodus laufen, wird oftmals der Mauszeiger ausgeblendet. Das erreichen wir über die beiden Funktionen

```
// Mauszeiger verstecken  
CGDisplayHideCursor(kCGDirectMainDisplay);
```

und

```
// Mauszeiger wieder einblenden  
CGDisplayShowCursor(kCGDirectMainDisplay);
```

Da wir dieses Mal einen Tiefentest machen wollen, erstellen wir uns beim Anlegen des Pixel-Formats einen Z-Buffer über

```
NSOpenGLPFADepthSize, 24,
```

In der Methode *enterMainLoop* wird der Depth-Test dann mittels

```
glEnable( GL_DEPTH_TEST );
```

aktiviert. Neu dazugekommen ist noch folgende Zeile:

```
// RefreshSync einschalten  
GLint value = 1;  
[_context setValues:&value forParameter:kCGLCPSwapInterval];
```

Darüber wird unser Render-Output mit der Bildwiederholrate des Monitors synchronisiert. Wenn wir das nicht tun, werden bei einer schnellen Bewegung der Kamera die Objekte verzerrt dargestellt.

Danach erzeugen wir einen Viewport und eine Projektion.

```
glViewport( 0, 0, _screenWidth, _screenHeight);
```

```
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 45.0, _screenWidth/_screenHeight, 1.0, 200.0 );
glMatrixMode( GL_MODELVIEW );
```

Und zum Schluss noch eine Kamera:

```
// Kamera erzeugen
_camera = [[CFXFPSCamera alloc] init];
[_camera positionCamera:makeVector(0.0,4.0, 10.0) lookAt:make-
EmptyVector() up:makeVector(0.0, 1.0, 0.0)];
```

Der Rendercode selbst ist nicht weiter spannend, wichtig darin sind eigentlich nur diese drei Zeilen:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Hiermit löschen wir den Farb- **und** den Tiefenpuffer

```
glLoadIdentity();
```

Einheitsmatrix laden

```
// Kamera updaten
[_camera updateCamera];
```

entspricht der Funktion *gluLookAt(...)*.

AUFGEPASST

Die Funktion *gluLookAt(...)* muss immer ganz zu Anfang des Rendervorgangs aufgerufen werden.

Kommen wir direkt zu dem Event-Handling, in dem wir die Kamera steuern wollen.

Über die Pfeiltasten bewegen wir die Kamera noch vorne, hinten und seitwärts (strafe).

```
if(_leftArrowPressed)
{
    [_camera strafeCamera:-CAMERASPEED*_deltaTime];
}
```

```
if(_rightArrowPressed)
{
    [_camera strafeCamera:CAMERASPEED*_deltaTime];
}
if(_upArrowPressed)
{
    [_camera moveCamera:CAMERASPEED*_deltaTime];
}
if(_downArrowPressed)
{
    [_camera moveCamera:-CAMERASPEED*_deltaTime];
}
```

Mit der Maus können wir wie gesagt die Blickrichtung ändern.

```
if(type == NSMouseMoved)
{
    float deltaX = [event deltaX] / MOUSESENSITIVITY;
    float deltaY = [event deltaY] / MOUSESENSITIVITY;
    CFXVector tempAxis = crossProduct(vectorSubtract([_camera
    lookAt], [_camera position]), [_camera up]);
    tempAxis = normalizeVector(tempAxis);
    // Rotiere auf X/Z-Achse
    [_camera rotateView:-deltaY axis:tempAxis];
    // Rotiere auf Y-Achse
    [_camera rotateView:-deltaX axis:makeVector(0.0, 1.0, 0.0)];
}
```

Die beiden Werte *deltaX* und *deltaY* enthalten die Änderungen der X|Y-Position der Maus.

Wir erstellen einen Vektor, der uns das Kreuzprodukt aus der Blickrichtung und des »up-Vektors« zurückgibt. Wir erinnern uns:

TIPP

Das Kreuzprodukt ist ein Vektor, der senkrecht auf der Fläche steht, die wir über 2 Vektoren definiert haben.

Diesen normalisieren wir dann (wir brauchen nur die Richtung, nicht den Betrag des Vektors) und machen damit anschließend eine Rotation um die XZ-Achse (nach oben und unten schauen).

Zum Schluss erfolgt dann noch eine Rotation um die Y-Achse, die im Prinzip genauso funktioniert, mit dem Unterschied, dass wir als Achse (*axis*) einen festen Y-Wert angeben.

Schauen wir uns nun den Code der Kamera an (CFXFPSCamera). Die Bewegungen der Kamera (vor, zurück, seitwärts) übernehmen die beiden folgenden Methoden.

Beim Seitwärtsbewegen (*strafe*) berechnen wir das Kreuzprodukt aus den beiden Vektoren (Blickrichtung und Position), normalisieren das Ergebnis und addieren es zur Position und zur Blickrichtung hinzu.

Über das Kreuzprodukt wissen wir ja, wie unsere Kamera im 3D-Raum ausgerichtet ist.

Um es sich besser vorstellen zu können, hier ein kleines Beispiel: Strecken Sie Ihre Hand (die soll hier einmal die Kamera sein) so aus, als ob Sie jemanden die Hand schütteln wollen. Stellen Sie sich nun vor, dass aus Ihrem Handrücken eine Linie zeigt, die in dieselbe Richtung zeigt wie Ihr Handrücken. Bei einer Drehung der Hand dreht sich natürlich auch die gedachte Linie, wodurch wir immer genau die Ausrichtung haben, die wir benötigen.

```
-(void)strafeCamera:(float)speed
{
    CFXVector cross = crossProduct(vectorSubtract(_lookAt,
        _position),_up);
    _strafe = normalizeVector(cross);
    _position.x+=_strafe.x*speed;
    _position.z+=_strafe.z*speed;
    _lookAt.x+=_strafe.x*speed;
    _lookAt.z+=_strafe.z*speed;
}
```

Das Bewegen der Kamera funktioniert ähnlich: Wir erstellen einen Vektor aus der Blickrichtung und der Position (dieser zeigt einfach in die Blickrichtung) und normalisieren und addieren ihn auch wieder zur Position und zur Blickrichtung.

```
-(void)moveCamera:(float)speed
{
    CFXVector v = vectorSubtract(_lookAt, _position);
    v = normalizeVector(v);
    _position.x+=v.x*speed;
    _position.z+=v.z*speed;
```

```

    _lookAt.x+=v.x*speed;
    _lookAt.z+=v.z*speed;
}

```

Die Rotation sieht ein wenig wild aus. Hierbei handelt es sich um eine sogenannte Axis-Angle-Rotation. Diese rotiert einen Punkt (Blickrichtung der Kamera) um eine beliebige Position (Position der Kamera) im 3D-Raum.

```

- (void)rotateView:(float)angle axis:(CFXVector)axis
{
    // Alte Blickrichtung
    CFXVector v = vectorSubtract(_lookAt, _position);
    axis = normalizeVector(axis);
    // Neue X-Position
    float newX = (cos(angle) + (1 - cos(angle)) * axis.x * axis.x)
    * v.x;
    newX +=((1 - cos(angle)) * axis.x * axis.y - axis.z *
    sin(angle)) * v.y;
    newX +=((1 - cos(angle)) * axis.x * axis.z + axis.y *
    sin(angle)) * v.z;
    // Neue Y-Position
    float newY = ((1- cos(angle)) * axis.x*axis.y + axis.z *
    sin(angle)) * v.x;
    newY +=( cos(angle) + (1 - cos(angle)) * axis.y * axis.y) *
    v.y;
    newY +=((1 - cos(angle)) * axis.y * axis.z - axis.x *
    sin(angle)) * v.z;
    // Neue Z-Position
    float newZ = ((1-cos(angle)) * axis.x * axis.z -axis.y *
    sin(angle)) * v.x;
    newZ +=((1 - cos(angle)) * axis.y * axis.z + axis.x *
    sin(angle)) * v.y;
    newZ +=(cos(angle) + (1 - cos(angle)) * axis.z * axis.z)
    * v.z;
    CFXVector newView;
    newView = makeVector(newX, newY, newZ);
    // Neue Blickrichtung
    _lookAt = vectorAdd(_position, newView);
}

```

Die Berechnung der Rotation basiert auf der Beschreibung von Paul Bourke: (<http://local.wasp.uwa.edu.au/~pbourke/geometry/rotate/>), welche auch Beispiel-Codes beinhaltet.

Rotationen und der Gimbal Lock

Passend zum Thema möchte ich das Kapitel mit einem Tipp abschließen:

Wenn Sie den nächsten Teil von Descent oder einem ähnlichen Spiel machen möchten, in dem man die Kamera um 360 Grad in alle Richtungen drehen kann, werden Sie früher oder später mit dem sogenannten Gimbal Lock Bekanntschaft machen.

Dieser tritt auf, wenn Sie eine Rotation mit Hilfe von Eulerwinkeln um mehrere Achsen gleichzeitig durchführen wollen.

Das Problem dabei ist, dass unter Umständen 2 Achsen dieselbe Drehung machen und danach in die gleiche Richtung zeigen. Dadurch »fehlt« dem Programm eine Achse, wodurch es zu falschen Ergebnissen bei der Rotation kommt.

Abhilfe für dieses Problem schaffen sogenannte Quaternionen (komplexe Zahlen), die aber nicht Inhalt dieses Buches sind, weshalb ich hier auf das Internet verweisen möchte, wo Sie jede Menge Informationen dazu finden.

Sie sollten es sich vielleicht im Hinterkopf behalten, dass, wenn sich ein Objekt beim Rotieren »seltsam« verhält, es am Gimbal Lock liegen kann.

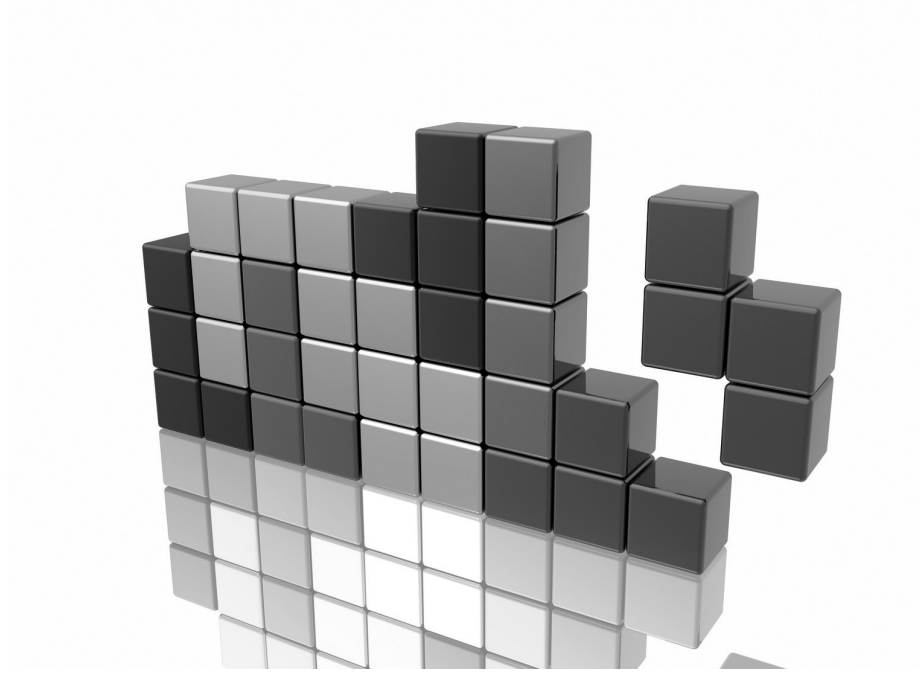
Zusätzliche Informationen

<http://www.xmission.com/~nate/tutors.html>

Das Projekt »The projection« von Nate Robins, in welchem man zur Laufzeit z. B. zwischen den verschiedenen Projektionen umschalten kann

http://de.wikipedia.org/wiki/Gimbal_Lock
Gimbal Lock

<http://de.wikipedia.org/wiki/Quaternion>
Quaternion



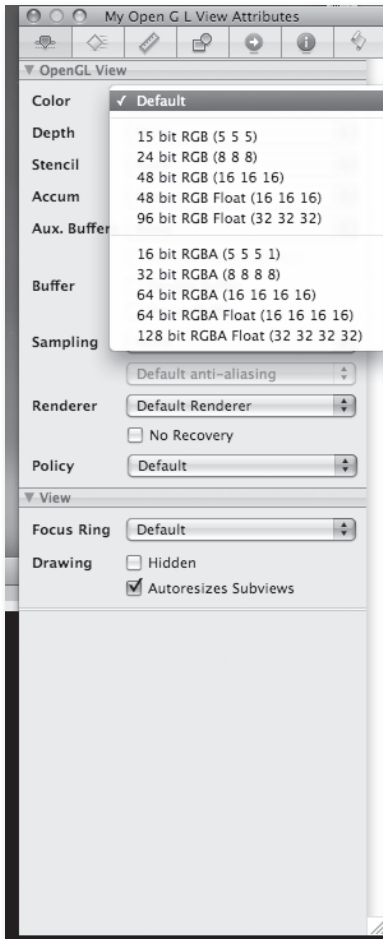
Farben, Materialien und das Licht

Farben, Materialien und das Licht

Wie man Farben in OpenGL definiert, haben wir bereits gelernt. Schauen wir uns nun die Farbgebung einmal genauer an.

Wenn wir eine OpenGL-Anwendung erstellen, müssen wir die Farbtiefe des Color-Buffers angeben.

Für ein NSOpenGLView geht das bequem über den Interface-Builder.



Einstellen der Farbtiefe für unser OpenGLView

Und bei einer Vollbild-Anwendung über den Parameter

`NSOpenGLPFAColorSize, 24,`

beim Erstellen des Pixel-Formats.

Üblicherweise wählt man ein Format, das über 16-Bit liegt. OpenGL unterstützt zwar auch einen sogenannten Index-Mode (256 Farben), dieser ist aber längst überholt.

Bei den Formaten 24-Bit und 32-Bit wird man kaum einen Unterschied sehen, es gibt aber trotzdem 2 Gründe, weshalb man ein 32-Bit-Format wählen sollte.

- Die Speicher-Architektur arbeitet schneller mit 4 anstatt 3 Bits.
- Die zusätzlichen 8 Bit können für einen Alpha-Kanal genutzt werden.

Farben verwenden

Die Farben werden in OpenGL separat für Rot, Grün und Blau angegeben, die man üblicherweise über die Intensität steuert.

Die Werte dafür reichen dann von 0.0 (keine Intensität) bis 1.0 (volle Intensität), womit sich der komplette RGB-Farbraum darstellen lässt.

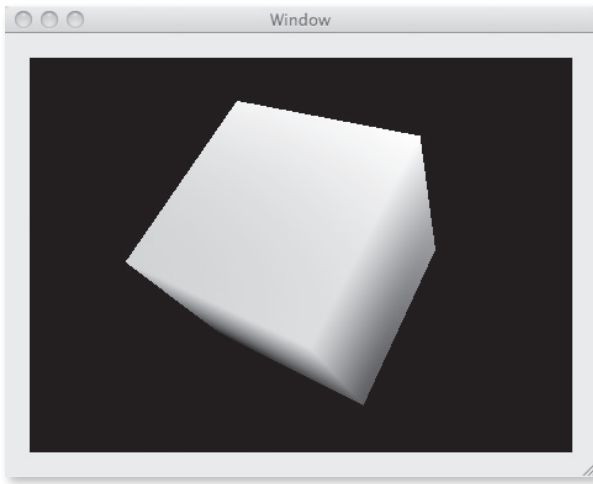
TIPP

Wenn Ihnen das Umrechnen des Farbwertes zu umständlich ist, können Sie eine Farbe auch direkt als RGB-Wert angeben, was dann so aussehen würde:

```
glColor3ub(0, 255, 129) = RGB (0, 128, 129)
```

OpenGL rechnet diese Angaben dann in Float-Werte um. Wie man sich denken kann, kostet das natürlich Zeit, weshalb man besser die »Version« mit den Intensitäten (0.0-1.0) verwenden sollte.

Ich habe zum Thema Farben ein Testprogramm »Kapitel 5/ColorCube« erstellt, welches einen Würfel zeigt, der den kompletten RGB-Farbraum enthält.



Der komplette RGB-Farbraum in einem Würfel

Darin bekommt jeder Eckpunkt einen anderen Farbewert. OpenGL interpoliert dann zwischen diesen Werten, womit wir einen weichen Farbverlauf bekommen.

Um mal eine andere Möglichkeit der Farbdefinition zu zeigen, habe ich bei diesem Beispiel die Farben nicht wie üblich über 3 Float-Werte angegeben, sondern über ein Array.

```
glBegin(GL_QUADS);  
    glColor3fv(colors[1]);glVertex3fv(vertices[1]);  
    glColor3fv(colors[5]);glVertex3fv(vertices[5]);  
    glColor3fv(colors[7]);glVertex3fv(vertices[7]);  
    glColor3fv(colors[3]);glVertex3fv(vertices[3]);
```

Shading

OpenGL kennt 2 Methoden, Farben für Flächen zu berechnen: Die eine nennt sich Smooth-Shading (welche wir eben gesehen haben) und die andere Flat-Shading.

Smooth-Shading

Das Smooth-Shading erzeugt einen Farbverlauf zwischen 2 Farben. Man gibt dabei pro Eckpunkt je eine Farbe an. Zwischen den beiden Eckpunkten wird dann die Farbe durch eine lineare Interpolation bestimmt, was natürlich einiges an Rechenzeit kostet, aber entsprechend schön aussieht.

Flat-Shading

Beim Flat-Shading wird die Fläche mit der Farbe gefüllt, welche für den **letzten Eckpunkt** angegeben wurde. Alle anderen Farbinformationen werden einfach ignoriert.

Einzige Ausnahme ist *GL_POLYGON*, hier wird die Farbe genommen, die für den **ersten Eckpunkt** definiert wurde.

Das Beispiel

```
glBegin(GL_QUADS);  
    glColor3f(1, 0, 0); glVertex3f(0, 0, 0);  
    glColor3f(0, 1, 0); glVertex3f(1, 0, 0);  
    glColor3f(0, 0, 1); glVertex3f(1, 1, 0);  
    glColor3f(1, 1, 1); glVertex3f(0, 1, 0);  
glEnd;
```

würde ein weißes Quadrat zeichnen, da die letzte Farbdefinition *glColor3f(1, 1, 1);* war.

Das Shading wird über den Befehl

```
glShadeModel (GLenum mode);
```

gesteuert. Wobei *mode* entweder *GL_SMOOTH* (Standard) oder *GL_FLAT* sein kann.

Licht

Bis jetzt wirkten die Szenen, welche wir entworfen haben, sehr »unrealistisch«, was daran lag, dass wir unsere Objekte mit einfachen Farben definiert und sie dann ohne jegliche Beleuchtung gerendert haben.

Eine Szene wirkt aber weitaus realistischer, wenn die Objekte, die darin enthalten sind, beleuchtet werden.

Schon alleine über das Licht kann man einer Szene viel mehr Atmosphäre verleihen. Stellen wir uns nur einmal einen düsteren 3D-Shooter vor, in dem es nicht diese dunklen Räume geben würde, der Gruselfaktor würde gegen Null gehen.

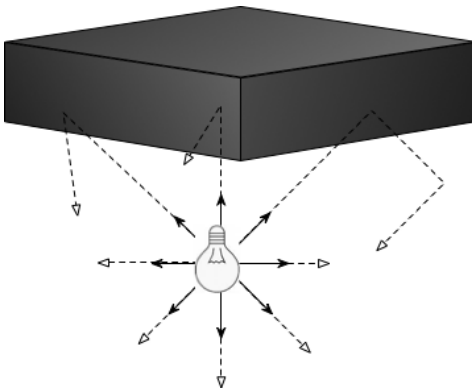
Die verschiedenen Lichtarten

Bevor wir uns an die Beleuchtung machen, müssen wir einige theoretische Dinge darüber wissen.

Schauen wir uns dazu zuerst einmal an, welche unterschiedlichen Lichtarten es gibt.

Ambientes Licht (Umgebungslicht)

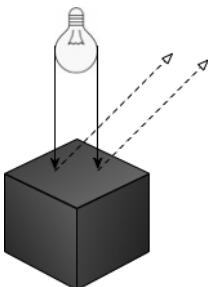
Ein »Ambientes Licht« kommt aus keiner bestimmten Richtung. Dadurch kann man seine Position auch nicht erkennen. Es entsteht dadurch, dass es mehrfach an Flächen reflektiert wird.



Ambientes Licht

Diffuses Licht

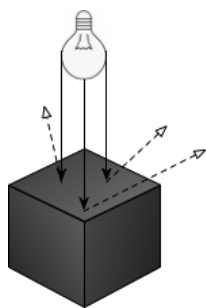
Beim »Diffusen Licht« kann man erkennen, aus welcher Richtung es kommt. Dabei werden Flächen, die sich näher an der Lichtquelle befinden, heller beleuchtet. Je weiter man eine Fläche von der Quelle entfernt, umso dunkler wird sie.



Diffuses Licht

Glanzlicht (Specular)

Das »Glanzlicht« ist auch ein Richtungslicht, mit in einer Richtung gehenden starken Reflektion. Dieses Glanzlicht kann man sehr oft auf metallischen Gegenständen erkennen (z. B. Chrome). Es eignet sich gut dazu, glänzende Oberflächen zu erzeugen.



Glanzlicht (bzw. Specular)

Nachfolgend eine Tabelle, in der alle Lichteigenschaften von OpenGL aufgelistet sind.

Eigenschaft	Wert	Beschreibung
GL_AMBIENT	RGBA	ambienter Anteil des Lichts
GL_DIFFUSE	RGBA	diffuser Anteil des Lichts
GL_SPECULAR	RGBA	Glanzanteil des Lichts
GL_POSITION		Position des Lichts
GL_SPOT_DIRECTION		Richtung des Spotlights
GL_SPOT_EXPONENT		Spotlight-Exponent
GL_SPOT_CUTOFF		Spotlight Sperrwinkel
GL_xxx_ATTENUATION		Abschwächung des Lichts mit der Entfernung

Lichtmodel

Im Gegensatz zu den individuellen Eigenschaften, die man pro Lichtquelle definieren kann, bietet OpenGL sogenannte globale, für alle Lichtquellen gemeinsam geltende Einstellungen.

Diese sind:

- Globales ambientes Licht
- Betrachter-Position in der Szene oder unendlich weit entfernt
- Lichtberechnung für Vorder- und Rückseite der Polygone
- Separate Berechnung des Glanzanteils

Diese Eigenschaften werden über die Funktion

```
glLightModelf( GLenum pname, GLfloat param )
```

gesteuert, dabei ist der erste Parameter die Eigenschaft des Lichtmodels und der zweite der Wert, welchen die Eigenschaft haben soll.

Globales Licht

GL_LIGHT_MODEL_AMBIENT: setzt ein von definierten Lichtquellen unabhängiges Umgebungslicht. Damit lässt sich der Szene eine gewisse Grundhelligkeit geben. Dies ist zum Beispiel dann sinnvoll, wenn die Ausleuchtung der eingeschalteten Lichtquellen nicht ausreicht.

Hier ein Beispiel, welches die Szene mit einem leichten Blau beleuchtet:

```
float blue [] = {0.0, 0.0, 0.3, 1.0};  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, blue);
```

Betrachter-Position

GL_LIGHT_MODEL_LOCAL_VIEWER: Hierüber kann man definieren, ob die Reflexionen auf Oberflächen aus der Sicht eines lokalen (local) oder nicht lokalen Betrachters durchgeführt werden sollen.

Ein Beispiel, bei dem die Reflexion von einem nicht lokalen Betrachter berechnet werden soll:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 0);
```

Beidseitige Beleuchtung der Polygone

GL_LIGHT_MODEL_TWO_SIDED: Ermöglicht es OpenGL anzuweisen, dass Vorder- und Rückseiten der Polygone beleuchtet werden sollen. In Verbindung mit den Materialien (davon weiter unten mehr) ist es möglich, für beide Seiten eines Polygons verschiedene Beleuchtungseffekte zu erzielen. Der Aufwand der Berechnung ist natürlich höher, als wenn nur eine Seite beleuchtet werden soll.

Separate Berechnung des Glanzanteils

GL_LIGHT_MODEL_COLOR_CONTROL: Normalerweise wird die im Licht berechnete Farbe mit der Textur (dazu später mehr) multipliziert, wodurch der Glanzanteil verloren geht.

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL,  
GL_SEPARATE_SPECULAR_COLOR);
```


sorgt nun dafür, dass der Glanzanteil addiert statt multipliziert wird, was den Effekt behebt. Wir werden erst im späteren Verlauf sehen, wie das funktioniert, da wir im Moment noch keine Texturen behandelt haben.

Licht-Abschwächung

Ein Licht leuchtet natürlich nicht unendlich weit, sondern wird irgendwann immer schwächer. Um diesen Effekt nachzubilden, bietet OpenGL einen sogenannten Attenuation-Faktor. Dieser Faktor wird mit dem Diffusen, Ambienten und dem Glanzanteil der Farbe multipliziert.

Die Lichtabschwächung wird über folgende 3 Parameter gesteuert:

Parameter	Standardwert
GL_CONSTANT_ATTENUATION	1.0
GL_LINEAR_ATTENUATION	0.0
GL_QUADRATIC_ATTENUATION	0.0

Die Formel lautet folgendermaßen, wobei »d« die Distanz zwischen Vertex und der Lichtquelle ist:

$$\text{Attenuation-Factor} = 1 / (\text{GL\_CONSTANT\_ATTENUATION} + \text{GL\_LINEAR\_ATTENUATION} * d + \text{GL\_QUADRATIC\_ATTENUATION} * d * d);$$

Hier ein Beispielaufruf:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 1.0);  
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.02);  
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.008);
```

Materialien

Nicht weniger wichtig als die verschiedenen Lichtarten sind in der 3D-Programmierung die Materialien, aus denen unsere Objekte »bestehen«.

Wenn wir einen Gegenstand beschreiben müssten, würde früher oder später der Satz fallen: »Er ist aus diesem oder jenem Material.«

Anhand der Beschreibung des Materials könnten wir uns ein Bild davon machen, wie dieser Gegenstand aussehen würde, ohne dass wir ihn tatsächlich sehen müssten. Alleine das Wort »Chrome« sagt uns schon, das etwas sehr stark glänzt. Genau

so wie man ein Material in der realen Welt beschreiben würde, tut man dies auch in OpenGL.

Materialien bestehen aus Eigenschaften, wir können ihnen z. B. einen ambienten Anteil zuweisen, der aussagt, wie stark dieser das Licht reflektieren soll.

Wenn wir z. B. ein Material mit folgenden Parameter definieren

```
float ambient[] = {1.0, 0.0, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
```

dann reflektiert es den roten Ambiente-Anteil des Lichts, da die anderen beiden Werte für grün und blau auf Null gesetzt sind.

Die folgende Tabelle soll einmal verdeutlichen, welche verschiedenen Materialeigenschaften es in OpenGL gibt.

Eigenschaft	Wert	Beschreibung
GL_AMBIENT	RGBA	Reflexionsgrad vom ambienten Anteil des Lichts
GL_DIFFUSE	RGBA	Reflexionsgrad vom diffusen Anteil des Lichts
GL_SPECULAR	RGBA	Reflexionsgrad vom Glanzanteil des Lichts
GL_AMBIENT_AND_DIFFUSE	RGBA	Kombination aus den beiden
GL_EMISSION	RGBA	Emission des Materials (passive Strahlereigenschaft)
GL_SHININESS	0.0 – 128.0	Glanz-Exponent

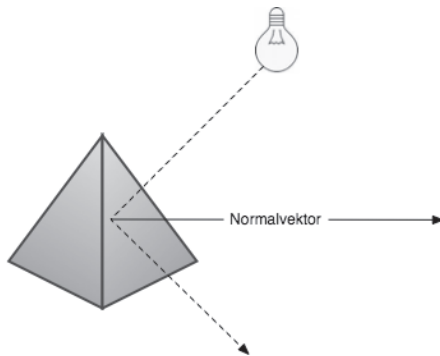
Eine Besonderheit ist die Eigenschaft *GL_EMISSION*, diese kann man sich wie eine eigenständige Lichtquelle vorstellen, die aber **kein** Licht auf andere Objekte einer Szene wirft. Sie »leuchtet« auch bei komplett dunklen Szenen noch.

Normale

Im Kapitel über Vektoren hatte ich schon kurz angedeutet, dass man bei der Beleuchtung der Objekte einen Normal-Vektor benötigt, der OpenGL mitteilt, wie die Fläche ausgerichtet ist.

Anhand dieses Vektors und der Position der Lichtquelle wird die Farbe des Lichts berechnet.

Hier nochmals eine Abbildung, welche die Thematik veranschaulicht.



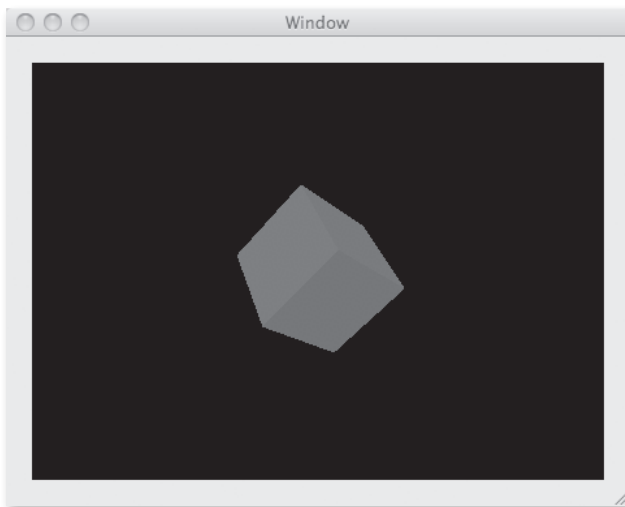
Flächennormale zur Berechnung der Lichtfarbe

TIPP

Hier nochmals der Hinweis: Der Normal-Vektor muss eine Länge von 1 haben (Einheitsvektor), damit die Beleuchtung korrekt funktioniert. OpenGL bietet zwar die Möglichkeit, über die Funktion `glEnable(GL_NORMALIZE)` die Normale automatisch zu normalisieren, was sich aber negativ auf die Performance niederschlägt und deshalb nicht verwendet werden sollte.

Nach der ganzen Theorie wird es nun endlich Zeit, das Licht einmal einzuschalten.

Beginnen wir wieder mit einem recht einfachen Testprogramm »Kapitel 5/ Simple Light«.



Würfel, der beleuchtet dargestellt wird

Wir definieren zuerst einmal die Materialeigenschaften für unseren Würfel und das Licht.

TIPP

Die Reihenfolge, in der Sie das Material und die Lichteigenschaften festlegen, spielt keine Rolle.

Das Licht soll in diesem Beispiel die gleichen Parameter haben, wie das Material.

Dies ist kein Muss, man kann selbstverständlich unterschiedliche Werte festlegen.

Ein mittleres Grau für Ambiente:

```
float ambient[] = {0.5, 0.5, 0.5, 1.0};
```

und Weiß für den diffusen Anteil:

```
float diffuse[] = {1.0, 1.0, 1.0, 1.0};
```

Danach legen wir diese beiden Eigenschaften an:

```
// Materialeigenschaften  
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);  
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
```

Der erste Parameter bestimmt, für welche Seite der Flächen wir das Material definieren. Wie wir im Kapitel 3 erfahren haben, ist es möglich, für beide Seiten ein anderes Material zu vergeben.

GRUNDLAGEN

Um für die Vorderseite der Fläche ein Material zu vergeben, benutzt man den Parameter

```
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient1);
```

und für die Rückseite entsprechend:

```
glMaterialfv(GL_BACK, GL_AMBIENT, ambient2);
```

Wir möchten aber in diesem Beispiel für beide Seiten das gleiche Material.

Als Nächstes kommt die Definition des Lichts an die Reihe:

Wir legen die Position der Lichtquelle fest und stellen die Werte für ambienten und diffusen Anteil ein.

```
float position[] = {-1.0, 2.0, 0.0, 0.0};
// Lichteigenschaften
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

Über den ersten Parameter sagen wir OpenGL, für welches Licht wir die Werte festlegen möchten.

GRUNDLAGEN

In OpenGL haben wir 8 Lichtquellen, die wir separat behandeln können.

Diese reichen von *GL_LIGHT0* bis zu *GL_LIGHT7*.

Danach schalten wir die erste Lichtquelle an.

```
glEnable(GL_LIGHT0);
```

Zum Schluss aktivieren wir die globale Beleuchtung.

```
glEnable(GL_LIGHTING);
```

Man kann sich das so vorstellen: In einer Wohnung gibt es acht Lichter, die man natürlich unabhängig voneinander ein- und ausschalten kann. Der Schalter *GL_LIGHTING* wäre dann der »Hauptschalter«, mit dem man alle Lichtquellen auf einmal an- bzw. ausschalten kann.

Kommen wir zum Rendercode:

```
glBegin(GL_QUADS);
// Oben
glNormal3f(0.0, 1.0, 0.0);
glVertex3f( 1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);
// Unten
```

```
glNormal3f(0.0, -1.0, 0.0);
glVertex3f( 1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f( 1.0f, -1.0f, -1.0f);
// Vorne
glNormal3f(0.0, 0.0, 1.0);
glVertex3f( 1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);
// Hinten
glNormal3f(0.0, 0.0, -1.0);
glVertex3f( 1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f( 1.0f, 1.0f, -1.0f);
// Links
glNormal3f(-1.0, 0.0, 0.0);
glVertex3f(-1.0f, 1.0f, 1.0f);
glVertex3f(-1.0f, 1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, -1.0f);
glVertex3f(-1.0f, -1.0f, 1.0f);
// Rechts
glNormal3f(1.0, 0.0, 0.0);
glVertex3f( 1.0f, 1.0f, -1.0f);
glVertex3f( 1.0f, 1.0f, 1.0f);
glVertex3f( 1.0f, -1.0f, 1.0f);
glVertex3f( 1.0f, -1.0f, -1.0f);
glEnd();
```

Es wird ein Würfel erstellt, der dieses Mal aber 2 Besonderheiten aufweist.

- Es wurde keine Farbe angegeben. Die Farbgebung erledigt nun das Material.
- Es wurde pro Seite ein Normalvektor erstellt.

Der Normalvektor wird über die Funktion *glNormal()* festgelegt. Da es sich um einen simplen Würfel handelt, schaffen wir die Definition dieses Vektors auch ohne Berechnung, er zeigt lediglich in die Richtung, die unsere Seite darstellen soll.

AUFGEPASST

Wenn Sie über den Befehl *glScale* ein Objekt skalieren, werden auch seine Normale mitskaliert, was, wie wir nun wissen, nicht passieren darf, da sonst die Länge der Normale nicht mehr stimmt.

Abhilfe schafft hier wieder *glEnable(GL_NORMALIZE)*.

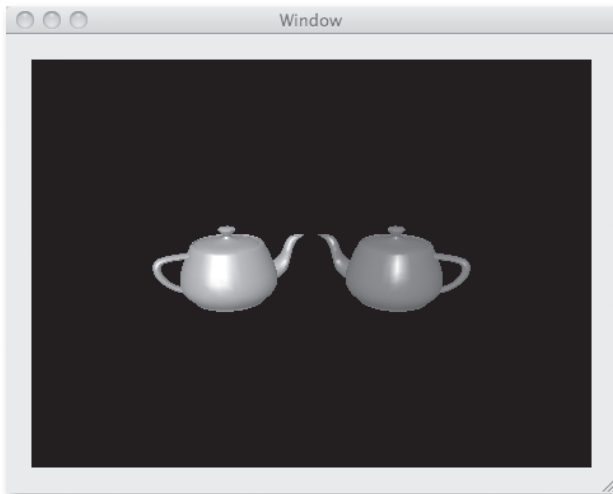
Wie man in dem Beispiel erkennen kann, sieht der Würfel doch nun um einiges »realistischer« aus.

Die Flächen, die nicht direkt beleuchtet werden, sind entsprechend dunkler als jene, die dem Licht zugewandt sind, was den 3D-Effekt natürlich noch zusätzlich unterstützt.

Glanzeffekte

Kommen wir nun zu einem Beispiel, in dem die Oberflächen mittels *GL_SHININESS* einen gewissen Glanz aufweisen.

Dieser Effekt ist besonders nützlich, wenn man metallisch wirkende Objekte darstellen möchte.



Glanzeffekt mittels *GL_SHININESS*

Das Beispielprogramm »Kapitel 5/Specular« rendert zwei Teekessel (mal was anderes) aus unterschiedlichen Materialien.

Schauen wir uns zunächst die Definition des Lichts an:

```
// Lichteigenschaften
float position[] = {-1.0, 2.0, 5.0, 1.0};
float ambient[] = {0.6, 0.6, 0.6, 1.0};
float diffuse[] = {0.6, 0.6, 0.6, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
```

In der *prepareOpenGL* erstellen wir ein leicht gedämmtes Licht (0.6 für alle 3 Farbkanaäle), positionieren und aktivieren es.

Da wir zwei verschiedene Materialien für unsere Teekessel haben möchten, müssen wir diese direkt im Rendercode zuweisen.

```
// 1. Teapot
float spec_01[] = {1.0, 1.0, 1.0, 1.0};
float ambient[] = {0.4, 0.4, 0.4, 1.0};
float diffuse[] = {0.4, 0.8, 0.4, 1.0};
glMaterialfv(GL_FRONT, GL_SPECULAR, spec_01);
glMaterialf(GL_FRONT, GL_SHININESS, 10.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
glPushMatrix();
    glTranslatef(-0.5, 0.0, 0.0);
    glRotatef(_rotation, 0.0, 1.0, 0.0);
    renderTeapot();
glPopMatrix();
```

Der erste Teekessel bekommt eine grüne Farbe mit einem Glanzwert von 10.0.

GRUNDLAGEN

Je kleiner der Wert für *GL_SHININESS*, umso weiter breitet sich der Glanzeffekt auf die Fläche aus.


```
// 2. Teapot
float spec_02[] = {0.8, 0.8, 0.8, 1.0};
float ambient_and_diffuse[] = {0.6, 0.2, 0.2, 1.0};
glMaterialfv(GL_FRONT, GL_SPECULAR, spec_02);
glMaterialf(GL_FRONT, GL_SHININESS, 125.0);
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE,
ambient_and_diffuse);
glPushMatrix();
    glTranslatef(0.5, 0.0, 0.0);
    glRotatef(-_rotation, 0.0, 1.0, 0.0);
    renderTeapot();
glPopMatrix();
```

Der zweite Teekessel hat ein rotes Material. Wie man sieht, kann man die Werte für *GL_AMBIENT* und *GL_DIFFUSE* auch zusammenfassen (*GL_AMBIENT_AND_DIFFUSE*), dies hat dann den gleichen Effekt, als würde man die beiden Werte separat zuweisen.

Der Wert für *GL_SHININESS* beträgt 125.0, was einen relativ kleinen Glanz auf der Oberfläche erzeugt.

Wie Sie sehen, habe ich das Material nur den Vorderseiten der Flächen zugewiesen, dies ist auch sinnvoll, da man die Rückseiten sowieso nicht sehen kann. Im vorherigen Beispiel hatte ich ja das Material für beide Seiten aktiviert, dies sollte aber nur zu Lernzwecken dienen.

TIPP

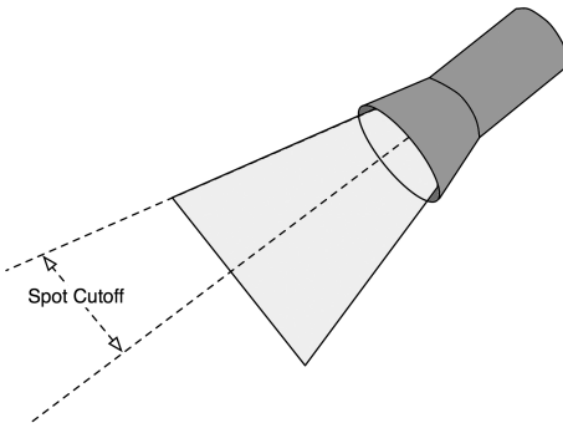
Wenn möglich, sollte nur die Vorderseite der Geometrie ein Material haben, weil dadurch der Renderprozess beschleunigt wird.

Spotlicht

Eine weitere Lichtart in OpenGL ist das sogenannte *Spot-Light*. Wie der Name schon vermuten lässt, handelt es sich hierbei um eine Art Taschenlampe, mit der man eine Szene beleuchten kann.

Um eine solche Lichtquelle zu beschreiben, nutzt man folgende Parameter:

Eigenschaft	Beschreibung
GL_SPOT_DIRECTION	Gibt die Richtung an, in die das Spotlicht zeigt
GL_SPOT_CUTOFF	Der Winkel des Lichtkegels (ausgehend vom Mittelpunkt)
GL_SPOT_EXPONENT	Beschreibt, wie stark die Lichtstärke nach außen hin abnimmt



OpenGL Spotlicht

Selbstverständlich lässt sich mit einem Spotlicht keine komplette Szene ausleuchten, aber man kann damit interessante Effekte erzielen, wie wir im nächsten Beispiel sehen werden.

Lichtposition

Bei der Lichtposition ist es wichtig zu wissen, dass sie, wie die Vertices auch, durch die Modelviewmatrix manipuliert wird.

Das heißt, man kann ein Licht rotieren und es auch im 3D-Raum bewegen.

Schauen wir uns zunächst aber noch eine Besonderheit an der Position der Lichtquelle an:

Bei dem Beispiel eben wurde die Lichtposition folgendermaßen festgelegt:

```
float position[] = {-1.0, 2.0, 5.0, 1.0};
```

Dabei ist der vierte Parameter (w) von besonderer Bedeutung.

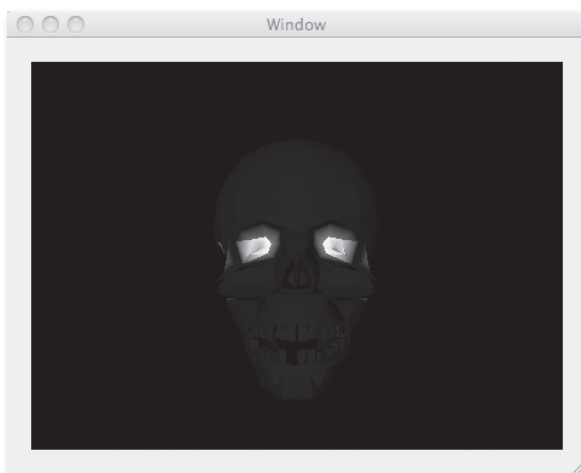
GRUNDLAGEN

Wenn dieser vierte Parameter 0.0 ist, wird nur die Richtung der Lichtquelle in den Berechnungen verwendet, nicht aber ihre Position, man spricht dann von einem »Richtungslicht«.

Wenn aber die Lichtquelle an einem bestimmten Punkt in der Szene positioniert sein soll, muss die letzte Komponente 1.0 sein.

Mit dem Wissen über Spotlichter und über die Lichtposition werden wir ein kleines Testprogramm »Kapitel 5/Rotation Light« schreiben, bei welchem drei Lichtquellen zum Einsatz kommen.

Darin sehen wir einen Schädel, in dessen Augenhöhlen je ein rotes Spotlicht platziert ist. Da sich wie gesagt mit einem Spotlicht keine Szene komplett ausleuchten lässt, habe ich zusätzlich ein »normales« Licht mit eingebaut, um eine gewisse Grundhelligkeit zu bekommen.



Szene mit 2 Spotlichtern, die das Leuchten der Augen simulieren sollen

Zuerst definieren wir wieder ein Material für das Objekt. Da wir keinen Ambiente-Wert verwenden, wirkt die Szene dunkel und dramatisch.

Den diffusen Anteil setzen wir auf 0.9 für alle Kanäle, damit wir überhaupt etwas von unserem Objekt zu sehen bekommen, und den Glanzwert drehen wir auf höchste Stufe auf (weiß).

```
// Material
float mat_ambient[] = {0.0, 0.0, 0.0, 1.0};
float mat_diffuse[] = {0.9, 0.9, 0.9, 1.0};
float mat_specular[] = {1.0, 1.0, 1.0, 1.0};
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, 10.0);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
```

Dann folgt das erste Spotlicht. Die Position setzen wir direkt im Rendercode, was wir uns gleich anschauen werden:

```
// Licht 0 Spotlicht
float light0_color_am[] = {0.3, 0.3, 0.3, 1.0};
float light0_color_diff[] = {1.0, 0.0, 0.0, 1.0};
float light0_color_spec[] = {1.0, 1.0, 1.0, 1.0};
float light0_direction[] = {-0.15, 0.05, -1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, light0_color_am);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_color_diff);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0_color_spec);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 50.0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light0_direction);
```

Der Wert für *GL_SPOT_CUTOFF* beträgt 50.0, was einem Winkel von 100 Grad entspricht, da dieser Winkel von der Mitte der Lichtquelle ausgeht. Das Licht zeigt dabei in Richtung der negativen Z-Achse.

Das zweite Spotlicht hat mit Ausnahme der Richtung exakt die gleichen Parameter, weshalb wir uns gleich das dritte Licht anschauen wollen:

```
// Licht 2 Diffuses Licht
float light2_color_am[] = {0.0, 0.0, 0.0, 1.0};
float light2_color_diff[] = {0.01, 0.01, 0.1, 1.0};
glLightfv(GL_LIGHT2, GL_AMBIENT, light2_color_am);
glLightfv(GL_LIGHT2, GL_DIFFUSE, light2_color_diff);
```

Auch hier habe ich den ambienten Anteil auf 0.0 gesetzt und die Werte für den diffusen Anteil auf ein sehr dunkles Blau eingestellt.

Die Positionierung erfolgt auch hier im Rendercode:

```
glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(0.0, 0.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// Rotierendes Licht
glPushMatrix();
float light2_position[] = {0.0, 2.0, -25.0, 1.0};
glRotatef(_rotation, 0.0, 1.0, 0.0);
glLightfv(GL_LIGHT2, GL_POSITION, light2_position);
glPopMatrix();
// Spotlicht 1 linke Augenhöhle
float light0_pos[] = {-0.15, 0.05, 0.3, 1};
glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);
// Spotlicht 2 rechte Augenhöhle
float light1_pos[] = {0.15, 0.05, 0.3, 1};
glLightfv(GL_LIGHT1, GL_POSITION, light1_pos);
renderSkull();
```

Zuerst bringen wir unsere Kamera an ihre Position, danach wird das blaue, diffuse Licht um den Schädel auf der Y-Achse rotiert. Sie sehen also, man kann auch eine Lichtquelle durch die Modelviewmatrix transformieren.

Dann kommen die beiden Spotlichter, die ich jeweils in den beiden Augenhöhlen des Schädels platziert habe, und zum Schluss wird der Schädel gerendert.

AUFGEPASST

Je mehr Lichter Sie in einer Szene haben, umso langsamer wird der Rendervorgang.

Color-Tracking

Die Farbgebung mittels Materialien hat auch gewisse Nachteile, welche ich Ihnen natürlich nicht vorenthalten möchte:

- Laut OpenGL-Dokumentation ist die Verwendung von Materialien langsam.

- Durch die vielen verschiedenen Funktionsaufrufe entsteht ein gewisser Overhead.
- Materialien können bei erweiterten Render-Techniken wie z. B. Vertex-Arrays (dazu kommen wir im Kapitel 8) nicht beliebig geändert werden. Dadurch sind sie natürlich nur bedingt nützlich.

Aus diesen Gründen gibt es eine Möglichkeit, ein Material mittels des *glColor*-Befehls zu simulieren, diese Technik nennt sich »Color-Tracking«. Dabei wird die Farbe nicht über das Material definiert, sondern einfach über die Funktion *glColor*, die Sie ja schon kennen. Schauen wir uns zunächst die Funktionsweise an:

- `glColorMaterial ( GL_FRONT_AND_BACK, GL_AMBIENT ) ;`

Die Farbgebung mittels *glColor* bezieht sich auf den ambienten Anteil und gilt für die Vorder- und Rückseiten eines Polygons.

- `glEnable ( GL_COLOR_MATERIAL ) ;`

aktiviert das Color-Tracking.

- `glColor3f(1.0, 0.0, 0.0);`

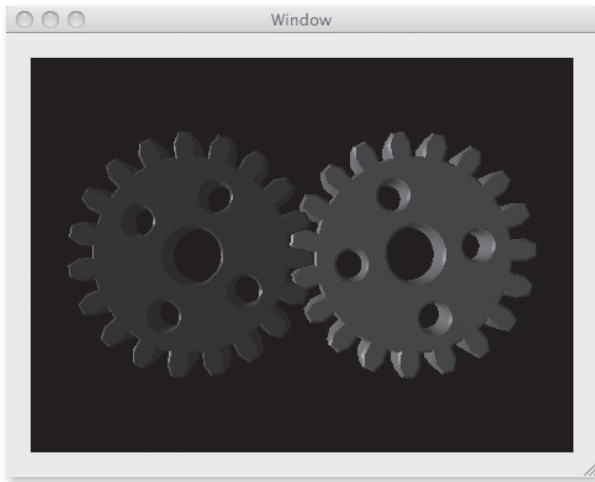
definiert die Farbe, die benutzt werden soll.

Folgende Parameter können für *glColorMaterial* definiert werden:

glColorMaterial (GLenum face, GLenum mode);

<i>face</i>	Welche Fläche soll durch die Farbe beeinflusst werden? <i>GL_FRONT</i> , <i>GL_BACK</i> , und <i>GL_FRONT_AND_BACK</i> (Standard)
<i>mode</i>	Auf welche Materialeigenschaften soll sich die Farbe auswirken? <i>GL_EMISSION</i> , <i>GL_AMBIENT</i> , <i>GL_DIFFUSE</i> , <i>GL_SPECULAR</i> , <i>GL_AMBIENT_AND_DIFFUSE</i> , <i>GL_AMBIENT_AND_DIFFUSE</i> (Standard)

Das Testprogramm »Kapitel 5/Colortracking« zeigt nochmals die Funktionsweise dieser Technik.



Farbgebung mittels `glColorMaterial(...)`

Nachfolgend die Definition der Farben für das Beispielprogramm:

```
// Licht
float position[] = {-10.0, 1.0, 0.0, 1.0};
float am[] = {0.0, 0.0, 0.0, 1.0};
float diff[] = {0.8, 0.8, 0.8, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, am);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
// Farbdefinition mittels glColor betrifft Ambient und Diffuse
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// Colortracking aktivieren
glEnable(GL_COLOR_MATERIAL);
```

Zunächst wird wieder die Lichtquelle erzeugt, was nichts Neues mehr beinhaltet. Der Befehl `glColorMaterial(...)` betrifft nur die Vorderseite der Polygone und bezieht sich auf den ambienten und diffusen Anteil.

Zum Schluss aktivieren wir das Color-Tracking.

In der `drawRect`-Methode definieren wir noch die beiden Farben und rendern die beiden Zahnräder.

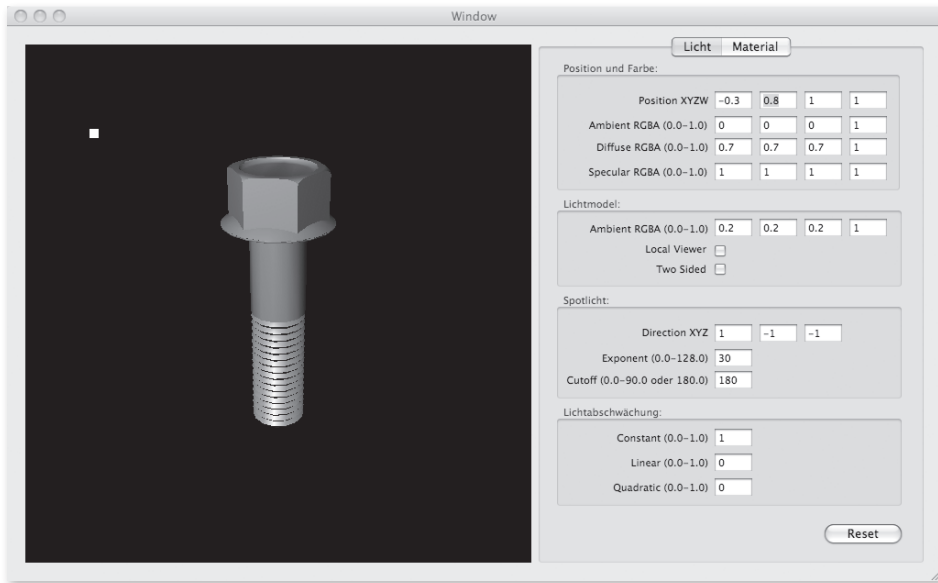
```
glPushMatrix();
    glTranslatef(-0.45, 0.0, 0.0);
    glRotatef(_rotation, 0.0, 0.0, 1.0);
    glColor3f(0.0, 0.0, 0.8); // Blaues Zahnrad
    renderGear();
glPopMatrix();
glPushMatrix();
    glTranslatef(0.45, 0.0, 0.0);
    glRotatef(-10.0, 0.0, 0.0, 1.0);
    glRotatef(-_rotation, 0.0, 0.0, 1.0);
    glColor3f(0.8, 0.0, 0.0); // Rotes Zahnrad
    renderGear();
glPopMatrix();
```

Wie Sie sehen, ist die Farbdefinition mit dem *glColor*-Befehl einfacher als über die Materialien.

AUFGEPASST

Bei eingeschaltetem Licht zeigt der *glColor*-Befehl **ohne** Color-Tracking keine Wirkung.

Wie Ihnen vielleicht schon aufgefallen ist, kann es ziemlich zeitaufwendig sein, die richtigen Parameter für das Licht und die Materialien einzustellen. Aus diesem Grund habe ich zum Schluss des Themas nochmals ein Testprogramm »Kapitel 5/Material_Light_Editor« geschrieben, mit welchem Sie alle Parameter in Echtzeit verändern können.



Editor, in dem das Licht und die Materialparameter in Echtzeit geändert werden können

Die Bedienung der jeweiligen Parameter dürfte selbsterklärend sein.

Ich habe noch zusätzlich zwei Kleinigkeiten mit eingebaut: Das Vorschauenfenster enthält ein Kontext-Menü (rechte Maustaste), mit dem Sie zwischen 4 3D-Modellen umschalten können. Weiterhin können Sie auch im Vorschauenfenster bei gedrückter linker Maustaste die X-Position der Kamera verschieben.

Der kleine, weiße Punkt links oben ist die Position des Lichts.

Zusätzliche Informationen

<http://www.xmission.com/~nate/tutors.html>

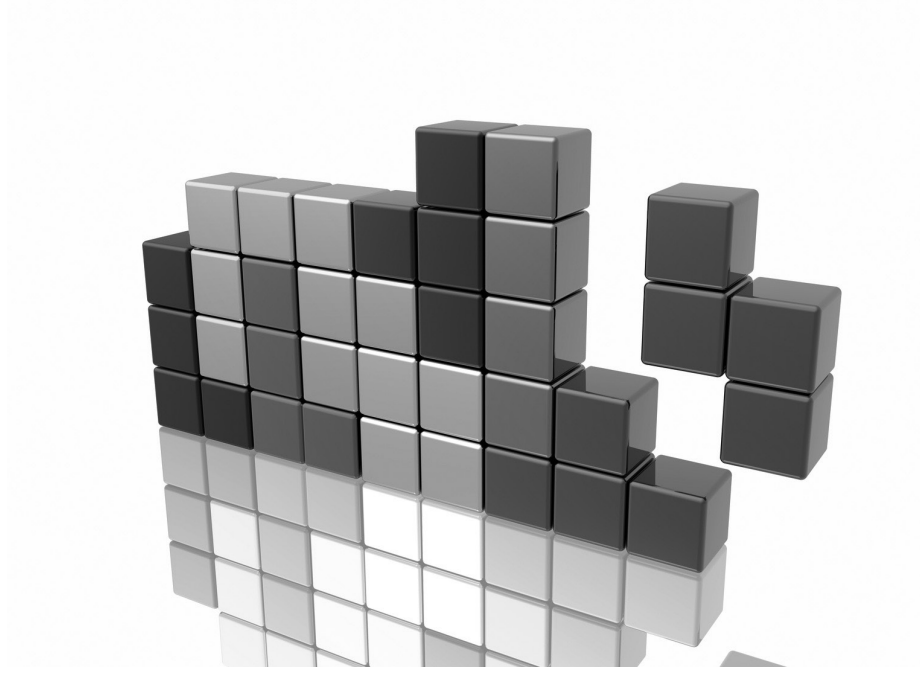
Hier findet man die beiden Programme »lightposition« und »lightmaterial«, in welchem man zur Laufzeit sämtliche Parameter verstellen kann.

<http://wiki.delphigl.com/index.php/Materialsammlung>

Eine Sammlung von verschiedenen Materialien wie z. B. Glas, Kunststoff oder Metall

<http://www.falloutsoftware.com/tutorials/gl/gl8.htm>

Ein sehr gutes Tutorial (englisch) zum Thema Licht und Materialien



Alpha-Blending

Das Alpha-Blending (oder nur Blending) ist eine Technik in OpenGL, mit deren Hilfe man transparente Materialien wie z. B. Glas oder Wasser simulieren kann, aber auch Partikeleffekte verwenden in irgendeiner Form das Blending, welches wir uns in diesem Kapitel einmal anschauen wollen.

Im den vergangenen Beispielen war bei der Definition von Farben und Materialien immer wieder die Rede von einem Alpha-Wert, den wir bis jetzt ganz dezent ignoriert haben, weil wir noch keine transparenten Objekte dargestellt haben.

Über diesen Alphawert regeln wir, wie durchsichtig ein Pixel gezeichnet werden soll. Ein Alphawert von 1.0 wäre komplett undurchsichtig, während ein Wert von 0.0 völlig transparent wäre.

OpenGL braucht nun diese Information, um Objekte in irgendeiner Form miteinander zu kombinieren, was sich dann Blending nennt.

Wie funktioniert das Blending?

Beim »Übereinanderblenden« kombinieren wir »eingehende« Pixel mit denen, die schon im Framebuffer vorhanden sind, unter Berücksichtigung des Alphawerts der Farben.

GRUNDLAGEN

Mit »eingehenden« Pixeln sind diejenigen gemeint, welche in den Framebuffer geschrieben werden.

Diese »eingehenden« Pixel werden in der Regel »source« genannt, während solche, die schon im Framebuffer stehen, »destination« heißen.

OpenGL berechnet das Blending dann mit Hilfe verschiedener Faktoren, in die »source« und »destination« mit einbezogen werden. Nachdem die beiden Pixel multipliziert wurden, werden sie zurück in den Framebuffer geschrieben.

Hier einmal ein Beispiel:

Beschreibung:

(R_s, G_s, B_s, A_s) »source« eingehende Farbe.

(R_d, G_d, B_d, A_d) »destination« vorhandene Farbe im Frame-Buffer.

(S_r, S_g, S_b, S_a) Blending-Faktor für »source«.

(D_r, D_g, D_b, D_a) Blending-Faktor für »destination«.

Formel:

$(R_s * S_r) + (R_d * D_r) = \text{rot}$

$(G_s * S_g) + (G_d * D_g) = \text{grün}$

$(B_s * S_b) + (B_d * D_b) = \text{blau}$

$(A_s * S_a) + (A_d * D_a) = \text{alpha}$

Ausgabe:

`glVertex4f(rot, grün, blau, alpha);`

Blending einschalten

Um das Blending zu verwenden, müssen wir es zunächst wie gewohnt aktivieren.

Die Funktion dazu heißt `glEnable(GL_BLEND);`

Nachdem das Blending aktiviert wurde, müssen die beiden Faktoren für »source« und »destination« festgelegt werden, was man über die Funktion `glBlendFunc(...)` macht.

Blendfunktionen

Bevor wir zu unserem ersten Beispiel kommen, schauen wir uns erst die Faktoren an, die wir für »source« und »destination« setzen können.

Funktionen für »source«

Faktor	Beschreibung
GL_ZERO	Die Farbe für »source« ist (0,0,0,0).
GL_ONE	die Farbe beibehalten (Standard)
GL_DST_COLOR	multipliziert »source« mit »destination«
GL_ONE_MINUS_DST_COLOR	multipliziert »source« mit (1,1,1,1)-»destination«
GL_SRC_ALPHA	multipliziert »source« mit dem Alphawert von »source«
GL_ONE_MINUS_SRC_ALPHA	multipliziert »source« mit dem 1-Alphawert von »source«
GL_DST_ALPHA	multipliziert »source« mit dem Alphawert von »destination«
GL_ONE_MINUS_DST_ALPHA	multipliziert »source« mit dem 1-Alphawert von »destination«
GL_SRC_ALPHA_SATURATE	multipliziert »source« mit dem Minimum von »source« und (1-»destination«)

Funktionen für »destination«

Faktor	Beschreibung
GL_ZERO	Die Farbe für »destination« ist (0,0,0,0) (Standard).
GL_ONE	die Farbe beibehalten
GL_SRC_COLOR	multipliziert »destination« mit »source«
GL_ONE_MINUS_SRC_COLOR	multipliziert »destination« mit (1,1,1,1)-»source«
GL_SRC_ALPHA	multipliziert »destination« mit dem Alphawert von »source«
GL_ONE_MINUS_SRC_ALPHA	multipliziert »destination« mit dem 1-Alphawert von »source«
GL_DST_ALPHA	multipliziert »destination« mit dem Alphawert von »destination«
GL_ONE_MINUS_DST_ALPHA	multipliziert »destination« mit dem 1-Alphawert von »destination«
GL_SRC_ALPHA_SATURATE	multipliziert »destination« mit dem Minimum von »source« und (1-»destination«)

Wie den beiden Tabellen entnommen werden kann, ergeben sich eine Vielzahl von Kombinationen, die zwar nicht immer sinnvoll sind, aber zum Schluss doch ganz interessante Effekte hervorbringen können.

Polygone sortieren

Problematisch wird das Blending, wenn eine Szene aus mehreren transparenten Polygonen besteht, die übereinander liegen, weil das Ergebnis ja von der Tiefen-anordnung der Pixel abhängig ist. Es ist also nicht egal, welches Pixel der Vordergrund »destination« und welches der Hintergrund »source« ist, da das Ergebnis ja davon abhängt, welches der beiden Pixel zuerst im Framebuffer steht.

Man kann sich aber helfen, indem man sich folgende Faustregeln merkt:

- Zeichne zuerst alle Polygone, die ohne Blending dargestellt werden sollen.
- Sortiere alle Polygone von hinten nach vorne, die mit aktivem Blending gezeichnet werden sollen.

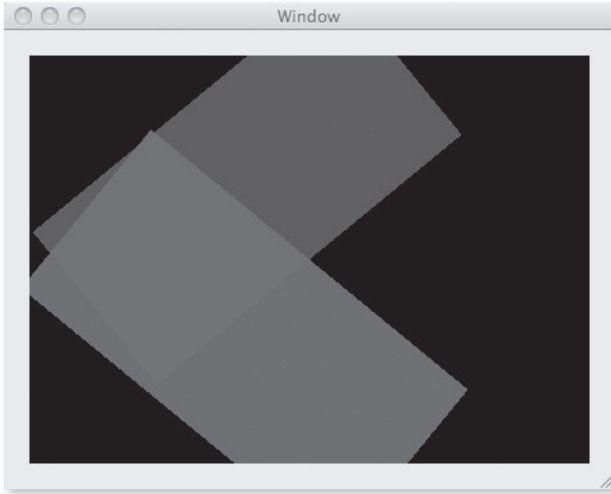
Für das Sortieren von Polygonen bietet OpenGL keine Funktionen, weshalb man sich darum selbst kümmern muss.

Ein weiteres Problem ist der Z-Buffer, der ja anhand der Z-Koordinate entscheidet, ob ein Pixel gezeichnet wird oder eben nicht, was natürlich ein Problem ist, wenn wir z. B. Objekte **hinter** einer Glasscheibe anzeigen möchten.

Bei aktivem Z-Buffer würde das Pixel ja verworfen, da es hinter ein schon bestehendes Pixel gezeichnet werden soll. Eine Möglichkeit, das zu umgehen, besteht darin, das Schreiben in den Z-Buffer zu unterdrücken, was ich Ihnen im Kapitel 10 noch zeigen werde. Am Ende des Kapitels finden Sie einen Link, der ein paar gute Tipps zum Thema Alpha-Blending enthält.

Transparente Objekte

In unserem ersten Beispiel »Kapitel 6/Simple Blending« werden 2 Rechtecke rotiert und übereinander geblendet.



Einfache Transparenz mit Hilfe von Alpha-Blending

Da auf dem Bildschirm nicht viel passiert, fällt unser Code entsprechend kurz aus:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(0.0, 0.0, 25.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
glPushMatrix();
    glRotatef(-_rotation, 0.0, 0.0, 1.0);
    glColor3f(0.0, 0.0, 1.0);
    glBegin(GL_QUADS);
        glVertex3f(-10.0, -10.0, 0.0 );
        glVertex3f( 0.0, -10.0, 0.0 );
        glVertex3f(0.0, 10.0, 0.0 );
        glVertex3f(-10.0, 10.0, 0.0 );
    glEnd();
glPopMatrix();
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glPushMatrix();
    glRotatef(_rotation, 0.0, 0.0, 1.0);
    glColor4f(1.0, 0.0, 0.0, 0.7);
```



```
glBegin(GL_QUADS);
    glVertex3f(-10.0, -10.0, 1.0 );
    glVertex3f( 0.0, -10.0, 1.0 );
    glVertex3f(0.0, 10.0, 1.0 );
    glVertex3f(-10.0, 10.0, 1.0 );
glEnd();
glPopMatrix();
glDisable(GL_BLEND);
glFlush();
```

Das erste Viereck wird ohne Blendfunktion gerendert, weshalb wir die Farbe als normalen RGB-Wert ohne Alphakanal übergeben können.

Bevor wir das zweite Viereck rendern, schalten wir das Blending ein und definieren den Blend-Faktor über *glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)*;

welches die Blendformel für Transparenz ist.

Für den zweiten Farbwert benötigen wir jetzt einen Alphakanal, weshalb wir die Farbe über

```
glColor4f(1.0, 0.0, 0.0, 0.7);
```

und damit die Transparenz auf 0.7 setzen.

GRUNDLAGEN

Je kleiner der Alpha-Wert, umso durchsichtiger wird das Objekt.

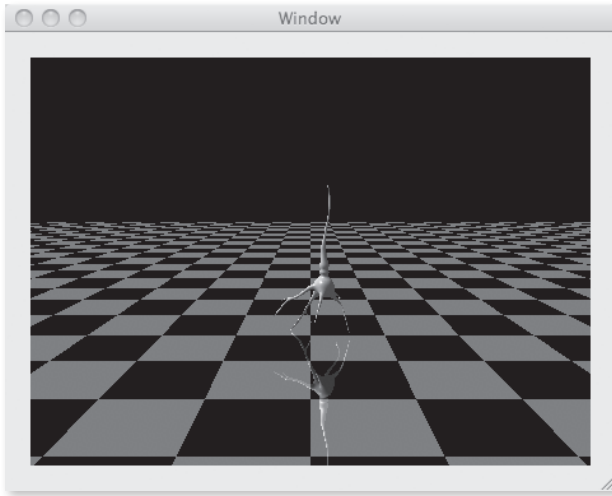
Wie gesagt ist die Anordnung der Polygone wichtig, weshalb zuerst das blaue (undurchsichtige) Viereck gerendert wird und anschließend das rote im Vordergrund.

Naja, zwei rotierende Vierecke wirken recht unspektakulär, damit lässt sich kein Blumentopf gewinnen.

Zu Anfang war ja die Rede davon, dass sich mit Blending interessante Effekte erzielen lassen, weshalb wir im nächsten Beispiel mal etwas in die Trickkiste greifen werden.

Reflektionen

Wir schauen uns einmal an, wie man mit Hilfe des Blendings Reflektionen erstellen kann.



Mit dem Blending kann man auch Reflektionen erstellen.

Wie man auf der Abbildung erkennen kann, wird das Objekt vom Boden »reflektiert«.

Der Effekt ist natürlich »gefaked«, da OpenGL selbst so etwas nicht kann.

TIPP

Ein gutes Beispiel für diesen Effekt ist das Dock von Mac OS X, welches ja auch diese Reflektion hat.

Wie funktioniert's?

Zunächst erzeugen wir uns ein Licht und aktivieren das *Color-Tracking*.

```
// Licht einstellen
// Wir moechten keine Grundauslaeuchtung der Szene.
GLfloat ambient_model[] = { 0.0f, 0.0f, 0.0f, 0.0f };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_model);
// Gedimmtes Licht
GLfloat ambientLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
```

```
// Diffuse und Glanzanteil
GLfloat diffuse_specular[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse_specular);
glLightfv(GL_LIGHT0, GL_SPECULAR, diffuse_specular);
// Licht an
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
// Colortracking aktivieren
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
float spec[] = {1.0, 1.0, 1.0, 1.0};
// Glanz einstellen
glMaterialfv(GL_FRONT, GL_SPECULAR, spec);
glMateriali(GL_FRONT, GL_SHININESS, 120);
Die beiden Lichtpositionen (ja diesmal benötigen wir 2) haben
folgende Werte.
GLfloat _lightPosition[4] = {-5.0, 5.0, 1.0, 1.0};
GLfloat _lightPositionReflection[4] = {-5.0, -5.0, 1.0, 1.0};
```

Das Rendern funktioniert folgendermaßen:

Zuerst rendern wir die Reflektion des Models. Dazu beleuchten wir es von unten (deshalb die zweite Lichtposition) und spiegeln es mit Hilfe von *glScalef(...)* auf der Y-Achse.

POWER

Wenn Sie *glScalef(...)* einen negativen Wert übergeben, können Sie damit Objekte auch spiegeln.

Danach wird es ein Stück nach unten verschoben, damit es unter dem Boden »liegt«. Anschließend rotieren und rendern wir es.

Weil wir das Objekt gespiegelt haben, müssen wir das Culling (Zeichenrichtung) der Polygone auch ändern, da es sonst zu falschen Render-Ergebnissen kommen würde.

Das bedeutet, dass nun alle Vertices, die **im Uhrzeigersinn** definiert sind, gerendert werden (Standard ist ja gegen den Uhrzeigersinn *GL_CCW*).

```
// Zuerst rendern wir die Spiegelung
glPushMatrix();
```

```

// Das Licht kommt nun von unten (gespiegelt)
glLightfv(GL_LIGHT0, GL_POSITION, _lightPositionReflection);
glPushMatrix();
    glColor3f(0.4, 0.8, 0.4);
    // Durch die Spiegelung, muessen wir die Zeichenrichtung
    // umdrehen
    // Alles was im Uhrzeigersinn definiert ist, ist vorne
    glFrontFace(GL_CW);
    // Objekt auf der Y-Achse spiegeln
    glScalef(1.0f, -1.0f, 1.0f);
    // Nach unten verschieben (unter den Boden)
    glTranslatef(0.0, 1.0, 0.0);
    // Rotieren
    glRotatef(_rotation, 0.0, 1.0, 0.0);
    // Rendern
    renderNeuron();
    // Zeichenrichtung wieder zuruecksetzen
    glFrontFace(GL_CCW);
glPopMatrix();

```

Danach rendern wir den Boden (ohne Beleuchtung) mit aktivem Blending über die Spiegelung, wodurch wir die Reflektion simulieren.

```

// Nun wird der Boden ohne Licht, mit aktiviertem Blending ueber
// die Spiegelung gerendert
glDisable(GL_LIGHTING);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
[self renderPlane];
glDisable(GL_BLEND);
glEnable(GL_LIGHTING);

```

Und zum Schluss wird das Objekt nochmals »ganz normal« gerendert. Und fertig ist die Illusion.

