OpenGL
Modellierung und Beleuchtung

Proseminar Computergrafik

Rene Lützner

Inhanltsverzeichnis:

1 Einleitung
1.1 Begriffserklärung
1.1.1 OpenGL
1.1.2 GLU
1.1.3 Rendern
1.1.4 Rendering-Pipeline
1.2 Historische Entwicklung
1.3 Erweiterbarkeit
1.4 OpenGl als Zustandsautomat
1.5 OpenGL Rendering Pipeline
1.5.1 Darstellungslisten
1.5.2 Evaluators
1.5.3 Per-Vertex Operations
1.5.4 Primitive Assembly
1.5.5 Pixel Operations Texture
1.5.6 Assembly Rasterization
1.5.7 Fragment Operations
1.6 OpenGL Syntax
2 Modellierung
2.1 Löschen / Übermalen des Fensters
2.2 Festlegen der Farbe
2.3 Erzwingen des vollständigen Zeichnens
2.4 Punkte, Linie und Polygone
2.4.1 Punkte
2.4.2 Linen
2.4.3 Polygone
2.4.4 Rechtecke
2.4.5 Kurven
2.5 Spezifizierung von Vertices
2.6 OpenGL Zeichnen vonPrimitiven
2.7 Zustands Management
2.8 Quadrics
2.9 Bezier-Kurven
2.10 Normalen Vektoren
3 Beleuchtung
3.1 Hidden Surface Removal
3.2 Reale and OpenGL Beleuchtung
3.3 Ambientes, Diffuses, and Spekulares Licht
3.4 Festlegen der Material Eigenschaften
3.4.1 Diffuse and ambiente Reflection
3.4.2 Spekulare Reflexion
3.4.3 Emission
3.5 RGB Werte für Licht und Material
3.6 Erstellen einer Lichtquelle
3.7 Aktivieren der Beleuchtung
3.8 Position und Dämpfung
4 Quellen

1. Einleitung

1.1 Begriffserklärung

1.1.1 OpenGL

OpenGL (**Open G**raphics **L**ibrary) bezeichnet eine plattform- und programmiersprachenunabhängige API (Application Programming Interface), die der Entwicklung von 3D-Computergraphiken dient. Der OpenGL-Standard umfasst etwa 250 Befehle, die die Darstellung von komplexen 3D-Szenen in Echtzeit ermöglichen.
Um die Plattformenunabhängigkeit zu gewährleisten, werden von OpenGL keine Befehle zum Erstellen von Fenstern und zur Verarbeitung von Nutzereingaben bereitgestellt, stattdessen muss der Programmierer selbst sich darum kümmern auf dem jeweiligen System diese Dinge umzusetzen.
Auch werden keine „High-Level“ Befehle für die Beschreibung von komplexen 3D Modellen unterstützt, mit denen man Körperteile, Flugzeuge oder Moleküle spezifizieren könnte.
Die gewünschten Formen müssen in OpenGL aus einer kleinen Menge von geometrischen Primitiven – Punkten, Linien und Polygonen erstellt werden.

1.1.2 GLU

GLU (Open**GL U**tility Libary) ist eine Funtionsbibliothek, die auf Open GL aufsetzt und die viele zusätzliche Modellierungsmöglichkeiten bietet, wie Quadric Oberflächen, NURBS Kurven und Oberflächen. Somit ist GLU ein standard Teil jeder OpenGL Implementation.

1.1.3 Rendern

Rendern bezeichnet den Proßess bei dem der Rechner ein Bild des jeweiligen Objektes erzuegt. Diese Objekte, wie Eingangs schon erwähnt bestehen aus geometrischen Primitiven, welche durch Vertices spezifiziert werden.

Das letztendlich zu sehende Bild besteht aus Pixeln, welche auf den Bildschirm angezeigt werden. Informationen zu diesen Pixeln (z.B. die Farbe) werden im Speicher als BitPlanes abgelegt. Ein BitPlane ist ein Speicherfeld das ein Bit an Infomationen für jedes Pixdel auf dem Bildschirm beinhaltet. Dieses Bit kann dann z.B. angeben wie rot das jeweilige Pixel ist. Die BitPlanes selber befinden sich wiederum im FrameBuffer, der alle Infomationen hält die der Grafikbildschirm benötigt um die Farbe und Intensität aller Pixel auf dem Bildschirm zu bestimmen.

1.2 Historische Entwicklung

OpenGL entstand ursprünglich aus dem von Silicon Graphics (SGI) entwickelten Iris GL. Im sogenannten *Fahrenheit-Projekt* versuchten Microsoft und SGI ihre 3D-Standards zu vereinheitlichen, das Projekt wurde jedoch aufgrund finanzieller Schwierigkeiten auf Seiten von SGI abgebrochen. Seit dem 31. Juli 2006 liegt die Weiterentwicklung der OpenGL-API in der Hand der Khronos Group.

Aufgrund seiner Plattformunabhängigkeit ist OpenGL im professionellen Bereich als 3D-Standard nach wie vor führend. Im Bereich der Computerspiele wurde es jedoch in den letzten Jahren zunehmend von Microsofts DirectX 3D verdrängt und hält sich hauptsächlich noch wegen der Beliebtheit der Engines von id Software und der Portabilität auf andere Plattformen. Mit dem Erscheinen von Windows Vista und DirectX 10 (mittlerweile bereits 10.1) könnte sich dies allerdings wieder entscheidend ändern, da die Spieleprogrammierer und Firmen lieber für eine breitere Hard- und Softwarebasis programmieren als für ein paar wenige High-End-Systeme mit entsprechenden Grafikkarten, Prozessoren und Betriebssystemen. OpenGL 2.1 und die geplanten Folgeversionen bieten zumindest die Möglichkeit, den "zwangsweisen" Betriebssystemwechsel zu umgehen, da hier kein künstlicher technischer Schnitt gemacht wurde (wie von Microsoft mit DirectX 10). Derzeit werden aus diesem Grund sämtliche neuen Spiele für DirectX 10 parallel dazu auch noch für DirectX 9.0c programmiert.

1.3 Erweiterbarkeit

Eine wichtige Eigenschaft von OpenGL ist dessen Erweiterbarkeit. Einzelne Anbieter (typischerweise Grafikkartenhersteller) können die Zustandsmaschine von OpenGL um weitere Zustände erweitern. Wenn ein Hersteller eine Erweiterung realisieren möchte, so liefert er eine C-Headerdatei aus, in der er die Erweiterung mit den nötigen Konstanten und evtl. Funktionsprototypen definiert. Die Funktionsnamen und Konstanten erhalten dann einen herstellerspezifisches Postfix. Wenn sich mehrere Hersteller finden und darüber einigen das sie die selbe Erweiterung anbieten möchten, so erhalten die Funktionsnamen und Konstanten den Postfix EXT. Einigt sich schließlich das ARB (Architecture Review Board) darauf, die Erweiterung zu standardisieren, erhalten alle Namen den Postfix ARB. Die meisten vom ARB standardisierten Erweiterungen werden in der folgenden OpenGL-Spezifikation dann in den Core aufgenommen. Das heißt, sie werden Bestandteil von OpenGL selbst und führen ab dann keinen Postfix mehr. Diese von den Graphikkartenherstellern ausgehenden Erweiterungen und die darauf folgenden Standartisierungen erlauben es die neuesten Möglichkeiten der Grafikhardware zu nutzen und dennoch OpenGL abstrakt genug zu halten.

