GL Galore 1.1

Super smooth OpenGL® scripting with Hollywood

Andreas Falkenhahn

Inhaltsverzeichnis

1 Allgemeine Information

1.1 Einführung

GL Galore ist ein Plugin für Hollywood, mit dem Sie direkt von Hollywood auf den OpenGL® 1.1-Befehlssatz zugreifen können. Dies ermöglicht es, Skripte zu schreiben, die die 3D-Hardware des Host-Systems verwenden, um leistungsstarke 2D- und 3D-Animationen zu erstellen, die vollständig in der Hardware der GPU Ihrer Grafikkarte berechnet werden. Dies führt zu einer enormen Leistungssteigerung gegenüber der klassischen Hollywood-Grafik-API, die größtenteils in der Software implementiert ist. Insbesondere Systeme mit langsameren CPUs profitieren stark vom hardwarebeschleunigten Zeichnen, das von OpenGL angeboten wird.

OpenGL ist eine portable Software-Schnittstelle zur Grafikhardware. Es ist für fast jede Plattform in verschiedenen Versionen erhältlich. Auf AmigaOS und Kompatibelen ist OpenGL als MiniGL auf AmigaOS 4, TinyGL auf MorphOS, StormMesa auf AmigaOS 3 und Mesa 3D auf AROS verfügbar. Windows-, Mac OS X- und Linux-Systeme werden normalerweise mit einem bereits installierten OpenGL-Treiber ausgeliefert. Weitere Informationen zu OpenGL erhalten Sie unter http://www.opengl.org. Sie können gute Tutorials über das Erlernen von OpenGL im Internet finden.

Es gibt zwei Möglichkeiten, GL Galore zu verwenden: Sie können entweder direkt auf die OpenGL 1.1-API zugreifen oder Sie können die Hardwarepinsel-Befehle von Hollywood verwenden, ohne direkte Aufrufe der OpenGL-API vorzunehmen. Wenn GL Galore aktiviert ist, werden Hollywood-Hardwarepinsel direkt auf OpenGL-Texturen abgebildet, so dass sie auf allen unterstützten Systemen sehr schnell gezeichnet und transformiert werden können. Dies ist besonders nützlich unter Windows, Mac OS X und Linux, da Hollywood standardmäßig keine doppelten Hardware-Puffer und Pinsel auf diesen Plattformen unterstützt. Mit GL Galore können jetzt aber auch Hardware-Doppelpuffer und Pinsel auf diesen Plattformen eingesetzt werden. So kann GL Galore hier auch als ein Helfer-Plugin fungieren, das diese Funktionalität zu Hollywood hinzufügt, ohne dass Sie eine einzige Zeile OpenGL-Code schreiben müssen, um es zu benutzen!

Darüber hinaus bietet GL Galore Wrapper-Funktionen für die meisten Befehle der OpenGL 1.1-API. Diese Befehle werden direkt mit wenigen oder gar keinen Änderungen ihrer ursprünglichen Syntax umschlossen. Die einzige Ausnahme betrifft OpenGL-Befehle, die einen Zeiger erwarten: In diesem Fall bietet GL Galore normalerweise eine Variante des Befehls an, so dass er mit Hollywood-Tabellen funktioniert. Die ursprüngliche Zeigervariante ist jedoch auch in GL Galore verfügbar und kann für zeitkritische Skripte verwendet werden. Darüber hinaus bietet GL Galore auch einige Überbrückungs-Funktionen, mit denen Sie Hollywood-Pinsel in OpenGL-Texturen konvertieren können und umgekehrt, um OpenGL-Code zu nutzen!

GL Galore kann auch nützlich sein, um in OpenGL geschriebene Software schnell zu entwickeln. Leute, die OpenGL mit C programmiert haben, werden Hollywoods komfortable Multimedia-API, die Befehle für fast alle gängigen Aufgaben bietet, sehr zu schätzen wissen. Wenn Sie zum Beispiel GL Galore verwenden, um OpenGL-Programme zu schreiben, können Sie den ganzen Aufwand vermeiden, ein GL-Fenster mit einem der vielen verschiedenen Toolkits zu verwalten. Außerdem werden Aufgaben wie Bilder laden, Ton- oder Vi-

deowiedergabe, Schriftbearbeitung und Bildbearbeitung dank Hollywoods mächtigem Befehlssatz, der fast 700 Befehle umfasst, lächerlich einfach.

GL Galore verwendet die neue Plugin-Schnittstelle für den Bildschirmadapter, die mit Hollywood 6.0 eingeführt wurde. Daher funktioniert das Plugin nicht mit älteren Versionen von Hollywood. Es erfordert mindestens Hollywood 6.0. Wenn GL Galore aktiviert ist, werden alle Grafikausgaben automatisch über OpenGL weitergeleitet. Um von der Hardwarebeschleunigung zu profitieren, müssen Hollywood-Skripte jedoch einigen Regeln folgen, die in diesem Handbuch beschrieben sind.

GL Galore enthält umfangreiche Dokumentationen in verschiedenen Formaten wie PDF, HTML, AmigaGuide und CHM, die eine vollständige OpenGL-Referenz und Informationen zu speziellen Befehlen in GL Galore enthalten. Darüber hinaus sind viele Beispielskripte im Distributionsarchiv enthalten, um wirklich schnell loslegen zu können.

All das macht GL Galore zum ultimativen OpenGL-Scripting-Erlebnis, bei dem das Beste aus beiden Welten in einem leistungsstarken Plug-in kombiniert wird: Hollywoods umfangreiches und praktisches Multimediafunktionsset und OpenGLs rohe Grafikleistung!

1.2 Lizenzbestimmungen

FÜR DIESES PROGRAMM GIBT ES KEINE GARANTIE, SOWEIT ES DIE ANZUWENDENDEN GESETZE ZULASSEN. SOFERN ANDERSWO NICHTS GEGENTEILIGES GESCHRIEBEN STEHT STELLEN DER AUTOR UND/ODER DRITTE DAS PROGRAMM "SO WIE ES IST" ZUR VERFÜGUNG, OHNE IRGENDEINE GARANTIE, WEDER DIREKT NOCH INDIREKT. DIES BEINHALTET, IST ABER NICHT DARAUF BESCHRÄNKT, VERKÄUFLICHKEIT UND EIGNUNG FÜR EINEN BESTIMMTEN VERWENDUNGSZWECK. DAS VOLLSTÄNDIGE RISIKO DER QUALITÄT UND AUSFÜHRBARKEIT DES PROGRAMMS LIEGT BEIM ANWENDER. SOLLTE SICH DAS PROGRAMM ALS DEFEKT HERAUSSTELLEN, LIEGEN ALLE KOSTEN FÜR SERVICE, INSTANDSETZUNG ODER NACHBESSERUNG BEIM ANWENDER.

KEIN COPYRIGHT-INHABER ODER DRITTER, DER DAS PROGRAMM WIE OBEN ERLAUBT WEITERVERKAUFT, KANN FÜR SCHÄDEN IRGENDWELCHER ART HAFTBAR GEMACHT WERDEN (DIES BEINHALTET, IST ABER NICHT BESCHRÄNKT AUF, DATENVERLUST INFOLGE UNFÄHIGKEIT DES PROGRAMMS, MIT ANDEREN PROGRAMMEN ZUSAMMENZUARBEITEN), SELBST WENN EIN SOLCHER INHABER ODER DRITTER AUF DIE MÖGLICHKEIT EINES SOLCHEN SCHADENS HINGEWIESEN WURDE, AUSSER ES BESTEHT EINE SCHRIFTLICHE EINWILLIGUNG ODER WIRD VOM GESETZ VERLANGT.

1.3 Anforderungen

- Hollywood 6.0 oder besser
- Windows: Erfordert mindestens Windows 2000
- Mac OS X: Erfordert mindestens 10.5 auf PowerPC oder 10.6 auf Intel Macs
- MorphOS: Benötigt TinyGL mit MorphOS 3.8 oder besser
- AmigaOS 3: Erfordert StormMesa, 68040 oder 68060 und eine FPU
- AmigaOS 4: Benötigt MiniGL
- AROS: Benötigt Mesa 3D

1.4 Installation

Die Installation von GL Galore ist unkompliziert und einfach: Kopieren Sie einfach die Datei `glgalore.hwp` für Ihre Plattform in Hollywoods Plugin-Verzeichnis. Auf allen Systemen außer auf AmigaOS und kompatiblen Systemen müssen Plugins in einem Verzeichnis mit dem Namen `Plugins` gespeichert werden, das sich im selben Verzeichnis wie das Hauptprogramm von Hollywood befindet. Auf AmigaOS und kompatiblen Systemen müssen Plugins stattdessen in `LIBS:Hollywood` installiert werden. Unter Mac OS X muss sich das Verzeichnis `Plugins` im Verzeichnis `Resources` des Programmpakets befinden, d.h. im Verzeichnis `HollywoodInterpreter.app/Contents/Resources`. Beachten Sie, das dass `HollywoodInterpreter.app` im Programmpaket `Hollywood.app` selbst gespeichert ist, nämlich in `Hollywood.app/Contents/Resources`.

Kopieren Sie anschließend den Inhalt des Ordners `Examples` in den Ordner `Examples`, der Teil Ihrer Hollywood-Installation ist. Alle GL Galore-Beispiele werden dann in Hollywoods GUI angezeigt und Sie können sie bequem über die Hollywood-GUI oder die IDE starten und anzeigen.

Unter Windows sollten Sie auch die Datei `GLGalore.chm` in das Verzeichnis `Docs` Ihrer Hollywood-Installation kopieren. Dann können Sie die Online-Hilfe aufrufen, indem Sie F1 drücken, wenn sich der Cursor in der Hollywood IDE über einer GL Galore-Befehl befindet.

Kopieren Sie unter Linux und Mac OS das Verzeichnis `GLGalore`, das sich im Verzeichnis `Docs` des GL Galore-Distributionarchivs befindet, in das Verzeichnis `Docs` Ihrer Hollywood-Installation. Beachten Sie, dass sich das `Docs`-Verzeichnis unter Mac OS im Programmpaket `Hollywood.app` befindet, d.h. `Hollywood.app/Contents/Resources/Docs`.

2 Über GL Galore

2.1 Danksagungen

GL Galore wurde von Andreas Falkenhahn geschrieben. Zusätzliche Codierung von Fabio Guerra, Cleyde Marlyse und Antonio Scuri. Die Arbeit an diesem Projekt wurde im Januar 2014 als einen Nachweis-von-Konzept-Demonstration der leistungsfähigen neuen Display-Adapter-API von Hollywood 6.0 gestartet, die es Plugins ermöglicht, Hollywoods gesamten Display-Handler zu übernehmen und durch einen benutzerdefinierten Treiber zu ersetzen. Es wurde dann sukzessive zu einem vollständigen Wrapper für die API OpenGL 1.1 erweitert, inklusive einiger Überbrückungs-Funktionen zwischen Hollywood und OpenGL sowie einem Hardwarepinsel und Doppel-Puffer-Treiber für alle Plattformen, die nicht von Hollywoods eingebautem Hardwarepinsel und Doppel-Puffer-Treiber unterstützt werden.

Ein Dank geht an Frank Mariak, Mark Olsen, Hans de Ruiter, Frank Wille, Krzysztof Smiechowicz, und Sam Jordan.

Ein besonderer Dank geht an Helmut Haake und Dominic Widmer für die Übersetzung des Handbuchs ins Deutsche. Fehler oder Verbesserungsvorschläge bzgl. des deutschen Hollywood-Handbuchs bitte an das Übersetzungsteam richten, welches unter `handbuch@gmx.ch` oder unter `https://amiga-resistance.info` erreicht werden kann.

Wenn Sie mich kontaktieren möchten, können Sie entweder eine E-Mail senden an `andreas@airsoftsoftwair.de` oder benutze das Kontaktformular auf `http://www.hollywood-mal.com`.

2.2 Häufig gestellte Fragen

In diesem Abschnitt werden einige häufig gestellte Fragen behandelt. Bitte lesen Sie sie zuerst, bevor Sie in der Mailingliste oder im Forum nachfragen, da Ihr Problem hier möglicherweise behandelt wurde.

Q: Warum meldet GL Galore alle Tastaturereignisse in Großbuchstaben?