```

// Beleuchtung wieder zuruecksetzen
// Objekt normal rendern
glLightfv(GL_LIGHT0, GL_POSITION, _lightPosition);
glColor3f(0.4, 0.8, 0.4);
glRotatef(_rotation, 0.0, 1.0, 0.0);
    renderNeuron();
glPopMatrix();

```

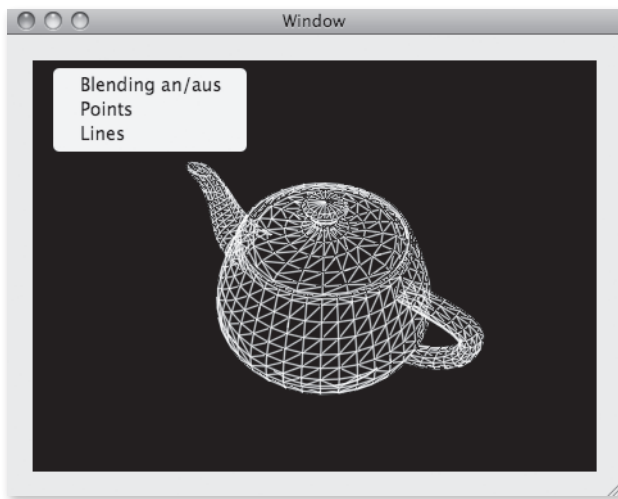
Das Ganze funktioniert hier deshalb so einfach, weil es sich um eine Ebene handelt, auf der wir die Reflektion rendern. Bei einem gekrümmten bzw. unebenen »Untergrund« ist das nicht mehr so einfach, weil dann doch einiges mehr an Mathematik nötig ist.

Antialiasing

Eine weitere Möglichkeit des Blendings ist das »Antialiasing«. Ohne diese Kanten-glättung wirken Linien und Punkte recht unschön, was im Allgemeinen als »Treppeneffekt« bekannt ist. Dieser entsteht, wenn die Geometrie auf den Bildschirm, welcher ja nicht unendlich viele Pixel darstellen kann, gerastert wird.

Mit Hilfe des Blendings hat man nun die Möglichkeit, dem Treppeneffekt entgegenzuwirken. OpenGL nimmt dazu einfach die benachbarten Pixel und blendet sie weich aus.

Das Programm »Kapitel 6/Antialiasing« zeigt diese Technik, welche sich sehr einfach nutzen lässt.



Antialiasing mittels Blending

Das Weichzeichnen wird über die beiden Funktionen

```
glEnable(GL_POINT_SMOOTH);  
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
```

bei Punkten bzw. über

```
glEnable(GL_LINE_SMOOTH);  
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
```

bei Linien eingeschaltet.

Als Blendfaktor wird dann

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

gesetzt.

Über ein Kontext-Menü können Sie zwischen den beiden Zeichenmodi (*GL_LINE* und *GL_POINT*) um- und das Blending ein- bzw. ausschalten.

Die Punktgröße wurde mittels *glPointSize(3.0f)*; auf 3.0 gesetzt, um den Effekt etwas mehr hervorzuheben.

Verhaltensregeln (Hints)

Interessant ist die Funktion *glHint(...)*, mit der man einige Verhaltensregeln von OpenGL einstellen kann.

```
void glHint( GLenum target, GLenum mode )
```

Dabei erwartet *target* das Ziel (z. B. *GL_POINT_SMOOTH_HINT*), und *mode* bestimmt das Verhalten.

Die folgende Tabelle zeigt, in welchem Zusammenhang man diese Hints nutzen kann.

<i>target</i>	Beschreibung
GL_GENERATE_MIPMAP_HINT	besagt, ob es bei der Erstellung der MipMaps (Textur) auf Geschwindigkeit oder Qualität ankommen soll
GL_FOG_HINT	bestimmt die Genauigkeit der Nebelberechnung
GL_LINE_SMOOTH_HINT	Samplequalität der Kantenglättung
GL_PERSPECTIVE_CORRECTION_HINT	Qualität der perspektivisch korrekten Darstellung
GL_POINT_SMOOTH_HINT	Samplequalität der Kantenglättung
GL_POLYGON_SMOOTH_HINT	Samplequalität der Kantenglättung

Hier die Beschreibungen für das Verhalten:

<i>Mode</i>	Beschreibung
GL_FASTEST	Die Geometrie wird so schnell wie möglich gerendert (meistens mit geringerer Qualität).
GL_NICEST	Die Geometrie wird am »schönsten« gerendert (meist langsamer).
GL_DONT_CARE	OpenGL soll selbst bestimmen, wie die Ausgabe erfolgt.

In unserem Beispiel betreffen die Hints also das Zeichnen der Punkte und Linien:

```
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);  
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
```

AUFGEPASST

Diese Hints sind treiberabhängig, was bedeutet, dass es der OpenGL-Implementierung überlassen ist, wie sie mit diesen umgeht.

Zusammenfassung

Wie Sie gesehen haben, kann man mit dem Alpha-Blending sehr interessante Effekte wie z. B. Transparenz und Reflektionen erzielen. Auch die Kantenglättung ist damit möglich. Wir werden im nächsten Kapitel noch sehen, wie man damit auch das sogenannte Alpha-Masking erzielen kann, welches sehr oft bei Sprites benutzt wird.

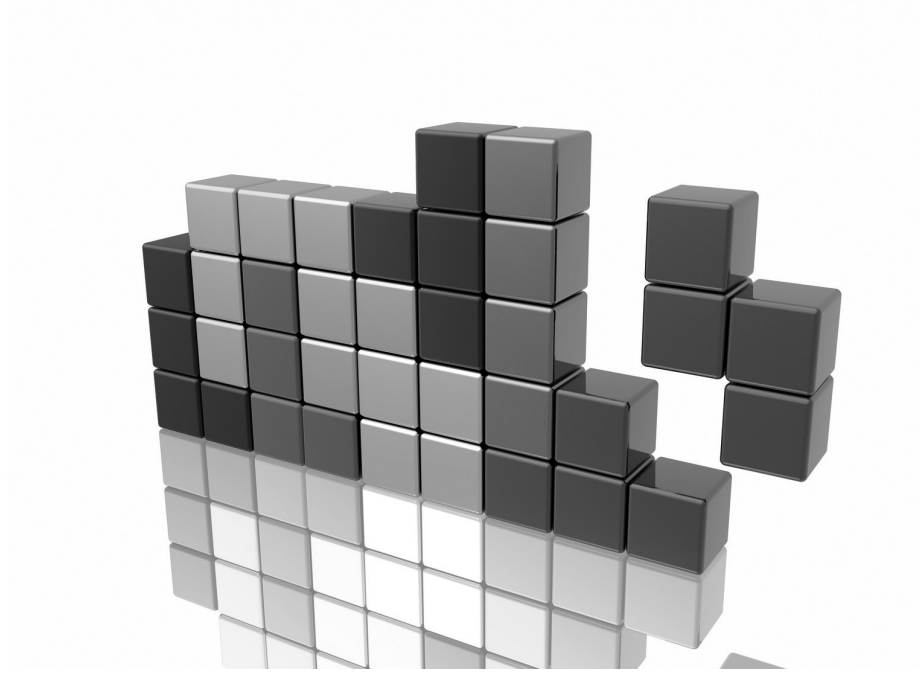
Zusätzliche Informationen

<http://www.3dsourc.de/faq/transparency.htm>

FAQ's zu Transparenz und Blending

http://www.opengl.org/wiki/index.php/Alpha_Blending#Another_Good_Trick

Ein paar gute Tipps zum Alpha-Blending



Texturierung

Kommen wir nun zu einer Technik, mit der wir unserer Szene noch mehr »Pfiff« verleihen können.

Bunte Teekannen sind nicht unbedingt das, was wir möchten, wir benötigen mehr »Realität«. Über die Beleuchtung sind wir der Sache zwar schon näher gekommen, aber über das sogenannte Texture-Mapping können wir unsere Objekte so darstellen, wie wir sie aus unserer Umgebung kennen.

Das Texture-Mapping ist ein Verfahren, mit welchem man zweidimensionale Bilder (Texturen) auf 3D-Objekte »mapt«.

Nehmen wir als einfaches Beispiel eine Kugel und texturieren sie mit dem Bild der Erde.



Einfach schön, unsere Erde

Das Ergebnis kann sich sehen lassen, mit etwas mehr als 500 Dreiecken und einem einfachen Bild kann man schon recht beeindruckende Ergebnisse erzielen.

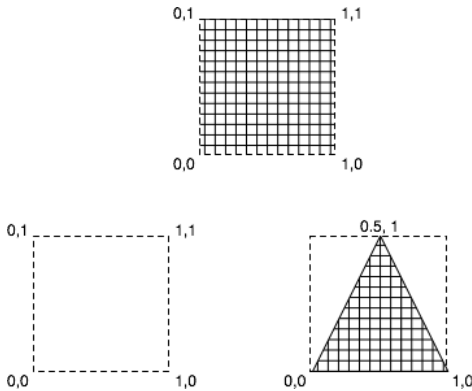
Die Frage ist nun: Wie kommt ein zweidimensionales Bild auf ein 3D-Objekt?

Sämtlichen Polygonen eines Objekts werden sogenannte Textur-Koordinaten zugewiesen, welche von 0.0 (links unten) bis 1.1 (rechts oben) reichen.

GRUNDLAGEN

Die Textur-Koordinaten werden oft als »uv«- bzw. »st-Koordinaten« bezeichnet.

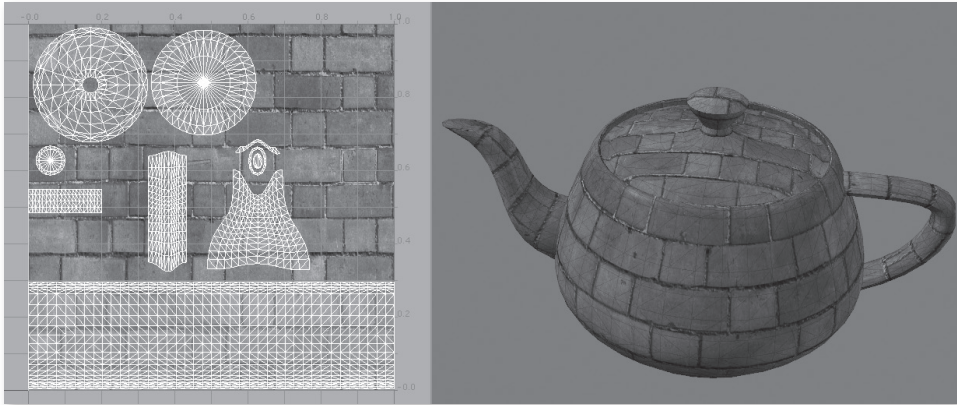
Diese Koordinaten werden dann auf das zweidimensionale Bild übertragen, wodurch sich anschließend die Farbe an der entsprechenden Position im Polygon ergibt. Folgende Abbildung soll das nochmals verdeutlichen.



Übertragen von Textur-Koordinaten auf ein zweidimensionales Bild

Links unten haben wir ein Viereck, was der Idealfall wäre, da sich hier die Textur-Koordinaten natürlich sehr einfach übertragen lassen. Rechts daneben ist ein Dreieck mit der gleichen Textur, wie Sie sehen, ändert sich hier nur der Wert für den oberen Eckpunkt.

Das sind natürlich sehr einfache Beispiele, in der Regel haben wir es mit viel komplexeren Formen zu tun, wobei es dann nicht mehr so ohne Weiteres möglich ist, die Textur-Koordinaten selbst zu berechnen. Diesen Part übernimmt dann normalerweise ein 3D-Modelling-Programm, wie es die folgende Abbildung zeigt.



Textur-Koordinaten für eine Teekanne

Auf der linken Seite sieht man die flach abgewinkelte Geometrie, die genau in den Bereich der uv-Koordinaten passt. Auf der rechten Seite dann die Teekanne mit der Textur selbst.

GRUNDLAGEN

Im Zusammenhang mit Texturen fällt sehr oft der Begriff »Texel«, dabei handelt es sich um die einzelnen Pixel der Textur.

Texturen laden

Wie die Textur auf die Geometrie kommt, wissen wir nun, als Nächstes müssen wir uns anschauen, wie man eine Textur erstellt. Hierbei gibt es mehrere Möglichkeiten, man kann z. B. eine Textur

- von einem Datenträger laden
- von einem bereits gerenderten Bild erzeugen (Render to Texture)
- aus einem Cocoa-View erstellen (inkl. einem Quicktime-Film)
- aus einer mathematischen Formel erstellen (z. B. Noise-Texture)

Uns genügt es zu Anfang, eine Textur von einem Datenträger zu laden.

OpenGL bietet uns zu diesem Zweck drei Funktionen an, wobei wir nur mit der Funktion arbeiten werden, welche eine 2D-Textur erstellt.

GRUNDLAGEN

Die beiden anderen Typen unterscheiden sich fast nur an der Anzahl der Dimensionen, OpenGL kann nämlich auch 1D und 3D Texturen erstellen, wobei die Funktionen dann *glTexImage1D(...)* bzw. *glTexImage3D(...)* heißen. 1D-Texturen werden oft für das sogenannte Toon-Shading benutzt, welches man im Spiel XIII (Ubisoft) sieht. 3D-Texturen findet man eher in der Medizin (Schichtaufnahmen).

Wir erstellen unsere 2D-Textur mit der Funktion

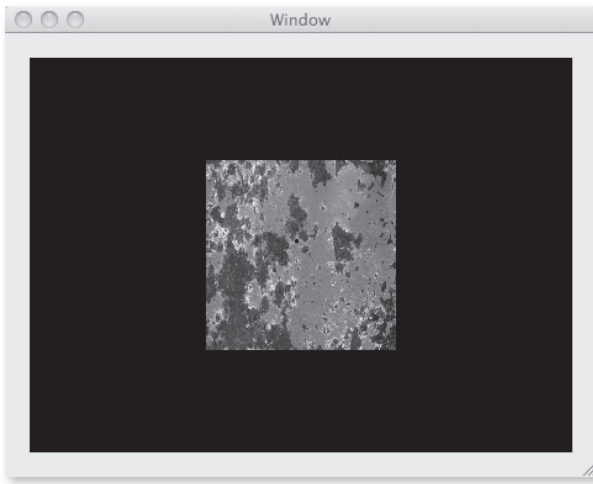
```
void glTexImage2D(GLenum target, GLint level, GLint internalformat,
GLsizei width, GLsizei height, GLint border, GLenum format,
GLenum type, const GLvoid *pixels )
```

Diese erwartet folgende Parameter (stark gekürzt):

Parameter	Beschreibung
Target	GL_TEXTURE_2D
Level	MipMap Level
internalformat	RGB / RGBA
Width	Breite
Height	Höhe
Border	Breite des Rahmens um die Textur
Format	GL_RGB / GL_RGBA
Type	Pixeltyp (GL_UNSIGNED_BYTE, usw...)
Pixels	die eigentlichen Daten

Ich werde die einzelnen Parameter im weiteren Verlauf noch genauer erklären.

Jetzt brauchen wir noch eine Möglichkeit, die Textur zu laden. Hier kommt jetzt eine der Annehmlichkeiten von Cocoa, die ich in der Einführung schon erwähnt habe: Mit Hilfe der Klasse *NSBitmapImageRep* benötigen wir etwas mehr als eine handvoll Code, um uns eine OpenGL-Textur zu erstellen, was wir jetzt in unserem ersten Beispiel »Kapitel 7/Simple Texture« auch tun werden. Wie Sie in der folgenden Abbildung sehen können, ist es recht unspektakulär und macht nicht mehr, als ein texturiertes Viereck auszugeben.



Ausgabe unseres texturierten Vierecks

Schauen wir uns den Code dazu an.

Wir definieren zuerst eine Variable, die später die Textur beinhaltet:

```
GLuint _texture;
```

GRUNDLAGEN

Eine gültige OpenGL-Textur ist nichts anderes als ein einfacher *unsigned int*-Wert.

Dann wird in der *prepareOpenGL*-Methode die Texturierung mittels

```
glEnable(GL_TEXTURE_2D); // Texturierung einschalten
```

aktiviert.

Anschließend laden wir die Textur.

```
_texture = [self textureByName:[NSBundle mainBundle]pathForResource:@"texture" ofType:@"tiff"]];
```

Die Laderoutine schauen wir uns gleich noch genauer an.

Wenn die Textur erfolgreich geladen wurde (*_texture* muss einen Wert größer Null haben), müssen wir sie »binden«, was folgende Funktion macht:

```
glBindTexture(GL_TEXTURE_2D, _texture); // Textur binden
```

Über dieses »Binden« setzen wir die Textur als »aktiv«, das bedeutet, dass sämtliche Geometrie (State-Machine), die über Texturkoordinaten verfügt, mit dieser Textur gerendert wird.

GRUNDLAGEN

Wenn Sie vergessen, Texturkoordinaten zu erstellen, wird die Geometrie auch nicht texturiert gerendert.

Nun ist es aber so, dass wir das meistens gar nicht wollen, in der Regel will man nämlich, dass jedes Objekt mit seiner eigenen Textur gezeichnet werden soll.

Das ist kein Problem, denn alles, was wir tun müssen, ist während des Rendervorgangs einfach die jeweilige Textur zu binden, was z. B. so aussehen könnte:

```
glBindTexture(GL_TEXTURE_2D, _textureRaumschiff);  
rendere Raumschiff  
glBindTexture(GL_TEXTURE_2D, _textureMonster);  
rendere Monster  
glBindTexture(GL_TEXTURE_2D, _textureSpieler);  
render Spieler
```

usw.

Nachdem nun die Textur aktiviert wurde, wird unser Viereck gerendert, dieses Mal müssen wir zusätzlich zu den normalen Koordinaten noch die uv-Koordinaten mittels *glTexCoord2f(...)* übergeben, und zwar **bevor** wir die Vertices definieren.

```
glBegin(GL_QUADS);  
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);  
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);  
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, 0.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, 0.0f);  
glEnd();
```

Wir fangen wieder links unten an und erstellen das Viereck gegen den Uhrzeigersinn (*GL_CCW*). Wie Sie sehen, reichen die uv-Koordinaten von (0,0) für links unten, bis (1,1) für rechts oben. Alle Koordinaten müssen in diesem Bereich liegen.

AUFGEPASST

uv-Koordinaten müssen vor den Vertex-Daten definiert werden.

Kommen wir nun zu der eigentlichen Laderoutine, die eine Textur erstellt. Diese erfolgt in zwei Schritten.

Im ersten Schritt wird das Bild mittels der Klasse *NSBitmapImageRep* geladen, welche uns die Rohdaten liefert.

```
NSBitmapImageRep *LoadImage(NSString *path, int
shouldFlipVertical)
{
    NSBitmapImageRep *bitmapimagerep;
    NSImage *image;
    image = [[[NSImage alloc] initWithContentsOfFile: path]
autorelease];
    bitmapimagerep = [[[NSBitmapImageRep alloc] initWithData:[im-
age TIFFRepresentation]]];
    if (shouldFlipVertical)
    {
        int bytesPerRow, lowRow, highRow;
        unsigned char *pixelData, *swapRow;
        bytesPerRow = [bitmapimagerep bytesPerRow];
        pixelData = [bitmapimagerep bitmapData];
        swapRow = (unsigned char *)malloc(bytesPerRow);
        for (lowRow = 0, highRow = [bitmapimagerep pixelsHigh]-1;
lowRow < highRow; lowRow++, highRow--)
        {
            memcpy(swapRow, &pixelData[lowRow*bytesPerRow],
bytesPerRow);
            memcpy(&pixelData[lowRow*bytesPerRow], &pixelData[-
highRow*bytesPerRow], bytesPerRow);
            memcpy(&pixelData[highRow*bytesPerRow], swapRow,
bytesPerRow);
        }
        free(swapRow);
    }
}
```



```

        return bitmapimagerep;
    }

```

Die Variable *shouldFlipVertical* wird benötigt, da unter Umständen das Bild gespiegelt abgespeichert wurde (einige Bildbearbeitungsprogramme machen das anscheinend so, wofür ich leider keine Erklärung habe).

Im zweiten Schritt wird aus diesen Rohdaten dann eine OpenGL-Textur erzeugt.

```

- (GLuint)textureByName:(NSString *)textureName
{
    NSBitmapImageRep *bitmapimagerep;
    NSRect rect;
    bitmapimagerep = LoadImage(textureName, 1);
    rect = NSMakeRect(0, 0, [bitmapimagerep pixelsWide], [bitmapimagerep pixelsHigh]);
    GLuint tmpTexture;
    glGenTextures(1, &tmpTexture);
    glBindTexture(GL_TEXTURE_2D, tmpTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_FALSE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, rect.size.width, rect.size.height, 0,
        ([[bitmapimagerep hasAlpha]]?(GL_RGBA):(GL_RGB)),
        GL_UNSIGNED_BYTE,
        [bitmapimagerep bitmapData]);
    return tmpTexture;
}

```

Gehen wir die einzelnen Codeabschnitte einmal genauer durch.

Zunächst generieren wir eine temporäre OpenGL-Textur und binden diese an *GL_TEXTURE_2D*, was OpenGL mitteilen soll, dass wir eine zweidimensionale Textur erstellen möchten.

```

GLuint tmpTexture;
glGenTextures(1, &tmpTexture); //generiere 1 Textur mit dem
                                // Namen tmpTexture
glBindTexture(GL_TEXTURE_2D, tmpTexture);

```

Über die Funktion

```
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_FALSE);
```

sagen wir, dass wir keine Mip-Maps haben wollen. Was es damit auf sich hat, sehen wir an späterer Stelle in diesem Kapitel.

Danach legen wir die Filtereigenschaften für die Textur fest, auch diese schauen wir uns später genauer an.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Und zum Schluss wird dann die Textur erstellt und zurückgegeben.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, rect.size.width, rect.size.height, 0,
    ([[bitmapimagerep hasAlpha]]?(GL_RGBA):(GL_RGB)), GL_UNSIGNED_BYTE, [bitmapimagerep bitmapData]);
return tmpTexture;
```

Hier nochmals die einzelnen Parameter:

Parameter	Beschreibung
GL_TEXTURE_2D	erstellen eine zweidimensionale Textur
0	erstellen keine Mip-Maps
GL_RGB	ist das Format (wir haben keinen Alphakanal)
rect.size.width und rect.size.height	geben die Größe der Textur an
0	erzeugen keinen Rand um die Textur
([[bitmapimagerep hasAlpha]]?(GL_RGBA):(GL_RGB))	das interne Format
GL_UNSIGNED_BYTE	Typ der Daten
[bitmapimagerep bitmapData]	die eigentlichen Rohdaten des Bildes

Fassen wir also zusammen. Die Texturierung kann man grob in drei Schritte zusammenfassen:

1. Texturierung einschalten

```
glEnable(GL_TEXTURE_2D);
```

2. Textur laden und binden

```
_texture = [self textureByName:[NSBundle mainBundle]pathForResource:@"texture" ofType:@"tiff"]];
glBindTexture(GL_TEXTURE_2D, _texture);
```

3. Geometrie mit uv-Koordinaten rendern.

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, 0.0f);
glEnd();
```

Textur löschen

Da eine OpenGL-Textur im Grafikkarten-Speicher abgelegt wird, ist es keine schlechte Idee, sie zu löschen, wenn man sie nicht mehr benötigt, was man mit Hilfe folgender Funktion macht:

```
glDeleteTextures(1, &_amp;_texture); // Texture loeschen
```

Man übergibt der Funktion einfach die Anzahl und den Namen der freizugebenden Texturen.

Textur-Größe

Hinsichtlich der Größe einer Textur gibt es einige Einschränkungen, die sehr von der verwendeten Grafikkarte (bzw. dem System) abhängig sind. Zum Beispiel erfahren Sie mit der Funktion

```
GLint texSize; glGetIntegerv(GL_MAX_TEXTURE_SIZE, &texSize);
```

die maximale Größe, welche eine Textur haben darf.

Weiterhin ist es auf einigen Systemen (wahrscheinlich den meisten) nicht erlaubt, Texturen mit einer Größe zu erzeugen, die nicht durch 2 teilbar ist.

POWER

Der Begriff »durch 2 teilbar« ist besser bekannt als Power-Of-Two-Textur.

Das heißt, wenn Sie eine Textur mit der Größe von 132 x 342 erzeugen möchten, wird OpenGL diese einfach nicht laden, und Sie werden anstelle der Textur nur ein »schönes« Weiß auf Ihrer Geometrie sehen. Auf neueren Systemen ist es aber möglich, mit einer Extension (*ARB_texture_rectangle*) Texturen zu erzeugen, die diese Einschränkung nicht haben. Empfehlenswert ist dies aber nicht, da wie gesagt die meisten Systeme diese Erweiterung noch nicht unterstützen und auch der volle Funktionsumfang für diese Art von Texturen nicht zur Verfügung steht.

Textur-Umgebung

Über die Textur-Umgebung sagen wir OpenGL was mit der Farbe, die unsere Geometrie hat, passieren soll, wenn wir eine Textur »darüber legen«.

Dieses Verhalten wird mit der Funktion

```
void glTexEnv( GLenum target, GLenum pname, GLfloat param )
```

gesteuert. Wobei ich hier nur den letzten Parameter genauer erklären möchte, welcher einer der Folgenden sein kann:

- *GL_MODULATE (Standard)*
multipliziert die Textur-Farbe (Texel) mit der Geometrie-Farbe (nachdem sie beleuchtet wurde).
- *GL_REPLACE*
ersetzt die Geometrie-Farbe mit der Textur-Farbe (Beleuchtungs-Effekte gehen dadurch auch verloren)
- *GL_DECAL*
Wenn die Textur einen Alpha-Kanal besitzt, scheint die Geometrie-Farbe an dieser Stelle hindurch, wenn nicht, arbeitet der Modus wie *GL_REPLACE*.
- *GL_BLEND*
Textur-Farbe wird mit einer konstanten Blend-Farbe gemischt.

- *GL_ADD*
addiert die Oberflächenfarbe mit der Textur-Farbe.
- *GL_COMBINE*
kombiniert die Farben anhand eines Kombinations-Modus (Texture-Combiners).

Auf diesen letzten Modus werden wir aber aus folgenden Gründen nicht näher eingehen:

- Das Thema ist meiner Meinung nach zu komplex für den Einstieg.
- Durch den Einzug der Shader-Programmierung ist es veraltet.

Im Kapitel 12, »Shader«, werden wir eine bessere Möglichkeit kennenlernen, um Texturen miteinander zu kombinieren.

Mit dem Beispielprogramm »Kapitel 7/Texture Environment« können Sie die einzelnen Modi mal in der Praxis ausprobieren. Es zeigt nochmals das Viereck aus dem vorangegangenen Beispiel mit einem Kontext-Menü, über welches Sie zwischen den verschiedenen Modi umschalten können.

Die Textur hat nun einen Alphakanal (Farbverlauf von oben nach unten), den man im Modus *GL_DECAL* sehr schön erkennen kann. Im Modus *GL_REPLACE* kann man nochmals sehen, dass die Beleuchtung komplett ignoriert wird.

Der Code zu diesem Beispiel bringt nicht viel Neues, außer dass für den Modus *GL_BLEND* eine Blendfarbe definiert werden musste, wodurch wir zusätzlich ein Blau über das Viereck blenden.

```
GLfloat blendColor[4] = {0.0, 0.0, 1.0, 0.0};
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, blendColor);
```

Beim Generieren der Textur wird jetzt ein Format mit Alphakanal angelegt, da wir diesen ja im Modus *GL_DECAL* benötigen.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, rect.size.width, rect.size.height, 0,
    (((bitmapimagerep hasAlpha)?(GL_RGBA):(GL_RGB)), GL_UNSIGNED_BYTE, [bitmapimagerep bitmapData]));
```

Texturen »wickeln«

Wie man die Textur-Koordinaten definiert, haben wir weiter oben ja schon gesehen, bleibt noch die Frage, was passiert, wenn wir Textur-Koordinaten angeben, die nicht im Bereich zwischen 0.0 – 1.0 liegen.

Über die beiden Funktionen

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, ...);
```

und

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, ...);
```

können wir OpenGL sagen, wie das System mit solchen Textur-Koordinaten verfahren soll.

Das Verhalten kann man für beide Richtungen (s und t, bzw. u und v) separat angeben.

Folgende Werte sind erlaubt:

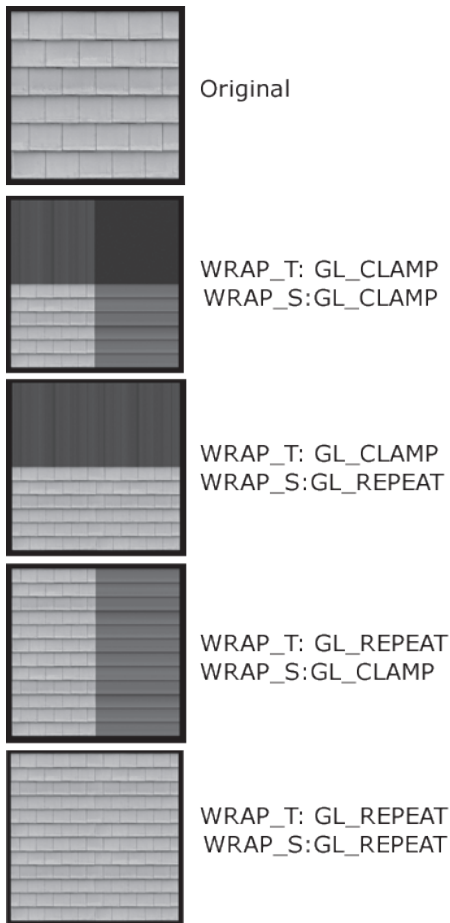
- *GL_CLAMP* beschränkt die Reichweite der Koordinaten zwischen 0.0 - 1.0.
- *GL_REPEAT* (Standard) wiederholt die Koordinaten so oft, wie sie angegeben wurden. Wenn Sie z. B. folgenden Code haben:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);  
glTexCoord2f(10.0, 10.0);
```

würde die Textur zehnmal in beide Richtungen wiederholt werden.

- *GL_CLAMP_TO_EDGE* wird benutzt, wenn ein Rand für eine Textur definiert wurde (Border = 1), dieser aber ignoriert werden soll.

Hier eine Abbildung, welche die Parameter nochmals verdeutlichen soll.

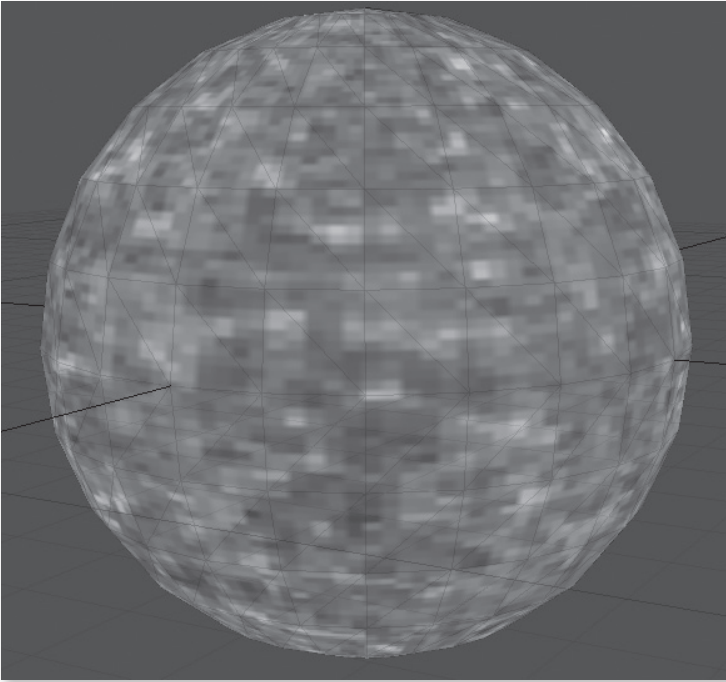


*Die verschiedenen »Wicklungs-Funktionen«
von OpenGL*

Das oberste Bild hat Textur-Koordinaten, die von 0.0–1.0 reichen. Bei allen anderen Bildern gehen die Textur-Koordinaten von 0.0 bis 2.0.

Texturen filtern

Ein Problem bei der Verwendung von Texturen ist, dass die Größe der einzelnen Texel einer Textur so gut wie nie der Größe der Geometrie (Pixel) entspricht. Dadurch müssen die Texel zum Teil sehr stark vergrößert bzw. verkleinert werden, was zu unschönen Ergebnissen führen kann, wie man an folgender Abbildung sehr schön sieht.



Eine zu kleine Textur wirkt sehr pixelig.

Um diesem Problem entgegenzuwirken, bietet OpenGL die Möglichkeit, Texturen zu filtern.

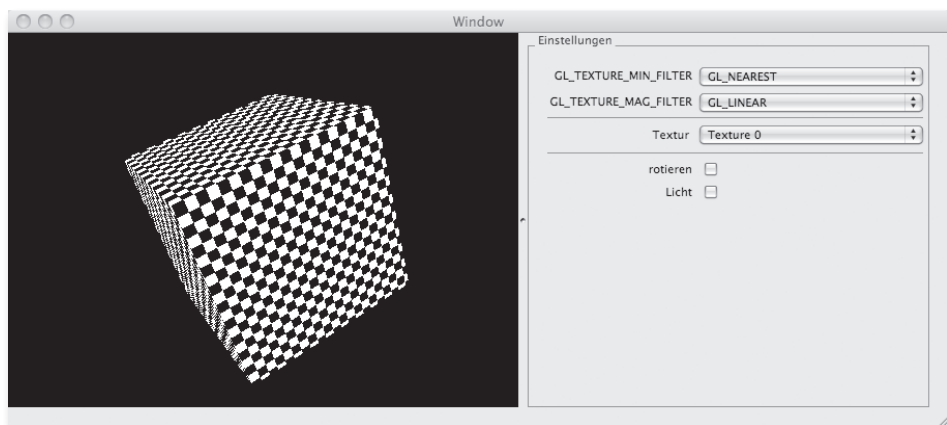
Dabei kann man für beide Fälle (Textur zu groß bzw. zu klein) einstellen, wie sie gefiltert werden sollen. Steuern lässt sich das mit folgender Funktion:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Der erste Parameter muss `GL_TEXTURE_2D` sein, der zweite Parameter steht für »verkleinern« (`GL_TEXTURE_MIN_FILTER`) bzw. »vergrößern« (`GL_TEXTURE_MAG_FILTER`) und der letzte Parameter sagt aus, wie gefiltert werden soll. Folgende Filter stehen zur Auswahl:

Filter	Beschreibung
GL_NEAREST	nimmt das Texel, das am nächsten am Zentrum des texturierten Pixels liegt
GL_LINEAR	lineare Interpolation zwischen den 4 umliegenden Pixeln
GL_NEAREST_MIPMAP_NEAREST	wählt das Bild, dessen Größe am besten zu der Größe des Polygons passt unter Verwendung von <i>GL_NEAREST</i>
GL_NEAREST_MIPMAP_LINEAR	wählt das Bild, dessen Größe am besten zu der Größe des Polygons passt unter Verwendung von <i>GL_LINEAR</i>
GL_LINEAR_MIPMAP_NEAREST	lineare Interpolation zwischen den beiden Bildern, deren Größe am besten zu der Größe des Polygons passt unter Verwendung von <i>GL_NEAREST</i>
GL_LINEAR_MIPMAP_LINEAR	lineare Interpolation zwischen den beiden Bildern, deren Größe am besten zu der Größe des Polygons passt unter Verwendung von <i>GL_LINEAR</i>

Das Beispielprogramm »Kapitel 7/Texture Filtering« zeigt die einzelnen Filter in der Praxis.



Beispielprogramm zu den einzelnen Textur-Filtern

Die Bedienung dürfte wieder selbsterklärend sein. Bei gedrückter linker Maustaste können Sie die Szene zoomen, dadurch lässt sich der Filter-Effekt besser erkennen.

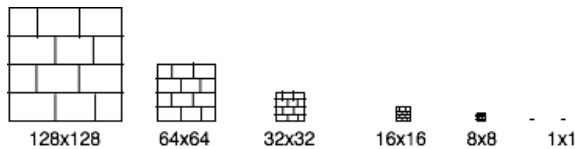
Da in dem Beispiel Mip-Maps verwendet werden, hat sich die Laderoutine für die Texturen ein wenig geändert.

```
- (GLuint)textureByName:(NSString *)textureName
{
    NSBitmapImageRep *bitmapimagerep;
    NSRect rect;
    bitmapimagerep = LoadImage(textureName, 1);
    rect = NSMakeRect(0, 0, [bitmapimagerep pixelsWide], [bitmapimagerep pixelsHigh]);
    GLuint tmpTexture;
    glGenTextures(1, &tmpTexture); //generiere 1 Textur mit dem
                                   // Namen tmpTexture
    glBindTexture(GL_TEXTURE_2D, tmpTexture);
    glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_NEAREST);
    /*glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, rect.size.width, rect.size.height, 0,
        ([[bitmapimagerep hasAlpha])?(GL_RGBA):(GL_RGB)),
        GL_UNSIGNED_BYTE, [bitmapimagerep bitmapData]);*/
    // MipMaps erzeugen. glTexImage2D wird dadurch automatisch
    // aufgerufen
    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, rect.size.width,
        rect.size.height, ([[bitmapimagerep hasAlpha])?(GL_RGBA):(GL_RGB)),
        GL_UNSIGNED_BYTE,
        [bitmapimagerep bitmapData]);
    return tmpTexture;
}
```

Der Aufruf von *gluBuild2DMipmaps* (eine Funktion aus der GLU-Bibliothek) erzeugt automatisch eine Textur, weshalb nun der Aufruf von *glTexImage2D(...)* wegfällt. Lüften wir nun das Geheimnis um die Mip-Maps:

Mip-Maps

Bei den Mip-Maps handelt es sich um verkleinerte Versionen der Texturen, die OpenGL auf Wunsch für uns erzeugt. Wenn wir z. B. eine Textur mit der Größe von 256 x 256 Pixeln hätten, würden die Mip-Maps wie in der folgenden Abbildung aussehen:



Erzeugung von Mip-Maps

Je nachdem, wie groß nun unser Objekt angezeigt wird, ersetzt OpenGL die Textur mit der Mip-Map, die am besten der Größe der Geometrie entspricht. Das hat zur Folge, dass sich dadurch die Darstellungsqualität mitunter stark verbessert, weil Texturen nicht mehr umgerechnet werden müssen, sondern einfach durch die passende Mip-Map ausgetauscht werden.

POWER

Der Prozess, welcher die passende Mip-Map auswählt, nennt sich »Level of Detail« oder kurz »LOD«.

Wie Sie im obigen Beispiel gesehen haben, werden die Mip-Maps auch bei der Filterung verwendet, was auch zu einer Verbesserung der Ausgabe führen kann. Um Mip-Maps zu erzeugen, gibt es zwei Wege: Zum einen über die Funktion *glTexImage2D(...)*, wobei Sie jede Größe der Mip-Maps extra angeben müssen:

```
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB, 128, 128, 0, GL_RGB GL_UNSIGNED_BYTE, data);
glTexImage2D(GL_TEXTURE_2D,1,GL_RGB, 64, 64, 0, GL_RGB GL_UNSIGNED_BYTE, data);
glTexImage2D(GL_TEXTURE_2D,2,GL_RGB, 32, 32, 0, GL_RGB GL_UNSIGNED_BYTE, data);
glTexImage2D(GL_TEXTURE_2D,3,GL_RGB, 16, 16, 0, GL_RGB GL_UNSIGNED_BYTE, data);
```

usw.

oder über die Funktion *gluBuild2DMipmaps(...)*, welche die Erzeugung automatisch vornimmt. Wie oben schon erwähnt, wird mit *gluBuild2DMipmaps(...)* auch die Textur erstellt, weshalb der Aufruf von *glTexImage2D(...)* entfallen kann.

Secondary Color

Im obigen Beispiel »Kapitel 7/Texture Filtering« haben wir das Material ja folgendermaßen definiert:

```
// Material
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glMaterialfv(GL_FRONT, GL_SPECULAR, fullLight);
glMateriali(GL_FRONT, GL_SHININESS, 128);
```

Wie man sieht, hat dieses auch einen Glanzanteil. Schaut man sich das Beispielprogramm nochmals an, wird man feststellen, dass der Glanzeffekt vollkommen fehlt. Warum ist das so? Funktioniert das nur ohne Textur?

Nein natürlich nicht. Das »Problem« ist, dass normalerweise die beleuchtete Farbe mit der Textur multipliziert wird, wodurch dann der Glanzanteil komplett verloren geht. Dieses Problem umgeht man mit einer Technik, die sich »secondary-specular-color« nennt. Dabei wird das Glanzlicht separat berechnet und nachträglich zur Textur hinzuaddiert, wodurch dann wieder der übliche Glanz zu sehen ist.

Um diese »Secondary Color« Technik zu nutzen, fügen wir einfach in die *prepare-OpenGL*-Methode folgende Zeile ein:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL,
GL_SEPARATE_SPECULAR_COLOR);
```

Schon haben wir wieder den gewünschten Glanzeffekt.

GRUNDLAGEN

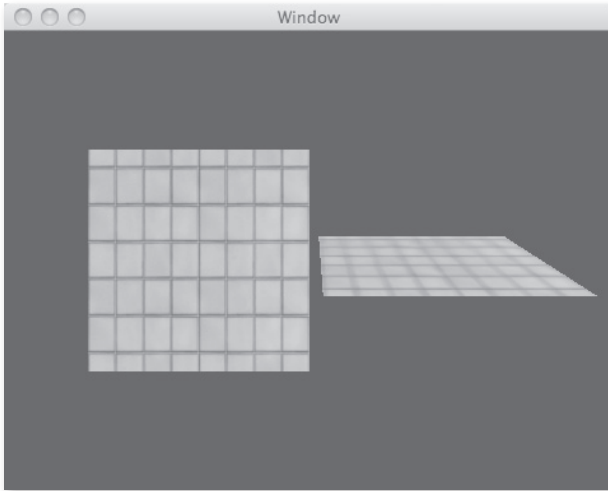
Dabei handelt sich um eine Eigenschaft des OpenGL Lichtmodels, welches wir ja in Kapitel 5 kennengelernt haben.

Abschalten lässt sich das wieder mit der Funktion (was übrigens das Standardverhalten ist)

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_COLOR_SINGLE);
```

Anisotropes Filtern

Normalerweise werden die Texturen in beide Richtungen gleichermaßen gefiltert (Isotrop), was unter Umständen zu sehr »verwaschenen« Ergebnissen führen kann. Das ist immer dann der Fall, wenn die Texturen nicht senkrecht zum Betrachter verlaufen und deshalb verzerrt werden müssen. Je schräger die Textur im Raum liegt, desto mehr leidet die Qualität der Ausgabe.



Nachdem die Textur rotiert wurde (rechts), wirkt sie sehr unscharf.

Um diesem Effekt entgegenzuwirken, bietet OpenGL einen sogenannten anisotropischen Filter, bei diesem ist der Filterkegel nicht mehr quadratisch, sondern eher rechteckig, dadurch werden weiter im Raum liegende Teile der Textur immer noch relativ scharf dargestellt.

Dieser Filter funktioniert mit allen Texturen, welche Mip-Maps enthalten, und ist über eine Erweiterung verfügbar, weshalb vor der Nutzung ein wenig Vorarbeit geleistet werden muss.

Zunächst muss überprüft werden, ob diese Erweiterung vom System unterstützt wird, was man mit folgender Funktion macht:

```
const GLubyte * strExtension = glGetString (GL_EXTENSIONS);  
gluCheckExtension ((const  
GLubyte*)"GL_EXT_texture_filter_anisotropic",strExtension);
```

Diese liefert dann im Erfolgsfall `GL_TRUE` zurück, und wir können den Filter verwenden. Mit der Funktion

```
GLfloat fLargest;  
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &fLargest);
```

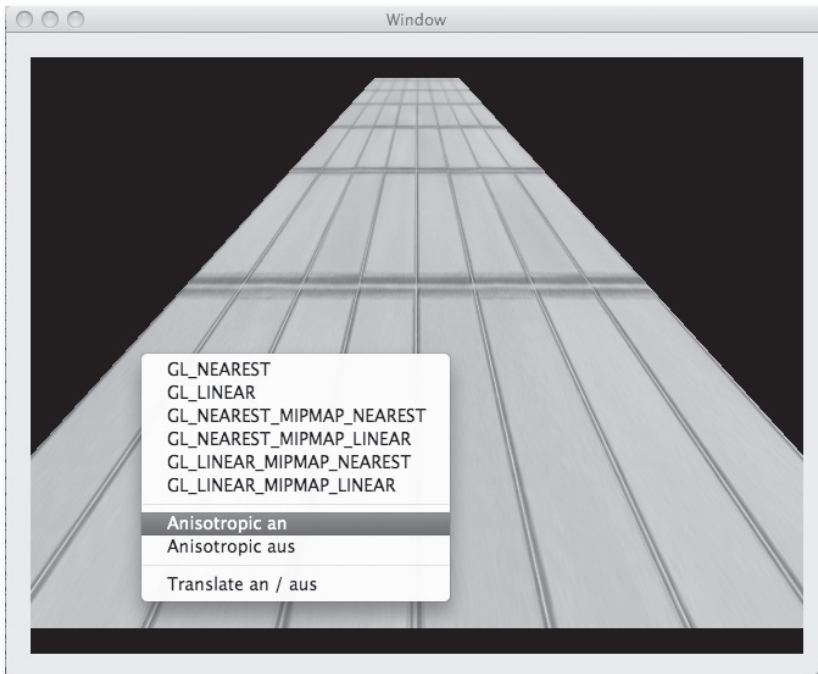
erfragen wir den maximalen Wert der Filterung und setzen diesen dann entsprechend mit folgender Funktion.

TIPP

Sie müssen nicht unbedingt den größten Wert für *fLargest* setzen. Unter Umständen kann ein kleinerer Wert auch schon gute Ergebnisse liefern.

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT,  
fLargest);
```

Je höher Sie den Wert für *fLargest* angeben, desto öfter wird ein Texel gefiltert, was natürlich auch Rechenzeit benötigt. Ein Wert von 1.0 entspricht dem Standard (Isotrop).



Beispielprogramm »Anisotropic Filter«

Im Beispielprogramm »Kapitel 7/Anisotropic Filter« können Sie über ein Kontextmenü zwischen den verschiedenen Filtern wählen und entsprechend die anisotropische Filterung dazuschalten.

Je nachdem, wie hoch der Wert für *fLargest* auf Ihrem System ist, sollten Sie einen deutlichen Qualitätsunterschied erkennen können.

Abschließend möchte ich noch anmerken, dass man sich bei relativ kleinen Objekten (z. B. Partikeln) diese Art von Filter getrost sparen kann, da man eine Verbesserung der Qualität sowieso nicht erkennen würde.

Textur-Transformation

Im Beispielprogramm »Kapitel 7/Anisotropic Filter« habe ich noch eine kleine Spielerei mit eingebaut, nämlich die, dass die Textur verschoben wird. Wie wir ja im Kapitel über Transformationen gelernt haben, kann man auch Texturen durch Matrizen transformieren, dazu muss lediglich die richtige Matrix gesetzt werden.

Schauen wir uns kurz den Code dazu noch an: Zunächst wird die Textur-Matrix aktiviert und mit *glLoadIdentity()* initialisiert, danach erfolgt die Verschiebung und zum Schluss wird wieder in die Modelviewmatrix zurückgeschaltet. Das ist schon alles.

```
glMatrixMode(GL_TEXTURE);  
glLoadIdentity();  
glTranslatef(0.0, _translation, 0.0);  
glMatrixMode( GL_MODELVIEW );
```

Sie können mit der Textur-Matrix genauso mit Stapeln (Push / Pop Matrix) arbeiten, wie Sie es aus den vergangenen Beispielen mit der Modelviewmatrix gesehen haben. Damit lassen sich sehr schöne Effekte erzielen.

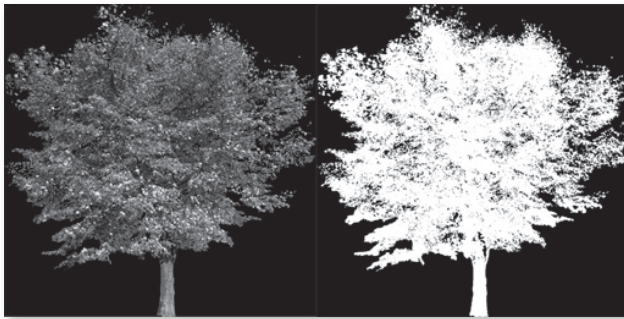
Alpha-Masking

Mit dem Alpha-Masking ist es möglich, **Teile einer Textur** transparent darzustellen. Diese Technik wird oft dazu verwendet, um Vegetationen in Outdoor-Levels darzustellen. Schauen wir uns dazu folgende Abbildung einmal an.



Das Beispielprogramm »Kapitel 7/Alpha-Masking«

Die Bäume, die Sie auf der Abbildung sehen können, bestehen allesamt aus nur einem Quadrat mit einer Textur darauf. Die Transparenz innerhalb dieser Texturen kommt aus einem Alpha-Kanal, welcher so aussieht:



Textur mit Alpha-Kanal

Alle weißen Bereiche des Alpha-Kanals werden später durch die tatsächliche Farbe der Textur ersetzt, alles, was schwarz ist, wird transparent.

Schauen wir uns nun den eigentlichen Vorgang an, der diese Maskierung vornimmt: Zuerst aktivieren wir den Alphatest mittels

```
//Alphatest einschalten  
glEnable(GL_ALPHA_TEST);
```


Danach müssen wir festlegen, was OpenGL mit den eingehenden Fragmenten tun soll.

Wir sagen hier einfach, alles, was einen Alphawert größer (*GL_GREATER*) 0.0 hat, soll gerendert werden, und alle anderen Fragmente (schwarz) sollen transparent dargestellt werden.

```
// Alles was groesser als 0.0, soll gerendert werden
glAlphaFunc(GL_GREATER, 0.0);
```

Mehr ist nicht nötig, um einen transparenten Bereich innerhalb einer Textur zu definieren. Ich habe bei dem Beispiel das Blending aktiviert, da es ohne Blending zu unschönen Rändern an der Textur kommen kann. Am besten deaktivieren Sie es einmal und schauen sich dann das Ergebnis an.

AUFGEPASST

Stellen Sie sicher, dass Sie die Textur im *GL_RGBA*-Format erstellen, sonst funktioniert der Alphatest natürlich nicht.

In dem Beispielprogramm habe ich noch eine passende Hintergrund-Textur und einen Schatten für die Bäume erstellt, um die Szene realistischer wirken zu lassen.

Bei den Schatten werden einfach nochmals die Bäume gerendert, allerdings werden sie diesmal um 90 Grad auf der Z-Achse rotiert und danach auf die XZ-Achse projiziert.

Die Grastextur wird 20 Mal über die ganze Fläche wiederholt, wie man an den UV-Koordinaten sehen kann.

```
glBindTexture(GL_TEXTURE_2D, _grass);
glBegin(GL_QUADS);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-20.0f, 0.0f, 10.0f);
    glTexCoord2f(20.0f, 0.0f); glVertex3f( 20.0f, 0.0f, 10.0f);
    glTexCoord2f(20.0f, 20.0f); glVertex3f( 20.0f, 0.0f,
        -10.0f);
    glTexCoord2f(0.0f, 20.0f); glVertex3f(-20.0f, 0.0f, -10.0f);
glEnd();
```

Der Rest des Codes sollte selbsterklärend sein, da sonst nichts Neues dazugekommen ist.

Multi-Texturing

OpenGL bietet die Möglichkeit, auch mehrere Texturen gleichzeitig auf ein Polygon zu mappen. Diese Technik nennt sich Multi-Texturing. Dabei wird pro Textur-Einheit eine Textur gebunden, welche völlig unabhängig von den anderen Textur-Einheiten behandelt wird. Nahezu alle heutigen Spiele verwenden diese Technik, da man damit unzählige Spezialeffekte setzen kann.



Teekanne mit 2 Texturen

Die Anwendung von Multi-Texturing ist recht simpel und geschieht im Großen und Ganzen in drei Schritten.

Testen, ob das System Multi-Texturing unterstützt

Die Funktion

```
GLuint units;  
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &units);
```

gibt einen Wert größer eins zurück (nämlich die Anzahl der zur Verfügung stehenden Textur-Einheiten), wenn das System in der Lage ist, Multi-Texturing durchzuführen.

Sollte ein System kein Multi-Texturing unterstützen (was sicher kaum noch vorkommt), kann man sich mit einer Technik behelfen, die sich Multipass-Multi-Texturing nennt. Diese funktioniert folgendermaßen:

- Rendere Geometrie mit der ersten Textur
- Rendere Geometrie nochmals mit der zweiten Textur und aktiviertem Blending

Wie man sich denken kann, ist diese Methode nicht die schnellste, da alles doppelt gerendert werden muss, aber immerhin ein Weg, um den gewünschten Effekt zu erzielen. Nun aber weiter im Text.

Textur-Einheit aktivieren

Die einzelnen Einheiten werden über die Funktion *glActiveTexture(GL_TEXTUREn)* aktiviert. Wobei *n* für die Textur-Einheit steht und stets mit Null beginnt.

Der Befehl

```
glActiveTexture(GL_TEXTURE1);
```

aktiviert somit die zweite Textur-Einheit.

GRUNDLAGEN

Alle Angaben, die wir für eine Textur vornehmen (Umgebung, Textur-Koordinaten usw.) beziehen sich immer auf die aktive Textur-Einheit, und zwar so lange, bis wir entweder auf eine andere Einheit umschalten, oder aber die Texturierung deaktivieren.

Nachdem die Textur-Einheit aktiviert wurde, muss die Texturierung für diese mittels *glEnable(GL_TEXTURE_2D)* aktiviert werden.

Hier nochmals die Schritte, die nötig wären, um die dritte Textur-Einheit zu aktivieren und die Textur-Umgebung auf *GL_MODULATE* zu setzen.

```
glActiveTexture(GL_TEXTURE2);  
glEnable(GL_TEXTURE_2D);  
glBindTexture(GL_TEXTURE_2D, _texture);  
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Textur-Koordinaten festlegen

Das Anlegen der Textur-Koordinaten geschieht fast genauso wie beim »normalen« Texturieren, wobei nun die Koordinaten nicht mehr mittels *glTexCoord2f()* übergeben werden, sondern über die Funktion

```
void glMultiTexCoord2f( GLenum target, GLfloat s, GLfloat t );
```

Der Parameter *target* steht für die Einheit, für welche wir die Koordinaten definieren möchten.

```
glMultiTexCoord2f( GL_TEXTURE1, 0.0, 1.0);
```

Die Texture-Koordinaten beziehen sich somit auf die Textur-Einheit 1.

AUFGEPASST

Achtung, auch beim Multi-Texturing müssen die Textur-Koordinaten vor den Vertex-Daten angegeben werden.

Folgender Code soll die Vorgehensweise nochmals verdeutlichen:

```
glBegin(GL_QUADS);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glMultiTexCoord2f(GL_TEXTURE0, 1.0f, 0.0f);
    glMultiTexCoord2f(GL_TEXTURE1, 1.0f, 0.0f);
    glVertex3f(0.5f, 0.5f, 0.5f);
    glMultiTexCoord2f(GL_TEXTURE0, 1.0f, 1.0f);
    glMultiTexCoord2f(GL_TEXTURE1, 1.0f, 1.0f);
    glVertex3f(0.5f, 0.5f, -0.5f);
    glMultiTexCoord2f(GL_TEXTURE0, 0.0f, 1.0f);
    glMultiTexCoord2f(GL_TEXTURE1, 0.0f, 1.0f);
    glVertex3f(-0.5f, 0.5f, -0.5f);
    glMultiTexCoord2f(GL_TEXTURE0, 0.0f, 0.0f);
    glMultiTexCoord2f(GL_TEXTURE1, 0.0f, 0.0f);
    glVertex3f(-0.5f, 0.5f, 0.5f);
glEnd();
```

Wenn Sie aus Versehen die Textur-Koordinaten mit der Funktion *glTexCoord2f(...)* übergeben, hat dies denselben Effekt, als würden Sie schreiben:

```
glMultiTexCoord2f(GL_TEXTURE0, ...);
```

Sie definieren also die Textur-Koordinaten nur für die erste Textur-Einheit.

Nun ist es wieder Zeit für etwas Praktisches. Das Beispielprogramm »Kapitel 7/ Multi-Texturing« zeigt nochmals das Terrain aus Kapitel 3, dieses Mal aber nicht einfarbig, sondern mit Hilfe zweier übereinander liegenden Texturen.



Terrain mit 2 übereinander liegenden Texturen

Zunächst wird ein Hintergrundbild gerendert, um die Szene etwas realistischer wirken zu lassen.

```
glBindTexture(GL_TEXTURE_2D, _background);
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( -40.0f,  0.0f,
0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 157.0f, 0.0f,
0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 157.0f, 70.0f,
0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( -40.0f,  70.0f,
0.0f);
glEnd();
```

POWER

Wenn Sie möchten, können Sie auch das Hintergrundbild transformieren, damit es so aussieht, als würden sich die Wolken bewegen.

Wie Sie sehen, ist hier nichts Neues hinzugekommen, da noch kein Multi-Texturing verwendet wird. Anschließend folgt das Terrain, welches ja mit 2 Texturen gerendert werden soll. Die Textur-Einheit eins soll die Gras-Textur beinhalten:

```
//Basistextur an Textur-Einheit 1 binden
glActiveTexture( GL_TEXTURE1 );
glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D, _base );
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

Dazu wird zuerst die Textur-Einheit 1 aktiviert und die Texturierung dafür eingeschaltet, danach wird die Gras-Textur gebunden und die Textur-Umgebung auf *GL_REPLACE* gesetzt, da die Textur die Farbe des Terrains ersetzen soll. Das Selbe passiert für die Textur-Einheit zwei:

```
//Detailtextur an Textur-Einheit 2 binden
glActiveTexture( GL_TEXTURE2 );
glEnable( GL_TEXTURE_2D );
glBindTexture( GL_TEXTURE_2D, _detail );
glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Die Textur-Umgebung wird nun aber auf *GL_MODULATE* gesetzt, da die Detail-Textur mit der darunter liegenden Gras-Textur multipliziert werden soll. Anschließend wird das Terrain gerendert:

```
for( z=0; z<_mapSize-1; z++ )
{
    glBegin( GL_TRIANGLE_STRIP );
    for( x=0; x<_mapSize-1; x++ )
    {
        //Texture-Koordinaten berechnen
        left  = ( float )x/_mapSize;
        bottom= ( float )z/_mapSize;
        top   = ( float )( z+1 )/_mapSize;
        glMultiTexCoord2f( GL_TEXTURE1, left, bottom );
        if(_showDetailTexture)
            glMultiTexCoord2f( GL_TEXTURE2, left*repeatDetailMap,
                bottom*repeatDetailMap );
        glVertex3f( ( float )x,
            [self scaledHeightAtPosition:x zPosition:z],
            ( float )z );
        glMultiTexCoord2f( GL_TEXTURE1, left, top );
    }
}
```

```

        if(_showDetailTexture)
            glMultiTexCoord2f( GL_TEXTURE2, left*repeatDetailMap,
            top*repeatDetailMap );
            glVertex3f( ( float )x,
[self scaledHeightAtPosition:x zPosition:z+1],
( float )z+1 );
        }
        glEnd( );
    }
}

```

Die Detail-Textur wird dabei um den Faktor *repeatDetailMap* über das Terrain gekachelt.

Über die Leertaste können Sie diese Detail-Textur ein- bzw. ausschalten. Wenn sie ausgeschaltet ist, werden Sie feststellen, dass die Gras-Textur ziemlich verwaschen aussieht, das liegt daran, dass die Textur mit 1024 x 1024 Pixeln recht klein im Vergleich zum Terrain ist, weshalb man sehr oft die Technik mit der gekachelten Detail-Textur verwendet (Detail-Mapping), um das Ganze ein wenig detaillierter aussehen zu lassen.

Zum Schluss müssen die beiden Textur-Einheiten wieder deaktiviert werden, um auf das normale Texturing zurückzuschalten (wegen des Hintergrund-Bildes):

```

if(_showDetailTexture)
{
    glActiveTexture( GL_TEXTURE2 );
    glDisable( GL_TEXTURE_2D );
}
glActiveTexture( GL_TEXTURE1 );
glDisable( GL_TEXTURE_2D );

```

Wie Sie sehen, kann man mit dieser Technik ohne großen Aufwand schon recht ansehnliche Dinge machen.

Da das Laden und Verwenden von Texturen ständig benötigt wird, habe ich daraus zwei Klassen gebaut, die uns die meiste Arbeit dazu abnehmen sollen.

Die beiden Klassen wären

- CFXTextureManager-Singleton-Klasse, welche die Texturen verwaltet
- CFXTextureObject, eine einfach Textur

Eine Textur wird über die Methode

```
(GLuint) textureByName:(NSString *)textureName
```

des Textur-Managers erstellt. Sollte die Textur noch nicht vorhanden sein, wird sie erstellt und im Erfolgsfall zurückgegeben. Wenn es die Textur schon gibt, wird die bereits existierende zurückgegeben. Der Rest der Methoden sollte selbsterklärend sein, da alle hier im Kapitel erläutert wurden. In den weiteren Kapiteln werden Sie die Handhabung der beiden Klassen noch genauer kennenlernen.

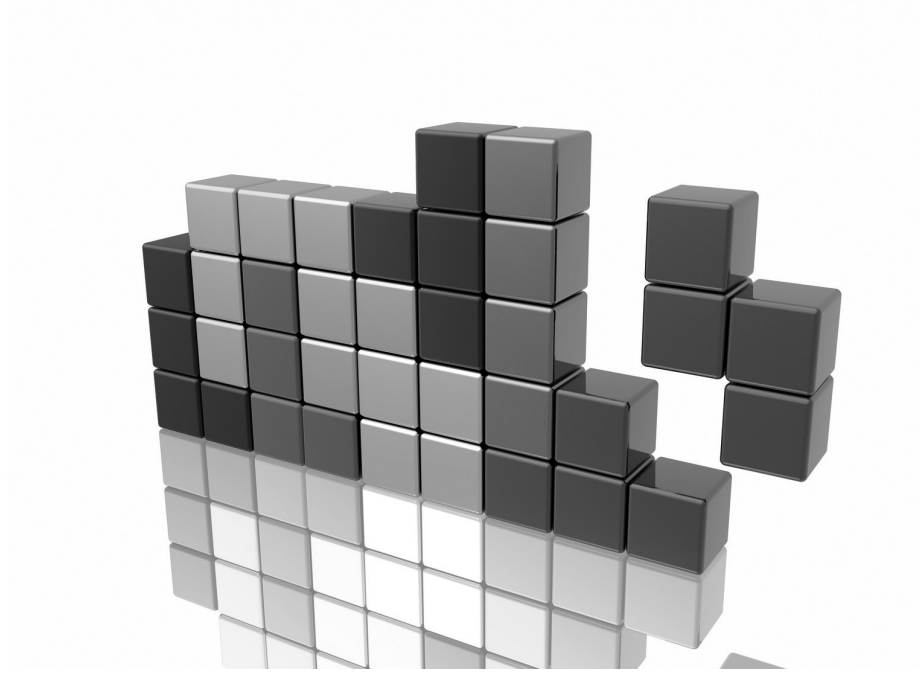
Zusammenfassung

In diesem Kapitel haben wir doch eine Menge gelernt. Sie wissen nun, wie man Texturen auf Polygone »klebt«. Weiterhin haben wir uns die verschiedenen Texturumgebungen und Wicklungsfunktionen angeschaut. Sie wissen nun auch, was sich hinter dem Begriff »Mip-Maps« verbirgt und wie man diese zur Textur-Filterung gebrauchen kann. Zum Schluss haben wir noch gesehen, wie sich mehrere Texturen gleichzeitig auf Polygone mappen lassen.

Zusätzliche Informationen

<http://www.glprogramming.com/red/chapter09.html>
Sehr ausführliche Erklärung aus dem »RedBook«

http://www.gamasutra.com/features/19990723/opengl_texture_objects_01.htm
Artikel zum Thema »Texturierung« auf Gamasutra.com, von Richard S. Wright Jr, dem Autor der OpenGL SuperBible



**Rendervorgang
beschleunigen**

Rendervorgang beschleunigen

Als Entwickler von Grafikprogrammen sollten Sie stets ein Auge auf die erzielten FPS werfen. Nichts ist für den Spieler frustrierender, als dass ein Spiel im Daumenkinotempo über den Bildschirm ruckelt.

Das stellt uns vor ein Problem; Mit steigender Komplexität der Spiele steigt natürlich auch der Rechenaufwand. Die Hardware wird zwar immer leistungsfähiger, aber nach wie vor ist einer der größten Flaschenhälse die grafische Ausgabe. Glücklicherweise bietet OpenGL einige Mechanismen, die den Rendervorgang erheblich beschleunigen können. Die wollen wir uns in diesem Kapitel anschauen.

Bei den bisherigen Beispielen war der Rendervorgang noch recht überschaubar, das ändert sich aber schlagartig, wenn wir mehrere hundert oder gar tausend Objekte rendern möchten. Hier ist eine gut durchdachte Datenorganisation unabdingbar, und diese beginnt damit, dass wir, wann immer es geht, eine Redundanz vermeiden müssen.

Stellen wir uns ein einfaches Beispiel vor, in welchem wir z. B. einhundert Würfel rendern möchten. Ein Ansatz wäre, dass wir das Rendern des Würfels in eine Funktion auslagern und diese dann innerhalb einer Schleife 100 Mal aufrufen. Wenn wir uns jetzt noch vorstellen, dass die Würfel allesamt gleich aussehen und nur eine andere Position innerhalb der Szene haben, könnte man doch auf den Gedanken kommen, dass es hier eine bessere Lösung geben muss.

Genau diese gibt es. Sie nennt sich in OpenGL »Display-Lists«, diese Listen wollen wir uns nun anschauen.

Display-Lists

Eine Display-List kann man sich vorstellen wie eine Abfolge von OpenGL-Befehlen, die sich ständig wiederholen (so eine Art Makro also). Diese Technik eignet sich nun hervorragend für das oben genannte Szenario. Wir benötigen nun keine Schleife mehr, in der immer dieselben Befehle ausgeführt werden, sondern rufen immer nur die Liste auf und haben dadurch das Problem der Redundanz gelöst.

Display-Lists erstellen

Bevor wir nun aber eine Display-List mit irgendetwas füllen können, muss man sie zuerst einmal erzeugen, was man mit der Funktion `glGenLists(...)` macht.

```
GLuint glGenLists(GLsizei range);
```

Die Funktion erwartet die Anzahl der Listen, die wir erzeugen möchten, und gibt uns im Erfolgsfall eine eindeutige ID (größer Null) der ersten Display-List zurück. Wenn wir auf die zweite Liste zugreifen möchten, addieren wir einfach eine 1 zur ID hinzu, bei der dritten Liste eine 2 usw.

Mit der Funktion

```
GLboolean glIsList(GLuint listName);
```

können Sie abfragen, ob eine Display-List eine gültige ID besitzt. Wenn *listName* eine gültige ID ist, liefert die Funktion *GL_TRUE* zurück.

Display-Lists mit Daten füttern

Nachdem wir die Liste nun angelegt haben, können wir sie mit Befehlen füttern. Damit OpenGL weiß, was zu einer Liste gehört, müssen wir zunächst den Anfang der Liste definieren, was die Funktion

```
void glNewList(GLuint listName, GLenum mode);
```

macht.

Als ersten Parameter tragen wir die Liste ein, die wir mit Befehlen füllen möchten. Dies kann eine neue, leere Liste sein, die wir zuvor angelegt haben, oder aber eine bereits bestehende, welche dann mit den neuen Befehlen überschrieben wird.

Der zweite Parameter ist entweder *GL_COMPILE*, wobei die Liste nur kompiliert wird, und OpenGL sich zunächst einmal nur die Kommandos merkt. Die Alternative ist *GL_COMPILE_AND_EXECUTE*, was zur Folge hat, dass dabei die Liste kompiliert und sofort ausgeführt wird. Meistens ist es aber so, dass man zunächst die Liste nur anlegt (*GL_COMPILE*) und zu einem späteren Zeitpunkt ausführt.

Wenn wir die Liste fertig definiert haben, müssen wir sie mit dem Befehl *glEndList()* abschließen.

Bei der Definition der OpenGL-Befehle innerhalb einer Liste gibt es einige Einschränkungen, die darin bestehen, dass folgende Befehle **nicht** in eine Display-List »verpackt« werden dürfen:

- `glColorPointer`
- `glDeleteLists`

- `glDisableClientState`
- `glEdgeFlagPointer`
- `glEnableClientState`
- `glFeedbackBuffer`
- `glFinish`
- `glFlush`
- `glGenLists`
- `glIndexPointer`
- `glInterleavedArrays`
- `glIsEnabled`
- `glIsList`
- `glNormalPointer`
- `glPopClientAttrib`
- `glPixelStore`
- `glPushClientAttrib`
- `glReadPixels`
- `glRenderMode`
- `glSelectBuffer`
- `glTexCoordPointer`
- `glVertexPointer`
- alle `glGet` Routinen.

Display-Lists ausführen

Nachdem die Liste nun angelegt wurde, kann man sie ganz einfach mit der Funktion

```
glCallList(GLuint listName);
```

aufrufen. Die Befehle innerhalb der Liste werden dann in der Reihenfolge ausgeführt, in der sie definiert wurden. Der Vollständigkeit halber sei noch auf die eher selten genutzte Funktion

```
glCallLists(GLsizei num, GLenum type, const void *lists);
```

hingewiesen, die nichts anderes tut, als alle angegebenen Listen nacheinander aufzurufen.

Display-Lists löschen

Display-Lists liegen direkt im Grafikkarten-Speicher. Da dieser sehr begrenzt ist, sollte man die Listen, die man nicht mehr benötigt, aus dem Speicher löschen, was die Funktion

```
glDeleteLists(GLuint listName, GLsizei range);
```

macht. Der erste Parameter ist wieder die ID der Liste und der zweite die Anzahl der Listen, die gelöscht werden sollen.

Nach all der Theorie nun ein Beispiel-Programm »Kapitel 8/Quads-Display-Lists«, welches die Verwendung der Listen zeigen soll.

Dazu habe ich das Beispielprogramm »Quads« aus Kapitel 3 genommen und es ein wenig modifiziert. Anstatt nun die Geometrie jedes Mal neu zu definieren (*glBegin(...)/glEnd()*), habe ich das Ganze in zwei Listen verpackt.

Schauen wir uns einmal die relevanten Änderungen an. In der *prepareOpenGL*-Methode werden zunächst zwei leere Listen erstellt:

```
_displayList1 = glGenLists(2); // Erzeuge 2 Display Listen
```

Diese beiden werden dann in der Methode *generateDisplayLists* mit Daten gefüllt.

```
glNewList(_displayList1, GL_COMPILE);
// Definiere einen Wuerfel
glBegin(GL_QUADS);
    // Oben
    glColor3f(0.0f,0.0f,0.0f);glVertex3f( 1.0f, 1.0f,-1.0f);
    glColor3f(0.0f,0.0f,1.0f);glVertex3f(-1.0f, 1.0f,-1.0f);
    glColor3f(0.0f,1.0f,1.0f);glVertex3f(-1.0f, 1.0f, 1.0f);
    glColor3f(0.0f,1.0f,0.0f);glVertex3f( 1.0f, 1.0f, 1.0f);
    .
    .
glEndList();
_displayList2 = _displayList1+1;
glNewList(_displayList2, GL_COMPILE);
glDisable(GL_CULL_FACE);
```

```

glPushMatrix();
glTranslatef(-1,0,-10);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
// Buchstabe G
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_POLYGON);
    glVertex3f(0.85, 0, 0);
.
.
.
glEndList();

```

Der Rendervorgang hat sich auch nur geringfügig geändert.

```

.
.
glPushMatrix();
glRotatef(-_rotation, 0.0f, 0.0f, 1.0f);
glTranslatef(x, y, -50.0);
glRotatef(_rotation, 1.0f, 1.0f, 1.0f);
// Jeden zweiten Wuerfel im Wireframe-Modus zeichnen
if(i%2 == 0)
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
else
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glCallList(_displayList1);
glPopMatrix();
// Buchstaben zeichnen
glCallList(_displayList2);
.
.

```

Wie Sie sehen, ist die Handhabung recht simpel, am Programm selbst hat sich nur sehr wenig bzw. gar nichts geändert. Es sind lediglich die Befehle für die Display-Listen hinzugekommen.

Auf meinem System ist der Leistungszuwachs durch die Verwendung der Listen etwa um zehn FPS gestiegen. Das hört sich zwar nach nicht viel an, ändert sich aber gerade bei komplexer Geometrie unter Umständen sehr schnell. Man kann hier aber leider keine genauen Angaben machen, inwiefern die erzielten Frames in die Höhe gehen. Der Geschwindigkeitsvorteil einer Display-List hängt nämlich sehr stark von der OpenGL-Implementierung ab.

GRUNDLAGEN

Nachfolgend noch einige Dinge, die Sie bei der Verwendung von Display-Listen beachten sollten:

- Display-Listen sind statisch, das heißt, wenn man sie einmal angelegt hat, kann man sie nicht mehr ändern.
- Display-Listen brauchen recht viel Speicher, da eine Kopie aller Daten erstellt wird.
- Befehle innerhalb einer Liste können nicht abgefragt werden, das bedeutet, man kann nicht feststellen, ob z. B. die Beleuchtung eingeschaltet ist.

Im Kapitel über die Textausgabe werde ich Ihnen nochmals die Verwendung von Display-Listen an einem etwas komplexeren Beispiel zeigen.

Vertex-Arrays

Vertex-Arrays sind eine weitere Technik, mit der man den Rendervorgang erheblich beschleunigen kann.

Anders als bei den Display-Listen, wo der Render-Ablauf ja immer statisch ist, kann man die Daten eines Vertex-Arrays beliebig modifizieren, was sie natürlich universell einsetzbar macht. In diesem Abschnitt möchten wir uns diese Art von Arrays einmal genauer ansehen.

In allen bisher gezeigten Beispielen wurde die Geometrie ja immer mit Hilfe der *glBegin()* / *glEnd()*-Funktionen gerendert, was ungefähr so aussah:

```
glBegin(GL_QUADS);  
glColor3f(0.0f,0.0f,0.0f);glVertex3f( 1.0f, 1.0f,-1.0f);  
glColor3f(0.0f,0.0f,1.0f);glVertex3f(-1.0f, 1.0f,-1.0f);  
.  
.  
.  
glEnd();
```

Bei Objekten, die aus einer Handvoll Polygonen bestehen, ist es natürlich kein Problem, die Daten wie hier gezeigt zu rendern. Ganz anders sieht es schon aus, wenn wir Models rendern, die aus Tausenden von Dreiecken bestehen.

Machen wir dazu mal ein kleines Rechenbeispiel. Wir haben eine Spielfigur, die aus 1500 Dreiecken besteht. Natürlich benötigen wir auch Normale und Textur-Koordinaten, alles zusammen soll über `GL_TRIANGLES` gerendert werden.

Das ergibt 4500 (ein Dreieck = drei Vertices) Mal den Aufruf von `glVertex3f()`, `glTexCoord2f()`, `glNormal3f()`.

Schon alleine die Anzahl der Funktionsaufrufe lässt erahnen, dass es nicht die schnellste Technik ist, um Daten an die Grafikkarte zu senden.

Mit der Verwendung von Vertex-Arrays lassen sich die Funktionsaufrufe auf ein Minimum reduzieren.

Diese Vertex-Arrays sind, wie der Name schon vermuten lässt, Arrays, die unsere Rohdaten (Vertices, Textur-Koordinaten usw.) in einer besonderen Form bereitstellen, damit sie von OpenGL effizienter gerendert werden können.

Die Verwendung dieser Arrays ist gerade am Anfang ein wenig gewöhnungsbedürftig, da sie sich doch komplett von der Art und Weise unterscheidet, in der wir bisher die grafische Ausgabe gemacht haben. Aber keine Sorge, mit ein wenig Übung und den folgenden Beispielen geht die Handhabung in Fleisch und Blut über.

Vertex-Arrays benutzen

Die Nutzung von Vertex-Arrays erfolgt in vier Schritten.

1. Die Rohdaten müssen so aufbereitet werden, dass sie in einem oder mehreren Arrays gespeichert werden, dabei lassen sich die Daten zur Laufzeit berechnen oder aber von einem Datenträger geladen werden.
2. Wir müssen OpenGL mitteilen, wo sich die Daten befinden.
3. Wir müssen festlegen, welche Arrays wir nutzen wollen. Wie bereits gesagt, ist es möglich, die Rohdaten auf mehrere Arrays zu verteilen.
4. Wir geben den Befehl, dass die Geometrie gerendert werden soll.

Ein einfaches Beispiel

Auf an die Arbeit. Unser erstes Beispiel »Kapitel 8/Starfield« ist noch recht einfach gehalten und macht nicht mehr Mühe, als in paar Punkte zu zeichnen.

Zunächst einmal wird eine Struktur definiert, welche die Position der einzelnen Punkte speichern soll, dies ist kein Muss, fördert aber die Lesbarkeit ungemein.