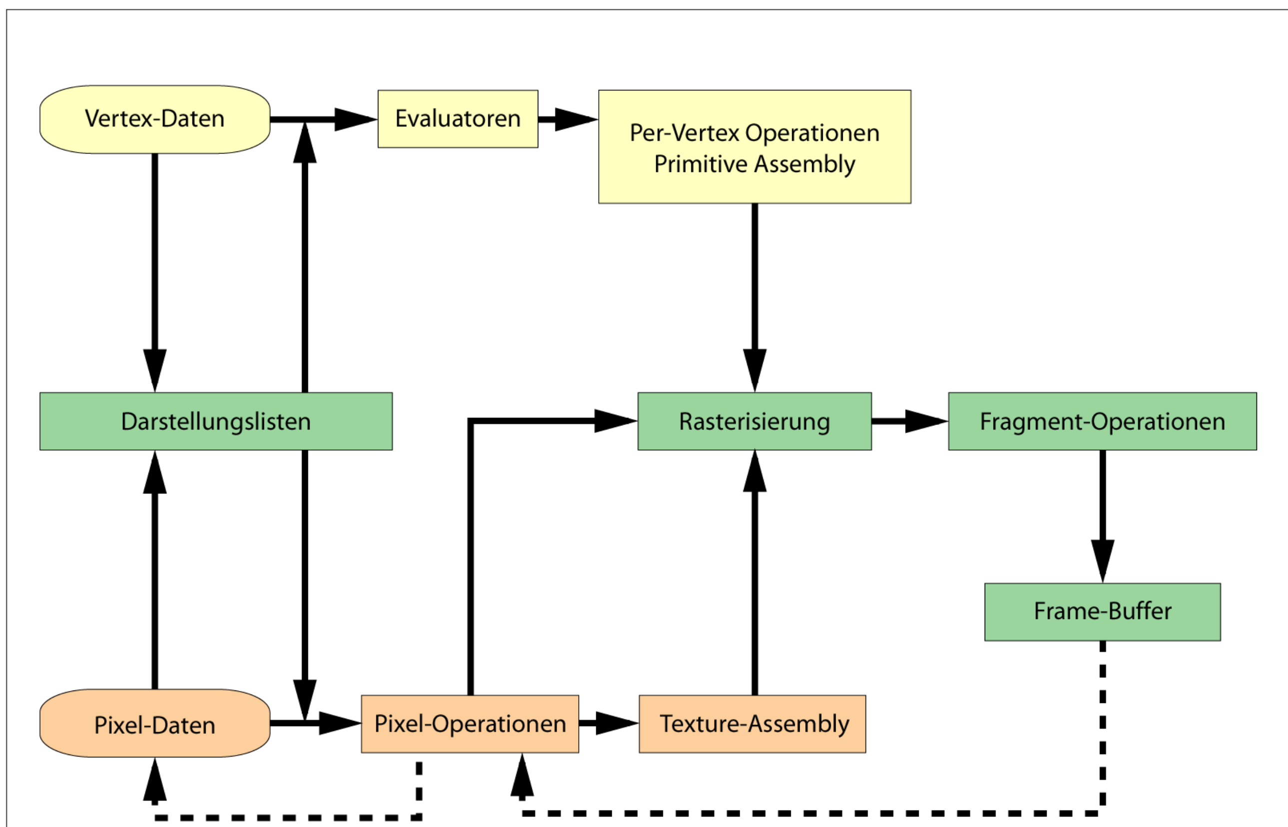
1.4 OpenGL als Zustandsautomat

Die Darstellung von geränderten Objekten ist von vielen Parametern abhänging z.B. vom Licht, Texturen und Oberflächenbeschaffenheiten. OpenGL wurde als Zustandsautomat entworfen, d.h. dass nicht bei jedem Funktionsaufruf alle Parameter übergeben werden, sondern die bereits vorhandenen Werte solange benutzt werden, bis die entsprechenden Zustände geändert wurden. Dies lässt sich z.B. an der Funktion **glColor()** verdeutlichen, mit der man die gerade für einen Vertex zu verwendende Farbe festlegt. Die festgelegte Farbe wird solange benutzt bis die Funktion erneut aufgerufen wird und der Zustand sich ändert. Die momentane Farbe ist somit nur eine von vielen Zustandsvariablen die OpenGL besitzt. Andere bestimmen die Art der gerade gentzten Projektion, Transformation, Polygonen oder der Lichteigenschaften. Viele Variablen verweisen auch auf die Zustände / Modi, die gerade ein oder ausgestellt sind. Das Aktivieren und Deaktivieren dieser Zustände erfolgt durch die Befehle **glEnable()** und **glDisable()**. Jede Zustandsvariable hat einen Default-Wert und zu jedem Zeitpunkt ist es möglich den momentanen Wert vom System zu erfragen. Typischerweise verwendendet man einen der folgenden sechs Befehle **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, oder **glIsEnabled()**. Welchen man davon nimmt hängt vom Typ ab den man als Antwort erwartet. Für manche Zustandsvariablen gibt es spezifischere Befehle wie **glGetLight*()**, **glGetError()**, oder **glGetPolygonStipple()**. Zudem lässt sich auch eine Reihe von Zustandsvariablen mit **glPushAttrib()** oder **glPushClientAttrib()** auf dem Stack ablegen, wenn man sie für eine bestimmten Zeitpunkt modifizieren will und danach wieder die ursprünglichen Werte benötigt, die mit **glPopAttrib()** oder **glPopClientAttrib()** zurückgeladen werden können. Mit Hilfe dieser ZustandsVariablen kann man aufwändige Reorganisationen der Graphikpipeline, die faste jede Änderung des Zeichenmodus nach sich zieht, solange wie möglich vermeiden. Oft können viele tausend Vertices bearbeitet werden, bevor wieder ein Zustand geändert werden muss, während manche Zustände sogar nie geändert werden.

1.5 OpenGL Rendering Pipeline

Eine **Computergrafik-Pipeline,** auch Rendering-Pipeline oder einfach *Grafikpipeline,* ist eine Modellvorstellung in der Computergrafik, die beschreibt, welche Schritte ein Grafiksystem zum Rendern, also zur Darstellung einer 3D-Szene auf einem Bildschirm, durchführen muss. Da diese Schritte sowohl von der Soft- und Hardware als auch von den gewünschten Darstellungseigenschaften abhängen, gibt es keine allgemein gültige Grafikpipeline. Zur Ansteuerung von Grafikpipelines werden üblicherweise Grafik-API`s wie OpenGL oder Direct3D verwendet, die die zugrundeliegende Hardware abstrahieren und dem Programmierer viele Aufgaben abnehmen. Das Modell der Grafikpipeline findet üblichwerweise beim Echtzeitrendern Anwendung. Oft sind hier die meisten Schritte der Pipeline in Hardware implementiert, was besondere Optimierungen ermöglicht. Die Bezeichnung „Pipeline“ wird in einem ähnlichen Sinn wie die Pipeline bei Prozessoren verwendet: die einzelnen Schritte der Pipeline laufen zwar parallel ab, sind jedoch solange blockiert, bis der langsamste Schritt beendet wurde.

Bild 1.2 : Befehlsfolge



1.5.1 Darstellungslisten

Alle Daten, egal ob sie geometrisch oder als Pixel beschrieben sind, können in einer Darstellungsliste, für die sofortige oder spätere Benutzung, gespeichert werden. Die Alternative dazu wäre die sofortige Berechnung der Daten, was als Immediate Mode bezeichnet wird. Wenn eine Darstellungsliste ausgeführt wird, so werden die Daten so von der Anwendung versendet als wären sie im Immediate Mode sofort berechnet worden. Der Vorteil der Darstellungslisten besteht darin, das bereits berechnete Objekte für eine spätere Nutzung abgespeichert werden können, was zu einer Verbeessreung der Performanz führt. Da man einmal die geometrische Berechnugen durchführt, die Änderungen abspeichert und sie danach mehrere Male wieder aufrufen kann ohne die Berechnugen wiederhohl zu müssen.

1.5.2 Evaluators

Alle geometrischen Primitiven werden durch Vertices beschrieben. Parameterisierte Kurven und Oberflächen werden initial durch Kontrollpunkte und polynomielle Funktionen (Bezier-Kurven) beschrieben. Evaluators unterstützen eine Methode die Vertices zu erhalten, welche nötig sind um diese Oberfläche, ausgehend von den Kontrollpunkten, darzustellen. Diese Methode wird auch als polynomiales Mapping bezeichnet und produziert Oberflächen, Normalen, Texturen, Koordinaten und Farben.

1.5.3 Per-Vertex Operationen

Nach dem Auswerten der Kurven liegen nur noch Vertex-Daten vor, für die der nächste Schritt die Per-Vertex Operationen sind, welche diese zunächst transformieren und zwar durch Multiplikation mit der *Modelview-Matrix*. Die Belichtung findet in OpenGL auf Vertex-Ebene statt, das heist es wird zu jedem Vertex oder Polygon ein Normalenvektor angegeben, anhand dessen OpenGL die resultierende Helligkeit des Polygones bestimmt. Diese Lichtberechnungen finden ebenfalls in den Per-Vertex Operationen statt, dabei spielen die Position des Lichtes und die Materialeigenschaften eine wichtige Rolle. Auch wenn Texturen genutzt werden, so müssen deren Koordinaten hier ebenfalls generiert und transformiert werden.