A: Das ist eine Einschränkung von GLFW, die von GL Galore verwendet wird. Heutzutage kann die Tastaturüberwachung nicht zwischen Groß- und Kleinschreibung von Tasten unterscheiden, wenn sie rohe Tastaturüberwachung verwenden. Siehe Abschnitt 2.3 [Bekannte Probleme], Seite 6, für Details. Wenn Sie Hollywood 7.0 oder höher haben, können Sie einfach den Ereignis-Handler `VanillaKey` überwachen, um die rohen Tastaturereignisse mit voller Unicode-Unterstützung zu erhalten.

Q: Warum werden die Y- und Z-Tasten auf deutschen Tastaturen getauscht?

A: Das liegt daran, dass die rohe Tastaturüberwachung von GLFW auf dem US-Tastaturlayout basiert. Siehe Abschnitt 2.3 [Bekannte Probleme], Seite 6, für Details. Wenn Sie Hollywood 7.0 oder höher verwenden, können Sie stattdessen den `VanillaKey` Ereignis-Handler überwachen. Dadurch erhalten Sie echte Tastaturereignisse mit voller Unicode-Unterstützung.

Q: Wie beende ich Skripts, die im Vollbildmodus ausgeführt werden?

A: Drücken Sie einfach STRG+C. Dies funktioniert immer, außer wenn STRG+C explizit mit dem Hollywood-Befehl `CtrlCQuit()` deaktiviert wurde.

Q: Gibt es ein Hollywood-Forum, in dem ich mit anderen Nutzern in Kontakt treten kann?

A: Ja, bitte sehen Sie sich den Abschnitt "Forum" im offiziellen Hollywood Portal http://www.hollywood-mal.com online an.

Q: Wo kann ich um Hilfe bitten?

A: Es gibt ein lebhaftes englischsprachiges Forum auf http://forums.hollywood-mal.com und wir haben auch eine Mailing-Liste (auch englischsprachig), auf die Sie unter airsoft_hollywood@yahoogroups.com zugreifen können. Besuchen Sie https://www.hollywood-mal.de/ für Informationen darüber, wie Sie der Mailingliste beitreten können. Ausserdem ist ein deutschsprachiges Forum vorhanden, welches Sie unter https://www.amiga-resistance.info/ erreichen können.

Q: Ich habe einen Fehler gefunden.

A: Bitte poste darüber in den speziellen Bereichen des Forums oder der Mailingliste.

2.3 Bekannte Probleme

Hier ist eine Liste von Dingen, die GL Galore noch nicht unterstützt oder die in irgendeiner Weise verwirrend sein könnten:

- Menüs werden nicht unterstützt
- Das Mausrad wird nicht unterstützt
- Die Tastaturüberwachung, die Hollywoods `OnKeyDown-` und `OnKeyUp`-Ereignis-Handler zugeordnet ist, unterstützt derzeit nur Rohtastencodes basierend auf dem US-Tastaturlayout; das bedeutet, dass alle Tastaturzeichen als Großbuchstaben zurückgegeben werden und bei deutschen Tastaturen die Position der Y- und Z-Tasten vertauscht ist. Dies ist auf eine Beschränkung in GLFW zurückzuführen, die eine fein abgestimmte Überwachung (d.h. Tastenrunterdrücken, Tastenwiederholung, Tastenloslassen) unter Verwendung internationaler Tastaturen nicht unterstützt; Leute, die Hollywood 7.0 oder besser haben, können stattdessen einfach den Ereignis-Handler `VanillaKey` verwenden; Dieser Ereignis-Handler wird echte Tastaturereignisse einschließlich vollständiger Unicode-Unterstützung bereitstellen.
- Nicht alle Displaystile werden unterstützt

2.4 Zukunft

Hier sind einige Dinge, die auf meiner Liste stehen:

- Unterstützung für Tesselation hinzufügen
- Verbesserung der Unterstützung für OpenGL 1.2
- Integration von FTGL für 3D-Texteffekte
- Weitere Komfortfunktionen hinzufügen, die die Verwendung von OpenGL erleichtern

– Unterstützung für das Einbetten von OpenGL-Displays in MUI-GUIs über MUI Royale

Zögern Sie nicht, mich zu kontaktieren, wenn GL Galore eine bestimmte Funktion fehlt, die für Ihr Projekt wichtig ist.

2.5 Geschichte

Bitte beachten Sie die auf englisch verfasste Datei `history.txt` für ein vollständiges Änderungsprotokoll von GL Galore.

3 Verwendung

3.1 GL Galore aktivieren

Alles, was Sie tun müssen, damit Ihr Skript OpenGL anstelle des integrierten Grafiktreibers von Hollywood verwendet, ist die folgende Zeile am Anfang des Skripts hinzuzufügen:

```
@REQUIRE "glgalore"
```

Wenn Sie Hollywood von einer Konsole aus verwenden, können Sie Ihr Skript auch folgendermaßen starten:

```
Hollywood test.hws -requireplugins glgalore
```

Sobald das GL Galore-Plugin für Ihr Skript aktiviert wurde, wird es alle Grafikausgaben von Hollywood über OpenGL umleiten. Beachten Sie, dass dies bei Scripten, die nicht für OpenGL optimiert sind, normalerweise langsamer ist als bei Hollywoods integriertem Grafiktreiber. Um eine optimale Leistung mit OpenGL zu erzielen, benötigt Ihr Skript einen hardwarebeschleunigten Doppelpuffer. Siehe Abschnitt 3.3 [Verwendung eines Hardware-Doppelpuffers], Seite 10, für Details.

GL Galore akzeptiert die folgenden Argumente in seinem Aufruf mit `@REQUIRE`:

`ForceFullRefresh:`
: Wenn dieser Tag auf `False` gesetzt ist, aktualisiert GL Galore nur die Teile des Displays, die sich tatsächlich geändert haben. Dies ist zwar schneller, funktioniert aber nicht mit älteren OpenGL-Implementierungen (insbesondere auf Amiga), da diese oft keine pixelgenaue Positionierung von Grafiken bieten. Aus diesem Grund ist diese Variable standardmäßig auf `True` gesetzt, was bedeutet, dass GL Galore immer das gesamte Display aktualisiert, wenn etwas gezeichnet wird. Beachten Sie, dass dieser Tag nur verwendet wird, wenn GL Galore ohne einen Hardware-Doppelpuffer ausgeführt wird. Im doppelt gepufferten Hardware-Modus verursachen Vordergrund- und Hintergrundpuffer sowieso eine vollständige Aktualisierung.

Hier ist ein Beispiel, wie Argumente an die Präprozessor-Anweisung `@REQUIRE` übergeben werden:

```
@REQUIRE "glgalore", {ForceFullRefresh = False}
```

Alternativ können Sie auch das Argument `-requiretags` verwenden, um diese Argumente zu übergeben. Weitere Informationen finden Sie im Hollywood-Handbuch.

3.2 Zugriff auf OpenGL aus Hollywood heraus

GL Galore fügt alle OpenGL-Funktionen direkt in Hollywood ein, wobei die ursprüngliche Syntax kaum oder gar nicht geändert wird. Nach dem Aufruf von GL Galore mit `@REQUIRE` werden alle GL-Befehle innerhalb einer `"gl"`-Tabelle und die GLU-Befehle werden in einer `"glu"`-Tabelle verfügbar gemacht.

Das Aufrufen von OpenGL-Befehle aus Hollywood ist viel einfacher als die direkte Verwendung von OpenGL, da die Argumentsdefinition (z.B.,'2d','3f','4sv') am Ende der meisten OpenGL-Befehlsnamen entfernt wurde. Zum Beispiel bindet der GL Galore-Befehl

`gl.Light()` die OpenGL-Befehle: `glLightf`, `glLightfv`, `glLighti`, `glLightiv`. Die Anzahl der an `gl.Light()` übergebenen Parameter definiert den richtig zu verwendenden Befehl.

GL Galore verwendet normalerweise die Gleitkomma-Versionen aller OpenGL-Befehle mit der höchstmöglichen Genauigkeit. Einige Befehle, die einen Typparameter haben, verwenden einfach das höchstmögliche (normalerweise `#GL_DOUBLE` oder `#GL_FLOAT`) und der Format-Parameter wird nicht verwendet. Wenn `stride` nicht verwendet wird, wird 0 angenommen.

Die Farbe und die Vektordaten können als Hollywood-Tabelle oder als mehrere Parameter übergeben werden. Ein Vektor kann 2, 3 oder 4 Werte (x, y, z, w) und Farben können 3 oder 4 Werte haben (rot, grün, blau, alpha).

Beispielsweise:

```
v1 = {0, 0}
v2 = {1, 1}
yellow = {1, 1, 0}
gl.Color(yellow)
gl.Vertex(v1)
gl.Vertex(v2)
```

Oder Sie können auch dies tun:

```
gl.Color(1, 1, 0)
gl.Vertex(0, 0)
gl.Vertex(1, 1)
```

Es gibt einige OpenGL-Befehle, die einen Zeiger erwarten. In diesem Fall bietet GL Galore normalerweise auch eine Variante des Befehls an, womit er auch mit Hollywood-Tabellen funktioniert. Die ursprüngliche Zeigervariante ist jedoch auch in GL Galore verfügbar und kann für zeitkritische Skripte verwendet werden. Zum Beispiel liest `gl.ReadPixels()` Pixeldaten aus dem Bildspeicher in eine Hollywood-Tabelle, während `gl.ReadPixelsRaw()` Pixeldaten direkt in einen Speicherpuffer liest, was natürlich sehr viel schneller ist, aber Sie müssen dann mit Zeigern arbeiten. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details.

3.3 Verwendung eines Hardware-Doppelpuffers

Wenn Sie möchten, dass Ihr Skript von den hardwarebeschleunigten Zeichnungsfunktionen von OpenGL profitiert, müssen Sie einen Hardware-Doppelpuffer verwenden und alle Ihre Zeichnungen innerhalb dieses Doppelpuffers ausführen. Durch die Verwendung eines Hardware-Doppelpuffers wird außerdem sichergestellt, dass die Grafikausgabe mit der Aktualisierungsrate des Monitors synchronisiert wird, um Flimmern zu vermeiden. Um eine optimale Leistung mit OpenGL zu erhalten, sollte Ihre Hauptschleife immer so aussehen:

```
@REQUIRE "glgalore"

BeginDoubleBuffer(True)  ; richtet einen Hardware-Doppelpuffer ein

Repeat
    ....           ; zeichnet hier das nächste Einzelbild
    Flip()         ; wartet auf die vertikale Aktualisierung...
                   ; ... und tauscht dann die Puffer aus
```

```
    CheckEvent()  ; Callback-Funktion ausführen
Forever
```

Der Aufruf von `CheckEvent()` ist nur erforderlich, wenn Ihr Skript auf Ereignis-Handler warten muss, die mit `InstallEventHandler()` installiert wurden. Beachten Sie, dass Sie das nächste Bild nicht in einem Intervall-Callback zeichnen sollten, der mit einer konstanten Bildrate (etwa 50 fps) läuft. Ein solches Setup garantiert nicht, dass die Zeichnung mit der vertikalen Aktualisierung synchronisiert wird, weil verschiedene Monitore unterschiedliche Bildwiederholraten verwenden und somit Sie flackernde Grafiken erhalten. Wenn Sie Ihre Zeichnung wie oben beschrieben ausführen, können Sie sicher sein, dass die vorderen und hinteren Puffer in perfekter Synchronisation mit der vertikalen Aktualisierung des Monitors gedreht werden.

Darüber hinaus müssen Sie darauf achten, wie Sie Ihre Grafiken tatsächlich zeichnen, da die meisten Zeichnungsbefehle von Hollywood vollständig im Softwaremodus arbeiten und daher nicht von der Hardwarebeschleunigung profitieren. Siehe Abschnitt 3.4 [Grafik zeichnen], Seite 11, für Details.

Wichtig: OpenGL ist für die Verwendung mit doppelten Puffern vorgesehen. Daher müssen alle OpenGL-Zeichenbefehle innerhalb eines Doppelpuffers aufgerufen werden. Das Zeichnen außerhalb eines Doppelpuffers mit OpenGL wird nicht unterstützt.

3.4 Grafik zeichnen

Um eine optimale Leistung zu erzielen, müssen Sie beim Zeichnen Ihrer Grafiken sehr vorsichtig sein. Die meisten von Hollywoods Zeichenbefehlen sind nur in der Software implementiert, d.h. Sie zeichnen mit der CPU statt mit der GPU. Dies kann insbesondere bei langsameren CPUs zu einem Engpass werden. Daher sollten Sie direkt und jederzeit mit den von GL Galore angebotenen OpenGL-Befehlen zeichnen.