```
typedef struct _Stars
{
    float x;
    float y;
    float z;
}Stars;
```

Danach werden in der *prepareOpenGL*-Methode 1000 dieser Punkte mit zufällig gewählten Positionen erstellt.

```
int i;
for(i=0; i<1000; i++)
{
    _stars[i].x = randomFloat(-10.0, 10.0);
    _stars[i].y = randomFloat(-10.0, 10.0);
    _stars[i].z = randomFloat(-10.0, 10.0);
}
```

Das eigentlich Wichtige an unserem Beispiel folgt dann in der *drawRect*-Methode. Die Funktion

```
glEnableClientState(GL_VERTEX_ARRAY);
```

bereitet das System für die Verwendung von Vertex-Arrays vor.

Zunächst erfolgt die Übergabe der Rohdaten:

```
glVertexPointer(3, GL_FLOAT, 0, _stars);
```

Diese Parameter sind:

- 3 für die XYZ-Position
- *GL_FLOAT* ist der Datentyp (float)
- 0 der Stride zwischen den einzelnen Daten (dazu später mehr)
- *_stars* der Zeiger auf die Rohdaten

Und zum Schluss werden alle 1000 Punkte mit einem einzigen Aufruf von

```
glDrawArrays(GL_POINTS, 0, 1000);
```

gerendert.

Die Parameter hierfür sind:

- `GL_POINTS` für den Primitiv-Typ (`GL_TRIANGLES` usw.)
- 0 das erste Element im Array
- 1000 die Anzahl der Elemente im Array

Abgeschlossen wird der Rendervorgang mit dem Aufruf von

```
glDisableClientState(GL_VERTEX_ARRAY);
```

welcher die Verwendung von Vertex-Arrays wieder deaktiviert.

Wie Sie sehen, entfällt nun der `glBegin() / glEnd()`-Block komplett. OpenGL rendert die Geometrie jetzt in einem Rutsch, was natürlich sehr viel schneller geht. Zugegeben, das Beispiel ist doch recht simpel, aber an einfach gehaltenen Beispielen lässt sich die Funktionsweise sehr viel besser nachvollziehen.

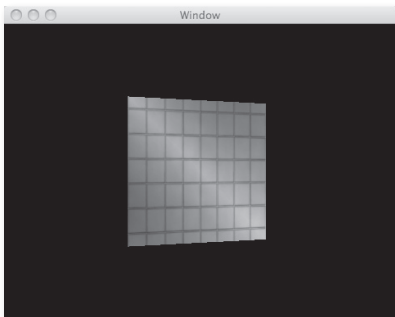
TIPP

Bevor wir zum nächsten Beispiel kommen, noch ein paar Worte zu den beiden `glBegin() / glEnd()`-Funktionen: Die Technik, mit der man Primitive über diese beiden Funktionen definiert, nennt sich »Immediate-Mode« und wird gerade von Einsteigern sehr gerne verwendet. Leider werden die beiden in der nächsten Version von OpenGL nicht mehr dabei sein, weshalb man sich die Verwendung am besten gleich wieder abgewöhnt und stattdessen auf Vertex-Arrays oder andere Techniken setzen sollte.

So, nun aber weiter im Text.

Vertex-Arrays mit Textur

Das nächste Beispiel, »Kapitel 8/Simple VertexArray«, ist auch wieder relativ simpel gehalten und macht nicht mehr als ein texturiertes Quadrat auszugeben.



Ein Quadrat, das mit Hilfe von Vertex-Array gerendert wird

Bevor wir zum Code kommen, ein Hinweis:

Für das Laden der Texturen verwende ich nun die beiden Klassen, die ich im vergangenen Kapitel angesprochen habe (CFXTextureObject, CFXTextureManager), damit Sie die Funktionsweise auch gleich kennenlernen.

Jetzt aber zum Beispiel-Code: Zuerst wird wieder eine Struktur definiert, welche unsere Vertices speichern soll.

```
typedef struct _Vertex
{
    CFXVector  position;
    CFXVector  color;
    CFXVector  texCoords;
}Vertex;
```

Ein Vertex hat also eine Position, eine Farbe und einen Satz Textur-Koordinaten.

Die Textur, die wir benötigen, laden wir dieses Mal über den Textur-Manager

```
_texture = [[CFXTextureManager sharedManager]textureByName:[NS-
Bundle mainBundle] pathForResource:@"texture2" ofType:@"tif"]];
```

Die Handhabung ist recht einfach, man muss lediglich Dateinamen und Pfad angeben, um eine Textur zu laden und zu erstellen.

Anschließend füllen wir unser Vertex-Array mit Daten:

```
// Vertex Array mit Daten füllen
_vertices[0].position = makeVector(-1, -1, 0);
_vertices[0].color = makeVector(1.0, 0.0, 0.0);
_vertices[0].texCoords = makeVector(0.0, 0.0, 0.0);
_vertices[1].position = makeVector(1, -1, 0);
_vertices[1].color = makeVector(1.0, 1.0, 0.0);
_vertices[1].texCoords = makeVector(1.0, 0.0, 0.0);
_vertices[2].position = makeVector(1, 1, 0);
_vertices[2].color = makeVector(1.0, 0.0, 1.0);
_vertices[2].texCoords = makeVector(1.0, 1.0, 0.0);
_vertices[3].position = makeVector(-1, 1, 0);
_vertices[3].color = makeVector(0.0, 1.0, 1.0);
_vertices[3].texCoords = makeVector(0.0, 1.0, 0.0);
```

TIPP

Die Funktion *makeVector(...)* kommt aus der Datei »CFXVector.h« und erstellt einen Vektor mit den übergebenen Werten.

Wie Sie sehen, wird für jeden Eckpunkt ein Satz Koordinaten, eine Farbe und ein Satz Textur-Koordinaten angegeben. Dann folgt der Rendervorgang. Da wir nun nicht mehr nur Vertex-Daten haben, sondern auch Farben und Textur-Koordinaten, müssen wir die Ausgabe entsprechend vorbereiten. Das sieht dann folgendermaßen aus:

```
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, sizeof(Vertex), &_vertices[0].
position);
glEnableClientState(GL_COLOR_ARRAY);
glColorPointer(3, GL_FLOAT, sizeof(Vertex), &_vertices[0].color);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, GL_FLOAT, sizeof(Vertex), &_vertices[0].
texCoords);
```

Wir aktivieren mit der Funktion *glEnableClientState(...)* das entsprechende Array, welches wir verwenden möchten, und übergeben dann die Daten mit Hilfe der Funktionen *glVertexPointer(...)*, *glColorPointer(...)*, *glTexCoordPointer(...)* ein.

Die Parameter für diese Funktionen sind folgende:

- 3 bzw. 2: die Anzahl der Daten
- *GL_FLOAT*: der Datentyp, den wir übergeben
- *sizeof(Vertex)*: Der Stride (zur Erklärung siehe weiter unten in diesem Kapitel) zwischen den einzelnen Datensätzen
- *&_vertices[0].position*: die eigentlichen Rohdaten
- *&_vertices[0].color*
- *&_vertices[0].texCoords*

Danach folgt wieder der Befehl zum Rendern:

```
glDrawArrays(GL_QUADS, 0, 4);
```

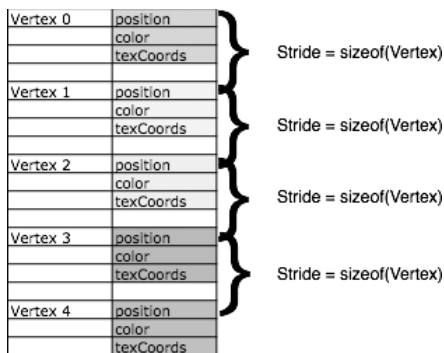
Was wieder bedeutet; Zeichne ein Viereck, beginnend am Anfang des Arrays mit der Größe von vier Vertices!

Und zum Schluss werden die drei Arrays wieder deaktiviert.

```
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

Der Stride-Parameter

Die folgende Abbildung soll einmal verdeutlichen, was es mit dem Stride-Parameter auf sich hat. Sie zeigt, wie die Vertex-Daten innerhalb des Arrays gespeichert wurden.



Anordnung der Daten innerhalb des Arrays

Wenn man nun z. B. nur die Vertex-Positionen haben möchte, muss man einfach nur die Größe eines Vertex (sizeof(Vertex)) zur aktuellen Position im Array hinzuzählen und schon hat man den Stride (Offset) errechnet.

Wie man sieht, habe ich alle Vertex-Daten in ein einziges Array verpackt. Es ist auch möglich, für die jeweiligen Rohdaten ein separates Array zu erstellen und den Pointer entsprechend auf dieses Arrays zu setzen. Allerdings wäre das nicht sonderlich effizient, da dann drei Arrays zur Grafikkarte transportiert werden müssten.

Indexierte Vertex-Arrays

In der Einleitung dieses Kapitels habe ich davon gesprochen, dass, wann immer es geht, eine Daten-Redundanz vermieden werden sollte. Davon war bis jetzt noch nicht viel zu sehen, da wir mitunter mehrfach genutzte Vertices auch mehr als ein Mal definiert haben. Schauen wir uns zunächst folgende Abbildung an.

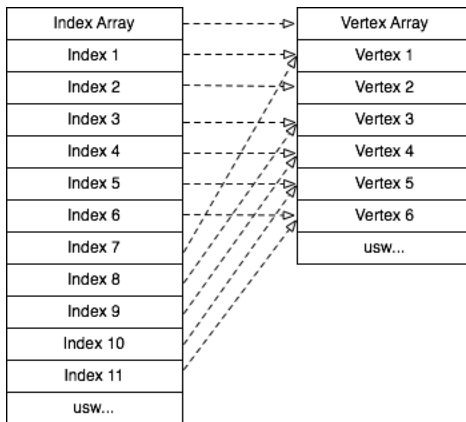


Gemeinsam genutzte Eckpunkte zweier Triangle-Strips

Wie man unschwer erkennen kann, werden die mittleren Eckpunkte von beiden Triangle-Strips verwendet und dadurch zweimal definiert, wodurch die besagte Redundanz entsteht.

Eine bessere Lösung wäre aber, die Eckpunkte, die mehrfach genutzt werden, nur ein Mal zu definieren und sie dann immer wieder zu verwenden. Genau diesen Ansatz verfolgen wir mit den indexierten Vertex-Arrays. Dabei wird das Array nun nicht mehr von Anfang bis zum Ende durchlaufen, sondern mit Hilfe eines zweiten Arrays (Index) wird eine Reihenfolge festgelegt, die den Ablauf des Rendervorgangs bestimmt.

Folgende Abbildung soll die Technik verdeutlichen.

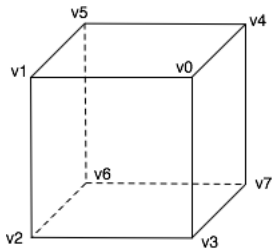


Aufbau eines indexierten Vertex-Arrays

Auf der rechten Seite haben wir das »normale« Vertex-Array mit den Rohdaten, und auf der linken Seite stehen die Indexe (die Verweise), die nun auf die Rohdaten zeigen und somit festlegen, welcher Eckpunkt wann und wie oft gerendert werden soll, was unser Problem mit der Redundanz löst.

Wir wollen uns im folgenden Beispiel die Funktionsweise einmal anschauen. Dazu nehmen wir einen simplen Würfel und rendern diesen mit Hilfe eines indexierten Vertex-Arrays.

Auf der folgenden Abbildung können Sie sehen, dass wir lediglich acht (anstatt 24) Eckpunkte benötigen, um diesen Würfel zu erstellen.



Ein Würfel mit gemeinsam genutzten Eckpunkten

Im Beispiel »Kapitel 8/Indexed VertexArray« erstellen wir zuerst die beiden Arrays, welche sich in der Datei »cube.model« befinden.

```
// Vertex-Daten
GLfloat _vertices[] = {
1.0f, 1.0f, 1.0f,
-1.0f, 1.0f, 1.0f,
-1.0f, -1.0f, 1.0f,
1.0f, -1.0f, 1.0f,    // v0 v1 v2 v3
1.0f, 1.0f, -1.0f,
-1.0f, 1.0f, -1.0f,
-1.0f, -1.0f, -1.0f,
1.0f, -1.0f, -1.0f}; // v4 v5 v6 v7
// Indexe
GLubyte _indices[] = {
0,1,2,3,    // vorne
0,3,7,4,    // rechts
4,5,6,7,    // hinten
1,2,6,5,    // links
0,4,5,1,    // oben
3,7,6,2};   // unten
```

HILFE

Um einen Bezug zu den Indices zu bekommen, vergleichen Sie einfach mal die Werte in *indices* mit obiger Abbildung.

Nachdem nun unsere beiden Arrays definiert sind, können wir sie schon rendern. Der Vertex-Pointer selbst ändert sich dabei nicht, lediglich der Würfel wird nun mit einer neuen Funktion gerendert.

```
glEnableClientState(GL_VERTEX_ARRAY);  
glVertexPointer(3, GL_FLOAT, 0, _vertices);  
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, _indices);  
glDisableClientState(GL_VERTEX_ARRAY);
```

Die Funktion *glDrawElements(...)* erwartet folgende Parameter:

- *GL_QUADS*. Primitiv-Typ
- 24. Anzahl der Indexe
- *GL_UNSIGNED_BYTE*. Datentyp der Indexe
- *_indices*. Zeiger auf die Indexe

Wie Sie sehen, ist diese Technik nicht sonderlich schwierig zu verstehen, allerdings hat sie auch einen Haken. Wenn man nämlich zu den reinen Vertex-Daten noch die Normale- und Textur-Koordinaten dazu nimmt, wird es unter Umständen schwierig, die Daten so zu sortieren, dass die Indices auf alle Rohdaten gleichzeitig passen, da man lediglich einen einzigen Index-Array pro Geometrie übergeben kann.

Die Königsklasse Vertex-Buffer-Objects (VBOs)

VBOs verbinden den Vorteil von Display-Listen (Daten liegen im Grafikkarten-Speicher) mit der »Dynamik« von Vertex-Arrays und machen sie dadurch zur allerersten Wahl, wenn es darum geht, Objekte mit vielen Polygonen zu rendern. VBOs werden im Prinzip so verwendet wie Vertex-Arrays auch und benötigen lediglich drei zusätzliche Schritte, die wir uns nun anschauen werden.

1. Puffer erzeugen

Zuerst erzeugen wir mit Hilfe der Funktion

```
glGenBuffers(GLsizei n, GLuint *buffer);
```

einen Puffer, welcher später unsere Daten aus dem Array speichern soll. Das Besondere daran ist, dass die Daten dann direkt im Speicher der Grafikkarte liegen und somit sehr schnell verfügbar sind.

Diese Funktion erwartet als ersten Parameter die Anzahl der Puffer, die wir benötigen, und als zweiten Parameter eine eindeutige ID.

2. Puffer binden

Wie beim Erzeugen von Texturen auch, müssen wir, bevor wir Daten in den Puffer schreiben können, diesen zuerst binden (aktivieren), was die Funktion

```
glBindBuffer(GLenum target, GLuint buffer);
```

übernimmt.

Der erste Parameter steht für die Art des Arrays, welches wir binden möchten, und kann entweder *GL_ARRAY_BUFFER* (Rohdaten) oder *GL_ELEMENT_ARRAY_BUFFER* (Indices) sein.

Der zweite Parameter ist der Puffer, den wir füllen möchten.

3. Puffer mit Daten füllen

Im letzten Schritt befüllen wir mit der Funktion

```
glBufferData(GLenum target, GLsizeiptr size, GLvoid *data, GLenum usage);
```

den Puffer mit den entsprechenden Daten.

- *target* ist entweder wieder *GL_ARRAY_BUFFER* oder aber *GL_ELEMENT_ARRAY_BUFFER*.
- *size* ist die Größe des Arrays.
- **data* sind die eigentlichen Daten des Arrays.
- *usage* dient OpenGL als Performance-Hinweis und kann einer der folgenden Werte sein:
 - *GL_DYNAMIC_DRAW* - Die Daten werden wiederholt durch die Anwendung festgelegt und häufig benutzt.
 - *GL_STATIC_DRAW* - Die Daten werden ein Mal durch die Anwendung festgelegt und häufig benutzt.
 - *GL_STREAM_DRAW* - Die Daten werden ein Mal durch die Anwendung festgelegt und selten benutzt.

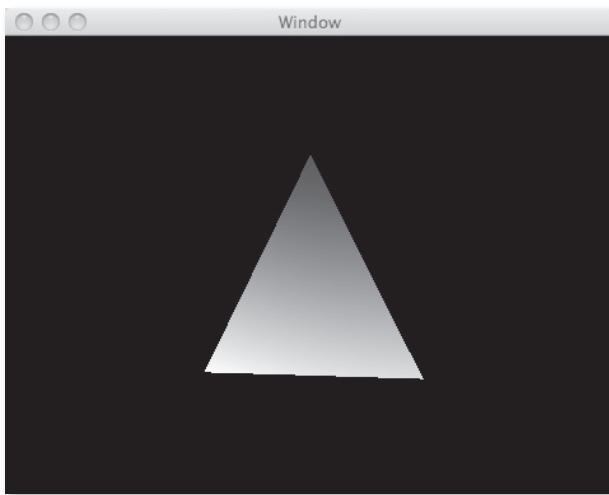
In der Regel genügt es, wenn man für statische Objekte den Parameter (*GL_STATIC_DRAW*) und für z. B. animierte Models den Wert (*GL_DYNAMIC_DRAW*) verwendet.

VBOs rendern

Das Rendern von VBOs unterscheidet sich in zwei Dingen vom Rendern normaler Vertex-Arrays:

1. Ein Puffer muss aktiviert werden, der die Daten enthält.
2. Je nachdem, ob unsere Rohdaten in einem bzw. mehreren Arrays liegen, müssen wir einen Offset definieren, der aussagt, wie unsere Daten organisiert sind.

Das Beispiel »Kapitel 8/Simple VBO« zeigt die einzelnen Schritte, die nötig sind, um mit VBOs zu rendern. Es wurde absichtlich wieder einfach gehalten, damit man die einzelnen Schritte besser nachvollziehen kann.



Dreieck, das mit Hilfe von VBOs gerendert wurde

Um den Code übersichtlich zu halten, wurde je ein Puffer für die Vertices und die Farbangaben erstellt.

Zuerst erstellen wir den Puffer für die Vertices und befüllen ihn mit den Daten.

```
// Puffer erstellen
glGenBuffers(1, &_vertexBuffer);
// Puffer binden
glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
// Puffer mit Daten füllen
glBufferData(GL_ARRAY_BUFFER, sizeof(_vertexData), _vertexData,
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Danach folgt der gleiche Vorgang, für die Farbangaben

```
// Puffer erstellen
glGenBuffers(1, &_colorBuffer);
// Puffer binden
glBindBuffer(GL_ARRAY_BUFFER, _colorBuffer);
// Puffer mit Daten füllen
glBufferData(GL_ARRAY_BUFFER, sizeof(_colorData), _colorData,
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Der abschließende Aufruf von `glBindBuffer(GL_ARRAY_BUFFER, 0);` sagt aus, dass wir nun mit der Arbeit an den Puffern fertig sind.

Die Daten, die aus den beiden Arrays `_vertexData` und `_colorData` kommen, wurden nun in den Speicher der Grafikkarte kopiert, weshalb man die Arrays im Prinzip löschen könnte. Mehr Vorarbeit ist nicht nötig, um VBOs zu erstellen, man sieht, dass es genau die drei Schritte waren, die oben besprochen wurden.

Anschließend wird die Geometrie gerendert.

```
// VBO rendern
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, 0);
glBindBuffer(GL_ARRAY_BUFFER, _colorBuffer);
glColorPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
```

Mit `glEnableClientState(...)` aktivieren wir zuerst wieder die beiden Arrays, die wir benötigen (Vertices, Farben). Danach binden wir den ersten Puffer und übergeben dieses Mal nicht die Daten des Arrays, sondern setzen die Werte einfach auf Null:

```
glVertexPointer(3, GL_FLOAT, 0, 0);
```

Das ist deshalb so, weil die Daten ja nun nicht mehr aus dem Array selbst kommen, sondern aus dem Speicher der Grafikkarte. Dasselbe machen wir dann mit dem

Puffer für die Farbangaben. Anschließend folgt der Befehl zum Rendern, den wir ja schon aus dem Abschnitt über die Vertex-Arrays kennen. Zum Schluss »räumen« wir wieder auf, indem wir die beiden Puffer auf Null setzen und die Vertex-Arrays deaktivieren, das war schon alles.

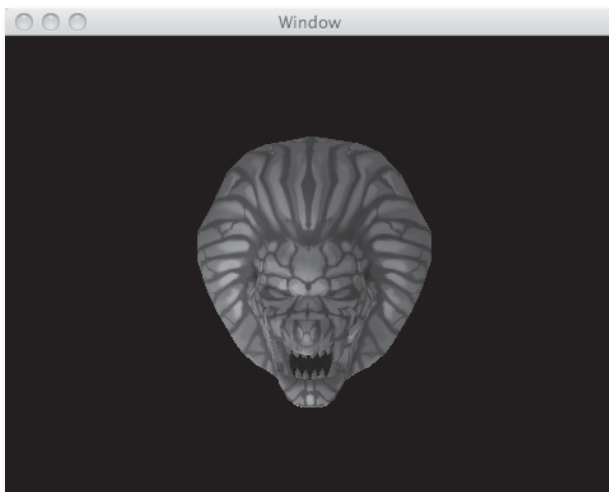
VBOs löschen

Da die Puffer ja Speicherplatz in der Grafikkarte belegen (genau wie die Display-Listen auch), ist es eine gute Idee, sie zu löschen, wenn man sie nicht mehr benötigt. Dazu rufen wir die Funktion `glDeleteBuffers(1, &_vertexBuffer);` auf und übergeben ihr als ersten Parameter die Anzahl der zu löschenden Puffer und als zweiten Parameter den Puffer selbst.

Nun ist das Beispiel, wie schon erwähnt, sehr einfach gehalten, in der Praxis würde man kein einzelnes Dreieck mit Hilfe von VBOs rendern, da der Funktions-Overhead größer wäre als der Geschwindigkeitsvorteil, den wir erreichen wollten. Es kann somit durchaus passieren, dass unser Programm durch den Overhead sogar »langsamer« läuft, obwohl wir VBOs verwendet haben.

Indexierte VBOs

Das folgende Beispiel »Kapitel 8/Advanced VBO« ist da schon ein wenig komplexer. Es zeigt, wie man indexierte VBOs rendert, und eignet sich daher sehr gut für Objekte mit einer hohen Polygonanzahl.



3D-Model, gerendert mit Hilfe von indexierten VBOs

Um das Programm übersichtlich zu halten, habe ich es in zwei Dateien unterteilt:

- »Mesh.h« ist die Klasse, über welche wir die VBOs für das 3D-Model erzeugen und rendern werden.
- »Mesh.model« enthält die Rohdaten des 3D-Models. Diese wurden aus einem 3D-Programm exportiert.

Diese Rohdaten haben folgenden Inhalt:

`static short face_indicies[1390][9]` – Indices, über die wir später rendern werden

`static GLfloat vertices [713][3]` – die Vertices (Eckpunkte) der Polygone

`static GLfloat normals [1031][3]` – die Normale

`static GLfloat textures [397][2]` – Textur-Koordinaten

Mesh erzeugen

Beginnen wir mit dem Erstellen des Meshs:

```
// Mesh erzeugen
_mesh = [[Mesh alloc]init];
// Mesh hat maximal die Anzahl der (Dreiecke * Vertices) Daten
[_mesh createEmptyMeshWithNumberOfVertices:1390 * 3];
```

Nachdem das Objekt erzeugt wurde, müssen wir der Methode *createEmptyMeshWithNumberOfVertices* mitteilen, aus wie viele Dreiecken unser Model besteht (1390*3). Dadurch wird Speicher für die vier Arrays allokiert, in denen später die Rohdaten bzw. Indexe gespeichert werden sollen.

Bevor wir nun die Daten in die Arrays kopieren, stellen wir sicher, dass keine Dubletten vorhanden sind. Diese sollen nämlich vom Programm aussortiert werden, damit keine Redundanz entsteht.

Die Vorgehensweise dazu erfolgt in zwei Schritten: Zunächst einmal werden die Rohdaten anhand ihres Indexes aus den einzelnen Arrays (in *Mesh.model*) gelesen, was folgender Code tut:

```
// Alle Polygone durchlaufen
for( triangles = 0; triangles < 1390; triangles++)
{
```

```

        // Dreiecke zusammenbauen
        for(vertex = 0; vertex < 3; vertex++)
        {
            // Anhand des Index (face_indicies) die passenden Daten aus //
            // den 3 Arrays auslesen
            v[vertex] = makeVector(vertices[face_indicies[triangles][vertex]][0],
                                   vertices[face_indicies[triangles][vertex]][1],
                                   vertices[face_indicies[triangles][vertex]][2]);

            n[vertex] = makeVector(normals[face_indicies[triangles][vertex+3]][0],
                                   normals[face_indicies[triangles][vertex+3]][1],
                                   normals[face_indicies[triangles][vertex+3]][2]);
            t[vertex] = makeTextureCoords(textures[face_indicies[triangles]
            [vertex+6]][0],
                                   textures[face_indicies[triangles][vertex+6]]
                                   [1]);
        }
        // Prüfen ob das Dreieck hinzugefügt werden muss
        [_mesh addTriangle:&v[0] normals:&n[0] texCoords:&t[0]];
    }

```

Innerhalb dieser Schleife übergeben wir der Methode `[_mesh addTriangle:&v[0] normals:&n[0] texCoords:&t[0]]`; die ausgelesenen Daten, welche ihrerseits die Daten auf Dubletten überprüfen.

Dabei schauen wir, ob der Unterschied zwischen den Werten, welche wir schon in die Arrays gespeichert haben, und denen, die wir übergeben haben, kleiner ist als 0.000001. Wenn ja, speichern wir die Rohdaten selbst nicht mehr, sondern nur noch den Index-Wert, der auf diese Rohdaten verweist. Wenn der Schwellerwert größer ist als 0.000001, dann werden sowohl die Rohdaten als auch die Indices gespeichert. Kurz gesagt prüfen wir, ob es schon einmal ein Polygon mit fast denselben Werten im Array gibt.

Hier der wichtigste Code-Ausschnitt zu dieser Methode:

```

-(void)addTriangle:(CFXVector*)vertices normals:(CFXVector*)
normals
texCoords:(CFXTextureCoords*)texCoords
{

```

```

// Pruefen, ob es ein bestehendes Polygon in der Liste gibt, dass
// die Selben Werte besitzt wie das, welches uebergeben wurde.
float delta = 0.000001;
int vertex;
for(vertex = 0; vertex < 3; vertex++)
{
    int tmp = 0;
    for(tmp = 0; tmp < _numVertices; tmp++)
    {
        // Ist die Vertex-Position gleich ?
        if(isAlmostSame(_vertices[tmp].x, vertices[vertex].x,
            delta) &&
            isAlmostSame(_vertices[tmp].y, vertices[vertex].y,
            delta) &&
            isAlmostSame(_vertices[tmp].z, vertices[vertex].z,
            delta) &&
            // Sind die Normale gleich ?
            isAlmostSame(_normals[tmp].x, normals[vertex].x,
            delta) &&
            isAlmostSame(_normals[tmp].y, normals[vertex].y,
            delta) &&
            isAlmostSame(_normals[tmp].z, normals[vertex].z,
            delta) &&
            // Sind die Textur-Koordinaten gleich ?
            isAlmostSame(_texCoords[tmp].u, texCoords[vertex].u, delta)
            &&
            isAlmostSame(_texCoords[tmp].v, texCoords[vertex].v,
            delta))
        {
            // Wenn ja, nur den Index (also den Verweis auf
            // das Polygon) in die Liste einfuegen
            _indicies[_numIndexes] = tmp;
            _numIndexes++;
            break;
        }
    }
}
// Polygon ist noch nicht in der Liste, also einfuegen
if(tmp == _numVertices) {
    _vertices[_numVertices] = vertices[vertex];
    _normals[_numVertices] = normals[vertex];
    _texCoords[_numVertices] = texCoords[vertex];
    _indicies[_numIndexes] = _numVertices;
}

```

```

_numIndexes++;
_numVertices++;
    }
}

```

Durch das Aussortieren von Dubletten sparen wir nicht nur kostbaren Speicherplatz, sondern vermindern auch die Zahl der Vertices, die wir rendern wollen.

Nachdem die Daten erzeugt wurden, werden dann in der Methode *-(void)buildVBOs* die Puffer erzeugt:

```

-(void)buildVBOs
{
    // Puffer anlegen
    glGenBuffers(4, _buffers);
    // Daten in Grafikkarten-Speicher kopieren
    // Vertices
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*3, _
vertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    // Normale
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*3,
_normals, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    // Textur-Koordinaten
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[2]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*2,
_texCoords, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    // Indexe
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _buffers[3]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort)*_num-
Indexes, _indices, GL_STATIC_DRAW);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    // Da die Daten nun in der Grafikkarte liegen, koennen die
    // Arrays geloescht werden
    if(_indices)
        free(_indices);
    if(_vertices)
        free (_vertices);
}

```



```
    if(_normals)
        free(_normals);
    if(_texCoords)
        free(_texCoords);
}
```

Für jedes Array erzeugen wir nun einen eigenen Puffer, der die Daten speichert. Anschließend werden die eigentlichen Arrays gelöscht, da diese nicht länger gebraucht werden.

AUFGEPASST

Achtung: Bei den Indices lautet der erste Parameter von *glBindBuffer(...)* nun nicht mehr *GL_ARRAY_BUFFER*, sondern *GL_ELEMENT_ARRAY_BUFFER*.

Der eigentliche Rendervorgang unterscheidet sich zu dem im vorangegangenen Beispiel nur dadurch, dass wir jetzt mit indexierter Geometrie arbeiten. Deshalb verwenden wir nun nicht mehr *glDrawArrays(...)*, welche die Arrays ja in einem Rutsch rendert, sondern die Funktion:

```
glDrawElements(GL_TRIANGLES, _numIndexes, GL_UNSIGNED_SHORT, 0);
```

Diese erwartet als ersten Parameter den Zeichenmodus, als zweiten die Anzahl der Indices und als dritten Parameter deren Datentyp.

Der letzte Parameter ist auch hier wieder Null, da wir auf die Indices selbst ja über die Funktion

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _buffers[3]);
```

verweisen.

VBOs ändern

Nun stellt sich die Frage, wie die Daten, die in den Puffern liegen, geändert werden können. Dazu müssen die Daten zunächst wieder von der Grafikkarte in den Hauptspeicher übertragen und nach der Änderung wieder zur Grafikkarte »hochgeladen« werden. Wenn wir z. B. die Normale aus dem obigen Beispiel normalisieren wollten, müssten wir folgendermaßen vorgehen:

```
glBindBuffer(GL_ARRAY_BUFFER, _buffers[1]);
```

 Puffer, der die Normale beinhaltet, binden (aktivieren)

`CFXVector *normals = (CFXVector *)glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);` Normale aus dem Grafikkarten-Speicher in den Hauptspeicher kopieren

`glMapBuffer(...)` erwartet als ersten Parameter wieder die Art des Arrays. Über den zweiten Parameter müssen wir festlegen, was wir mit den Daten tun möchten, dabei sind folgende Werte erlaubt, die selbsterklärend sein sollten: `GL_READ_WRITE`, `GL_WRITE_ONLY`, `GL_READ_ONLY`.

Jetzt können wir die Daten ändern

```
if(normals!= NULL)
{
    int i;
    for(i = 0; i < _numVertices; i++)
        normals[i] = normalizeVector(normals[i]);
    glUnmapBuffer(GL_ARRAY_BUFFER);
}
```

Bevor man nun den Puffer wieder verwenden kann, muss man `glUnmapBuffer(...)` aufrufen, der OpenGL mitteilt, dass das Ändern der Daten abgeschlossen ist.

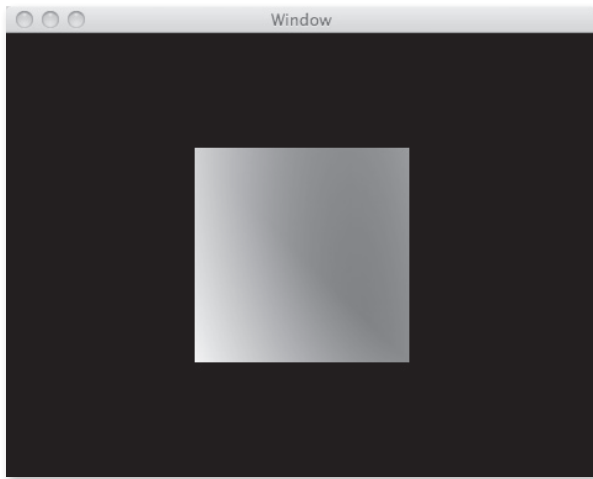
Statische und dynamische Daten mischen

Es ist natürlich kein Problem, die beiden Arten von Daten miteinander zu mischen. Man kann z. B. die Vertices in VBOs speichern, die Textur-Koordinaten aber weiterhin in normalen Vertex-Arrays halten, um sie z. B. bei jedem Renderdurchlauf zu verändern (zu animieren). Diese Vorgehensweise kann durchaus schneller sein, als ständig die Daten in den Hauptspeicher zu kopieren, zu ändern und wieder zur Grafikkarte hochzuschieben.

Hier hilft eigentlich nur ausprobieren, da es von vielen Faktoren abhängig ist, was schneller ist.

VBOs mit Offset

Das Thema VBOs möchte ich mit einem Beispielprogramm »Offset VBO« beenden, indem die Daten nun nicht mehr in verschiedenen Arrays gespeichert werden, sondern alle Daten in einem Array liegen und mit Hilfe eines Offset-Werts bestimmt wird, wie die Daten organisiert und gerendert werden.



Ein simples Quadrat, das mit Hilfe eines Offset-Wertes innerhalb des VBOs gerendert wird

Zunächst einmal wird wieder eine Struktur definiert, welche ein Eckpunkt inklusive der Farbe speichert.

```
typedef struct _SimpleVertex
{
    CFXVector position;
    CFXVector color;
}SimpleVertex;
```

Dann werden die Vertices mit den Farben definiert. Da es nur vier Stück sind, lässt sich das leicht händisch erledigen.

```
// Quadrat definieren
_vertices[0].position = makeVector(-1.0, -1.0, 0.0);
_vertices[1].position = makeVector( 1.0, -1.0, 0.0);
_vertices[2].position = makeVector( 1.0,  1.0, 0.0);
_vertices[3].position = makeVector(-1.0,  1.0, 0.0);
_vertices[0].color = makeVector( 1.0, 1.0, 0.0);
_vertices[1].color = makeVector( 1.0, 0.0, 0.0);
_vertices[2].color = makeVector( 1.0, 0.0, 1.0);
_vertices[3].color = makeVector( 0.0, 1.0, 0.0);
```

Anschließend wird nur ein einziger Puffer erzeugt, der das komplette Array speichern soll:

```
// VBO erzeugen
glGenBuffers(1, &_buffer);
glBindBuffer(GL_ARRAY_BUFFER, _buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(SimpleVertex)*4, _vertices,
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Beim Rendern werden die Vertices nun folgendermaßen übergeben:

```
glBindBuffer(GL_ARRAY_BUFFER, _buffer);
// Stride zwischen den Daten hat die Größe von SimpleVertex
// Die Vertexdaten beginnen ab der Position 0 in SimpleVertex
glVertexPointer(3, GL_FLOAT, sizeof(SimpleVertex), 0);
```

Sie sehen, der dritte Parameter ist nun nicht mehr eine Null, sondern hat die Größe der Struktur von *SimpleVertex*, was so viel bedeutet wie »Alle Vertex-Daten liegen hintereinander im Array mit einem Abstand von *SimpleVertex* dazwischen und beginnen bei der Position Null (letzter Parameter) im Array.«

Da nun alle Daten in einem Puffer liegen, müssen wir OpenGL mitteilen, welche davon Vertices und welche die Farben sind. Das machen wir mit Hilfe des besagten Offset-Werts:

```
glBindBuffer(GL_ARRAY_BUFFER, _buffer);
// Stride zwischen den Daten hat die Größe von SimpleVertex
// Die Vertexdaten beginnen ab der Position 3 in SimpleVertex
glColorPointer(3, GL_FLOAT, sizeof(SimpleVertex),
BUFFER_OFFSET(3*sizeof(float)));
```

Der letzte Parameter (Offset) erwartet einen Zeiger. Wir haben aber »nur« einen *int*-Wert, weshalb wir uns mit dem Makro:

```
#define BUFFER_OFFSET(i) ((char *)NULL+(i))
```

weiterhelfen, welches den *int*-Wert entsprechend zu einem Pointer »verbiegt«.

Durch diesen Offset-Wert sagen wir, dass die Farbdaten ab der dritten Stelle im Array beginnen.

Wenn wir uns nochmals die Struktur *SimpleVertex* anschauen, sehen wir, dass die Farben nach *position* beginnen (CFXVector hat ja 3 Elemente).

Der Rest des Programms ist dann identisch mit den vorhergehenden, weshalb ich nicht näher darauf eingehen möchte.

Durch diese Art der Datenübergabe haben wir die wohl bestmögliche Performance, da wir nur noch ein einziges Array haben, welches gespeichert werden muss. Um dem Ganzen nun noch die Krone aufzusetzen, könnte man auch hier, um die Daten noch mehr zu optimieren, mit indexierter Geometrie arbeiten.

Zusammenfassung

In diesem Kapitel ging es einzig und allein darum zu lernen, wie man den Rendervorgang beschleunigen kann.

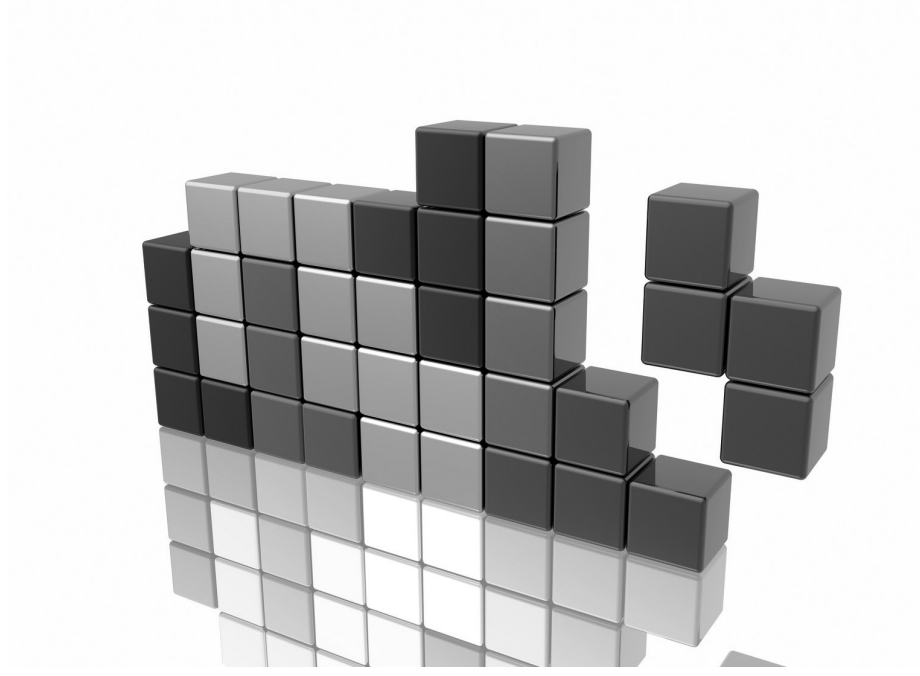
Dabei haben wir zuerst die Display-Lists kennengelernt, welche sich gut dazu eignen, immer wiederkehrende Dinge in eine Liste zu »verpacken« und mit einem einzigen Aufruf zu rendern. Dabei ist der Performance-Zuwachs aber sehr stark von der OpenGL-Implementierung abhängig. Gerade bei Matrix-Operationen bringen Display-Lists aber einen echten Geschwindigkeitszuwachs. Wir wissen nun auch, dass Display-Lists vollkommen statisch sind, was sie nur bedingt brauchbar macht.

Im Gegensatz dazu eignen sich Vertex-Arrays sehr gut für dynamische Objekte, die sehr oft während des Rendervorgangs geändert werden. Vertex-Arrays haben auch den Vorteil, dass die Daten sehr schnell zur Grafikkarte transportiert werden können, da der Rendervorgang sozusagen in einem Rutsch getätigt wird.

Wir haben auch gesehen, dass man mit indexierten Vertex-Arrays eine Datenredundanz vermeiden kann, was natürlich bei großen Objekten mit vielen Polygonen ein enormer Vorteil ist.

Zum Schluss haben wir noch die VBOs kennengelernt, welche die Vorteile der Display-Lists mit denen der Vertex-Arrays miteinander verbindet und sie dadurch zur allerersten Wahl macht, wenn es darum geht, statische oder dynamische Objekte mit einer hohen Polygonanzahl zu rendern.

Gerade das Thema Vertex-Arrays und VBOs ist hiermit bei weitem noch nicht erschöpft, weshalb ich an dieser Stelle unbedingt auf die Dokumentation zu den Themen verweisen möchte. Apple bietet zu den Vertex-Arrays einige interessante Erweiterungen an, die Sie sich unbedingt einmal anschauen sollten (Dokumentation zu Xcode).



Textausgabe in OpenGL

Die Textausgabe in OpenGL ist nicht ganz trivial, was daran liegt, dass OpenGL selbst keine Funktionen dafür bereithält. Das bedeutet, wir müssen diese Funktionalität komplett selbst entwickeln. Das Ziel dieses Kapitels soll sein, ein Textsystem (CFXTextToScreen) zu entwickeln, welches sowohl Fonts, die auf dem System installiert sind, und solche, die von extern geladen werden, ausgeben kann.

Der Grund dafür, weshalb das Thema Text erst jetzt behandelt wird, ist der, dass schon einige Kenntnisse über OpenGL nötig sind, die wir nun aber haben. An die Arbeit!

Die Idee, die hinter unserem Textsystem liegt, ist folgende: Wir erstellen aus einem beliebigen Font einen Satz Buchstaben, Zahlen und Sonderzeichen, die auf der ASCII-Tabelle basieren. Dabei beginnen wir ab dem Zeichen mit dem Dezimalwert 32 (Leerzeichen) und gehen bis zu dem Zeichen 127 (DEL). Aus jedem dieser Glyphe (95 Stück) erstellen wir dann ein Bitmap, welches wir in einer Display-Liste speichern. Wenn wir einen String ausgeben möchten, müssen wir nur noch die entsprechende Display-Liste zusammenbauen und ausgeben.

Soweit der theoretische Hintergrund. Nun kommt der praktische Teil:

Beginnen wir zuerst mit der Methode, die einen beliebigen Font laden soll.

```
- (void) buildFont:(NSString *)fontName size:(int)fontSize from-
External:(BOOL)ex
{
    _baseList = glGenLists( 95 );           // Erzeuge 95 Display-
                                           // Lists fuer jede Glyph
                                           // eine extra Liste

    FSRef myFSRef;
    FSSpec fontFSSpec;
    Boolean isDir = 0;
    NSString *pathToFont = [[NSBundle mainBundle] pathForResource:fontName ofType: @"ttf"];
    //Wenn der Font aus dem MainBundle kommen soll
    if(ex)
    {
        OSStatus status = FSPathMakeRef ((const UInt8 *)[pathToFont
        fileSystemRepresentation], &myFSRef, &isDir );
        if ( status != noErr )
```



```

        NSLog(@"Fehler beim Erzeugen von FSPathMakeRef: %c",
              [pathToFont fileSystemRepresentation]);
        status = FSGetCatalogInfo ( &myFSRef, kFSCatInfoNone,
        NULL, NULL, &fontFSSpec, NULL);
        if ( status != noErr )
            NSLog(@"Fehler beim lesen von FSGetCatalogInfo");
        status = ATSTFontActivateFromFileSpecification ( &fontFS-
        Spec, kATSTFontContextLocal, kATSTFontFormatUnspecified,
        NULL, kATSTOptionFlagsDefault, NULL);
        status = ATSTFontActivateFromFileReference(&myFSRef, kATS-
        FontContextLocal, kATSTFontFormatUnspecified, NULL, kATS-
        OptionFlagsDefault, NULL);
    }
    _theFont = [ NSFont fontWithName:fontName size:fontSize ];
    if( _theFont == nil )
        NSLog( @"Font ist nil");
    if( ![ self buildDisplayLists:' , count:95 baseList:_baseList
    ] )
        NSLog( @"Displaylist fuer Font konnte nicht erstellt wer-
        den" );
}

```

Wir übergeben der Methode den Namen des Fonts und die Größe. Der Parameter *fromExternal* besagt, ob es sich um einen Font handelt, der von extern geladen werden soll (NO), oder ob es eine Schrift ist, welche schon auf dem System installiert ist. Dadurch haben wir die Möglichkeit, auch Fonts zu nutzen, die wir unseren Spielen mitgeben möchten.

Font laden

Wenn wir z. B. die Schrift Arial mit der Größe von 20 Punkten laden möchten, rufen wir die Methode folgendermaßen auf:

```
[_font buildFont:@"Arial" fontSize:20 fromExternal:NO];
```

Der letzte Parameter kann hier *No* sein, da Arial wohl auf allen Systemen vorhanden ist.

Display-Listen erstellen

Als Nächstes schauen wir uns die Methode an, welche die besagten Display-Listen erstellt:

```

- (BOOL) buildDisplayLists:(unichar)firstCharacter count:(int)
count baseList:(GLint)base
{
    // Pixel unpacking sichern
    glPushClientAttrib( GL_CLIENT_PIXEL_STORE_BIT );
    glPixelStorei( GL_UNPACK_ALIGNMENT, 1 ); // Ausrichtung der
                                           // Pixel im Bitmap
                                           // sind Shortint

    NSColor *black = [ NSColor blackColor ];
    NSDictionary *attributes = [ NSDictionary
dictionaryWithObjectsAndKeys:
    _theFont, NSFontAttributeName,
    [ NSColor whiteColor ],
    NSForegroundColorAttributeName,
    black, NSBackgroundColorAttributeName,
    nil ];
    NSRect characterRect;
    characterRect.origin.x = 0;
    characterRect.origin.y = 0;
    NSImage *tmpImage = [ [ [ NSImage alloc ] initWithSize:NS-
MakeSize( 0, 0 ) ] autorelease ];
    BOOL success = TRUE;
    GLint listNumber;
    NSString *currentCharacter;
    unichar currentUnichar;
    NSSize charSize;
    for( listNumber = base, currentUnichar = firstCharacter; cur-
rentUnichar < firstCharacter + count; listNumber++, current-
Unichar++ )
    {
        currentCharacter = [ NSString stringWithCharacters:&cur-
rentUnichar length:1 ];
        charSize = [ currentCharacter sizeWithAttributes:attri-
butes ];
        characterRect.size = charSize;
        characterRect = NSIntegralRect( characterRect );
        if( characterRect.size.width > 0 && characterRect.size.
height > 0 )
        {
            [ tmpImage setSize:characterRect.size ];
            [ tmpImage lockFocus ];

```

```

        [ [ NSGraphicsContext currentContext ] setShouldAnti-
        alias:NO ];
        [ black set ];
        [ NSBezierPath fillRect:characterRect ];
        [ currentCharacter drawInRect:characterRect withAttri-
        butes:attributes ];
        [ tmpImage unlockFocus ];
        if( ![ self makeListFromImage:listNumber with-
        Image:tmpImage ] )
        {
            success = FALSE;
            break;
        }
    }
}
glPopClientAttrib();
return success;
}

```

Der erste Parameter der Methode ist das Zeichen, mit welchem unsere Display-Liste beginnen soll, Wie gesagt starten wir mit einem Leerezeichen (dezimal 32). Der zweite Parameter ist die Anzahl der zu erstellenden Listen (95), und der letzte ist die ID der Display-Liste.

Gehen wir die wichtigsten Teile der Methode genauer durch:

```

// Pixel unpacking sichern
glPushClientAttrib( GL_CLIENT_PIXEL_STORE_BIT ); // Pixelaus-
richtung schuetzen
glPixelStorei(.....);

```

glPixelStorei legt fest, wie das Bitmap (dazu gleich mehr) im Speicher abgelegt und wieder ausgelesen wird.

Bevor wir diese Einstellungen machen, weisen wir OpenGL über die Funktion

```
glPushClientAttrib( GL_CLIENT_PIXEL_STORE_BIT );
```

an, die aktuellen Einstellungen zu sichern. *glPushClientAttrib(...)* arbeitet wie *glPushMatrix(...)* auch mit der Stapeltechnik, nur dass wir hier keine Matrizen sichern, sondern bestimmte Attribute, die durch die State-Machine gesetzt sind.

```

NSDictionary *attributes = [ NSDictionary
dictionaryWithObjectsAndKeys:
    _theFont, NSFontAttributeName,
    [ NSColor whiteColor ],
    NSForegroundColorAttributeName,
    black, NSBackgroundColorAttributeName,
    nil ];

```

erstellt ein Dictionary, welches die Attribute unseres Fonts speichern soll, da wir diese gleich benötigen.

Anschließend erzeugen wir ein temporäres Image, in das wir die Glyphen zeichnen:

```

theImage = [ [ NSImage alloc ] initWithSize:NSMakeSize( 0, 0 ) ]
autorelease ];

```

Font erstellen

Zum Schluss erzeugen wir eine Schleife, in der wir alle Glyphen nacheinander in ein Image zeichnen.

```

for( listNumber = base, currentUnichar = firstCharacter; current-
Unichar < firstCharacter + count; listNumber++, currentUnichar++ )
{
    currentCharacter = [ NSString stringWithCharacters:&cur-
rentUnichar length:1 ];
    charSize = [ currentCharacter sizeWithAttributes:attri-
butes ];
    characterRect.size = charSize;
    characterRect = NSIntegralRect( characterRect );
    if( characterRect.size.width > 0 && characterRect.size.
height > 0 )
    {
        [ tmpImage setSize:characterRect.size ];
        [ tmpImage lockFocus ];
        [ [ NSGraphicsContext currentContext ] setShouldAnti-
alias:NO ];
        [ black set ];
        [ NSBezierPath fillRect:characterRect ];
        [ currentCharacter drawInRect:characterRect withAttri-
butes:attributes ];
        [ tmpImage unlockFocus ];
    }
}

```

```

        if( ![ self makeListFromImage:listNumber with-
Image:tmpImage ] )
        {
            success = FALSE;
            break;
        }
    }
}

```

Das war nun der zweite Schritt. Wie gesagt wollen wir die Glyphen ja in einem Bild abspeichern. Leider kann OpenGL aber mit einem `NSImage` nichts anfangen, weshalb dieses noch in ein OpenGL-taugliches Bitmap kopiert werden muss, was folgende Methode macht:

```

- (BOOL) makeListFromImage:(GLint)listNumbers withImage:(NSImage
*)theImage
{
    NSBitmapImageRep *bitmapRep = [ NSBitmapImageRep imageRep-
WithData:[ theImage TIFFRepresentation] ];
    int height =[bitmapRep pixelsHigh];
    int width =[bitmapRep pixelsWide];
    unsigned char *bytes =[bitmapRep bitmapData];
    int bytesPerRow =[bitmapRep bytesPerRow];
    int samples =[bitmapRep samplesPerPixel];
    unsigned char *buffer = calloc( ceil( (float) bytesPerRow /
8.0 ), height );
    if( buffer == NULL )
    {
        NSLog(@"Fehler beim Speicher anlegen in
makeDisplayList:withImage:");
        return FALSE;
    }
    unsigned char *move = buffer;
    // Image durchlaufen und Pixel in newBuffer (Bitmap) kopieren
    int rowIndex;
    int columnIndex;
    int current;
    int value;
    for( rowIndex = height - 1; rowIndex >= 0; rowIndex-- )
    {
        current = 128;
        value = 0;

```

```

        for( columnIndex = 0; columnIndex < width; columnIndex++ )
        {
            if( bytes[ rowIndex * bytesPerRow + columnIndex * samples ] )
            {
                value |= current;
            }
            current >>= 1;
            if( current == 0 )
            {
                *move++ = value;
                current = 128;
                value = 0;
            }
        }
        if( current != 128 )
        {
            *move++ = value;
        }
    }
    glNewList( listNumbers, GL_COMPILE );// Display-List
                                           // erstellen
    glBitmap( width, height, 0, 0, width, 0, buffer ); // OpenGL-
                                           // Bitmap aus dem Image erstellen
    glEndList();// Display-List Ende
    free( buffer );
    return TRUE;
}

```

Zunächst erstellen wir aus dem übergebenen Image ein *NSBitmapImageRep*, damit wir an die Rohdaten herankommen. Mit der Anweisung

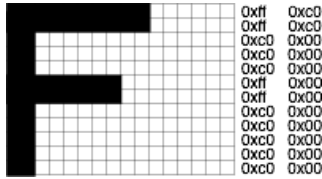
```
unsigned char *buffer = calloc( ceil( (float) bytesPerRow / 8.0 ),
height );
```

erzeugen wir einen Puffer, der später unser fertiges Bitmap enthalten soll. In der Schleife durchlaufen wir die einzelnen Pixel und kopieren sie in den Puffer. Zum Schluss wird dann eine Display-Liste erstellt, welche unser Bitmap enthalten soll.

Bitmaps

Nun aber nochmals zu dem Bitmap, da dieses ja noch vollkommen neu für Sie ist.

Ein Bitmap ist, wie der Name schon sagt, nicht anderes als ein 2D-Raster (ein Bild) in dem einzelne Bits gesetzt werden oder nicht. Um dies zu verdeutlichen, hier einmal eine Abbildung:



Der Buchstabe »F« als Bitmap

Auf der linken Seite haben wir den Buchstaben selbst und auf der rechten stehen die Werte für die »gesetzten« Punkte innerhalb des Rasters.

POWER

Wer in den Achtzigerjahren auf einem 8-Bit-Computer schon mal Sprites erstellt hat, dem dürfte diese Technik bekannt vorkommen, da diese exakt dieselbe ist.

Um nun dieses Bitmap auszugeben, verwenden wir die Funktion `glBitmap(...)`, die folgendermaßen aussieht:

`glBitmap (GLsizei width, GLsizei height, GLfloat xorig, GLfloat yorig, GLfloat xmove, GLfloat ymove, const GLubyte *bitmap);`

Schauen wir uns die Parameter im Einzelnen an:

- *width, height* sind die Abmessungen des Bitmaps
- *xorig, yorig*: Der Ursprung ist links unten
- *xmove, ymove*: Dieser Offset-Wert wird zu der aktuellen Raster-Position addiert, nachdem das Bitmap gezeichnet wurde.
- **bitmap* enthält die Rohdaten, die unser Bitmap darstellen soll.

Um jetzt den besagten Buchstaben aus der Abbildung oben auszugeben, würde man die Rohdaten ungefähr so definieren:

```
GLubyte buffer[24] = {
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xff, 0x00,
    0xff, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xc0, 0x00,
    0xff, 0xc0,
    0xff, 0xc0};
```

Wenn man genauer hinschaut, sieht man, dass der Buchstabe von unten nach oben definiert werden muss, die Ausgabe erfolgt dann aber »richtig herum«.

Text ausgeben

Jetzt fehlt nur noch die eigentliche Ausgabe des Strings.

```
- (void) drawTextToScreen:(NSString*)theString
    screenSizeX:(int)sizeX
    screenSizeY:(int)sizeY
    onPositionX:(float)xPos
    onPositionY:(float)yPos
{
    // In 2D Modus umschalten
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(0, sizeX, 0, sizeY, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    // x,y Koordinaten links unten = 0,0
    glRasterPos2f(xPos, yPos);
    // vorhandene Display-Lists schuetzen
```



```

glPushAttrib( GL_LIST_BIT );
// Offset ab dem die aufzurufende Liste beginnt
glListBase( _baseList - 32 );
// Speicher allokieren
unichar *buffer = calloc( [ theString length ], sizeof
( unichar ) );
// String in C-String umwandeln
[ theString getCharacters:buffer ];
// Liste aufrufen
glCallLists( [ theString length ], GL_UNSIGNED_SHORT,
buffer );
// Speicher loeschen
free( buffer );
// Stapel zuruecksetzen
glPopAttrib();
// In 3D Modus zurueckschalten
glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
}

```

Wir uebergeben der Methode die GröÙe des Views, den String und die Position, an welcher der String ausgegeben werden soll.

Da wir keine Z-Position für unseren Text haben, ist es einfacher, diesen im 2D-Modus zu rendern, weshalb zunächst in den Ortho-Modus umgeschaltet wird.

```

// In 2D Modus umschalten
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glOrtho(0, sizeX, 0, sizeY, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

```

Der Text wird mit Hilfe der Funktion *glRasterPos2f* (*GLfloat x*, *GLfloat y*); auf dem Schirm positioniert, ausgehend von der Position (0,0) für links unten.

```

// x,y Koordinaten links unten = 0,0
glRasterPos2f(xPos, yPos);

```

Nun folgt das Setzen des Startwerts der Display-Listen:

```
glListBase( _base - 32 );
```

Die Funktion macht nichts anderes, als den Startwert (Offset) für die Display-Listen zu modifizieren (Standard ist 0), welche wir aufrufen wollen.

Da unsere Display-Listen nicht alle 256 Zeichen der ASCII-Tabelle (0-255) enthalten, sondern nur 95 Zeichen (32-), müssen wir OpenGL mitteilen, dass das Zeichen mit der Nummer 32 eigentlich das erste Zeichen in der Liste ist, weshalb wir von der Display-Liste ID 32 abziehen müssen, damit der Zugriff auf die einzelnen Glyphen stimmt.

Ein kleines Rechenbeispiel, damit das auch alles seine Richtigkeit hat:

Wir möchten ein »A« ausgeben (ASCII-Code 65), also sieht unsere Rechnung so aus:

$$65 - 32 = 33$$

was stimmt, da das »A« an der 33. Stelle in unseren Listen steht.

Gut, noch ein Beispiel: Wir brauchen ein Leerzeichen (ASCII-Code 32), die Rechnung dazu ist wieder

$$32 - 32 = 0$$

Das stimmt auch, da wie gesagt unsere Listen mit einem Leerzeichen beginnen.

Da bestehende Display-Listen nicht von dem modifizierten Startwert betroffen sein sollen, schützen wir diese durch den Aufruf von

```
glPushAttrib( GL_LIST_BIT );
```

Um nun den String auszugeben, rufen wir die Funktion

```
glCallLists( [ theString length ], GL_UNSIGNED_SHORT, buffer );  
// Liste aufrufen
```

auf, welche alle benötigten Display-Listen auf einmal aufruft.

Ganz zum Schluss wird wieder in den 3D-Modus zurückgeschaltet.

```
// In 3D Modus zurueckschalten
glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
```

So, nachdem wir nun die Klasse besprochen haben, schauen wir uns noch das Beispiel dazu an:

Das Programm »Kapitel 9/Font Demo« lädt zunächst zwei Fonts.

```
_font1 = [[CFXTextToScreen alloc]init];
[_font1 buildFont:@"Verdana" size:20 fromExternal:NO];
_font2 = [[CFXTextToScreen alloc]init];
[_font2 buildFont:@"SubatomicTsunami" size:20 fromExternal:YES];
```

Der erste Font ist »*Verdana*« und sollte auf allen Systemen installiert sein, weshalb hier der Parameter *fromExternal* wieder auf *No* steht.

Der zweite Font wird aus dem MainBundle geladen, weshalb *fromExternal* nun auf *YES* steht.

Nachdem die Fonts geladen wurden, können wir sie verwenden:

```
glColor3f(1.0, 0.0, 0.0);
[_font1 drawTextToScreen:@"OpenGL ist cool"
                      screenSizeX:bounds.size.width
                      screenSizeY:bounds.size.height
                      onPositionX:120
                      onPositionY:bounds.size.height-120];
glColor3f(1.0, 1.0, 0.0);
[_font2 drawTextToScreen:@"OpenGL ist cool"
                      screenSizeX:bounds.size.width
                      screenSizeY:bounds.size.height
                      onPositionX:50.0
                      onPositionY:10];
```

Zuerst wird die Zeichenfarbe definiert und anschließend der String ausgegeben. Die einzelnen Parameter wurden ja oben besprochen.



Einfache Textausgabe mit OpenGL

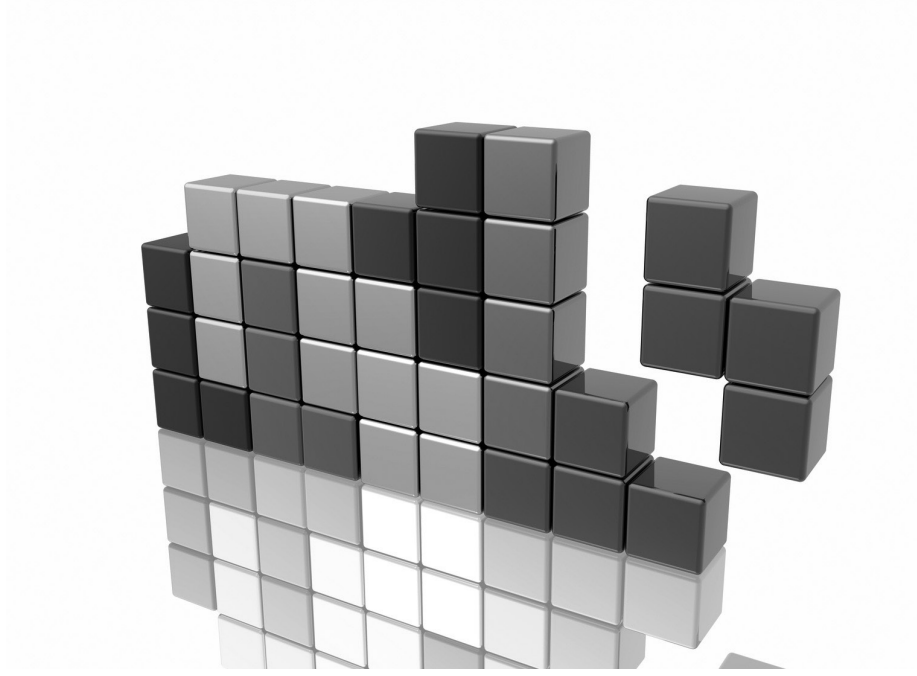
So, das war schon alles. Wie Sie gesehen haben, ist es nicht ganz einfach, Text auf dem Schirm auszugeben. Mit unsere Klasse CFXTextToScreen haben wir nun ein Werkzeug, mit dem wir relativ einfach dieses Problem umgehen können (sogar mit externen Fonts). Es gibt noch verschiedene andere Möglichkeiten, um Texte auszugeben. Eine recht simple Methode wäre, einen Satz Glyphen in einer Textur zu speichern (diese kann man sich mit jedem Bildbearbeitungsprogramm recht einfach erstellen) und dann über die Textur-Koordinaten auf die einzelnen Charakter zuzugreifen.

Das zeige ich Ihnen dann, wenn wir in Kapitel 15 das Spiel erstellen.

Zusätzliche Informationen

<http://www.opengl.org/resources/features/fontsurvey/>

Gute Seite mit Informationen zum Rendern von Text (auch 3D-Text)



Spezialeffekte

In diesem Kapitel werden wir uns anschauen, wie man die »Eye-Candys« programmiert, die man heute in fast jedem Videospiel bestaunen kann. Wenn Sie das Kapitel durchgearbeitet haben, wissen Sie, wie man Explosionen, Feuer, Rauch, Nebel und einiges mehr erstellen kann. Diese Effekte sind wie das Salz in der Suppe, ohne sie würde kein Spiel so richtig Spaß machen.

So toll die Effekte auch sein mögen, bei der Entwicklung müssen wir stets die Performance beachten, damit sich unser Spiel nicht in ein Daumenkino verwandelt, da bildschirmfüllende Explosionen erst einmal berechnet werden wollen.

Billboards

Bevor wir anfangen in die Trickkiste zu greifen, müssen wir uns noch eine Technik anschauen, die sich »Billboarding« nennt. Billboards sind texturierte Polygone (meistens Quadrate), die immer so ausgerichtet werden, dass sie zur Kamera zeigen. Dabei sind die Anwendungsgebiete für Billboards sehr unterschiedlich. Oft werden sie dazu benutzt, um komplexe geometrische Objekte in einer Szene zu ersetzen, was den Rendervorgang erheblich beschleunigt.

Die Idee, die dahinter steckt, ist folgende. Man nimmt das 3D-Model, welches ersetzt werden soll, und rendert es in eine Textur. Aus dieser Textur erstellt man dann das Billboard (ein Quadrat mit eben dieser Textur darauf). Im Spiel wird dann nicht das eigentliche Model gerendert, sondern das Billboard, was natürlich um ein Vielfaches schneller geht, da es ja aus viel weniger Polygonen besteht.

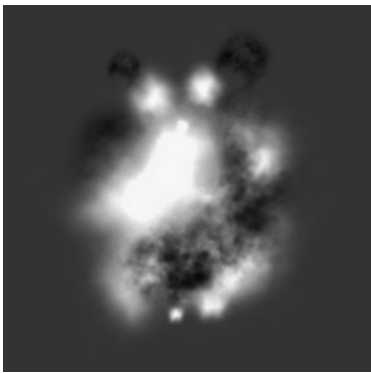
Die folgende Abbildung verdeutlicht nochmals das Prinzip: auf der linken Seite das echte Model und rechts daneben das Billboard.



Auf der linken Seite sieht man den echten Teekessel und auf der rechten Seite das Billboard.

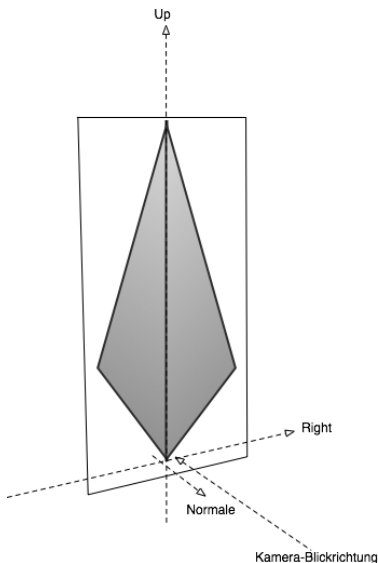
Allerdings stößt diese Technik in diesem Zusammenhang sehr schnell an ihre Grenzen, da der Trick spätestens dann auffällt, wenn die Kamera nahe genug am Billboard ist und man sieht, dass es sich nicht um ein echtes 3D-Model handelt (weil eben die Tiefe fehlt).

Auch für dieses Problem gibt es eine Lösung, die in sehr vielen Spielen eingesetzt wird. Dabei wird (wenn die Kamera nahe genug am Billboard ist) dieses einfach durch das eigentliche 3D-Model ersetzt. Dabei verwendet man in der Regel das Blending, wobei das echte Model über das Billboard geblendet wird, so dass der Spieler nichts von dem Trick bemerkt.



Ein weiteres Anwendungsbeispiel für Billboards sind Partikel-Systeme wie z.B. Explosionen oder Rauch. Der Unterschied zu dem Beispiel oben ist, dass man hier nicht ein einzelnes Quadrat verwendet, sondern hunderte bzw. tausende und diese dann mit Hilfe von Blending übereinander blendet, was wir uns später im Kapitel noch genauer anschauen werden.

Partikelsystem, das aus einzelnen Billboards zusammengesetzt wurde



Billboards erstellen

Nachdem wir nun wissen, was Billboards sind und wozu man sie verwenden kann, wird es Zeit, dass wir uns einmal anschauen, wie man sie erstellt. Wie gesagt, wird das Billboard immer so ausgerichtet (rotiert), dass es zur Kamera zeigt. Dabei verläuft sein Normal-Vektor parallel zum View-Vektor der Kamera.

Ausgangspunkt für die Ausrichtung des Billboards

Nun müssen wir den Up- und Right-Vektor der aktuellen Modelviewmatrix extrahieren und mit Hilfe dieser beiden Vektoren ein Viereck (Quad) aufspannen.

Im ersten Schritt benötigen wir natürlich erst einmal den Inhalt der Modelviewmatrix, welchen wir uns über die folgende Funktion holen:

```
float fMatrix[16];
glGetFloatv( GL_MODELVIEW_MATRIX, fMatrix );
```

Diese Funktion speichert den Inhalt der Modelviewmatrix in *fMatrix*.

Right	m0	m4	m8	m12
Up	m1	m5	m9	m13
	m2	m6	m10	m14
	m3	m7	m11	m15

Up- und Right-Vektor der Modelviewmatrix

Als Nächstes erstellen wir uns die beiden besagten Vektoren aus der Modelviewmatrix:

```
CFXVector right = makeVector(fMatrix[0], fMatrix[4], fMatrix[8] );
CFXVector up = makeVector(fMatrix[1], fMatrix[5], fMatrix[9] );
```

Zum Schluss »spannen« wir ein Viereck mit Hilfe dieser beiden Vektoren auf. Dabei sind die Variable *size*, die Größe unseres Billboards und *pos* die Position.

```
glBegin(GL_QUADS);
    // Unten links
    tmp.x=pos.x+(right.x + up.x)*-size;
    tmp.y=pos.y+(right.y + up.y)*-size;
    tmp.z=pos.z+(right.z + up.z)*-size;
    glVertex3f(tmp.x, tmp.y, tmp.z);
    // Unten rechts
    tmp.x=pos.x+(right.x - up.x)*size;
    tmp.y=pos.y+(right.y - up.y)*size;
    tmp.z=pos.z+(right.z - up.z)*size;
    glVertex3f(tmp.x, tmp.y, tmp.z);
    // Oben rechts
    tmp.x=pos.x+(right.x + up.x)*size;
    tmp.y=pos.y+(right.y + up.y)*size;
    tmp.z=pos.z+(right.z + up.z)*size;
    glVertex3f(tmp.x, tmp.y, tmp.z);
    // Oben links
    tmp.x=pos.x+(up.x - right.x)*size;
    tmp.y=pos.y+(up.y - right.y)*size;
    tmp.z=pos.z+(up.z - right.z)*size;
    glVertex3f(tmp.x, tmp.y, tmp.z);
glEnd();
```


Beispiel-Billboards

Da dies ja ein Buch über Spielprogrammierung und nicht nur allein über OpenGL ist, habe ich zu den Billboards ein etwas komplexeres Programm erstellt, welches man schon als Grundlage verwenden könnte, um ein kleines Outdoor-Spiel zu entwickeln.