1.5.4 Primitive Assembly

Um die räumlichen Geometriedaten auf einer 2D-Bildfläche darstellen
zu können, müssen sie auf eine Ebene projiziert werden. Hierfür müssen zunächst weitere Transformationen erfolgen, die das Sichtfeld (*Viewing Volume*) der Kamera festlegen. Dies geschieht durch Multiplikation der einzelnen Vertices mit einer weiteren Matrix, der *Projection Matrix*. Durch sie wird die Form und Tiefe des Viewing Volumes festgelegt. In den meisten Fällen hat dieses die Form einer abgeschnittenen Pyramide. Durch diese Form erscheinen weiter entfernte Objekte kleiner, wodurch ein räumlicher Eindruck ensteht. Objekte, die außerhalb des Bildrandes liegen oder sich hinter der Kamera befinden, werden nun entfernt, bzw. abgeschnitten, falls sie noch zu einem Teil im Bild liegen (Clipping). Da in diesem Schritt die Raumkoordinaten durch die Projektionsmatrix lediglich in Kamerakoordinaten umgewandelt wurden, bleiben die Tiefeninformationen noch für spätere Phasen der Pipeline erhalten.

1.5.5 Pixel Operationen

Während die Geometriedaten den Weg durch die OpenGL Rendering Pipeline gehen, nehmen die Pixeldaten einen anderen Weg.
Diese können aus einer Datei geladen werden, sie können aber auch
zur Laufzeit algorithmisch erstellt oder aus dem Framebuffer gelesen werden. Die resultierenden Daten stehen dann als Arrays von Farbwerten im Speicher. Um sie als Textur verwenden zu können, liest OpenGL die Bilddaten, wandelt sie in das richtige Format um und schreibt sie in den Texturspeicher. Hierbei muss es sich nicht zwingend um 2-Dimensionale Texturen handeln, denn OpenGL unterstützt auch 1D- und 3D-Texturen.

1.5.6 Texture Assembly

Texturen können auf geometrische Objekte gemapped werden, damit diese realistischer ausehen. Wenn man mehrere dieser Texturen benutzt, so ist es sinnvoll diese in Textur Objeket, die das vorbereitete Bild, den Namen der Textur und ihre Priorität beinhalten, zu packen, wodurch man dann leicht zwischen ihnen hin und her wechseln kann
Auf diese Texturen kann dann sehr schnell zugegriffen werden, da sie so lange im Texturspeicher behalten werden, bis sie explizit gelöscht oder von einem Texture Object mit höherer Priorität überschrieben werden.

1.5.7 Rasterisierung

Rasterization ist die Umwandlung von Geometriedaten und Pixeldaten in Fragments. Jedes Fragment-Quadrat gehört zu einem Pixel im Framebuffer. In diesem Schritt werden die Vertices zu Linien bzw. Polygonen verbunden,dann werden sie auf das 2D-Punktgitter projiziert. Die Punkte (Fragments) werden entsprechend eingefärbt und bekommen eine Tiefe zugewiesen.

1.5.8 Fragment Operations

Die einzelnen Pixel einer Textur werden als *Texel* bezeichnet.
Wenn nun eine Textur auf ein Objekt gelegt werden soll, muss die Textur entsprechend der Lage des Objekts um Raum skaliert werden. Dann wird die Textur über die Fragments gelegt und diese werden entsprechend der Farbe der zugehörigen Texel eingefärbt. Wenn mehrere Texel auf ein Fragment fallen, wird der Mittelwert ihrer Farben gebildet und diese wird dem Fragment zugewiesen.
Es folgen die Berechnung der Veränderungen der Farbwerte durch Nebel und (falls aktiviert) diverse Tests, beispielsweise der *Scissor Test*, der nur Fragments zulässt, die in einem vom Entwickler angegebenen Rechteck liegen, oder der *Depth Test*. Hiernach stehen die zu zeichnenden Fragments und deren Farbwerte fest. Diese warten nun noch auf weitere ankommende Bilddaten aus der Pipeline, die gegen diese getestet werden. Wenn alle Daten verarbeitet wurden, wird das Bild ausgegeben.

1.6 OpenGL Befehls Syntax

OpenGL Befehle nutzen einen Präfix **gl** und alle Befehle fangen nach diesem Präfix mit einem Großbuchstaben an. Für Konstanten gilt das sie mit **GL_** anfangen und nur Großbuchstaben und Unterstriche benutzt werden. Auch am Ende von einigen Befehlen treten Suffixe auf z.B. bei **glColor3f(),** die **3** steht dafür das drei Parameter übergeben werden und das **f** gibt an das es sich dabei um floating-point Variablen handelt. Eine andere Version von **glColor** nimmt dagegen vier Parameter an. Durch die verschiedenen Funktionen kann OpenGL mehrere Formate von Nutzerdaten anehmen. OpenGL bietet auch eigene Typen an, deren durchgängige Verwendung den Vorteil bietet, dass beim portieren zwischen verschiedenen Implementationen es zu keinen Typ-Mismatches kommen kann. Einige OpenGL Befehle können auch am Ende ein **v** tragen, welches darauf hin weist das der Befehl einen Pointer auf einen Vektor oder eine Menge von individuellen Parametern erwartet. Viele Befehle bieten sowohl eine Variante mit Vektoren als auch eine ohne an. Im Folgenden ein Beispiel dazu:

```
ohne Pointer:

glColor3f(1.0, 0.0, 0.0);

mit Pointer:

GLfloat color_array[] = {1.0, 0.0, 0.0};
glColor3fv(color_array);
```

Als Typ für diese Pointer ist unter OpenGL **GLvoid** festgelegt.

Man kann auch anstatt der spezifischen Funktionsnamen folgen Syntax verwenden **glColor*()** der für alle Variationen dieser Funktion steht. Nach dem Stern lässt sich auch noch ein Suffix angeben um die Variationen noch einzuschränken bzw. zu spezifizieren. **glVertex*v()** z.B. bezieht sich auf alle Varianten die Vektoren nutzen.

2.1 Löschen / Übermalen des Fensters

Da sich auf einem Recher im Speicher meist genau das Bild, welches man zuvor gezeichnet hat, befindet, ist es nötig das alte zu löschen um ein neues Bild zu malen. Dies geschieht mit folgenden zwei Funktionen, die bereits im Beispiel 1.1 erläutert wurden :

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Neben dem Color-Buffer gibt es noch weiter Buffer in OpenGL. Einer weiterer ist der Depth Buffer das Löschen mit:

```
glClearDepth(1.0);
glClear(GL_DEPTH_BUFFER_BIT);
```

führt dazu das alle Werte im Depth Buffer die auf den mit **glClearDepth()** vorher festgelegten Wert zurückgesetzt werden.

Tabelle 2-1 : Löschen der Buffer

Buffer	Name
Color buffer	GL_COLOR_BUFFER_BIT
Depth buffer	GL_DEPTH_BUFFER_BIT
Accumulation buffer	GL_ACCUM_BUFFER_BIT
Stencil buffer	GL_STENCIL_BUFFER_BIT

OpenGL erlaubt auch die Spezifikation von mehreren Buffern gleichzeitig. Das ist besonders dann sinnvoll wenn die Grafik-Hardware ein simultanes Löschen der Buffer umsetzen kann, da das Löschen der Buffer eine relativ langsame Operation ist, bei der für jedes Pixel der Wert geändert werden muss.

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

2.2 Festlegen der Farbe

In OpenGL ist die Beschreibung der Form unabhängig von der der Farbe. Jedesmal wenn ein Objekt gezeichnet wird benutzt es das gerade festgelegte Farb-Schema. Das bedeutet das zuerst das Farbe des jeweiligen Objektes festgelegt wird und dann das Object gezeichnet wird. Falls das nächste Objekt eine andere Farbe bekommen soll, so muss vorher eine neue Farbe festgelegt werden (siehe: Zustandsmaschine). Zum Setzen der Farbe wird der Befehl **glColor3f()** genutzt. Er benötigt drei Parameter: Rot Grün und Blau (RGB) als Floating-Point Wert zwischen 0.0 und 1.0, aus denen sich die eigentliche Farbe ergibt. Für ein rotes Objekt sieht der Befehl wie folgt aus:

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

Aufällig hierbei ist natürlich das der Befehl glClearColor() vier Parameter haben möchte. Bei dem vierten zusätzlichen Parameter handelt es sich um den Alpha-Kanal / Wert der die Transparenz einer best. Farbe angibt und wenn er nicht genutzt wird, standardmäßig den Wert 0.0 (absolut undurchsichtig) zugewiesen bekommt.