Dennoch gibt es einige Hollywood-Befehle, die direkt auf OpenGL umgeleitet werden, wenn GL Galore aktiviert wurde. Dies sind die folgenden:

```
Box()
Cls()
Line()
Plot()
DisplayBrush()
```

Sie können diese Befehle mit OpenGL ohne Leistungseinbußen verwenden. Es gibt jedoch einige Einschränkungen: `Box()`, `Line()` und `WritePixel()` werden nur dann zu OpenGL umgeleitet, wenn der Füllstil entweder `#FILLNONE` oder `#FILLCOLOR` ist und keine anderen Formstile wie `#EDGE` oder `#SHADOW` aktiv sind. Sobald Sie mit anderen Füll- oder Formstilen zeichnen möchten, greifen diese Befehle auf ihre Software-Gegenstücke zurück und sind daher sehr langsam.

`DisplayBrush()` verwendet OpenGL nur direkt, wenn es mit einem Hardwarepinsel aufgerufen wird. Siehe Abschnitt 3.5 [Verwenden von Hardwarepinseln], Seite 12, für Details. Wenn `DisplayBrush()` mit einem Software-Pinsel verwendet wird, d.h. einem Pinsel, der sich nicht im Videospeicher befindet, zeichnet er den Pinsel mit der viel langsameren CPU.

Beim Mischen von Hollywood- und OpenGL-Zeichnungsbefehlen gibt es jedoch ein weiteres potenzielles Problem, das Sie beachten müssen: Da OpenGL mit Zuständen arbeitet,

können Änderungen am GL-Status, die von einem der Zeichnungsbefehle Hollywoods vorgenommen werden, nachfolgende Aufrufe von OpenGL-Befehlen beeinflussen. So kann es sein, dass Sie nach dem Aufruf eines Hollywood-Befehls, der auf OpenGL umgeleitet wird, bestimmte Zustände wiederherstellen müssen, z.B. die aktuelle Farbe, Transformationsmatrix, Matrixmodus, Texturierung oder Einblendung wieder aktivieren, etc. Dies kann sehr mühsam werden, so dass es oft einfacher ist, nur OpenGL-Befehle zu verwenden, um nicht nach dem Aufruf von Hollywood-Befehlen Zustände wiederherstellen zu müssen.

Vergessen Sie schließlich nicht, dass Sie alle Ihre Zeichnung in einer Hardware-Doppelpufferschleife erstellen sollten. Siehe Abschnitt 3.3 [Verwendung eines Hardware-Doppelpuffers], Seite 10, für Details.

3.5 Verwenden von Hardwarepinseln

GL Galore unterstützt die Erstellung von Hardwarepinsel. Hardwarepinsel befinden sich im GPU-Speicher und können somit in kürzester Zeit gezeichnet werden. Auf den meisten Grafikkarten können sie auch sehr effizient von der GPU skaliert und transformiert werden. Damit Hollywood einen Hardwarepinsel erstellt, müssen Sie lediglich den optionalen Tag "Hardware" auf `True` setzen. Dieser Tag wird von den meisten Hollywood-Befehlen unterstützt, die Pinsel erstellen.

Hier ist ein Beispiel:

```
@REQUIRE "glgalore"  ; stellen Sie sicher, dass diese Zeile ...
                     ; ... die erste ist
@BRUSH 1, "sprites.png", {Hardware = True}
```

Im obigen Code erstellt GL Galore Pinsel 1 im Videospeicher. Er kann dann fast ohne Aufwand mit der GPU gezeichnet werden. Beachten Sie jedoch, dass Hardwarepinsel nur auf Hardware-Doppelpuffer gezeichnet werden können. Siehe Abschnitt 3.3 [Verwendung eines Hardware-Doppelpuffers], Seite 10, für Details.

Um einen Hardwarepinsel zu transformieren, können Sie die Befehle `ScaleBrush()`, `RotateBrush()` und `TransformBrush()` verwenden. Transformationen von Hardwarepinseln sind in der Regel auch GPU-beschleunigt und damit um ein Vielfaches schneller als die von der CPU durchgeführten Transformationen.

Beachten Sie, dass Hardwarepinsel nur auf dem Display gezeichnet werden können, welches bei der Zuweisung angegeben wurde. Wenn Ihr Skript also mehrere Displays verwendet, müssen Sie Hollywood den Identifikator des Displays mitteilen, mit dem Sie diesen Hardwarepinsel verwenden möchten. Dies kann durch Angabe des Tags "Display" hinter dem Tag "Hardware" erfolgen. Hier ist ein Beispiel:

```
@REQUIRE "glgalore"  ; stellen Sie sicher, dass diese Zeile ...
                     ; ... die erste ist
@DISPLAY 1, {Title = "First display"}
@DISPLAY 2, {Title = "Second display"}
@BRUSH 1, "sprites.png", {Hardware = True, Display = 1}
@BRUSH 2, "sprites.png", {Hardware = True, Display = 2}
```

Der obige Code wird den Pinsel 1 so zuweisen, dass er auf dem Display 1 gezeichnet werden kann und er wird den Pinsel 2 so zuweisen, dass er auf dem Display 2 gezeichnet werden kann. Es ist jedoch nicht möglich, den Pinsel 2 auf dem Display 1 zu zeichnen oder Pinsel 1

auf dem Display 2 anzuzeigen! OpenGL-Hardwarepinsel sind immer Displayabhängig und können nur auf dem Display gezeichnet werden, für die sie zugewiesen wurden.

Weitere Informationen zu Hardwarepinseln und Hardware-Doppelpuffern finden Sie im Hollywood-Handbuch.

Sie können GL Galore auch als Hilfs-Plugin verwenden, um Hollywood unter Windows, Mac OS X und Linux Hardwarepinsel-Unterstützung hinzuzufügen. Standardmäßig unterstützt Hollywood keine Hardwarepinsel auf diesen Systemen, aber GL Galore kann diese Funktion Hollywood hinzufügen. Siehe Abschnitt 3.11 [GL Galore als Helfer-Plugin], Seite 15, für Details.

Das Skript `SmoothScroll.hws`, das mit GL Galore geliefert wird, demonstriert, wie Hardwarepinsel und ein Hardware-Doppelpuffer ohne Aufrufe von OpenGL verwendet werden.

3.6 Mehrere Displays

Bei Verwendung mehrerer Displays verwaltet GL Galore für jedes Display einen separaten OpenGL-Kontext. Daher müssen Sie OpenGL mitteilen, in welchem Kontext alle Aufrufe an den GL laufen sollen. Dies geschieht durch Aufruf des Befehls `gl.SetCurrentContext()`, wodurch der GL-Kontext des angegebenen Hollywood-Displays den aktuellen Kontext anzeigt. Siehe Abschnitt 9.6 [gl.SetCurrentContext], Seite 263, für Details.

Wenn Sie mit Hardwarepinseln arbeiten, müssen Sie auch vorsichtig sein, wenn Sie mehrere Displays verwenden, da Hardwarepinsel in OpenGL Displayabhängig sind. Sie können nur auf dem Display gezeichnet werden, dem sie zugewiesen wurden. Siehe Abschnitt 3.5 [Verwenden von Hardwarepinseln], Seite 12, für Details.

3.7 Mit Zeigern arbeiten

Mehrere OpenGL-Befehle erwarten, dass Sie ihnen einen Zeiger auf einen Roh-Speicher übergeben. Das direkte Arbeiten mit Zeigern ist der effizienteste Weg, um mit OpenGL zu interagieren, da es jegliche Verwaltungsdaten vermeidet, die dadurch entstehen, wenn der Inhalt von Hollywood-Tabellen zuerst in den Speicherpuffer gelesen werden muss.

Sie können die Befehle der Speicherblockbibliothek von Hollywood verwenden, um Speicherpuffer zuzuordnen, sie zu lesen oder zu schreiben und einen Zeiger auf ihren Roh-Speicherpuffer zu erhalten. Um einen Speicherpuffer zuzuweisen, verwenden Sie den Befehl `AllocMem()`, um aus einem Speicherpuffer zu lesen den Befehl `Peek()`, während `Poke()` zum Schreiben in einen Speicherpuffer verwendet werden kann. Schließlich gibt `GetMemPointer()` den Zeiger eines Speicherblockobjekts zurück. Sie können den Rückgabewert von `GetMemPointer()` an alle OpenGL-Befehle übergeben, die ein Zeigerargument haben.

Hier ist ein Beispiel:

```
AllocMem(1, 640*480*4)
Local ptr = GetMemPointer(1)
gl.ReadPixelsRaw(0, 0, 640, 480, #GL_BGRA, #GL_UNSIGNED_BYTE, ptr)
... ; do something with the data
FreeMem(1)
```

Der obige Code liest 480 Zeilen von 640 Pixeln in das Speicherblockobjekt 1 ein. Sie könnten dann die Daten mit `WriteMem()` in eine Datei schreiben, Sie könnten sie in eine Tabelle mit-

hilfe von `MemToTable()` konvertieren oder einzelne Werte daraus mit `Peek()` lesen. Weitere Informationen finden Sie in der Dokumentation der Speicherblockbibliothek von Hollywood.

3.8 Hollywood-Brücke

GL Galore bietet einige zusätzliche Befehle, die nicht Teil der offiziellen OpenGL API sind. Mit diesen Befehlen können Sie Hollywood-Objekte wie Pinsel mit OpenGL bequem verwenden. Mit dem Befehl `gl.TexImageFromBrush()` können Sie z.B. einen Hollywood-Pinsel als OpenGL-Textur laden und mit den Befehlen `gl.GetTexImageToBrush()` können Sie eine OpenGL-Textur zurück in einen Hollywood-Pinsel konvertieren.

Im Kapitel "IX. Hollywood-Brücke" finden Sie alle verfügbaren Befehle.

3.9 Erhöhung der Ausführungsgeschwindigkeit

Um die Geschwindigkeit der reinen Ausführung Ihres Skripts zu erhöhen, können Sie die Hollywood-Funktion "LineHook" mit den Befehlen `DisableLineHook()` deaktivieren und mit `EnableLineHook()` wieder aktivieren. Dies wird die Ausführungsgeschwindigkeit Ihres Skripts erheblich verbessern, falls viel Hollywood-Code ausgeführt werden muss, um das nächste Einzelbild zu zeichnen. Beachten Sie jedoch, dass Sie den "LineHook" für jedes von Ihnen gezeichnete Einzelbild wieder aktivieren müssen, oder Ihr Fenster reagiert nicht mehr. So könnte eine geschwindigkeitsoptimierte Implementierung der Hauptschleife aussehen:

```
@REQUIRE "glgalore"

BeginDoubleBuffer(True)  ; richtet einen Hardware-Doppelpuffer ein

Repeat
    DisableLineHook() ; deaktiviert den "LineHook"
    p_DrawFrame()     ; zeichnet hier das nächste Einzelbild
    EnableLineHook()  ; aktiviert den "LineHook" erneut
    Flip()            ; wartet auf die vertikale Aktualisierung, ...
                      ; und tauschet dann die Puffer aus
    CheckEvent()      ; Callback-Funktion ausführen
Forever
```

Beachten Sie, dass Sie hier nur einen Geschwindigkeitsunterschied bemerken, wenn `p_DrawFrame()` viele Zeilen Hollywood-Code ausführt. Wenn `p_DrawFrame()` nur aus 20 Codezeilen besteht, werden Sie keinen Unterschied bemerken. Es ist nur mit Hunderten von Codezeilen oder langen Schleifen bemerkbar.

Weitere Informationen finden Sie in der Dokumentation von `DisableLineHook()` und `EnableLineHook()` im Hollywood-Handbuch.

3.10 Moduswechsel-Schalter

Wie Sie vielleicht wissen, bietet Hollywood eine Tastaturkombination zum Umschalten zwischen Fenster- und Vollbildmodus. Immer wenn der Benutzer ALT+RETURN oder COMMAND+RETURN drückt, wechselt Hollywood automatisch zwischen den Modi "Fenster" und "Vollbild". Dieses Verhalten ist standardmäßig aktiviert. Es kann deaktiviert werden, indem `ModeTag` in der Präprozessor-Anweisungen `@DISPLAY` auf `False` gesetzt wird.