Bäume, die über Billboards dargestellt werden

Da wir im Prinzip viele Teile dieses Programms schon vorher in anderen Beispielen besprochen haben, will ich mich auf die Dinge beschränken, die neu hinzugekommen sind. Wie man an der Abbildung sehen kann, wird ein Terrain gerendert, auf welchem mehrere Bäume zu sehen sind. Diese Bäume wiederum »bestehen« aus einfachen Billboards.

Der Code für das Terrain basiert im Wesentlichen auf dem Beispiel »Multi-Texturing« aus dem Kapitel 7, neu hinzugekommen ist, dass nun nicht mehr im »intermediate mode« gerendert wird, sondern mittels indexierten VBOs, was den Renderprozess um einiges beschleunigt.

Das Laden der Heightmap (dort stehen ja die Höheninformationen für das Terrain) hat sich nicht geändert, weshalb wir ihn nicht nochmals besprechen.

Gehen wir gleich zu der Stelle, an welcher die VBOs für das Terrain erzeugt werden.

```
- (BOOL)buildVBOs
{
    int z;
```

```

int x;
int length = ((_mapSize-1)*(_mapSize * 2+2))-2;
int textureOffset = _mapSize*_mapSize*3;
int currentIndex = 0;
int counter = 0;

_indicies = calloc(length, sizeof(GLuint));
if(_indicies == NULL)
    return NO;

_vertices = calloc(_mapSize*_mapSize*5, sizeof(GLfloat));
if(_vertices == NULL)
    return NO;

```

Zunächst wird wieder Speicher für die beiden Arrays (*_vertices* und *_indicies*) erzeugt, wobei *_vertices* das Array ist, welches die eigentlichen Rohdaten enthält, und *_indicies* das Array mit den Indexen ist.

```

// Vertices
for( z=0; z<_mapSize; z++)
{
    for( x=0; x<_mapSize; x++)
    {
        _vertices[counter++] = x*TERRAIN_MULTIPLIKATOR;
        _vertices[counter++] = [self scaledHeightAtPosition:x
        zPosition:z];
        _vertices[counter++] = z*TERRAIN_MULTIPLIKATOR;
    }
}

```

Das Erzeugen der Vertexdaten ist recht simpel, hier durchlaufen wir einfach zwei Schleifen und übernehmen deren Werte als X/Z-Koordinaten. Diese multiplizieren wir mit dem Wert aus *TERRAIN_MULTIPLIKATOR*, um das Terrain ein wenig größer zu machen. Die Y-Koordinate wird anhand der X/Z-Koordinate aus der Heightmap gelesen und mit *TERRAIN_HEIGHTSCALE* multipliziert, damit das Terrain nicht so spitzig wirkt.

```

// UV's
for( z=0; z<_mapSize; z++)
{
    for( x=0; x<_mapSize; x++)
    {

```

```

_vertices[textureOffset++] = (float)x * (TEXTURE_TILE_AMOUNT /
(float)_mapSize);
    _vertices[textureOffset++] = (float)z * (TEXTURE_TILE_AMOUNT /
(float)_mapSize);
}
}

```

Als Nächstes folgen die Textur-Koordinaten. Diese werden in das Array (*_vertices*) direkt hinter die Vertex-Koordinaten geschrieben, weshalb wir zuerst den Offset berechnen müssen, ab wo die Daten geschrieben werden sollen.

Der Offset-Wert errechnet sich aus der Größe des Terrains, multipliziert mit drei (X/Y/Z-Koordinate).

```
int textureOffset = _mapSize*_mapSize*3;
```

Die Textur soll über das Terrain gekachelt (*GL_REPEAT*) werden, weshalb die Textur-Koordinaten noch mit dem Wert

```
(float)z * (TEXTURE_TILE_AMOUNT / (float)_mapSize);
```

multipliziert werden.

Nachdem die Textur-Koordinaten auch berechnet sind, kann das Array in einen VBO kopiert werden.

```

glGenBuffers(1, &_vertexBufferForTerrain);
glBindBuffer(GL_ARRAY_BUFFER, _vertexBufferForTerrain);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_mapSize*_mapSize*5,
_vertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

```

Die Puffergröße errechnet sich aus der Größe des Terrains, multipliziert mit fünf (X/Y/Z-Koordinate + U/V-Koordinate).

Als Nächstes werden die Indexe berechnet:

```

// Indexe
for( z=0; z<_mapSize-1; z++)
{
    for( x=0; x<_mapSize; x++)
    {

```

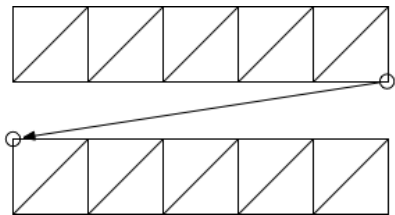
```

    if( (x == 0) && (z != 0) )
    {
        _indices[currentIndex++] = x+(z*_mapSize);
    }
    _indices[currentIndex++] = x+(z*_mapSize);
    _indices[currentIndex++] = x+((z+1)*_mapSize);

    if( (x == (_mapSize-1)) && (z != (_mapSize-2)) )
    {
        _indices[currentIndex++] = x+((z+1)*_mapSize);
    }
}
}

```

Da das Terrain später über Triangle-Strips gerendert wird, ist das Berechnen der Indexe nicht sonderlich schwierig. Einziges Problem dabei ist, wenn eine Reihe mit Triangle-Strips fertig ist und zur nächsten Reihe übergegangen werden muss. Wie wir wissen, werden Triangle-Strips immer zusammenhängend gerendert, was aber ein Problem für unser Terrain ist, da jede Reihe des Terrains unabhängig von der vorhergehenden gerendert werden soll.



Der letzte und erste Eckpunkt werden dupliziert, damit beim Übergang zwischen den einzelnen Strips nicht gezeichnet wird.

Der Trick ist nun, jeden außenliegenden Eckpunkt doppelt in die Liste der Indexe mit aufzunehmen. Da diese einzelnen Eckpunkte kein vollständiges Dreieck ergeben, werden sie von OpenGL auch nicht gerendert (außer im Wireframe-Modus / `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`) und wir können so zur nächsten Reihe springen.

Zum Schluss wird das Array auch in einen VBO gespeichert:

```

glGenBuffers(1, &_indexBufferForTerrain);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _indexBufferForTerrain);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*length, _indices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

```

Soweit das Terrain, kommen wir nun zu den Billboards.

Für diese wird zunächst eine Struktur definiert, um das Ganze wieder übersichtlich zu halten.

```
typedef struct _Billboard
{
    CFXVector position;
    GLuint texture;
    float width;
    float height;
} Billboard;
```

Dabei ist *position* die Position des Billboards, *texture* die Textur, *width* und *height* die Größe.

Danach werden die Billboards in einer Schleife erzeugt. Damit nicht alle gleich aussehen, werden fünf verschiedene Texturen erzeugt, welche dann zufällig den Billboards zugewiesen werden.

```
// Billboards
int i;
srand(time(NULL));
for(i=0; i<NUM_BILLBOARDS; i++)
{
    int t = rand()%5+1;

    switch(t)
    {
        case 1:
        {
            _billboards[i].texture = _billboardTexture1;
        }break;
        case 2:
        {
            _billboards[i].texture = _billboardTexture2;
        }break;
        case 3:
        {
            _billboards[i].texture = _billboardTexture3;
        }break;
        case 4:
```

```

        {
            _billboards[i].texture = _billboardTexture4;
        }break;
        case 5:
        {
            _billboards[i].texture = _billboardTexture5;
        }break;
    };
    float size = (float)(rand()%30)+20;
    _billboards[i].width = size;
    _billboards[i].height = size;

    float x = randomFloat(1.0,
        (TERRAIN_SIZE-1)*TERRAIN_MULTIPLIKATOR);
    float z = randomFloat(1.0,
        (TERRAIN_SIZE-1)*TERRAIN_MULTIPLIKATOR);
    float y = [self heightAtPosition:x zPosition:z]+
        _billboards[i].height;

    _billboards[i].position = makeVector(x, y, z);
}

```

Die Y-Position ermitteln wir aus den beiden Koordinaten (X/Z) des Terrains und addieren noch die Höhe der Billboards hinzu, damit sie direkt auf dem Boden stehen.

Kommen wir zum Render-Code; Das Terrain wird wie gesagt mittels VBOs gerendert, der Code dafür sollte nichts Neues für Sie sein, weshalb wir ihn außen vor lassen. Schauen wir uns noch die Zeichenroutine der Billboards an.

Auch für diese nutzen wir Blending bzw. Alpha-Masking (Kapitel 7), damit nur die »Bäume« an sich zu sehen sind, ohne den schwarzen Hintergrund der Textur.

```

// Billboards rendern
// Blending einschalten
glEnable(GL_BLEND);
// Blendfunktion festlegen
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
//Alphatest einschalten
glEnable(GL_ALPHA_TEST);
// Alles was groesser als 0.1, soll gerendert werden
glAlphaFunc(GL_GREATER, 0.1);

```

Jetzt folgt der Teil, über welchen wir die Position bzw. die Ausrichtung der Billboards berechnen.

Wir holen uns den Inhalt der Modelviewmatrix und speichern die beiden Vektoren (right, up):

```
float fMatrix[16];
glGetFloatv( GL_MODELVIEW_MATRIX, fMatrix );
CFXVector right = makeVector(fMatrix[0], fMatrix[4], fMatrix[8] );
CFXVector up = makeVector(fMatrix[1], fMatrix[5], fMatrix[9] );
```

Da wir eine bewegliche Kamera haben und sich dadurch der Inhalt der Modelviewmatrix bei jedem Renderdurchgang ändern kann, müssen wir diesen ein Mal pro Frame abfragen. Bei einer feststehenden Kamera muss dies natürlich nur ein Mal beim Start des Programms erfolgen und später dann nicht mehr.

Nun durchlaufen wir alle Billboards in einer Schleife (weil es einfacher ist, verwende ich hier den »intermediate mode«) und »spannen« über die beiden Vektoren (right, up) ein Quad auf.

```
for(i=0; i<NUM_BILLBOARDS; i++)
{
    glBindTexture(GL_TEXTURE_2D, _billboards[i].texture);
    CFXVector position = _billboards[i].position;

    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0);
        tmp.x=position.x+(right.x + up.x)*_billboards[i].width;
        tmp.y=position.y+(right.y + up.y)*_billboards[i].height;
        tmp.z=position.z+(right.z + up.z)*_billboards[i].width;
        glVertex3f(tmp.x, tmp.y, tmp.z);

        glTexCoord2f(1.0, 0.0);
        tmp.x=position.x+(right.x - up.x)*_billboards[i].width;
        tmp.y=position.y+(right.y - up.y)*_billboards[i].height;
        tmp.z=position.z+(right.z - up.z)*_billboards[i].width;
        glVertex3f(tmp.x, tmp.y, tmp.z);

        glTexCoord2f(1.0, 1.0);
        tmp.x=position.x+(right.x + up.x)*_billboards[i].width;
        tmp.y=position.y+(right.y + up.y)*_billboards[i].height;
        tmp.z=position.z+(right.z + up.z)*_billboards[i].width;
```

```

        glVertex3f(tmp.x, (tmp.y)*2, tmp.z);

        glTexCoord2f(0.0, 1.0);
        tmp.x=position.x+(up.x - right.x)*_billboards[i].width;
        tmp.y=position.y+(up.y - right.y)*_billboards[i].height;
        tmp.z=position.z+(up.z - right.z)*_billboards[i].width;
        glVertex3f(tmp.x, (tmp.y)*2, tmp.z);
    glEnd();
}

```

Das war schon alles.

Zum Schluss vielleicht noch ein paar Worte zur Funktion

-(float)heightAtPosition:(float)x zPosition:(float)z

über welche die Y-Position der Kamera und der Billboards berechnet wird:

Wie Sie vielleicht gemerkt haben, »fliegt« die Kamera nicht über das Terrain, sondern sie folgt der Höhe an der entsprechenden Stelle (genau wie die Billboards, die auf dem Terrain stehen).

Das Ganze funktioniert folgendermaßen: Wir holen uns die absolute Position der Kamera innerhalb des Terrains und speichern diese:

```

// Terrain durch Terrain_Multiplikator teilen
// um die tatsaechliche Groesse zu erhalten
projCameraX = x / TERRAIN_MULTIPLIKATOR;
projCameraZ = z / TERRAIN_MULTIPLIKATOR;

```

Anschließend wird der Höhenwert des Terrains direkt unter der Kamera berechnet:

```

// Hoehenwerte vom Terrain direkt unter dem Objekt
int column0 = (int)projCameraX;
int row0 = (int)projCameraZ;
int column1 = column0 + 1;
int row1 = row0 + 1;

```

Jetzt werden die 4 Eckpunkte ermittelt, welche direkt an die Kameraposition angrenzen:

```

// Die 4 umliegenden Vertices holen

```



```
float height00 = TERRAIN_HEIGHTSCALE*(float)_map.data[column0 +  
row0*_mapSize];  
float height01 = TERRAIN_HEIGHTSCALE*(float)_map.data[column1 +  
row0*_mapSize];  
float height11 = TERRAIN_HEIGHTSCALE*(float)_map.data[column1 +  
row1*_mapSize];  
float height10 = TERRAIN_HEIGHTSCALE*(float)_map.data[column0 +  
row1*_mapSize];
```

Zum Schluss wird zwischen den ermittelten Werten bilinear interpoliert, dadurch erreichen wir einen weichen Übergang zwischen den vier Eckpunkten.

```
// Position des Objekts relativ zu den beiden Zellen  
float tx = projCameraX - (float)column0;  
float ty = projCameraZ - (float)row0;  
  
// Bilinear zwischen den 4 Werten interpolieren  
// um extreme Höhenunterschiede der 4 Vertices auszugleichen  
float txtty = tx * ty;  
float final_height = height00 * (1.0f - ty - tx + txtty)  
+ height01 * (tx - txtty)  
+ height11 * txtty  
+ height10 * (ty - txtty);  
return final_height;
```

Partikel

Einer der interessantesten Aspekte der Spiele-Entwicklung ist das Erstellen von Partikel-Systemen. Mit Hilfe dieser Partikel ist es möglich, Effekte wie z. B. Rauch, Feuer, Explosionen usw. zu erstellen. Bevor wir uns aber an die Arbeit machen, müssen wir klären, was Partikel überhaupt sind.

Ein Partikel ist zunächst einmal ein individuelles Element, welches über Eigenschaften wie z. B. Größe, Farbe und Position verfügt. Über diese Eigenschaften können wir das Verhalten und Aussehen der Partikel innerhalb unseres Programms steuern. Alle Partikel verhalten sich völlig unabhängig von einander, was bedeutet, dass wir zur Laufzeit des Programms sämtliche Partikel einzeln berechnen und rendern müssen.

Daraus ergeben sich folgende Anforderungen an unser Programm:

- Neue Partikel müssen erstellt werden.
- Allen Partikeln müssen eigene Eigenschaften zugewiesen werden.
- Partikel, die nicht mehr sichtbar sein sollen (tot), müssen als solche deklariert werden.
- Die Partikel müssen bewegt werden.
- Alle sichtbaren (nicht toten) Partikel müssen gerendert werden.

Wie man sehen kann, erfordert das Erstellen der Partikel einiges an Aufwand, was in der Regel durch sogenannte Partikel-Systeme gemacht wird, die sich um die Verwaltung der Partikel kümmern. Dazu kommen wir gleich.

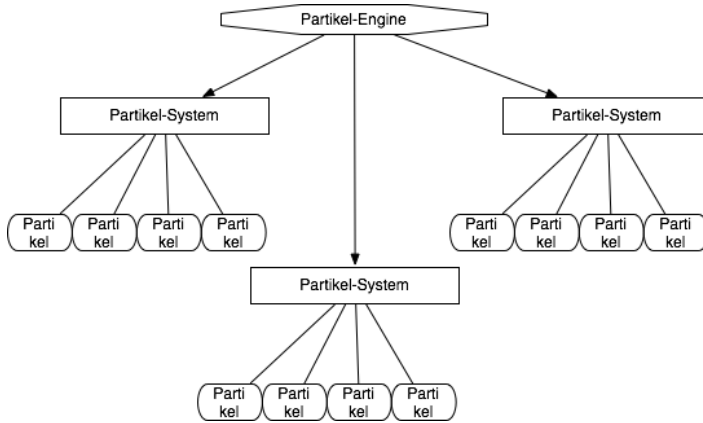
Schauen wir uns zunächst einmal die Eigenschaften eines Partikels genauer an.

- **Position**
Entspricht der Position im 3D- (oder 2D)-Raum und wird üblicherweise durch einen Vektor dargestellt.
- **Geschwindigkeit**
Der Faktor, über welchen die Partikel im Raum bewegt werden (Vektor).
- **Farbe**
Da bei Partikeln sehr oft Blending eingesetzt wird, speichert man zusätzlich zu dem RGB-Wert noch einen Alphawert (4 float-Werte).
- **Lebenszeit**
Darüber wird geregelt, wie lange ein Partikel sichtbar ist (float).
- **Größe**
Die Größe wird sehr oft von der Lebenszeit beeinflusst. Rauch wird, je länger er »lebt«, umso größer, wobei z. B. Funken mit der Zeit immer kleiner werden (float).
- **Darstellung**
Hier gibt es mehrere Möglichkeiten, man kann z. B. die Partikel über Billboards darstellen, über einfache Linien oder nur als Punkte. In der Regel verwendet man aber Billboards, da diese doch am besten aussehen.

Diese Eigenschaften sind nur ein kleiner Auszug dessen, was möglich ist. Es gibt durchaus Effekte, die noch viel mehr Eigenschaften benötigen.

Partikel-Systeme

Einfach ausgedrückt sind Partikel-Systeme nichts anderes als die Manager der Partikel, diese werden auch sehr oft als »Emitter« bezeichnet. Sie kümmern sich z. B. um das Bewegen, Erzeugen und Rendern der einzelnen Partikel. Die Idee, die dahinter steckt, ist die, dass, wenn wir einen bestimmten Effekt erzeugen möchten, wir uns nicht um die Verwaltung der einzelnen Partikel selbst kümmern müssen. Stattdessen erstellen wir einfach nur ein neues Partikel-System und geben diesem dann die Aufgabe.



Zusammenspiel einer Partikel-Engine

Die einzelnen Partikel-Systeme werden auch sehr oft zu einer Partikel-Engine zusammengefasst, die sich ihrerseits um die Partikel-Systeme kümmert.

Die Partikel-Systeme haben selbst auch wieder verschiedene Eigenschaften, über welche wir ihr Verhalten steuern können.

- **Position** - Die Position im Raum, von welcher aus neue Partikel erzeugt werden sollen (Vektor)
- **Blending** - Die Art, wie die Partikel übereinander geblendet werden sollen
- **Status** - Darüber wird geregelt, ob das ganze System noch sichtbar ist.

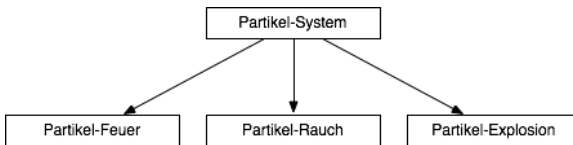
Auch das ist wieder nur ein kleiner Auszug an möglichen Eigenschaften, die nach Belieben erweitert werden können.

Wie man sieht, können Partikel-Systeme doch recht unterschiedlich sein (Rauch sieht nun mal anders aus als eine Explosion), was die Frage aufwirft, wie man sie am besten implementiert,

Die große Herausforderung besteht ja darin, so viele Effekte wie möglich darstellen zu können, und das alles in einer annehmbaren Geschwindigkeit.

Hier gibt es zwei Ansätze:

Der erste besteht darin, eine abstrakte Basis-Klasse zu erstellen, von der man dann die verschiedenen Systeme ableitet, welche ihrerseits für einen speziellen Effekt zuständig sind.



Aufbau einer Partikel-Engine über eine abstrakte Basis-Klasse

Der Vorteil dabei ist, dass der Arbeitsaufwand recht gering ist und man somit sehr schnell zu einem Ergebnis kommt. Leider hat diese Vorgehensweise einen großen Nachteil: Wenn nämlich Effekte benötigt werden, die dieses System nicht unterstützt, müssen Teile des Systems geändert bzw. erweitert werden, und zwar im Quellcode selbst!

Die zweite Variante wäre ein gescriptetes System (wird in kommerziellen Spielen wohl am meisten verwendet). Der große Vorteil dabei ist, dass es sehr flexibel ist, weil dabei nämlich der eigentliche Code für die Effekte zur Laufzeit geladen und geparkt wird.

Solch ein System könnte z. B. so aussehen:

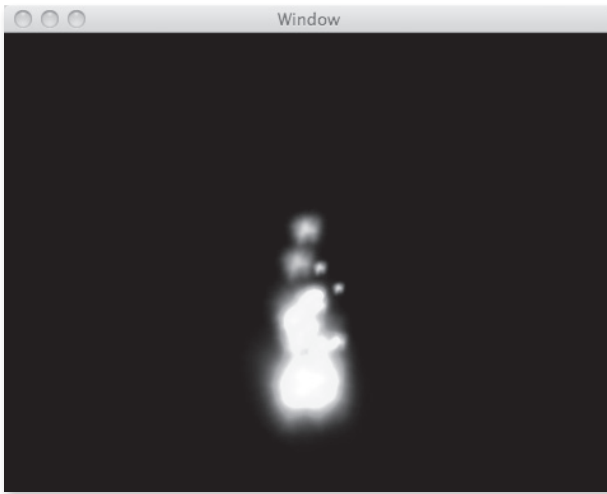
```

particleSystem snow 2.0
{
position: (0.0, 10.0, -23.2);
blendmode_source: GL_SRC_ALPHA;
blendmode_dest: GL_ONE;
numParticles: 1000;
gravity: 9.15;
emitradius: (random(0.0, 10.0) 20.0, 32.12 );
lifetime: 10;
texture: snow.tiff;
color: rgba(1.0, 1.0, 1.0, 1.0);
velocity: (0.1, -1.0, 0.0);
usw...
}
  
```

Hier würde es sich z. B. anbieten, die XML-Klassen von Cocoa zu verwenden, da man mit diesen ein solches System sehr einfach erstellen kann.

Partikel Feuer-Effekt

Nach der ganzen Theorie kommen wir zu unserem ersten Beispiel »Kapitel 10/ Fire«, welches einen Feuer-Effekt darstellen soll.



Ein einfacher Feuer-Effekt

Die Grundlage für diesen Effekt sind texturierte Billboards, welche über mehrere zufällig gesetzte Eigenschaften gerendert werden.

Zunächst erstellen wir eine Struktur für die einzelnen Partikel:

```
typedef struct particle
{
    CFXVector position; // Position
    CFXVector velocity; // Geschwindigkeit
    float age; // Lebensdauer
    float maxAge; // maximale Lebensdauer
    float size; // Groesse
}Particle;
```

Da unser Effekt recht einfach gehalten ist, kommen wir mit relativ wenigen Eigenschaften aus.

Auffällig ist vielleicht, dass gar keine Farbe definiert wurde. Der Grund dafür ist, dass wir eine Textur nutzen, die selbst schon die für Feuer typischen Farben enthält, welche wir später dann nur noch in der Standardfarbe (weiß) rendern müssen.

Danach werden die einzelnen Partikel initialisiert:

```
// Partikel initialisieren
int i;
for(i=0; i<NUM_PARTICLES; i++)
{
    _particles[i].position = makeVector(randomFloat(-0.2, 0.2),
    -1.5, -1.0);
    _particles[i].velocity = makeVector(randomFloat(-0.1, 0.1),
    randomFloat(0.1, 2.0),
    randomFloat(-0.1, 0.1));
    _particles[i].age = 0.0;
    _particles[i].maxAge = randomFloat(0.1, 1.0);
    _particles[i].size = randomFloat(0.1, 1.0);
}
```

Um das Feuer nicht so künstlich wirken zu lassen, werden die meisten Parameter mit Zufallswerten initialisiert. Alle diese Werte sind von mir willkürlich ausgewählt worden, probieren Sie ruhig einmal andere Werte aus.

Das Rendern erfolgt in der Regel in zwei Schritten. Im ersten Schritt werden die einzelnen Partikel durchlaufen und aktualisiert (Position, Größe, usw.). Müssten aufwendige physikalische Berechnungen (Kollisionsabfrage etc.) gemacht werden, wäre hier der richtige Platz dafür.

```
-(void)updateParticles
{
    int i;
    for(i=0; i<NUM_PARTICLES; i++)
    {
        // Position aktualisieren
        _particles[i].position.x+= _particles[i].velocity.x *
        _deltaTime;
        _particles[i].position.y+= _particles[i].velocity.y *
        _deltaTime;
        _particles[i].position.z+= _particles[i].velocity.z *
        _deltaTime;
    }
}
```

```

    // Partikel altern lassen
    _particles[i].age +=_deltaTime;

    // je laenger die Partikel sichtbar sind, umso kleiner
    // werden sie
    _particles[i].size -=0.6*_deltaTime;

    // Wenn sie nicht mehr zu sehen sind, werden sie neu
    // initialisiert
    if( _particles[i].size <= 0.0)
    {
        _particles[i].position = makeVector(randomFloat(-0.2,
        0.2), -1.5, -1.0);
        _particles[i].velocity = makeVector(randomFloat(-0.1,
        0.1),randomFloat(0.1, 2.0),randomFloat(-0.1, 0.1));
        _particles[i].age = 0.0;
        _particles[i].maxAge = randomFloat(0.1, 1.0);
        _particles[i].size = randomFloat(0.1, 1.0);
    }
}
}

```

Im zweiten Schritt werden sie dann gerendert:

```

-(void)renderParticles
{
    // Billboarding
    float fMatrix[16];
    glGetFloatv( GL_MODELVIEW_MATRIX, fMatrix );
    CFXVector right = makeVector(fMatrix[0], fMatrix[4], fMatrix[8]
    );
    CFXVector up = makeVector(fMatrix[1], fMatrix[5], fMatrix[9] );

    CFXVector tmp;
    int i;

    // Blending einschalten
    glEnable(GL_BLEND);

    //Blendfunktion festlegen
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
}

```

```
// Nicht in den Z-Puffer schreiben
glDepthMask(GL_FALSE);

for(i=0; i<NUM_PARTICLES; i++)
{
    CFXVector position = _particles[i].position;

    //Partikel anhand ihres "Alters" faden
    glColor4f(1.0, 1.0, 1.0, _particles[i].age);

    glBegin(GL_QUADS);
        glTexCoord2f(0.0, 0.0);
        tmp.x=position.x+(right.x + up.x)*_particles[i].size;
        tmp.y=position.y+(right.y + up.y)*_particles[i].size;
        tmp.z=position.z+(right.z + up.z)*_particles[i].size;
        glVertex3f(tmp.x, tmp.y, tmp.z);

        glTexCoord2f(1.0, 0.0);
        tmp.x=position.x+(right.x - up.x)*_particles[i].size;
        tmp.y=position.y+(right.y - up.y)*_particles[i].size;
        tmp.z=position.z+(right.z - up.z)*_particles[i].size;
        glVertex3f(tmp.x, tmp.y, tmp.z);

        glTexCoord2f(1.0, 1.0);
        tmp.x=position.x+(right.x + up.x)*_particles[i].size;
        tmp.y=position.y+(right.y + up.y)*_particles[i].size;
        tmp.z=position.z+(right.z + up.z)*_particles[i].size;
        glVertex3f(tmp.x, (tmp.y), tmp.z);

        glTexCoord2f(0.0, 1.0);
        tmp.x=position.x+(up.x - right.x)*_particles[i].size;
        tmp.y=position.y+(up.y - right.y)*_particles[i].size;
        tmp.z=position.z+(up.z - right.z)*_particles[i].size;
        glVertex3f(tmp.x, (tmp.y), tmp.z);
    glEnd();

}
glDisable(GL_BLEND);
glDepthMask(GL_TRUE);
}
```


Hier ist es zunächst einmal wichtig, OpenGL daran zu hindern, Werte in den Z-Puffer zu schreiben `glDepthMask(GL_FALSE)`; Der Grund hierfür ist, dass OpenGL ja versucht, Polygone anhand ihrer Z-Position zu sortieren. Wenn nun aber diese Polygone identische bzw. sehr ähnliche Z-Positionen haben, scheitert die Sortierung, und es kommt zu unschönen Artefakten, was wir ja nicht wollen.

Danach schalten wir das Blending ein und definieren die Blend-Funktion. Das Rendern der Partikel ist identisch mit dem der Billboards aus dem ersten Beispiel.

Zum Schluss wird das Blending wieder ausgeschaltet und das Schreiben in den Z-Puffer wieder aktiviert.

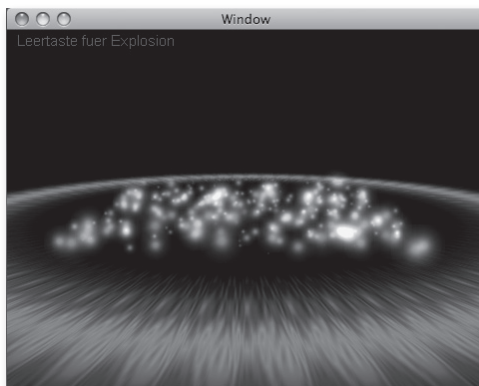
POWER

Der Renderprozess ist nicht besonders schnell, weshalb man hier unbedingt auf Vertex-Arrays bzw. VBOs umstellen sollte.

Sie sehen, der Aufwand ist recht gering, wobei das Ergebnis gar nicht so schlecht aussieht. Wenn Sie ein wenig mit den einzelnen Parametern herumspielen, werden Sie sehr schnell merken, dass es ziemlich zermürend sein kann, den gewünschten Effekt zu erzielen. Das ist leider ein weiterer Nachteil dieser hardcodierten Partikel-Systeme.

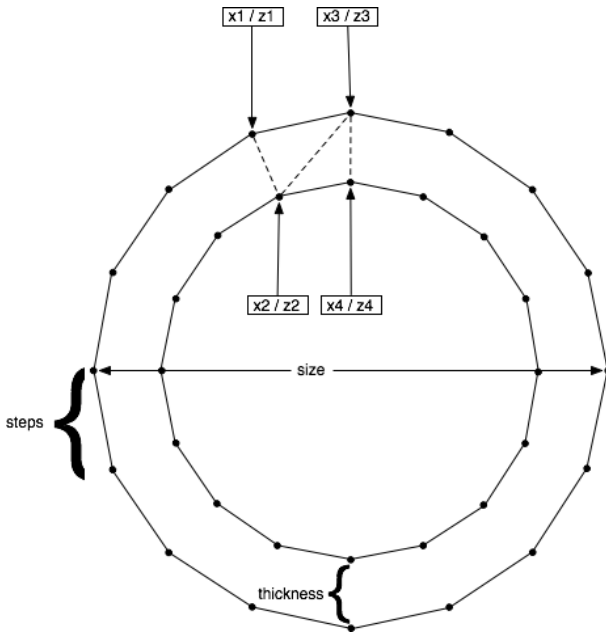
Shockwave-Effekt

In vielen Konsolenspielen bzw. Weltraum-Shootern sieht man sehr oft Explosionen, welche sich mit einzelnen Partikeln nur sehr schlecht bzw. gar nicht darstellen lassen. Wir wollen uns im nächsten Beispiel einmal einen Effekt anschauen, der mit Hilfe von Triangle-Strips eine Schockwelle erzeugt, die recht imposant wirkt. Um den Effekt noch ein wenig zu erhöhen, bauen wir noch einen weiteren dazu, welcher aus umherfliegenden Partikeln besteht.



Schockwelle mit Explosion

Beginnen wir mit dem Erstellen der Schockwelle »Kapitel 10/Shockwave«, welche aus einem texturierten Ring besteht, der wiederum aus Triangle-Strips zusammengesetzt wurde. Dieser Ring wird dann später einfach mit Hilfe des *glScale(...)*-Befehls skaliert. Damit dieser nicht unendlich groß wird, faden wir ihn mit dem Alphawert von *glColor(...)* langsam aus.



Ein Ring, der aus Triangle-Strips zusammengesetzt wurde

Um den Rendervorgang zu beschleunigen, wird dieser Ring in einer Display-List gespeichert, da er sich während des Zeichnens ja nicht mehr verändert. Der Code dazu sieht folgendermaßen aus:

```
// DisplayList erzeugen
-(void)buildDisplayListForShockWave
{
    float thickness = 0.5;
    float steps = 20.0f;
    float size = 1.0;

    _shockwaveList = glGenLists(1);
    glNewList(_shockwaveList, GL_COMPILE);

    glBindTexture(GL_TEXTURE_2D, _shockwaveTexture);
```

```

glBegin(GL_TRIANGLE_STRIP);
    float i;
    for(i=0.0f; i<360.0f; i+=steps)
    {
        glTexCoord2f(0.0, 1.0);
        float x1 = size *sin(DEG2RAD(i));
        float z1 = size *cos(DEG2RAD(i));
        glVertex3f(x1, 0.0, z1);

        glTexCoord2f(0.0, 0.0);
        float x2 = (size -thickness)*sin(DEG2RAD(i));
        float z2 = (size -thickness)*cos(DEG2RAD(i));
        glVertex3f(x2, 0.0, z2);

        glTexCoord2f(1.0, 1.0);
        float x3 = size *sin(DEG2RAD(i+steps));
        float z3 = size *cos(DEG2RAD(i+steps));
        glVertex3f(x3, 0.0, z3);

        glTexCoord2f(1.0, 0.0);
        float x4 = (size-thickness) *
sin(DEG2RAD(i+steps));
        float z4 = (size-thickness) *
cos(DEG2RAD(i+steps));
        glVertex3f(x4, 0.0, z4);
    }
    glEnd();
    glEndList();
}

```

Da wir einen Kreis bauen wollen, verwenden wir die beiden trigonometrischen Funktionen *sin()* bzw. *cos()*, um die XZ- Positionen der Vertices zu berechnen. Für die Y-Achse können wir einen fixen Wert nehmen, da der Ring ja flach auf dieser Achse liegt. Über den Parameter *steps* können wir die Rundung des Rings beeinflussen, je kleiner dieser Wert ist, umso runder wird unsere Schockwelle, was aber auch mehr Rechenleistung benötigt.

Die Explosion

Das Erzeugen der Explosion ist recht unspektakulär, es werden die einzelnen Parameter wieder mit Zufallswerten initialisiert. Auch hier habe ich die Werte völlig willkürlich gewählt.

Neu dazu gekommen ist nun eine Farbe für die Partikel, und zwar deshalb, weil die Textur jetzt nur noch aus Grauwerten besteht. Der Vorteil dieser Vorgehensweise ist zum einen, dass nun die Farben für alle Partikel frei gewählt werden, und zum Anderen, dass die Textur auch für andere Effekte (mit anderen Farben) verwendet werden kann.

```
-(void)buildExplosion
{
    int i;

    for (i=0; i<PARTICLE_EXPLOSION; i++)
    {
        _p[i].velocity.x = randomFloat(-6.0, 6.0);
        _p[i].velocity.y = randomFloat(-0.5, 0.5);
        _p[i].velocity.z = randomFloat(-6.0, 6.0);

        _p[i].position = makeVector(0.0, 0.0, 0.0);
        _p[i].maxAge = randomFloat(0.1, 5.0);
        _p[i].age = 0.0;
        _p[i].size = randomFloat(0.1, 0.5);
        _p[i].color = makeColor(0.7, 0.41, 0.1, 0.0);

        _p[i].isDead = NO;
    }
}
```

Auch die *update*-Funktion der Partikel hat sich nicht wirklich geändert. Der einzige Unterschied besteht nun darin, dass »tote« Partikel nicht mehr neu erzeugt werden und dadurch später auch nicht mehr gerendert werden müssen. Ein Partikel gilt als »tot«, wenn entweder seine Größe kleiner als 0.0 ist oder aber seine Lebensdauer die maximale Lebensdauer überschritten hat.

```
-(void)updateExplosion
{
    int i;
    for (i=0; i<PARTICLE_EXPLOSION; i++)
    {
        _p[i].age+=_deltaTime;
        _p[i].size-= 1.5*(_p[i].age / _p[i].maxAge)*_deltaTime;

        _p[i].position.x+=_p[i].velocity.x*_deltaTime;
        _p[i].position.y+=_p[i].velocity.y*_deltaTime;
```

```

        _p[i].position.z+=_p[i].velocity.z*_deltaTime;

        if( (_p[i].age >= _p[i].maxAge) || _p[i].size <=0.0)
        {
            _p[i].isDead = YES;
        }
    }
}

```

Die *Render*-Funktion der Partikel können wir außen vor lassen, da sie identisch mit der aus dem letzten Beispiel ist. Weshalb wir uns nur noch die *drawRect*-Methode anschauen müssen:

```

if(_go)
{
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    [self updateExplosion];
    [self renderExplosion];

    glPushMatrix();
        glColor4f(1.0, 1.0, 1.0, _alpha);
        glScalef(_scale, 1.0, _scale);
        glCallList(_shockwaveList);
    glPopMatrix();

    glDisable(GL_BLEND);

    _scale+=15.0*_deltaTime;

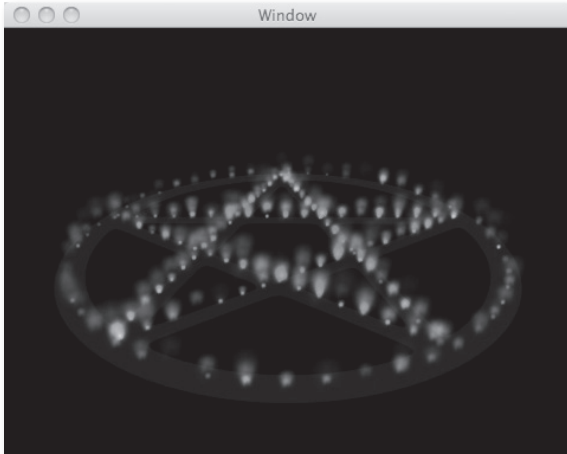
    _alpha-=1.0*_deltaTime;
}

```

Der Effekt wird beim Drücken der Leertaste gestartet (*_go*), dabei wird zunächst die Explosion gerendert und anschließend die Schockwelle. Bei dieser wird, wie schon oben erwähnt, der Alpha-Wert (*_alpha*) langsam verringert (ausfaden) und der Skalier-Faktor erhöht, um sie »wachsen« zu lassen. Mehr ist nicht nötig. Wie man sehen kann, ist der Aufwand recht gering, wobei der Effekt schon recht spektakulär aussieht.

Das Pentagramm

Im nächsten Beispiel zu den Partikeln »Kapitel 10/Trail« erstellen wir ein rotierendes Partikel-System, welches die Form eines Pentagramms haben soll. Dieses erinnert ein wenig an das Spiel Diablo von Blizzard, welches einen ähnlichen Effekt beinhaltet.



Ein Pentagramm, das aus Partikeln zusammengesetzt wurde

Beginnen wir mit der *init*-Methode der Partikel:

```
// Partikel initialisieren
-(void)initParticle:(int)number
{
    // Positionen der Partikel werden zufaellig aus der Datei
    // Models.model ausgewaehlt
    int ra = rand()%239;
    _p[number].position = makeVector(pentagram[ra][0],
    pentagram[ra][1], pentagram[ra][2]);

    // Farbe der Partikel ist abhaengig von der Distanz zum
    // Mittelpunkt
    CFXVector center = makeEmptyVector();
    CFXVector dist = vectorSubtract(_p[number].position, center);
    float length = lengthOfVector(dist);
    length/=10.0;

    _p[number].color = makeColor(1-length, 0.15,  length, 1.0);

    _p[number].isDead = NO;
}
```

```

    _p[number].age = 0.0;
    _p[number].maxAge = randomFloat(0.5, 2.0);
    _p[number].size = 0.05;
    _p[number].velocity = makeVector(0.0, randomFloat(0.1, 0.4),
    0.0);
}

```

Die einzelnen Positionen der Partikel werden nun nicht mehr willkürlich gesetzt, sondern kommen aus einem Array in der Datei »Models.model«, wo die Vertex-Daten für die Form des Pentagramms definiert sind.

Aus diesen Werten wählen wir eine zufällige Position für die Partikel aus.

Die Farben der Partikel reichen von Rot bis Lila. Diese werden abhängig von den Partikel-Positionen zum Mittelpunkt berechnet. Das heißt, je weiter ein Partikel vom Ursprung entfernt ist, desto mehr geht seine Farbe in einen Lila-Ton. Dadurch erhalten wir einen schönen Farbverlauf von innen nach außen. An den beiden Methoden (*updateParticle*, *renderParticle*) hat sich zu den vorhergehenden Beispielen nichts geändert, weshalb wir uns wiederum nur die *drawRect*-Methode anschauen.

```

// Blending fuer Boden-Textur einschalten
// Pentagram-Muster mit Hilfe des Alpha-Kanals der Textur
// festlegen
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glColor4f(1.0, 1.0, 1.0, 0.1);
glBindTexture(GL_TEXTURE_2D, _pentagramTexture);

// Boden rendern
glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);glVertex3f(-5.0, -0.1, 5.0);
    glTexCoord2f(1.0, 0.0);glVertex3f(5.0, -0.1, 5.0);
    glTexCoord2f(1.0, 1.0);glVertex3f(5.0, -0.1, -5.0);
    glTexCoord2f(0.0, 1.0);glVertex3f(-5.0, -0.1, -5.0);
glEnd();

// Partikel
[self updateParticle];
[self renderParticle];

glDisable(GL_BLEND);

```

Um die Szene noch ein wenig düsterer zu machen, wird unter dem Partikel-System ein Quadrat gerendert, dessen Textur auch die Form eines Pentagramms hat. Diese Form wiederum wird im Alpha-Kanal der Textur gespeichert, was folgende Abbildung zeigt.



Die Textur für den Boden: auf der linken Seite der Farbverlauf und auf der rechten Seite der Alpha-Kanal

Durch das aktive Blending wird dann nur die Pentagramm-Form aus der Textur »gestanzt«. Den Rest der *drawRect*-Methode kennen Sie ja bereits.

Point-Sprites

Point-Sprites sind **hardwarebeschleunigte** Billboards, die primär für die Entwicklung von Partikel-Systemen gemacht wurden.

Wie der Name schon sagt, sind Point-Sprites nur Punkte (*GL_POINTS*) und keine Polygone, weshalb sie sehr schnell (auch in großer Anzahl) gerendert werden können. Hardwarebeschleunigt bedeutet in diesem Fall, dass die Ausrichtung der Punkte zur Kamera (Billboarding) und die Generierung der Textur-Koordinaten von OpenGL selbst übernommen werden, so dass man sich nicht mehr selbst darum kümmern muss.

Die Verwendung von Point-Sprites ist relativ simpel und erfolgt in der Regel in drei Schritten:

1. Point-Sprites aktivieren
2. Textur-Parameter auf *GL_COORD_REPLACE* setzen (Erklärung folgt gleich)
3. Partikel mit Hilfe von *GL_POINTS* rendern

Auch hierzu zuerst wieder ein Beispiel »Kapitel 10/Point Sprites«, welches auf dem Programm »Kapitel 10/Fire« von weiter oben basiert. Dieses wurde so modifiziert, dass die Partikel nun mit Hilfe von Point-Sprites gerendert werden, weshalb wir uns nur die relevanten Änderungen anschauen müssen.

Zuerst aktivieren wir die Point-Sprites:

```
glEnable(GL_POINT_SPRITE);
```

Die Generierung der Textur-Koordinaten erfolgt durch OpenGL selbst. Wir setzen lediglich die Textur-Umgebung so, dass die Textur-Koordinaten interpoliert über den Punkt dargestellt werden.

```
glTexEnv(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
```

TIPP

Für die Texturierung der Point-Sprites gelten dieselben Regeln wie auch für Polygone, das heißt, man kann ihre Umgebung setzen (GL_MODULATE, GL_REPLACE, ...) oder aber auch mehrere Texturen auf einmal aufbringen (Multi-Texturing).

Da Punkte nicht beliebig groß von OpenGL dargestellt werden können, müssen wir die maximale Ober- und Untergrenze abfragen, was folgende Funktion macht:

```
float _pointSpriteSizes[2];  
// Minimale und maximale Größe von Punkten (GL_POINTS) bestimmen.  
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, _pointSpriteSizes);
```

Der untere Wert `_pointSpriteSizes[0]` ist standardmäßig 0.0, wobei die Obergrenze (`_pointSpriteSizes[1]`) wie gesagt auf einen bestimmten Wert beschränkt ist.

Das Initialisieren und Aktualisieren der Partikel ist vollkommen identisch zum oberen Beispiel geblieben, weshalb wir gleich zum Rendervorgang gehen können:

```
-(void)renderParticles  
{  
    int i;  
  
    // Blending einschalten  
    glEnable(GL_BLEND);
```

```
//Blendfunktion festlegen
glBlendFunc(GL_SRC_ALPHA, GL_ONE);

// Nicht in den Z-Puffer schreiben
glDepthMask(GL_FALSE);

// Maximale Point-Size setzten
glPointSize(_pointSpriteSizes[1]);

glBegin(GL_POINTS);
for(i=0; i<NUM_PARTICLES; i++)
{
    glColor4f(1.0, 1.0, 1.0, _particles[i].age);
    glVertex3f(_particles[i].position.x, _particles[i].position.y, _particles[i].position.z);
}
glEnd();

glDisable(GL_BLEND);
glDepthMask(GL_TRUE);
}
```

Das Auslesen der Modelviewmatrix ist komplett weggefallen, da wie gesagt OpenGL die Ausrichtung der Geometrie zur Kamera übernimmt. Da wir nun nur noch Punkte rendern müssen, ist auch der Code dazu entsprechend kurz geworden.

Wir setzen zunächst die maximale Punktgröße für unsere Partikel-Größe ein (hier kann man natürlich auch kleinere Werte nehmen) und rendern anschließend unsere Partikel mit Hilfe von Punkten. Der Rest der Render-Methode ist gleich geblieben.

Vergleicht man beide Beispiele, merkt man sofort, dass der Aufwand mit den Point-Sprites sehr viel geringer ist, weshalb man sich fragen darf, weshalb man überhaupt noch Billboards nutzt, schließlich lassen sich Point-Sprites auch schneller rendern.

Point-Sprites haben einige Einschränkungen gegenüber »handgemachten« Billboards:

- Sie können nicht beliebig groß sein (haben wir oben gesehen).
- Da sie nur Punkte sind, kann man sie nicht rotieren.

- Da sie relativ neu sind, sind sie unter Umständen nicht auf allen Systemen als Kernfunktion von OpenGL enthalten. Im Notfall kann man sich mit der Extension (*GL_POINT_SPRITE_ARB*) weiterhelfen.
- Sie werden immer auf allen 3 Achsen ausgerichtet, was manchmal gar nicht erwünscht ist.

Trotz dieser Einschränkungen (den dritten Punkt mal außen vor gelassen), haben Point-Sprites gegenüber handgemachten Billboards einige Vorteile, weshalb man ihnen, wann möglich, den Vorzug geben sollte.

Nebel

OpenGL unterstützt »von Haus« aus Nebel, der sehr oft unverzichtbar bei der Entwicklung von Spielen ist. Mal abgesehen davon, dass man mit Nebel sehr viel Atmosphäre in ein Spiel (meist Outdoor) bringen kann, wird er oft dazu benutzt, Objekte, die sich darin befinden (und dadurch nicht bzw. nur teilweise sichtbar sind), aus dem Renderprozess auszuschließen.

Der Nebel wird in OpenGL mit Hilfe von Blending realisiert, wobei die Nebelfarbe abhängig von einigen Faktoren (dazu gleich mehr) über die bestehenden Pixel geblendet werden.

Wie gesagt ist der Nebel (oder Fogging) in OpenGL schon integriert, weshalb die Benutzung recht einfach ist.

Nachdem man mit *glEnable(GL_FOG)* den Nebel eingeschaltet hat, kann man ihn mit Hilfe der Funktion *glFogf()* (welche es auch wieder in verschiedenen Versionen, *glFogf(...)*, *glFogi(...)* usw. gibt) an die jeweiligen Bedürfnisse anpassen, was wir uns nun einmal anschauen wollen.

Nebel-Parameter

GL_FOG_MODE legt die Gleichung des Blendfaktors (*GL_LINEAR*, *GL_EXP* oder *GL_EXP2*) fest.

Bei *GL_LINEAR* müssen zusätzlich die beiden Werte für *GL_FOG_START* bzw. *GL_FOG_END* angegeben werden. *GL_FOG_START* sagt aus, ab welcher Entfernung eines Objektes zur Kamera der Nebeleffekt einsetzen soll. *GL_FOG_END* wird dazu benutzt, um den Wert festzulegen, ab welchem ein Objekt komplett im Nebel verschwindet.

Ein kleines Beispiel: Ist ein Objekt näher am Betrachter, als über `GL_FOG_START` angegeben wurde, wird es ganz normal gezeichnet, liegt es weiter entfernt, als über `GL_FOG_END` definiert wurde, wird es komplett in der Nebel-Farbe (`GL_FOG_COLOR`) gezeichnet, liegt das Objekt irgendwo zwischen diesen beiden Werten, wird linear interpoliert.

Hat man anstatt `GL_LINEAR` den Wert `GL_EXP` bzw. `GL_EXP2` angegeben, sind die beiden Werte für Start und End gleichgültig, da der Nebel nun mit einer exponentiellen Gleichung berechnet wird. Das bedeutet, je höher der Wert der Nebeldichte (`GL_FOG_DENSITY`), umso schneller verschwindet ein Objekt im Nebel.

Fassen wir zusammen:

- `GL_FOG_MODE` legt die Gleichung fest (`GL_LINEAR`, `GL_EXP`, `GL_EXP2`).
- `GL_FOG_DENSITY` ist ein positiver float-Wert, der die Nebeldichte angibt (Standard 1.0).
- `GL_FOG_START` Startwert für den Nebel
- `GL_FOG_END` Endwert für den Nebel
- `GL_FOG_INDEX` Farbindex der Nebelfarbe (bei 8-bit Farben). Wird heute nicht mehr genutzt.
- `GL_FOG_COLOR` Nebel-Farbe (array) Standard, ist schwarz.

Wollen wir einen linearen Nebel (welcher gleichmäßiger aussieht), brauchen wir zusätzlich die beiden Werte für Start- und End.

Möchten wir dagegen einen exponentiellen Nebel (welcher gleich zu Anfang sehr stark ist), benötigen wir zusätzlich den Wert für die Nebeldichte (`GL_FOG_DENSITY`).

Zusätzlich gibt es noch die Möglichkeit, mittels `glHint(GL_FOG_HINT, ...)`; die Render-Qualität des Nebels zu beeinflussen.

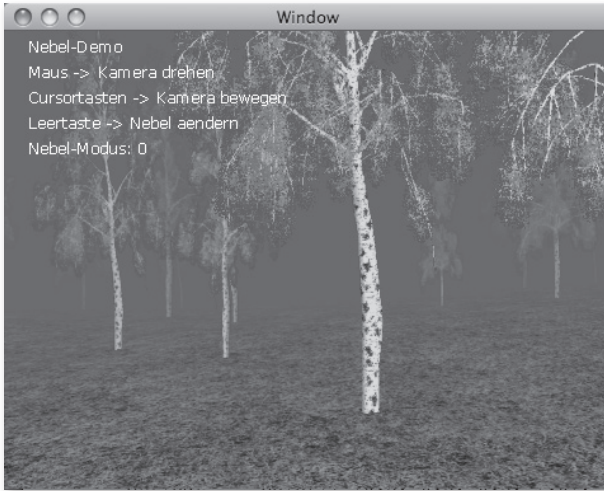
Der Aufruf von `glHint(GL_FOG_HINT, GL_NICEST)`; liefert das bessere Ergebnis, da der Nebel per Pixel berechnet wird, wobei `glHint(GL_FOG_HINT, GL_FASTEST)`; nicht ganz so gut aussieht (Nebel-Berechnung per Vertex), aber schneller in der Ausführungsgeschwindigkeit ist.

Auch hier möchte ich nochmals darauf hinweisen, dass diese »Hints« sehr stark vom Treiber abhängig sind und deshalb nicht auf allen Systemen gleichermaßen behandelt werden.

TIPP

Im Kapitel 12 (Shader) zeige ich Ihnen, wie Sie auch ohne »Hints« einen sehr gut aussehenden Nebel erstellen können.

Das Beispielprogramm »Kapitel 10/Fog« ist eine Erweiterung des Billboards-Beispiels von oben, welches nun aber mit aktiviertem Nebel gerendert wird.



Billboards-Szene mit aktiviertem Nebel

Über die Leertaste kann zwischen den 3 verschiedenen Nebel-Arten umgeschaltet werden. Hier die Änderung des Programms:

Zu Anfang wird der Nebel aktiviert und auf linear eingestellt:

```
// Nebel
glEnable(GL_FOG);
GLfloat color[]={0.3, 0.3, 0.3, 1.0};
glFogfv(GL_FOG_COLOR, color);
glFogf(GL_FOG_START, 2.0);
glFogf(GL_FOG_END, 400.0);
glFogi(GL_FOG_MODE, GL_LINEAR);
```

Die Nebel-Farbe ist gleich der Lösch-Farbe (*glClearColor(...)*), wodurch der Hintergrund komplett mit dem Nebel verschmilzt.

Später im Programm wird nur noch auf den Tastendruck der Leertaste reagiert und entsprechend der Modus für den Nebel geändert:

```
- (void)keyUp:(NSEvent *)theEvent
{
    switch([theEvent keyCode])
    {
        case 49:
        {
            _currentFogMode++;
            if(_currentFogMode > 2)
                _currentFogMode = 0;

            if(_currentFogMode == FOG_LINEAR)
            {
                glFogf(GL_FOG_START, 2.0);
                glFogf(GL_FOG_END, 400.0);
                glFogi(GL_FOG_MODE, GL_LINEAR);
            }
            else if(_currentFogMode == FOG_EXP)
            {
                glFogf(GL_FOG_DENSITY, 0.1);
                glFogi(GL_FOG_MODE, GL_EXP);
            }
            else if(_currentFogMode == FOG_EXP2)
            {
                glFogf(GL_FOG_DENSITY, 0.1);
                glFogi(GL_FOG_MODE, GL_EXP2);
            }
        }

        .
        .
        .
    }
}
```

Die Szene sieht doch schon besser aus, wobei der lineare Nebel meiner Meinung nach das schönste Ergebnis liefert.

Die Handhabung des Nebels ist recht einfach, liefert aber nicht immer das gewünschte Ergebnis. Stellen Sie sich z. B. eine Szene vor, in der die Nebelschwaden über einer Wiese liegen.

Mit den bisher gezeigten Methoden lässt sich solch ein Effekt nicht erstellen. Das liegt einfach daran, dass OpenGL die Koordinaten des Nebels selbst berechnet und so die **ganze** Szene damit füllt. Wir werden uns nun ein Beispiel anschauen, in dem wir diesen Nebelschwaden-Effekt nachbauen.

Volumetrischer Nebel

Diese Art von Nebel, welche übrigens auch in vielen kommerziellen Spielen genutzt wird, basiert darauf, dass ein ganz bestimmter Bereich (Volumen) definiert wird, in welchem der Nebel zu sehen sein soll. Dieses Volumen wird im folgenden Beispiel mit Hilfe der Funktion:

```
void glFogCoordf(GLfloat fFogDistance);
```

definiert.

Diese erwartet als Parameter die Nebel-Distanz zum Betrachter pro Vertex. Dieser Wert wird dann durch OpenGL über das komplette Polygon interpoliert.

Das Beispiel »Kapitel 10/Volumetric Fog« zeigt diese Technik, indem ein Terrain gerendert wird, in welchem Nebelschwaden über dem Boden liegen.



Volumetrischer Nebel in einem Terrain

Schauen wir uns nun die wichtigsten Codeausschnitte dazu an. Beim Start des Programms wird zunächst wieder der Nebel eingeschaltet und definiert:

```
// Nebel
float fogColor[4] = {0.87f, 0.83f, 0.75f, 1.0f};
glEnable(GL_FOG);
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_START, 0.0);
glFogf(GL_FOG_END, 60.0);
```

Neu hinzugekommen ist nun diese Funktion:

```
glFogi(GL_FOG_COORDINATE_SOURCE, GL_FOG_COORDINATE);
```

Über diese teilen wir OpenGL mit, dass wir uns selbst um die Nebel-Koordinaten kümmern möchten.

GRUNDLAGEN

Um wieder auf den Nebel umzuschalten, welcher durch OpenGL berechnet werden soll, ändern wir einfach den letzten Parameter:

```
glFogi(GL_FOG_COORDINATE_SOURCE, GL_FRAGMENT_DEPTH);
```

Da der Nebel nun per Vertex angegeben wird, müssen wir die Nebel-Koordinaten (*glFogCoordf(...)*) innerhalb der Berechnung unseres Terrains angeben:

```
for( z=0; z<_mapSize-1; z++ )
{
    glBegin( GL_TRIANGLE_STRIP );
    for( x=0; x<_mapSize-1; x++ )
    {
        // Texture-Koordinaten berechnen
        left  = ( float )x/_mapSize;
        bottom= ( float )z/_mapSize;
        top   = ( float )( z+1 )/_mapSize;

        // Nebel-Koordinaten Vertex 1
        tmp=[self heightAtPosition:x zPosition:z];
        [self setFogCoord:tmp];

        // Texture-Koordinaten Vertex 1
        glTexCoord2f(left*TEXTURE_STEP, bottom*TEXTURE_STEP);

        // Vertex 1
```



```

    // Textur soll um TEXTURE_STEP ueber das Terrain
    // gekachelt werden
    y = [self scaledHeightAtPosition:x zPosition:z];
    glVertex3f( (float)x, y, (float)z );

    // Nebel-Koordinaten Vertex 2
    tmp=[self heightAtPosition:x zPosition:z+1];
    [self setFogCoord:tmp];

    // Texture-Koordinaten vertex 2
    // Textur soll um TEXTURE_STEP ueber das Terrain
    // gekachelt werden
    glTexCoord2f(left*TEXTURE_STEP, top*TEXTURE_STEP);

    // Vertex 2
    y = [self scaledHeightAtPosition:x zPosition:z+1];
    glVertex3f( (float)x, y, (float)z+1 );
}
glEnd( );
}

```

Die Berechnung der eigentlichen Nebel-Koordinaten wurde in eine separate Methode ausgelagert, um den Code übersichtlicher zu halten.

```

-(void)setFogCoord:(float)terrainHeight
{
    float fogY = 0;

    // Wenn die Terrain-Hoehe hoeher als die Nebel-Tiefe dann
    kein Nebel
    if(terrainHeight > FOG_DEPTH)
        fogY = 0;
    // Nebel-Tiefe basierend auf der Terrain-Hoehe
    else
        fogY = -(terrainHeight - FOG_DEPTH);

    glFogCoordf(fogY);
}

```

Die Nebelschwaden entstehen folgendermaßen:

Es wird geprüft, ob die Höhe des Terrains höher ist als die Nebel-Höhe (*#define FOG_DEPTH 80.0*), wenn ja, wird kein Nebel gezeichnet (Nebel-Koordinate = 0.0), wenn nicht, wird die Nebeltiefe anhand der Höhe des Terrains berechnet, wodurch wir den besagten Effekt erhalten.

POWER

Die Angabe der Nebel-Koordinaten funktioniert natürlich auch mit den beiden Nebel-Modi `GL_EXP` bzw. `GL_EXP2`.

Render-Alternative

Das Rendern von Nebel muss nicht unbedingt über die eingebaute Funktion von OpenGL erfolgen, alternativ könnte man dies auch mit Hilfe von einem Partikelsystem erledigen. Dadurch wäre es sehr einfach, den Nebel zu bewegen bzw. seine Farbe zu ändern. Zum Beispiel könnte man damit Schwefeldampf erzeugen, der aus einem defekten Rohr austritt.

Im Kapitel über Shader werden wir uns wie schon gesagt noch einmal mit dem Nebel beschäftigen, und zwar sehen wir dann, wie man die Nebelfarbe noch gezielter beeinflussen kann, was sehr interessante Effekte ermöglicht.

Zusätzliche Informationen

<http://www.gamedev.net/reference/articles/article672.asp>

Zwei gute Artikel über volumetrischen Nebel

<http://www.wondertouch.com/>

Ein Partikel-Editor, der keine Wünsche übrig lässt. Eignet sich sehr gut dazu, wenn man Anregungen für eigene Effekte zum Nachbauen benötigt.

<http://www.particlesystems.org/>

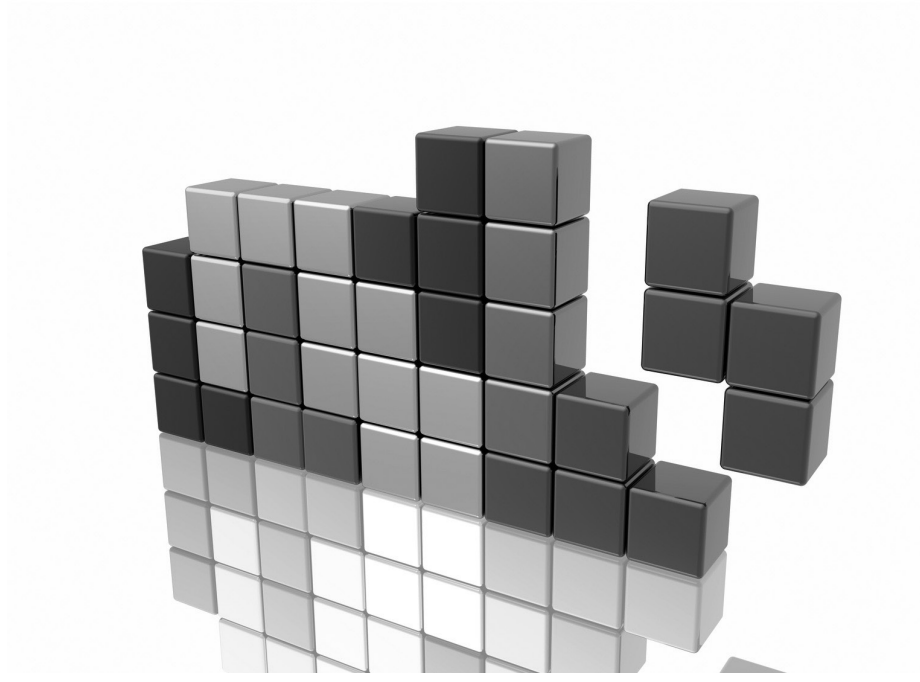
http://www.gamasutra.com/features/20000623/vanderburg_01.htm

<http://www.2ld.de/gdc2007/>

<http://www.2ld.de/gdc2004/>

<http://www.2ld.de/gh2004/>

Sehr gute Seiten, die viele Informationen zu Partikel-Systemen beinhalten



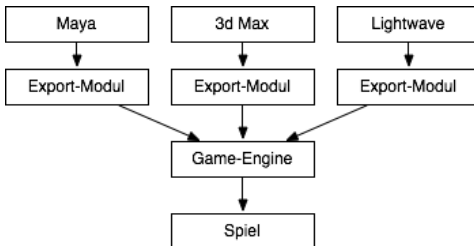
3D-Models

Wenn wir anfangen, Computerspiele zu entwickeln, stehen wir irgendwann vor der Aufgabe, 3D-Modelle in unser Spiel einzubauen. In den vergangenen Kapiteln haben wir ja unsere 3D-Objekte mehr oder weniger von Hand »modelliert«, was auch bis jetzt völlig ausreichend war. Mit dieser Vorgehensweise stoßen wir aber spätestens dann an unsere Grenzen, wenn wir versuchen, einen Spieler-Charakter komplett mit Textur-Koordinaten und Normale zu erstellen. In diesem Kapitel werden wir unsere OpenGL-Wrapper-Klassen um einen 3D-Model-Loader erweitern, so dass wir diese laden und anzeigen können.

3D-Formate

Typischerweise entwickeln professionelle Entwicklungsstudios ihr eigenes 3D-Model-Format. Dies hat mehrere Gründe; Zum einen kann dadurch das Format an die jeweiligen Bedürfnisse angepasst werden, und zum anderen lässt sich so das fertig erstellte 3D-Model nicht mehr außerhalb des Spieles öffnen.

Dabei werden die Models zunächst mit Programmen wie z. B. Maya, 3d Max, Lightwave usw. erstellt und anschließend über ein extra entwickeltes Export-Modul so abgespeichert, dass sie später direkt von der Game-Engine gelesen werden können.



3D-Models werden von Modeling-Tools erstellt und exportiert.

Das Entwickeln solch eines Export-Moduls ist nicht ganz einfach, da man sich zunächst einmal mit dem SDK (wenn vorhanden) des jeweiligen 3D-Programms auseinandersetzen muss. Glücklicherweise gibt es aber schon jede Menge Formate, die sehr weit verbreitet und auch gut dokumentiert sind.

Das Wavefront-Format

In diesem Kapitel wird es darum gehen, einen 3D-Model-Loader zu schreiben, welcher Models laden kann, die im Wavefront .obj-Format gespeichert wurden.

Da dieses Format sehr einfach aufgebaut ist, eignet es sich hervorragend dazu, einen ersten Einstieg in dieses Thema zu bekommen. Ein weiterer Vorteil dieses Formats ist, dass so gut wie jedes 3D-Programm .obj-Dateien exportieren kann. Das Format basiert auf ganz normalem Text und lässt sich somit mit jedem beliebigen Text-Editor öffnen und bearbeiten. Dabei unterstützt es nicht nur polygonale Objekte, sondern auch NURBS (gekrümmte Flächen). Wir werden uns aber nur die polygonalen Flächen genauer anschauen, da NURBS in einem Spiel eher unüblich sind.

Ein .obj-Model besteht aus zwei Dateien, zum einen aus der .obj-Datei, welche die Rohdaten (Vertices, Textur-Koordinaten, Normale) für das Model beinhaltet, und einer .mtl-Datei, in der die Materialeigenschaften des Models gespeichert sind. Die Rohdaten werden dabei als indexierte Geometrie gespeichert, wie das funktioniert, haben wir ja im Kapitel über Vertex-Arrays bzw. VBOs bereits gesehen.

Schauen wir uns zunächst einmal das .obj-Format genauer an.

Das obj-Format intern

Typischerweise stehen folgende Zeilen in solch einer Datei:

Dies ist ein Kommentar

Zeilen, die mit dem #-Zeichen beginnen, sind reiner Text, der überlesen werden kann. Also nichts anderes als Kommentare.

v float float float

Beschreibt einen Eckpunkt (Vertex), der durch 3 float-Werte definiert ist.

vn float float float

Normale, durch 3 float-Werte definiert.

vt float float

Textur-Koordinate, durch 2 float-Werte definiert.

f int int int

Face (Index), welcher auf die Vertices verweist.

f int/int int/int int/int

Face (Index), welcher auf die Vertices und die Textur-Koordinaten verweist.

f int/int/int int/int/int int/int/int

Face (Index), welcher auf die Vertices, die Textur-Koordinaten und die Normale verweist.

Bei den Faces (*f*) ist zu beachten, dass, wenn z. B. keine Textur-Koordinaten definiert wurden, eine Zeile folgendermaßen aussieht:

```
f int//int int//int int//int
```

Eine Zeile, in der keine Normale, aber Text-Koordinaten exportiert wurden, sieht so aus:

```
f int/int/ int/int/ int/int/
```

Und eine Zeile, in der nur die Vertices exportiert wurden, sieht so aus:

```
f int// int// int//
```

Hier einmal beispielhaft eine .obj-Datei, die einen Würfel darstellt:

```
# Exported from Wings 3D 0.99.00b
mtllib cube2.mtl
o Mesh
#8 vertices, 12 faces
v -0.50000000 -0.50000000 -0.50000000
v 0.50000000 -0.50000000 -0.50000000
v 0.50000000 -0.50000000 0.50000000
v -0.50000000 -0.50000000 0.50000000
v -0.50000000 0.50000000 -0.50000000
v 0.50000000 0.50000000 -0.50000000
v 0.50000000 0.50000000 0.50000000
v -0.50000000 0.50000000 0.50000000
vt 0.0000000e+0 0.33333300
vt 0.0000000e+0 0.66666700
vt 0.25000000 0.0000000e+0
vt 0.25000000 0.33333300
vt 0.25000000 0.66666700
vt 0.25000000 1.00000000
vt 0.50000000 0.0000000e+0
vt 0.50000000 0.33333300
vt 0.50000000 0.66666700
vt 0.50000000 1.00000000
vt 0.75000000 0.33333300
vt 0.75000000 0.66666700
vt 1.00000000 0.33333300
vt 1.00000000 0.66666700
vn -0.57735027 -0.57735027 -0.57735027
vn 0.57735027 -0.57735027 -0.57735027
vn 0.57735027 -0.57735027 0.57735027
vn -0.57735027 -0.57735027 0.57735027
```

```

vn -0.57735027 0.57735027 -0.57735027
vn 0.57735027 0.57735027 -0.57735027
vn 0.57735027 0.57735027 0.57735027
vn -0.57735027 0.57735027 0.57735027
g Mesh
usemtl Default
f 1/3/1 3/8/3 4/4/4
f 1/13/1 6/12/6 2/11/2
f 1/1/1 8/5/8 5/2/5
f 2/7/2 3/8/3 1/3/1
f 2/11/2 6/12/6 3/8/3
f 3/8/3 6/12/6 7/9/7
f 3/8/3 8/5/8 4/4/4
f 4/4/4 8/5/8 1/1/1
f 5/14/5 6/12/6 1/13/1
f 5/6/5 8/5/8 6/10/6
f 6/10/6 8/5/8 7/9/7
f 7/9/7 8/5/8 3/8/3

```

Nochmals als Zusammenfassung: Die Zeilen, die mit einem (v, vt, vn) beginnen, sind die Rohdaten an sich. Die Zeilen, die mit einem (f) beginnen sind, die Indexe auf die Rohdaten.

AUFGEPASST

Achtung, die Indexe beginnen bei 1 und nicht, wie man es vielleicht gewohnt ist, bei 0, doch dazu später mehr.

Wie man sehen kann, sind die Daten alle zeilenweise abgespeichert, so dass es nicht sonderlich schwer fällt, diese einzulesen.

Wir werden unseren Model-Loader (CFXWavefrontMesh) in zwei Schritten entwickeln, im ersten Schritt geht es zunächst einmal darum, die Rohdaten einzulesen und zu rendern. Später, im zweiten Schritt, kommen dann noch das Material und die Textur hinzu.

Wavefront-Model laden

Beginnen wir mit den Variablen:

```

@interface CFXWavefrontMesh : NSObject {
    GLushort      *_indices;    // Array Indices
    CFXVector     *_vertices;   // Array Vertices

```

```

CFXVector      *_normals; // Array Normale
CFXTextureCoords *_texCoords; // Array Textur-Koordinaten
int            _numIndexes; // Anzahl Indices
int            _numVertices; // Anzahl Vertices
GLuint         _buffers[4]; // VBO Puffer
}

```

Wir deklarieren 4 Arrays, die später unsere Rohdaten beinhalten werden, weiterhin benötigen wir 4 Puffer für die VBOs.

Die beiden Variablen *_numIndexes* und *_numVertices* werden zum Rendern benötigt, da diese die Anzahl der Daten (Daten in den Arrays) beinhalten.

Daten einlesen

Das eigentliche Einlesen der .obj-Datei erfolgt in 4 Schritten:

1. Größe der Arrays berechnen, welche die Rohdaten speichern sollen
2. Rohdaten einlesen und zwischenspeichern
3. Polygone in Arrays speichern
4. VBOs aus den Arrays generieren

Diese Schritte werden wir uns jetzt einer nach dem anderen genau anschauen.

Schritt 1

Größe der Arrays berechnen:

```

-(BOOL)loadMesh:(NSString*)name
{
    NSError *error;

    // String fuer den Pfad des Models zusammenbauen
    NSString *path = [[NSBundle mainBundle] pathForResource:name
ofType:@"obj"];
    // Model einlesen
    NSString *content = [NSString stringWithContentsOfFile:path
encoding:NSUTF8StringEncoding error:&error];

    if(!content)
        return NO;

    // Schritt 1

```



```

// Groesse der Arrays berechnen in welchen die Rohdaten
// gespeichert werden sollen
int numIndicies = 0;
NSArray *lines = [content componentsSeparatedByString:@"\n"];
NSEnumerator *enumerator = [lines objectEnumerator];
NSScanner *scanner;
NSString *object;
while ((object = [enumerator nextObject]))
{
    scanner = [NSScanner scannerWithString:object];
    //Faces / Indexe
    if([object hasPrefix:@"f "])
    {
        numIndicies+=9;
    }
}
// Speicher allokieren
_indices = calloc(numIndicies, sizeof(GLushort));
_vertices = calloc(numIndicies, sizeof(CFXVector));
_normals = calloc(numIndicies, sizeof(CFXVector));
_texCoords = calloc(numIndicies, sizeof(CFXTextureCoords));

if(_indices==NULL || _vertices==NULL || _normals==NULL ||
_texCoords==NULL)
    return NO;

```

Zunächst wird ein String für den Dateinamen zusammengebaut, dabei wird davon ausgegangen, dass sich das Model im Main-Bundle des Programms befindet. Anschließend wird der Inhalt der Datei in einen String eingelesen. Danach wird mit einem Scanner jede Zeile eingelesen und geprüft, ob es sich um eine Zeile mit Indizes (f) handelt. Wenn dem so ist, wird die Variable numIndicies um 9 erhöht, da pro Zeile ja 3 Werte für Vertices, Textur-Koordinaten und Normale stehen.