2.3 Erzwingen des vollständigen Zeichnens

Die meisten modernen Graphiksysteme kann man sich als eine zusammengesetzte Abarbeitungs-Linie vorstellen. Die CPU gibt einen Zeichnen-Befehl aus und andere noch vorhandene Hardware übernimmt die geometrische Transformation, Clipping, Shading, Texturierung und schließlich das Schreiben in die Bitplane. In High-End Architekturen werden diese Schritte alle von unterschiedlichen Hardwarekomponenten erledigt. Es gibt dort also keinen Grund für die CPU darauf zu warten bis der Zeichnen-Befehl durchgeführt wurde und dann erst den nächsten Befehl auszugeben. In einem solche System würde das Warten der CPU auf den vollständig ausgeführten Befehl zu einem großen Performanz Verlust führen. Deswegen werden Packet übergeben statt nur eines einzelnen Vertexes. Doch CPU weis nicht wann ein Frame vollständig ist bzw. wann alle Daten im Buffer stehen und wenn dieser nicht voll wird, so kann es sein, dass die Daten vielleicht nie abgesendet werden. Um dies zu verhindern gibt es den Befehl **glFlush()** der für ein sofortiges Absenden des Packetes sorgt. Wenn auf einem System natürlich sofort alle Befehle abgearbeitet werden und kein Versenden von Nachrichten bzw. Daten über ein Netzwerk stattfindet, so hat **glFlush()** keinen Effekt. Eine weitere Funtkion die sich ähnlich wie **glFlush()** verhält ist **glFinish(),** welche zusätzlich noch auf eine Rückmeldung von der Graphik-Hardware warte. Beim Synchronisieren von best. Anwendungen kann diese Eigenschaft recht hilfreich sein, wenn man wissen muss, ob sein geründertes Objekt bereits auf dem Bildschirm zu sehen ist.

2.4 Punkte, Linie und Polygone

Die math. Bedeutung von Punkten Linien und Polygonen entspricht recht stark der die sie unter OpenGL haben, jedoch nicht ganz.

Ein Unterschied ergibt sich aus der begrenzten Genauigkeit der Datentypen die auf dem Computer genutzt werden, dadurch ergeben sich dann Rundungsfehler. Darunter leiden natürlich auch die Koordinaten der Punkte usw. unter OpenGL.

Ein weiterer entscheidender Unterschied resultiert aus der Rasterung des Bildschirms. Auf einem Bildschirm ist das kleinste darstellbare Objekt ein Pixel und dessen Größe überschreitet die der math. Definition einens Punktes bei weitem. Punkte werden zwar oft aber nicht immer als einzelne Pixel in OpenGL dargestellt. Mehrere Punkte mit unterschiedlichen Koordinaten können mit ein und dem selben Pixel representiert werden.

2.4.1 Punkte

Ein Punkt wird durch eine Mange von Floating-Point-Zahlen representiert den man Vertex nennt. Alle internen Berechnungen werden so vorgenommen als wären diese Vertices dreidimensional. Vertice welche als zweidimensional festgelegt wurden liegen in der x-y-Ebene und ihre z Koordinate ist immer Null.

2.4.2 Linen

In OpenGL bezieht sich die Bezeichnung Linie auf ein Liniensegment, da es einen Anfang und ein Ende gibt und sie somit nicht unendlich lang ist. Durch die Angabe der Endpunkte lassen sich relativ leicht Folgen von Linien erstellen.

Bild 2.1 : zwei Serien von Liniensegmenten

2.4.3 Polygone

Polygone sind abgeschlossene Bereiche umschlossen von einzelnen Liniensegment-Schleifen. Für gewöhnlich sind die umschlossenen Flächen der Polygone gefüllt. Sie lassen sich aber auch als Umrandungen darstellen. Bezüglich der Komplexität gibt es einige Beschränkungen von OpenGL, es muss sich um primitive Polygone handeln. Als erstes dürfen sich die Ecken des Polygones nicht überschneiden. Zweitens müssen die Polygone Konvex sein, d.h. alle Punkte müssen in einer Ebene und auf einem „Pfad“ bzw. durchgehenden Liniensegment-Folge liegen. Ein Polygon mit einem Loch in der Mitte wäre nicht Konvex da man min. zwei Liniensement-Folgen bräuchte um das Polygon darzustellen. Versucht man dennoch diese Polygone zu füllen können unvorhergesehene Ergebnisse z.B. nur zu Teil gefüllte Polygone dabei herauskommen.

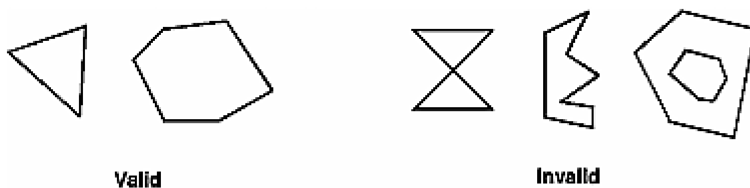


Bild 2.2 : Gültige und Ungültige Polygone

Die Ursache für diese Begrenzung liegt darin, das sich gültige Polygone, die diesen Anforderungen gerecht werden viel schneller Rendern lassen. Letztendlich existieren diese Einschrenkungen also aus Performanz Gründen.

In der realen Welt gibt es jedoch nicht konvexe Polygone, diese lassen sich aber aus einfachen Polygonen zusammensetzten. OpenGL bietet dafür eine Reihe von Funktionen an.

Da OpenGL Vertices immer dreidimensional sind gibt es außer bei Dreiecken keinen Garant dafür, dass alle Punkte, die das Polygon formen in einer Ebene liegen. Selbst durch Transformation und Rotationen kann es dazu kommen das aus einem planaren Polygon ein nicht primitives Polygon ensteht wie in Bild 2.4 zu sehen ist. Dieses Problem lässt sich wie Eingangs erwähnt, durch die Verwendung von Dreiecken vermeiden.

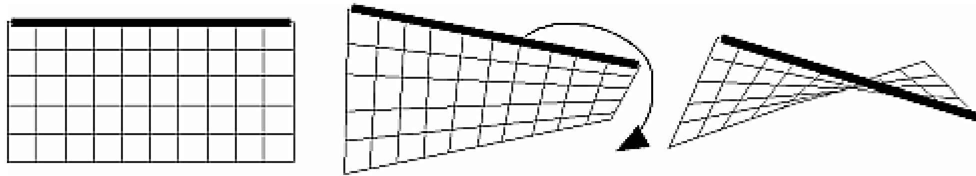


Bild 2.3 : primitives Polygon in ein nicht primitives Polygon tranformieren

2.4.4 Rechtecke

Da Rechtecke recht häufig in graphischen Anwendungen vorkommen, unterstützt OpenGL die Erstellung von gefüllten Primitiven durch **glRect*()** Man kann seine Rechtecke auch als Polygonen zeichenen, jedoch ist diese Funktion unter OpenGL bereits optimiert.

2.4.5 Kurven

Es ist möglich jede Folge von Liniensegmenten an eine geschwungenen Kurvenform beliebiger Genauigkeit zu aproximieren bzw. anzugleichen. Dazu werden die Liniensegmente durch hinzufügen weiterer Punkte in kleinere Segment unterteilt und die neu entstandenen Punkte werden dann auf die Kurvenbahn verschoben. Dies kann man nun so oft wiederholen bis die Abstände zwischen denn Punkten kleiner als ein Pixel sind und somit die best möglich Approximation erreicht ist.

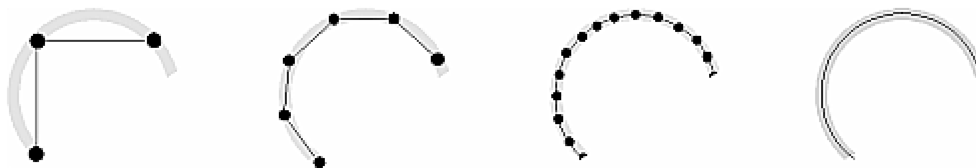


Bild 2.4 : Kurven Aproximation

2.5 Spezifizierung von Vertices

In OpenGL werden alle geometrischen Objekte als Folge von Vertices beschrieben. Man nutzt zur Angabe eines Vertices den Befehl **glVertex*()**.

Beispiele 2.1 : Nutzung von glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

Zum letzten Beispiel ist zu sagen das auf manchen Rechnern die Übergabe eines einzelnen Parameters (Pointer) zum Graphiksystem effizienter geht und wenn man über ein solches System verfügt, sich die Performance steigern ließe durch die Verwendung von Vertex Arrays.