Bei Verwendung des integrierten Bildschirmtreibers von Hollywood werden die Modiwechsel automatisch von Hollywood übernommen und das Skript muss nichts tun. Dies ist bei GL Galore anders. Mit GL Galore müssen Sie Ihren GL-Kontext nach einem Moduswechsel neu initialisieren. Dies ist notwendig, da der alte GL-Kontext gelöscht wird, wenn Hollywood den Modus wechselt. Somit gehen alle aktuellen GL-Zustände in einem Moduswechsel verloren. Dazu gehören auch Texturen und Displaylisten, die Ihr Skript zugewiesen hat. Nach einem Moduswechsel wird der GL-Kontext Ihres Skripts durch einen Vanilla-GL-Kontext ersetzt, der mit demjenigen identisch ist, mit dem Ihr Skript gestartet wurde.

Um den Moduswechsel mit GL Galore zu unterstützen, müssen Sie mit `InstallEventHandler()` eine Überwachung im Tag `ModeSwitch` installieren. Immer wenn dieser Ereignis-Handler ausgelöst wird, müssen Sie Ihren GL-Kontext neu initialisieren und jeden Status auf die gewünschten Werte setzen. Normalerweise müssen Sie Ihren Initialisierungscode, der Ihren GL-Kontext am Anfang des Skripts einrichtet, nur dann erneut ausführen, wenn das `ModeSwitch`-Ereignis ausgelöst wird. Wenn Ihnen das zu viel Ärger bereitet, können Sie auch den automatischen Moduswechsel deaktivieren.

Bitte beachten Sie, dass es nicht notwendig ist, den Moduswechsel manuell zu handhaben, falls Sie die OpenGL API nicht direkt verwenden. Die von GL Galore zugewiesenen Hardwarepinsel werden von GL Galore automatisch in den neuen GL-Kontext übertragen, sodass Sie nichts dafür unternehmen müssen. Es ist nur notwendig, wenn OpenGL direkt programmiert wird.

3.11 GL Galore als Helfer-Plugin

GL Galore kann auch als Hilfs-Plugin verwendet werden, um das Problem zu umgehen, dass Hollywood nur hardwarebeschleunigte Doppelpuffer und Pinsel auf AmigaOS und Kompatible unterstützt. Sie werden unter Windows, Mac OS X oder Linux nicht unterstützt. Wenn Sie GL Galore installieren und `@REQUIRE` aufrufen, werden Hardware-Doppelpuffer und Hardwarepinsel-Unterstützung auch unter Windows, Mac OS X und Linux verfügbar sein, da GL Galore dies unterstützt.

Daher können Sie GL Galore auch als Hilfs-Plugin verwenden, um hardwarebeschleunigte Doppelpuffer-Unterstützung für Windows, Mac OS X und Linux zu erhalten. Sie müssen nicht einen der OpenGL-Befehle direkt verwenden. Sie können einfach nur `@REQUIRE` in GL Galore einen Hardware-Doppelpuffer einrichten und dann mit Hardwarepinseln darauf zeichnen. Dadurch können Sie die Hardwarebeschleunigung nutzen, ohne eine einzelne Zeile OpenGL-Code schreiben zu müssen!

Auf AmigaOS und kompatiblen Systemen ist dies nicht notwendig, da Hollywood hardwarebeschleunigte Doppelpuffer und Pinsel standardmäßig bereits unterstützt. Dennoch kann die Verwendung von GL Galore auf AmigaOS als Hardware-Doppelpuffertreiber im Vollbildmodus von Vorteil sein, da GL Galore Zeichnungen verwendet, die perfekt mit der vertikalen Aktualisierung des Monitors synchronisiert ist, so dass sie normalerweise besser aussehen als die von Hollywood direkt verwalteten Doppelpuffer.

Siehe Abschnitt 3.3 [Verwendung eines Hardware-Doppelpuffers], Seite 10, für Details.

Siehe Abschnitt 3.5 [Verwenden von Hardwarepinseln], Seite 12, für Details.

Das Skript `SmoothScroll.hws`, das mit GL Galore geliefert wird, demonstriert, wie Hardwarepinsel und ein Hardware-Doppelpuffer ohne Aufrufe von OpenGL verwendet werden.

3.12 Interne Pixelformate

OpenGL-Befehle, die Texturen erzeugen, z.B. `gl.TexImage2D()` oder `gl.CopyTexImage()` akzeptieren einen Parameter `internalFormat`, mit dem Sie das interne Format der Textur festlegen können. Die folgenden Formatkonstanten werden derzeit von GL Galore unterstützt:

```
#GL_ALPHA
#GL_ALPHA4
#GL_ALPHA8
#GL_ALPHA12
#GL_ALPHA16
#GL_LUMINANCE
#GL_LUMINANCE4
#GL_LUMINANCE8
#GL_LUMINANCE12
#GL_LUMINANCE16
#GL_LUMINANCE_ALPHA
#GL_LUMINANCE4_ALPHA4
#GL_LUMINANCE6_ALPHA2
#GL_LUMINANCE8_ALPHA8
#GL_LUMINANCE12_ALPHA4
#GL_LUMINANCE12_ALPHA12
#GL_LUMINANCE16_ALPHA16
#GL_INTENSITY
#GL_INTENSITY4
#GL_INTENSITY8
#GL_INTENSITY12
#GL_INTENSITY16
#GL_RGB
#GL_R3_G3_B2
#GL_RGB4
#GL_RGB5
#GL_RGB8
#GL_RGB10
#GL_RGB12
#GL_RGB16
#GL_RGBA
#GL_RGBA2
#GL_RGBA4
#GL_RGB5_A1
#GL_RGBA8
#GL_RGB10_A2
#GL_RGBA12
#GL_RGBA16
#GL_DEPTH_COMPONENT
```

Beachten Sie, dass `gl.TexImage1D()` und `gl.TexImage2D()` auch die speziellen Werte 1, 2, 3 und 4 als gültige interne Pixelformate akzeptieren, aber `gl.CopyTexImage()` unterstützt dies nicht.

4 Tutorial

4.1 OpenGL Tutorial

Leider gibt es zur Zeit kein Tutorial, um mit GL Galore zu beginnen. Das Internet ist jedoch voll von Einsteiger-Tutorials für OpenGL, mit denen Sie in die Engine einsteigen können. Da GL Galore nur die OpenGL-API umschließt, ist es meist einfach und unkompliziert, Code der für andere Programmiersprachen geschrieben wurde, in GL Galore zu portieren. Die Beispiele, die mit GL Galore geliefert werden, können Ihnen auch helfen, mit GL Galore anzufangen.

Schließlich sind die Hollywood-Foren immer ein guter Ort, um Ihre Frage zu stellen, wenn Sie mit der Programmierung von GL Galore beschäftigt sind. Einfach `http://forums.hollywood-mal.com` besuchen und fragen.

5 Beispiele

5.1 Beispiele mit GL Galore

GL Galore enthält eine Reihe von Beispielen, die bestimmte Befehle demonstrieren und Ihnen einen schnellen Einstieg ermöglichen. Hier ist eine Liste von Beispielen, die mit GL Galore verteilt werden:

BlockTube
: Ein wirbelnder, fallender Tunnel aus reflektierenden Platten, die von Farbton zu Farbton verschwinden. Originalcode von Lars Damerow.

Boing
: Ein Klon der ersten Grafikdemo für den Amiga 1000, der von Dale Luck und RJ Mical während einer Pause auf der CES 1984 geschrieben wurde. Originalcode von Jim Brooks.

Cel shading
: Demonstriert das Verfahren "Cel Shading". Originalcode von Jeff Molofee.

Cityflow
: Wellenbewegungen über ein Meer von Boxen. Die Stadt schwillt an. Die Wände schließen sich. Originalcode von Jamie Zawinski.

Cube
: Das OpenGL-Äquivalent von "Hello World".

Gears
: Die klassische OpenGL-Getriebe-Demo. Originalcode von Brian Paul.

Gears 2
: Eine Variante der OpenGL-Getriebe-Demo basierend auf dem Code im MiniGL SDK.

Gears 3
: Die OpenGL-Getriebe-Demo, diesmal mit einer Textur.

GLMatrix
: Der digitale 3D-Regeneffekt, wie er in der Titelsequenz eines populären Films zu sehen ist. Basierend auf Code von Jamie Zawinski.

Morph3D
: Platonische Körper, die sich nach innen wenden und spitz werden. Basierend auf dem Code von Marcelo F. Vianna.

MultiDisplays
: Demonstriert, wie mehrere Displays mit GL Galore verwendet werden.

Simple
: Einfaches drehendes GL-Dreieck. Basierend auf dem Code von Camilla Berglund.

SmoothScroll
: Verwendet OpenGL für die hardwarebeschleunigte 2D-Zeichnung von Hollywood-Pinseln.

SplitView
: Rendert vier Ansichten derselben Szene in einem Fenster. Basierend auf dem Code von Camilla Berglund.

Spots
: Demonstriert GL-Lichter. Basierend auf dem Code von Mark J. Kilgard.

Sproingies
: Aufreizend-ähnliche Kreaturen gehen eine unendliche Treppe hinunter und explodieren gelegentlich! Basierend auf dem Code von Ed Mackey.

Warp
: Beispiel dafür, was ein extremes Sichtfeld leisten kann.

Wave
: Wellensimulation in OpenGL. Basierend auf dem Code von Jakob Thomsen.

6 GL Referenz

6.1 gl.Accum

BEZEICHNUNG

gl.Accum – definiert die Arbeitsweise des Akkumulationspuffers

ÜBERSICHT

```
gl.Accum(op, value)
```

BESCHREIBUNG

Der Akkumulationspuffer ist ein erweiterter Farbpuffer. Bilder werden nicht darin wiedergegeben. Stattdessen werden Bilder, die in einem der Farbpuffer berechnet werden, nach der Berechnung zu den Inhalten des Akkumulationspuffers hinzugefügt. Effekte wie Antialiasing (von Punkten, Linien und Polygonen), Bewegungsunschärfe und Tiefenschärfe können durch Akkumulieren von Bildern erzeugt werden, die mit verschiedenen Transformationsmatrizen erzeugt wurden.

Jedes Pixel im Akkumulationspuffer besteht aus Rot-, Grün-, Blau- und Alpha-Werten. Die Anzahl der Bits pro Komponente im Akkumulationspuffer hängt von der Implementierung ab. Sie können diese Zahl untersuchen, indem Sie `gl.Get()` viermal jeweils mit den Argumenten `#GL_ACCUM_RED_BITS`, `#GL_ACCUM_GREEN_BITS`, `#GL_ACCUM_BLUE_BITS` und `#GL_ACCUM_ALPHA_BITS` aufrufen. Unabhängig von der Anzahl der Bits pro Komponente beträgt der Wertebereich für jede Komponente -1 bis 1. Die Akkumulationspufferpixel werden eins zu eins mit Bildpufferpixeln abgebildet.

`gl.Accum()` arbeitet mit dem Akkumulationspuffer. Das erste Argument, `op`, ist eine symbolische Konstante, die eine Akkumulationspufferoperation auswählt. Das zweite Argument `value` ist ein Gleitkommawert, der in dieser Operation verwendet wird. Fünf Operationen sind angegeben: `#GL_ACCUM`, `#GL_LOAD`, `#GL_ADD`, `#GL_MULT`, und `#GL_RETURN`.

Alle Akkumulationspufferoperationen sind auf den Rahmen des aktuellen Bereichs beschränkt und werden identisch auf die Rot-, Grün-, Blau- und Alpha-Komponenten jedes Pixels angewendet. Wenn eine gl.Accum()-Operation zu einem Wert außerhalb des Bereichs von -1 bis 1 führt, sind die Inhalte einer Akkumulationspufferpixel-Komponente nicht definiert.

Die Operationen sind wie folgt:

`#GL_ACCUM`
: Ermittelt R-, G-, B- und A-Werte aus dem aktuell zum Lesen ausgewählten Puffer. Siehe Abschnitt 6.121 [gl.ReadBuffer], Seite 188, für Details. Jeder Komponentenwert wird durch 2^n - 1 geteilt, wobei n die Anzahl von Bits ist, die jeder Farbkomponente in dem gegenwärtig ausgewählten Puffer zugeordnet ist. Das Ergebnis ist ein Gleitkommawert im Bereich von 0 bis 1, der mit dem Wert multipliziert und zu der entsprechenden Pixelkomponente in dem Akkumulationspuffer addiert wird, wodurch der Akkumulationspuffer aktualisiert wird.

`#GL_LOAD`
: Ähnlich wie `#GL_ACCUM`, außer dass der aktuelle Wert im Akkumulationspuffer bei der Berechnung des neuen Wertes nicht verwendet wird. Das heißt,

die R-, G-, B- und A-Werte aus dem gegenwärtig ausgewählten Puffer werden durch 2^n - 1 geteilt, mit dem Wert multipliziert und dann in der entsprechenden Akkumulationspufferzelle gespeichert, wobei der aktuelle Wert überschrieben wird.