AUFGEPASST

Achtung, achten Sie darauf, dass beim Exportieren Ihres Models aus einem 3D-Programm alle Daten (Vertices, Normale und UVs) in .obj-Datei stehen. Manche Programme exportieren z. B. keine Normale, was in unserem Model-Loader zu Problemen führen kann.

Nach dieser Prozedur wissen wir, wie groß die Arrays sein müssen, welche die Rohdaten zwischenspeichern sollen.

Man kann diesen Schritt auch komplett ausfallen lassen, indem man statt C-Arrays NSMutableArrays zum Speichern der Rohdaten verwendet. Ich habe einen Test mit dieser Vorgehensweise gemacht und kann sagen, dass die hier gezeigte Methode schneller funktioniert.

Schritt 2

Rohdaten einlesen und zwischenspeichern:

Genau wie in Schritt 1 lesen wir nun die Daten ein, mit dem Unterschied, dass wir nun nicht nur die Indexe lesen, sondern alle Rohdaten, entsprechend lang wird der Code dazu, also nicht erschrecken.

```
// Schritt 2
// Rohdaten auslesen und in temp-Arrays zwischenspeichern
float x,y,z;
int _verticesCounter = 0;
int _normalsCounter = 0;
int _texCoordsCounter = 0;
int _indiciesCounter = 0;

GLushort *tmpIndicies;
CFXVector *tmpVertices;
CFXVector *tmpNormals;
CFXTextureCoords *tmpTexCoords;

// Speicher allokieren
tmpIndicies = calloc(numIndicies, sizeof(GLushort));
tmpVertices = calloc(numIndicies, sizeof(CFXVector));
tmpNormals = calloc(numIndicies, sizeof(CFXVector));
tmpTexCoords = calloc(numIndicies, sizeof(CFXTextureCoords));

if(tmpIndicies==NULL || tmpVertices==NULL || tmpNormals==NULL ||
tmpTexCoords==NULL)
    return NO;

// Model-Datei zeilenweise einlesen und Daten extrahieren
NSArray *strings = [content componentsSeparatedByString:@"\n"];
NSEnumerator *rawData = [strings objectEnumerator];
while ((object = [rawData nextObject]))
{
    scanner = [NSScanner scannerWithString:object];
    //Vertex
    if([object hasPrefix:@"v "])
```

```
{
    [scanner scanString:@"v" intoString:nil];
    [scanner scanFloat:&x];
    [scanner scanFloat:&y];
    [scanner scanFloat:&z];
    tmpVertices[_verticesCounter].x = x;
    tmpVertices[_verticesCounter].y = y;
    tmpVertices[_verticesCounter].z = z;

    _verticesCounter++;
}
//Normale
else if([object hasPrefix:@"vn"])
{
    [scanner scanString:@"vn" intoString:nil];
    [scanner scanFloat:&x];
    [scanner scanFloat:&y];
    [scanner scanFloat:&z];
    tmpNormals[_normalsCounter].x = x;
    tmpNormals[_normalsCounter].y = y;
    tmpNormals[_normalsCounter].z = z;

    _normalsCounter++;
}
//TextureKoordinaten
else if([object hasPrefix:@"vt"])
{
    [scanner scanString:@"vt" intoString:nil];
    [scanner scanFloat:&x];
    [scanner scanFloat:&y];
    tmpTexCoords[_texCoordsCounter].u = x;
    tmpTexCoords[_texCoordsCounter].v = y;

    _texCoordsCounter++;
}
//Faces / Indexe
else if([object hasPrefix:@"f"])
{
    int v1;

    [scanner scanString:@"f" intoString:nil];
```

```

        // Indexe beginnen im Wavefront-Model bei 1
        // unsere Arrays aber bei 0, deshalb ziehen wir vom
        // ausgelesenen Index 1 ab
        // damit spaeter der Zugriff auf die Arrays stimmt
[scanner scanInt:&v1];[scanner scanString:@"/" intoString:nil];
tmpIndicies[_indiciesCounter] = v1-1; _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    [scanner scanInt:&v1];[scanner scanString:@"/"
intoString:nil];tmpIndicies[_indiciesCounter] = v1-1;
    _indiciesCounter++;
    }
}

```

Sie sehen, es wird mit einem Scanner die Datei zeilenweise eingelesen, und wenn eine entsprechende Zeile gefunden wurde (v, vt, vn, f), werden die Werte daraus extrahiert und in die Arrays gespeichert.

Bei den Indexen passiert genau dasselbe, mit dem kleinen Unterschied, dass die eingelesenen Werte um 1 verringert werden, da wie gesagt die Indexe bei 1 beginnen, unsere Arrays aber bei 0.

Diese Kleinigkeit führte schon oft zu großem Kopferbrechen, aber Sie wissen ja nun Bescheid.

Schritt 3

Polygone speichern:

Im letzten Schritt speichern wir nun die Polygone in den vier Arrays, welche wir in der Header-Datei für die VBOs angelegt haben. Dabei werden diese aber nicht blindlings eingefügt, viel mehr wird davor geprüft, ob sich in den Arrays schon Polygone befinden, die fast identische Werte besitzen. Wenn dem so ist, wird das Polygon selbst nicht noch einmal gespeichert, sondern einfach der Index, der auf dieses Polygon zeigt.

Zuerst aber die Zeilen, in denen die Polygone aus den temporär angelegten Arrays ausgelesen werden:

```
// Schritt 3
// Rohdaten in Arrays einfüegen
CFXVector v[3];
CFXVector n[3];
CFXTextureCoords t[3];
int triangles;

// Alle Polygone durchlaufen und in die beiden Arrays (_vertices,
//_indicies) einfüegen
for( triangles = 0; triangles < numIndicies; triangles+=9)
{
    v[0] = tmpVertices[tmpIndicies[triangles]];
    t[0] = tmpTexCoords[tmpIndicies[triangles+1]];
    n[0] = tmpNormals[tmpIndicies[triangles+2]];

    v[1] = tmpVertices[tmpIndicies[triangles+3]];
    t[1] = tmpTexCoords[tmpIndicies[triangles+4]];
    n[1] = tmpNormals[tmpIndicies[triangles+5]];

    v[2] = tmpVertices[tmpIndicies[triangles+6]];
    t[2] = tmpTexCoords[tmpIndicies[triangles+7]];
    n[2] = tmpNormals[tmpIndicies[triangles+8]];

    [self addTriangle:&v[0] normals:&n[0] texCoords:&t[0]];
}
```

Wir lesen je 3 Vertices (ein Dreieck) mit den dazugehörigen UVs (Textur-Koordinaten) und Normale aus und übergeben die Werte der Methode, welche die Prüfung auf Redundanz durchführen soll.

```
// Polygon in die Liste mit aufnehmen
-(void)addTriangle:(CFXVector*)vertices normals:(CFXVector*)
normals texCoords:(CFXTextureCoords*)texCoords
{
    // Annaehrung zwischen Polygon1 und Polygon2, wird dieser
    // Wert unterschritten gelten die beiden Polygone als gleich
    float delta = 0.000001;

    // Normale normalisieren, falls diese nicht normalisiert
    // exportiert wurden
    normals[0] = normalizeVector(normals[0]);
    normals[1] = normalizeVector(normals[1]);
    normals[2] = normalizeVector(normals[2]);

    // Prüfen, ob es ein bestehendes Polygon in der Liste gibt,
    // dass die Selben Werte besitzt wie das, welches
    // uebergeben wurde.
    int vertex;
    for(vertex = 0; vertex < 3; vertex++)
    {
        int tmp;
        for(tmp = 0; tmp < _numVertices; tmp++)
        {
            // Ist die Vertex-Position gleich ?
            if(isAlmostSame(_vertices[tmp].x, vertices[vertex].x,
delta) &&
                isAlmostSame(_vertices[tmp].y, vertices[vertex].y,
delta) &&
                isAlmostSame(_vertices[tmp].z, vertices[vertex].z,
delta) &&

                // Sind die Normale gleich ?
                isAlmostSame(_normals[tmp].x, normals[vertex].x,
delta) &&
                isAlmostSame(_normals[tmp].y, normals[vertex].y,
delta) &&
                isAlmostSame(_normals[tmp].z, normals[vertex].z,
delta) &&
```

```

        // Sind die Textur-Koordinaten gleich ?
        isAlmostSame(_texCoords[tmp].u, texCoords[vertex]
        .u, delta) &&
        isAlmostSame(_texCoords[tmp].v, texCoords[vertex]
        .v, delta))
    {
        // Wenn ja, nur den Index (also den verweis auf
        // das Polygon) in die Liste einfüegen
        _indices[_numIndexes] = tmp;
        _numIndexes++;
        break;
    }
}

// Polygon ist noch nicht in der Liste, also einfüegen
if(tmp == _numVertices)
{
    _vertices[_numVertices] = vertices[vertex];
    _normals[_numVertices] = normals[vertex];
    _texCoords[_numVertices] = texCoords[vertex];
    _indices[_numIndexes] = _numVertices;

    _numIndexes++;
    _numVertices++;
}
}
}

```

Der Wert *delta* ist unser Wert, der festlegt, ab wann 2 Polygone als identisch gelten. Als Nächstes wird die Normale normalisiert, das ist kein Muss, sondern eine reine Vorsichtsmaßnahme, falls das 3D-Programm das beim Export nicht getan hat.

Anschließend werden die Daten, welche der Methode übergeben wurden, durchlaufen, und der besagte Test wird mit allen schon gespeicherten Daten durchgeführt. Die Funktion *isAlmostSame(...)* ist in der Datei »CFXMath.h« definiert und sieht folgendermaßen aus:

```

// Prueft, ob der Abstand zwischen value und referenceValue
// kleiner als div ist
static inline bool isAlmostSame(float value, float referenceValue,
float div)
{

```

```
    return (fabs(value - referenceValue) < div);  
}
```

Nachdem alle Daten gespeichert wurden, können die VBOs erzeugt werden:

```
// VBO's erzeugen  
-(void)buildVBOs  
{  
    // Puffer anlegen  
    glGenBuffers(4, _buffers);  
  
    // Daten in Grafikkarten-Speicher kopieren  
    // Vertices  
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[0]);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*3,  
        _vertices, GL_STATIC_DRAW);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    // Normale  
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[1]);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*3,  
        _normals, GL_STATIC_DRAW);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    // Textur-Koordinaten  
    glBindBuffer(GL_ARRAY_BUFFER, _buffers[2]);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*_numVertices*2,  
        _texCoords, GL_STATIC_DRAW);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
  
    // Indexe  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _buffers[3]);  
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort)*_num-  
        Indexes, _indices, GL_STATIC_DRAW);  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);  
  
    // Da die Daten nun in der Grafikkarte liegen, koennen die  
    // Arrays geloescht werden  
    if(_indices)  
        free(_indices);  
  
    if(_vertices)
```



```

        free (_vertices);

    if(_normals)
        free(_normals);

    if(_texCoords)
        free(_texCoords);

    _indicies = NULL;
    _vertices = NULL;
    _normals = NULL;
    _texCoords = NULL;
}

```

Die Funktionsweise sollte nichts Neues mehr für Sie sein, da wir das ja schon im Kapitel über VBOs gesehen haben. Wir generieren für jedes Array einen Puffer und speichern diesen darin. Zum Schluss werden die Arrays wieder gelöscht, da unsere Daten nun im Grafikarten-Speicher liegen.

TIPP

Ein kleiner Hinweis: Wenn ein System keine VBOs unterstützt, dürfen Sie die Arrays natürlich nicht löschen, da der Rendervorgang dann über Vertex-Arrays gemacht wird, und diese liegen ja im Hauptspeicher.

Einige der Methoden habe ich als »privat« deklariert, das ist natürlich Geschmacksache und kann selbstverständlich wieder geändert werden.

Das Beispielprogramm »Kapitel 11/WavefrontMesh« zeigt die Funktionsweise unseres Loaders, es sind lediglich 2 Aufrufe nötig, um ein Model zu laden und zu rendern.

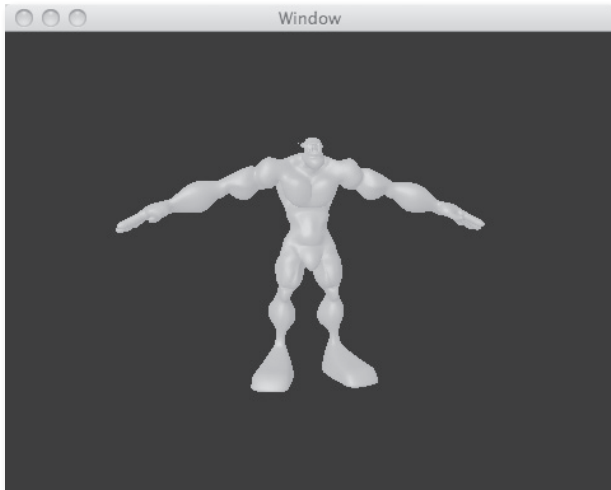
```

// Mesh laden
_mesh = [[CFXWavefrontMesh alloc] init];
[_mesh loadMesh:@"ogre"];
.
.
.
[_mesh renderMesh];

```

Achtung: Da das Testmodel »ogre.obj« aus über 25.000 Dreiecken besteht, kann der Ladevorgang einen Moment dauern, also nicht erschrecken. Wenn alles funktio-

niert hat, sollten Sie eine Ausgabe wie in der folgenden Abbildung auf dem Schirm haben.



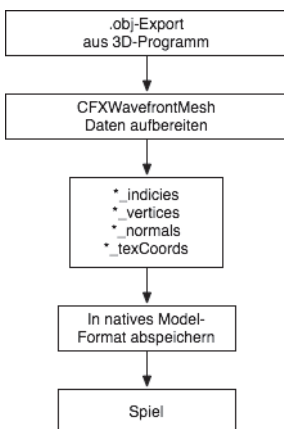
Unser fertig geladenes Wavefront-Model

Über die Leertaste können Sie noch zwischen den beiden Render-Modi `GL_LINE` und `GL_FILL` umschalten.

Nochmals ein paar Worte zum eigenen Model-Format:

Wie man an unserem Beispiel sehen kann, dauert das Laden großer Modelle mit vielen Polygonen recht lange (je nach System natürlich), was einfach daran liegt, dass die Daten zuerst aufbereitet werden müssen. Hier liegt es auf der Hand, ein eigenes natives Format zu erstellen, welches sich schneller laden lässt. Wenn wir

uns nochmals den Codeabschnitt anschauen, in welchem die VBOs erzeugt werden, fällt auf, dass unser Model im Prinzip aus 4 Arrays besteht. Es sollte also kein Problem sein, diese 4 Arrays zusammen in eine Datei binär abzuspeichern und später auch wieder auszulesen. Dadurch sollte der Ladevorgang um einiges schneller werden.



Der Weg vom .obj-Model zum eigenen nativen Format

TIPP

Im Kapitel 15, in welchem wir unser Spiel machen, zeige ich Ihnen, wie man das Format sehr einfach in einer plist abspeichern kann.

Material hinzufügen

Alles, was unserem Loader noch fehlt, ist das Einlesen der Material-Datei. Auch die Datei (*.mtl*) ist eine reine Text-Datei, die beispielsweise so aussehen kann:

```
# Exported from Wings 3D 0.98.32a
newmtl Default
Ns 2.0
d 1.00000
illum 2
Kd 1 1 1
Ka 1 1 1
Ks 0.1 0.1 0.1
Ke 0.00000e+0 0.00000e+0 0.00000e+0
map_Kd ogre.tif
```

Zeilen, die mit einem # beginnen, sind wieder Kommentare. Die für uns wichtigen Zeilen sind:

```
Kd 1 1 1
steht für den Diffuse-Anteil des Materials (3 float-Werte)
```

```
Ka 1 1 1
der Ambient-Anteil des Materials (3 float-Werte)
```

```
Ks 0.1 0.1 0.1
der Glanz-Anteil des Materials (3 float-Werte)
```

```
Ke 0.00000e+0 0.00000e+0 0.00000e+0
Emmissions-Anteil des Materials (3 float-Werte)
```

```
Ns 20.0
Shininess-Anteil des Materials (1 float-Wert)
```

```
map_Kd ogre.tif
die Texture (Diffuse). Diese muss im gleichen Verzeichnis wie die .mtl-Datei liegen.
```

Alle anderen Werte in der Datei sind für uns nicht weiter wichtig, weshalb wir sie außen vor lassen.

Unser nächster Loader »Kapitel 11/WavefrontMesh Material« wird nun so erweitert, dass er diese .mtl-Datei lesen und auch anzeigen kann.

Zunächst sind einige Variablen für das Material und die Textur hinzugekommen:

```
BOOL      _haveMaterial;    // Materialliste vorhanden
//Materialien
GLfloat   _matDiffuse[4];
GLfloat   _matAmbient[4];
GLfloat   _matSpecular[4];
GLfloat   _matEmission[4];
GLfloat   _matShininess;
GLuint    _texture;
```

Das Einlesen der .mtl-Datei erfolgt in der Methode

```
-(BOOL)loadMesh:(NSString*)name
```

Nachdem die VBOs erzeugt wurden.

```
.
.
.
// VBO's aus den beiden Arrays erzeugen
[self buildVBOs];

// Material laden
[self buildMaterials:name];
```

Es spielt dabei keine Rolle, ob das Material vor oder nach dem Erzeugen der VBOs gelesen wird.

Als Nächstes folgt die eigentliche Methode, die nun die .mtl-Datei liest.

```
// Material Laden
-(void)buildMaterials:(NSString*)name
{
    // String fuer das Material zusammenbauen
    NSError *error;
```

```
NSString *path = [[NSBundle mainBundle] pathForResource:name
ofType:@"mtl"];
NSString *content = [NSString stringWithContentsOfFile:path
encoding:NSUTF8StringEncoding error:&error];
if(!content)
{
    return;
}

NSArray *lines = [content componentsSeparatedByString:@"\n"];
NSEnumerator *enumerator = [lines objectEnumerator];
NSScanner *scanner;
NSString *object;

while ((object = [enumerator nextObject]))
{
    scanner = [NSScanner scannerWithString:object];
    float x,y,z;
    //Material Diffuse
    if([object hasPrefix:@"Kd "])
    {
        [scanner scanString:@"Kd" intoString:nil];
        [scanner scanFloat:&x];
        [scanner scanFloat:&y];
        [scanner scanFloat:&z];
        _matDiffuse[0] = x;
        _matDiffuse[1] = y;
        _matDiffuse[2] = z;
        _matDiffuse[3] = 1.0;
    }
    //Material Ambient
    if([object hasPrefix:@"Ka "])
    {
        [scanner scanString:@"Ka" intoString:nil];
        [scanner scanFloat:&x];
        [scanner scanFloat:&y];
        [scanner scanFloat:&z];
        _matAmbient[0] = x;
        _matAmbient[1] = y;
        _matAmbient[2] = z;
        _matAmbient[3] = 1.0;
    }
}
```

```
//Material Specular
if([object hasPrefix:@"Ks «"])
{
    [scanner scanString:@"Ks" intoString:nil];
    [scanner scanFloat:&x];
    [scanner scanFloat:&y];
    [scanner scanFloat:&z];
    _matSpecular[0] = x;
    _matSpecular[1] = y;
    _matSpecular[2] = z;
    _matSpecular[3] = 1.0;
}
//Material Emission
if([object hasPrefix:@"Ke"])
{
    [scanner scanString:@"Ke" intoString:nil];
    [scanner scanFloat:&x];
    [scanner scanFloat:&y];
    [scanner scanFloat:&z];
    _matEmission[0] = x;
    _matEmission[1] = y;
    _matEmission[2] = z;
    _matEmission[3] = 1.0;
}
//Material Shininess
if([object hasPrefix:@"Ns"])
{
    [scanner scanString:@"Ns" intoString:nil];
    [scanner scanFloat:&x];
    _matShininess = x;
}
//Texture Diffuse
if([object hasPrefix:@"map_Kd"])
{
    NSString *fileName;
    [scanner scanString:@"map_Kd" intoString:nil];
    [scanner scanUpToString:@"\r" intoString:&fileName];

    NSArray *a = [fileName pathComponents];
    // 3D-Model Programme exportieren den Pfad zu einer
    // Texture sehr unterschiedlich
```

```

        // manchmal steht nur der Name der Texture drin,
        // manchmal aber der komplette Pfad zur Texture
        // wir benoetigen lediglich den Namen, also muessen
        // wir diesen extrahieren
        // dieser steht an letzter Stelle im Array
        NSString *tmp = [a lastObject];
        path = [[NSBundle mainBundle] pathForResource:[tmp
        stringByDeletingLastPathComponent] ofType:[tmp
        lastPathComponent]];
        if(path)
            _texture = [[CFXTextureManager sharedManager]
            textureByName:path];
    }

}
_haveMaterial = YES;
}

```

Die Vorgehensweise ist exakt dieselbe wie beim Parsen der Rohdaten im .obj-File, einzig das Auslesen des Textur-Pfades ist ein wenig umständlich, was daran liegt, dass einige Programme nur den Namen der Textur exportieren, andere aber den kompletten Pfad. Für uns ist nur der eigentliche Name der Textur wichtig, weshalb wir nur ihn aus dem Array extrahieren.

Nachdem das Material nun gelesen wurde, muss es beim Rendern noch aktiviert werden:

```

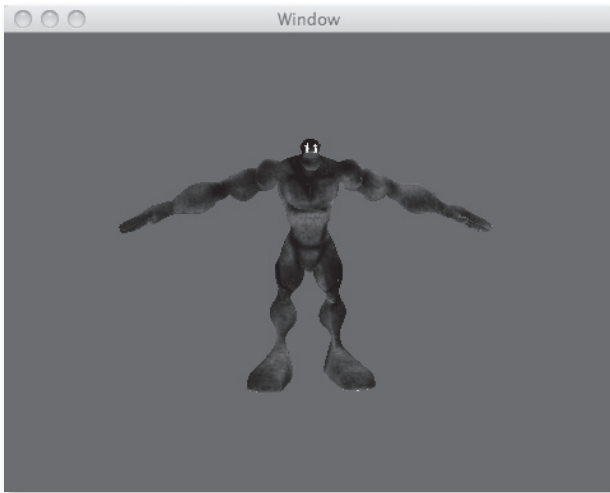
- (void)renderMesh
{
    if(_haveMaterial)
    {
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, _matDiffuse);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, _matAmbient);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
        _matSpecular);
        glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,
        _matEmission);
        glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS,
        _matShininess);

        glBindTexture(GL_TEXTURE_2D, _texture);
    }
}

```

```
glEnableClientState(GL_VERTEX_ARRAY);  
glEnableClientState(GL_NORMAL_ARRAY);  
glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
.  
.  
.
```

Es werden die Parameter in die Materialeigenschaften eingesetzt, welche aus der .mtl-Datei gelesen wurden, und schon wird das Model mit Textur und Materialien gerendert.



Unser fertig texturiertes Model mit Materialien

Das war schon alles. Wie Sie sehen, ist es nicht sonderlich schwer gewesen, dieses Format zu lesen.

Wir werden im Kapitel 14 nochmals auf den Loader zurückkommen, wenn es darum geht, Kollisionstests zu machen.

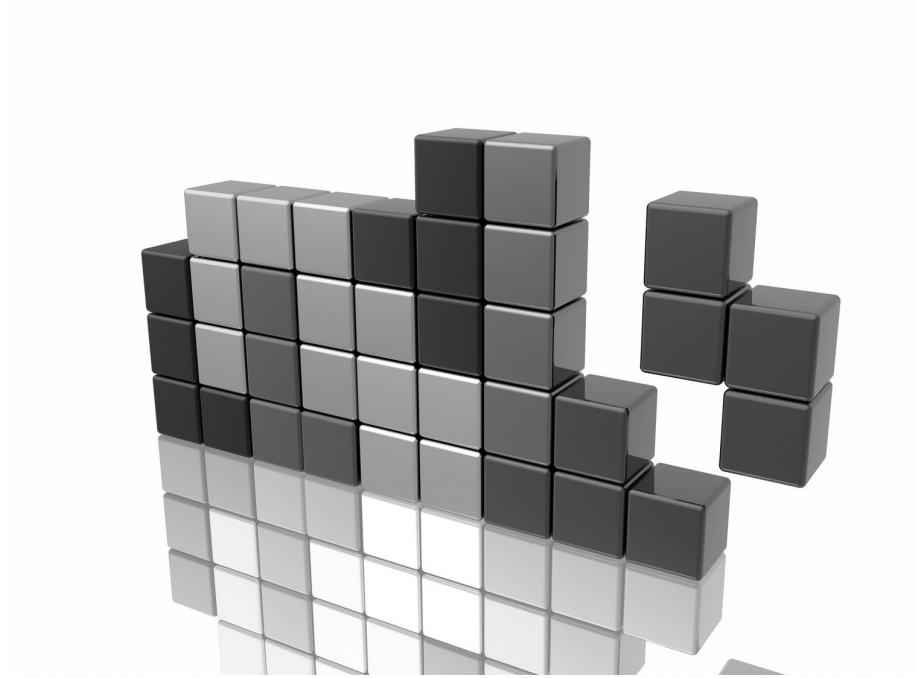
Zusätzliche Informationen

<http://www.martinreddy.net/gfx/3d/OBJ.spec>

Eine komplette Beschreibung zu dem Wavefront-Format

<http://jeux.developpez.com/sources/opengl/?page=modeles>

Eine Sammlung verschiedener Model-Loader (unter anderem auch dem Doom3-Format). Die Seite ist zwar auf Französisch, aber das ist ja nicht weiter schlimm.



Shader

Da es sich bei diesem Buch eigentlich um ein Einsteiger-Buch handelt, war ich mir zuerst nicht ganz sicher, ob ich das Thema Shader besprechen soll, da dieses doch alleine schon ein ganzes Buch füllt.

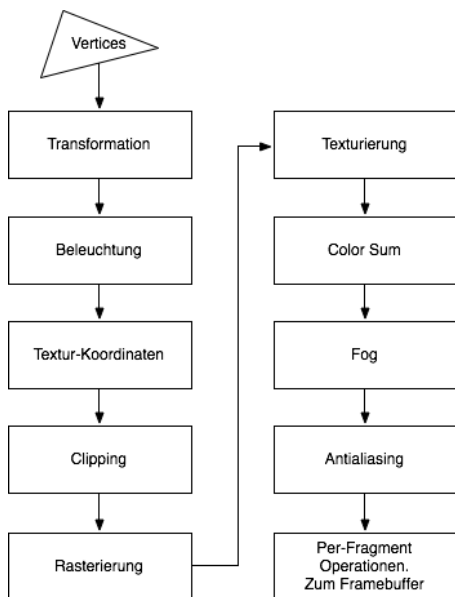
Weil es aber heutzutage so gut wie kein Spiel mehr gibt, das nicht auf Shader setzt, habe ich mich doch entschlossen, hier eine kleine Einführung zu geben. Ich will aber gleich zu Anfang darauf hinweisen, dass es keine komplette Referenz zu diesem Thema sein wird, vielmehr möchte ich Ihnen die Grundzüge der Shader-Programmierung zeigen.

Nun, wie gesagt, kann man über die Shader-Entwicklung alleine schon ein Buch füllen, was einfach daran liegt, dass es ein recht komplexes Thema ist, welches fundierte Kenntnisse in OpenGL und in der linearen Algebra erfordert. Bevor Sie nun aber zum nächsten Kapitel weiterblättern, möchte ich Sie beruhigen, wir werden auch hier wieder mit ganz einfachen Beispielen beginnen.

Ich bin mir sicher, dass Sie staunen werden, wenn Sie sehen, was man mit dieser Technik alles anstellen kann.

Was sind Shader

Zunächst einmal stellt sich die Frage, was diese Shader eigentlich sind bzw. wozu sie man sie gebrauchen kann. Auch hier muss man wieder ein wenig ausholen: Alle Vertices, die innerhalb eines OpenGL-Programmes ausgegeben werden sollen, durchlaufen die sogenannte Fixed-Pipeline, die folgende Abbildung verdeutlichen soll.

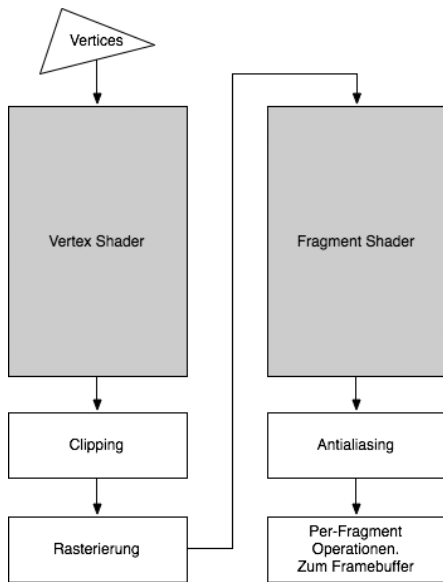


Die Fixed-Pipeline von OpenGL

Wie man sehen kann, durchlaufen die Vertices mehrere Stationen, bis sie irgendwann auf dem Bildschirm landen. Auf diese einzelnen Stationen können wir mehr oder weniger Einfluss nehmen, indem wir Änderungen an der State-Machine vornehmen. Der Einfluss ist aber auch hier nur minimal, da OpenGL feste Vorgaben hat, die z. B. regeln, wie die Beleuchtung ausgegeben wird.

Mit Shadern ist es nun möglich, die festen Vorgaben der Render-Pipeline teilweise zu umgehen, indem man ein Programm schreibt, welches direkt auf der Grafikkarte ausgeführt wird.

Dazu nochmals eine Abbildung, die zeigt, welche Teile der Pipeline davon betroffen sind.



Die programmierbaren Teile der Render-Pipeline

Wie man auf der Abbildung sehen kann, handelt es sich um zwei verschiedene Bereiche (Programme), die jeweils für ganz bestimmte Aufgaben verantwortlich sind.

- Vertex-Shader

Ersetzt die Transformation, Beleuchtung und die Generierung der Textur-Koordinaten, per Vertex!

- Fragment-Shader oder Pixel-Shader

Ersetzt die Texturierung, Color-Sum (Seconadry Color) und den Nebel, per Pixel!

GRUNDLAGEN

Nach der Rasterierung werden aus den Vertices sogenannte Fragmente.

Shader sind also Programme, welche direkt auf der Grafikkarte ausgeführt werden. Sie werden für OpenGL mit Hilfe der Sprache GLSLang (oder kurz GLSL) entwickelt. Das Schöne an GLSL ist, dass diese Sprache sehr viel Ähnlichkeit mit C hat, wodurch die Einarbeitung relativ kurz ist (wenn man C kann, aber wem erzähle ich dass).

Warum Shader benutzen

- Mit Shadern hat man einen größeren Einfluss auf das Renderergebnis.
- Sie können Daten vertexweise oder pixelweise sehr schnell ändern.
- Einmal erstellt, kann man sie in anderen Projekten nutzen.
- Viele Effekte sind nur mit Hilfe von Shadern realisierbar.

Alle diese Punkte werden wir uns im weiteren Verlauf genauer anschauen.

Voraussetzungen

Shader gehören seit der OpenGL-Version 2.0 zu den Kernfunktionen, davor (ab OpenGL-Version 1.4) wurde der Zugriff mit Hilfe von Extensions geregelt.

Um herauszufinden, ob ein System Shader unterstützt, muss man einfach nur die OpenGL-Version abfragen, was so aussehen könnte:

```
float version = atof(glGetString(GL_VERSION));
```

Sollte *version* kleiner als 2.0 sein, kann man mit Hilfe des folgenden Codes testen, ob zumindest die Extensions unterstützt werden:

```
_shadersAvailable = YES;
const GLubyte* extensions = glGetString(GL_EXTENSIONS);
if ((GL_FALSE == gluCheckExtension((GLubyte *) "GL_ARB_shader
_objects", extensions)) ||
(GL_FALSE == gluCheckExtension((GLubyte *) "GL_ARB_shading
_language_100", extensions)) ||
(GL_FALSE == gluCheckExtension((GLubyte *) "GL_ARB_vertex_shader",
extensions)) ||
(GL_FALSE == gluCheckExtension((GLubyte *) "GL_ARB_fragment
_shader", extensions)))
{
    _shadersAvailable = NO;
}
```

Die Arbeitsweise zwischen den Kernfunktionen und Extensions ist gleich, lediglich die Schreibweise ist ein wenig anders.

Als Beispiel rufen wir einmal die Funktion auf, welche einen Shader kompiliert.

Zuerst über die Kernfunktion:

```
void glCompileShader(GLuint shader);
```

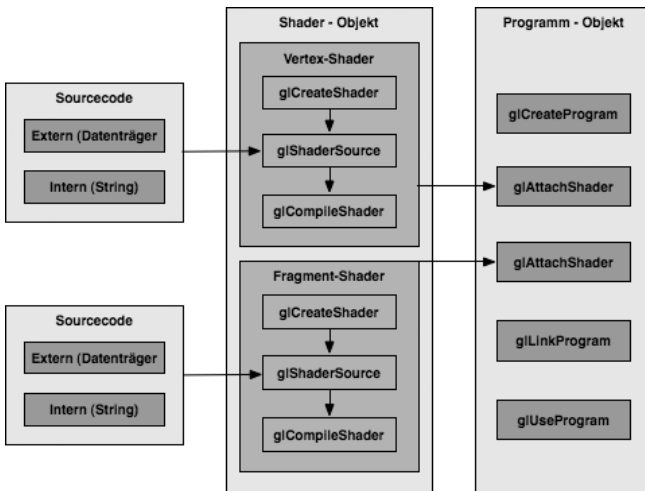
Und nun die Extension-Version:

```
void glCompileShaderARB(GLhandleARB shader);
```

Wie Sie sehen können, unterscheiden sich die beiden Funktionsaufrufe nur geringfügig. Um keine Verwirrung zu stiften, werden wir in diesem Kapitel nur die Kernfunktionen verwenden.

Handhabung der Shader

Shader werden in GLSL über zwei Objekte gemanagt. Diese sind zum einen die Shader-Objekte, welche den eigentlichen Programm-Code enthalten und kompiliert werden müssen, und zum anderen die Programm-Objekte, die sich um den Ablauf der Shader-Objekte kümmern.



Handhabung der Shader unter GLSL

Schauen wir uns zuerst die Shader-Objekte an. Wie man auf der Abbildung sehen kann, werden Shader-Objekte in 3 Schritten erzeugt.

Shader-Objekte

Shader-Objekte werden über die Funktion `glCreateShader(...)` erzeugt, diese Funktion erwartet als einzigen Parameter den Typ des Shaders, welchen wir erzeugen möchten, weshalb hier nur `GL_VERTEX_SHADER` bzw. `GL_FRAGMENT_SHADER` in Frage kommen kann.

```
GLuint _vertexShader = glCreateShader(GL_VERTEX_SHADER);  
GLuint _fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

Wenn das Objekt erfolgreich angelegt werden konnte, gibt die Funktion einen Wert größer null zurück. Nicht mehr benötigte Objekte können mit der Funktion

```
glDeleteShader(_vertexShader), bzw. glDeleteShader(_fragmentShader);
```

zum Löschen markiert werden. Das bedeutet, dass OpenGL entscheidet, wann die Objekte gelöscht werden, und sie nicht, wie z. B. bei den Texturen, sofort aus dem Speicher entfernt werden.

Shader-Sourcecode verwalten

GLSL erwartet den eigentlichen Sourcecode als C-String, dieser kann entweder direkt in das Programm eingebettet sein oder aber (meine Empfehlung) aus einer externen Datei geladen werden.

```
glShaderSource(_vertexShader, 1, vsStringPtr, NULL);
```

Den Sourcecode in eine externe Datei auszulagern, ist deshalb sinnvoll, da man ihn dann ohne Probleme für andere Projekte nutzen kann.

Shader kompilieren

Der letzte Schritt besteht nun darin, den Sourcecode zu kompilieren.

```
glCompileShader(_vertexShader);
```

Nachdem der Code kompiliert wurde, können wir prüfen, ob irgendwelche Fehler aufgetaucht sind. Dazu bietet OpenGL die Funktion `glGetShaderInfoLog(...)` an, diese erwartet als ersten Parameter das Shader-Objekt und als weitere Parameter verschiedene Einstellungen zum Puffer (`GLchar`), in dem der Status der Kompilierung gespeichert werden soll. Wir werden uns das später noch genauer anschauen.

```
GLint success;  
glGetShaderiv(myVertexShader, GL_COMPILE_STATUS, &success);  
if (!success)
```

```

{
    GLchar infoLog[MAX_INFO_LOG_SIZE];
    glGetShaderInfoLog(myVertexShader, MAX_INFO_LOG_SIZE, NULL,
        infoLog);
    NSLog(@"Fehler beim Compilieren von: %@\n%@", pathToVertex-
        Shader, [NSString stringWithCString:infoLog]);
    return NO;
}

```

TIPP

Die Funktion *glGetShaderInfoLog(...)* beinhaltet nicht nur Fehlermeldungen, sondern auch andere nützliche Informationen (Warnungen, Diagnose-Nachrichten) über ein Shader-Objekt.

Programm-Objekte

Das zweite Objekt ist wie gesagt das Programm-Objekt, dieses dient als Container für Shader-Objekte und macht aus diesen ein ausführbares Programm. Zunächst wird über die Funktion *glCreateProgram()* ein leeres Programm-Objekt erzeugt, welches später ein Shader-Objekt beinhalten soll. Das Löschen geschieht analog zu den Shader-Objekten.

```

GLuint _program = glCreateProgram();
.
.
glDeleteProgram(_program);

```

Nachdem nun der »Container« erzeugt wurde, wird ihm ein Shader-Objekt übergeben.

```

glAttachShader(_program, _vertexShader);
glAttachShader(_program, _fragmentShader);

```

Es ist dabei möglich, nur Teile der Fixed-Pipeline zu ersetzen, indem man einfach den betreffenden Teil nicht bindet.

Wollten wir beispielsweise, dass die Transformation durch die Fixed-Pipeline vorgenommen wird, dann würden wir nur den entsprechenden Shader binden:

```

glAttachShader(_program, _fragmentShader);

```


Man kann auch zu Laufzeit zwischen der Fixed-Pipeline und GLSL umschalten, indem man die Funktion

```
glDetachShader(_program, _fragmentShader);
```

aufruft. Dadurch würde z. B. der Part des Fragment-Shaders wieder durch die Fixed-Pipeline übernommen. Um wieder komplett zur Fixed-Pipeline zurückzukehren, müsste man einfach beide Shader lösen:

```
glDetachShader(_program, _fragmentShader);  
glDetachShader(_program, _vertexShader);
```

Programm linken

Bevor nun über GLSL gerendert werden kann, muss das Programm noch gelinkt werden. Bei diesem Vorgang wird aus den zuvor kompilierten Shadern ein ausführbares Programm

```
glLinkProgram(_program);
```

Auch hier ist es ratsam, den Status des Linkvorgangs abzufragen, dieser ist im Prinzip derselbe wie beim Kompilieren.

```
GLint success;  
glGetProgramiv(_program, GL_LINK_STATUS, &success);  
if (!success)  
{  
    GLchar infoLog[MAX_INFO_LOG_SIZE];  
    glGetProgramInfoLog(_program, MAX_INFO_LOG_SIZE, NULL,  
        infoLog);  
    NSLog(@"Fehler beim Linken: %@", [NSString  
        stringWithCString:infoLog]);  
    return NO;  
}
```

Programm überprüfen

Auch wenn das Programm fertig kompiliert und gelinkt wurde, kann es durchaus vorkommen, dass während der Laufzeit Fehler auftreten (wie beim Programmieren üblich).

Nun ist es aber nicht möglich, das Programm einfach in Xcode zu debuggen, hier muss also eine Möglichkeit her, die uns Informationen über auftretende Fehler ausgibt.

Genau diese gibt es (Sie werden es schon vermutet haben), und sie funktioniert wieder genau so, wie das Abfragen des Status beim Linken bzw. Kompilieren:

```
GLint success;
glValidateProgram(_program, GL_VALIDATE_STATUS, &success);
if (!success)
{
    GLchar infoLog[MAX_INFO_LOG_SIZE];
    glGetProgramInfoLog(_program, MAX_INFO_LOG_SIZE, NULL,
        infoLog);
    NSLog(@"Fehler beim ausfuehren: %@", [NSString
        stringWithCString:infoLog]);
}
```

Dieser Vorgang ist kein Muss, spart aber eine Menge Ärger beim Auffinden von Fehlern. Wenn die Shader wie gewünscht funktionieren, kann man diese Abfrage ruhig abschalten, wodurch man ein wenig an Performance gewinnt.

Programm nutzen

Der allerletzte Schritt besteht nun darin, das Programm zu nutzen.

```
glUseProgram(_program);
```

Um GLSL wieder zu deaktivieren, übergibt man dieser Funktion einfach eine Null.

```
glUseProgram( 0 );
```

Dadurch kehrt man wieder vollständig zur Fixed-Pipeline zurück. Abschließend nochmals eine kurze Zusammenfassung der einzelnen Schritte:

Zusammenfassung

1. Shader-Objekt erzeugen: *glCreateShader(...)*;
2. Source-Code definieren
3. Shader kompilieren: *glCompileShader(...)*;
4. Programm-Objekt erzeugen: *glCreateProgram(...)*;
5. Shader-Objekt binden: *glAttachShader(...)*;
6. Programm-Objekt linken: *glLinkProgram(...)*;
7. Programm-Objekt nutzen: *glUseProgram(...)*;

Shader-Hilfsklassen

Das war zunächst einmal alles, was nötig ist, um Shader zu nutzen. Damit man diese ganze Prozedur nicht jedes Mal neu eingeben muss, habe ich 2 Klassen erstellt, welche die Arbeit mit dem Shader um einiges vereinfachen. Die Arbeitsweise der beiden Klassen erinnert sehr stark an unsere Textur-Klassen und lässt sich ebenso einfach handhaben.

Im folgenden Abschnitt schauen wir uns die beiden einmal genauer an.

CFXShaderManager

Zunächst einmal ist diese Klasse wieder als Singleton angelegt, ihre Aufgabe besteht darin, Shader-Objekte (CFXShaderObject) zu verwalten bzw. Shader zu laden und für die Nutzung vorzubereiten (kompilieren, linken usw.). Ich habe nur die wichtigsten Funktionen eingebaut, damit es übersichtlich bleibt.

Bei der Initialisierung wird geprüft, ob GLSL überhaupt unterstützt wird, wenn ja, wird ein leeres Array erzeugt, in welches später die Shader-Objekte gespeichert werden sollen.

Ich habe hier auch gleich noch den Code vorbereitet, der nötig ist, um GLSL über Extensions abzufragen:

```
-(id)init
{
    self = [super init];
    if(self)
    {
        _currentShaderObject = nil;
        _shadersAvailable = YES;
        // Die OpenGL-Version abfragen
        // wenn >= 2 dann ist GLSL als Kern verfuegbar
        float version = atof((char*)glGetString(GL_VERSION));
        // Wenn man GLSL mit Hilfe von Extensions nutzen muss
        // (OpenGL-Version < 2) dann kann man mit folgenden
        // Zeilen abfragen ob diese Extensions unterstuetzt
        // werden wenn dem nicht so ist, werden Shader nicht
        // unterstuetzt
        /*const GLubyte* extensions = glGetString(GL_EXTENSIONS);
        if ((GL_FALSE == gluCheckExtension((GLubyte *)"GL_ARB
        _shader_objects", extensions)) ||
```

```

        (GL_FALSE == gluCheckExtension((GLubyte *)"GL_ARB
        _shading_language_100", extensions)) ||
        (GL_FALSE == gluCheckExtension((GLubyte *)"GL_ARB
        _vertex_shader", extensions)) ||
        (GL_FALSE == gluCheckExtension((GLubyte *)"GL_ARB
        _fragment_shader", extensions)))*/

    // Um es einfach zu halten wird nur die Kern-Version von
    // GLSL verwendet
    if(version < 2.0)
    {
        _shadersAvailable = NO;
    }
    else
    {
        _shadersAvailable = YES;
        _loadedShaders = [[NSMutableArray alloc] init];
        return self;
    }
}
return nil;
}

```

Wie gesagt, unterstützt unsere Klasse GLSL nur über die Kernfunktion, es sollte aber kein großes Problem darstellen, sie so zu erweitern, dass auch Extensions unterstützt werden.

Die Hauptarbeit liegt in der Methode:

```

- (BOOL)createShader:(NSString*)shaderName
    vertexShader:(NSString*)pathToVertexShader
    fragmentShader:(NSString*)pathToFragmentShader

```

Diese macht für uns all die Schritte, die wir oben gesehen haben, also Anlegen, Laden, Kompilieren usw., so dass wir uns um nichts mehr kümmern müssen.

```

- (BOOL)createShader:(NSString*)shaderName
    vertexShader:(NSString*)pathToVertexShader
    fragmentShader:(NSString*)pathToFragmentShader
{
    if(!_shadersAvailable)

```

```
        return NO;

NSError *error;

NSString *vertexShaderContent = [NSString stringWithContentsOfFile:pathToVertexShader encoding:NSUTF8StringEncoding error:&error];
if (vertexShaderContent == nil)
{
    NSLog(@"Fehler beim Lesen von: %@\n%@", pathToVertexShader, [error localizedFailureReason]);
    return NO;
}

NSString *fragmentShaderContent = [NSString stringWithContentsOfFile:pathToFragmentShader encoding:NSUTF8StringEncoding error:&error];
if (fragmentShaderContent == nil)
{
    NSLog(@"Fehler beim Lesen von: %@\n%@", pathToVertexShader, [error localizedFailureReason]);
    return NO;
}

GLint success;

//Vertex Shader
//Shader Objekte erzeugen
const GLchar *vsStringPtr[1];
GLuint myVertexShader = glCreateShader(GL_VERTEX_SHADER);

vsStringPtr[0]=[vertexShaderContent UTF8String];
glShaderSource(myVertexShader, 1, vsStringPtr, NULL);

//Shader compilieren
glCompileShader(myVertexShader);
glGetShaderiv(myVertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    GLchar infoLog[MAX_INFO_LOG_SIZE];
    glGetShaderInfoLog(myVertexShader, MAX_INFO_LOG_SIZE, NULL, infoLog);
}
```

```
        NSLog(@"Fehler beim Compilieren von: %@\n%@", pathTo
        VertexShader, [NSString stringWithCString:infoLog]);
        return NO;
    }

    //Fragment Shader
    //Shader Objekte erzeugen
    const GLchar *fsStringPtr[1];
    GLuint myFragmentShader =glCreateShader(GL_FRAGMENT_SHADER);
    fsStringPtr[0]=[fragmentShaderContent UTF8String];
    glShaderSource(myFragmentShader, 1, fsStringPtr, NULL);

    //Shader compilieren
    glCompileShader(myFragmentShader);
    glGetShaderiv(myFragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        GLchar infoLog[MAX_INFO_LOG_SIZE];
        glGetShaderInfoLog(myFragmentShader, MAX_INFO_LOG_SIZE,
        NULL, infoLog);
        NSLog(@"Fehler beim Compilieren von: %@\n%@", pathTo
        FragmentShader, [NSString stringWithCString:infoLog]);
        return NO;
    }

    //Shader Programm erzeugen
    program = glCreateProgram();

    //Shader dransbinden
    glAttachShader(program, myVertexShader);
    glAttachShader(program, myFragmentShader);

    //Shader linken
    glLinkProgram(program);
    glGetProgramiv(program, GL_LINK_STATUS, &success);
    if (!success)
    {
        NSLog(@"Fehler beim Linken des Shaders");
    }

    //Shader Objekt erzeugen und in Array einhängen
```

```

    CFXShaderObject *shaderObject = [[CFXShaderObject alloc]
    init];
    [shaderObject setShaderName:shaderName];
    [shaderObject setVertexShader:myVertexShader];
    [shaderObject setFragmentShader:myFragmentShader];
    [shaderObject setProgram:program];

    [_loadedShaders addObject:shaderObject];
    [shaderObject release];

    return YES;
}

```

Die Schrittfolge ist so, wie wir sie oben gesehen haben. Einzig die Shader-Nutzung (*glUseProgram(...)*) ist noch nicht aktiviert worden, so kann man zu Beginn eines Programms mehrere Shader laden, ohne sie gleich zu aktivieren.

Um nun einen Shader zu nutzen, verwenden wir folgende Methode, diese sucht so lange im Array, bis sie das entsprechende Shader-Objekt gefunden hat, und aktiviert es dann.

```

- (void)useShader:(NSString*)shaderName
{
    NSEnumerator *enumerator = [_loadedShaders objectEnumerator];
    CFXShaderObject *element;
    while(element = [enumerator nextObject])
    {
        if([[element shaderName]isEqualToString:shaderName])
            glUseProgram([element program]);
    }
}

```

Um GLSL zu deaktivieren und wieder zur Fixed-Pipeline zurückzukehren, genügt ein Aufruf von:

```

- (void)disableShader
{
    glUseProgram(0);
}

```

Den Rest der Methoden werden wir später noch kennenlernen.

CFXShaderObject

Wie gesagt, stellt diese Klasse ein einfaches Shader-Objekt dar und wird über den Shader-Manager verwaltet.

Sie ist recht einfach aufgebaut und enthält jeweils einen Verweis auf einen Vertex-Shader, einen auf einen Fragment-Shader und zusätzlich das Shader-Programm.

Der Name *_shaderName* dient dazu, das Objekt später über seinen Namen ansprechen zu können.

Der Rest der Klasse besteht aus Setter und Getter-Methoden.

```
@interface CFXShaderObject : NSObject {

    GLuint    _vertexShader; // Vertex Shader Objekt
    GLuint    _fragmentShader; // Fragment Shader Objekt
    GLuint    _program;      // Shader Objekt
    NSString*  _shaderName;  // Shader Name
}

/**
 * Getter und Setter
 */
- (GLuint)vertexShader;
- (void)setVertexShader:(GLuint)value;

- (GLuint)fragmentShader;
- (void)setFragmentShader:(GLuint)value;

- (GLuint)program;
- (void)setProgram:(GLuint)value;

- (NSString *)shaderName;
- (void)setShaderName:(NSString *)value;
```


Ein erster Versuch

Mit dem bisher gelernten Wissen werden wir uns an unser erstes Beispiel wagen. Das Beispielprogramm »Kapitel 12/Shader 01« macht zunächst einmal nicht mehr, als ein rotes Quadrat auf den Bildschirm zu zeichnen.

Wichtig ist zunächst einmal, dass Sie sehen, wie die Klasse `CFXShaderManager` funktioniert. Zu Anfang holen wir uns eine Instanz des Managers.

```
// Shader erzeugen
_shaderManager = [CFXShaderManager sharedManager];
```

Wenn wir wissen möchten, ob GLSL überhaupt verfügbar ist, rufen wir folgende Methode auf:

```
// Werden Shader unterstützt ?
NSLog(@"Shader unterstützt?: %d", [_shaderManager
shadersAvailable]);
```

Wenn Shader unterstützt werden, können wir sie laden, wie gesagt ist es am besten, diese in eine externe Datei auszulagern, damit man sie später in anderen Projekten wieder verwenden kann.

```
[_shaderManager createShader:@"Simple 01"
    vertexShader:[NSBundle mainBundle] pathForResource:
@"Simple01" ofType: @"vert"]
    fragmentShader:[NSBundle mainBundle] pathForResource:
@"Simple01" ofType: @"frag"]];
```

Wir übergeben der Methode einen Namen (frei verfügbar) und die beiden Pfade zu den Shadern.

Wenn alles geklappt hat, gibt sie ein *YES* zurück, was bedeutet, dass die Shader korrekt geladen, kompiliert und gelinkt wurden.

TIPP

Der Name und das Suffix für die Shader können Sie beliebig vergeben, man könnte also auch schreiben: *Simple01.vertexShader*.

Um nun die beiden Shader zu nutzen, genügt folgender Aufruf:

```
[_shaderManager useShader:@"Simple 01"];
```

Wir übergeben der Methode den Namen des Shader-Objektes, welches wir nutzen wollen, möchte man auf ein anderes Shader-Objekt umschalten, muss man einfach den anderen Namen angeben:

```
[_shaderManager useShader:@"Simple 25"];
```

Selbstverständlich muss dieses Objekt zunächst wieder wie oben beschrieben geladen werden.

Das zunächst einmal alles, was man zu den beiden Klassen wissen muss:

Die beiden Shader

Kommen wir nun zu den eigentlichen Shader-Codes.

»Simple01.vert«

```
void main()
{
    gl_Position = ftransform();
}
```

Auf den ersten Blick sieht der Code wie ein gewöhnliches C-Programm aus, was daran liegt, dass GLSL sehr an die Syntax von C angelehnt ist, dass heißt, es gibt Funktionen, Variablen, Strukturen usw.

Wir werden uns die wichtigsten Dinge alle im weiteren Verlauf noch anschauen.

Die Funktion *main()* ist der Einstiegspunkt für unseren Shader, sollte diese fehlen, quittiert uns GLSL das mit einem Fehler.

Wirklich interessant ist der Code, der in der *main()*-Funktion steht:

```
gl_Position = ftransform();
```

Dadurch wird die Vertex-Position (*glVertex3f(...)*) von den sogenannten Objekt-Koordinaten zu den Clip-Koordinaten transformiert. Man kann auch einfach sagen, das Objekt wird im 3D-Raum platziert (Multiplikation mit der Modelviewmatrix und der Projektionsmatrix).

In vielen Online-Tutorials zu GLSL sieht man auch folgende Zeile:

```
gl_Position = GL_ModelViewProjektionMatrix * gl_vertex;
```

Der Unterschied zu der oben gezeigten Version ist, dass *ftransform()* die eingehenden Eckpunkte exakt so transformiert, wie es die Fixed-Pipeline tut, wobei es bei der zweiten Version zu minimalen Abweichungen kommen kann.

Um keine Verwirrung zu stiften, verwenden wir im weiteren Verlauf *ftransform()*.

GRUNDLAGEN

ftransform() ist eine sogenannte build-in Funktion von GLSL, von denen es natürlich noch sehr viel mehr gibt.

Mehr steht nicht in unserem Vertex-Shader.

Sehr wichtig ist dabei zu wissen, dass es das absolute Minimum im Vertex-Shader ist, die Vertices entsprechend zu transformieren, da wir sonst überhaupt nichts auf dem Bildschirm sehen.

Das ist die Kehrseite von Shadern. Wenn wir sie benutzen wollen, müssen wir uns auch um die Dinge kümmern, die normalerweise die Fixed-Pipeline für uns übernimmt.

Kommen wir nun zum Fragment-Shader.

»Simple01.vert«

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

Hier gelten die gleichen Regeln wie für den Vertex-Shader, das heißt, wir müssen mindestens eine *main()*-Funktion haben.

gl_FragColor ist wie *gl_Position* eine vordefinierte Variable von GLSL, wobei *gl_FragColor* für die Farbe der ausgehenden Fragmente zuständig ist.

Wir übergeben hier einen Vektor, der eine rote Farbe definiert, wodurch wir unser besagtes rotes Quadrat auf dem Bildschirm zu sehen bekommen.

Auch hier gilt: Der Fragment-Shader muss mindestens eine Farbe definieren (*gl_FragColor(...)*), damit man überhaupt etwas auf dem Bildschirm sehen kann.

Bevor wir weiter machen, fassen wir die wichtigsten Dinge zusammen, die wir bei der Shader-Entwicklung beachten müssen:

- Prüfen, ob Shader verfügbar sind
- Beide Shader (Vertex und Fragment-Shader) werden beim Programmstart aus einer externen Datei geladen.
- Diese müssen vor der Nutzung aktiviert werden.
- Beide Shader müssen eine *main()*-Funktion haben.
- Der Vertex-Shader muss die einkommenden Vertices transformieren: *transform();*
- Der Fragment-Shader muss die einkommenden Fragmente einfärben: *gl_FragColor(...);*
- Um zu Fixed-Pipeline zurückzukehren, nutzen wir die Methode *disableShader*, welche im CFXShader-Manager definiert ist.

GLSL-Grundlagen

Nachdem wir die Einbindung der Shader in unser Programm kennengelernt haben, werden wir uns nun mit den grundlegenden Sprachelementen von GLSL beschäftigen.

Beginnen wir mit den Datentypen. Die meisten davon sind Ihnen bereits bekannt, wobei GLSL einige Datentypen unterstützt, die speziell auf den 3D-Bereich zugeschnitten wurden.

Datentyp	Erklärung
void	Funktion ohne Rückgabewert
bool	true / false
int	Integer
float	Float
vec2	2-Komponenten Float-Vektor
vec3	3-Komponenten Float-Vektor
vec4	4-Komponenten Float-Vektor
bvec2	2-Komponenten Bool-Vektor
bvec3	3-Komponenten Bool-Vektor
bvec4	4-Komponenten Bool-Vektor
ivec2	2-Komponenten Int-Vektor
ivec3	3-Komponenten Int-Vektor
ivec4	4-Komponenten Int-Vektor
mat2	2x2 Float-Matrix
mat3	3x3 Float-Matrix
mat4	4x4 Float-Matrix
sampler1D	Zugriff auf eine 1D-Textur
sampler2D	Zugriff auf eine 2D-Textur
sampler3D	Zugriff auf eine 3D-Textur
samplerCube	Zugriff auf eine Cubemap
sampler1DShadow	Zugriff auf eine 1D-Tiefentextur mit Vergleichsoperation
sampler2DShadow	Zugriff auf eine 2D-Tiefentextur mit Vergleichsoperation

Vektoren

Wie man an der Tabelle erkennen kann, sind Vektoren und Matrizen schon vorhanden, so dass man sich diese nicht selbst zusammenbauen muss. Die Handhabung ist ähnlich wie bei CFXVector.

Hier einige Beispiele:

```
vec3 vector;
vector.x;
vector.y;
vector.z;
```

Auch der Zugriff wie bei einem Array ist erlaubt:

```
float z = vector[2];
```

Folgende Werte (Namesets) können beim Zugriff auf Vektoren verwendet werden:

x, y, z, w	um z. B. Positionen zu speichern
r, g, b, a	um Farben zu speichern
s, t, p, q	um Textur-Koordinaten zu speichern

Die Verwendung dieser Namesets bleibt dem Entwickler selbst überlassen, allerdings ist es nicht erlaubt, verschiedene Namesets miteinander zu vermischen.

```
v4.rgb = vec4(1.0, 1.0, 0.0, 0.0); // Ok
v2.xx = vec2(1.7, 4.5);           // Nicht ok, da zweimal die gleiche
Komponente vorhanden ist
v4.rgzw = vec4(0.0, 1.0, 1.0, 1.0); // Nicht ok, da unterschiedliche
Namesets
v2.rgb = vec3(0.0, 1.0, 0.0);      // Nicht ok, da vec2 nur r und g be-
sitzt, aber kein b
```

Matrizen

Auch Matrizen sind in GLSL schon vorhanden, diese gibt es aber nur in der float-Variante:

```
mat2 matrix           // Eine 2x2 float-Matrix
mat3 matrix:          // Eine 3x3 float-Matrix
mat4 matrix           // Eine 4x4 float-Matrix
```

In der Regel werden diese Matrizen dazu verwendet, um die schon bekannten Matrizen (Modelview, Projektion usw.) zu speichern, allerdings steht es jedem frei, auch andere Dinge damit zu tun.

Der Zugriff auf die Elemente innerhalb der Matrix ist nicht weiter schwierig:

```
mat4 matrix;  
matrix[1] = vec4(0.0);
```

setzt die komplette zweite Reihe der Matrize auf Null.

```
mat3 matrix;  
matrix[0][0] = 1.25;
```

setzt das Element oben links auf 1.25

Sie sehen: Der Zugriff ist ähnlich wie bei normalen Arrays, wobei das erste Element die Spalte und das zweite Element die Reihe betrifft.

Es ist ohne Probleme möglich, Operationen zwischen einem Vektor und einer Matrize durchzuführen, ohne diese irgendwie vorher zu konvertieren.

Hier ein Beispiel:

```
vec3 result;  
vec3 color;  
mat3 matrix;  
result = color * matrix;
```

Typenqualifizierer

In GLSL gibt es sogenannte Typenqualifizierer, welche die Aufgabe haben, Informationen zwischen den Shadern und dem OpenGL-Quellcode auszutauschen. Nur über diese Typenqualifizierer ist es möglich, zur Laufzeit Daten an die Shader zu schicken bzw. diese zu manipulieren.

Die wichtigsten sind:

attribute

Dies sind Werte, die vom Shader (Vertex-Shader) nur gelesen werden können! Sie werden z. B. dazu genutzt, um zusätzliche Vertex-Informationen an den Shader zu übergeben.

uniform

Werden benutzt, um Daten von der Anwendung an beide Shader zu übergeben.

varying

Stellen eine Verbindung zwischen Vertex und Fragment-Shader her. Sie werden im Vertex-Shader geschrieben und dann interpoliert an den Fragment-Shader weitergegeben, wobei dieser dann damit weiterarbeiten kann.

Diese müssen in beiden Shadern mit demselben Namen deklariert werden!

const

Festgelegte Konstante, die nur lesbar ist.

Weiterhin gibt es noch: *default*, *in*, *out*, *inout*, welche uns aber im Moment nicht weiter interessieren.

Folgende Regeln sind bei der Verwendung der Typenqualifizierer zu beachten:

- *attribute*, *varying*, *uniform* müssen global deklariert werden.
- *varying* ist nur nutzbar mit den Datentypen *float*, *vec2*, *vec3*, *vec4*, *mat2*, *mat3* und *mat4*. Alle anderen Datentypen bzw. Strukturen können nicht verwendet werden.
- Typenqualifizierer müssen vor den Datentyp geschrieben werden, weiterhin haben sie keinen Standardtyp.

Auch hierzu wieder einige Beispiele.

Beispiel 1

Es soll ein Vektor (*myVector*) vom Vertex-Shader an den Fragment-Shader übergeben werden.

Vertex-Shader:

```
varying vec3 myVector;  
main()  
{  
    myVector = vec3(1.0, 0.0, 0.0);  
}
```

Fragment-Shader:

```
varying vec3 myVector;  
main()  
{  
    vec3 tmp = myVector;  
}
```


Beispiel 2

Es soll eine int-Variable (*mode*) vom OpenGL-Programm an den Vertex-Shader übergeben werden.

OpenGL-Programm:

```
[_shaderManager sendUniform1Int:@"mode" parameter:0];
```

Vertex-Shader:

```
uniform int mode;
main()
{
    int tmp = mode;
}
```

Beispiel 3

Es soll eine konstante rote Farbe (*redColor*) im Fragment-Shader definiert werden.

Fragment-Shader:

```
const vec4 redColor = vec4(1.0, 0.0, 0.0, 1.0);
main()
{
    gl_FrontColor = redColor;
}
```

Built-In Variablen

GLSL bringt auch eine Menge build-in-Variablen mit, wobei man schon am Namen erkennen kann, wofür sie stehen.

Vertex-Shader Build-In Variablen

Die folgenden Variablen stehen nur im Vertex-Shader zur Verfügung.

Spezielle Ausgabe Variablen (Zugriff Read / Write)

```
vec4  gl_Position;      // muss vom Shader geschrieben werden
float gl_PointSize;     // kann vom Shader geschrieben werden
vec4  gl_ClipVertex;    // kann vom Shader geschrieben werden
```

Die Variable *gl_PointSize* liefert die aktuelle Punktgröße. *gl_ClipVertex* wird für das Clipping verwendet.

Eingehende Attribute (Zugriff Read only)

```
attribute vec4 gl_Vertex;  
attribute vec4 gl_Normal;  
attribute vec4 gl_Color;  
attribute vec4 gl_SecondaryColor;  
attribute vec4 gl_MultiTexCoord0;  
attribute vec4 gl_MultiTexCoord1;  
attribute vec4 gl_MultiTexCoord2;  
attribute vec4 gl_MultiTexCoord3;  
attribute vec4 gl_MultiTexCoord4;  
attribute vec4 gl_MultiTexCoord5;  
attribute vec4 gl_MultiTexCoord6;  
attribute vec4 gl_MultiTexCoord7;  
attribute float gl_FogCoords;
```

Die einzelnen Attribute stehend stellvertretend für die Funktionen, welche im OpenGL-Code benutzt wurden. Zum Beispiel haben wir mit dem Attribut *gl_MultiTexCoord0* Zugriff auf die Textur-Koordinaten der ersten Textur-Einheit.

Varying Ausgabe (Zugriff Read / Write)

```
varying vec4 gl_FrontColor;  
varying vec4 gl_BackColor;  
varying vec4 gl_FrontSecondaryColor;  
varying vec4 gl_BackSecondaryColor;  
varying vec4 gl_TexCoord[];  
varying float gl_FogFragCoord;
```

Die beiden *gl_FrontColor* und *gl_BackColor* betreffen die Farben für die Vorder- und Rückseite der Polygone, Gleiches gilt für die «Secondary-Colors».

Die Textur- und Fog-Koordinaten stehen in *gl_TexCoord[]* bzw. *gl_FogFragCoord*.

Fragment-Shader Build-In-Variablen

Folgende Variablen sind nur im Fragment-Shader nutzbar:

Ausgabe Variablen (Zugriff Read / Write)

```
vec4 gl_FragColor;  
vec4 gl_FragData[];  
float gl_FragDepth;
```

Die erste Variable *gl_FragColor* kennen wir ja bereits, sie enthält die Farbe der Fragmente. Mit *gl_FragData[]* hat man die Möglichkeit, verschiedene Daten (Farben) in unterschiedliche Puffer zu schreiben. Über die Variable *gl_FragDepth* hat man die Möglichkeit, die Tiefenwerte, die durch die Fixed-Pipeline berechnet wurden, zu überschreiben.

Varying Eingabe (Zugriff Read only)

```
varying vec4 gl_Color;  
varying vec4 gl_SecondaryColor;  
varying vec4 gl_TexCoord[];  
varying vec4 gl_FogFragCoord;
```

Dies sind die Werte, welche aus dem Vertex-Shader kommen (Vertex-Shader Varying Ausgabe).

Spezielle Eingabe-Variablen (Zugriff Read only)

```
vec4 gl_FragCoord;    Pixel-Koordinaten  
bool gl_FrontFacing;
```

Die Variable *gl_FragCoord* beinhaltet die aktuelle Position des Fragments relativ zur Fensterposition im Format x,y,z,1/w, wobei z der Tiefenwert ist, welcher durch die Fixed-Pipeline berechnet wurde. Die letzte Variable *gl_FrontFacing* gibt an, ob das Fragment die Vorder- oder die Rückseite ist.

Build-in Uniform-Variablen (Zugriff Read only im Vertex- bzw. Fragment-Shader)

Die eingebauten Uniform-Variablen erleichtern den Zugriff auf einzelne Teile der OpenGL-Eigenschaften. Zum Beispiel erhalten wir durch einen Aufruf von

```
vec4 ambient = gl_FrontMaterial.ambient;
```

die Werte, die wir in unserem Programm mittels

```
float ambientMaterial[] = {0.1, 0.1, 0.1, 1.0};  
glMaterialfv(GL_FRONT, GL_AMBIENT, ambientMaterial);
```

gesetzt haben.

Wenn die Werte der einzelnen Variablen nicht im Programm geändert bzw. gesetzt wurden, beinhalten sie die Standardwerte, welche durch OpenGL vergeben werden.

Der Übersicht halber habe ich die Liste ein wenig gekürzt.

Standard-Matrizen

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
uniform mat3 gl_NormalMatrix;      (Inverse Modelview-Matrix,
transponiert)
```

Nebel

```
struct gl_FogParameters {
    vec4 color;
    float density;
    float start;
    float end;
    float scale;
};
uniform gl_FogParameters gl_Fog;
```

Licht

```
struct gl_LightSourceParameters
{
    vec4  ambient;           // Acli
    vec4  diffuse;           // Dcli
    vec4  specular;          // Scli
    vec4  position;           // Ppli
    vec4  halfVector;         // Berechnet: Hi
    vec3  spotDirection;      // Sdli
    float spotExponent;        // Srli
    float spotCutoff;          // Crli
    float spotCosCutoff;       // Berechnet: cos(Crli)
    float constantAttenuation; // K0
    float linearAttenuation;   // K1
    float quadraticAttenuation; // K2
};
uniform gl_LightSourceParameters  gl_LightSource[gl_MaxLights];
```

```

struct gl_LightModelParameters
{
    vec4 ambient;          // Acs
};
uniform gl_LightModelParameters gl_LightModel;

struct gl_LightModelProducts
{
    vec4 sceneColor;       // Berechnet. Ecm + Acm * Acs
};
uniform gl_LightModelProducts gl_FrontLightModelProduct;
uniform gl_LightModelProducts gl_BackLightModelProduct;

struct gl_LightProducts
{
    vec4 ambient;          // Acm * Acli
    vec4 diffuse;          // Dcm * Dcli
    vec4 specular;         // Scm * Scli
};
uniform gl_LightProducts gl_FrontLightProduct[gl_MaxLights];
uniform gl_LightProducts gl_BackLightProduct[gl_MaxLights];

```

Material

```

struct gl_MaterialParameters
{
    vec4 emission;         // Ecm
    vec4 ambient;          // Acm
    vec4 diffuse;          // Dcm
    vec4 specular;         // Scm
    float shininess;       // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;

```

Wie Sie sehen, werden die meisten in Strukturen zusammengefasst. Der Zugriff auf die einzelnen Elemente ist so, wie man es aus C her auch kennt. Zusätzlich zu den Standard-Variablen gibt es einige Variablen, die von OpenGL selbst nicht zur Verfügung gestellt werden, ein Beispiel dafür wäre der Vektor

```
gl_LightSource[0].halfVector
```

den wir später noch kennenlernen werden.

Vorab schon einmal einige Beispiele für diese build-in Uniform-Variablen.

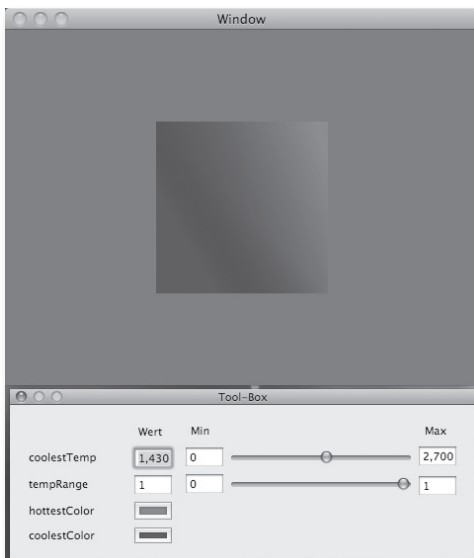
```
vec4 color = gl_FrontMaterial.specular;  
vec4 ambient = gl_FrontMaterial.ambient * gl_LightSource[0].  
ambient;  
vec4 position = gl_ModelViewMatrix * gl_Vertex;  
vec3 tangent = gl_MultiTexCoord1.xyz;
```

Weiterhin bietet GLSL noch eine Fülle von build-in-Funktionen, die man beim Entwickeln von 3D-Anwendungen benötigt. Wir werden einige dieser Funktionen im weiteren Verlauf noch kennenlernen.

Mit diesen ganzen neuen Informationen werden wir uns nun an die Arbeit machen und einmal schauen, was man mit Shadern so alles anstellen kann. Am Ende des Kapitels finden Sie einen Hinweis zur Kurzreferenz der oben angesprochenen Variablen. Diese ist sehr praktisch, da man sie ausgedruckt immer zur Hand hat. Es versteht sich von selbst, dass niemand die GLSL-Referenz auswendig kann.

Beispiel 2, der Farbverlauf

Unser zweites Beispiel »Kapitel 12/Shader 02« rendert ein Quadrat mit einem Farbverlauf, welcher sich zur Laufzeit über das Toolbox-Fenster verändern lässt.



Shader, der einen Farbverlauf erzeugt

Die Idee zum Quelltext der beiden Shader stammt übrigens aus dem Orange-Book (dazu später mehr).

Beginnen wir wieder mit dem Vertex-Shader:

```
uniform float   coolestTemp;
uniform float   tempRange;
attribute float  vertexTemp;
varying float    temperature;
void main()
{
    gl_Position = ftransform();
    temperature = (vertexTemp - coolestTemp) / tempRange;
}
```

Sie sehen, hier ist eine Menge Neues dazugekommen. Außerhalb der *main()*-Funktion wurden nun einige Variablen deklariert:

```
uniform float   coolestTemp;
uniform float   tempRange;
attribute float  vertexTemp;
varying float    temperature;
```

Die beiden *uniform*-Variablen *coolestTemp* und *tempRange* sind Variablen, die vom OpenGL-Programm aus an den Shader übergeben werden und sich auf das komplette Objekt (unser Quadrat) beziehen.

Die Variable *vertexTemp* ist ein Attribut, welches wir zusätzlich an unsere Vertices binden, sie bezieht sich somit auf die einzelnen Eckpunkte. Die Variable *varying temperature* ist eine Variable, die in beiden Shadern deklariert sein muss, und dient, wie wir nun wissen, dem Datenaustausch zwischen den beiden.