2.6 OpenGL Zeichnen vonPrimitiven

Um ein geometrisches Objekt aus mehreren Vertices aufzubauen muss man diese Vertices innerhalb von **glBegin()** und **glEnd()** angeben, da sie sonst ignoriert würden. Mit **glBegin()** kann auch den Typ des Objektes bestimmen den man mit seinen Vertices erzeugen möchte.

Beispiel 2.2 : gefülltes Polygon

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
glEnd();
```

Bild 2.5 : Zeichnen eines Polygones und einer Menge von Punkten

Wenn man GL_POINTS statt GL_POLYGON nutzt, enstehen lediglich fünf Punkte.

Bild 2.6 zeigt einige Beispiele von geometrische Primitiven. Es gibt oft mehrere Methoden um eine Primitive zu zeichnen, wobei die Wahl der Methode abhängig vom Typ der Vertex Daten ist die man verwendet.

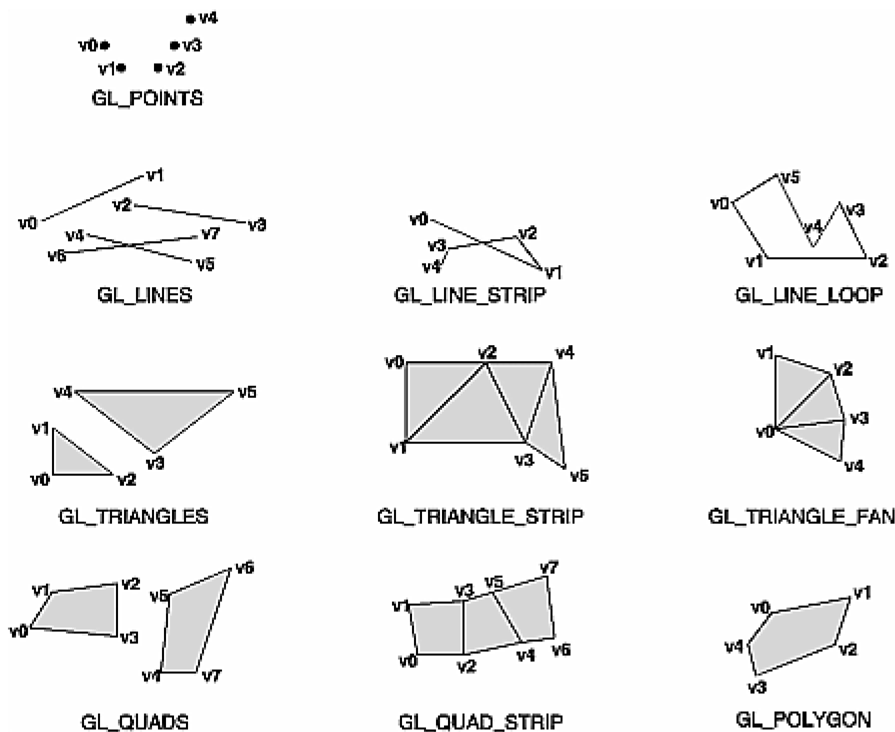


Bild 2.6 : geometrische Primitive

Hier ein kleines Bsp. Für ein OpenGL Programm, welches das in Bild 2.7 zu sehende Rechteck zeichnet

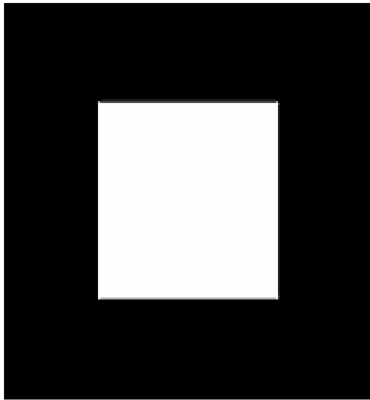


Bild 2.7 : Weißes Rechteck auf schwarzem Grund

```
main() {

   InitializeAWindowPlease();

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
   glFlush();

   UpdateTheWindowAndCheckForEvents();
}
```

Code Erläuterung:

Die erste Zeile der Main() Funktion initialisiert ein Fenster auf dem Bildschirm. Es handelt sich hierbei um einen Platzhalter für systemabhängige, fensterspezifische Funktionen, die nicht zu OpenGL gehören. Bei den nächste beiden Zeilen handelt es sich um OpenGL Befehle. Die an **glClearColor()** übergebenen Parametern geben an in welcher Farbe der Hintergrund übermalt werden soll (in diesem Fall: Schwarz). Mit dem Befehl **glClear()** wird dann der Hintergrund in der vorher mit **glClearColor()** angegebenen Farbe übermalt bzw. „gelöscht“. Wenn die Farbe eimal angegebenen wurde reicht es in Zukunft dann aus nur **glClear()** aufzurufen, um das Fenster in der festgelegten Farbe zu übermalen. Falls man die Farbe wieder ändern möchte ruft man einfach erneut **glClearColor()** auf. Parallel dazu legt **glColor3f()** die Farbe der darauf gezeichneten Objekte fest. Der nächste Befehl im Programm **glOrtho()** bezieht sich auf das Koordinatensystem und gibt an wie das Bild auf den Bildschirm angezeigt wird. Zwischen den Befehlen **glBegin()** und **glEnd()** befindet sich der Code der das zu zeichende Objekt definiert. Die Ecken des Polygones werden mit **glVertex3f()** festgelegt. Wie man vermuten kann handelt es sich bei den Parametern der Funktion um die (x,y,z) Koordinaten. Das Polygon stellt ein Rechteck in der z-Ebene dar. Zum Schluss stellt **glFlush()** sicher das die Zeichen-Befehle auch durchgeführt werden und nicht auf weiter Befehle gewartet wird. Bei **UpdateTheWindowAndCheckForEvents()** handelt es sich dann wieder um einen Platzhalter für Routinen die sich um den Fensterinhalt und die Eventbehandlung kümmern.

2.7 Zustands Managment

Am Anfang wurden schon ein paar Beispiele für Zustandsvariablen, wie die Variable für die gerade aktive RGB Farbe, gezeigt. Als Grundeinstellung sind viele Zustände inaktiv. Um diese zu de-/aktivieren nutz man zwei einfache Funktionen.

void glEnable(GLenum cap);
void glDisable(GLenum cap);

2.8 Quadrics

Kugel, Zylinder und andere „einfache“ geometrische Körper lassen mit der Quadrics Bibliothek mit Hilfe von wenigen Funktionsaufrufen erstellen. Als erstes muss dazu ein Quadric Object erstellt werden mit **gluNewQuadric().** Da hierbei Speicher allokiert wird muss dieses am Ende des Programmes auch wieder gelöscht werden mit **gluDeleteQuadric()**. Falls erforderlich kann man mit **gluQuadricCallback()** und einem Funktionspointer als Parameter auch eine Callback Funktion anmelden um Fehlermeldungen zu erhalten.

Nachdem das Object erstellt ist kann man dessen Eigenschaften festlegen.

Quadric Attribute:

gluQuadricOrientation()

Mit den Parametern GLU_OUTSIDE und GLU_INSIDE, lässt sich einstellen in Welche Richtung die Normalen zeigen sollen.

gluQuadricDrawStyle()

Die Darstellungsart des Objektes kann hier eingestellt werden (Darstellung in Punkten, Linien, als Silhouette oder gefüllte Flächen).

gluQuadricNormals()

Ob Normalen verwendet werden und ob diese pro Vertex oder pro durch mehrere Vertices beschrieben Fläche erstellt werden, lässt sich mit dieser Funktion einstellen.

gluQuadricTexture()

Als Übergabe Parameter sind True und False möglich, mit denen man OpenGL mitteilt ob man ein Textur verwenden möchte.

Zum letztendlichen Zeichnen stehen dann folgende Primitven zur Verfügung: gluSphere(), gluCylinder(), gluDisk(), und gluPartialDisk().

2.9 Bezier-Kurven

Bezier-Kurven gibt es in verschieden Graden für die graphische Darstellung von Flächen z.B. sind vorallem die Kurven 3. Grades interessant. Eine derartige Kurve wird mit vier Kontrollpunkten beschrieben, welche eines der Argumente für die Funktion **glMap1f()** sind mit der man letzlich seine Bezierkurve beschreibt.

Kontrollpolygon für t e [0,1] (Definitionsbereich)

$$C(t)=\sum_{i=0}^{n} B_{i,n} P_n \qquad B_{i,n}(t)=\binom{n}{i} t^i (1-t)^{n-i}$$

Der folgende Beispielcode erzeugt eine derartige Bezierkurve und soll zur Erläuterung der Funktionweise der benötigten Methoden dienen:

```
GLfloat ctrlpoints[4][3] = {
        { -4.0, -4.0, 0.0}, { -2.0, 4.0, 0.0},
        {2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpoints[0][0]);
   glEnable(GL_MAP1_VERTEX_3);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glBegin(GL_LINE_STRIP);
      for (i = 0; i <= 30; i++)
         glEvalCoord1f((GLfloat) i/30.0);
   glEnd();

}
```

Wie im Beispiel zu sehen benötigt **glMap1f()** noch weitere Parameter. GL_MAP1_VERTEX_3 gibt hierbei an das eine Kurve 3. Grades erstellt werden soll, die folgenden Float-Werte repräsentieren die obere und untere Schanke bzw. den Wertebereich für die Laufvariabele t (siehe Kontrollpolygon), der Parameter mit dem Wert 3 gibt an wie die Kontrollpunkte im Array abgelegt sind (hier befinden sich zwischen zwei Kontrollpunkten immer drei Floatwerte) und der Parameter mit dem Wert 4 gibt den Grad des Splines an, der immer genau eins höher ist als der der Kurve. Der Evaluator muss danach aktiviert werden. Es können mehrere Eavluatoren gleichzeitig genutzt werden (z.B. noch einen weiteren um die Farbe des Objektes festzulegen zu können), jeoch kann immer nur ein Evaluator für die Festlegung der Vertices aktiv sein. Das Zeichen an sich erfolgt mit der Funktion **glEvalCoord1f(),** die sich ähnlich nutzen lässt wie glVertex***()** . In einer For-Schleife werden im Beispiel mittels dieser Funktion 30 Linien zu einer Bezierkurve zusammengesetzt.

2.10 Normalen Vektoren

Ein Normalen Vektor oder kurz Normale ist wie im math. Sinne auch ein zu einer Eben senkrecht stehender Vektor. In OpenGL kann man für jedes Polygon oder jeden Vertex eine Normale angeben. Die Normalen eines Objektes legen die Orientierung der Oberfläche im Raum und damit ihre relative Orientierung zur Lichtquelle fest. Diese Vekoren werden in OpenGL dazu genutzt um zu ermitteln wie viel Licht ein Objekt an seinen Vertices bekommt. Die Normalen werden zur selben Zeit definiert wie auch die Geometrie des Objektes festgelegt wird. **glNormal*()** wird genutzt um die Normale zu setzen, danach legt man den Vertex fest, dem dann die Normale zugewiesen wird.

Beispiel 2.3 : Oberflächen-Normalen für Vertexe

```
glBegin (GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

Mittels Kreuzprodukt lassen sich die Normalen für eine Oberfläche ermitteln. Prinzipiell ist die Länge der Normalen egal, jedoch sollte man die Länge vor der Lichtberechnung normalisieren. Die Länge des Vektor ergibt sich wie folgt:

$$\text{Length} = \sqrt{x^2 + y^2 + z^2}$$

Bei gewöhnlichen Transformationen wie einer Rotation ändern sich diese Normalen nicht, führt man jedoch z.B. eine Skalierung, eine Transformation durch Multiplikation mit einer Matrix usw. oder man nutzt nicht normierte Vektoren, so sollte man nach der Transformation die Vektoren normalisieren mit **glEnable()** und GL_NORMALIZE als Parameter. Die autom. Normalisierung ist als Grundeinstellung ausgestellt, da dadurch natürlich zusätzliche Berechnungen anfallen die sich auf die Performance auswirken können.

Ikosaeder

Um zu veranschaulichen wie eine Approximation eines Objektes abläuft, hier ein Beispiel. Wie in der Überschrift abzulesen handelt es sich bei dem Grundkörper um einen Ikosaeder, welcher sich gut dafür eignet um eine Kugel zu approximieren. Zum Zeichen des Ikosaeders selbst kann man sich seine besonderen geom. Eigenschaften zu Nutze machen. Alle zwölf Punkte befinden sich auf Rechtecken deren Seitenverhältnis dem Golden Schnitt entspricht. Diese Rechtecke stehen nun wieder alle senkrecht zueinander.

Beispiel 2.4 : Icosaeder

```
#define X .525731112119133606
#define Z .850650808352039932

static GLfloat vdata[12][3] = {
   {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
   {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
   {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
static GLuint tindices[20][3] = {
   {0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
   {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
   {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
   {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
int i;

void drawtriangle(float *v1, float *v2, float *v3)
{
   glBegin(GL_TRIANGLES);
      glNormal3fv(v1); vlVertex3fv(v1);
      glNormal3fv(v2); vlVertex3fv(v2);
      glNormal3fv(v3); vlVertex3fv(v3);
   glEnd();
}
```

Die seltsamen Werte für X und Z resultieren daraus, das der Abstand der Punkte zum Ursprung eins betragen soll, somit wird letztendlich ein Kreis mit dem Radius eins approxiert. Die Koordianten der zwölf Vertices befinden sich in dem Array vdata[][], Das Array tindices beschreibt wie die Vertice zu verbinden sind damit Dreiecke enstehen. Als Beispiel : das erste Dreieck ergibt sich aus dem nulltem, vierten und ersten Vertex.

Beispiel 2.5 : Berechnung des normalisierten Vektors

```
void normalize(float v[3]) {
   GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
   if (d == 0.0) {
      error("zero length vector");
      return;
   }
   v[0] /= d; v[1] /= d; v[2] /= d;
}
```

Wenn der Ikosaeder zu einer Kugel aproximiert wird, müssen die neu enstandenen Punkte noch auf die Kugel „geschoben“ bzw. projeziert werden. Dies ist in diesem Fall recht einfach da die Kugel den Radius eins hat und somit nur dafür zu sorgen ist, dass alle neu enstanden Punkte den Abstand eins zum Mittelpunkt (hier der Ursprung) haben, was sich schon automatisch durch die Normierung der Vektoren ergibt.

Beispiel 2.6 : Zeichnen und „Aufspitten“ der Dreiecke

```
glBegin(GL_TRIANGLES);

for (i = 0; i < 20; i++) {
      glNormal3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][0]][0]);
      glNormal3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glNormal3fv(&vdata[tindices[i][2]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();

void subdivide(float *v1, float *v2, float *v3, long depth)
{
   GLfloat v12[3], v23[3], v31[3];
   GLint i;

   if (depth == 0) {
      drawtriangle(v1, v2, v3);
      return;
   }
   for (i = 0; i < 3; i++) {
      v12[i] = v1[i]+v2[i];
      v23[i] = v2[i]+v3[i];
      v31[i] = v3[i]+v1[i];
   }
   normalize(v12);
   normalize(v23);
   normalize(v31);
   subdivide(v1, v12, v31, depth-1);
   subdivide(v2, v23, v12, depth-1);
   subdivide(v3, v31, v23, depth-1);
   subdivide(v12, v23, v31, depth-1);
}
```

3. Beleuchtung

Wie in der Einleitung zuvor erwähnt berechnet OpenGL für jedes Pixel am Ende die Farbe. Ein Teil dieser Berechnung hängt vom Licht ab, welches in der Szene verwendet wird. Beleuchtet man z.B. eine weiße Kugel mit rotem Licht so erscheint diese wiederum selbst auch rot. Zudem erkennt man einige 3D-Objekte erst wenn diese beleuchtet werden, da diese sonst als flache Flächen erscheinen, wie dies im Bild 3.1 zu sehen ist.

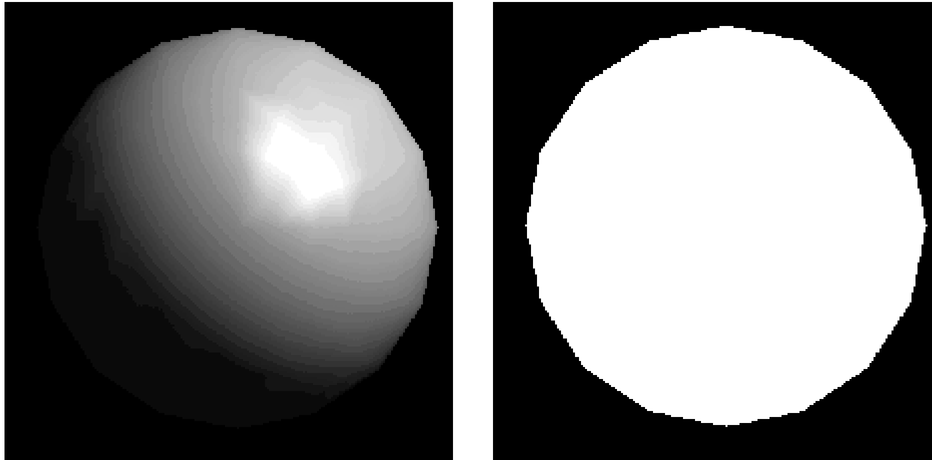


Bild 3.1 : Beleuchtete und Unbeleuchtete Sphere

Mit OpenGL ist möglich die Beleuchtetung und Objekte zu manipulieren und damit die verschiedensten Effekte zu erzeugen.