`#GL_ADD` Fügt jedem R, G, B und A im Akkumulationspuffer einen Wert hinzu.

`#GL_MULT` Multipliziert jedes R, G, B und A im Akkumulationspuffer mit dem Wert und gibt die skalierte Komponente an die entsprechende Akkumulationspufferposition zurück.

`#GL_RETURN`
Überträgt die Akkumulationspufferwerte in den Farbpuffer oder in den Puffer, der derzeit zum Schreiben ausgewählt wurde. Jede Komponente R, G, B und A wird mit einem Wert multipliziert, dann mit 2^n - 1 multipliziert, auf den Bereich [0,2^n - 1] fixiert und in der entsprechenden Anzeigepufferzelle gespeichert. Die einzigen Fragmentoperationen, die auf diese Übertragung angewendet werden, sind Pixelbesitz-, Schere-, Dither- und Farbschreibmasken.

Um den Akkumulationspuffer zu löschen, rufen Sie `gl.ClearAccum()` mit R-, G-, B- und A-Werten auf, um den Puffer zu setzen. Dann rufen Sie `gl.Clear()` mit aktiviertem Akkumulationspuffer auf.

Nur Pixel innerhalb der aktuellen Scherenbox werden durch eine gl.Accum()-Operation aktualisiert. Siehe Abschnitt 6.128 [glScissor], Seite 197, für weitere Informationen über die Scherenbox.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`op` spezifiziert die Akkumulationspufferoperation. Symbolische Konstanten `#GL_ACCUM`, `#GL_LOAD`, `#GL_ADD`, `#GL_MULT` und `#GL_RETURN` werden akzeptiert

`value` gibt einen Gleitkommawert an, der in der Akkumulationspufferoperation verwendet wird. `op` legt fest, wie `value` verwendet wird

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn op kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird generiert, wenn kein Akkumulationspuffer vorhanden ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.Accum()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ACCUM_RED_BITS`

`gl.Get()` mit dem Argument `#GL_ACCUM_GREEN_BITS`

`gl.Get()` mit dem Argument `#GL_ACCUM_BLUE_BITS`

`gl.Get()` mit dem Argument `#GL_ACCUM_ALPHA_BITS`

6.2 gl.AlphaFunc

BEZEICHNUNG

gl.AlphaFunc – setzt die Alpha-Test-Funktion

ÜBERSICHT

```
gl.AlphaFunc(func, ref)
```

BESCHREIBUNG

Der Alpha-Test verwirft Fragmente in Abhängigkeit vom Ergebnis eines Vergleichs zwischen dem Alpha-Wert eines eingehenden Fragments und einem konstanten Referenzwert. `gl.AlphaFunc()` gibt den Referenzwert und die Vergleichsfunktion an. Der Vergleich wird nur durchgeführt, wenn der Alpha-Test aktiviert ist. Standardmäßig ist er nicht aktiviert. (Siehe `gl.Enable()` und `gl.Disable()` unter `#GL_ALPHA_TEST`.)

`func` und `ref` geben die Bedingungen an, unter denen das Pixel gezeichnet wird. Der eingehende Alpha-Wert wird in `ref` mit der in `func` angegebenen Funktion verglichen. Wenn der Wert den Vergleich besteht, wird das eingehende Fragment gezeichnet, wenn es auch nachfolgende Schablonen- und Tiefenpuffer-Tests (Stencil- und Depth-Puffertest) durchläuft. Wenn der Wert den Vergleich nicht besteht, wird keine Änderung an dem Bildpuffer an dieser Pixelstelle vorgenommen. Die Vergleichsfunktionen sind wie folgt:

`#GL_NEVER`
: Niemals erfolgreich.

`#GL_LESS`
: Ist erfolgreich, wenn der ankommende Alpha-Wert kleiner als der Referenzwert ist.

`#GL_EQUAL`
: Ist erfolgreich, wenn der ankommende Alpha-Wert gleich dem der Referenzwert ist.

`#GL_LEQUAL`
: Ist erfolgreich, wenn der ankommende Alpha-Wert kleiner oder gleich dem Referenzwert ist.

`#GL_GREATER`
: Ist erfolgreich, wenn der ankommende Alpha-Wert größer als der Referenzwert ist.

`#GL_NOTEQUAL`
: Ist erfolgreich, wenn der ankommende Alpha-Wert nicht mit dem Referenzwert übereinstimmt.

`#GL_GEQUAL`
: Ist erfolgreich, wenn der ankommende Alpha-Wert größer oder gleich dem Referenzwert ist.

`#GL_ALWAYS`
: Immer erfolgreich (Voreingestellt).

`gl.AlphaFunc()` funktioniert bei allen Pixelschreibvorgängen, einschließlich denjenigen, die sich aus der Scanumwandlung von Punkten, Linien, Polygonen und Bitmaps ergeben und aus Pixelzeichnungs- und Pixelkopiervorgängen. `gl.AlphaFunc()` wirkt sich nicht auf Bildschirmlöschvorgänge aus.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`func` Alpha-Vergleichsfunktion (siehe oben)

`ref` Gibt den Referenzwert an, mit dem eingehende Alpha-Werte verglichen werden. Dieser Wert wird auf den Bereich 0 bis 1 festgelegt, wobei 0 den niedrigsten möglichen Alpha-Wert und 1 den höchstmöglichen Wert darstellt (der anfängliche Referenzwert ist 0).

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `func` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.AlphaFunc()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ALPHA_TEST_FUNC`

`gl.Get()` mit dem Argument `#GL_ALPHA_TEST_REF`

`gl.IsEnabled()` mit dem Argument `#GL_ALPHA_TEST`

6.3 gl.AreTexturesResident

BEZEICHNUNG

gl.AreTexturesResident – bestimmt, ob Texturen im Texturspeicher geladen sind

ÜBERSICHT

```
residencesArray = gl.AreTexturesResident(texturesArray)
```

BESCHREIBUNG

GL erstellt einen Arbeitssatz von Texturen, die sich im Texturspeicher befinden. Diese Texturen können viel effizienter an ein Texturziel gebunden werden als Texturen, die nicht resident sind.

`gl.AreTexturesResident()` fragt den Textur-Anwesenheitsstatus der n Texturen ab, die in `texturesArray` angegeben werden und gibt ihren Status in der Tabelle `residencesArray` zurück.

Der Anwesenheitsstatus einer einzelnen gebundenen Textur kann auch durch Aufruf von `gl.GetTexParameter()` abgefragt werden, wobei das Zielargument auf das Ziel festgelegt ist, an das die Textur gebunden ist und das Argument `pname` auf `#GL_TEXTURE_RESIDENT` gesetzt ist. Nur so kann der Anwesenheitsstatus einer Standardtextur abgefragt werden.

`gl.AreTexturesResident()` gibt den Anwesenheitsstatus der Texturen zum Zeitpunkt des Aufrufs zurück. Es kann nicht garantiert werden, dass die Texturen zu einem anderen Zeitpunkt noch im Texturspeicher anwesend sind.

Wenn sich Texturen im virtuellen Speicher befinden (falls es keinen Texturspeicher vorhanden ist), werden sie immer als anwesend betrachtet.

Einige Implementierungen laden eine Textur möglicherweise erst bei der ersten Verwendung dieser Textur.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`texturesArray`
: ein Feld, das die Namen der Texturen enthält, die abgefragt werden sollen

RÜCKGABEWERTE

`residencesArray`
: ein Feld, in dem der Status der Anwesenheit zurückgegeben wird

FEHLER

`#GL_INVALID_VALUE` wird generiert, wenn ein beliebiges Element in `texturesArray` 0 oder kein gültiger Texturname ist. In diesem Fall gibt der Befehl `Nil` zurück.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.AreTexturesResident()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexParameter()` mit dem Parameternamen `#GL_TEXTURE_RESIDENT` wird der Aufenthaltsstatus einer aktuell gebundenen Textur abgerufen.

6.4 gl.ArrayElement

BEZEICHNUNG

gl.ArrayElement – rendert einen Scheitelpunkt mit dem angegebenen Scheitel-Feldelement

ÜBERSICHT

```
gl.ArrayElement(i)
```

BESCHREIBUNG

`gl.ArrayElement()` Befehle werden innerhalb von `gl.Begin()` und `gl.End()` verwendet, um Scheitel- und Attributdaten für Punkt-, Linien- und Polygon-Grundelemente anzugeben. Wenn `#GL_VERTEX_ARRAY` aktiviert ist und `gl.ArrayElement()` aufgerufen wird, wird ein einzelner Scheitelpunkt gezeichnet. Dieser verwendet Scheitel- und Attributdaten, die vom Speicherort `i` der aktivierten Felder stammen. Wenn `#GL_VERTEX_ARRAY` nicht aktiviert ist, wird keine Zeichnung erstellt, aber die Attribute, die den aktivierten Feldern entsprechen, werden geändert.

Verwenden Sie `gl.ArrayElement()`, um Grundelemente zu konstruieren, indem Scheiteldaten indiziert werden, anstatt Datenfelder in der Reihenfolge von vorne nach hinten zu durchlaufen. Da jeder Aufruf nur einen einzelnen Scheitelpunkt angibt, ist es möglich, durch einfache Attribute wie z.B. eine einzelne Normale für jedes Dreieck explizit anzugeben.

Änderungen an Felddaten zwischen der Ausführung von `gl.Begin()` und `gl.End()` können Aufrufe von `gl.ArrayElement()` beeinflussen, die innerhalb der gleichen `gl.Begin()` und `gl.End()`-Periode auf nicht sequentielle Weise ausgeführt werden. Das heißt, ein Aufruf von `gl.ArrayElement()`, der einer Änderung an Felddaten vorausgeht, kann auf die geänderten Daten zugreifen und ein Aufruf, der einer Änderung an Felddaten folgt, kann auf Originaldaten zugreifen.

glArrayElement wird von Displaylisten aufgenommen. Wenn `gl.ArrayElement()` in eine Display-Liste eingegeben wird, werden auch die erforderlichen Felddaten (bestimmt

durch die Feldzeiger und Aktivierungen) in die Display-Liste eingetragen. Da die Feldzeiger und -ereignisse Klientseitig sind, wirken sich ihre Werte bei der Erstellung der Listen auf Display-Listen aus, nicht jedoch bei der Ausführung der Listen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`i` gibt einen Index für die aktivierten Scheitel-Datenfelder an

FEHLER

`#GL_INVALID_VALUE` kann generiert werden, wenn `i` negativ ist.

`#GL_INVALID_OPERATION` wird generiert, wenn ein Pufferobjektname ungleich Null an ein aktiviertes Feld gebunden ist und der Datenspeicher des Pufferobjekts derzeit zugeordnet ist.

6.5 gl.Begin

BEZEICHNUNG

gl.Begin – begrenzt die Scheitelpunkte eines Grundelements oder einer Gruppe ähnlicher Grundelemente

ÜBERSICHT

```
gl.Begin(mode)
```

BESCHREIBUNG

`gl.Begin()` und `gl.End()` begrenzen die Scheitelpunkte, die ein Grundelement oder eine Gruppe ähnlicher Grundelemente definieren. `gl.Begin()` akzeptiert ein einzelnes Argument, das angibt, auf welche der zehn Arten die Scheitelpunkte interpretiert werden. Wenn wir n als eine ganze Zahl beginnend bei eins und N als die Gesamtzahl der angegebenen Scheitelpunkte nehmen, sind die Interpretationen wie folgt:

`#GL_POINTS`
: Behandelt jeden Scheitelpunkt als einen einzelnen Punkt. Scheitel n definiert Punkt n. n Punkte werden gezeichnet.

`#GL_LINES`
: Behandelt jedes Scheitelpunktpaar als unabhängiges Liniensegment. Die Scheitelpunkte 2^n - 1 und 2^n definieren die Linie n. N/2 Linien werden gezeichnet.

`#GL_LINE_STRIP`
: Zeichnet eine verbundene Gruppe von Liniensegmenten vom ersten zum letzten Punkt. Die Scheitelpunkte n und n + 1 definieren die Linie n. N - 1 Linien werden gezeichnet.

`#GL_LINE_LOOP`
: Zeichnet eine verbundene Gruppe von Liniensegmenten vom ersten Punkt zum letzten und dann zurück zum ersten. Die Scheitelpunkte n und n + 1 definieren die Linie n. Die letzte Linie wird jedoch durch die Scheitelpunkte N und 1 definiert. N Linien werden gezeichnet.