Der Rest des Shaders macht nichts anderes, als unsere Vertices zu transformieren (*transform*) und die Variable *temperature* für jeden Eckpunkt separat zu berechnen.

```
void main()
{
    gl_Position = ftransform();
    temperature = (vertexTemp - coolestTemp) / tempRange;
}
```

Schauen wir uns nun den Fragment-Shader an.

```
uniform vec3  coolestColor;
uniform vec3  hottestColor;
varying float temperature;
void main()
{
    vec3 color = mix(coolestColor, hottestColor, temperature);
    gl_FragColor = vec4(color, 1.0);
}
```

Auch hier wieder zunächst zwei Uniform-Variablen, welche vom OpenGL-Source-code aus gesteuert werden. Wie man sieht, ist hier auch wieder die *varying temperature* deklariert, und zwar deshalb, weil wir mit dieser Variable im Fragment-Shader weitere Berechnungen durchführen wollen.

Wir berechnen die endgültige Fragment-Farbe mit der build-in-Funktion *mix()*, welche einen linearen Verlauf zwischen den 3 Werten berechnet.

Zum Schluss wird die Farbe über *gl_FragColor* ausgegeben.

Kommen wir nun zum eigentlichen Programm, welches auch einige Neuerungen beinhaltet. Wie die Shader erzeugt und geladen werden, das überspringen wir, da dies immer der gleiche Vorgang ist. Nachdem wir die Shader aktiviert haben, übergeben wir die Variablen *hottestColor*, *coolestColor*, *coolestTemp* und *tempRange*.

```
// Farben setzten
[_shaderManager sendUniform3Float:@"hottestColor"
                        parameterX:1.0
                        parameterY:0.0
                        parameterZ:0.0];

[_shaderManager sendUniform3Float:@"coolestColor"
                        parameterX:0.0
                        parameterY:0.0
                        parameterZ:1.0];
```

Die beiden Farben werden dabei direkt aus dem *NSColorWell* ausgelesen und über die Methode *sendUniform3Float* an den Shader übergeben.

Diese Methode sieht folgendermaßen aus:

```
- (void)sendUniform3Float:(NSString*)name parameterX:(float)x
parameterY:(float)y parameterZ:(float)z
{
    int loc = -1;
    loc = glGetUniformLocation(program, [name cStringUsing-
    Encoding:NSUTF8StringEncoding]);
    if(x!=-1)
        glUniform3f(loc, x,y,z);
}
```

Wir übergeben der Methode den Namen der *uniform*-Variablen (z. B. *coolestColor*) und die Werte, welche diese *uniform*-Variable haben soll.

Über die Funktion *glGetUniformLocation(...)* wird zuerst der Ablageort der Variable gesucht, und wenn dieser gefunden wurde, wird die Variable durch die drei Werte modifiziert (*glUniform3f(...)*).

Dasselbe machen wir auch mit den beiden anderen Variablen.

```
// Werte setzten
[_shaderManager sendUniform1Float:@"coolestTemp" parameter:
[_coolestTempValue floatValue]];
[_shaderManager sendUniform1Float:@"tempRange" parameter:
[_tempRangeValue floatValue]];
```

Der Unterschied ist hier, dass wir nicht mehr drei Float-Werte übergeben, sondern nur noch einen, entsprechend anders sieht auch die Methode im *CFXShaderManager* aus:

```
- (void)sendUniform1Float:(NSString*)name parameter:(float)x
{
    int loc = -1;
    loc = glGetUniformLocation(program, [name cStringUsing-
    Encoding:NSUTF8StringEncoding]);
    if(x!=-1)
        glUniform1f(loc, x);
}
```

Geändert hat sich nur die Funktion *glUniform1f(...)*, der Rest ist gleich geblieben. Diese Methode gibt es im *CFXShaderManager* in drei verschiedenen Versio-

nen, welche sich nur vom Datentyp bzw. der Anzahl der zu übergebenden Werte unterscheiden.

```
// Schickt eine int-Variable an GLSL
- (void)sendUniform1Int:(NSString*)name parameter:(int)x;

// Schickt eine float-Variable an GLSL
- (void)sendUniform1Float:(NSString*)name parameter:(float)x;

// Schickt 3 float-Variablen an GLSL
- (void)sendUniform3Float:(NSString*)name parameterX:(float)x
parameterY:(float)y parameterZ:(float)z;
```

Wir haben die Variablen, welche in den Shadern als *uniform* deklariert sind, mit Werten gefüllt. Was noch fehlt, ist die *attribute*-Variable *vertexTemp*. Diese ist wie gesagt ein zusätzliches Attribut für unsere Vertices so wie z. B. die Farbe, die Textur-Koordinaten usw. Über dieses Attribut haben wir die Möglichkeit, zusätzliche Informationen an unsere Vertices zu heften. Bevor wir das aber tun können, müssen wir erst wieder den Ablageort der *attribute*-Variable herausfinden, was wir mit dem Aufruf von

```
// Location fuer vertexTemp holen
_location = [_shaderManager attributeLocation:@"vertexTemp"];
```

machen. Wir übergeben auch hier den Namen der Variable, und der *CFXShaderManager* gibt uns den Ablageort zurück, welchen wir bei der Definition unseres Quadrats einsetzen müssen:

```
glBegin(GL_QUADS);
    glVertexAttrib1f(_location, 1.0);    glVertex3f(-1.0f, -1.0f,
    0.0f);
    glVertexAttrib1f(_location, 2.2);    glVertex3f( 1.0f, -1.0f,
    0.0f);
    glVertexAttrib1f(_location, 2.5);    glVertex3f( 1.0f,  1.0f,
    0.0f);
    glVertexAttrib1f(_location, 1.7);    glVertex3f(-1.0f,  1.0f,
    0.0f);
glEnd();
```

Über die Funktion *glVertexAttrib1f(...)* können wir wie gesagt zusätzliche Informationen an unsere Vertices binden. Diese erwartet als ersten Parameter den Ablageort und als zweiten den Wert, welchen das neue Attribut haben soll.

GRUNDLAGEN

Die *glVertexAttrib1f(...)*-Funktion gibt es wieder in verschiedenen Versionen, so wie wir es von anderen Funktionen auch gewohnt sind.

Die Werte, die übergeben wurden, sind willkürlich von mir gewählt und können nach Belieben geändert werden.

Noch ein Wort zum Rendervorgang des Beispiels: Das zusätzliche Attribut ist nicht nur im Immediate-Mode verfügbar, sondern auch bei den Vertex-Arrays, der Code dazu könnte z. B. so aussehen:

```
float vertices[8] = {-1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0};  
float attribs[4] = {2.0, 2.0, -2.0, -2.0};
```

```
...
```

```
_location = [_shaderManager attributeLocation:@"myAttributes"];
```

```
glEnableClientState(GL_VERTEX_ARRAY);  
glEnableVertexAttribArray(_location);
```

```
glVertexPointer(2, GL_FLOAT, 0, vertices);  
glVertexAttribPointer(_location, 1, GL_FLOAT, 0, 0, attribs);
```

Bursting Mesh

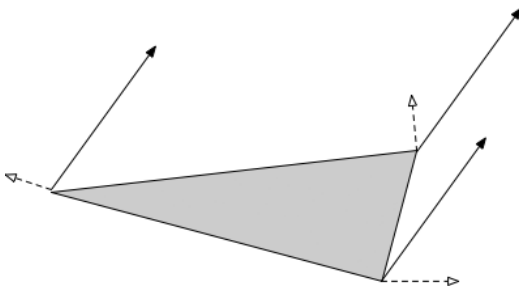
Das nächste Beispiel »Kapitel 12/Bursting Mesh« ist ein klassischer Fall dafür, dass man mit Shadern Dinge tun kann, die ohne sie nur schwer bzw. gar nicht möglich wären. Stellen wir uns dazu Folgendes vor: Wir haben ein 3D-Objekt, welches so »zerplatzen« soll, dass die einzelnen Polygone im Raum umherfliegen.

Der erste Lösungsansatz wäre vielleicht, das Objekt im Immediate-Mode zu rendern, da dabei der Zugriff auf die einzelnen Vertices sehr einfach möglich ist. Dieser Ansatz fällt aber unter den Tisch, da unser Objekt als VBO gerendert werden soll.

Im Kapitel über Vertex-Buffer-Objekte haben wir ja gesehen, dass man mit *glMapBuffer(...)* Zugriff auf die Daten im VBO hat, was eine Lösung wäre, Dadurch müsste man pro Frame ein Mal die Vertices vom Grafikkarten-Speicher abholen, modifizieren und wieder hochladen, was auch funktionieren würde. Sie merken schon, worauf ich hinaus will: Mit einem einfachen Vertex-Shader und gerade einmal 2 Zeilen Quellcode lässt sich das ohne Probleme bewerkstelligen.

Beginnen wir dieses Mal mit dem OpenGL-Code und dem Laden des 3D-Models.

Das Beispiel ist so aufgebaut, dass die Polygone in die Richtung fliegen, in welche ihre Normale zeigen. Wenn wir unser Model mit Hilfe der Klasse CFXWavefront-Mesh laden würden, hätten wir ein Problem mit den Normalen, diese werden nämlich dort pro Vertex extra berechnet, das bedeutet, wir haben für jeden Eckpunkt einen Normal-Vektor, der je nachdem, wie der Eckpunkt im Raum liegt, in eine andere Richtung zeigt, die folgende Abbildung verdeutlicht das anhand der gestrichelten Linien.



Berechnung der Vertex-Normalen in der Klasse CFXWavefrontMesh

Wenn wir nun diese Normale für unser Beispiel nehmen würden, würde das bedeuten, dass jeder Eckpunkt in die Richtung fliegt, in welche sein Normal-Vektor zeigt, was aber nicht das ist, was wir wollen. Wir möchten ja, dass das komplette Dreieck in eine Richtung fliegt.

Damit nun nicht die Klasse CFXWavefrontMesh geändert werden muss, habe ich den Code für unser Model »von Hand« erstellt und in der Datei »model.model« abgespeichert.

Bei diesem zeigen nun alle Eckpunkte eines Polygons in dieselbe Richtung, auf der Abbildung sieht man das anhand der durchgezogenen Linien.

Soweit nun die Vorarbeit. Aus den Daten der Datei »model.model« werden die VBOs erzeugt, welche nur aus Vertices und Normale bestehen. Der Code dazu ist nichts Neues mehr, weshalb wir ihn jetzt überspringen. Das »Zerplatzen« des Models erfolgt, nachdem auf die Leertaste gedrückt wurde, der komplette Code dazu sieht so aus:

```
[_shaderManager useShader:@"Bursting Mesh"];
if(_go)
{
    _speed+=2.0*_deltaTime;
```

```

    [_shaderManager sendUniform1Float:@"deltaTime"
parameter:_speed];
}

else
{
    _rotation+=15.0*_deltaTime;
}
glRotatef(-_rotation, 0.0, 1.0, 0.0);

// VBO rendern
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);

glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
glVertexPointer(3, GL_FLOAT,0, 0);

glBindBuffer(GL_ARRAY_BUFFER, _normalsBuffer);
glNormalPointer(GL_FLOAT,0, 0);

glDrawArrays(GL_TRIANGLES, 0, 1274*3);

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);

[_shaderManager disableShader];

glColor3f(0.0, 0.0, 1.0);
[_font drawTextToScreen:@"Leertaste -> B00000M"
                screenSizeX:bounds.size.width
                screenSizeY:bounds.size.height
                onPositionX:30
                onPositionY:bounds.size.height-50];

```

Wichtig darin ist für uns nur, dass, bevor das Model gerendert wird, der Shader aktiviert werden muss.

```

[_shaderManager useShader:@"Bursting Mesh"];

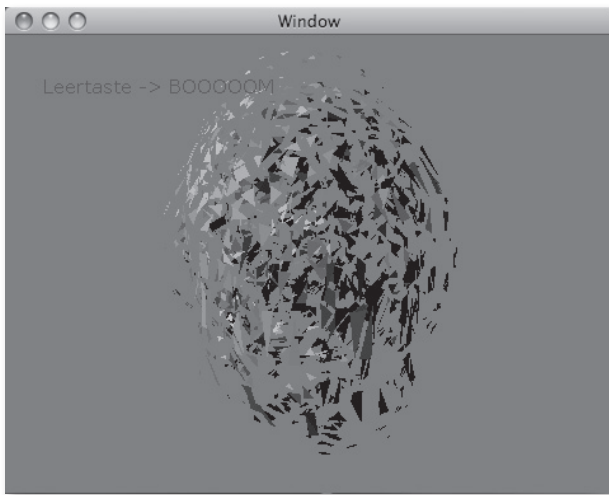
```

Und dann, wenn die Leertaste gedrückt wurde (`_go`), dem Shader die Variable `_speed` übergeben wird, mit welcher die Positionen der Polygone berechnet wird.

Danach schalten wir den Shader wieder ab, da die Textausgabe nicht davon betroffen sein soll.

```
[_shaderManager disableShader];
```

Mehr ist nicht im OpenGL-Code, alles andere geschieht im Vertex-Shader, welchen wir uns jetzt anschauen wollen.



Der Vertex-Shader bringt unser 3D-Model zum platzen.

```
uniform float deltaTime;
```

```
void main()
{
    vec3 normal, lightDir;
    vec4 diffuse;
    float NdotL;

    // Vertex-Normale in Eye-Space Koordinaten
    normal = normalize(gl_NormalMatrix * gl_Normal);

    // Lichtrichtung in Eye-Space Koordinaten
```

```

// Richtungslicht, wobei die Position gleich die Richtung
// ist
lightDir = normalize(vec3(gl_LightSource[0].position));

// Winkel zwischen Vertex-Normale und Lichtrichtung
// Wert liegt zwischen 0.0 - 1.0
NdotL = max(dot(normal, lightDir), 0.0);

// Diffuse Wert berechnen
diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].
diffuse;
gl_FrontColor = NdotL * diffuse;

// Vertices anhand ihrer Normale bewegen
vec4 v = gl_Vertex+ vec4(gl_Normal*deltaTime, 0.0);

// Vertices transformieren
gl_Position = gl_ModelViewProjectionMatrix * v;
}

```

Auch hier ist wieder jede Menge Neues dazu gekommen, wobei die versprochenen 2 Zeilen Quellcode die unteren beiden sind, alle anderen werden für die Berechnung des Lichts benötigt.

Was zunächst passiert, ist, dass wir ein diffuses Licht für unsere Szene berechnen wollen, und zwar deshalb, weil ohne eine Beleuchtung das Model nicht sonderlich schön aussehen würde.

Diese Berechnung des Lichts müssen wir nun selbst erledigen, da wir diesen Part der Fixed-Pipeline ja aushebeln. Schauen Sie sich am besten nochmals die beiden Abbildungen dazu an, diese zeigen, welche Parts wir mit den Shadern selbst übernehmen müssen, wenn diese aktiviert sind.

Diffuses Licht

Bevor wir uns den Code anschauen, müssen wir wissen, wie man das diffuse Licht berechnet. Folgende Formel wird dazu in OpenGL verwendet:

$$C_{diff} = \max\{N \cdot L, 0\} * Md * Ld$$

Formel für diffuses Licht

N = Vertex-Normal

L = Licht-Richtung

Md = Diffuse Farbe des Materials

Ld = Diffuse Farbe des Lichts

C_{diff} = ist die resultierende diffuse Farbe

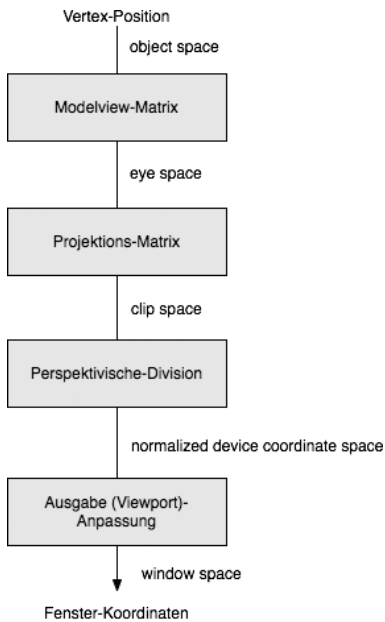
Gehen wir nun die einzelnen Zeilen im Shader nacheinander durch:

Zuerst müssen wir den Vertex-Normal-Vektor (*gl_Normal(...)*) in die sogenannten Eye-Space-Koordinaten umrechnen, da Lichtberechnungen dort gemacht werden müssen. Anschließend wird der Vektor normalisiert *normalize(...)*.

```
// Vertex-Normale in Eye-Space Koordinaten
normal = normalize(gl_NormalMatrix * gl_Normal);
```

Die Position der Lichtquelle ist schon in die Eye-Space-Koordinaten umgerechnet, weshalb hier dieser Schritt entfallen kann. Da wir mit einem Richtungslicht arbeiten, ist die Richtung gleich der Position des Lichts. Auch diesen Vektor müssen wir wieder normalisieren.

```
// Lichtrichtung in Eye-Space Koordinaten
// Richtungslicht, wobei die Position gleich die Richtung ist
lightDir = normalize(vec3(gl_LightSource[0].position));
```

Koordinaten-Räume und Transformationen in OpenGL

Danach wird der Winkel ($\text{dot}(\dots)$) zwischen dem Normal-Vektor und dem Richtungsvektor bestimmt und mit Hilfe der *build-in*-Funktion $\text{max}(\dots)$ in einen Bereich zwischen 0.0–1.0 gebracht.

```
// Winkel zwischen Vertex-Normale und Lichtrichtung
// Wert liegt zwischen 0.0 - 1.0
NdotL = max(dot(normal, lightDir), 0.0);
```

Als Nächstes wird der diffuse Anteil berechnet und in die *build-in*-Variable `gl_FrontColor` gespeichert.

`gl_FrontColor` ist eine *varying*-Variable, welche die Farbe für die Vorderseite des Polygons speichert. Diese wird dann automatisch an den Fragment-Shader übergeben.

```
// Diffuse Wert berechnen
diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
gl_FrontColor = NdotL * diffuse;
```

Damit hätten wir die Formel für das Licht abgearbeitet, nun fehlt nur noch das Bewegen der Vertices, was nicht weiter schwierig ist, auch hier nutzen wir wieder eine *build-in*-Variable (`gl_Normal`), welche ja der Normal-Vektor der Vertices ist. Mit Hilfe dieser Variable und dem *uniform*-Wert `deltaTime` werden die Vertices bewegt.

```
// Vertices anhand ihrer Normale bewegen  
vec4 v = gl_Vertex+ vec4(gl_Normal*deltaTime, 0.0);
```

Zum Schluss müssen wir die diese noch entsprechend transformieren, und fertig ist der Effekt.

```
// Vertices trannsformieren  
gl_Position = gl_ModelViewProjectionMatrix * v;
```

Der Fragment-Shader fällt sehr einfach aus:

```
void main()  
{  
    gl_FragColor = gl_Color;  
}
```

Dabei ist *gl_Color*, wie wir aus den Grundlagen zu GLSL wissen, ein *build-in-Attribut* (*attribute*), welches die aktuelle Farbe enthält. Diese wird dann über *gl_FragColor* ausgegeben. Das war schon alles.

Wenn Sie nochmals in die *prepareOpenGL*-Methode schauen, werden Sie feststellen, dass das Licht gar nicht aktiviert wurde und trotzdem vorhanden ist (diffuses Licht). Der Grund dafür ist, dass wir die Beleuchtung ja nun selbständig im Shader erledigen. Ein sehr großer Vorteil dabei ist auch noch, dass nun keine Änderungen an der State-Machine mehr vorgenommen werden müssen (Licht an, Licht aus usw.), was sich positiv auf die Performance auswirkt.

Material und Beleuchtung

Im folgenden Abschnitt wollen wir uns anschauen, wie man die Beleuchtung von OpenGL mit Hilfe von Shadern nachbauen kann. Wir beginnen zunächst mit einer Per-Vertex-Beleuchtung und werden uns später anschauen, wie man diese so umbauen kann, damit sie noch besser aussieht.

Ambientes Licht

Im vorangegangenen Beispiel haben wir ja bereits den diffusen Anteil für die Beleuchtung selbst berechnet, im nächsten Schritt werden wir den ambienten Anteil mit einbeziehen. Die Formel dazu sehen Sie in der nachfolgenden Abbildung.

$$C_{amb} = G_a * M_a + L_a * M_a$$

Formel für den ambienten Anteil der Farbe

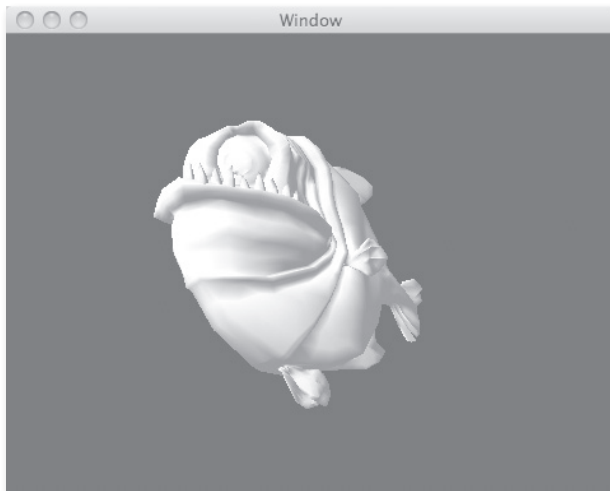
Zerlegen wir nun die Formel:

Ga = globaler ambienter Anteil (Lightmodel)

Ma = Material ambient

La = Licht ambient

Bewaffnet mit dieser Formel, können wir nun unsere Beleuchtung um den ambienten Anteil erweitern. Das Beispielprogramm »Kapitel 12/Vertex Light 01« zeigt ein 3D-Model, welches den neuen Shader beinhaltet



Model mit diffusem und ambientem Licht / Material

Schauen wir uns den Vertex-Shader an (*VertexLight01.vert*):

```
void main()
{
    vec3 normal, lightDir;
    vec4 diffuse, ambient;
    float NdotL;

    // Vertex-Normale in Eye-Space Koordinaten
    normal = normalize(gl_NormalMatrix * gl_Normal);

    // Lichtrichtung in Eye-Space Koordinaten
    // Richtungslicht, wobei die Position gleich die Richtung
    // ist
    lightDir = normalize(vec3(gl_LightSource[0].position));

    // Winkel zwischen Vertex-Normale und Lichtrichtung
    // Wert liegt zwischen 0.0 - 1.0
    NdotL = max(dot(normal, lightDir), 0.0);

    // Diffuse Wert berechnen
    diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].
diffuse;

/*****/
    // Ambient Wert berechnen
    ambient = gl_FrontMaterial.ambient * gl_LightSource[0].
ambient;
    ambient+= gl_LightModel.ambient * gl_FrontMaterial.
ambient;

    // finale Farbe
    gl_FrontColor = NdotL * diffuse + ambient;

    // Vertices transformieren
    gl_Position = ftransform();
}
```

Die Berechnung des diffusen Anteils bis zu der Trennlinie kennen wir ja bereits, was nun folgt, ist der ambiente Anteil.

```
ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
```

Wir nehmen den ambienten Anteil des Materials (Front) und multiplizieren ihn mit dem ambienten Anteil des Lichts.

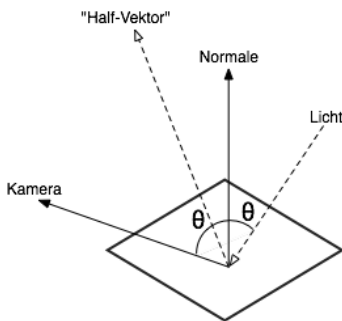
```
ambient+= gl_LightModel.ambient * gl_FrontMaterial.ambient;
```

Anschließend multiplizieren wir noch den ambienten Anteil des *Lighting-Models* mit dem Anteil des Materials und addieren das dann zu unserer Variablen *ambient* hinzu.

Zum Schluss addieren wir den Wert aus *ambient* noch zu der endgültigen Farbe, und schon haben wir den ambienten Anteil berechnet.

Glanz-Anteil

Was uns nun noch fehlt, ist die Berechnung des Glanz-Anteils, welche nicht ganz so einfach ist. Um diesen zu berechnen, verwenden wir das sogenannte Blinn-Phong-Lichtmodell, darin wird der Glanzanteil mit Hilfe des sogenannten Half-Vektors berechnet. Dieser Half-Vektor ist ein Einheitsvektor, der zwischen dem Kamera-Vektor und dem Licht-Vektor liegt. Die folgende Abbildung soll das verdeutlichen.



Berechnung des Glanzanteils im Blinn-Phong-Lichtmodell

Die Intensität des Glanzanteils wird nun anhand des Winkels zwischen dem Half-Vektor und dem Normal-Vektor berechnet. Den Half-Vektor müssen wir nicht selbst berechnen, da GLSL das für uns tut, Was wir aber natürlich müssen, ist, den Glanzanteil zu berechnen. Die Formel dazu zeigt die folgende Abbildung.

$$Spec = (N \bullet H)^s * L_s * M_s$$

Formel zur Berechnung des Glanzanteils

Zerlegen wir wieder die Formel:

N = Normale

H = Half-Vektor

s = konstanter Exponent (Shininess)

L_s = Specular Licht

M_s = Specular Material

Und hier der entsprechende Codeausschnitt aus dem Beispiel »Kapitel 12/Vertex Light 02«, der den Glanzanteil basierend auf der oben genannten Formel berechnet, wobei die Variable $NdotL$ den Winkel zwischen Normal-Vektor und Licht-Vektor beinhaltet und $NdotHV$ den Winkel zwischen dem Normal-Vektor und dem Half-Vektor enthält.

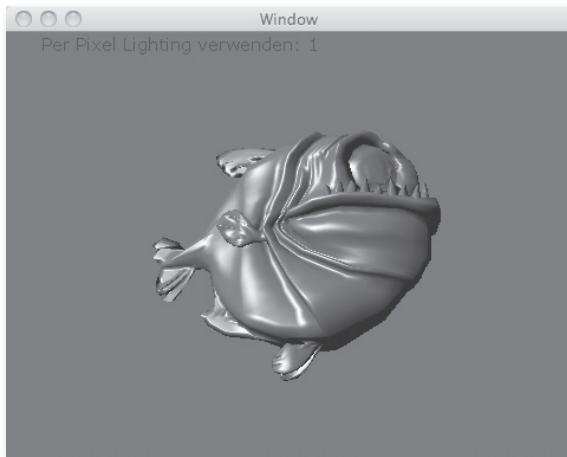
VertexLight02.vert

```
// Wenn der Winkel zwischen Licht und Normale groesser Null, dann
// Glanzanteil berechnen
if (NdotL > 0.0)
{
    NdotHV = max(dot(normal, normalize(gl_LightSource[0].halfVector.
xyz)),0.0);
    specular = gl_FrontMaterial.specular * gl_LightSource[0].specular
* pow(NdotHV,gl_FrontMaterial.shininess);
}
// finale Farbe
gl_FrontColor = NdotL * diffuse + ambient + specular;
```

Das Beispiel beinhaltet die *Build-In*-Funktion *pow(...)*, welche den exponentialen Wert zurückliefert.

Per-Pixel-Beleuchtung

Wir wollen das Thema Licht mit dem Beispiel »Kapitel 12/Per Pixel Light« beenden, in welchem die Beleuchtung der Geometrie nun nicht mehr per Vertex vorgenommen wird, sondern per Pixel, was zum Schluss um einiges besser aussieht. Der Aufwand dazu ist gering, da wir die Vorarbeit schon geleistet haben. Alles, was wir tun müssen, ist, nun die Berechnung der endgültigen Farbe in den Fragment-Shader zu verschieben.



Per-Pixel-Beleuchtung des Models

Beginnen wir mit dem Vertex-Shader:

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

void main()
{
    // Vertex-Normale in Eye-Space Koordinaten
    normal = normalize(gl_NormalMatrix * gl_Normal);

    // Lichtrichtung in Eye-Space Koordinaten
    // Richtungslicht, wobei die Position gleich die Richtung ist
    lightDir = normalize(vec3(gl_LightSource[0].position));

    // Half-Vektor zwischen Kamera-Vektor und
    // Licht-Vektor
    halfVector = normalize(gl_LightSource[0].halfVector.xyz);
```

```

// Diffuser Wert
diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].
diffuse;

// Ambienter Wert
ambient = gl_FrontMaterial.ambient * gl_LightSource[0].
ambient;
ambient += gl_LightModel.ambient * gl_FrontMaterial.ambient;

gl_Position = ftransform();
}

```

Die 3 Vektoren *normal*, *lightDir* und *halfVektor* und die Werte für den diffusen und ambienten Anteil werden weiterhin darin berechnet. Man könnte zwar die beiden Farbinformationen auch im Fragment-Shader berechnen, dies würde aber dazu führen, dass die Arbeitsverteilung der beiden Prozessoren (Vertex und Fragment) sehr ungleichmäßig wäre und der Fragment-Shader im Gegensatz zum Vertex-Shader viel mehr belastet werden würde.

Sie sehen also, dass hier nichts Neues dazugekommen ist, weshalb wir gleich zum Fragment-Shader gehen können.

```

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

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

    vec4 color = ambient;

    // varying Variablen koennen nur gelesen werden
    // deshalb brauchen wir eine neue Variable
    n = normalize(normal);

    // Dot-Produkt zwischen Normal-Vektor und Licht-Vektor
    NdotL = max(dot(n,lightDir),0.0);

    // Diffuser Anteil dazurechnen
    color += diffuse * NdotL;
}

```



```
// Wenn der Winkel zwischen Licht und Normale groesser Null,  
// dann Glanzanteil berechnen  
if (NdotL > 0.0)  
{  
    halfV = normalize(halfVector);  
    NdotHV = max(dot(n,halfV),0.0);  
    color += gl_FrontMaterial.specular * gl_LightSource[0].  
            specular * pow(NdotHV, gl_FrontMaterial.shininess);  
}  
  
// finale Farbe  
gl_FragColor = color;  
}
```

Auch hier ist nicht viel Neues passiert, lediglich die Berechnungen basieren nun auf den *varying*-Variablen, welche vom Vertex-Shader übergeben wurden. Der Rest ist identisch zum vorherigen Beispiel.

TIPP

Noch ein paar Worte zum Normalisieren der Vektoren;

Wie wir aus dem Mathematik-Kapitel wissen, brauchen wir für eine korrekte Lichtberechnung normalisierte Normal-Vektoren. Wenn Sie im Vertex-Shader die Normale normalisieren und an den Fragment-Shader weitergeben, müssen Sie dort die gleiche Prozedur noch einmal durchführen, da beim Interpolieren der Vertex-Daten die Normale auch interpoliert werden, was dazu führt, dass sie unter Umständen nicht mehr die Länge von 1.0 haben.

Im oberen Vertex-Shader könnte man sich nun die Normalisierung der drei Vektoren (*normal*, *lightDir*, *halfVector*) sparen, da der Vorgang, wie gesagt, im Fragment-Shader wiederholt werden muss. Ich habe es deshalb nicht getan, damit sich der Code von dem des vorhergehenden Beispiels nicht zu sehr unterscheidet und dadurch vielleicht zu Verwirrungen führt.

Wenn Sie das Beispiel einmal starten, werden Sie staunen, die Per-Pixel-Beleuchtung sieht um einiges besser aus als die Standard-Beleuchtung von OpenGL, was einfach daran liegt, dass nun für jedes Pixel extra die Beleuchtung durchgeführt wird.

Mit der Leertaste können Sie auf die Per-Vertex-Beleuchtung von vorhin zurückschalten, damit Sie nochmals den Unterschied erkennen können.

Die ganze Schönheit hat aber auch ihren Preis, die Per-Pixel-Beleuchtung ist aufwendiger in der Berechnung und somit langsamer in der Ausführung. Auf der heutigen Hardware dürfte das allerdings nicht weiter ins Gewicht fallen, weshalb man sagen kann, dass diese Art von Beleuchtung schon zum Standard gehört.

Texturierung

Im folgenden Abschnitt werden wir uns einmal anschauen, wie man mit Hilfe von Shadern Polygone texturieren kann. Zunächst werden wir ohne Licht und Material arbeiten, um die Beispiele übersichtlich zu halten.

Bevor wir aber beginnen, müssen wir uns erst noch anschauen, wie der Zugriff auf die Textur-Koordinaten bzw. die Texturen in GLSL geregelt ist.

Textur-Koordinaten

Auf die Textur-Koordinaten, welche wir in unserem OpenGL-Programm definiert haben, können wir mit Hilfe von Attribut-Variablen zugreifen. Da OpenGL über 8 separate Textur-Einheiten verfügt, gibt es auch entsprechend für jede Einheit eigene Variablen, die folgendermaßen definiert sind:

```
attribute vec4 gl_MultiTexCoord0;  
attribute vec4 gl_MultiTexCoord1;  
attribute vec4 gl_MultiTexCoord2;  
attribute vec4 gl_MultiTexCoord3;  
attribute vec4 gl_MultiTexCoord4;  
attribute vec4 gl_MultiTexCoord5;  
attribute vec4 gl_MultiTexCoord6;  
attribute vec4 gl_MultiTexCoord7;
```

AUFGEPASST

Nur der Vertex-Shader hat Zugriff auf diese *attribute*-Variablen.

Der Vertex-Shader verarbeitet diese Textur-Koordinaten und gibt sie an den Fragment-Shader weiter, wobei sie dort in Form der eingebauten *varying*-Variable *gl_TexCoord[i]* zur Verfügung stehen.

Hier einmal ein einfaches Beispiel, in welchem auf die Textur-Koordinaten zugegriffen wird. Diese wurden im OpenGL-Programm für die erste Textur-Einheit festgelegt.

```
void main()
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position = ftransform();
}
```

Hier passiert nicht mehr, als dass die Textur-Koordinaten (erste Textur-Einheit) vom OpenGL-Programm entgegen genommen (*gl_MultiTexCoord0*) und an den Fragment-Shader weitergereicht werden.

Auch das ist das Mindeste, was wir tun müssen, um die Texturierung in GLSL (Vertex-Shader betreffend) zu nutzen.

Weiterhin hat man auch die Möglichkeit, auf die Textur-Matrix zuzugreifen, welche als *uniform*-Array gespeichert ist:

```
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
```

Auch hierzu nochmals ein kleines Beispiel:

```
void main()
{
    gl_TexCoord[2] = gl_TextureMatrix[2] * gl_MultiTexCoord2;
    gl_Position = ftransform();
}
```

Texturen

Wie man Zugriff auf die Koordinaten bzw. die Textur-Matrix hat, wissen wir nun, schauen wir uns jetzt an, wie man an die eigentlichen Texturen herankommt. GLSL bietet dazu einen speziellen Datentyp für den Fragment-Shader, welcher sich *sampler* nennt. Da es in OpenGL verschiedene Textur-Typen gibt (1D, 2D, 3D usw.), existieren auch verschiedene Versionen dieses Datentyps.

sampler1D	Zugriff auf eine 1D-Textur
sampler2D	Zugriff auf eine 2D-Textur
sampler3D	Zugriff auf eine 3D-Textur
samplerCube	Zugriff auf eine Cubemap
sampler1DShadow	Zugriff auf eine 1D-Tiefentextur mit Vergleichsoperation
sampler2DShadow	Zugriff auf eine 2D-Tiefentextur mit Vergleichsoperation

Der Zugriff auf die Texel innerhalb der Textur erfolgt über die Funktion *texture2D(...)*, welche so aussieht:

```
vec4 texture2D(myTexture, gl_TexCoord[0].st);
```

Der zweite Parameter bezieht sich auch hier wieder auf die verwendete Textur-Einheit (0-7). Die beiden Werte (Namesets) *st* stehen für die ersten beiden Elemente im Vektor.

Hier ein einfacher Fragment-Shader, der den Zugriff auf Texturen nochmals verdeutlichen soll:

```
uniform sampler2D myTexture;
void main()
{
    vec4 finalColor = texture2D(myTexture, gl_TexCoord[0].st);
    gl_FragColor = finalColor;
}
```

Es wird eine *sampler2D*-Variable deklariert (*myTexture*), welche aus unserem OpenGL-Programm übergeben wird. Anschließend berechnen wir die aktuelle Farbe (Texel) mit *texture2D(...)* und übernehmen diese dann für unsere aktuelle Farbe.

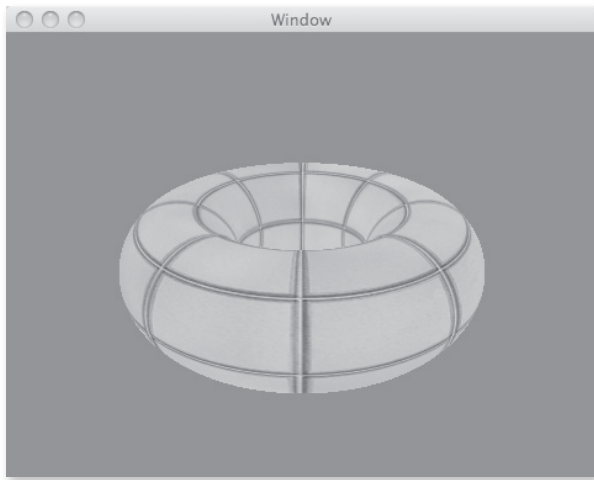
TIPP

Diese Art der Texturierung entspricht »*GL_REPLACE*« beim Setzen der Textur-Umgebung in OpenGL.

Was nun noch fehlt, ist das Übergeben der Textur an den Shader aus unserem OpenGL-Programm, was wir uns im nächsten Beispiel anschauen werden.

Textur-Transformation

Unser erstes Beispiel »Kapitel 12/Simple Texturing« zum Thema Texturen in GLSL macht nichts anderes, als einen Donut (Mesh) zu rendern, bei welchem die Textur-Koordinaten im Vertex-Shader transformiert werden.



Textur-Transformation mit Hilfe des Vertex-Shaders

Beginnen wir zuerst wieder mit dem OpenGL-Programm an sich.

Nachdem unser Shader geladen und aktiviert wurde, laden wir unser Model über die bereits bekannte Klasse CFXWavefrontMesh.

```
// Mesh laden
_mesh = [[CFXWavefrontMesh alloc]init];
[_mesh loadMesh:@"donut"];
```

Nun übergeben wir dem Shader unsere Textur, welche unsere Klasse ja automatisch durch den Eintrag (map_Kd) in der Materialliste (.mtl) geladen hat:

```
#define TEXTURE_UNIT_0 0
[_shaderManager sendUniform1Int:@"texture"
parameter:TEXTURE_UNIT_0];
```

Der Name (*texture*) muss wieder identisch mit der *uniform*-Variablen im Shader sein. Der Parameter, den wir übergeben, ist **nicht** das Textur-Handle (also die ID), sondern die Textur-Einheit (*TEXTURE_UNIT_0*), an welche die Textur gebunden wurde.

Da wir in diesem Beispiel nur eine einzige Textur haben und diese automatisch in der Einheit »Null« liegt, können wir diesen Wert auch so übergeben. Wenn wir mehrere Textur-Einheiten aktiviert haben, müssen wir entsprechend die richtige übergeben (0-7), was wir später auch noch sehen werden.

AUFGEPASST

Bei der Verwendung von Texturen im Shader wird die Textur-Einheit und nicht das Textur-Handle übergeben!

Weil wir eine kleine Animation in unserem Shader haben (dazu kommen wir gleich), müssen wir noch die verstrichene Zeit übergeben, da diese Animation darauf basiert:

```
_speed+=_deltaTime;  
[_shaderManager sendUniform1Float:@"velocity" parameter:_speed];
```

Mehr ist es im OpenGL-Programm nicht.

Kommen wir nun zu den beiden Shadern, in ihnen soll die Textur animiert werden. Die Hauptarbeit liegt dabei im Vertex-Shader.

```
uniform float velocity;  
void main()  
{  
    // Vertex transformieren  
    gl_Position = ftransform();  
  
    // Texture-Koordinaten  
    vec4 c = gl_MultiTexCoord0;  
  
    //Textur-Koordinaten animieren  
    c.y+=sin(velocity);  
  
    // Textur-Koordinaten uebergeben  
    gl_TexCoord[0]=c;  
}
```

Nachdem wir unser Vertex transformiert haben, speichern wir die Textur-Koordinate zuerst einmal in einer temporären Variablen, da wir ja *gl_MultiTexCoord0* nicht verändern können (*attribute*).

Anschließend verändern wir die Y-Koordinate mit einem einfachen Sinuswert und übergeben das Ganze an die Variable *gl_TexCoord[0]*.

Nun der Fragment-Shader dazu:

```
uniform sampler2D texture;  
void main()  
{  
    // finale Farbe  
    gl_FragColor = texture2D(texture, gl_TexCoord[0].st);  
}
```

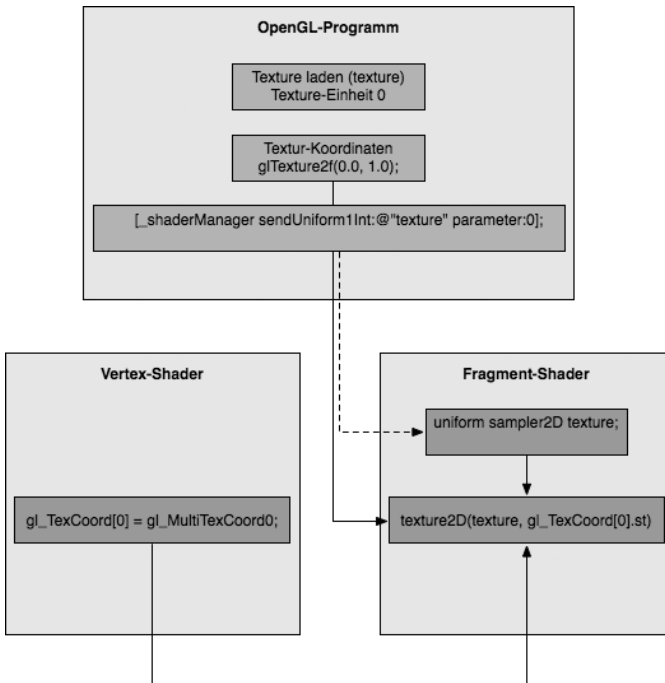
Im Fragment-Shader deklarieren wir unsere Textur (*texture*), nehmen diese und rendern sie als finale Farbe auf den Schirm. Die animierten Textur-Koordinaten befinden sich im Nameset (*st*) des Vektors.

Abschließend zu diesem Beispiel drei Quizfragen:

1. Warum wird kein Material angezeigt, obwohl es in unserem Mesh durch die Material-Datei (.mtl) aktiviert wurde?
2. Wir haben keine Texturierung eingeschaltet, warum sieht man die Textur trotzdem?
3. Was müssen wir tun, damit die Textur auch auf der X-Achse transformiert wird?

Die Auflösung dazu finden Sie am Ende des Kapitels.

Bevor wir weitermachen, soll Ihnen die folgende Abbildung nochmals die genannten Zusammenhänge verdeutlichen.



Zusammenhang bei der Texturierung zwischen dem OpenGL-Programm und den beiden Shadern

Multi-Texturing

Wie man eine Textur benutzt, haben wir nun gesehen, schauen wir uns jetzt an, was nötig ist, um mehrere Texturen gleichzeitig auf die Geometrie aufzubringen.

Im folgenden Beispiel »Kapitel 12/Multi Texturing 01« nutzen wir zwei Texturen mit den gleichen Textur-Koordinaten. Deshalb ändert sich nur der Fragment-Shader.

```
uniform sampler2D texture0;
uniform sampler2D texture1;
void main()
{
    // finale Farbe
    vec4 color0 = texture2D(texture0, gl_TexCoord[0].st);
    vec4 color1 = texture2D(texture1, gl_TexCoord[0].st);

    gl_FragColor = color0+color1;
}
```

Die endgültige Farbe entsteht nun dadurch, dass wir die Texel der beiden Texturen addieren.

Da wir nun zwei Texturen benutzen, müssen wir diese auch in verschiedene Textur-Einheiten binden, der Code dazu ist auch relativ einfach:

```
// Textur laden
_texture0 = [[CFXTextureManager sharedManager] texture-
ByName:[[NSBundle mainBundle]pathForResource:@"texture2"
ofType:@"tif"]];
_texture1 = [[CFXTextureManager sharedManager] texture-
ByName:[[NSBundle mainBundle]pathForResource:@"detail"
ofType:@"tif"]];

// Erste Textur kommt in Einheit 1
glActiveTexture(GL_TEXTURE0);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, _texture0);

// Zweite Textur kommt in Einheit 2
glActiveTexture(GL_TEXTURE1);
glEnable(GL_TEXTURE_2D);
```

```

glBindTexture(GL_TEXTURE_2D, _texture1);

// Shader erzeugen
_shaderManager = [CFXShaderManager sharedManager];

// Werden Shader unterstützt ?
NSLog(@"Shader unterstützt?: %d", [_shaderManager
shadersAvailable]);

[_shaderManager createShader:@"Multi Texture 01"
vertexShader:[NSBundle mainBundle] pathForResource:@"Multi-
Texture01" ofType:@"vert"]
fragmentShader:[NSBundle mainBundle] pathForResource:@"Multi-
Texture01" ofType:@"frag"]];

[_shaderManager useShader:@"Multi Texture 01"];

[_shaderManager sendUniform1Int:@"texture0"
parameter:TEXTURE_UNIT_0];
[_shaderManager sendUniform1Int:@"texture1"
parameter:TEXTURE_UNIT_1];

```

Nachdem die Texturen geladen wurden, wird jeweils eine Textur-Einheit aktiviert und die entsprechende Textur gebunden. Dem Shader übergeben wir dann einfach die beiden Einheiten.

Der Render-Code sieht entsprechend so aus:

```

glBegin(GL_QUADS);
    glMultiTexCoord2f(GL_TEXTURE0, 0.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE1, 0.0, 0.0);
    glVertex3f(-2.0f, -2.0f, 0.0f);

    glMultiTexCoord2f(GL_TEXTURE0, 1.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE1, 1.0, 0.0);
    glVertex3f( 2.0f, -2.0f, 0.0f);

    glMultiTexCoord2f(GL_TEXTURE0, 1.0, 1.0);
    glMultiTexCoord2f(GL_TEXTURE1, 1.0, 1.0);
    glVertex3f( 2.0f,  2.0f, 0.0f);

    glMultiTexCoord2f(GL_TEXTURE0, 0.0, 1.0);

```

```
glMultiTexCoord2f(GL_TEXTURE1, 0.0, 1.0);  
glVertex3f(-2.0f, 2.0f, 0.0f);  
glEnd();
```

Darin werden pro Eckpunkt die Textur-Koordinaten für beide Textur-Einheiten definiert.

Texturen kombinieren

Wenn wir uns die Zeile im Fragment-Shader nochmals anschauen, in welcher die endgültige Farbe berechnet wurde, könnte man doch auf die Idee kommen, auch andere Verknüpfungen der beiden Farben vorzunehmen.

Ganz einfach könnte man z. B. die beiden subtrahieren:

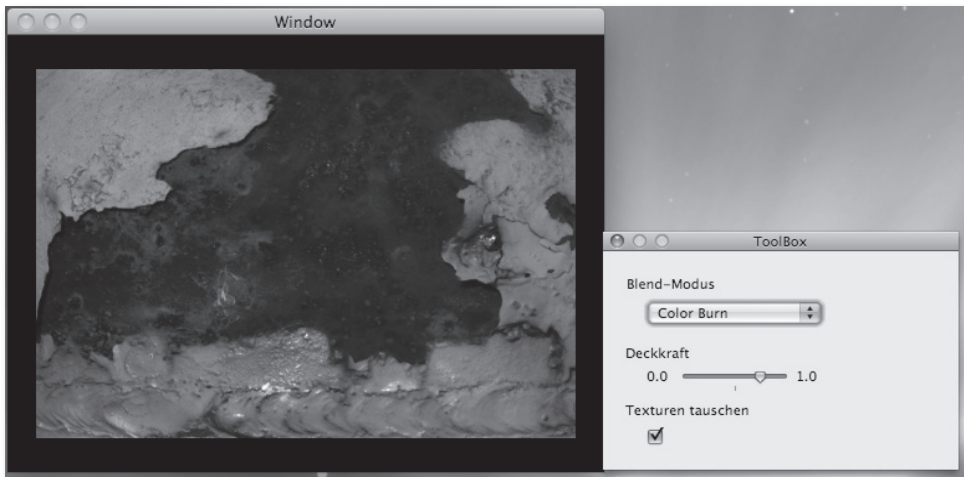
```
gl_FragColor = color0-color1;
```

oder aber multiplizieren oder andere mathematische *build-in*-Funktionen von GLSL nutzen.

Sie werden sehr schnell merken, dass man damit jede Menge interessanter Effekte erzielen kann. Wenn Ihnen beim Ausprobieren die Ideen ausgehen sollten, dann können Sie sich das Programm »Kapitel 12/Blend Modes« einmal genauer anschauen. Darin werden verschiedene Möglichkeiten gezeigt, wie man 2 Texturen übereinander blenden kann. Die benutzten *build-in* Funktionen von GLSL habe ich im Code nochmals kurz erklärt.

Die Idee zum Shader-Code stammt übrigens auch aus dem OrangeBook, wobei die Formeln dazu von Jens Gruschel (<http://www.pegtop.net/delphi/articles/blend-modes/>) kommen. Auf seiner Webseite finden Sie noch mehr Anregungen zu diesem Thema.

Die Ergebnisse sehen zum Teil so aus, wie man sie aus verschiedenen Bildbearbeitungsprogrammen kennt.



Bildbearbeitung mit GLSL à la Photoshop

Texture-Combiners

Im Kapitel 7 hatte ich es ja bereits kurz angesprochen, dass man diese Art der Textur-Kombinierung auch mit sogenannten Texture-Combiners und der Fixed-Pipeline nachbilden kann. Die Handhabung dieser Combiners ist aber, wie ich finde, recht umständlich, was folgender Code zeigen soll:

```
// Textur-Combiners aktivieren
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);

// Es sollen die RGB-Werte moduliert werden
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);

// Erste Textur
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE0);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERANDO_RGB, GL_SRC_COLOR);

// Zweite Textur
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_TEXTURE1);
glTexEnvf(GL_TEXTURE_ENV, GL_OPERANDO_RGB, GL_SRC_COLOR);
```

Dieses Beispiel macht nichts anderes, als zwei Texturen miteinander zu kombinieren. Schon allein die Anzahl der Funktionsaufrufe macht klar, dass man so etwas besser mit Hilfe von Shadern macht, da es dort zum einen schneller geht und zum anderen auch noch übersichtlicher ist.

TIPP

Auf Systemen ohne Shader-Unterstützung sind Texture-Combiners die einzige Möglichkeit, um Texturen flexibel miteinander zu kombinieren, weshalb sie immer noch ihre Daseinsberechtigung haben.

Mehr Multi-Texturing

Das nächste Beispiel »Kapitel 12/Multi Texturing 02« verwendet 4 verschiedene Texturen und verknüpft diese dann zur endgültigen Farbe. Auch hier nehmen wir wieder für alle Texturen die gleichen Textur-Koordinaten, wodurch wir uns wieder nur den Fragment-Shader anschauen müssen, da der Vertex-Shader unverändert geblieben ist:

```
uniform sampler2D texture;  
uniform sampler2D textureRed;  
uniform sampler2D textureGreen;  
uniform sampler2D textureBlue;  
  
void main (void)  
{  
    vec4 tex = texture2D(texture, gl_TexCoord[0].st);  
    vec4 red = texture2D(textureRed, gl_TexCoord[0].st);  
    vec4 green = texture2D(textureGreen, gl_TexCoord[0].st);  
    vec4 blue = texture2D(textureBlue, gl_TexCoord[0].st);  
  
    gl_FragColor = tex+(red*0.9)+(green*0.7)+(blue*0.6);  
}
```

Sie sehen, es passiert nichts Magisches, es werden hier nun anstatt zwei vier Texturen benutzt, welche mit einem beliebigen Wert multipliziert und dann zur endgültigen Farbe dazugerechnet werden.

Je geringer diese Werte sind, umso weniger wird man die jeweilige Textur sehen können.

Wenn Sie pro Textureinheit andere Texturkoordinaten verwenden möchten, müssen Sie den Vertex-Shader so abändern, dass er die veränderten Texturkoordinaten an den Fragment-Shader übergibt.

Vertex-Shader:

```
void main()
{
    // Vertex transformieren
    gl_Position = ftransform();

    // Texture-Koordinaten an Fragment-Shader uebergeben
    gl_TexCoord[0]=gl_MultiTexCoord0;
    gl_TexCoord[1]=gl_MultiTexCoord1;
    .
    .
}
```

Der Fragment-Shader könnte dann z. B. so aussehen:

```
uniform sampler2D texture;
uniform sampler2D textureRed;
uniform sampler2D textureGreen;
uniform sampler2D textureBlue;

void main (void)
{
    vec4 tex = texture2D(texture, gl_TexCoord[0].st);
    vec4 red = texture2D(textureRed, gl_TexCoord[1].st);
    .
    .
    .
}
```

Texturen mit Material und Licht kombinieren

Nun werden wir zu der Texturierung noch die Material- und Lichteigenschaften miteinbeziehen. Wie wir aus dem Kapitel über die Texturierung bereits wissen, gibt es in OpenGL sechs Möglichkeiten, Texturen mit Farben bzw. Materialien zu kombinieren.

Diese waren:

- *GL_MODULATE (Standard)*
- *GL_REPLACE*
- *GL_DECAL*
- *GL_BLEND*
- *GL_ADD*
- *GL_COMBINE*

Im folgenden Beispiel »Kapitel 12/Texture And Material« wollen wir die ersten drei der sechs Modi im Fragment-Shader einmal nachbauen.

Bevor wir aber beginnen, schauen wir uns an, wie sich die endgültige Farbe zusammensetzt:

Modus	Farbe	Alpha
<i>GL_REPLACE</i>	$C = C_t$	$A = A_t$
<i>GL_MODULATE</i>	$C = C_t * C_f$	$A = A_t * A_f$
<i>GL_DECAL</i>	$C = C_f * (1 - A_t) + C_t * A_t$	$A = A_f$

Die Parameter *Ct* und *Cf* stehen für die Farbe der Textur bzw. für die der eingehenden Farbe. Gleiches gilt für *At* (Alphawert Textur) und *Af* (Alphawert eingehende Farbe).

- Der erste Modus (*GL_REPLACE*) macht wie wir wissen nichts anderes, als die Farbe durch die Textur-Farbe zu ersetzen, entsprechend einfach sieht auch die Formel aus.
- Beim Modus *GL_MODULATE* wird einfach die Fragment-Farbe mit der Textur-Farbe multipliziert.
- Der letzte Modus *GL_DECAL* ist ein wenig aufwendiger, wenn die Textur einen Alpha-Kanal besitzt, wird dieser in der Berechnung berücksichtigt, wenn nicht, ist das Ergebnis dasselbe wie im Modus *GL_REPLACE*.

Der Vertex-Shader ist identisch mit demjenigen aus dem Beispiel »Kapitel 12/ Per Pixel Light«, welchen wir oben schon gesehen haben, deshalb lassen wir ihn außen vor und kommen gleich zum Fragment-Shader.

```
varying vec4 diffuse,ambient;
varying vec3 normal,lightDir,halfVector;
uniform sampler2D texture;
uniform int mode;

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

    vec4 color = ambient;

    // varying Variablen koennen nur gelesen werden
    // deshalb brauchen wir eine neue Variable
    n = normalize(normal);

    // Dot-Produkt zwischen Normal-Vektor und Licht-Vektor
    NdotL = max(dot(n,lightDir),0.0);

    // Diffuser Anteil dazurechnen
    color += diffuse * NdotL;

    // Wenn der Winkel zwischen Licht und Normale groesser Null,
    // dann Glanzanteil berechnen
    if (NdotL > 0.0)
    {
        halfV = normalize(halfVector);
        NdotHV = max(dot(n,halfV),0.0);
        color += gl_FrontMaterial.specular * gl_LightSource[0].
            specular * pow(NdotHV, gl_FrontMaterial.shininess);
    }

    vec4 texel = texture2D(texture,gl_TexCoord[0].st);

    // finale Farbe
    if(mode == 0) // GL_REPLACE Farbe wird durch Textur-Farbe
    ersetzt
    {
```



```

        gl_FragColor = texel;
    }
    else if(mode == 1)// GL_MODULATE // Farbe wird mit Textur-Farbe
    multipliziert
    {
        gl_FragColor = texel*color;
    }
    else if(mode == 2)// GL_DECAL // Alpha-Kanal wird berueck-
    sichtigt, wenn vorhanden, wenn nicht ist das Ergebnis wie bei
    GL_REPLACE
    {
        vec3 tmp = color.rgb *(1.0-texel.a)+texel.rgb*texel.a;
        gl_FragColor = vec4(tmp,  color.a);
    }
}

```

Der Fragment-Shader basiert auch auf dem »Per-Pixel-Light«-Beispiel, wobei jetzt noch die *uniform*-Variable *mode* hinzugekommen ist.

Die Berechnung des Lichts und des Materials ist gleich geblieben. Kommen wir zu dem Teil, in welchem zwischen den 3 Modi umgeschaltet wird.

Auffallend ist zunächst einmal, dass keine *switch*-Anweisung verwendet wurde, was daran liegt, dass GLSL diese nicht kennt. Deshalb muss ein einfacher *if*-Block die Arbeit übernehmen.

Die Berechnungen der Fragment-Farbe basieren genau auf den Formeln, die oben in der Tabelle stehen.

Hervorheben möchte ich nochmals den Zugriff auf die einzelnen Elemente im Vektor. Man sieht, dass es ohne Probleme möglich ist, auf mehrere Elemente (*color.rgb*) gleichzeitig zuzugreifen. GLSL kümmert sich darum, dass die richtigen Werte zugewiesen werden.

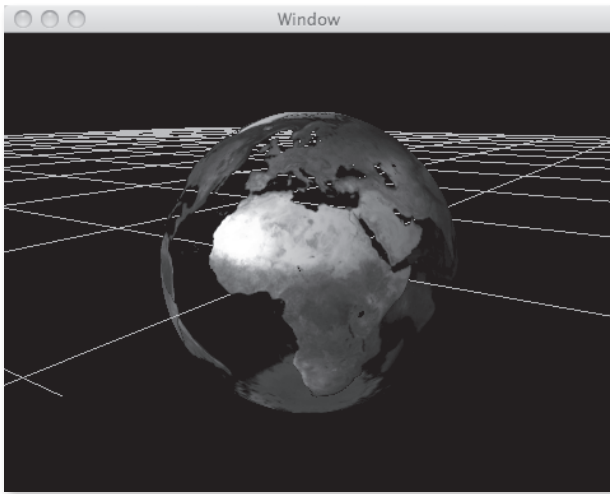
Wichtig ist dabei, dass man keine Elemente aus anderen Namesets mischt, eine Zuweisung wie *color.rgz* wäre demnach nicht erlaubt.

Alpha-Masking

Im Kapitel 7 haben wir schon gesehen, wie man mit Hilfe von Alpha-Masking Teile einer Textur transparent darstellen kann. Diesen Effekt kann man auch sehr

einfach mit einem Shader nachbauen. Das Prinzip dabei ist gleich geblieben, das bedeutet dass man mit Hilfe des Alpha-Kanals in der Textur die Teile maskiert (schwarz), welche später transparent sein sollen.

Für das Maskieren selbst benutzen wir das Keyword *discard*. Dieses macht nichts anderes, als das Shader-Programm vorzeitig zu beenden.



Alpha-Masking mit Hilfe des GLSL-Keywords discard

Schauen wir uns also in einem Beispielprogramm »Kapitel 12/Discard« an, wie man das Alpha-Masking mit einem Shader und dem Keyword *discard* macht.

GRUNDLAGEN

Achtung: Das Keyword *discard* ist nur im Fragment-Shader verfügbar.

Die Funktionsweise ist so trivial, dass Sie mit Sicherheit auch selbst darauf kommen würden. Zuerst brauchen wir natürlich eine Textur, die einen Alpha-Kanal beinhaltet.

```
vec4 texel = texture2D(texture, gl_TexCoord[0].st);
```

Anschließend prüfen wir, ob der Alpha-Wert der Textur *texel.a* kleiner als 0.1 (schwarz) ist. Wenn das so ist, beenden wir den Shader über *discard* und schreiben dadurch keine Informationen in den Framebuffer, wodurch wir unsere transparenten Bereiche erhalten.

```
if(texel.a <0.1)
{
    discard;
}
```

Das war schon alles. Den Rest des Shader-Codes kennen Sie bereits.

Alpha-Masking ohne Alpha-Kanal

Wenn Sie eine Textur nutzen wollen bzw. müssen, die keinen Alpha-Kanal hat, funktioniert das Alpha-Masking trotzdem, wobei dann ein wenig mehr Arbeit nötig ist. Zuerst benötigen Sie 2 Texturen, eine Farb-Textur und eine schwarz-weiße, welche die Alpha-Maske darstellen soll. Beide Texturen übergeben Sie dem Fragment-Shader, der so aussehen könnte:

```
uniform sampler2D texture;
uniform sampler2D alpha;

vec4 texel = texture2D(texture, gl_TexCoord[0].st);
vec4 alpha_color = texture2D(alpha, gl_TexCoord[0].st);

if(alpha_color.r<0.1)
{
    discard;
}
gl_FragColor = texel*color;
```

Sie prüfen also nur, ob z. B. der Rotwert der schwarz-weißen Textur kleiner als 0.1 ist, und springen dann aus dem Shader-Code heraus.

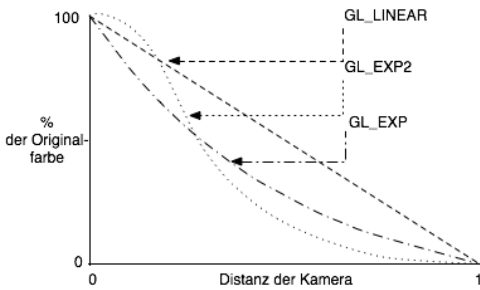
Nebel

Im Kapitel 10 haben wir uns ja schon einmal mit dem Nebel beschäftigt und auch gesehen, dass man mit Hilfe der Funktion *glHint(GL_FOG_HINT, GL_NICEST)*; den Nebel per Pixel berechnen lassen kann, wobei die Betonung auf »kann« liegt, da diese *Hints* ja nur Hinweise und keine Befehle im klassischen Sinn sind.

Wir werden im folgenden Abschnitt den Nebel nun selbst per Pixel berechnen und auch gleich noch einen netten Effekt miteinbauen.

Nebelberechnung

Wie wir wissen, wird die Nebelfarbe mit dem eingehenden Fragment anhand eines Blendfaktors zusammengemischt, wobei wir die drei Faktoren (*GL_EXP*, *GL_EXP2*, *GL_LINEAR*) schon in einem Beispiel kennengelernt haben.



Nebelfaktorentwicklung, basierend auf dem Abstand der Kamera

Da wir nun aber den Nebel selbst berechnen wollen, müssen wir zunächst einmal die Formeln kennen, auf welchen die Blendfaktoren basieren. Wichtig dabei ist zu wissen, dass *Blendfaktor* im Bereich zwischen 0.0 – 1.0 liegen muss, was wir aber später noch sehen werden.

$$\text{Blendfaktor} = e^{-(\text{density} \cdot z)}$$

Exponential GL_EXP

$$\text{Blendfaktor} = e^{-(\text{density} \cdot z)^2}$$

Exponential GL_EXP2

$$\text{Blendfaktor} = \frac{\text{end} - z}{\text{end} - \text{start}}$$

Linear GL_LINEAR

Jede einzelne dieser Berechnungen können nun sowohl im Vertex-Shader als auch im Fragment-Shader durchgeführt werden, wobei auch hier der Per-Fragment- (oder Per-Pixel-)Nebel besser aussieht. Bei sehr aufwendigen Szenen wird man aber bei dem Per-Pixel-Nebel auf jeden Fall einen Performance-Verlust feststellen können, wodurch man auch hier als Entwickler selbst abschätzen muss, was »besser« ist.

Innerhalb des Shaders hat man Zugriff auf die einzelnen Werte des Nebels mittels der *build-in-uniform*-Variablen

```

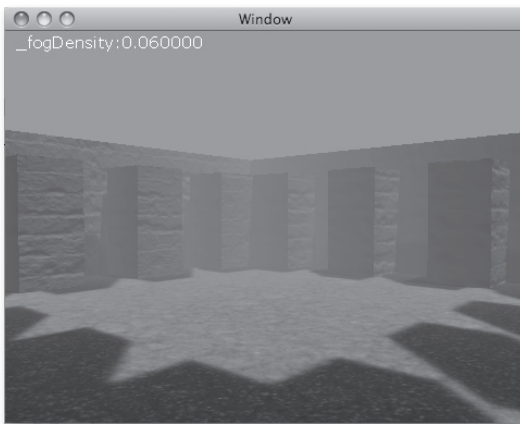
struct gl_FogParameters {
    vec4 color;
    float density;
    float start;
    float end;
    float scale;
};
uniform gl_FogParameters gl_Fog;

```

die wir im Prinzip alle schon im Kapitel 10 kennengelernt haben, bzw. der *build-in-varying*-Variable *gl_FogFragCoord*, welche die Nebelkoordinaten des Fragmentes enthält.

Per-Pixel-Nebel

Im letzten Beispiel »Kapitel 12/Per Pixel Fog« zum Thema Shader werden wir den Nebel anhand des Blendfaktors *GL_EXP2* erstellen. Die Szene kommt komplett ohne Beleuchtung aus, wodurch der Code schön übersichtlich bleibt.



Per-Pixel-Nebel mit einem weichen Farbverlauf

Da wir den Nebel komplett im Fragment-Shader berechnen, fällt der Vertex-Shader sehr einfach aus.

Vertex-Shader:

```

void main()
{
    gl_FrontColor = gl_Color;
    gl_Position = ftransform();
}

```

```

        gl_TexCoord[0] = gl_MultiTexCoord0;
    }

```

Alles, was wir tun, ist, die eingehenden Werte an den Fragment-Shader durchzureichen.

Fragment-Shader:

```

uniform sampler2D texture;
uniform float density;
void main(void)
{
    //Exponential Nebel
    const float e = 1.442695; // = 1 / log
    // Fragment Z-Koordinate
    float z = gl_FragCoord.z / gl_FragCoord.w;
    // GL_EXP2
    float fogFactor = exp2(-density*density*z*z*e);
    // Wert zwischen 0.0 - 1.0 druecken
    fogFactor = clamp(fogFactor, 0.0, 1.0);
    // Nebel Farbe
    vec4 fogColor = vec4(0.7, 0.5, 0.2+ (1.0-z), 1.0);
    // Textur
    vec4 texel = texture2D(texture, gl_TexCoord[0].st);
    // finale Farbe
    gl_FragColor = mix(fogColor, texel, fogFactor);
}

```

Die Nebeldichte (*density*) kann vom Programm aus über die Plus- bzw. Minustaste verändert werden.

Wir berechnen zuerst die Distanz des Pixels zur Kamera und speichern diesen Wert in der Variablen *z*. Dann wird der Faktor für den *GL_EXP2* Nebel berechnet, dies geschieht mit der *build-in*-Funktion *exp2*, welche den Wert hoch 2 zurückgibt. Mit der Funktion *clamp* beschränken wir dann den zurückgelieferten Wert auf einen Bereich von 0.0–1.0.

Anschließend wird die Nebelfarbe definiert. Diese ist von mir willkürlich gewählt worden und enthält den zu Anfangs erwähnten Zusatzeffekt. Der Blauwert wird nämlich abhängig von der Z-Koordinate des Fragments berechnet, wodurch der Nebel einen Farbverlauf bekommt. Dies ist zwar nicht sonderlich realistisch, wirkt aber irgendwie nicht so »normal«.

TIPP

Die offizielle Farbgebung des Nebels ohne Farbverlauf wäre demnach

```
vec4 fogColor = vec4(0.7, 0.5, 0.2, 1.0);
```

worauf Sie aber bestimmt auch selbst gekommen wären.

Abschließend mischen (*mix*) wir die Farbe anhand der Nebelfarbe, des Faktors und der Textur zur endgültigen Farbe.

Soweit erst einmal unser Ausflug in die Shader-Programmierung. Was wir bis jetzt gesehen haben, ist nur ein kleiner Bruchteil dessen, was mit Shadern möglich ist. Wenn Sie sich aktuelle Spiele anschauen, werden Sie sehr schnell feststellen, dass so gut wie kein aktueller Titel mehr ohne sie auskommt. Erst mit Einsatz von Shadern lässt sich die Grafikkarte bis auf das letzte Pixel ausreizen. Wenn Sie Gefallen an der Shader-Entwicklung gefunden haben, empfehle ich Ihnen auf jeden Fall das »Orange Book«, in welchem Sie alle nötigen Informationen zum Thema finden werden.

Entwicklungsumgebung

Da es sich bei GLSL um eine echte Programmiersprache mit all ihren Tücken handelt, stellt sich die Frage, in welcher Umgebung man Shader eigentlich entwickelt. Leider sieht es in Bezug auf Entwicklungsumgebungen für die Shader-Entwicklung auf dem Mac sehr mager aus. Der einzige mir bekannte Editor (wenn man davon überhaupt sprechen kann) ist das GLSEditorSample-Beispiel, welches bei den Developer-Examples dabei ist. Dieser bietet aber leider nur die wichtigsten Funktionen, so dass er wie gesagt nur bedingt brauchbar ist.

Um einiges besser (wen wundert's) sieht es in der Windows-Welt aus. Hier gibt es gleich mehrere sehr gute Entwicklungsumgebungen (RenderMonkey, FXComposer), die auch gleich eine Menge Beispiel-Shader mitbringen. Die Liste finden Sie weiter unten.

Zusätzliche Informationen

Wenn Sie nach zusätzlichen Tutorials bzw. Informationen zu GLSL im Internet suchen, werden Sie sehr schnell merken, dass es nur sehr wenige Seiten gibt, die sich mit diesem Thema beschäftigen. Ein Grund dürfte wohl sein, dass diese Technologie noch recht jung ist, andererseits darf man nicht vergessen, dass die Mehrzahl der Entwickler mit DirectX (HLSL ist dort die Shader-Sprache) entwickelt.

Eine sehr interessante Alternative zu GLSL ist Cg von NVidia. Diese Shader-Sprache ist sehr an GLSL angelehnt, so dass sich mit ein wenig Übung Cg-Code in GLSL umschreiben lässt. Das Tolle an Cg ist weiterhin, dass es sehr viele Informationen dazu im Internet gibt. Ich empfehle Ihnen auf jeden Fall, sich Cg mal ein wenig näher anzusehen.

Cg

http://developer.nvidia.com/page/cg_main.html

Entwicklungsumgebungen

<http://www.typhoonlabs.com/>

<http://ati.amd.com/developer/rendermonkey/>

http://developer.nvidia.com/object/fx_composer_home.html

GLSL

Die GLSL-Bibel (Orange Book)

<http://www.3dshaders.com/home/>

Kurzreferenz zu GLSL

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf

Beleuchtungsmodelle

Blinn-Phong

<http://de.wikipedia.org/wiki/Blinn-Beleuchtungsmodell>

Phong

<http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell>

Lambert

<http://de.wikipedia.org/wiki/Lambert-Beleuchtungsmodell>

Ambient Occlusion

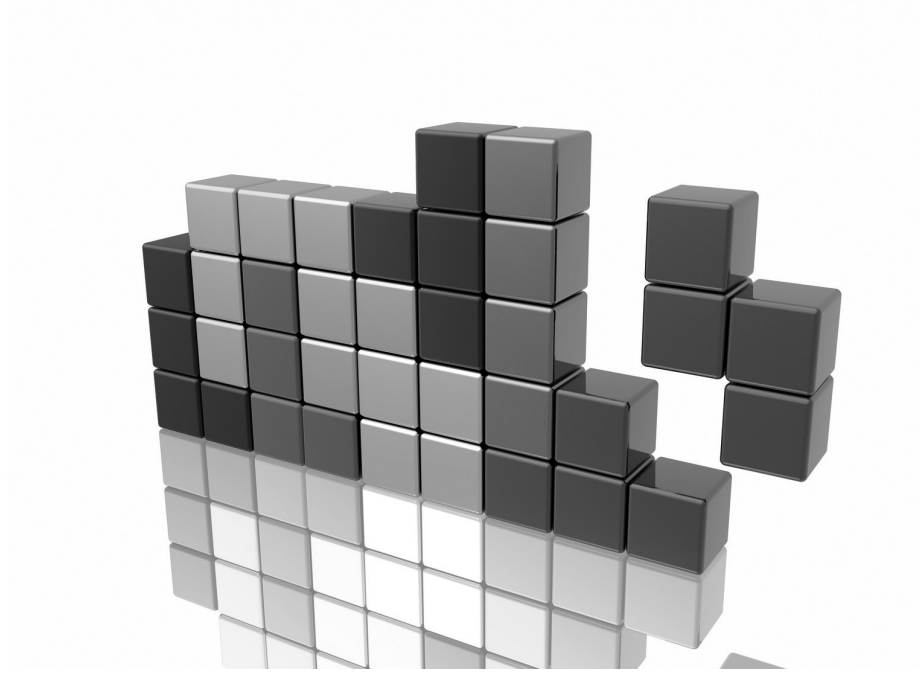
http://de.wikipedia.org/wiki/Ambient_Occlusion

Auflösung Textur-Quiz

Weil wir kein Licht / Material in unseren Shadern verarbeitet haben.

Es ist nicht nötig, die Texturierung einzuschalten, da wir die Fixed-Pipeline durch die Nutzung von Shadern ja umgehen.