3.1 Hidden Surface Removal

Mit geshadeten Polygonen wird es wichtig die Objekte die nah an der Betrachterposition sich befinden zu zeichnen und jene, die durch andere näher am Auge befindliche Objekte verdeckt werden, zu beseitigen. Wenn sich jedoch durch Rotation oder andere Transformationen die Betrachterposition bzw. der Blickwinkel ändert, ändert sich auch das Verhältniss bezüglich der Verdeckung. Somit lässt sich das Problem nicht durch eine festgelegte Zeichenreihenfolge der Objekte lösen, da eins immer später als das andere gezeichnet wird. Die Beseitigung von verdeckten Objekten nennt man „Hidden-Surface-Removal“ und der einfachst Weg dies zu errreichen ist die Nutzung des Depth Buffers.

Der Depth Buffer speichert für jedes Pixel die TiefenWerte (also wie weit sich das Pixel vom Betrachtungspunkt entfernt befindet). Initial sind diese Werte alle auf den Maximalwert gesetzt. Das Initialisieren erfolgt wie unter 2. Modellierung erwähnt, durch **glClear()** mit GL_DEPTH_BUFFER_BIT. In diesem Zustand werden die Objekte in beliebiger Reihenfolge gezeichnet. Bevor jedes Pixel auf dem Bildschirm gezeichnet wird, erfolgt zuvor ein Vergleich mit den Werten aus dem Depth Buffer und wenn der neue Pixel näher am Betrachter sich befindet so wird der alte Farbwert vom neuen überschrieben. Wenn das Gegenteil der Fall ist wird der neue Pixel verworfen. Damit dies auch geschieht muss der Depth Buffer aktiviert werden mit **glEnable()** mit GL_DEPTH_TEST. Natürlich hat dieser Depth Test auch wieder Einfluss auf die Performanz der Anwendung. Durch das Beseitigen der verdeckten Flächen kann die Performanz sich verbessern, wirklich entscheidend ist jedoch wie der Depth Buffer umgesetzt ist, ob in Hard- oder in Software.

3.2 Reale and OpenGL Beleuchtung

Wenn man ein reales Objekt betrachtet, dann hängt die wahrgenommen Farbe, von der vom Objekt zurückgeworfenen Lichtstrahlen bzw. Wellenlängen des Lichtes, die dan auf die entsprechenden Sehzellen des Auges treffen, ab. Diese Lichtstrahlen oder auch Photonen stammen von einer oder von meheren Lichtquellen. Einige von ihnen werden reflektiert und andere werden absorbiert. Die Oberfläche des angestrahlten Objektes kann ganz unterschiedliche Eigenschaften haben, so können die Lichtstrahlen alle in eine bestimmte Richtung zurückgeworfen werden oder in alle Richtungen verteilt werden.

OpenGL approximiert Licht und Beleuchtung als etwas das sich in rote, grüne und blaue Komponenten aufsplitten lässt. Dieses Lichtmodell ist relativ nahe am realen, wenn man jedoch ein genaueres Modell braucht, so muss man es selber schreiben.

Im Lichtmodell von OpenGL kommt Licht von mehreren Lichtquellen die individuell aus und ein geschaltet werden können. Ein Teil des Lichtes kommt von best. Positionen, ein anderer Teil ist über die ganze Szene „verteilt". Als Beispiel wäre eine Lampe geeignet welche selbst als Lichtquelle vorhanden ist und in eine bestimmte Richtung das Licht ausstrahlt und das Licht welches wiederum von den Wänden zurückgeworfen wird und sich überall hin verteilt ohne das man die Quelle des Lichtes erkennen kann. Jedoch verschwindet es, wenn seine ursprüngliche Lichtquelle ausgestellt wird.

In OpenGL hat eine Lichtquelle nur dann einen sichtbaren Effekt, wenn es eine Oberfläche gibt, die das Licht absorbiert und reflektiert. Jede Oberfläche verfügt über best. Materialeigenschaften. Ein Material kann selbst Licht emitieren, auftreffendes Licht streuen oder wie ein Spiegel in eine best. Richtung reflektieren.

Das OpenGL Lichtmodell teilt die Beleuchtung in vier unabhägige Gruppen ein : emissives, ambientes, diffuses, and spekulares Licht. Alle vier Komponenten werden seperat berechnet und dann anschließend zusammengesetzte.

3.3 Ambientes, Diffuses, and Spekulares Licht

Ambientes Licht ist das Licht, welches im ganzen Raum verstreut auftritt und dessen Richtung nicht feststellbar ist. Wenn Ambientes Licht auf eine Oberfläche trifft so wird es genauso in alle Richtungen getstreut.

Diffuses Licht kommt von einer bestimmten Richtung, somit ist es heller wenn es genau auf eine Oberfläche trifft, als wenn es ein Objekt nur schimmernd streift. Auch dieses Licht wird beim auftreffen auf eine Oberfläche in alle Richtungen gestreut.

Spekulares Licht kommt ebenfalls von einer best. Richtung und wird eine best. Richtung zurückgeworfen. Ein Laser der auf einen hochwertigen Spiegel trifft produziert fast 100% speculare Reflexion. Glänzendes Metall oder Plastik haben einen hohen Anteil an spekularen Licht und Kreide hat nahezu keinen. Unter spekularem Licht kann man sich so etwas wie Glanz vorstellen.

Zudem kann das Licht sich ganz unterschiedlich verhalten, z.B. erscheint das weiße Licht welches von roten Wänden zurückgeworfen wird rot und wenn es ein anderes Objekt direkt trifft erscheint dessen Glanzpkunkt weiß.

OpenGL erlaubt es die Rot, Grün und Blau Werte für jeden Lichttyp sepperat zu setzen.

3.4 Festlegen der Material Eigenschaften

Das OpenGL BeleuchtungsModell geht davon aus das die Farbe eines Materials davon abhängt wie viel Prozent des darauf treffenden roten, grünen und blauen Lichtes reflektiert werden. Ein perfekter roter Ball reflektiert das gesamte einstrahlende rote Licht und absorbiert das grüne und blaue. Dieser Ball erscheint sowohl unter weißem als auch roten Licht rot, unter grünem Licht, welches er vollständig absorbiert, jedoch schwarz. Wie das Licht selbst auch verfügen Materialien über verschiedene ambiente, diffuse und spekulare Farben, welche die dazu gehörigen Reflexionen des Materials bestimmen. Die ambiente Reflexion des Materials wird kombiniert mit den Ambienten Komponenten der Lichtquellen, welche das Material beleuchten.

Das Festlegen der Materialeigenschaften verläuft fast genauso wie das der Lichtquellen. Der Mechanismus zum Setzen ist der gleiche mit der Außnahme das der Befehl **glMaterial*()** lautet.

3.4.1 Diffuse and ambiente Reflektion

Die GL_DIFFUSE und GL_AMBIENT Parameter, die durch **glMaterial*()** gesetzt werden, beeinflussen die diffuse und ambiente Reflexion des Objektes. Dabei spielt die Diffuse Reflexion die wichtigste Rolle in der Bestimmung der Objektfarbe. Sie hängt stark von der Farbe des einfallenden diffusen Lichtes so wie vom Winkel indem es auftrifft ab. Die Position des Betrachtungspunktes beeinflusst dagegen die diffuse Reflexion nicht.

Die ambiente Reflexion beeinflusst die gesamte Farbe des Objektes. Da die diffuse Reflexion dort am hellsten ist, wo das Objekt direkt bestrahlt wird, ist die ambiente Reflexion dort besonders wahrzunehmen wo es keine direkte Beleuchtung gibt. Die vollständige ambiente Reflexion des Objektes wird durch das globale ambiente Licht und das ambiente Licht von anderen individuellen Lichtquellen bestimmt. Auch die ambiente Reflexion ist nicht vom Betrachtungspunkt abhängig.