`#GL_TRIANGLES`
: Behandelt alle drei Scheitelpunkte als unabhängiges Dreieck. Die Scheitel 3^n - 2, 3^n - 1 und 3^n definieren das Dreieck n. N/3 Dreiecke werden gezeichnet.

`#GL_TRIANGLE_STRIP`
: Zeichnet eine verbundene Gruppe von Dreiecken. Ein Dreieck ist für jeden Scheitelpunkt definiert, der nach den ersten beiden Scheitelpunkten dargestellt wird. Ungerade n, Scheitel n, n + 1 und n + 2 definieren das Dreieck n. Für gerade n definieren die Scheitelpunkte n + 1, n und n + 2 das Dreieck n. N - 2 Dreiecke werden gezeichnet.

`#GL_TRIANGLE_FAN`
: Zeichnet eine verbundene Gruppe von Dreiecken. Ein Dreieck ist für jeden Scheitelpunkt definiert, der nach den ersten beiden Scheitelpunkten dargestellt wird. Die Scheitelpunkte 1, n + 1 und n + 2 definieren das Dreieck n. N - 2 Dreiecke werden gezeichnet.

`#GL_QUADS`
: Behandelt jede Gruppe von vier Scheitelpunkte als unabhängiges Viereck. Die Scheitel 4^n - 3, 4^n - 2, 4^n - 1 und 4^n definieren das Viereck n. N/4 Vierecke werden gezeichnet.

`#GL_QUAD_STRIP`
: Zeichnet eine verbundene Gruppe von Vierecken. Ein Viereck ist für jedes Paar von Scheitelpunkten definiert, die nach dem ersten Paar dargestellt werden. Die Scheitelpunkte 2^n - 1, 2^n, 2^n + 2 und 2^n + 1 definieren das Viereck n. N/2 - 1 Vierecke werden gezeichnet. Beachten Sie, dass sich die Reihenfolge der Eckpunkte, um ein Viereck aus Reihendaten zu konstruieren, von derjenigen unterscheidet, die bei unabhängigen Daten verwendet wird.

`#GL_POLYGON`
: Zeichnet ein einzelnes, konvexes Polygon. Die Scheitelpunkte 1 bis N definieren dieses Polygon.

Nur eine Teilmenge der GL-Befehle kann zwischen `gl.Begin()` und `gl.End()` verwendet werden. Die Befehle sind `gl.Vertex()`, `gl.Color()`, `gl.Index()`, `gl.Normal()`, `gl.TexCoord()`, `gl.Material()`, `gl.EvalCoord()`, `gl. EvalPoint()`, `gl.EdgeFlag()` und `gl.ArrayElement()`. Es ist auch möglich, `gl.CallList()` oder `gl.CallLists()` zu verwenden, um Display-Listen auszuführen, die nur die vorhergehenden Befehle enthalten. Wenn ein anderer GL-Befehl zwischen `gl.Begin()` und `gl.End()` ausgeführt wird, wird das Fehler-Flag gesetzt und der Befehl ignoriert.

Unabhängig von dem für den Modus gewählten Wert gibt es keine Begrenzung für die Anzahl der Scheitelpunkte, die zwischen `gl.Begin()` und `gl.End()` definiert werden können. Unvollständig angegebene Linien, Dreiecke, Vierecke und Polygone werden nicht gezeichnet. Unvollständige Angaben ergeben sich, wenn entweder zu wenige Scheitelpunkte bereitgestellt werden, um ein einzelnes Grundelement zu definieren oder wenn ein falsches Vielfaches von Scheitelpunkten angeben ist. Das unvollständige Grundelement wird ignoriert; der Rest wird gezeichnet.

Die Mindestangabe der Scheitelpunkte für jedes Grundelement lautet wie folgt: 1 für einen Punkt, 2 für eine Linie, 3 für ein Dreieck, 4 für ein Viereck und 3 für ein Polygon. Modi, die ein bestimmtes Vielfaches von Scheitelpunkten benötigen, sind `#GL_LINES` (2), `#GL_TRIANGLES` (3), `#GL_QUADS` (4) und `#GL_QUAD_STRIP` (2).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` Gibt das Grundelement oder die Grundelemente an, die aus den Scheiteln zwischen `gl.Begin()` und dem nachfolgenden gl.End() erzeugt werden (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `mode` auf einen nicht gültigen Wert gesetzt ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.Begin()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.End()` ohne vorangestelltes glBegin ausgeführt wird.

`#GL_INVALID_OPERATION` wird generiert, wenn zwischen `gl.Begin()` und `gl.End()` ein nicht unterstützter Befehl ausgeführt wird. In Ihrem OpenGL-Referenzhandbuch finden Sie Befehle, die zwischen `gl.Begin()` und `gl.End()` ausgeführt werden können.

6.6 gl.BindTexture

BEZEICHNUNG

gl.BindTexture – bindet die angegebene Textur an ein Texturziel

ÜBERSICHT

```
gl.BindTexture(target, texture)
```

BESCHREIBUNG

Der Aufruf von `gl.BindTexture()` mit dem Ziel `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` und dem Textur-Set mit dem Namen der neuen Textur bindet den Texturnamen an das Ziel. Wenn eine Textur an ein Ziel gebunden ist, wird die vorherige Bindung für dieses Ziel automatisch unterbrochen.

Texturnamen sind vorzeichenlose Ganzzahlen. Der Wert Null ist reserviert, um die Standardtextur für jedes Texturziel darzustellen. Texturnamen und die entsprechenden Texturinhalte sind lokal im gemeinsamen Display-Listenraum des aktuellen GL-Render-Kontextes; zwei Rendering-Kontexte teilen nur Texturnamen, wenn sie auch Display-Listen teilen.

Sie können `gl.GenTextures()` verwenden, um eine Reihe von neuen Texturnamen zu generieren.

Wenn eine Textur zuerst gebunden wird, nimmt sie die Dimensionalität des angegebene Ziels an: Eine Textur, die zuerst an `#GL_TEXTURE_1D` gebunden wird, wird zu einer eindimensionalen Textur; eine Textur, die zuerst an `#GL_TEXTURE_2D` gebunden wird, wird zu einer zweidimensionalen Textur. Der Status einer eindimensionalen Textur unmittelbar nach der ersten Bindung entspricht dem Status der Standard-#GL_TEXTURE_1D bei der GL-Initialisierung und ist für zweidimensionale Texturen ähnlich.

Wenn eine Textur gebunden ist, wirken sich GL-Operationen auf das Ziel auch auf die gebundene Textur aus und Anfragen an das aktuelle Ziel geben die Einstellungen der aktuelle dort gebundenen Textur zurück. Wenn das Textur-Mapping auf dem Ziel aktiv ist, wird die gebundene Textur genutzt. Im Endeffekt werden die Ziele damit zu anderen Namen für die aktuell dort gebundene Textur. In der Tat werden die Texturziele zu Aliasen für die aktuell an sie gebundenen Texturen und der Texturname Null bezieht sich auf die Standardtexturen, die bei der Initialisierung an sie gebunden wurden.

Eine Texturbindung, die mit `gl.BindTexture()` erstellt wurde, bleibt aktiv, bis eine andere Textur an dasselbe Ziel gebunden wird oder bis die gebundene Textur mit `gl.DeleteTextures()` gelöscht wird.

Nach der Erstellung kann die angegebene Textur so oft wie nötig an ihr ursprüngliches Ziel gebunden werden. Es ist normalerweise viel schneller, `gl.BindTexture()` zu verwenden, um die angegebene Textur an eines der Texturziele zu binden, als das Texturbild mit `gl.TexImage1D()` oder `gl.TexImage2D()` zu verwenden. Um zusätzliche Kontrolle über die Leistung zu erhalten, verwenden Sie `gl.PrioritizeTextures()`.

`gl.BindTexture()` ist in Display-Listen enthalten.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt das Ziel an, an das die Textur gebunden ist. Muss entweder `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` sein

`texture` gibt den Namen einer Textur an

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `target` keiner der zulässigen Werte ist.

`#GL_INVALID_OPERATION` wird generiert, wenn die Textur zuvor mit einem Ziel erstellt wurde, das nicht mit dem von `target` übereinstimmt.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.BindTexture()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_TEXTURE_BINDING_1D`

`gl.Get()` mit dem Argument `#GL_TEXTURE_BINDING_2D`

6.7 gl.Bitmap

BEZEICHNUNG

gl.Bitmap – zeichnet ein Bitmap

ÜBERSICHT

```
gl.Bitmap(xorig, yorig, xmove, ymove[, bitmapArray])
```

BESCHREIBUNG

Eine Bitmap ist ein Binärbild. Wenn es gezeichnet wird, wird die Bitmap relativ zu der aktuellen Rasterposition positioniert und Bildpufferpixel, die Einsen in der Bitmap entsprechen, werden unter Verwendung der aktuellen Rasterfarbe oder des aktuellen

Indexes geschrieben. Rahmenpufferpixel, die Nullen in der Bitmap entsprechen, werden nicht modifiziert.

`gl.Bitmap()` nimmt bis zu fünf Argumente auf. `xorig` und `yorig` geben die Position des Bitmap-Ursprungs relativ zur unteren linken Ecke des Bitmap-Bildes an. `xmove` und `ymove` geben die X- und Y-Offsets an, die nach dem Zeichnen der Bitmap zur aktuellen Rasterposition hinzugefügt werden sollen. Das letzte Argument `bitmapArray` ist eine Tabelle, die Pixeldaten des Bitmap-Bildes selbst enthält.

Das Bitmap-Bild wird wie Bilddaten für den Befehl `gl.DrawPixels()` interpretiert, wobei die Breite und Höhe der Bitmap den Argumenten width und height des Befehls entspricht, der Typ auf `GL_BITMAP` gesetzt wird und das Format auf `GL_COLOR_INDEX` festgelegt wird. Die mit `gl.PixelStore()` angegebenen Modi beeinflussen die Interpretation von Bitmap-Bilddaten. Modi, die mit `gl.PixelTransfer()` angegeben wurden, nicht.

Wenn die aktuelle Rasterposition ungültig ist, wird `gl.Bitmap()` ignoriert. Andernfalls wird die untere linke Ecke des Bitmapbilds an den Fensterkoordinaten xw = xr - xo und yw = yr - yo positioniert, wobei (xr, yr) die Rasterposition und (xo, yo) der Ursprung der Bitmap ist. Fragmente werden dann für jedes Pixel erzeugt, das einer 1 (eins) in dem Bitmap-Bild entspricht. Diese Fragmente werden unter Verwendung der aktuellen Raster-z-Koordinate, der Farbe oder des Farbindex und der aktuellen Rastertexturkoordinaten erzeugt. Sie werden dann so behandelt, als wären sie durch einen Punkt, eine Linie oder ein Polygon erzeugt worden, einschließlich Textur-Mapping, Nebel und alle Pixeloperatoren wie Alpha- und Tiefentests.

Nachdem die Bitmap gezeichnet wurde, werden die X- und Y-Koordinaten der aktuellen Rasterposition um `xmove` und `ymove` versetzt. Es wird keine Änderung an der z-Koordinate der aktuellen Rasterposition, an der aktuellen Rasterfarbe, den Texturkoordinaten oder dem Index vorgenommen.

Um eine gültige Rasterposition außerhalb des Darstellungsbereichs festzulegen, legen Sie zunächst eine gültige Rasterposition im Darstellungsbereich fest und rufen Sie dann `gl.Bitmap()` ohne den Bitmap-Parameter und mit `xmove` und `ymove` die Offsets der neuen Rasterposition auf. Diese Technik ist nützlich, wenn Sie ein Bild im Ansichtsfenster schwenken.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`xorig` gibt den Ort des x-Ursprungs im Bitmap-Bild an. Der Ursprung wird von der unteren linken Ecke der Bitmap aus gemessen, wobei rechts und oben die positiven Achsen sind.

`yorig` gibt den Ort des y-Ursprungs im Bitmap-Bild an. Der Ursprung wird von der unteren linken Ecke der Bitmap aus gemessen, wobei rechts und oben die positiven Achsen sind.