```
c.xy+=sin(velocity)
```

Sound-Entwicklung mit OpenAL

Nachdem wir nun jede Menge Beispiele zu OpenGL erstellt haben, wird es nun Zeit, dass wir uns auch einmal dem Thema »Soundausgabe« widmen. Nicht weniger wichtig als eine coole Grafik ist in einem Spiel auch die Musik bzw. die Soundeffekte. Professionelle Entwickler-Studios gehen inzwischen soweit, dass die Musik für Spiele (z. B. Gothic 3) nicht mehr nur am Computer erstellt, sondern in einem Studio mit echtem Orchester live eingespielt wird. Der Aufwand, der heute dabei betrieben wird, gleicht dem einer Filmproduktion.

Dass man aber auch mit recht bescheidenen Mitteln eine gute musikalische Begleitung erstellen kann, haben Musiker wie z. B. Chris Hülsbeck oder Rob Hubbard schon vor über 20 Jahren gezeigt. Sie komponierten damals am 64er Musiktitel, die heute wahre Klassiker sind. Wer jemals Commando (Rob Hubbard) gespielt hat, wird wissen, was ich meine.

Das Erstellen von Musik und Sound-Effekten ist heute zwar immer noch eine Kunst für sich, aber mit einem Internetzugang und Programmen wie z. B. Garage-Band kann man auch als Laie etwas einigermaßen Vernünftiges auf die Beine stellen. Wir wollen uns in diesem Kapitel nicht mit dem Erstellen von Sounds am Mac beschäftigen, sondern damit, wie man sie ausgeben kann.

Soundausgabe am Mac

Verantwortlich für die Soundausgabe am Mac ist Core Audio. Diese Low-Level-Schnittstelle bietet umfangreiche Funktionalitäten für das Aufnehmen, Bearbeiten, Erzeugen und Abspielen von Sound- bzw. Midi-Dateien. Außerdem regelt Core Audio den Zugriff auf Audio-Hardware. Die Handhabung dieser Schnittstelle ist nicht ganz einfach (was wohl an der Vielzahl ihrer Möglichkeiten liegt), weshalb sie für unsere Zwecke nicht sonderlich geeignet ist. Es ist zwar ohne Probleme möglich, Core Audio für die Soundausgabe zu nutzen, aber wie gesagt ist es ziemlich mühselig, sich mit dem Thema zu beschäftigen.

OpenAL

Wie in der Einleitung schon erwähnt, werden wir OpenAL für die Soundausgabe nutzen. Diese Schnittstelle bietet genau die Funktionalität, welche wir benötigen, und ist obendrein noch sehr einfach in der Handhabung. OpenAL ist eine plattformunabhängige 3D-Audio-Bibliothek (entwickelt von Creative Labs) und bildet sozusagen das musikalische Gegenstück zu OpenGL. Sie wurde hauptsächlich für

die Spiele-Programmierung entwickelt, wobei es auch Audioprogramme gibt, die mit OpenAL arbeiten.

Leider hat sie den Nachteil, dass sie keine MP3-Dateien abspielen kann, was uns aber nicht weiter stört, da wir Quicktime dafür nehmen können.

ALUT (OpenAL Utility Kit)

Ähnlich wie das glut (OpenGL Utility Kit) gibt es für OpenAL eine zusätzliche Bibliothek, welche die Handhabung von OpenAL vereinfachen soll. Wir werden ALUT dazu verwenden, um wav-Dateien zu laden, da OpenAL selbst diese Möglichkeit nicht bietet. Alle anderen Features von ALUT werden wir außen vor lassen, da wir ja sehen möchten, wie OpenAL funktioniert.

OpenAL einbinden

Bevor wir mit OpenAL arbeiten können, müssen wir folgende Dateien in unser Projekt einbinden:

- Standard OpenAL-Header
`#import <OpenAL/al.h>`
- ALC-Header (Audio Library Context) dient der Verwaltung von Kontexten
`#import <OpenAL/alc.h>`
- Das OpenAL-Framework (OpenAL.framework)

OpenAL initialisieren

Die Initialisierung von OpenAL besteht im Grunde aus 4 Bestandteilen.

- Context
Stellt die Verknüpfung zum Sound-System am Rechner her. Vergleichbar mit dem OpenGL-Kontext, welcher die Verknüpfung zur grafischen Ausgabe herstellt.
- Buffer
Beinhaltet die Rohdaten der Sounddatei, inklusive einiger Informationen, welche z. B. festlegen, wie der Sound abgespielt werden soll.
- Source
Die Position, von der aus der Sound im Raum kommen soll. Vergleichbar mit einem Lautsprecher, der irgendwo im Zimmer aufgestellt ist.
- Listener
Der User bzw. die Kamera im Raum

Kontext erstellen

Die Erstellung eines Kontexts ist recht einfach, was folgende Zeilen zeigen:

```
ALCdevice *newDevice = alcOpenDevice(NULL);
ALCcontext *newContext = alcCreateContext(newDevice, 0);

alcMakeContextCurrent(newContext);
if(alcGetError() != AL_NO_ERROR)
{
    return NO;
}
```

Wir übergeben der Funktion *alcOpenDevice* *NULL*, was bedeutet, dass wir die Standard-Soundschnittstelle (Soundkarte) zur Ausgabe nutzen möchten. Als Nächstes erzeugen wir einen Kontext, der wie bei OpenGL die Verknüpfung zur Ausgabe darstellt. Diesem Kontext übergeben wir die Schnittstelle (*device*). Über den zweiten Parameter ist es möglich, weitere Attribute zu setzen, was wir aber nicht benötigen.

Über die Funktion *alcMakeContextCurrent* setzen wir unseren Kontext als den aktuellen.

Die letzte Zeile betrifft die Fehlerbehandlung.

OpenAL Fehlerbehandlung

Das Abfragen von Fehlern in OpenAL ist ein wenig gewöhnungsbedürftig und funktioniert folgendermaßen:

Sobald ein Fehler auftritt, wird ein Fehlerstatus gesetzt, der solange aktiv bleibt, bis er über die Funktion *alGetError* abgefragt wurde. Nachdem der Status »abgeholt« wurde, wird der Fehlerstatus wieder gelöscht, und der Zustand, der zuvor aktiv war (vor *glGetError*), wird wieder aktiviert.

Um nun einen Fehler abzufragen, geht man so vor (Pseudo):

```
glGetError
rufe eine OpenAL-Funktion auf
if(glGetError!= AL_NO_ERROR)
alles hat funktioniert
else
ein Fehler ist aufgetreten
```

Folgende Fehlercodes können dabei zurückgegeben werden:

AL_NO_ERROR	Es ist kein Fehler aufgetreten.
AL_INVALID_NAME	Ungültiger Namen-ID wurde übergeben.
AL_INVALID_ENUM	Ungültiger enum-Wert
AL_INVALID_VALUE	Ungültiger Wert
AL_INVALID_OPERATION	Ungültiger Funktionsaufruf
AL_OUT_OF_MEMORY	Es konnte nicht genügend Speicher allokiert werden.

Nun aber wieder zurück zum Kontext;

Der nächste Schritt besteht nun darin, eine Sounddatei zu laden. Dieser Schritt beinhaltet auch das Erstellen einer Soundquelle (Source), eines Puffers (Buffer) und eines Listeners (das sind wir vor dem Bildschirm), weshalb wir uns diesen ganzen Part in einem Durchlauf anschauen werden.

Wie schon erwähnt, kann OpenAL selbst keine Sounddateien laden, weshalb hier die ALUT aushelfen muss. Diese Bibliothek beinhaltet die Funktion *alutLoadWAVFile*, welche eine wav-Datei laden kann. Schauen wir uns zuerst einmal den Code an, der nötig ist, um eine Sounddatei zu laden.

```
ALenum format;
ALsizei size;
ALvoid* data;
ALsizei freq;

// Puffer anlegen
alGetError();
alGenBuffers(1, &_buffer);
if (alGetError() != AL_NO_ERROR)
    return NO;

// Wav laden
alGetError();
alutLoadWAVFile([path cStringUsingEncoding:NSUTF8StringEncoding],
&format, &data, &size, &freq);
if (alGetError() != AL_NO_ERROR)
    return NO;

// Wav puffern
```

```
alGetError();
alBufferData(_buffer, format, data, size, freq);
if (alGetError() != AL_NO_ERROR)
    return NO;

// Wav loeschen, da die daten nun im Puffer liegen
alGetError();
alutUnloadWAV(format, data, size, freq);
if (alGetError() != AL_NO_ERROR)
    return NO;

// Soundquelle erzeugen
alGetError();
alGenSources(1, &_source);
if (alGetError() != AL_NO_ERROR)
    return NO;

// Puffer an Soundquelle binden
alGetError();
alSourcei(_source, AL_BUFFER, _buffer);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Wie Sie am Anfang des Codes sehen können, bringt auch OpenAL einen eigenen Satz Datentypen mit, welche den gleichen Nutzen haben wie die Datentypen in OpenGL.

Zunächst einmal erstellen wir einen Puffer, der später unseren Sound beinhalten soll:

```
// Puffer anlegen
alGetError();
alGenBuffers(1, &_buffer);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Hier sehen Sie auch noch mal, wie die Fehlerabfrage eingebaut wird.

Anschließend wird die Datei geladen.

```
// Wav laden
alGetError();
```

```
alutLoadWAVFile([path cStringUsingEncoding:NSUTF8StringEncoding],
&format, &data, &size, &freq);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Am Präfix der Funktion können Sie erkennen, dass sie aus der ALUT stammt.

Um den Aufbau der wav-Datei (Samples, Bitrate usw.) müssen wir uns keine Gedanken machen, da dies alles von ALUT gemanagt wird. Die vier Parameter *format*, *data*, *size* und *freq*, werden für den Puffer benötigt, diese beinhalten einige Informationen, die dieser wiederum benötigt.

```
// Wav puffern
alGetError();
alBufferData(_buffer, format, data, size, freq);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Wenn der Ladevorgang erfolgreich war, werden die Daten der wav-Datei gepuffert, dabei werden der Funktion *glBufferData* genau diese Informationen mitgegeben, welche wir im Ladevorgang abgefragt haben.

Die wav-Datei kann nun wieder gelöscht werden, da die Daten ja nun in einem Puffer liegen.

```
// Wav loeschen, da die daten nun im Puffer liegen
alGetError();
alutUnloadWAV(format, data, size, freq);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Im nächsten Schritt müssen wir eine Soundquelle erzeugen. Diese ist unser virtueller Lautsprecher im 3D-Raum, von welchem der Sound später zu hören sein soll.

```
// Soundquelle erzeugen
alGetError();
alGenSources(1, &_source);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Und zum Schluss müssen wir noch den Puffer mit der Soundquelle verbinden:

```
// Puffer an Soundquelle binden
alGetError();
alSourcei(_source, AL_BUFFER, _buffer);
if (alGetError() != AL_NO_ERROR)
    return NO;
```

Das war alles.

Sie sehen, die Arbeitsweise ist sehr stark an OpenGL angelehnt, wodurch die Einarbeitungszeit relativ kurz ausfällt. Im Prinzip könnten wir nun die Sounddatei abspielen, wobei wir noch eine Kleinigkeit beachten müssen.

OpenAL ist eine 3D-Soundbibliothek, was bedeutet, dass wir zumindest eine Position für die Soundquelle und eine für den Zuhörer einrichten sollten, damit wir später wissen, »wer« sich »wo« im Raum befindet.

```
alListener3f(AL_POSITION, pos.x, pos.y, pos.z);
alSource3f(_source, AL_POSITION, pos.x, pos.y, pos.z);
```

GRUNDLAGEN

Auch in OpenAL werden die Eigenschaften (Position, Geschwindigkeit usw.) mit Standardwerten besetzt, die meistens einen Wert von 0 haben.

Wie auch in OpenGL gibt es in OpenAL die meisten Funktionen in verschiedenen Versionen:

```
alSourcef, alSource3f, alSourcefv, ...
alListenerf, alListener3f, alListenerfv, ...
```

Weiterhin stehen Funktionen zur Verfügung, mit welchen man bestimmte Statusinformationen abfragen kann.

```
alGetListenerfv(...)
alGetSource3f(...)
```

Eine vollständige Liste aller Funktionen finden Sie auf der Webseite zu OpenAL. Den Link dazu finden Sie am Ende des Kapitels.

POWER

Hierzu noch ein Tipp, welcher im Übrigen auch für OpenGL gültig ist; Vermeiden Sie es zur Laufzeit, über die Getter-Funktionen Informationen abzufragen. Diese holen Sie immer direkt vom Treiber, was zu Lasten der Performance geht. Besser ist es, die Statusinformationen beim Programmstart zu holen und dann in einer eigenen Klasse zu kapseln, da der Zugriff auf diese auf jeden Fall schneller ist.

OpenAL beenden

Wenn man OpenAL nicht länger benötigt, sollte man den Kontext und die Schnittstelle wieder freigeben.

```
// Kontext und Device wieder freigeben
ALCcontext *context = alcGetCurrentContext();
ALCdevice *device = alcGetContextsDevice(context);

alcDestroyContext(context);
alcCloseDevice(device);
```

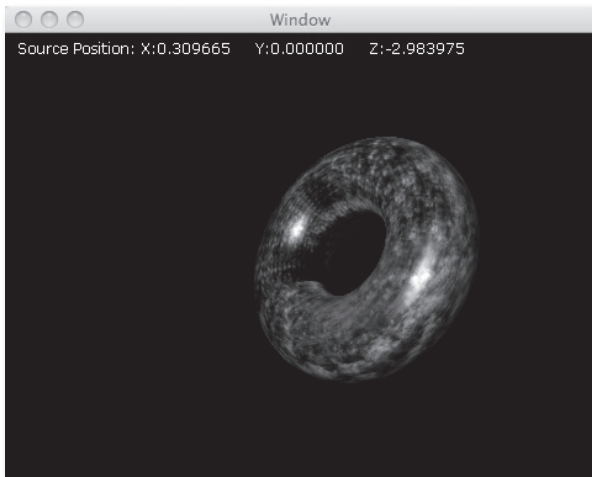
AUFGEPASST

Achtung: Die Reihenfolge, in welcher der Kontext und die Schnittstelle freigegeben werden, ist unbedingt einzuhalten, da eine Schnittstelle erst wieder freigegeben werden kann, wenn kein Kontext mehr besteht.

Alle Soundquellen (source) werden dadurch ebenfalls wieder gelöscht.

Beispiel

Nun wird es wieder Zeit, eine praktische Übung zu machen. Im Beispielprogramm »Kapitel 13/OpenAL simple« werden nochmals genau die Schritte wiederholt, die eben besprochen wurden.



Erstes Beispielprogramm zu OpenAL

Den Code zur Initialisierung von OpenAL können wir außen vor lassen, einzig die Zeile

```
// Source Loop
alSourcei(_source, AL_LOOPING, 1);
```

ist neu dazugekommen, diese macht nichts anderes, als unseren Sound geolooped abspielen.

Im Programm wird ein rotierender Donut gerendert, der die Position der Soundquelle darstellen soll. Der Code sollte bis auf die Berechnung der Position von der Soundquelle (*_meshPosition*) nichts Neues mehr für Sie sein. Hier wollte ich einfach einmal eine Alternative zu *glRotatef(...)* zeigen. Wenn Sie möchten, können Sie natürlich auch die Standard-OpenGL-Rotation verwenden.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
gluLookAt(_listenerPosition.x, _listenerPosition.y,
_listenerPosition.z,
0.0, 0.0, 0.0,
0.0, 1.0, 0.0);
```

```
// Mesh um den Ursprung rotieren
_meshPosition.x = cosf(DEG2RAD(_angle)) * -3.0f;
_meshPosition.z = sinf(DEG2RAD(_angle)) * -3.0f;
```

```

_angle +=30.0*_deltaTime;

// OpenGL Rotation Mesh
_rotation +=40.0*_deltaTime;

// Source Position updaten
alSource3f(_source, AL_POSITION, _meshPosition.x, _meshPosition.y, _meshPosition.z);

// Mesh rendern
glTranslatef(_meshPosition.x, _meshPosition.y, _meshPosition.z);
glRotatef(_rotation, 1.0, 0.0, 1.0);
[_mesh renderMesh];

// Beleuchtung ausschalten, da der Text nicht davon betroffen sein soll
glDisable(GL_LIGHTING);
glColor3f(1.0, 1.0, 1.0);
[_font drawTextToScreen:[NSString stringWithFormat:@"Source Position: X:%f      Y:%f      Z:%f", _meshPosition.x, _meshPosition.y, _meshPosition.z]
                      screenSizeX:bounds.size.width
                      screenSizeY:bounds.size.height
                      onPositionX:10
                      onPositionY:bounds.size.height-20];
glEnable(GL_LIGHTING);

```

Sie merken (bzw. hören), dass sich der Sound von einem Lautsprecher zum anderen bewegt, was durch die Funktion

```

alSource3f(_source, AL_POSITION, _meshPosition.x, _meshPosition.y, _meshPosition.z);

```

erreicht wird.

Durch dieses Panning kann man ganz interessante Effekte erstellen. Es klingt natürlich sehr viel besser, wenn das Gurren genau aus der Richtung kommt, in welcher sich auch das »Monster« befindet.

Mehrere Sounds

Möchte man mehrere Sounds gleichzeitig abspielen, wiederholt man einfach die einzelnen Schritte (bis auf die Initialisierung). Wir werden uns später zwei Hilfsklassen dafür bauen, die uns den Großteil der Arbeit abnehmen sollen.

TIPP

Vermeiden Sie es, wenn es geht, zur Laufzeit Sound nachzuladen, da dies je nach Größe der Dateien zu Aussetzern im Programm führen kann. Besser ist es, alle benötigten Sounds gleich zu Anfang zu laden.

Dopplereffekt

Bei diesem Effekt verändert sich die Frequenz eines Sounds im Verhältnis zum Abstand der Kamera. Das bedeutet: Je näher ein Objekt auf uns zukommt, umso höher wird die Frequenz des Sounds, und je weiter es sich von uns weg bewegt, desto tiefer wird sie wieder.

Ein gutes Beispiel wäre ein vorbeifahrender Zug.

Auch OpenAL beherrscht diesen Effekt, der sich über die Funktionen

```
void alDopplerFactor(ALfloat dopplerFactor);  
void alSpeedOfSound(ALfloat speed);
```

steuern lässt.

Über den Dopplerfaktor lässt sich der Effekt an sich beeinflussen, (normal ist 1.0). Höhere Werte führen zu einer Übertreibung des Effekts.

Mit der zweiten Funktion (Speed) stellt man die Geschwindigkeit ein (Standard 343.3 = Schallgeschwindigkeit).

GRUNDLAGEN

Der Effekt wird global ein- bzw. ausgeschaltet, das bedeutet, dass alle Sounds, die gerade abgespielt werden, davon betroffen sind.

Probleme mit dem Dopplereffekt

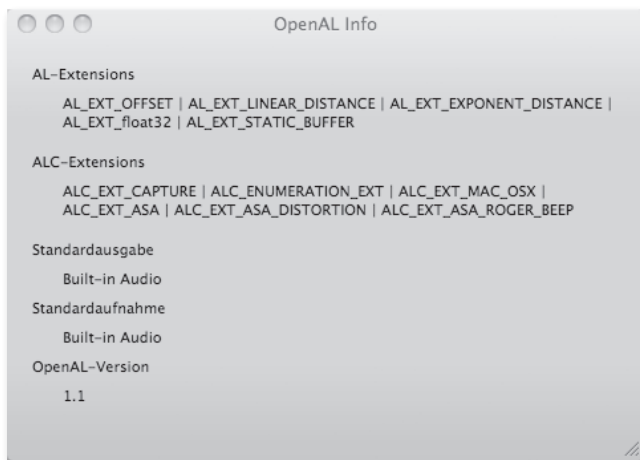
Damit der Effekt auch funktioniert, muss man logischerweise für die Geschwindigkeit (Velocity) von »Source« oder »Listener« einen Wert größer Null angeben. Was zum Schluss der beste Wert ist, kann man nicht pauschal sagen, hier hilft es, ein wenig mit den Werten für Velocity und Speed zu experimentieren.

Aus bestimmten Beiträgen von Foren weiß ich, dass der Dopplereffekt erst seit der OpenAL-Version 1.1 richtig funktioniert. Wenn Sie Leopard installiert haben, sollten Sie bereits die aktuelle Version haben. Wie man die OpenAL-Version abfragen kann, werden wir uns gleich anschauen.

Auch die *alSpeedOfSound(...)* steht erst seit der Version 1.1 zur Verfügung. Vorher hieß die Funktion *alDopplerVelocity(...)*.

OpenAL abfragen

Das folgende kleine Beispielprojekt »Kapitel 13/OpenAL Infos« zeigt, wie man bestimmte Informationen (Device, Version usw.) abfragen kann.



Informationen zu OpenAL

Diese Informationen holt man sich über die beiden Funktionen *alGetString(...)* bzw. *alcGetString(...)*, was in unserem Beispiel folgendermaßen aussieht:

```
// OpenAL initialisieren
LCdevice *newDevice = alcOpenDevice(NULL);
ALCcontext *newContext = alcCreateContext(newDevice, 0);

alcMakeContextCurrent(newContext);

if(alGetError() != AL_NO_ERROR)
    NSLog(@"Fehler beim initialisieren von OpenAL");

// Extensions
if (alGetString(AL_EXTENSIONS))
```

```

        [_extensions setStringValue:[NSString stringWithUTF8String:
        (const char *) alcGetString(AL_EXTENSIONS)]];

// ALC-Extensions
if ( alcGetString(alcGetContextsDevice(alcGetCurrentContext()),
ALC_EXTENSIONS))
[_alcExtensions setStringValue:[NSString stringWithUTF8String:
(const char *) alcGetString(alcGetContextsDevice(alcGetCurrent-
Context()), ALC_EXTENSIONS)]];

// Standard Ausgabe
if (alcGetString(NULL, ALC_DEFAULT_DEVICE_SPECIFIER))
[_defaultOutput setStringValue:[NSString stringWith-
UTF8String: (const char *) alcGetString(NULL,
ALC_DEFAULT_DEVICE_SPECIFIER)]];

// Standard Aufnahme
if (alcGetString(NULL, ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER))
[_defaultInput setStringValue:[NSString stringWith-
UTF8String: (const char *) alcGetString(NULL,
ALC_CAPTURE_DEFAULT_DEVICE_SPECIFIER)]];

// Version
if (alcGetString(AL_VERSION))
[_version setStringValue:[NSString stringWithUTF8String: (const
char *) alcGetString(AL_VERSION)]];

// Kontext und Device wieder freigeben
ALCcontext *context = alcGetCurrentContext();
ALCdevice *device = alcGetContextsDevice(context);

alcDestroyContext(context);
alcCloseDevice(device);

```

Wichtig ist wieder, dass zuerst OpenAL initialisiert wird, da sonst nicht alle Informationen abgefragt werden können. Zum Schluss sollten auch wieder der Kontext und die Schnittstelle freigegeben werden.

OpenAL Extensions

Wie Sie am Screenshot 12.13 sehen können, gibt es auch hier Extensions, welche wir uns ein wenig genauer anschauen wollen.

Um die Erweiterungen zu nutzen, muss man wie bei OpenGL auch zuerst einmal abfragen, ob diese verfügbar sind:

```
if (alcIsExtensionPresent( NULL, "ALC_EXT_ASA" ))
```

Die Funktion erwartet als ersten Parameter die Schnittstelle (NULL ist die Standardschnittstelle) und als zweiten den Namen der Erweiterung.

Nachdem geklärt ist, ob die Erweiterungen verfügbar sind, kann man sie nutzen, wobei hier im Gegensatz zu OpenGL ein wenig mehr Arbeit nötig ist.

Zunächst einmal muss man sich die Adresse der Funktion (der Extension) holen, welche man nutzen möchte:

```
alcGetProcAddress(NULL, (const ALCchar*)
"alcMacOSXRenderingQuality")
```

Auch hier ist der erste Parameter wieder unsere Schnittstelle und der zweite der Name der Erweiterung.

Die Funktion liefert im Erfolgsfall die Adresse der Funktion, welche wir benötigen. Sollte ein Fehler auftreten, kann man diesen mit *alcGetError()* abfragen.

Zum Schluss übergibt man der betreffenden Funktion den gewünschten Parameter. Hier einmal der komplette Vorgang, der nötig ist, um die räumliche Klangverbesserung (*ALC_MAC_OSX_SPATIAL_RENDERING_QUALITY_HIGH*) einzuschalten:

```
static alcMacOSXRenderingQualityProcPtr p = NULL;
if (p == NULL)
{
    p = (alcMacOSXRenderingQualityProcPtr) alcGetProcAddress(NULL,
(const ALCchar*) "alcMacOSXRenderingQuality");
}
if (p)
    p(ALC_MAC_OSX_SPATIAL_RENDERING_QUALITY_HIGH);
```

Alle zur Verfügung stehenden Erweiterungen bzw. die Funktionen, welche benötigt werden, stehen in der Datei »MacOSX_OALExtensions.h«, welche sich im »OpenAL.framework« befindet.

TIPP

Vermeiden Sie es, Stereo-Sounds zu nutzen, zum einen benötigen diese den doppelten Speicherplatz, und zum anderen sind Effekte wie Doppler, Hall usw. nur mit Mono-Sounds möglich.

Ein Stereo-Sound ist in der Regel nicht sinnvoll, da die Soundausgabe sowieso auf beiden Lautsprechern zu hören ist. Anders sieht es aus, wenn Sie ein Surround-System an Ihrem Mac angeschlossen haben, was aber nur sehr selten der Fall sein dürfte.

Beispiel OpenAL-Extensions

Zu den Erweiterungen habe ich noch ein kleines Beispielprojekt erstellt »Kapitel 13/OpenAL Extensions«, welches nochmals zeigt, wie man die Erweiterungen über ihre Funktionen anspricht. Das Beispiel ist auf das Nötigste reduziert, damit Sie sofort sehen können, wie es funktioniert.

Sehr interessant finde ich den Halleffekt, man könnte diesen z. B. in dem Moment einschalten, in dem der Protagonist in eine große Halle eintritt. Der Vorteil dabei ist, dass man die Sounds nicht doppelt erstellen muss (mit und ohne Hall).

Ich denke, Sie werden schon die nötigen Einsatzgebiete für die Effekte finden.

Im Examples-Ordner zu Xcode (»Examples/CoreAudio/Services/OpenAL-Example«) finden Sie ein OpenAL-Beispiel, das nochmals sehr gut die Fähigkeiten von OpenAL (inkl. Extensions) zeigt. Die Oberfläche ist zwar ein wenig unübersichtlich, aber darauf kommt es in dem Beispiel auch nicht an.

CFXOpenAL

Zur einfachen Nutzung von OpenAL habe ich zwei Hilfsklassen erstellt. Die Klasse CFXSoundObject kapselt OpenAL und stellt das eigentliche Soundobjekt dar (inkl. Source und Listener).

Die Klasse CFXSoundManager verwaltet die Soundobjekte und funktioniert im Prinzip genauso wie die Klasse CFXTextureManager.

Hier ein kleines Beispiel, welches einen Sound lädt und geloopt abspielt:

```
_soundManager = [CFXSoundManager sharedManager];
_sound = [_soundManager soundObjectByName:[NSBundle mainBundle]
pathForResource:@"Footsteps" ofType:@"wav"]];
[_sound setSourceLoop:YES];
[_sound playSound];
```

Die Hauptarbeit liegt in der Klasse CFXSoundObject, welche die wichtigsten Eigenschaften für Source und Listener beinhaltet. Ich will hier noch kurz die einzelnen Variablen besprechen, die alle über Setter- und Getter-Methoden verfügbar sind:

```
NSString *_soundName;
```

Der Pfad zur Sounddatei. Dieser wird dem CFXSoundManager übergeben, welcher dann im Erfolgsfall ein CFXSoundObject zurückliefert.

```
ALuint _buffer;
```

Der Puffer, der die Sounddatei enthält.

```
ALuint _source;
```

Die Soundquelle

```
// Source
```

```
CFXVector _sourcePosition;
```

Position des Sounds (Vektor)

```
CFXVector _sourceVelocity;
```

Geschwindigkeit des Sounds (Vektor)

```
CFXVector _sourceDirection;
```

Richtung, in welche die Quelle zeigt (Vektor). Vergleichbar mit der Aufstellrichtung eines Lautsprechers.

```
float _sourcePitch;
```

Höhe des Sounds

```
float _sourceGain;
```

Lautstärke des Sounds

```
BOOL _sourceLoop;
```

Sound soll geloopt abgespielt werden

```
// Listener
CFXVector _listenerPosition;
Kameraposition (Vektor)

CFXVector _listenerVelocity;
Kamerageschwindigkeit (Vektor)

CFXVector _listenerOrientationAt;
Kamerablickrichtung (Vektor)

CFXVector _listenerOrientationUp;
Kameraorientierung up-Vektor (Vektor).
```

Die Fähigkeit, Erweiterungen zu nutzen, habe ich in den beiden Klassen nicht berücksichtigt, es sollte aber keine Schwierigkeit sein, diese nachträglich einzubauen.

Alternativer Wav-Loader

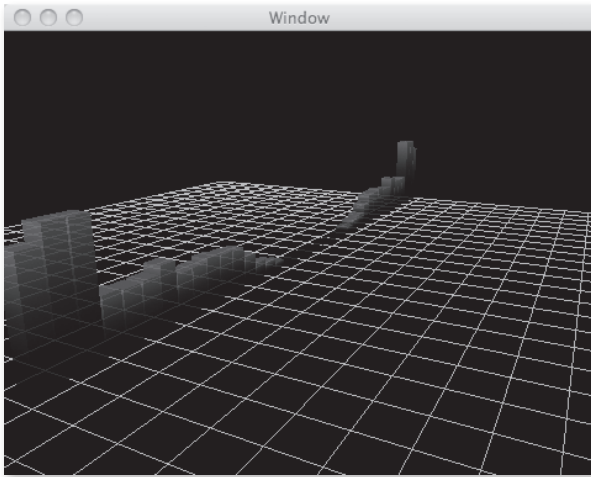
Die beiden Funktionen *alutLoadWAVFile* und *alutUnloadWAV* sind ab Mac OSX 10.5 deprecated. Da das Konzept und große Teile der Beispiele aber noch unter Tiger entstanden sind, habe ich sie trotzdem benutzt (sie funktionieren ja nach wie vor). Mit dem Beispiel »Kapitel 13/OpenAL WavLoader« habe ich ein Objekt erstellt, welches ohne die besagten Funktionen auskommt.

Die Handhabung des Loaders ist recht einfach. Geladen wird ein Sound über *loadMemoryFromFile(...)* und freigegeben wird er über *unloadMemory(...)*.

Musik mit Quicktime abspielen

Für die Wiedergabe von kompletten Musikstücken eignet sich das Wav-Format nicht sonderlich gut, da die Dateien ziemlich groß wären. Hier bietet es sich an, die Musik z. B. als MP3-Datei in das Spiel zu integrieren. Da OpenAL wie gesagt nicht direkt MP3-Dateien abspielen kann, werden wir Quicktime dafür benutzen.

Das Laden und Abspielen einer MP3-Datei mit Hilfe von Quicktime ist keine große Herausforderung, weshalb ich im nächsten Beispiel ein wenig in die Trickkiste gegriffen habe. Das Beispielprojekt »Kapitel 13/Quicktime« visualisiert die MP3-Datei in Form von Balken, so wie man es z. B. von iTunes her kennt.



Schöner als Excel: die Visualisierung von Musik

Das direkte Auslesen und Visualisieren des Audio-Streams ist keine einfache Sache, da man sich in der Regel mit komplexen Algorithmen (Fast Fourier Transform – FFT) herumschlagen muss. Glücklicherweise hat die Quicktime-API einige nützliche Funktionen, welche uns die ganze Arbeit abnehmen.

Das oben genannte Beispielprojekt nutzt die beiden Funktionen:

```
SetMovieAudioFrequencyMeteringNumBands(...)
```

und

```
GetMovieAudioFrequencyLevels(...)
```

die sich um das Auslesen der Audiodaten kümmern.

Gehen wir den wichtigsten Code einmal durch.

```
// Musik laden
NSBundle * b = [NSBundle mainBundle];
_music = [[QTMovie alloc] initWithFile:[b pathForResource:@"backgroundmusic.m4a" error:nil]];
if(!_music)
{
    [_music release];
    _music = nil;
}
```

```

}
[_music setAttribute:[NSNumber numberWithInt:YES]
forKey:QTMovieLoopsAttribute];
[_music play];

```

Im ersten Schritt laden wir unsere Musik-Datei und sagen auch gleich, dass wir die Datei geloopt abspielen möchten.

```

// Bends erstellen
_numberOfBandLevels = 20;    // Anzahl der Bands
_numberOfChannels = 2;      // Anzahl der Kanäle
_freqResults = NULL;
OSStatus err = SetMovieAudioFrequencyMeteringNumBands([_music
quickTimeMovie], kQTAudioMeter_StereoMix, &_numberOfBandLevels);
if(err)
{
    NSLog(@"Error SetMovieAudioFrequencyMeteringNumBands");
}
else
{
    _freqResults = malloc(sizeof(QTAudioFrequencyLevels, level
[_numberOfBandLevels * _numberOfChannels]));
    if (_freqResults == NULL)
    {
        err = memFullErr;
    }
    _freqResults->numChannels = _numberOfChannels;
    _freqResults->numFrequencyBands = _numberOfBandLevels;
}

```

Im nächsten Schritt konfigurieren wir die grafische Ausgabe.

- *numberOfBandLevels* ist die Anzahl der Balken.
- *numberOfChannels* ist die Anzahl der Kanäle, in unserem Fall 2 für Stereo,

Während des Rendervorgangs werden dann die Werte für die Visualisierung ausgelesen. Anhand dieser Werte, die *GetMovieAudioFrequencyLevels(...)* liefert, wird dann die Höhe der einzelnen Balken berechnet.

```

if (_freqResults != NULL)
{

```

```

OSStatus err = GetMovieAudioFrequencyLevels([_music quickTime-
Movie], kQTAudioMeter_StereoMix, _freqResults);
if (err)
{
    NSLog(@"Error GetMovieAudioFrequencyLevels");
}
else
{
    // Linker Kanal
    for (j = 0; j < _freqResults->numFrequencyBands; j++)
    {
        Float32 value = _freqResults->level[(_freqResults-
>numFrequencyBands) + j];
        value*=12.0;
        glPushMatrix();
        glTranslatef(position, 0.0, 0.0);
        [self renderCube:value];
        glPopMatrix();
        position+=offset;
    }

    // Rechter Kanal
    for (j=_freqResults->numFrequencyBands; j>=0; j--)
    {
        Float32 value = _freqResults->level[(_freqResults-
>numFrequencyBands) + j];
        value*=12.0;
        glPushMatrix();
        glTranslatef(position, 0.0, 0.0);
        [self renderCube:value];
        glPopMatrix();
        position+=offset;
    }
}
}

```

Innerhalb des Fragment-Shaders werden dann die Farben der Balken anhand ihres Y-Wertes berechnet. Der Alpha-Wert wird ein wenig reduziert, damit die Balken leicht transparent sind. Der Rest des Shader-Codes macht die Per-Pixel-Beleuchtung, die Sie ja schon kennen.

```
// Farbwert anhand der Y-Koordinate des Vertex berechnen
vec4 tmp= vec4( ypos-0.7, ypos-0.5, ypos-0.2, 1.0);

// Mit Licht / Material multiplizieren (modulate)
gl_FragColor =tmp*color;

// Alphawert
gl_FragColor.a = 0.9;
```

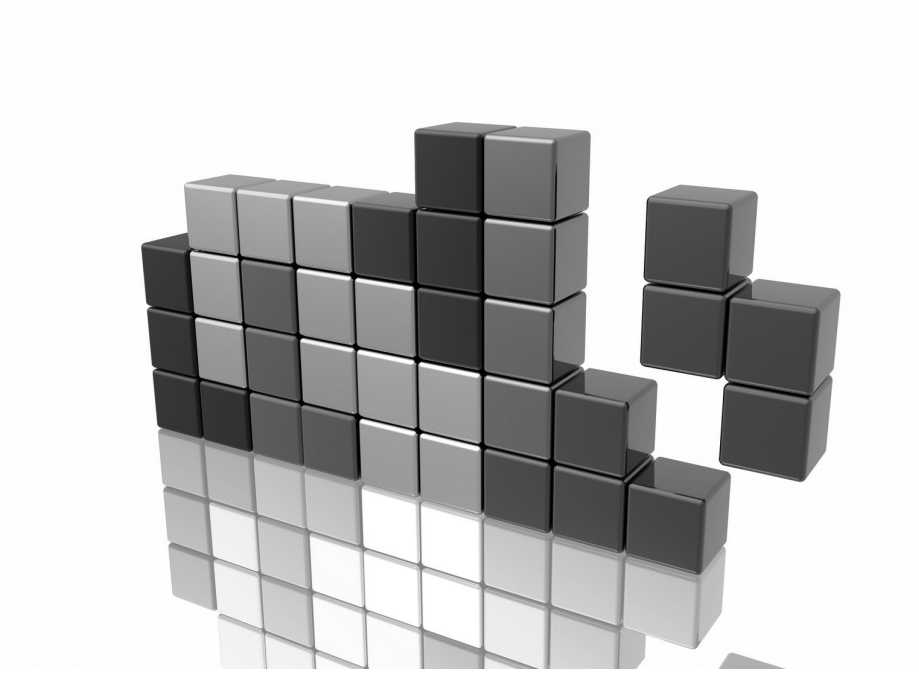
Wie Sie sehen, ist der Aufwand ziemlich gering, wobei sich das Ergebnis meiner Meinung nach sehen lassen kann. Nun steht es Ihnen frei, eigene Visualisierer zu schreiben, alles, was Sie tun müssen, ist, mit den Werten, die *GetMovieAudioFrequencyLevels(...)* liefert, etwas Interessantes auf den Schirm zu zaubern.

Zusätzliche Informationen

OpenAL Webseite
<http://www.openal.org/>

FFT - Fast Fourier Transform
http://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation

Ogg-Vorbis
Eine Sound-Bibliothek, die eine gute Alternative zum MP3-Format bietet.
<http://www.vorbis.com/>

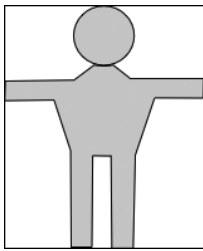


Kollisionserkennung

Die Erkennung und Behandlung einer Kollision zwischen zwei Objekten ist ein elementarer Bestandteil in der Spiele-Entwicklung. Zum Beispiel wird sie dazu benutzt, um den Spieler innerhalb der Spielwelt zu halten, oder aber, wenn man feststellen will, ob ein Schuss den Gegner getroffen hat. Die Berechnungen einer Kollision sind zum Teil sehr aufwendig, da man im Extremfall testen muss, ob sich nur ein einziges Polygon des Spielers mit einem des Gegners berührt hat. Wir werden uns in diesem Kapitel einige dieser Kollisionstests anschauen.

Bounding Box

Wir haben diese Art von Kollisionsprüfung zwar schon einmal im Kapitel 3 besprochen, ich möchte sie aber nochmals kurz aufgreifen. Bei der Bounding-Box wird eine Box um das Objekt definiert, welches dieses gerade so umschließt.



Bounding-Box um ein Mesh

Um die Koordinaten der Bounding-Box zu berechnen, durchläuft man alle Eckpunkte der Geometrie und speichert jeweils die kleinsten und größten XYZ-Werte. Aus diesen wird dann die Bounding-Box definiert.

```
// Liefert die Axis Aligned BBox (AABB) anhand der uebergebenen
// Vertices
static inline CFXBBox AABBBox(const CFXVector *vertices, const int
numVertices)
{
    CFXBBox bbox;
    bbox.min = makeEmptyVector();
    bbox.max = makeEmptyVector();

    if(vertices)
    {
        int i;
        for(i=0; i<numVertices; i++)
        {
```



```

// BBox maximum
if(vertices[i].x > bbox.max.x)
    bbox.max.x = vertices[i].x;

if(vertices[i].y > bbox.max.y)
    bbox.max.y = vertices[i].y;

if(vertices[i].z > bbox.max.z)
    bbox.max.z = vertices[i].z;

// BBox minimum
if(vertices[i].x < bbox.min.x)
    bbox.min.x = vertices[i].x;

if(vertices[i].y < bbox.min.y)
    bbox.min.y = vertices[i].y;

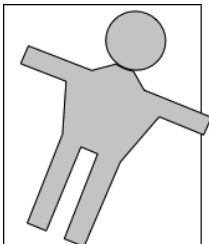
if(vertices[i].z < bbox.min.z)
    bbox.min.z = vertices[i].z;
    }
}
return bbox;
}

```

Diese Funktion liefert die Axis-Aligned-Bounding-Box (AABB) eines Objektes.

AABB

Bei einer AABB laufen die Achsen der Box parallel zu den Achsen des Raumes. Das bedeutet, dass selbst wenn das Objekt rotiert wird, die BBox immer gleich ausgerichtet bleibt.

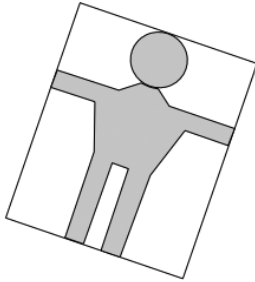


Selbst bei einer Rotation des Objektes bleibt die Bounding-Box gleich.

Wie man auf obiger Abbildung erkennen kann, wäre eine ordentliche Kollisionsprüfung, nachdem das Objekt rotiert wurde, nicht mehr möglich. Dieses Problem umgeht man mit einer sogenannten Oriented-Bounding-Box (OBB).

OBB

Bei dieser Form wird die Bounding-Box immer dann neu berechnet, wenn das Objekt rotiert wurde. Die OBB hat aber den Nachteil, dass die Kollisionsprüfung nicht ganz so einfach ist wie bei einer ABB.



Eine OBB umschließt das Objekt auch, nachdem es rotiert wurde.

Um es einfach zu halten, werden wir uns nur die AABB genauer ansehen.

Kollisionstest

Wie man eine AABB definiert, haben wir oben im Code schon gesehen. Bleibt noch die Frage, wie man auf eine Kollision prüft.

Eine Kollision findet dann statt, wenn sich mindestens eine Seite der beiden Boxen berührt. Bei Objekten, die sich innerhalb eines Spiels bewegen (nicht rotieren oder skalieren), ist es nötig, die Koordinaten der beiden Bounding-Boxen bei jedem Renderdurchlauf zu aktualisieren.

Wir nehmen einmal ein Beispiel, in welchem sich beide Objekte im Spiel bewegen:

```
CFXBBBox tmpBBBox1;
tmpBBBox1.min.x = _position1.x+_bbox1.min.x;
tmpBBBox1.min.y = _position1.y+_bbox1.min.y;
tmpBBBox1.min.z = _position1.z+_bbox1.min.z;

tmpBBBox1.max.x = _position1.x+_bbox1.max.x;
tmpBBBox1.max.y = _position1.y+_bbox1.max.y;
tmpBBBox1.max.z = _position1.z+_bbox1.max.z;

CFXBBBox tmpBBBox2;
tmpBBBox2.min.x = _position1.x+_bbox2.min.x;
tmpBBBox2.min.y = _position1.y+_bbox2.min.y;
tmpBBBox2.min.z = _position1.z+_bbox2.min.z;

tmpBBBox2.max.x = _position1.x+_bbox2.max.x;
```

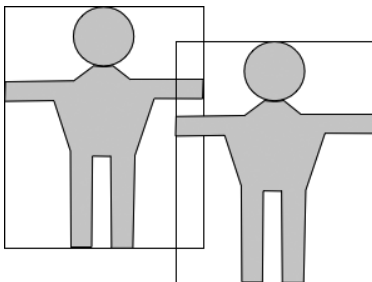
```
tmpBBox2.max.y = _position1.y+_bbox2.max.y;  
tmpBBox2.max.z = _position1.z+_bbox2.max.z;
```

Wir berechnen hier ausgehend von der Position eines Objektes die Ausdehnung der AABB. Dazu nehmen wir die Position (Mittelpunkt) des Objektes und addieren jeweils den kleinsten und größten Wert dazu und berechnen somit die Lage der Bounding-Box im Raum.

Nachdem nun die beiden Bounding-Boxen berechnet sind, können wir auf eine Kollision testen, was so aussieht:

```
// Prüft auf eine Kollision zwischen 2 AABB's  
static inline bool collisionBetweenTwoAABBs (const CFXBBox bbox1,  
const CFXBBox bbox2)  
{  
    if(bbox1.min.x > bbox2.max.x ||  
        bbox1.min.y > bbox2.max.y ||  
        bbox1.min.z > bbox2.max.z ||  
        bbox1.max.x < bbox2.min.x ||  
        bbox1.max.y < bbox2.min.y ||  
        bbox1.max.z < bbox2.min.z )  
    {  
        return false;  
    }  
  
    return true;  
}
```

Wir prüfen also, ob sich eine der Seiten der beiden Boxen überschneidet, wenn dem so ist, haben wir eine Kollision.



Kollision beider Boxen

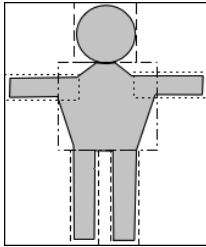
An der Abbildung kann man die Überschneidung der beiden Boxen in der Mitte erkennen, was eine Kollision auslösen würde. Wie man unschwer sehen kann, ist

diese Prüfung nicht sonderlich akkurat, reicht aber in vielen Fällen aus. Zum Beispiel wird man bei einem schnellen Weltraum-Shooter nicht bemerken, dass sich nur die Boxen berührt haben, nicht aber die Objekte selbst.

Wenn eine genauere Prüfung stattfinden muss, geht man in der Regel so vor, dass man zunächst die Bounding-Boxen (bzw. Hitboxen) testet, wenn dort eine Kollision vorliegt, geht man einen Schritt weiter und testet die einzelnen Polygone (Per-Polygon-Test) so lange, bis man diejenigen gefunden hat, welche sich berühren.

Hitbox

Eine weitere Möglichkeit für die Kollisionsprüfung mit Bounding-Boxen besteht darin, für ein Objekt mehrere Boxen (Hitboxes) zu definieren. Das heißt, man hat eine Box, die das komplette Model umschließt, und dann jeweils eine für z. B. Arme, Beine, Körper, Kopf usw.



Hitboxes für ein Model

Die Kollisionsprüfung wird dann pro Box durchgeführt, dabei beginnt man mit der äußeren, die das Model komplett umschließt, und arbeitet sich dann Schritt für Schritt durch alle anderen (Arme, Beine usw.). Diese Methode hat den Vorteil, dass sie genauer ist als eine einfache AABB bzw. OBB und allemal schneller als eine Per-Polygon-Prüfung. Um diese Hitboxes zu definieren, ist es natürlich nötig, dass man seine 3D-Models in mehrere Teilobjekte unterteilt, da man sonst die einzelnen Hitboxes für die Körperabschnitte nicht mehr definieren kann.

Beispiel AABB-AABB-Kollision

So, nun aber ein praktisches Beispiel. Das Projekt »Kapitel 14/AABB Collision« prüft auf eine Kollision zwischen zwei Models, welche aus einem Wavefront-Mesh geladen wurden.

Die Klasse CFXWavefrontMesh habe ich dafür ein wenig modifiziert, und zwar habe ich zwei Getter-Methoden eingebaut. Die eine davon liefert einen Zeiger auf die Vertices und die andere die Anzahl der Vertices.

Diese Vertices werden dann der Funktion `calcAABBBox(...)` übergeben, welche die Bounding-Box berechnet. Nach der Berechnung werden die Vertices in der Klas-

se CFXWavefrontMesh wieder gelöscht (*freeMemory*), da sie nicht weiter benötigt werden.

```
// Model 1 Cone
_mesh1 = [[CFXWavefrontMesh alloc]init];
[_mesh1 loadMesh:@"cone"];
// BBox berechnen
_bbox1 = calcAABBox([_mesh1 vertices], [_mesh1 numVertices]);
// Model positionieren
_position1 = makeVector(-1.0, 0.0, 0.0);
// Vertices wieder loeschen
[_mesh1 freeMemory];

// Model 2 Ball
_mesh2 = [[CFXWavefrontMesh alloc]init];
[_mesh2 loadMesh:@"ball"];
// BBox berechnen
_bbox2 = calcAABBox([_mesh2 vertices], [_mesh2 numVertices]);
// Model positionieren
_position2 = makeVector(1.0, 0.0, 0.0);
// Vertices wieder loeschen
[_mesh2 freeMemory];
```

Da sich das zweite Model nicht bewegt, kann seine Bounding-Box bei der Initialisierung berechnet werden.

```
_tmpBBox2.min.x = _position2.x+_bbox2.min.x;
_tmpBBox2.min.y = _position2.x+_bbox2.min.y;
_tmpBBox2.min.z = _position2.x+_bbox2.min.z;

_tmpBBox2.max.x = _position2.x+_bbox2.max.x;
_tmpBBox2.max.y = _position2.x+_bbox2.max.y;
_tmpBBox2.max.z = _position2.x+_bbox2.max.z;
```

Während des Rendervorgangs wird die Bounding-Box des ersten Models aktualisiert und auf eine Kollision mit der anderen Bounding-Box geprüft. Die beiden Bounding-Boxen werden zur Ansicht mitgezeichnet, den Code dazu lassen wir außen vor.

.

.

.

```
_position1.x+=0.3*_deltaTime;

CFXBBox tmpBBox1;
tmpBBox1.min.x = _position1.x+_bbox1.min.x;
tmpBBox1.min.y = _position1.y+_bbox1.min.y;
tmpBBox1.min.z = _position1.z+_bbox1.min.z;

tmpBBox1.max.x = _position1.x+_bbox1.max.x;
tmpBBox1.max.y = _position1.y+_bbox1.max.y;
tmpBBox1.max.z = _position1.z+_bbox1.max.z;

_collision = collisionBetweenTwoAABBs (tmpBBox1, _tmpBBox2);

glPushMatrix();
glTranslatef(_position1.x, _position1.y, _position1.z);
    [_mesh1 renderMesh];
    [self renderBBox:_bbox1];
glPopMatrix();

glPushMatrix();
glTranslatef(_position2.x, _position2.y, _position2.z);
    [_mesh2 renderMesh];
    [self renderBBox:_bbox2];
glPopMatrix();
.
.
.
```

Wenn es zu einer Kollision kommt, werden die beiden Boxen rot gezeichnet, ansonsten weiß.

Bounding-Sphere

Wenn eine Bounding-Box nur unzureichend die Form des Objekts umschließt (z. B. der Ball bei einer Fußballsimulation), kann eine Bounding-Sphere oft besser sein.



Bounding-Sphere um ein Model

Bei diesem Verfahren wird, wie der Name schon sagt, eine unsichtbare Kugel um das Objekt definiert. Der Radius der Kugel ist genauso groß, wie der Eckpunkt, welcher am weitesten vom Mittelpunkt des Objektes entfernt ist. Die Berechnung des Radius sieht folgendermaßen aus:

```
// Liefert den Radius einer Bounding-Sphere anhand der
// uebergebenen Vertices und dem Mittelpunkt centerPoint
static inline float boundingSphereRadius(const CFXVector
*vertices, const int numVertices, const CFXVector centerPoint)
{
    float currentDistance = 0.0;
    float maximumDistance = 0.0;

    if(vertices)
    {
        int i;
        for(i=0; i<numVertices; i++)
        {
            currentDistance = ( (vertices[i].x-centerPoint.x)
            *(vertices[i].x-centerPoint.x) )+
            ( (vertices[i].y-centerPoint.y)*
            (vertices[i].y-centerPoint.y) )+
            ( (vertices[i].z-centerPoint.z)*
            (vertices[i].z-centerPoint.z) );

            // maximale Distanz berechnen
            if(currentDistance > maximumDistance)
                maximumDistance = currentDistance;
        }
    }
}
```

```

    }

    return sqrt(maximumDistance);
}

```

Sie sehen: Der Code ähnelt ein wenig dem der Bounding-Box. Dabei wird jeder Eckpunkt durchlaufen und geprüft, wie weit er vom Mittelpunkt (*currentDistance*) entfernt ist. Der Eckpunkt, der am weitesten davon entfernt ist, ist dann der Wert (*maximumDistance*) für den Radius.

Sphere-Sphere-Kollision

Eine Kollision zwischen zwei Bounding-Spheres findet dann statt, wenn die Summe der beiden Radien kleiner bzw. gleich dem Abstand des Mittelpunktes (Position) der beiden ist.

```

// Prüft auf eine Kollision zwischen 2 Bounding-Spheres
static inline bool collisionBetweenTwoSpheres (const CFXVector
position1, const CFXVector position2, const float radius1, const
float radius2)
{
    CFXVector relPos = vectorSubtract(position1, position2);
    float dist = relPos.x * relPos.x + relPos.y * relPos.y + rel-
Pos.z * relPos.z;
    float minDist = radius1+radius2;
    return dist <= minDist * minDist;
}

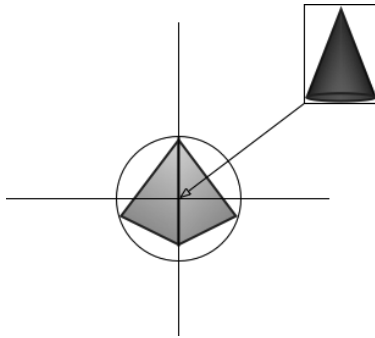
```

Ein Vorteil von Bounding-Spheres ist, dass man die Models beliebig rotieren kann, wobei die Kollisionsabfrage danach natürlich immer noch funktioniert.

Auch zu den Bounding-Spheres habe ich ein kleines Projekt angehängt, »Kapitel 14/Sphere Sphere Collision«. Der Code ist fast identisch zum vorherigen Beispiel, so dass wir ihn nicht extra besprechen müssen.

Sphere-AABB-Kollision

Die Kollision zwischen einer Bounding-Sphere und einer AABB lässt sich auch sehr einfach berechnen. Dazu prüfen wir zuerst, welcher Eckpunkt der AABB dem Mittelpunkt (Position) der Bounding-Sphere am nächsten liegt. Wenn dieser gefunden ist, wird geprüft, ob die Distanz dieses Eckpunktes zum Mittelpunkt der Bounding-Sphere kleiner ist als der Radius der Bounding-Sphere. Ist er das, dann liegt eine Kollision vor.



Kollision zwischen einer AABB und einer Bounding-Sphere

Die Funktion dazu sieht so aus:

```
// Prueft auf eine Kollision zwischen einer AABB und einer
// Bounding-Sphere
// position ist die Position der Bounding-Sphere
static inline bool collisionBetweenAABBAndSphere( const CFXBBox
bbox, const float radius, const CFXVector position)
{
    // Suche den Punkt der BBox der am naehsten an position liegt
    CFXVector r;
    if(position.x < bbox.min.x)
    {
        r.x = bbox.min.x;
    }
    else if(position.x > bbox.max.x)
    {
        r.x = bbox.max.x;
    }
    else
    {
        r.x = position.x;
    }

    if(position.y < bbox.min.y)
    {
        r.y = bbox.min.y;
    }
    else if(position.y > bbox.max.y)
    {
        r.y = bbox.max.y;
    }
}
```

```
    else
    {
        r.y = position.y;
    }

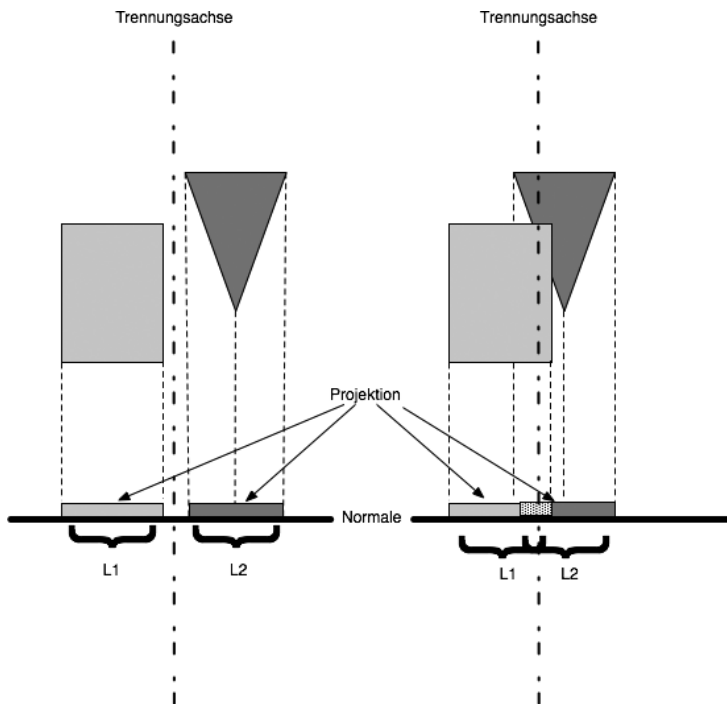
    if(position.z < bbox.min.z)
    {
        r.z = bbox.min.z;
    }
    else if(position.z > bbox.max.z)
    {
        r.z = bbox.max.z;
    }
    else
    {
        r.z = position.z;
    }

    // Distanz berechnen
    float dx = position.x - r.x;
    float dy = position.y - r.y;
    float dz = position.z - r.z;
    float distance = dx*dx + dy*dy + dz*dz;

    return distance < (radius*radius);
}
```

SAT

Für unser Spiel in Kapitel 15 benötigen wir noch eine Kollisionsabfrage, die prüft, ob ein Dreieck eine AABB schneidet. Da diese Art von Test nicht ganz trivial ist, hab ich eine fertige Lösung implementiert, welche auf der SAT (Separating Axis Theorem) basiert. Diese besagt, dass sich 2 Polygone nicht schneiden, wenn es möglich ist, zwischen beiden eine Gerade (Trennungsachse) zu finden, die sie trennt.



Projektion der Polygone auf einen Normal-Vektor. Links ist keine Überschneidung zu sehen. Rechts daneben findet eine Kollision statt (Überschneidung der Polygone bzw. gepunkteter Bereich unten).

Das Ganze funktioniert grob gesagt so: Man projiziert die beiden Polygone, welche man auf eine Kollision testen will, auf die Orthogonale der Geraden. Dabei macht man nichts anderes, als alle Eckpunkte auf eine Ebene »ausbreiten«, wodurch man dann 4 Zahlen (L1, L2) bekommt. Alles, was dann noch zu tun ist, ist zu prüfen, ob sich die Intervalle der beiden (L1, L2) überschneiden. Wenn ja, dann bedeutet das, dass eine Kollision stattgefunden hat.

Wie gesagt habe ich eine fertige Lösung für diese Aufgabe implementiert, welche Sie in der Datei »CFXCollision.h« finden. Dieser Lösungsansatz ist eine modifi-

zierte Version der SAT von Tomas Akenine-Moeller. Diese Version wird in etlichen frei verfügbaren 3D-Engines verwendet, da sie recht schnell ist und nebenbei einfach in ein bestehendes System implementiert werden kann.

Um nun auf eine Kollision zwischen einem oder mehrere Polygon/en und einer AABB zu testen, reicht folgende Zeile:

```
_collision= triBoxOverlap(boxcenter,boxhalfsize,triverts);
```

- *boxcenter* ist die Position des Objektes (z. B. Spieler).
- *boxhalfsize* ist die halbe Größe der AABB, die das Objekt (z. B. Spieler) umschließt.
- *triverts* sind die Polygone des Objektes, mit welchem wir auf eine Kollision testen wollen (z. B. Gegner).

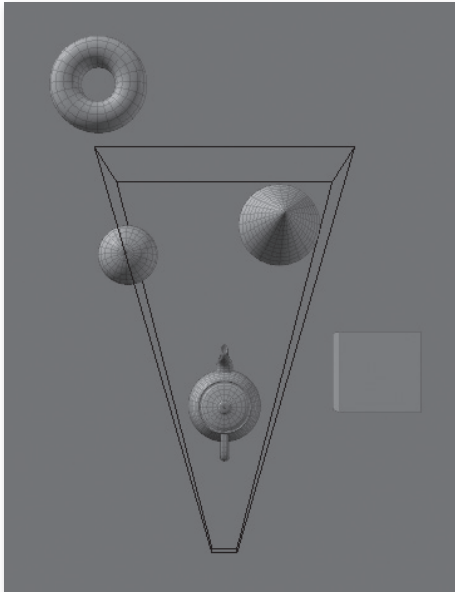
Im Kapitel 15 werden wir unser Spiel entwickeln, dort werden wir nochmals genauer sehen, wie diese Kollisionsabfrage funktioniert.

So, diese Arten von Kollisionstests sollen für unser Spielvorhaben erst einmal genügen. Die Bounding-Boxen bzw. Spheres sind aber noch für andere Dinge nützlich: Z. B. kann man sie dazu benutzen, um den allgemeinen Rendervorgang zu beschleunigen, was wir uns jetzt einmal anschauen wollen.

Frustum Culling

Wie schon mehrfach erwähnt, steht man als Entwickler immer vor dem Problem, die Frameraten des Spiels hoch zu halten, egal, wie viele Polygone gerade auf dem Schirm zu sehen sind. Auch die Verwendung von VBOs bringt irgendwann nichts mehr, wenn mehrere 100.000 Polygone gleichzeitig gerendert werden sollen. Der eigentliche Flaschenhals liegt nämlich immer in der Renderpipeline, das bedeutet, dass alle Polygone (sichtbar oder nicht) zunächst einmal den gleichen Weg durchwandern, bis sie dann durch OpenGL verworfen werden (Clipping, Backface-Culling).

Unsere Aufgabe besteht nun darin, diejenigen Polygone (bzw. Objekte), die nicht sichtbar sind, erst gar nicht durch die Renderpipeline zu schicken, was man »Frustum-Culling« nennt.



Das Sichtfeld der Kamera mit verschiedenen Objekten

In der Abbildung sehen wir ein Sichtfeld (Frustum) mit verschiedenen Objekten darin. Das Frustum ist definiert durch die 6 Flächen rechts, links, oben, unten, Near Clipping-Plane (vorne) und Far Clipping-Plane (hinten).

Man kann erkennen, dass der Würfel rechts und der Torus links oben komplett außerhalb unseres Sichtfeldes sind, wobei alle anderen komplett bzw. teilweise inner-

halb des Sichtfeldes liegen. Alle Objekte, die sich nun außerhalb befinden, sollen erst gar nicht durch die Pipeline geschickt werden (sie werden gecullt).

Das Frustum-Culling erfolgt nun in zwei Schritten. Im ersten Schritt wird das Frustum definiert und im zweiten Schritt werden die Bounding-Boxen bzw. Bounding-Spheres der Objekte gegen das Frustum gecullt.

Frustum extrahieren

Beim Extrahieren des Sichtfeldes müssen wir die 6 Flächen definieren, welche unsere Kamera im Moment »sieht«.

Dazu benötigen wir die Modelviewmatrix und die Projektionsmatrix, welche wir mit der Funktion `glGetFloatv(...)` holen. Diese beiden müssen dann miteinander multipliziert werden.

```
// Hole Projektions-Matrix
float projection[16];
glGetFloatv(GL_PROJECTION_MATRIX, projection);

// Hole Modelview-Matrix
float modelview[16];
glGetFloatv(GL_MODELVIEW_MATRIX, modelview);

// Bevor die 6 Flaechen extrahiert werden koennen
// muessen die beiden Matrizen multipliziert werden
float result[16];
[self multiplyMatrix:modelview b:projection r:result];
```

Danach werden die einzelnen Flächen des Sichtfeldes extrahiert und anschließend normalisiert.

```
// Jede Flaechе extrahieren und normalisieren
// rechts
frustum[0][A] = result[3] - result[0];
frustum[0][B] = result[7] - result[4];
frustum[0][C] = result[11] - result[8];
frustum[0][D] = result[15] - result[12];
[self normalizePlane:frustum[0] result:frustum[0] ];

// links
frustum[1][A] = result[3] + result[0];
```

```
frustum[1][B] = result[7] + result[4];
frustum[1][C] = result[11] + result[8];
frustum[1][D] = result[15] + result[12];
[self normalizePlane:frustum[1] result:frustum[1] ];

// unten
frustum[2][A] = result[3] + result[1];
frustum[2][B] = result[7] + result[5];
frustum[2][C] = result[11] + result[9];
frustum[2][D] = result[15] + result[13];
[self normalizePlane:frustum[2] result:frustum[2] ];

// oben
frustum[3][A] = result[3] - result[1];
frustum[3][B] = result[7] - result[5];
frustum[3][C] = result[11] - result[9];
frustum[3][D] = result[15] - result[13];
[self normalizePlane:frustum[3] result:frustum[3] ];

// hinten
frustum[4][A] = result[3] - result[2];
frustum[4][B] = result[7] - result[6];
frustum[4][C] = result[11] - result[10];
frustum[4][D] = result[15] - result[14];
[self normalizePlane:frustum[4] result:frustum[4] ];

// vorne
frustum[5][A] = result[3] + result[2];
frustum[5][B] = result[7] + result[6];
frustum[5][C] = result[11] + result[10];
frustum[5][D] = result[15] + result[14];
[self normalizePlane:frustum[5] result:frustum[5] ];
```

AUFGEPASST

Wenn im Spiel eine bewegliche Kamera vorgesehen ist, muss das Frustum bei jedem Renderdurchlauf neu berechnet werden, was zu Lasten der Performance geht, da das Normalisieren der Flächen (*normalizePlane*) eine Wurzelberechnung (*sqr*t) beinhaltet.

Nachdem das Frustum berechnet wurde, kann man prüfen, ob sich ein Punkt darin befindet. Die Berechnung dazu beruht auf der Ebenengleichung ($A \cdot x + B \cdot y + C \cdot z + D = 0$).

- $A \ B \ C$ ist Normalvektor der Ebene.
- $x \ y \ z$ sind Koordinaten des Punktes, welcher getestet werden soll.
- D ist die Distanz der Ebene zum Ursprung.

Diese Formel ergibt:

- 0, wenn der Punkt in der Ebene liegt ($D = 0$)
- eine negative Zahl, wenn der Punkt hinter der Ebene liegt
- eine positive Zahl, wenn der Punkt vor der Ebene liegt.

Punkt im Frustum

Die Funktion, die nun prüft, ob ein Punkt im Frustum liegt, sieht folgendermaßen aus:

```
// Prüft ob der Punkt im Frustum liegt
-(BOOL)isPointInFrustum:(CFXVector)p
{
    // Alle Seiten des Frustum durchlaufen
    int i;
    for (i = 0; i < 6; i++)
    {
        // Berechne Distanz
        float dist = frustum[i][A] * p.x +
            frustum[i][B] * p.y +
            frustum[i][C] * p.z +
            frustum[i][D];

        // Wenn Distanz negativ bzw. Null, dann ist der Punkt
        // außerhalb des Frustum
        if (dist <= 0)
            return NO;
    }

    // Der Punkt liegt im Frustum (vor der Ebene)
    return YES;
}
```


AABB im Frustum

Die Funktion, die testet, ob eine AABB im Frustum liegt, sieht folgendermaßen aus:

```
// Prueft ob die AABB im Frustum liegt
-(BOOL) isAABBInFrustum:(CFXBBBox)bbox
{
    int i;
    for (i = 0; i < 6; i++)
    {
        if (frustum[i][A]*(bbox.min.x) +
            frustum[i][B]*(bbox.min.y) + frustum[i][C]*(bbox.min.z) +
            frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.max.x) + frustum[i][B]*(bbox.
            min.y) + frustum[i][C]*(bbox.min.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.min.x) + frustum[i][B]*(bbox.
            max.y) + frustum[i][C]*(bbox.min.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.max.x) + frustum[i][B]*(bbox.
            max.y) + frustum[i][C]*(bbox.min.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.min.x) + frustum[i][B]*(bbox.
            min.y) + frustum[i][C]*(bbox.max.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.max.x) + frustum[i][B]*(bbox.
            min.y) + frustum[i][C]*(bbox.max.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.max.x) + frustum[i][B]*(bbox.
            max.y) + frustum[i][C]*(bbox.max.z) + frustum[i][D]>0)
            continue;
        if (frustum[i][A]*(bbox.max.x) + frustum[i][B]*(bbox.
            max.y) + frustum[i][C]*(bbox.max.z) + frustum[i][D]>0)
            continue;

        return NO;
    }
    return YES;
}
```

Auch diese ist nicht weiter schwierig, alles, was hier passiert, ist, dass alle 8 Seiten gegen jede Seite des Frustums getestet werden. Wenn mindestens ein Punkt im Frustum liegt, gibt die Funktion *YES* zurück.

Sphere in Frustum

Zum Schluss noch die Funktion, welche testet, ob eine Sphere im Frustum liegt.

```
// Prüft ob die Sphere mit der Position (position) und dem
// Radius (radius) im Frustum liegt
-(BOOL)isSphereInFrustum:(CFXVector)position radius:(float)radius
{
    // Alle Seiten des Frustum durchlaufen
    int i;
    for (i = 0; i < 6; i++)
    {
        // Berechnen Distanz
        float dist = frustum[i][A] * position.x +
frustum[i][B] * position.y +
frustum[i][C] * position.z +
frustum[i][D];

        // Wenn Distanz kleiner oder gleich dem Radius, dann ist
        // die Sphere nicht im Frustum
        if (dist <= -radius)
            return NO;
    }

    // Sphere liegt im Frustum
    return YES;
}
```

Diese ist fast identisch mit der *isPointInFrustum*-Methode. Der einzige Unterschied besteht nun darin, dass wir zum Mittelpunkt der Sphere noch den Radius hinzuaddieren müssen.

Beispiel Frustum-Culling

Mit all dem Wissen über das Frustum-Culling werden wir uns das nächste Beispiel »Kapitel 14/Frustum Culling« einmal anschauen.

Im Unterschied zu den anderen Beispielprogrammen läuft dieses im Vollbildmodus. Das Beispiel verlassen Sie mit der ESC-Taste.

AUFGEPASST

Achtung: Der Ladevorgang kann einen Moment dauern, da die 3D-Models zum Teil aus vielen Polygonen bestehen. Asteroiden-Model (über 12.000), alle anderen (über 1.000).

Es werden 50 mehr oder weniger verschiedene Models gezeigt, die alle gegen das Frustum gecullt und dann, wenn sichtbar, gerendert werden. Mit der Leertaste können Sie das Frustum-Culling ein- bzw. ausschalten, wobei Sie den Unterschied sofort an der erzielten Framerate sehen können. Ich habe hier absichtlich Models mit einer hohen Polygonzahl genommen, um den Rechner mal ein wenig ins Schwitzen zu bringen (na ja, meinen zumindest).

Wir werden uns nur die wichtigsten Teile des Codes anschauen.

In der Methode *captureDisplay* habe ich 2 Zeilen auskommentiert:

```
// Bildaufbau mit Swap Intervall synchronisieren
//int    value = 1;
//[_context setValues:&value forParameter:NSOpenGLCPSwapInterval];
```

Wenn Sie diese Kommentare herausnehmen, wird der Rendervorgang mit der Bildwiederholfrequenz Ihres Bildschirms synchronisiert. Es kann nämlich durchaus vorkommen, dass es mit inaktiver Synchronisation zu Fehlern (Schlieren) bei der Darstellung kommt, und zwar deshalb, weil das *flushBuffer()* ausgeführt wird, obwohl der Rendervorgang noch nicht ganz fertig ist.

Das wirklich Wichtige passiert in der *enterMainLoop*-Methode:

```
// Kamera updaten
[_camera updateCamera];

// Frustum extrahieren
[_frustum calculateFrustum];

_numMeshesInFrustum = 0;

int i;
for(i=0; i<NUM_MESHES; i++)
{
    // Rotation erhoehen
    _meshes[i].rotation+=_meshes[i].rotationSpeed*_deltaTime;
```

```

glPushMatrix();

    CFXVector p = _meshes[i].position;
    glTranslatef(p.x, p.y, p.z);
    glRotatef(_meshes[i].rotation, 0.0, 1.0, 0.0);
    CFXWavefrontMesh *m = _meshes[i].mesh;
    if(_useFrustumCulling)
    {
        if(!_frustum.isSphereInFrustum(p.radius*_meshes[i].
            radius))
        {
            _numMeshesInFrustum++;
            [m renderMesh];
        }
    }
    else
    {
        _numMeshesInFrustum++;
        [m renderMesh];
    }
glPopMatrix();
}

```

Nachdem zuerst die Kamera aktualisiert wurde, wird das Frustum neu berechnet. Wie gesagt, muss das sein, da sich die Kameraposition immer ändert. Danach werden alle Objekte durchlaufen und ihre Bounding-Spheres gegen das Frustum gecullt (wenn aktiv). Wenn diese im Frustum liegen, werden die Objekte gerendert, sonst nicht. Der Unterschied bei den erzielten Frameraten ist erstaunlich. Auf meinem doch schon etwas betagten Rechner erhalte ich 15000fps mit aktiven Culling und 35fps ohne, wenn kein Objekt im Sichtbereich liegt. Den Rest des Codes (Kamerabewegung, laden der Models usw.) kennen Sie ja bereits.