In der realen Welt besitzen diffuse und ambiente Reflexion normalerweise ein und die selbe Farbe. Aus diesem Grund unterstützt OpenGL einen geeigneten Weg um beiden Reflexionen gleichzeitig die Werte zuzuweisen.

```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
             mat_amb_diff);
```

In diesem Beispiel vertritt die RGBA Farbe (0.1, 0.5, 0.8, 1.0) – eine dunkel blaue Farbe – die ambiente und diffuse Reflexion für die vorderen und hinteren Polygone.

3.4.2 Spekulare Reflexion

Spekulare Reflexion erzeugt Reflexionspunkte. Anders als ambienter und diffuser Reflexion hängt die Menge des reflektierten Lichtes von Betrachtungspunkt ab. Es ist am hellsten entlang des direkten Reflexionswinkels. Um sich das vorstellen zu können, betrachte man eine metallische Kugel draußen im Sonnenlicht. Sobald amn den Kopf bewegt, bewegt sich dere Reflexionspunkt mit. Wenn man den Kopf jedoch zu weit bewegt wird der Reflexionspunkt verschwinden.

OpenGL erlaubt es solche Effekte zu setzen und auch die Größe und Helligkeit (GL_SHININESS) zu kontrollieren. Für die GL_SHININESS lässt sich ein Wert zwischen 0.0 und 128.0 angeben, je höher der Wert desto kleiner und heller ist der Reflexionspunkt.

```
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
```

3.4.3 Emission

Mit der Angabe der RGBA Farbe für GL_EMISSION, kann amn ein Objekt so erscheinen lassen als würde es selbst Licht aussenden. Es wird meistens dazu benutzt um Lampen und andere Lichtquellen zu simulieren, da es ansonsten nur wenige selbst leuchtende Objekte in der Realität gibt.

Eine Kugel mit grauem emissions Licht:

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

Die Kugel scheint zwar zu glimmen, jedoch verhält sie sich nicht wie eine Lichtquelle, um dies zu erreichen ist es notwendig eine Lichtquelle an der selben Position zu erstellen.

3.5 RGB Werte für Licht und Material

Die Spezifikation der Farbkomponenten für das Licht unterscheidet sich etwas von der für die Materalien. Für das Licht geben die Zahlen die Intensität an, z.B. wenn alle Werte 1.0 betragen entsteht weißes Licht mit voller Intensität und wenn alle Werte jedoch 0.5 betragen, so ist das Licht immer noch weiß, jedoch nur bei halber Intensität und erscheint somit grau.

Bei Materialien stehen die Zahlen für die Reflexionsverhältnisse dieser Farben. Wenn also für ein Material gilt R=1, G=0.5, und B=0, so reflektiert das Material alles auftreffend rote und ie Hälfte des auftreffenden grünen Lichtes und absorbiert das gesamte blaue Licht. Mit anderen Worten, wenn ein OpenGL Licht folgende Komponenten(LR, LG, LB) besitzt, und das Material hat diese Komponenten (MR, MG, MB), so folgt wenn man alle anderen Reflexionen vernachlässigt, dass das Licht welches beim Betrachter am Auge ankommt, sich folgender Maßen ergibt:

(LR*MR, LG*MG, LB*MB).

Genauso gilt für zwei Lichtquellen, von denen Licht mit folgenden Komponenten vom Betrachter wahrgenommen werden (R1, G1, B1) und (R2, G2, B2), das die Komponenten addiert werden (R1+R2, G1+G2, B1+B2). Falls eine der Summen größer als eins ist, so wird diese Komponente auf eins beschränkt.

3.6 Erstellen einer Lichtquelle

Lichtquellen besitzen eine Reihe von Eigenschaften, wie Farbe, Position, und Ausrichtung. Der Befehl der zum Erstellen einer Lichtquelle genutzt wird ist **glLight*()**, welcher drei Parameter benötigt. Dabei handedlt es sich um das Licht wessen Eigenschaften geändert werden sollen, dann welche Eigenschaft und schließlich den gewünschten Wert für die Eigenschaft bzw. einen Pointer auf ein Array indem sich die jeweiligen Werte dann befinden.

3.7 Aktivieren der Beleuchtung

In OpenGL müssen die Lichter explizit aktiviert werden. Wenn die Beleuchtung nicht aktiviert ist werden die Farben einfach den jeweiligen Vertex zugewiesen und es finden keinerlei Lichtberechnungen statt. Die Beleuchtung wird wie folgt aktiviert:

```
glEnable(GL_LIGHTING);
```

3.8 Position und Dämpfung

Es ist möglich eine Lichtquelle zu erzeugen die sich weit entfernt von der Szene befindet oder eine die sich relativ nahe an der Szene befindet. Beim ersten Fall handelt es sich um eine direktionale Lichtquelle, d.h. die Lichtquelle ist soweit entfernt, dass die Lichtstrahlen fasst parallel zueinander verlaufen, wenn sie das Objekt erreichen. In der realen Welt ist die Sonne dafür ein gutes Beispiel. Der zweite Fall wird als positionale Lichtquelle bezeichnet, bei dem die konkrete Position in der Szene und die Richtung aus der die Lichtstrahlen kommen, den Effekt, den die Lichtquelle in der Szene erzeugt, bestimmen. Eine Tischlampe wäre hierfür ein passendes Beispiel. Bei dem Licht, das in Beispiel 3.1 verwendet wird handelt es sich um ein direktionales Licht.

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Wie oben zu sehen ist gibt man für die Position des Lichtes einen Vektor aus vier Variablen (x,y,z,w), sowie GL_POSITION als zu verändernde Eigenschaft an. Die standard Position für Licht0 ist (0,0,1,0), das Licht verläuft dann also entlang der negativen z Achse. Wenn w verschieden von Null ist so geben die Werte die Position des Lichtes in homogenen Objekt-Koordinaten und nicht die Ausrichtung an.

Es kann bei großen Polygonen passieren, dass wenn sich die Lichtquelle zu nahe an diesem befindet, die Vertices zu weit entfernt sind und damit das Polygon dunkeler erscheint als gewollt. Die Urasche dafür darin, dass die Farbe des Polygons über die Farbe der Vertice definiert wird. Um dieses Problem zu beseitigen reicht es, das vorhandene Polygon in kleinere aufzuspalten.

In der realen Welt nimmt die Lichtintensität mit zunehmender Entfernung ab. Da ein direktionales Licht sehr weit entfernt ist es nicht besonders sinnvoll, dieses über die Entfernung zu dämpfen, weswegen die Dämpfung bei solchen Lichtern ausgestellt ist. Bei positionalen Lichtern ist das jedoch anders und so ergibt sich dort die Lichtintensität, indem die Werte der Lichtquelle mit einem Dämpfungsfaktor Multipliziert werden, der sich wie folg ergibt:

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

d = Distanz zwischen Lichtquelle und Vertex

kc = GL_CONSTANT_ATTENUATION

kl = GL_LINEAR_ATTENUATION

kq = GL_QUADRATIC_ATTENUATION

Im Grundzustand ist *k*c 1.0 und *k*l und *k*q sind 0.0, diese Werte lassen sich wie die anderen Eigenschgaften auch verändern:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Abgesehen vom emissiven und globalen ambienten Licht werden alle anderen Lichttypen gedämpft. Diese Lichtdämpfungen benötigen natürlich wieder zusätzlichen Berechnungen, die wiederum die Performanz der Anwendung beeinflussen.

Beispiel 3.1 zeigt wie man **glLight*()** nutzt:

Beispiel 3.1 : Festlegen der Farbe und der Position

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Wie man sehen kann werden Array´s definiert als Parameter für den Befehl **glLightfv().** In diesem Beispiel sind die ersten drei Funktionsaufrüfe redundant, da sie dafür genutzt werden die standard Werte zu setzen. Jedes Licht muss mit **glEnable()** aktiviert werden.

4. Quellen

http://www.glprogramming.com/red/

http://nehe.gamedev.net/

http://de.wikipedia.org/wiki/OpenGL

http://de.wikipedia.org/wiki/B%C3%A9zierkurve

http://www-users.rwth-aachen.de/Tobias.Weyand/OpenGL//OpenGL-Ausarbeitung.pdf

http://graphics.cs.uni-sb.de/Courses/ws9900/cg-seminar/Ausarbeitung/Philipp.Walter/openglstandard.htm

http://www.icare3d.org/myprojects/3d_gallery/opengl_3d_logo.html