`xmove` gibt den X-Versatz an, der nach dem Zeichnen der Bitmap zur aktuellen Rasterposition hinzugefügt werden soll

`ymove` gibt den Y-Versatz an, der nach dem Zeichnen der Bitmap zur aktuellen Rasterposition hinzugefügt werden soll

`bitmapArray`
optional: Tabelle mit Bitmap-Daten

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn glBitmap zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_COLOR`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_SECONDARY_COLOR`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_DISTANCE`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_INDEX`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_TEXTURE_COORDS`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.8 gl.BlendFunc

BEZEICHNUNG

gl.BlendFunc – definiert die Pixelarithmetik

ÜBERSICHT

```
gl.BlendFunc(sfactor, dfactor)
```

BESCHREIBUNG

Im RGBA-Modus können Pixel mit einer Funktion gezeichnet werden, die die eingehenden (Quell-) RGBA-Werte mit den RGBA-Werten mischt, die sich bereits im Bildpuffer befinden (die Zielwerte). Mischen ist anfänglich deaktiviert. Verwenden Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_BLEND`, um die Überblendung zu aktivieren und zu deaktivieren.

`gl.BlendFunc()` definiert den Vorgang des Mischens, wenn es aktiviert ist. `sfactor` gibt an, welche der neun Methoden zum Skalieren der Quellfarbkomponenten verwendet wird. `dfactor` gibt an, welche von acht Methoden zum Skalieren der Zielfarbkomponenten verwendet wird. Die elf möglichen Methoden sind in der folgenden Tabelle beschrieben. Jede Methode definiert vier Skalierungsfaktoren, je einen für Rot, Grün, Blau und Alpha.

In der Tabelle und in den nachfolgenden Gleichungen werden Quell- und Zielfarbkomponenten als (Rs, Gs, Bs, As) und (Rd, Gd, Bd, Ad) bezeichnet. Sie sind so zu verstehen, dass sie ganzzahlige Werte zwischen 0 und (kR, kG, kB, kA) haben, wobei

```
kc = 2^mc - 1
```

und (mR, mG, mB, mA) ist die Anzahl der roten, grünen, blauen und Alpha-Bit-Ebenen.

Quell- und Zielskalierungsfaktoren werden als (sR, sG, sB, sA) und (dR, dG, dB, dA) bezeichnet. Die in der Tabelle beschriebenen Skalierungsfaktoren (fR, fG, fB, fA) repräsentieren entweder Quell- oder Zielfaktoren. Alle Skalierungsfaktoren haben einen Bereich [0,1].

```
Parameter               |               (fR, fG, fB, fA)
```

```
-----------------------------------------------------------------------
#GL_ZERO                 |                  (0, 0, 0, 0)
#GL_ONE                  |                  (1, 1, 1, 1)
#GL_SRC_COLOR            |           (Rs/kR, Gs/kG, Bs/kB, As/kA)
#GL_ONE_MINUS_SRC_COLOR | (1, 1, 1, 1) - (Rs/kR, Gs/kG, Bs/kB, As/kA)
#GL_DST_COLOR            |          (Rd/kR, Gd/kG, Bd/kB, Ad/kA)
#GL_ONE_MINUS_DST_COLOR | (1, 1, 1, 1) - (Rd/kR, Gd/kG, Bd/kB, Ad/kA)
#GL_SRC_ALPHA            |          (As/kA, As/kA, As/kA, As/kA)
#GL_ONE_MINUS_SRC_ALPHA | (1, 1, 1, 1) - (As/kA, As/kA, As/kA, As/kA)
#GL_DST_ALPHA            |          (Ad/kA, Ad/kA, Ad/kA, Ad/kA)
#GL_ONE_MINUS_DST_ALPHA | (1, 1, 1, 1) - (Ad/kA, Ad/kA, Ad/kA, Ad/kA)
#GL_SRC_ALPHA_SATURATE  |                  (i, i, i, 1)
```

In der Tabelle,

```
i = min(As, kA  - Ad) / kA
```

Um die gemischten RGBA-Werte eines Pixels beim Zeichnen im RGBA-Modus zu bestimmen, verwendet das System die folgenden Gleichungen:

```
Rd = min(kR, Rs sR + Rd dR)
Gd = min(kG, Gs sG + Gd dG)
Bd = min(kB, Bs sB + Bd dB)
Ad = min(kA, As sA + Ad dA)
```

Trotz der scheinbaren Genauigkeit der obigen Gleichungen ist die Mischarithmetik nicht genau definiert, da das Mischen mit ungenauen ganzzahligen Farbwerten arbeitet. Ein Mischfaktor, der gleich 1 sein sollte, garantiert jedoch nicht, seinen Multiplikanten zu modifizieren und ein Mischfaktor von 0 reduziert seinen Multiplikanten auf 0. Zum Beispiel, wenn `sfactor #GL_SRC_ALPHA` ist, ist `dfactor #GL_ONE_MINUS_SRC_ALPHA` und As ist gleich kA, die Gleichungen reduzieren sich auf einfachen Ersatz:

```
Rd = Rs
Gd = Gs
Bd = Bs
Ad = As
```

Die Transparenz wird am besten mit der Mischfunktion (`#GL_SRC_ALPHA`, `#GL_ONE_MINUS_SRC_ALPHA`) implementiert, wobei die Grundelemente von am weitesten zum nächsten sortiert sind. Beachten Sie, dass diese Transparenzberechnung nicht das Vorhandensein von Alpha-Bitebenen im Framepuffer erfordert. Die Blend-Funktion (`#GL_SRC_ALPHA`, `#GL_ONE_MINUS_SRC_ALPHA`) ist auch nützlich, um Antialias-Punkte und -Linien in beliebiger Reihenfolge darzustellen.

Polygon-Antialiasing wird mithilfe der Überblendungsfunktion (`#GL_SRC_ALPHA_SATURATE`, `#GL_ONE`) mit Polygonen optimiert, die von der nächsten bis zur am weitesten entfernten Seite sortiert sind. Siehe Abschnitt 6.40 [gl.Enable], Seite 74, für Informationen zum Polygon-Antialiasing. Suchen Sie nach `#GL_POLYGON_SMOOTH`) Ziel-Alpha-Bitebenen, die vorhanden sein müssen, damit diese Mischfunktion ordnungsgemäß funktioniert, und speichern Sie die akkumulierte Deckkraft.

Ankommende (Quell-) Alpha wird korrekt als eine Materialdeckkraft betrachtet, die von 1.0 (KA) reicht, was eine vollständige Deckkraft darstellt, bis zu 0.0 (0), was eine vollständige Transparenz darstellt.

Wenn mehr als ein Farbpuffer für das Zeichnen aktiviert ist, führt GL das Mischen für jeden aktivierten Puffer separat durch, wobei der Inhalt dieses Puffers für die Zielfarbe verwendet wird. Siehe Abschnitt 6.34 [gl.DrawBuffer], Seite 64, für Details.

Die Überblendung wirkt sich nur auf das RGBA-Rendering aus. Es wird von Farbindex-Renderern ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`sfactor` gibt an, wie die Mischungsfaktoren für Rot, Grün, Blau und Alpha-Quellen berechnet werden (siehe oben)

`dfactor` gibt an, wie die Mischungsfaktoren für Rot, Grün, Blau und Alpha-Ziel berechnet werden (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn entweder `sfactor` oder `dfactor` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.BlendFunc()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_BLEND_SRC`

`gl.Get()` mit dem Argument `#GL_BLEND_DST`

`gl.IsEnabled()` mit dem Argument `#GL_BLEND`

6.9 gl.CallList

BEZEICHNUNG

gl.CallList – führt eine Display-Liste aus

ÜBERSICHT

```
gl.CallList(list)
```

BESCHREIBUNG

`gl.CallList()` bewirkt, dass die angegebene Display-Liste ausgeführt wird. Die in der Display-Liste gespeicherten Befehle werden der Reihe nach ausgeführt, so als ob sie ohne Verwendung einer Display-Liste aufgerufen würden. Wenn die Liste nicht als Display-Liste definiert wurde, wird `gl.CallList()` ignoriert.

`gl.CallList()` kann in einer Display-Liste erscheinen. Um die Möglichkeit einer unendlichen Rekursion zu vermeiden, die daraus resultiert, dass Display-Listen einander aufrufen, wird die Verschachtelungsebene von Display-Listen während der Ausführung der Display-Liste begrenzt. Dieses Limit ist mindestens 64 und hängt von der Implementierung ab.

Der GL-Status wird bei einem Aufruf von `gl.CallList()` nicht gespeichert und wiederhergestellt. Somit bleiben Änderungen am GL-Status während der Ausführung einer Display-Liste auch nach Abschluss der Ausführung der Display-Listen erhalten. Verwenden Sie `gl.PushAttrib()`, `gl.PopAttrib()`, `gl.PushMatrix()` und `gl.PopMatrix()`, um den GL-Status über gl.CallList()-Aufrufe hinweg beizubehalten.

Display-Listen können zwischen einem Aufruf von `gl.Begin()` und `gl.End()` ausgeführt werden, solange die Display-Liste nur Befehle enthält, die zwischen diesen Beiden Befehlen zulässig sind.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`list` gibt den Ganzzahlnamen der auszuführenden Display-Liste an

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MAX_LIST_NESTING`

6.10 gl.CallLists

BEZEICHNUNG

gl.CallLists – führt eine Liste von Display-Listen aus

ÜBERSICHT

```
gl.CallLists(listArray)
```

BESCHREIBUNG

`gl.CallLists()` bewirkt, dass jede Display-Liste, die in der Namensliste aufgeführt ist, ausgeführt werden. Als Ergebnis werden die in jeder Display-Liste gespeicherten Befehle der Reihe nach ausgeführt, so als ob sie ohne Verwendung einer Display-Liste aufgerufen würden. Namen von Display-Listen, die nicht definiert wurden, werden ignoriert.

`gl.CallLists()` bietet so ein effizientes Mittel zum Ausführen von mehr als einer Display-Liste.

Mit dem Befehl `gl.ListBase()` wird jedem Listennamen in der Liste einen vorzeichenlosen Versatz hinzugefügt, bevor diese Display-Liste ausgeführt wird. Somit wird eine zusätzliche Ebene der Indirektion bereitgestellt.

`gl.CallLists()` kann in einer Display-Liste erscheinen. Um die Möglichkeit einer unendlichen Rekursion durch gegenseitiges Aufrufen von Display-Listen zu vermeiden, wird die Verschachtelungsebene von Display-Listen während der Ausführung der Display-Liste auf mindesten 64 begrenzt. Dies hängt allerdings von der Implementierung ab.

Der GL-Status wird bei einem Aufruf von gl.CallList() nicht gespeichert und wiederhergestellt. Somit bleiben Änderungen am GL-Status während der Ausführung einer Display-Liste auch nach Abschluss der Ausführung der Display-Listen erhalten. Verwenden Sie `gl.PushAttrib()`, `gl.PopAttrib()`, `gl.PushMatrix()` und `gl.PopMatrix()`, um den GL-Status über gl.CallList()-Aufrufe hinweg beizubehalten.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`listArray`
gibt ein Feld mit Namensversätze in der Display-Liste an

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_LIST_BASE`

`gl.Get()` mit dem Argument `#GL_MAX_LIST_NESTING`

6.11 gl.Clear

BEZEICHNUNG

gl.Clear – löscht den Puffer und stellt ihn auf voreingestellte Werte ein

ÜBERSICHT

`gl.Clear(mask)`

BESCHREIBUNG

`gl.Clear()` setzt den Bitebene-Bereich des Fensters auf Werte, die zuvor von `gl.ClearColor()`, `gl.ClearIndex()`, `gl.ClearDepth()`, `gl.ClearStencil()` und `gl.ClearAccum()` eingestellt wurden. Mehrere Farbpuffer können gleichzeitig gelöscht werden, indem mehr als ein Puffer gleichzeitig mit `gl.DrawBuffer()` ausgewählt wird.

Der Pixel-Ownership-Test, der Scherentest, Dithering und die Puffer-Schreibmasken beeinflussen die Operation von `gl.Clear()`. Die Scherenbox begrenzt den gelöschten Bereich. Alpha-Funktion, Mischfunktion, logische Operation, Schablonierung (Stencil), Textur-Mapping und Tiefenpufferung werden von `gl.Clear()` ignoriert.

`gl.Clear()` nimmt ein einzelnes Argument an, welches das bitweise Or (ODER) mehrerer Werte ist und angibt, welcher Puffer gelöscht werden soll.

Die Werte sind wie folgt:

`#GL_COLOR_BUFFER_BIT`
: Kennzeichnet die Puffer, die derzeit zum Schreiben von Farben aktiviert sind.

`#GL_DEPTH_BUFFER_BIT`
: Kennzeichnet den Tiefenpuffer.