Ein angenehmer Nebeneffekt beim Frustum-Culling ist der, dass Sie, wenn Sie nicht gerade ein Echtzeitspiel programmieren wollen, auch die Bewegungen (Kollisionsabfragen der Objekte untereinander) erst dann aktualisieren müssen, wenn die Objekte im Sichtbereich sind. Das spart natürlich eine Menge Rechenzeit, was gerade auf älteren Rechnern ein großes Plus ist.

FPS-Counter

Die Berechnung der fps (Frames per Seconds) macht im obigen Beispiel folgende Methode:

```
-(void)countFPS
{
    static int FPS = 0.0f;
    unsigned long nextSecond = 0.0f;
    static unsigned long prevSecond = 0.0f;

    FPS++;

    nextSecond = TickCount() / 60;

    if(nextSecond - prevSecond > 1.0f)
    {
        prevSecond = nextSecond;
        _fpsCounter+=FPS;
        _counter++;
        _currentFPS = FPS;
        FPS = 0;
    }
}
```

Diese wird bei jedem Renderdurchlauf aufgerufen, dabei werden die Frames (*FPS*) hochgezählt.

Wenn eine Sekunde verstrichen ist, werden die erzielten Frames der Variable *currentFPS* zugewiesen, welche dann auf dem Bildschirm ausgegeben wird.

Occlusion Queries

Auch mit den sogenannten Occlusion-Queries kann man testen, ob ein Objekt sichtbar ist oder nicht. Dabei geht diese Technik sogar noch einen Schritt weiter als das Frustum-Culling. Hierbei kann man nämlich noch prüfen, ob ein Objekt durch ein anderes verdeckt wird.

TIPP

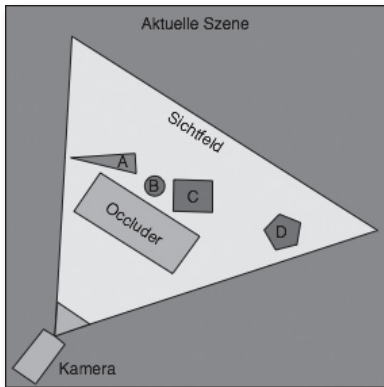
Eine Form des Occlusion-Cullings ist das Ihnen schon bekannte Backface-Culling (bei geschlossenen Objekten). Dort wird die verdeckte Rückseite eines Objektes auch nicht gerendert.

Dadurch eignet sich die Technik besonders für Spiele, in denen eher kleinere Objekte (Spieler, Fahrzeuge, Gegenstände) durch große wie Mauern, Berge usw. verdeckt werden.

Die Vorgehensweise beim Rendern ist folgende:

- Zeichnen Sie zuerst große Objekte, die möglicherweise andere, kleinere Objekte verdecken könnten.
- Zeichnen Sie die Bounding-Box der kleineren Objekte und prüfen sie, ob sie durch die großen Objekte verdeckt werden.
- Wenn sie verdeckt werden, machen Sie nichts, wenn nicht, zeichnen Sie die Objekte.

Die Abbildung soll das nochmals verdeutlichen.



Mehrere Objekte, die durch »Occluder« verdeckt werden

Sie sehen das Kamera-Sichtfeld (Frustum) und 4 Objekte (A B C D). Bei einem Renderdurchlauf müssten die beiden Objekte B und C nicht gerendert werden, da sie durch das Objekt (Occluder) komplett verdeckt werden.

Erzeugen und Löschen von Queries

Occlusion Queries werden über sogenannte Query-Objekte erzeugt, welche im Prinzip so funktionieren wie die Texturen (erstellen, nutzen, löschen). Wenn man sie nicht weiter benötigt, sollte man sie löschen, um den Speicher wieder freizugeben.

Um ein Query-Objekt zu erstellen, nutzt man die Funktion:

```
void glGenQueries(GLsizei number, GLuint, *id);
```

Der Parameter *number* ist die Anzahl der Query-Objekte und *id* ein Zeiger auf ein Array, welches die Namen der Objekte beinhaltet.

Die Objekte werden über die Funktion

```
void glDeleteQueries(GLsizei number, GLuint, *id);
```

wieder gelöscht, wobei die beiden Parameter wieder dieselben sind, wie oben beschrieben.

Queries nutzen

Die Benutzung der Queries ist recht einfach: Zu Beginn ruft man die Funktion

```
void glBeginQuery(GLenum target, GLuint id);
```

auf.

Der Parameter *target* sollte *GL_SAMPLES_PASSED* sein und *id* der Name des Query-Objektes.

Nachdem die Funktion aufgerufen wurde, kann man damit beginnen, die einzelnen Objekte bzw. deren Bounding-Boxen oder Bounding-Spheres zu rendern. OpenGL zählt dann die Fragmente, die den Depth-Test bestanden haben.

Da man nicht die Objekte selbst rendert, ist es ratsam, während dieses Vorgangs keine Informationen in den Framebuffer (*glColorMask()*) und in den Depthbuffer (*glDepthMask()*) zu schreiben, da man ja die Boxen nicht sehen möchte.

Hier ist es auch sinnvoll, alle nicht benötigten Renderstates (Beleuchtung, Texturierung usw.) auszuschalten, da man sie für den Test sowieso nicht benötigt.

Nachdem der Rendervorgang fertig ist (und bevor man einen neuen Querye beginnt), beendet man den Test mit der Funktion:

```
void glEndQuery(GLenum target);
```

Der Parameter *target* ist auch hier wieder *GL_SAMPLES_PASSED*. Um nun herauszufinden, wie viele Fragmente den Depth-Test bestanden haben (gerendert werden sollen), ruft man die Funktion

```
glGetQueryObjectiv(GLuint id, GLenum pname, GLint *params);
```

auf. Auch hier ist die *id* der Name, *pname* muss *GL_QUERY_RESULT* sein, *params* enthält die Daten, die über *pname* angefordert wurden.

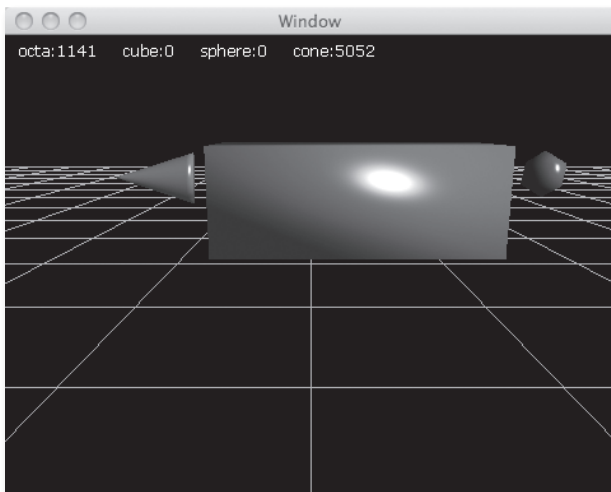
Es kann unter Umständen zu Verzögerungen bei der Abfrage der Objekte kommen, was daran liegt, dass Occlusion-Queries asynchron arbeiten. Daher ist es sinnvoll, zu prüfen, ob schon ein Ergebnis vorliegt, was so aussehen kann:

```
GLuint result = GL_FALSE;
glGetQueryObjectiv(query, GL_QUERY_RESULT_AVAILABLE, &result);
```

So lange die Funktion `GL_FALSE` zurückliefert, liegt noch kein Ergebnis vor. Das bedeutet, dass man in der Zeit andere Dinge tun kann (Physikberechnungen, Kollisionsabfragen usw.).

Ein Beispielprogramm

Im Beispielprogramm »Kapitel 14/Occlusion Queries« habe ich die Szene aus der ... Abbildung nachgebaut, um einen besseren Bezug zu bekommen. Den Quader vorne im Bild können Sie mit den Pfeiltasten bewegen.



Beispiel für Occlusion-Queries

Die Szene besteht aus den 5 Objekten:

```
CFXMesh _octa; // Mesh D
CFXMesh _cube; // Mesh C
CFXMesh _sphere; // Mesh B
CFXMesh _quader; // Mesh Occluder
CFXMesh _cone; // Mesh A
```

welche auf folgender Struktur basieren:


```
typedef struct _CFXMesh
{
    CFXVector      position; // Position
    CFXWavefrontMesh *mesh; // Model
    float          radius; // BSphere Radius
    CFXBBox        bbox;    // BBox
    GLuint         query; // Query Name
    GLint          result; // Query Result
}CFXMesh;
```

Für die Models *_cube* und *_cone* werden jeweils AABBs berechnet, alle anderen bekommen eine Sphere. Das Model *_quader* (Occluder) benötigt keine AABB bzw. keine Bounding-Sphere, da es nicht geprüft wird, sondern immer zu Anfang gerendert wird.

Das Erzeugen der Objekte sieht folgendermaßen aus:

```
// Models laden und Meshes erzeugen
_octaModel = [[CFXWavefrontMesh alloc]init];
[_octaModel loadMesh:@"octa"];
tmpRadius = boundingSphereRadius([_octaModel vertices],
[_octaModel numVertices], makeEmptyVector());
[_octaModel freeMemory];
_octa.position = makeVector(5.0, 1.0, -4.0);
_octa.mesh = _octaModel;
_octa.radius = tmpRadius;
glGenQueries(1, &_octa.query);
.
.
.
```

Nachdem das Model geladen wurde, wird zuerst die Bounding-Sphere berechnet, dann positioniert, danach das Model zugewiesen und zum Schluss eine ID für die Queries erstellt.

Der Rendervorgang geschieht nun in zwei Schritten; Zuerst werden die Bounding-Boxen bzw. die Bounding-Spheres gerendert, die Anzahl der sichtbaren Fragmente wird dann in *result* gespeichert, wichtig dabei ist, dass alle unnötigen Renderstates abgeschaltet werden, um den Vorgang zu beschleunigen.

Im zweiten Schritt wird dann geprüft, ob der Wert in *result* größer null ist, wenn ja, dann ist das Objekt zumindest teilweise sichtbar, und wir müssen es rendern.

Wie gesagt, wird der Quader ganz vorne nicht getestet, sondern immer gerendert (zu Anfang).

Hier nochmals auszugsweise der Code dazu:

```
// Shader aktivieren
[_shaderManager useShader:@"PerPixel Light"];

CFXWavefrontMesh *tmpMesh;

// Quader normal rendern
glPushMatrix();
    glTranslatef(_quader.position.x, _quader.position.y, _quader.
        position.z);
    tmpMesh = _quader.mesh;
    [tmpMesh renderMesh];
glPopMatrix();

// Nun wird getestet, welche Objekte durch den Quader verdeckt
// sind
// Dazu werden alle nicht benoetigten Renderstates abgeschaltet

// Shader deaktivieren
[_shaderManager disableShader];

// Nichts in den Tiefenpuffer schreiben
glDepthMask(GL_FALSE);

// Nichts in den Farbpuffer schreiben
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// Testen welches Objekt verdeckt wird
[self testOcclusion];

// Wieder in Tiefenpuffer schreiben
glDepthMask(GL_TRUE);

// Wieder in Farbpuffer schreiben
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

[_shaderManager useShader:@"PerPixel Light"];
```

```
// Objekte nur rendern, wenn sie nicht verdeckt werden
if(_octa.result)
{
    glPushMatrix();
        glTranslatef(_octa.position.x, _octa.position.y, _octa.
            position.z);
        tmpMesh = _octa.mesh;
        [tmpMesh renderMesh];
    glPopMatrix();
}
.
.
.
```

Die Methode *testOcclusion*, welche den eigentlichen Test durchführt, sieht so aus:

```
// Objekte abfragen
glBeginQuery(GL_SAMPLES_PASSED, _octa.query);
glPushMatrix();
glTranslatef(_octa.position.x, _octa.position.y, _octa.
    position.z);
    [self renderBSphere:_octa.radius];    // Sphere
glPopMatrix();
glEndQuery(GL_SAMPLES_PASSED);
glGetQueryObjectiv(_octa.query, GL_QUERY_RESULT, &_octa.result);
```

Zuerst wird ein Query eingeleitet, danach wird die Bounding-Sphere positioniert und gerendert. Anschließend wird das Query beendet und der Wert der gezählten Fragmente in *result* gespeichert. Der Wert in *result* ist Null, wenn das Objekt durch ein anderes verdeckt wird bzw. wenn es sich außerhalb des Sichtfeldes der Kamera befindet.

Man kann das Ganze jetzt noch ein wenig optimieren, indem man einen Schwellerwert angibt, ab welchem ein Objekt gerendert werden soll. Wenn *result* z. B. einen relativ kleinen Wert hat (z. B. unter 50), kann es durchaus sinnvoll sein, es noch nicht zu zeichnen. Gerade bei schnellen Actionspielen wird man den »Schwindel« wahrscheinlich nicht bemerken. Summiert man diese kleine Optimierung für alle Objekte, wird man bestimmt das eine oder andere Frame herauskitzeln können.

Möchte man nun diese Technik in Spielen nutzen, ist es sehr wichtig, dass man genau weiß, wie die einzelnen Levels organisiert sind. Es ist nämlich sinnlos, jedes

einzelne Objekt zu testen, weil dass sehr schnell die Performance in den Keller ziehen wird.

Ein kleines theoretisches Beispiel:

Sie befinden sich in einem Outdoor-Level und haben ein Gebäude vor sich. In diesem Gebäude befinden sich jede Menge Objekte. Jedes einzelne dieser Objekte muss demnach auf Sichtbarkeit getestet werden (inkl. dem Gebäude). Da wir clever sind, testen wir nicht jedes einzelne Objekt, sondern sagen: Solange das Gebäude nicht sichtbar ist, sind auch die Objekte darin nicht sichtbar.

Sie sehen an diesem kleinen Beispiel, wie wichtig die Datenorganisation in einem Spiel ist. Das ist einer der Punkte, der von vielen der »3D-Engine Bauern« im Internet leider komplett vergessen wird. Es genügt nämlich nicht, dass eine 3D-Engine tolle Effekte rendern kann.

Zusätzliche Informationen

Wenn Sie sich für Outdoor-Games interessieren, dann sollten Sie sich die beiden Culling-Techniken für Terrains einmal anschauen:

Quadtrees

<http://www.gamedev.net/reference/programming/features/quadtrees/>

Octrees

http://wiki.delphigl.com/index.php/Tutorial_Octree

Wenn Sie mehr auf Indoor-Spiele (Doom usw.) stehen, dann sollten Sie sich mit den BSPs (Binary Space Partitioning) einmal auseinandersetzen. Aber Achtung, die Technik ist nicht ganz einfach (meine Meinung natürlich).

<http://maven.smith.edu/~mcharley/bsp/>

Hitbox

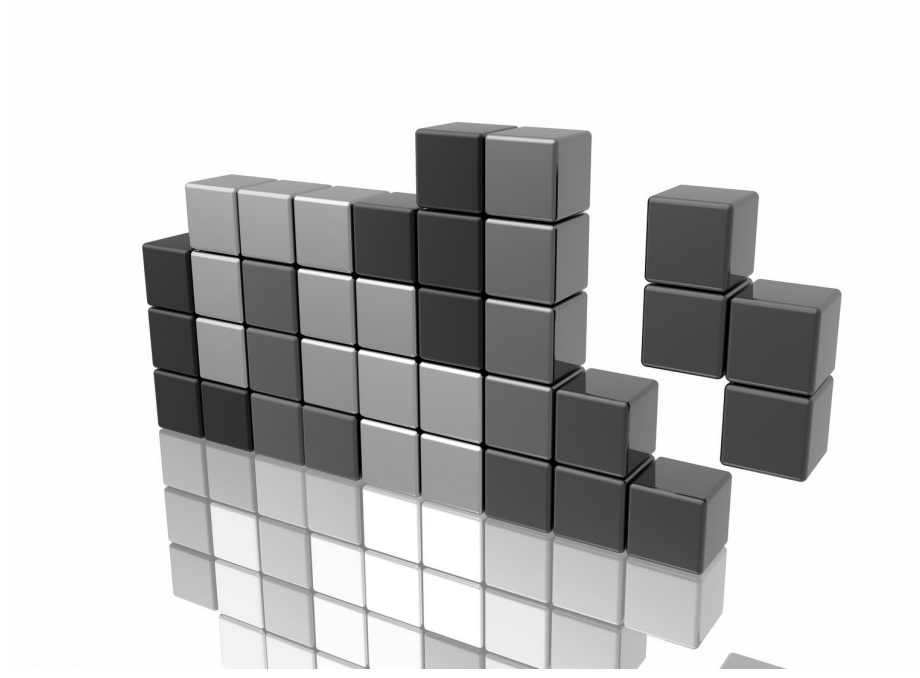
<http://de.wikipedia.org/wiki/Hitbox>

Eine große Ansammlung an verschiedenen Kollisionstests

<http://www.realtimerendering.com/int/>

Noch mehr Informationen zu Kollisionstests auf GameDev:

<http://www.gamedev.net/reference/list.asp?categoryid=45#199>



Das Spiel ScrambleX

Nachdem Sie nun das Buch bis hier durchgearbeitet haben, wird es Zeit, mit dem Erlernten ein »komplettes« Spiel zu entwickeln. Zunächst einmal muss man natürlich wissen, was für ein Spiel man gerne machen möchte. Wenn man noch recht neu in der Spieleentwicklung ist, neigt man gerne dazu, sich selbst zu überschätzen (ich schreibe aus Erfahrung), was meistens in einem Projekt endet, das nie fertig wird. Ist der Enthusiasmus zu Beginn noch riesengroß, merkt man doch recht schnell, dass es keine so gute Idee war, für das erste Projekt Doom 4 entwickeln zu wollen. Deshalb möchte ich auf folgende zwei Punkte hinweisen, die Sie bestimmt in dem einen oder anderen Programmierlehrbuch schon mal gelesen haben:

1. Starten Sie mit etwas Einfachem!
2. Bringen Sie es zu Ende!

Es spielt absolut gar keine Rolle, wie »gut« Ihr erstes Spiel wird, es ist nur wichtig, dass Sie es fertig stellen. Nichts ist frustrierender, als zig angefangene Projekte, die nie fertig wurden, nur weil man den Spaß daran verloren hat.

Wenn Sie keine Idee für ein eigenes Spiel haben, dann schauen Sie sich ruhig mal ein wenig in der 8-Bit-Szene um. Zu Beginn der Homecomputerzeiten gab es sehr viele witzige Spiele, die Sie als Vorlage nehmen können.

Aber auch hier sollte man sich gut überlegen, ob man solch einen »Remake« wirklich zu Ende bringen kann. Ich erinnere mich an einen Pac-Man-Klon, den ich vor einigen Jahren in Flash entwickeln musste. Da ich zuvor schon einige Spiele entwickelt hatte, wusste ich so ungefähr, was auf mich zukommen würde. Das Ganze lief ohne große Probleme bis zu dem Moment, in dem es darum ging, die AI (Künstliche Intelligenz) der Geister zu implementieren. Wenn man das Spiel nur spielt, merkt man nicht unbedingt, dass sich der »Vater« von Pac-Man etwas dabei gedacht hat, dass sich die Geister so bewegen, wie sie es tun. Dahinter verbirgt sich nämlich ein Algorithmus, der erst einmal verstanden werden will. Um es kurz zu machen: Es war nicht ganz einfach, das originale Spielgefühl nachzubauen.

Sie sehen also, auch vermeintlich einfache Spiele können durchaus ihre Tücken haben. Ich will Sie nicht entmutigen (ganz im Gegenteil), sondern Ihnen vielmehr frustrierende Stunden ersparen, die ich mit dem Programmieren schon hatte (und vermutlich auch noch haben werde).

Für das Spiel (ScrambleX) hier im Buch habe ich mich auch für einen Klassiker entschieden, den ich schon früher gerne spielte. In dem Spiel geht es darum, mit

einem Raumschiff durch eine Höhle zu fliegen und alles platt zu machen, was im Weg steht. Um das Ganze ein wenig schwieriger zu gestalten, muss ständig der Treibstoff im Auge behalten werden, da das Raumschiff bei leerem Tank abstürzt, was ein Leben weniger zur Folge hat. Auffüllen lässt sich der Treibstoff durch Abschießen von »Fuels«.

Das Spiel beinhaltet unterschiedliche Gegner wie z. B. Ufos oder Feuerbälle (welche nicht abgeschossen werden können). Ganz am Ende des Levels habe ich einen Endgegner eingebaut (war im Originalspiel nicht so, dort war es eine Basis), den man noch vom Bildschirm »blasen« muss, um das Spiel erfolgreich zu beenden.

TIPP

Ein weiterer Vorteil beim Nachbauen eines schon bestehenden Spieles ist, dass man schon im Vorfeld weiß, was man alles implementieren muss bzw. wie das Spiel zum Schluss aussehen soll. Natürlich steht es einem frei, noch zusätzliche Features einzubauen.

Bestandsaufnahme

Wir werden jetzt nicht wild drauflosprogrammieren, sondern uns zunächst einmal Gedanken darüber machen, was wir alles brauchen. Wir machen zuerst einmal so eine Art Bestandsaufnahme.

Da hätten wir für unser Spiel:

- den Spieler, der mit den Cursortasten bewegt werden kann (geschossen wird mit der S-Taste, die Bomben werden mit der A-Taste abgeworfen)
- die »Fuels«, welche unseren Tank auffüllen, wenn sie abgeschossen wurden
- einen einfachen Gegner, der nur auf dem Boden steht
- Raketen, die von unten nach oben fliegen, wenn sich der Spieler ihnen nähert
- Ufos, die sich nur auf und ab bewegen
- Feuerbälle, die nicht zerstört werden können
- und zum Schluss noch einen Endgegner.

Weiterhin brauchen wir:

- die Höhle
- ein HUD (Head-Up-Display)

- einen feststehenden und einen scrollenden Hintergrund
- verschiedene Partikelsysteme
- und natürlich die Sounds bzw. die Hintergrundmusik.

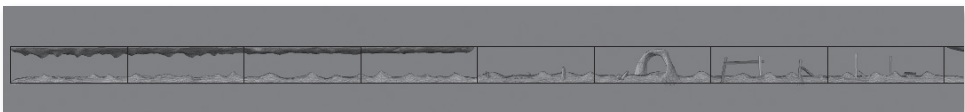
Nachdem wir wissen, was wir an Objekten benötigen, und den Spielablauf an sich auch kennen, können wir uns Gedanken um die Datenorganisation machen. Dieser Part darf auf gar keinen Fall unterschätzt werden, da es im Nachhinein schwierig bis unmöglich ist, Dinge mit einzubauen (bzw. zu verändern), die in diesem Schritt vergessen wurden. Beginnen wir damit, uns zu überlegen, wie wir die Objekte am besten organisieren können, um sie effizient zu rendern. Der größte Brocken ist die Höhle selbst. Da sie aus knapp 19.000 Polygonen besteht, müssen wir uns überlegen, wie wir sie am schnellsten rendern können.

TIPP

Natürlich bringen heute 19.000 Polygone keine Grafikkarte mehr zum Schwitzen, aber Sie lernen dadurch, wie man seine Daten effizient verwaltet. Eine schlecht organisierte Datenstruktur zwingt irgendwann jeden Rechner in die Knie, egal, wie schnell er ist.

Da wir keine freibewegliche Kamera haben (sie schaut immer entlang der Z-Achse und wird nur von links nach rechts gescrollt), müssen wir die Höhle auch nicht auf einmal anzeigen, sondern nur die Teile, welche aktuell zu sehen sind.

Wir müssen also die Höhle in mehrere Teilstücke zerlegen, was folgende Abbildung verdeutlichen soll.



Unterteilung des Höhlenmesh

Die schwarzen Linien zeigen die Bereiche (insgesamt zwölf), in welche wir das Modell unterteilen wollen. Mit den zwölf Unterteilungen stellen wir sicher (ich habe es ausprobiert), dass nie mehr als zwei Teile der Höhle zur gleichen Zeit gerendert werden müssen, da ein Teil gerade so in das Sichtfeld der Kamera passt.

Nachdem das Modell unterteilt wurde, wird jedes der Teilstücke gegen das Frustum gecullt. Wenn es sichtbar ist, wird es gerendert. Dadurch ist es möglich, relativ große Modelle in einer guten Geschwindigkeit auf dem Bildschirm auszugeben.

Höhlenmesh aufteilen

Beginnen wir mit dem Höhlenmesh. Die Unterteilung des Meshs ist einfacher, als man vielleicht glauben mag. Zuerst einmal erstellen wir wieder eine Struktur, welche solch einen Abschnitt (Chunk) darstellen soll:

```
// Ein Chunk ist eine Unterteilung des Höhlen-Meshs
typedef struct _Chunk
{
    CFXBBox      bBox;           // BBox des Chunks
    GLushort     *vertexIndicies; // Array Indicies
    GLuint       numIndicies;    // Anzahl Indicies
    GLuint       vboIndicies;    // VBO Puffer für die Indicies
    BOOL         inFrustum;      // Ist Chunk sichtbar
    CFXVector     *collisionVertices; // Kollisionsdreiecke im Chunk
    int          numCollisionVertices; // Anzahl der Kollisions-
                                     // dreiecke im Chunk
}Chunk;
```

Die BBox ist die Bounding-Box, welche den Abschnitt umschließt, diese benötigen wir später beim Rendern, um sie gegen das Frustum zu cullen.

Weiterhin werden die Indexe (*vertexIndicies*) und die Anzahl der Indexe (*numIndicies*) gespeichert. Dann brauchen wir einen VBO pro Abschnitt, was wir gleich noch sehen werden. Zusätzlich benötigen wir eine BOOL-Variable (*inFrustum*), die aussagt, ob der Abschnitt gerade zu sehen ist. Die letzten beiden Variablen (*collisionVertices* und *numCollisionVertices*) brauchen wir für die Kollisionsabfrage mit den Spielobjekten, was wir uns später noch genauer anschauen werden.

Nachdem das Model geladen wurde, generieren wir wieder unsere VBOs für die Rohdaten (Vertices, UVs, Normale), so wie wir es im Kapitel 11 gesehen haben. Dann wird es spannend: Wir generieren nun nicht mehr ein Index-Array für das komplette Model, sondern zwölf, für jeden Abschnitt ein eigenes. Durch diese Vorgehensweise müssen wir beim Rendern nur noch das Index-Array austauschen.

AUFGEPASST

Die Arrays (VBOs) für die Rohdaten beinhalten die Daten für das komplette Höhlenmesh, lediglich das Index-Array wird bei der Vorgehensweise getauscht.

Das bedeutet, wenn ein Chunk sichtbar ist, nehmen wir sein Array als aktuelles Index-Array und rendern nur die Polygone, welche in diesem Array beschrieben wurden.

Nun kommt der Teil, in dem wir schauen müssen, in welchem Chunk denn ein Polygon liegt. Auch das ist nicht weiter schwierig, schauen wir uns die Methode

```
-(void)splitMesh:(int)splits
```

in der Datei »Cave.m« einmal an.

Wir nehmen das Polygon und prüfen mit der Funktion

```
// Testet ob das Dreieck die Box schneidet
static inline int triBoxOverlap(float boxcenter[3],float boxhalf-
size[3],float triverts[3][3])
...
```

in welcher Bounding-Box es sich befindet. Wenn wir die betreffende Bounding-Box gefunden haben, speichern wir den Index des Polygons im Index-Array des Chunks und wiederholen das Ganze so lange, bis keine Polygone mehr übrig sind. Wenn alle Polygone durchlaufen wurden, generieren wir die Index-Arrays:

```
// VBOs generieren
for(x=0; x<_splits; x++)
{
    glGenBuffers(1, &_chunks[x].vboIndices);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _chunks[x].
        vboIndices);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, _chunks[x].numIn-
        dices * sizeof(GLushort), _chunks[x].vertexIndices,
        GL_STATIC_DRAW);
}
```

Höhle rendern

Um die Vorgehensweise der Unterteilung nochmals zu verdeutlichen, schauen wir uns gleich den Rendervorgang der Höhle an:

```
int x;
for(x=0; x<_splits; x++)
{
    BOOL inFrustum=[_sharedFrustum isAABBinFrustum: _chunks[x].
    bBox];
    if(inFrustum)
    {
        _chunksInFrustum++;
// Indexe
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _chunks[x].
        vboIndices);
        // Rendern
        glDrawElements(GL_TRIANGLES, _chunks[x].numIndices, GL_UN-
        SIGNED_SHORT, 0);
        // Chunk ist sichtbar
        _chunks[x].inFrustum = YES;
    }
    else
    {
        // Chunk ist nicht sichtbar
        _chunks[x].inFrustum = NO;
    }
}
```

Wir durchlaufen also alle Unterteilungen (*_splits*) und prüfen, ob die Bounding-Box eines Chunks im Frustum ist. Wenn ja, binden wir den Index-Array des Chunks und rendern den Teilabschnitt der Höhle. Mehr ist es nicht.

Sie sehen also, wir können auf diese Weise die Höhle unendlich lang machen (so lange genügend RAM da ist), ohne uns Sorgen über die Polygonzahl machen zu müssen.

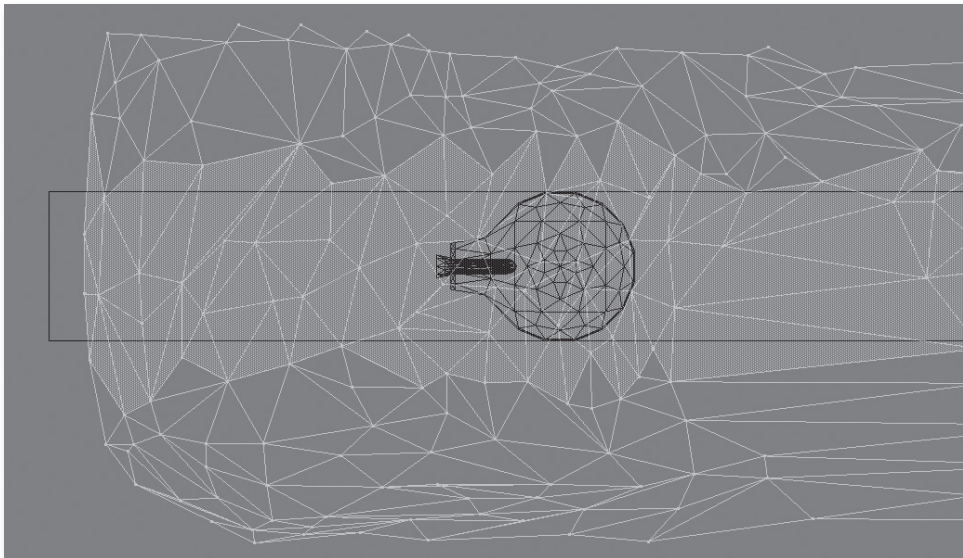
Diese Art von Unterteilung kommt in sehr vielen Spielen zum Einsatz. Zum Beispiel werden Terrains in Outdoor-Spielen auf eine ähnliche Art gesplittet und gerendert, was auch sinnvoll ist, da diese in der Regel auch aus sehr vielen Polygonen bestehen.

Kollision mit der Höhle

Was noch fehlt, ist eine Kollisionsprüfung zwischen den Spielobjekten und der Höhle. Hier kommt nun die SAT (Separating Axis Theorem) zum Einsatz. Wir prüfen hier, ob die AABB des Spielobjektes mit einem Polygon der Höhle kollidiert.

Auch hier müssen wir uns wieder zuerst überlegen, wie man die Kollisionsprüfung auf ein Minimum reduzieren kann, da diese ja auch Zeit in Anspruch nimmt. Es ist zunächst einmal klar, dass der Spieler nur mit den Teilen (Chunks) der Höhle kollidieren kann, welche auch sichtbar sind, wodurch wir einen Großteil der Polygone schon mal aussortieren können. Weiterhin kann sich der Spieler (das gilt auch für alle anderen Spielobjekte) nicht auf der Z-Achse bewegen, das bedeutet, er kann immer nur mit den Polygonen der Höhle kollidieren, welche sich auch im Bereich (Z-Position) seiner Bounding-Box befinden. Die folgende Abbildung macht das ein wenig klarer.

Sie sehen einen Höhlenabschnitt von oben (Z-Achse verläuft von oben nach unten), die hellgrau schattierten Flächen sind die, mit welchen der Spieler auch kollidieren kann. Die schwarze Box um das Spielermodell soll nochmals den Bewegungsbereich des Spielers verdeutlichen.



Kollisionsdreiecke der Höhle

Hierdurch vermindern wir nochmals den Kollisionstest der Spielobjekte mit der Höhle.

Kommen wir nun zu dem Code, in welchem die besagten Polygone aus dem Höhlenmesh extrahiert werden.

```
-(void)calculateCollisionMesh:(float)sizeZOfPlayersBBBox
splits:(int)splits
```

Wir übergeben der Methode die Ausdehnung der Spieler-Bounding-Box auf der Z-Achse und die Anzahl der Unterteilungen (12). Danach machen wir dasselbe wie bei der Unterteilung des Höhlenmeshs: Wir schauen wieder, ob sich ein Polygon in der Bounding-Box befindet, wenn ja, speichern wir das Polygon in *collisionVertices* des jeweiligen Chunks. Bei der Kollisionsprüfung müssen wir dann nur noch diejenigen Polygone nehmen, die in *collisionVertices* gespeichert wurden.

Um den Zusammenhang nochmals zu verdeutlichen, auch hier wieder der Code in der Datei »GameObject.m«, welcher auf eine Kollision zwischen einem Spielobjekt und der Höhle testet.

```
// Kollision mit der Hoehle
-(BOOL)collideWithCave
{
    Chunk *chunks = [_cave chunks];
    CFXVector *collisionvertices;
    int numCollisionVertices;
    int x;
    int i;
    float boxhalfsize[3];
    float boxcenter[3];
    float triverts[3][3];

    memcpy(boxcenter, &_amp;_position, sizeof(CFXVector));
    memcpy(boxhalfsize, &_amp;bBoxHalfSize, sizeof(CFXVector));

    CFXVector tmp1, tmp2, tmp3;

    for(i=0; i<NUM_SPLITS_OF_CAVE; i++)
    {
        if(chunks[i].inFrustum)
        {
            collisionvertices = chunks[i].collisionVertices;
            if(collisionvertices)
            {
```

```

        numCollisionVertices = chunks[i].
        numCollisionVertices;
        for(x=0; x<numCollisionVertices; x+=3)
        {
            tmp1 = collisionvertices[x];
            tmp2 = collisionvertices[x+1];
            tmp3 = collisionvertices[x+2];

            memcpy(&triverts[0][0], &tmp1,
                sizeof(CFXVector));
            memcpy(&triverts[1][0], &tmp2,
                sizeof(CFXVector));
            memcpy(&triverts[2][0], &tmp3,
                sizeof(CFXVector));

            if( triBoxOverlap(boxcenter,boxhalfsize,
                triverts) )
            {
                [_collisionSound playSound];
                return YES;
            }
        }
    }
}
return NO;
}

```

Wir durchlaufen nun wieder alle Unterteilungen der Höhle und prüfen zuerst, ob sich ein Chunk im Frustum befindet (nicht sichtbare brauchen wir nicht zu prüfen). Dann nehmen wir alle Polygone für die Kollision (*chunks[i].collisionVertices*) und prüfen sie auf eine Überschneidung *triBoxOverlap(boxcenter,boxhalfsize,triverts)* mit der AABB des jeweiligen Spielobjektes, mehr ist es auch hier nicht.

Fassen wir einmal zusammen, was wir durch diese Vorgehensweise erreicht haben:

- Wir rendern nur die Teile der Höhle, die auch zu sehen sind.
- Es wird nur auf eine Kollision zwischen den Polygonen und den Spielobjekten geprüft, welche sichtbar sind und sich im Bewegungsbereich der Spielobjekte befinden.

Hätten wir diese Punkte nicht berücksichtigt, müssten wir ständig die komplette Höhle rendern und auch immer alle Polygone der Höhle zum Kollisionstest hernehmen, was natürlich nicht sonderlich clever wäre. Damit hätten wir schon den Part mit der Höhle fertig. Kommen wir nun zu den Spielobjekten.

Spielobjekte

Wie gesagt, gibt es mehrere Spielobjekte, welche alle recht unterschiedlich sind.

- »Cave« ist die schon besagte Höhle.
- »Fuel« sind die Tanks, die abgeschossen werden müssen, um Treibstoff zu bekommen. Sie bewegen sich nicht und stehen nur am Höhlenboden.
- »Rocket« sind die Raketen, die von unten nach oben fliegen, wenn sich der Spieler ihnen nähert.
- »Saucer« sind die Ufos, die sich wellenartig von unten nach oben bewegen.
- »Bomb« bzw. »Shot« sind die Waffen des Spielers.
- »Enemy« (hier viel mir leider kein Name ein) sind Gegner, die nur am Boden stehen und nichts tun.
- »Fireball« sind die Feuerbälle, die von rechts nach links fliegen und nicht abgeschossen werden können.
- »Boss« ist der Endgegner.
- »Player« ist natürlich unser Raumschiff, welches wir über die Cursortasten steuern können.

All diese Objekte leiten von `GameObject` ab, welches die grundlegenden Methoden

```
-(void)update:(float)deltaTime cameraPosition:(CFXVector)camera-  
Position;// Objekt aktualisieren  
-(void)render;// Objekt rendern  
-(void)calculateBBBox;// Berechnet AABB (wird automatisch  
aufgerufen)  
-(BOOL)collideWithCave; // Objekt ist mit der Hoehle kollidiert  
-(BOOL)collideWithPlayer;// Objekt mit Spieler kollidiert  
-(BOOL)isInFrustum;// Ist Objekt sichtbar
```

beinhaltet.

Diese Spielobjekte wiederum werden durch die Klasse *GameObjectsManager* (Singleton) verwaltet. Dieser macht nicht mehr, als die Objekte zu aktualisieren / rendern und »tote« Objekte zu löschen. Die Spielobjekte haben die Besonderheit, dass sie sich fast völlig unabhängig von den anderen Spielobjekten selbst verwalten. Das bedeutet, sie prüfen selbst, ob sie mit dem Spieler bzw. der Höhle kollidieren, und reagieren dann entsprechend darauf. Diese Vorgehensweise erspart uns jede Menge verschachtelter Schleifen im eigentlichen Spiel, wodurch dieses sehr übersichtlich bleibt. Der eigentliche Rendervorgang ist recht unspektakulär, es wird nur geprüft, ob das Objekt sichtbar ist, und wenn ja, werden die Materialien und die entsprechende Textur gesetzt und gerendert:

```
-(void)render
{
    // Nur rendern, wenn das Objekt auch sichtbar ist
    if(!_isInFrustum)
    {
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, [_model diffuse]);
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, [_model ambient]);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, [_model specular]);
        glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, [_model emission]);
        glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, [_model shininess]);

        // Vertices
        glBindBuffer(GL_ARRAY_BUFFER, [_model bufferVertices]);
        glVertexPointer(3, GL_FLOAT, 0, 0);

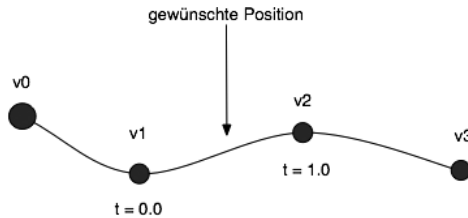
        // Normale
        glBindBuffer(GL_ARRAY_BUFFER, [_model bufferNormals]);
        glNormalPointer(GL_FLOAT, 0, 0);

        // Textur-Koordinaten
        glBindBuffer(GL_ARRAY_BUFFER, [_model bufferTexCoords]);
        glTexCoordPointer(2, GL_FLOAT, 0, 0);
        .
        .
        .
    }
}
```

Der Endgegner

Um das Abschießen des Endgegners ein wenig schwieriger zu machen, wird dieser entlang eines Splines bewegt. Dabei verwenden wir einen sogenannten Catmull-Rom-Spline, der es erlaubt, eine weiche Interpolation zwischen mehreren Stützpunkten zu erzeugen.

Die folgende Abbildung zeigt 4 Punkte, durch welche ein Spline verläuft.



Ein Spline, welcher durch vier Punkte beschrieben ist

Nun wollen wir, dass unser Objekt genau entlang dieses Splines bewegt wird. Zuerst einmal benötigen wir die Stützpunkte an sich. Diese sind in der Datei »Spline.h« als Array definiert.

Mit diesen einzelnen Stützpunkten bewaffnet, extrahieren wir die 4 Stützpunkte, die wir brauchen, um unseren Spline zu definieren.

In der Datei »Boss.m« sind das die Vektoren $p0$ - $p3$ aus dem Array (*spline*).

Mit folgender Formel wird dann der Punkt (t) berechnet, welcher zwischen den beiden Stützpunkten ($v1$, $v2$) liegt.

$$q(t) = 0.5 * ((2 * P1) + (-P0 + P2) * t + (2*P0 - 5*P1 + 4*P2 - P3) * t^2 + (-P0 + 3*P1 - 3*P2 + P3) * t^3)$$

Diese Formel müssen wir nun auf die X/Y-Koordinate unseres Spielobjekts anwenden:

```
// Catmull-Rom Interpolation zwischen den Punkten
x = 0.5f * ((2.0f * p1.x) +
(-p0.x + p2.x) * _t +
(2.0f * p0.x - 5.0f * p1.x + 4.0f * p2.x - p3.x) * pow(_t, 2.0f) +
(-p0.x + 3.0f * p1.x - 3.0f * p2.x + p3.x) * pow(_t, 3.0f));

y = 0.5f * ((2.0f * p1.y) +
(-p0.y + p2.y) * _t +
(2.0f * p0.y - 5.0f * p1.y + 4.0f * p2.y - p3.y) * pow(_t, 2.0f) +
(-p0.y + 3.0f * p1.y - 3.0f * p2.y + p3.y) * pow(_t, 3.0f));
```

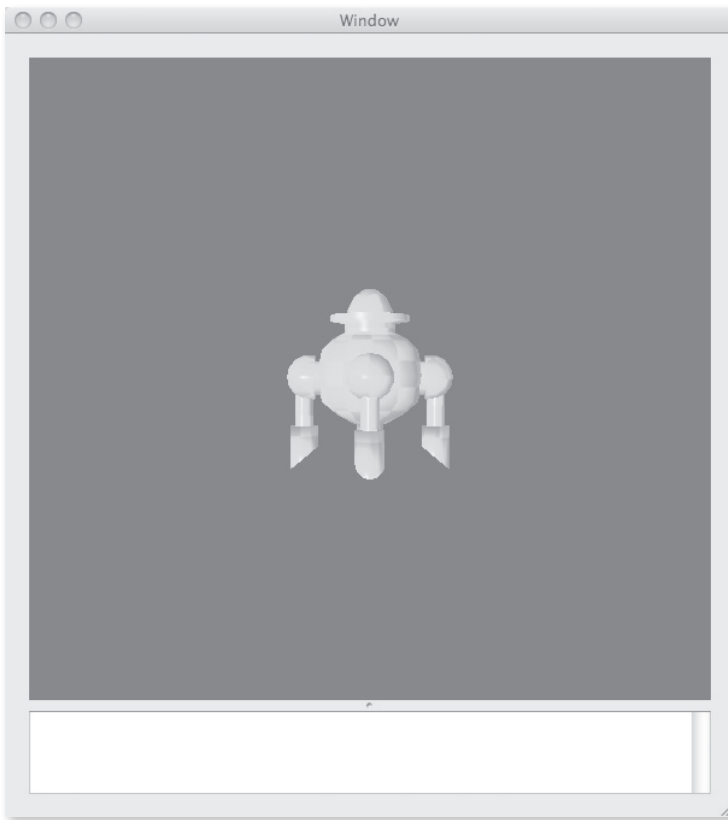
Sie sehen, die Formel ist für den X- bzw. Y-Wert identisch. Die Z-Koordinate wird außen vor gelassen, da eine Bewegung auf der Z-Achse nicht erwünscht ist.

POWER

Diese Art von Bewegung wird sehr oft dazu verwendet, um Kamerafahrten zu realisieren. Alles, was hierbei zu tun ist, ist die Kameraposition bzw. den Look-At-Vektor mit der oben gezeigten Formel zu berechnen und in die Funktion *gluLookAt(...)* einzusetzen.

Die Models

Die Models für die Objekte werden nun nicht mehr direkt als Wavefront-Model geladen, sondern zuerst konvertiert. Dazu habe ich einen kleinen Konverter geschrieben, den Sie im Ordner »Kapitel 15 / WavefrontMesh Konverter« finden.



Konverter, welcher Wavefront-Dateien in das Binär-Format speichert

Dieser tut nichts anderes, als ein obj-Model zu laden und es als plist (ohne Endung) zu speichern. Die Rohdaten werden darin binär als Key-Value-Paare gespeichert, wodurch sie sich relativ schnell wieder auslesen lassen.

AUFGEPASST

Wenn Sie den Konverter benutzen, beachten Sie bitte, dass sich die einzelnen Dateien, die zu einem 3D-Model gehören (.obj, .mtl, Texturdatei), alle im gleichen Ordner befinden müssen, da sonst die Konvertierung nicht funktioniert.

Geladen werden die Models dann im Spiel über die Klasse CFXBinaryWavefront-Mesh, welche im Prinzip so funktioniert wie die Ihnen bekannte Klasse CFXWavefrontMesh, mit dem Unterschied, dass die Polygone nun nicht mehr erst aufbereitet werden müssen, was den Ladevorgang natürlich erheblich beschleunigt. Die Funktionsweise des Konverters ist recht einfach, weshalb ich sie nur kurz andeuten werde. Es werden die Rohdaten, die aus dem obj-File gelesen wurden, einfach byte-weise in die plist geschrieben, was so aussieht:

```
-( BOOL)saveMeshAsBinary:(NSString*)filename
{
    NSMutableDictionary *dict = [[NSMutableDictionary alloc]
    init];
    [dict setObject:[NSNumber numberWithInt:_numIndexes]
    forKey:@"Number_of_Indicies"];
    [dict setObject:[NSNumber numberWithInt:_numVertices]
    forKey:@"Number_of_Vertices"];

    NSData *i = [NSData dataWithBytes:_indices
    length:sizeof(GLushort)*_numIndexes];
    [dict setObject:i forKey:@"Indicies"];
    .
    .
    .
}
```

Später beim Laden der Datei müssen die Bytes dann nur noch in die Arrays für die VBOs kopiert werden:

```
-( BOOL)loadMesh:(NSString*)filename
{
    BOOL result = YES;
```

```
NSMutableDictionary *dict = [NSMutableDictionary
dictionaryWithContentsOfFile:filename];

// Num Indices
NSNumber *n = [dict objectForKey:@"Number_of_Indices"];
_numIndexes = [n intValue];

// Num Vertices
n = [dict objectForKey:@"Number_of_Vertices"];
_numVertices = [n intValue];

// Indices
NSData *d = [dict objectForKey:@"Indices"];
_indicies = (GLushort *)calloc(_numIndexes,
sizeof(GLushort));
[d getBytes:_indicies length:_numIndexes*sizeof(GLushort)];
```

Der Geschwindigkeitszuwachs ist enorm, so dass auch Models mit sehr vielen Polygonen recht schnell geladen werden können. Dies wäre ein Weg, verschiedene Modelldformate in einem Spiel zu verwenden, da nach der Konvertierung eines Models immer nur eine plist »übrig« bleibt. Wenn man seine Datei vor fremden Blicken schützen möchte (eine plist kann man ja nach wie vor öffnen), könnte man die Daten z. B. als Array einfach in einem Rutsch (alles hintereinander) in eine Binärdatei schreiben. Das Auslesen dieser Datei wäre ja nicht weiter schwierig, da man genau weiß, wie viele Bytes man lesen muss (Offset), bevor die nächsten Daten kommen (also Num Indices, Num Vertices, Indices usw.).

Das Level

Das Level selbst wird auch aus einer plist geladen. In dieser stehen lediglich die Objekte mit den Werten für die Objektpositionen. All diese Objekte werden dann beim Start in den Objektmanager eingehängt:

```
// Alle Objekte im Dict durchgehen und in Array einhaengen
for(key in level)
{
    aArray = [level objectForKey:key];
    NSString *name = [aArray objectAtIndex:0];

    if([name isEqualToString:@"Enemy"])
    {
        Enemy *o = [[Enemy alloc]init];
        [o setModel:_enemyMesh];
        [o setPlayer:_player];
        [o setPosition:makeVector(
            [[aArray objectAtIndex:1]floatValue],
            [[aArray objectAtIndex:2]floatValue],
            [[aArray objectAtIndex:3]floatValue])];
        [_sharedGameObjectsManager addGameObject:o];
        [o release];
    }
    else if([name isEqualToString:@"Fuel"])
    {
        .
        .
        .
    }
}
```

Sie sehen, der Vorgang ist recht simpel, es wird zuerst der Type (*name*) des Objekts eingelesen, je nachdem, was für ein Typ von Spielobjekt es ist, wird dann ein Model zugewiesen, die Position gesetzt und es dem Objektmanager übergeben.

Das HUD

Die Informationen für Punktestand, Leben usw. werden über ein HUD ausgegeben. Um dieses zu rendern, wird zunächst in den Orthomodus umgeschaltet, die Texturkoordinaten gesetzt (je nachdem, was gerade ausgegeben werden soll) und dann mit Hilfe von Quads auf den Bildschirm gezeichnet:

```
.  
.   
.   
// Levelanzeige  
glBegin(GL_QUADS);  
glTexCoord2f(0.46535, 0.9677);  
glVertex3f(48.0f, (GLsizei)rect.size.height-36, -0.2);  
glTexCoord2f(0.70385, 0.9677);  
glVertex3f(370.0, (GLsizei)rect.size.height-36, -0.2);  
glTexCoord2f(0.70385, 0.9942);  
glVertex3f(370.0, (GLsizei)rect.size.height-9, -0.2);  
glTexCoord2f(0.46535, 0.9942);  
glVertex3f(48.0f, (GLsizei)rect.size.height-9, -0.2);  
glEnd();  
.   
.   
. 
```

Die Texturen für die HUD-Elemente befinden sich alle in einer einzigen Textur, was den Vorteil hat, dass nicht zur Laufzeit zwischen mehreren Texturen umgeschaltet werden muss. Es genügt dann vollkommen, die Texturkoordinaten entsprechend dem Element, welches gerade gerendert werden soll, anzupassen.

Diese Technik nennt sich »Textur-Atlas« und ist recht weit verbreitet. Nehmen wir z. B. unsere Spielobjekte: Anstatt für jedes Objekt eine neue Textur zu erzeugen, packen wir alle Texturen in eine einzige Textur und passen beim Rendern nur die Texturkoordinaten an, was eine Menge an Zeit spart, da das ständige Umschalten der Texturen entfällt.

Die Texturen

Das Spiel verwendet insgesamt 7 Texturen, die allesamt so zusammengefasst (Atlasung) wurden, dass so wenig wie nötig zwischen den einzelnen Textureinheiten umgeschaltet werden muss:

- »hud.tiff« ist für das HUD zuständig.
- »starfield.tiff« ist die Hintergrundtextur.
- »particles.tiff« ist die Textur für die Partikelsysteme.
- »ships.tiff« beinhaltet die Texturen für alle Spielobjekte.
- »cave.tiff« ist die Textur für die Höhle.
- »ships_glow.tiff« bzw. »cave_glow.tiff« ist eine besondere Art von Textur, welche sich »Glow-Textur« nennt.

Diese Textur wird im Fragmentshader zu der eigentlichen Textur hinzuaddiert:

```
gl_FragColor = color * (texture2D(texture, gl_TexCoord[0].st)
+texture2D(glowTexture, gl_TexCoord[0].st))+specular;
```

wodurch dieser »Glüheffekt« entsteht. Man kann ihn besonders in dem Höhlenabschnitt erkennen, in dem die Lava zu sehen ist.

Bevor wir diese Glow-Textur rendern, müssen wir sie aktivieren und an den Shader übergeben, was folgendermaßen aussieht:

```
// Textur Layer 0
glActiveTexture (GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, [_model texture]);

// Textur Layer 1
glActiveTexture (GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, _glowTexture);

_shaderManager sendUniform1Int:@"texture" parameter:0];
[_shaderManager sendUniform1Int:@"glowTexture" parameter:1];
```

Sie sehen also, dass an die erste Textureinheit die Textur für das Model gebunden wird und an die zweite Einheit die Glow-Textur.

Die Höhle selbst bekommt eine eigene Textur, was daran liegt, dass sie recht groß ist. Wenn wir sie zu den anderen Texturen für die Spielobjekte packen würden, wäre sie zu klein und würde beim Rendern nicht sonderlich schön (pixelig) aussehen.

Die Partikelsysteme

Alle Partikelsysteme im Spiel leiten von `ParticleSystemBase` ab, welche die Basis-Klasse ist. Ich habe für jeden Effekt (Explosion, Rauch usw.) eine eigene Klasse angelegt, welche auch nur einen einzigen Effekt rendern kann. Wie schon im Kapitel über die Spezialeffekte angedeutet, ist das nicht die allerbeste Lösung, da sie sehr unflexibel ist. Der Vorteil liegt aber darin, dass diese Methode sehr schnell zu einem Ergebnis führt und auch recht einfach zu verstehen ist.

Die Partikelsysteme werden durch den `ParticleSystemManager` (Singleton) verwaltet, der die Systeme aktualisiert und rendert.

Hier ein kurzes Beispiel, wie ein Raucheffekt erstellt wird:

```
ParticleSystem *t = [[ParticleSystem alloc] init];
[t initSystem:20 onPosition:_position];
[_sharedParticleSystem addSystem:t];
[t release];
```

Man übergibt der Methode `initSystem` die Anzahl der Partikel (20) und die Position, wo es gerendert werden soll.

Zum Schluss hängt man das System im Partikelmanager ein, der den Rest der Arbeit macht.

```
-(void)updateAndRender:(float)timeDelta cameraPosition:(CFXVector)
pos
{
    NSArray *a = [NSArray arrayWithArray:_systems];
    NSEnumerator *deadParticles = [a objectEnumerator];
    ParticleSystemBase *deadS;
    while(deadS = [deadParticles nextObject])
    {
        if([deadS isDead])
            [_systems removeObject:deadS];
    }
}
```



```
NSEnumerator *enumerator = [_systems objectEnumerator];
ParticleSystemBase *s;
while(s = [enumerator nextObject])
{
    [s update:timeDelta cameraPosition:pos];
    [s render];
}
}
```

Mit dem ersten Enumerator (*deadParticles*) holen wir uns alle »toten« Partikelsysteme und löschen sie zunächst, mit dem zweiten (*enumerator*) durchlaufen wir dann anschließend alle aktiven Systeme und aktualisieren bzw. rendern sie.

Wenn Sie mit Objective-C 2.0 arbeiten, können Sie anstatt eines NSEnumerator auch das

for in-Statement nutzen, da dieses laut Xcode-Dokumentation schneller sein soll.

AUFGEPASST

Bitte verändern Sie niemals ein Array (Löschen bzw. Hinzufügen von Objekten) während Sie es durchlaufen, da es sonst zu unschönen Abstürzen kommt.

Ich habe die Partikelsysteme (Particles) in der Ordnerstruktur von Xcode so angelegt, dass Sie sofort sehen, welcher Effekt »wohin« gehört.

Die Shader

Das Spiel verwendet eine einfache Per-Pixel-Beleuchtung, die Sie ja schon im Kapitel über Shader kennengelernt haben. Um die Szene ein wenig abzdunkeln, habe ich einen kleinen Trick eingebaut, und zwar wird das Licht kreisförmig abgeschwächt, was dann so ähnlich aussieht, als leuchtete eine Taschenlampe die Szene um den Spieler herum aus.

Das Ganze basiert auf einer Lichtabschwächung, die folgendermaßen berechnet wird:

Im Vertexshader wird ein Radius von 4.0 für das Licht definiert. Dieser Radius fließt in die Berechnung der Lichtrichtung mit ein.

```
float lightRadius = 4.0;
lightDir = (gl_LightSource[0].position.xyz - vertexPos) /
lightRadius;
.
.
.
```

Im Fragmentshader wird dann die Abschwächung des Lichts berechnet.

Die Formel dazu ist:

$$\text{Lichtabschwächung} = 1 - ((d * d) / (r * r))$$

d = Distanz zwischen Lichtposition und Vertex

r = Radius des Lichts

Vereinfacht sieht die Formel dann so aus:

```
float atten = max(0.0, 1.0 - dot(l, l));
```

Dieser Wert wird mit den einzelnen Parametern (ambient, diffuse usw.) des Lichtes multipliziert.

```
vec4 diffuse = gl_FrontLightProduct[0].diffuse * nDotL * atten;
.
.
.
```

Die Szene wäre mit dieser Art von Beleuchtung aber zu dunkel, weshalb ich ein globales ambientes Licht mit eingebaut habe:

```
// leichte Grundbeleuchtung der Szene
float global_ambient[] = { 0.15f, 0.15f, 0.15f, 1.0f };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
```

Der Rest

So, das war dann alles. Sie sehen nun, dass aus dem anfänglich gelben Dreieck doch noch etwas Anständiges geworden ist.

Ich habe im Spiel auf zusätzliche Features wie Highscoreliste, Videos, zusätzliche Levels usw. verzichtet, damit das Ganze übersichtlicher bleibt. Man kann aber schon sehr gut erkennen, welchen Umfang ein solches Spiel haben kann. Obwohl relativ wenig auf dem Bildschirm passiert, ist es doch ein ganz schöner Brocken geworden.



Das komplette Spiel ScrambleX

Schon alleine das Entwickeln der Wrapper-Klassen für das Spiel benötigte doch einiges an Zeit.

An dieser Stelle ist vielleicht die Frage angebracht »Warum so viel Arbeit für ein kleines Spielchen«. Nun, wenn man ein wenig über den Tellerrand schaut (und das ist hier durchaus angebracht) und sieht, was auf dem PC-Markt in dieser Richtung geht, kann man schon ein wenig neidisch werden. Ich spreche hier nicht von den

vielen Tutorials und Anleitungen, die es im Netz zur Spielprogrammierung für PCs gibt, sondern davon, warum Apple in diesem Sektor nicht ein wenig aktiver wird. Würde es etwas Vergleichbares wie DirectX für OS X geben, wäre es natürlich um einiges einfacher, Spiele für den Mac zu entwickeln. Aber was noch nicht ist, kann ja noch werden, Apple ist ja schon immer für eine Überraschung gut gewesen.

In diesem Sinne wünsche ich Ihnen viel Spaß beim Spielprogrammieren.

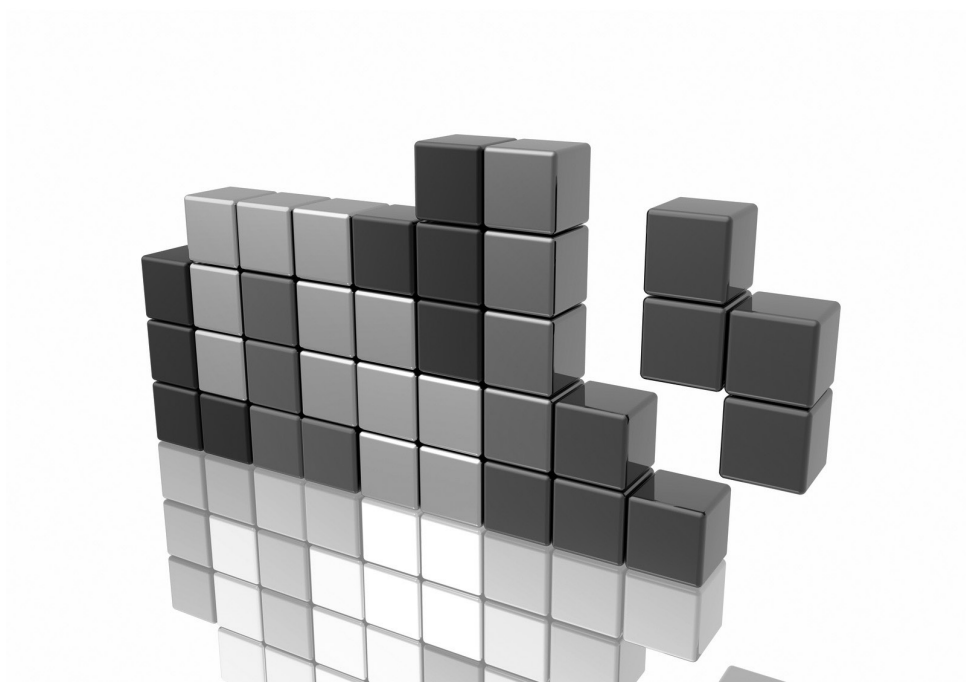
Zusätzliche Informationen

Fast Enumeration

Cocoa-Dokumentation

Catmull-Rom Spline und andere Spline-Arten

http://en.wikipedia.org/wiki/Catmull-Rom_spline#Catmull.E2.80.93Rom_spline



Index

Symbole

3d Max	288
3D-Model-Loader	288

A

AABB	115, 405
AGL	17
AI	434
ALCcontext	384
alcCreateContext	384
ALCdevice	384
alcGetProcAddress	395
alcGetString	393
alcIsExtensionPresent	395
ALC_MAC_OSX_SPATIAL_RENDERING_	
QUALITY_HIGH	395
alcMakeContextCurrent	384
alcOpenDevice	384
alDopplerFactor	392
alGetError	384
alGetString	393
alListener3f	388
Alpha-Blending	160
Alpha-Masking	195, 258, 373
Alphawert	160
alSource3f	388
alSpeedOfSound	392
ALUT	383
alutLoadWAVFile	385, 398
alutUnloadWAV	398
Ambientes Licht	138, 351
anisotropischen	193
Antialiasing	37, 169
ARB	16
ARB_texture_rectangle	184
ASCII	246
Asteroids	107
attribute	331
Audio-Streams	399
Autosizing	29
Axis-Aligned-Bounding-Box	115, 405
Axis-Angle Rotation	131

B

Backface-Culling	94
Bibliotheken	19

GL	19
GLU	19
GLUT	20
Bildwiederholfrequenz	423
Bildwiederholrate	127
Billboards	250
Blinn-Phong	353
Bounding-Box	115, 404
Bounding-Sphere	411
build-in Funktionen	338
build-in Variablen	333

C

C-Arrays	294
Catmull-Rom	444
CCW	93
Cg	380
CGDisplayHideCursor	127
CGDisplayShowCursor	127
CGL	17
Chunk	437
clamp	378
Clip-Koordinaten	326
Color-Tracking	154
const	332
Core Audio	382
cos	86, 271
CW	93

D

Datenorganisation	436
Depth-Test	104
Detail-Mapping	203
diffuses Licht	138, 348
Direct3D	16
DirectX	456
discard	374
Display-List	206
Dopplereffekt	392
Dopplerfaktor	392
Double-Buffering	37, 51
drawRect	34
Dreieck	92

E

Eulerwinkel	132
Explosion	271
Eye Coordinates	119
Eye-Space Koordinaten	348

F

Far Clipping-Plane	417
Fast Fourier Transform	399
Feuer-Effekt	265
Fixed-Pipeline	311
Flat-Shading	137
flushBuffer	51
Fogging	279
FPS	110
FPS-Counter	111
Fragmente	104
Fragment-Shader	312
Frame-Based-Motion	110
Framebuffer	18
Akkumulation-Buffer	18
Color-Buffer	18
Depth-Buffer	18
Stencil-Buffer	18, 37
Frames per Seconds	424
fread	100
Frustum	122
Frustum-Culling	417
ftransform	326
Fullscreen	45
FXComposer	379

G

GarageBand	382
GetMovieAudioFrequencyLevels	399
Gimbal Lock	132
glActiveTexture	199
GL_ADD	185
GL_ALIASED_POINT_SIZE_RANGE	277
glAlphaFunc	197
GL_ALPHA_TEST	196
Glanzlicht	138, 353
GL_ARRAY_BUFFER	221
glAttachShader	316
glBegin	35, 82
glBeginQuery	427
glBindBuffer	221
glBindTexture	179
glBitmap	243
GL_BLEND	161, 184
glBlendFunc	165
glBufferData	221, 387
glCallList	208
glCallLists	208
GL_CCW	94
GL_CLAMP	186
GL_CLAMP_TO_EDGE	186

glClear	106
glColor3ub	101
glColorMaterial	154
GL_COLOR_MATERIAL	154
GL_COMBINE	185
GL_COMPILE	207
GL_COMPILE_AND_EXECUTE	207
glCompileShader	315
glCreateProgram	316
glCreateShader	315
glCullFace	94
GL_CULL_FACE	94
GL_CW	94
GL_DECAL	184
glDeleteBuffers	224
glDeleteLists	209
glDeleteProgram	316
glDeleteQueries	427
glDeleteShader	315
glDeleteTextures	183
glDepthFunc	104
glDepthMask	106, 269
GL_DEPTH_TEST	104
glDetachShader	317
glDisableClientState	214
glDrawArrays	213
GL_DYNAMIC_DRAW	221
GL_ELEMENT_ARRAY_BUFFER	221
glEnableClientState	213
glEnd	35, 82
glEndList	207
glEndQuery	427
GL_FILL	95
glFlush	35
GL_FOG	279
GL_FOG_COLOR	280
GL_FOG_DENSITE	280
GL_FOG_END	280
GL_FOG_INDEX	280
GL_FOG_MODE	280
GL_FOG_START	280
gl_FragColor	327
GL_FRAGMENT_SHADER	315
glFrontFace	94
glFrustum	122
glGenBuffers	220
glGenLists	206
glGenQueries	426
glGetFloatv	418
glGetQueryObjectiv	427
glGetShaderInfoLog	315
glGetUniformLocation	341
GL_GREATER	197

glHint	170, 280	GLSL.....	312
glIsList.....	207	GLSLang.....	312
GL_LIGHTING.....	145	GLSL Datentypen	329
GL_LIGHT_MODEL_AMBIENT	140	GLSLEditorSample.....	379
GL_LIGHT_MODEL_COLOR		GL_SPOT_CUTOFF.....	152
_CONTROL.....	140	GL_STATIC_DRAW.....	221
glLightModelf	140	GL_STREAM_DRAW	221
GL_LIGHT_MODEL_LOCAL_VIEWER..	140	glTexCoord2f.....	179
GL_LIGHT_MODEL_TWO_SIDED.....	140	gl_TexCoord[i]	358
GL_LINE.....	95	glTexEnv	184
GL_LINE_LOOP.....	90	glTexImage2D	177, 182
GL_LINES.....	86	glTexParameterf.....	182
GL_LINE_SMOOTH	170	glTexParameterf.....	182
GL_LINE_SMOOTH_HINT		GL_TEXTURE_2D	178
GL_NICEST	170	GL_TEXTURE_MAG_FILTER.....	188
glLineStipple.....	87	GL_TEXTURE_MIN_FILTER.....	188
GL_LINE_STIPPLE.....	87	GL_TRIANGLES.....	93
GL_LINE_STRIP.....	90	gluBuild2DMipmaps.....	192
GL_LINE_WIDTH_GRANULARITY	87	Glüheffekt.....	451
GL_LINE_WIDTH_RANGE	87	gluLookAt.....	126
glLinkProgram.....	317	glUnmapBuffer	230
glMapBuffer.....	230, 343	gluOrtho2D.....	122
glMaterialfv	142	gluPerspective	122
GL_MAX_TEXTURE_SIZE.....	183	glUseProgram	318
GL_MODULATE.....	184	glut.....	383
gl_MultiTexCoord0.....	359	GLUT	17
glMultiTexCoord2f.....	200	glVertex.....	61
glNewList.....	207	glVertex3f.....	35
glNormal.....	146	GL_VERTEX_ARRAY	213
GL_NORMALIZE.....	143	glVertexAttrib1f.....	342
glOrtho	122	glVertexPointer	213
Glow-Textur	451	GL_VERTEX_SHADER.....	315
glPixelStorei.....	239	Glyphe.....	236
GL_POINT.....	94		
GL_POINTS.....	83	H	
glPointSize	84	Half-Vektor.....	353
GL_POINT_SIZE_GRANULARITY	84	Hitboxes.....	408
GL_POINT_SIZE_RANGE	84	HLSL	379
GL_POINT_SMOOTH.....	169		
GL_POINT_SMOOTH_HINT		I	
GL_NICEST	169	Immediate-Mode.....	214, 343
GL_POINT_SPRITE.....	277	indexierten Vertex-Arrays.....	218
GL_POINT_SPRITE_ARB	279	Inspector.....	29
gl_Position.....	326	Isotrop.....	193
glPushClientAttrib	239		
GL_QUADS	102	K	
GL_QUAD_STRIP	102	Kamerafahrten	446
glRasterPos2f.....	245	KeyUp-Event.....	96
GL_REPEAT	186	Key-Value	447
GL_REPLACE	184	Kollision.....	404
glShadeModel.....	137		
glShaderSource	315		
GL_SHININESS.....	147		

Kollisionsabfrage	115
Kollisionsprüfung	441
Kreuzprodukt	129

L

Level of Detail	191
Licht	137
Lichtabschwächung	141, 453
Lichtmodels	140
Lichtposition	150
Lightwave	288
Linien	86
Linienmuster	87
Linienstärke	87

M

Materialien	141
Matrizen	62, 330
eigene Matrizen verwenden	72
Einheitsmatrize	63
glPopMatrix	75
glPushMatrix	75
Modelviewmatrix	65
Projektionsmatrix	66
Rotation	64
Skalierung	65
Stapel	75
Texturmatrix	66
Verschiebung	63
Maya	288
Midi	382
Mip-Maps	191
Model-Format	288
ModelView	118
Modelviewmatrix	118, 418
ModelViewProjektionMatrix	327
MP3	398
Multipass-Multi-Texturing	199
Multi-Texturing	198

N

Namesets	330, 360
Near Clipping-Plane	417
Nebel	279, 375
No Recovery	38
NSAutoreleasePool	50
NSBitmapImageRep	177
NSColorWell	340
NSImage	241
NSKeyDown	51

NSMutableArray	108
NSMutableArrayArrays	294
NSOpenGLContext	18
NSOpenGLPixelFormat	134
NSOpenGLPixelFormat	18
NSOpenGLView	18
NURBS	289
NVIDIA	380

O

OBB	406
Occluder	426
Occlusion-Culling	425
Occlusion-Queries	425
OpenAL	15
OpenAL Extensions	394
OpenGL	16, 382
Datentypen	20
Erweiterungen	22
Funktionen	21
Orange-Book	339
Oriented-Bounding-Box	405
Ortho	121
Ortho-Modus	245

P

Partikel	261
Partikeleffekte	160
Partikelsysteme	452
Partikel-Systeme	251, 263
Pentagram	274
Per Pixel Beleuchtung	355, 453
per Vertex Beleuchtung	351
Pixel-Shader	312
plist	447
Point-Sprites	276
Policy	38
Polygon	103
Power-Of-Two-Textur	184
Programm-Objekt	316
Projection	118, 119
Projektionsmatrix	418
Punkt	55, 83
Punktgröße	84

Q

Quaternionen	132
Quicktime	383, 398

R

Rasterierung.....	312
Reflektionen	166
Renderer	37
Software-Renderer.....	37
RenderMonkey	379
Render-Pipeline.....	311
reshape	34
RGB-Farbraum	135
Richtungslicht	348

S

sampler.....	359
Sampling	37
Schockwelle	269
ScrambleX	434
SDK	288
SDL	19
Secondary Color	192
Separating Axis Theorem	415, 440
SetMovieAudioFrequencyMetering	
NumBands.....	399
Shader	310, 453
Shader-Objekte	314
Shared Kontext.....	44
sin	86, 271
Skalar.....	55
Smooth-Shading.....	136
Spirale.....	85
Splines	444
Spot-Light.....	150
State-Machine	16, 311
Stereo Buffer.....	37
Stippling.....	87
strafe	128
Stride-Parameter.....	217

T

Terrain.....	99, 253
Terrains	439
Texel	176
Textsystem	236
Textur-Atlasung	450
Texture-Combiners	185, 368
Texture-Mapping.....	174
Textur-Handle.....	361
Textur-Koordinaten	175, 358
Tiefenpuffer	104
Transformationen	
Reihenfolge.....	70

Rotieren	68
Skalieren	69
Verschieben	67
Treppeneffekt	169
Triangle-Strip	98
Triangulieren.....	92
Trianlge-Fan	98
Typenqualifizierer.....	331

U

uniform	331

V

varying	332
VBO's.....	220
Vektor.....	56
Addition	58
Dotproduct	59
Einheitsvektor	57
Kreuzprodukt	60
Multiplikation	59
Ortsvektor.....	56
Punktprodukt	59
Richtungsvektor.....	57
Vektorlänge.....	57
Vertex-Arrays.....	211
Vertex-Shader	312
Vierecke	102
Viewport	118, 119
Visualisierer.....	402
Volumetrischer Nebel	283

W

wav-Datei.....	385
Wavefront	288
Winding.....	93
Wrapper	54

X

X11	17
Xcode	23
XML	265

Z

Z-Buffer.....	104
Zeichenmodi	94
Zeichnen	82