`#GL_ACCUM_BUFFER_BIT`
: Kennzeichnet den Akkumulationspuffer.

`#GL_STENCIL_BUFFER_BIT`
: Kennzeichnet den Schablonenpuffer.

Der Wert, auf den jeder Puffer zurückgesetzt wird, hängt von der Einstellung des Löschwerts für diesen Puffer ab.

Wenn kein Puffer vorhanden ist, hat eine auf diesen Puffer gekennzeichnetes `gl.Clear()` keine Auswirkung.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mask` bitweises Or (ODER) von Masken, die die zu löschenden Puffer angeben. Die vier Masken sind `#GL_COLOR_BUFFER_BIT`, `#GL_DEPTH_BUFFER_BIT`, `#GL_ACCUM_BUFFER_BIT` und `#GL_STENCIL_BUFFER_BIT`.

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn ein anderes Bit als die vier definierten Bits in der Maske gesetzt ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Clear()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ACCUM_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_DEPTH_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_INDEX_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_COLOR_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_STENCIL_CLEAR_VALUE`

6.12 gl.ClearAccum

BEZEICHNUNG

gl.ClearAccum – gibt eindeutige Löschwerte für den Akkumulationspuffer an

ÜBERSICHT

```
gl.ClearAccum(red, green, blue, alpha)
```

BESCHREIBUNG

`gl.ClearAccum()` gibt die Rot-, Grün-, Blau- und Alphawerte an, die von `gl.Clear()` verwendet werden, um den Akkumulationspuffer zu löschen.

Die von `gl.ClearAccum()` angegebenen Werte werden auf den Bereich -1 bis 1 festgelegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`red`	definiert den Rotwert, der verwendet wird, wenn der Akkumulationspuffer gelöscht wird; der Anfangswert ist 0
`green`	definiert den Grünwert, der verwendet wird, wenn der Akkumulationspuffer gelöscht wird; der Anfangswert ist 0
`blue`	definiert den Blauwert, der verwendet wird, wenn der Akkumulationspuffer gelöscht wird; der Anfangswert ist 0
`alpha`	definiert den Alphawert, der verwendet wird, wenn der Akkumulationspuffer gelöscht wird; der Anfangswert ist 0

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ClearAccum()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ACCUM_CLEAR_VALUE`

6.13 gl.ClearColor

BEZEICHNUNG

gl.ClearColor – gibt eindeutige Löschwerte für die Farbpuffer an

ÜBERSICHT

```
gl.ClearColor(red, green, blue, alpha)
```

BESCHREIBUNG

`gl.ClearColor()` gibt die Rot-, Grün-, Blau- und Alphawerte an, die von `gl.Clear()` verwendet werden, um die Farbpuffer zu löschen. Die von `gl.ClearColor()` angegebenen Werte werden auf den Bereich 0 bis 1 festgelegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`red` definiert den Rotwert, der verwendet wird, wenn die Farbpuffer gelöscht werden; der Anfangswert ist 0

`green` definiert den Grünwert, der verwendet wird, wenn die Farbpuffer gelöscht werden; der Anfangswert ist 0

`blue` definiert den Blauwert, der verwendet wird, wenn die Farbpuffer gelöscht werden; der Anfangswert ist 0

`alpha` definiert den Alphawert, der verwendet wird, wenn die Farbpuffer gelöscht werden; der Anfangswert ist 0

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.ClearColor()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_COLOR_CLEAR_VALUE`

6.14 gl.ClearDepth

BEZEICHNUNG

gl.ClearDepth – gibt den Löschwert für den Tiefenpuffer an

ÜBERSICHT

```
gl.ClearDepth(depth)
```

BESCHREIBUNG

`gl.ClearDepth()` gibt den Tiefenwert an, der von `gl.Clear()` verwendet wird, um den Tiefenpuffer zu löschen. Die von `gl.ClearDepth()` angegebenen Werte werden auf den Bereich 0 bis 1 festgelegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`depth` gibt den Tiefenwert an, der verwendet wird, wenn der Tiefenpuffer gelöscht wird; der Anfangswert ist 1

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ClearDepth()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_DEPTH_CLEAR_VALUE`

6.15 gl.ClearIndex

BEZEICHNUNG

gl.ClearIndex – gibt den Löschwert für den Farbindexpuffer an

ÜBERSICHT

`gl.ClearIndex(c)`

BESCHREIBUNG

`gl.ClearIndex()` gibt den Index an, der von `gl.Clear()` verwendet wird, um die Farbindexpuffer zu löschen. `c` ist nicht fixiert. Statt dessen wird `c` mit unbestimmter Genauigkeit rechts neben dem Binärpunkt in einen Festkommawert umgewandelt. Der ganzzahlige Teil dieses Wertes wird dann mit 2^m-1 maskiert, wobei m die Anzahl der Bits in einem im Bildpuffer gespeicherten Farbindex ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`c` gibt den Index an, der verwendet wird, wenn die Farbindexpuffer gelöscht werden; der Anfangswert ist 0.

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.ClearIndex()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_INDEX_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_INDEX_BITS`

6.16 gl.ClearStencil

BEZEICHNUNG

gl.ClearStencil – gibt den Löschwert für den Schablonenpuffer an

ÜBERSICHT

`gl.ClearStencil(s)`

BESCHREIBUNG

`gl.ClearStencil()` gibt den Index an, der von `gl.Clear()` verwendet wird, um den Schablonenpuffer zu löschen. `s` wird mit 2^m-1 maskiert, wobei m die Anzahl der Bits im Schablonenpuffer ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`s` gibt den Index an, der verwendet wird, wenn der Schablonenpuffer gelöscht wird; der Anfangswert ist 0

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.ClearStencil()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_STENCIL_CLEAR_VALUE`

`gl.Get()` mit dem Argument `#GL_STENCIL_BITS`

6.17 gl.ClipPlane

BEZEICHNUNG

gl.ClipPlane – gibt eine Ebene an, auf die die gesamte Geometrie geschnitten wird

ÜBERSICHT

```
gl.ClipPlane(plane, equationArray)
```

BESCHREIBUNG

Geometrie wird immer gegen die Grenzen eines Stumpfes mit sechs Flächen in x, y und z geschnitten. `gl.ClipPlane()` erlaubt die Angabe zusätzlicher Flächen, die nicht unbedingt senkrecht zur x-, y- oder z-Achse stehen, gegen die die Geometrie abgeschnitten wird. Um die maximale Anzahl zusätzlicher Schnittflächen zu bestimmen, rufen Sie `gl.Get()` mit dem Argument `#GL_MAX_CLIP_PLANES` auf. Alle Implementierungen unterstützen mindestens sechs solcher Abschnittflächen. Da der resultierende Schnittbereich der Schnittpunkt der definierten Halbräume ist, ist er immer konvex.

`gl.ClipPlane()` definiert einen Halbraum unter Verwendung einer Vierkomponenten-Flächengleichung. Wenn `gl.ClipPlane()` aufgerufen wird, wird die Gleichung durch das Inverse der Modelansichtsmatrix transformiert und in den resultierenden Betrachter-Koordinaten (Augenkoordinaten) gespeichert. Nachfolgende Änderungen der Modelansichtsmatrix haben keinen Einfluss auf die gespeicherten Flächengleichungskomponenten. Wenn das Skalarprodukt (Punktprodukt) der Betrachter-Koordinaten eines Scheitelpunkts mit den gespeicherten Flächengleichungskomponenten positiv oder Null ist, ist der Scheitelpunkt innerhalb dieser Begrenzungsfläche, ansonsten ist er ausserhalb.

Um Schnittflächen zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_CLIP_PLANEi` auf, wobei i die Flächennummer ist.

Alle Schnittflächen sind anfänglich als (0, 0, 0, 0) in Augenkoordinaten definiert und deaktiviert.

Es ist immer so, dass `#GL_CLIP_PLANEi` = `#GL_CLIP_PLANE0 +` i ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`plane` gibt an, welche Schnittflächen positioniert wird; symbolische Konstanten der Form `#GL_CLIP_PLANEi` werden akzeptiert, wobei i eine Ganzzahl zwischen 0 und `#GL_MAX_CLIP_PLANES` -1 ist

`equationArray` gibt ein Feld mit vier Gleitkommawerten mit doppelter Genauigkeit an; diese Werte werden als Flächengleichung interpretiert

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn die Fläche kein akzeptierter Wert ist

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.ClipPlane()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetClipPlane()`

`gl.IsEnabled()` mit dem Argument `#GL_CLIP_PLANEi`

6.18 gl.Color

BEZEICHNUNG

gl.Color – stellt die aktuelle Farbe ein

ÜBERSICHT

```
gl.Color(red, green, blue[, alpha])
```

BESCHREIBUNG

GL speichert sowohl einen aktuellen einwertigen Farbindex als auch eine aktuelle vierwertige RGBA-Farbe. `gl.Color()` setzt eine neue vierwertige RGBA-Farbe. Wenn das optionale Argument `alpha` weggelassen wird, wird es auf 1.0 gesetzt.

Aktuelle Farbwerte werden im Gleitkommaformat gespeichert, so dass der größte darstellbare Wert auf 1.0 (volle Intensität) und 0 auf 0.0 (Null-Intensität) abgebildet wird.

Alternativ können Sie auch eine Tabelle übergeben, die drei oder vier Gleitkommawerte enthält, die die Werte für Rot, Grün, Blau und Alpha für die Farbe angeben.

Der Anfangswert für die aktuelle Farbe ist (1, 1, 1, 1).

Die aktuelle Farbe kann jederzeit aktualisiert werden. Insbesondere kann `gl.Color()` zwischen einem Aufruf von `gl.Begin()` und `gl.End()` aufgerufen werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`red`	gibt einen neuen roten Wert für die aktuelle Farbe an
`green`	gibt einen neuen grünen Wert für die aktuelle Farbe an
`blue`	gibt einen neuen blauen Wert für die aktuelle Farbe an
`alpha`	optional: Gibt einen neuen Alpha-Wert für die aktuelle Farbe an (standardmäßig 1.0)

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_COLOR`

`gl.Get()` mit dem Argument `#GL_RGBA_MODE`

6.19 gl.ColorMask

BEZEICHNUNG

gl.ColorMask – aktiviert und deaktiviert das Schreiben der Farbkomponenten in den Einzelbildpuffer

ÜBERSICHT

```
gl.ColorMask(red, green, blue, alpha)
```

BESCHREIBUNG

`gl.ColorMask()` gibt an, ob die einzelnen Farbkomponenten im Einzelbildpuffer (Framepuffer) geschrieben werden können oder nicht. Wenn rot `#GL_FALSE` ist, wird beispielsweise an der roten Komponente eines Pixels in einem der Farbpuffer unabhängig von der versuchten Zeichenoperation keine Änderung vorgenommen. Die Anfangswerte sind alle `#GL_TRUE`, was anzeigt, dass die Farbkomponenten geschrieben werden können.

Änderungen an einzelnen Komponentenbits können nicht gesteuert werden. Stattdessen werden Änderungen für ganze Farbkomponenten aktiviert oder deaktiviert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`red`	gibt an, ob Rot in den Einzelbildpuffer geschrieben werden kann oder nicht
`green`	gibt an, ob Grün in den Einzelbildpuffer geschrieben werden kann oder nicht
`blue`	gibt an, ob Blau in den Einzelbildpuffer geschrieben werden kann oder nicht
`alpha`	gibt an, ob Alpha in den Einzelbildpuffer geschrieben werden kann oder nicht

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.ColorMask()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_COLOR_WRITEMASK`

`gl.Get()` mit dem Argument `#GL_RGBA_MODE`

6.20 gl.ColorMaterial

BEZEICHNUNG

gl.ColorMaterial – bestimmt, welche Materialparameter sich auf die aktuelle Farbe auswirken

ÜBERSICHT

```
gl.ColorMaterial(face, mode)
```

BESCHREIBUNG

`gl.ColorMaterial()` gibt an, welche Materialparameter sich auf die aktuelle Farbe auswirkt. Wenn `#GL_COLOR_MATERIAL` aktiviert ist, wirken sich die in `mode` angegebenen Materialparameter oder Parameter, die in `face` angegebe Materials oder der angegebenen Materialien, zu jeder Zeit auf die aktuelle Farbe. `face` kann auf `#GL_FRONT`, `#GL_BACK` oder `#GL_FRONT_AND_BACK` gesetzt werden. Der Anfangswert ist `#GL_FRONT_AND_BACK`.

Die folgenden Werte können im Parameter `mode` übergeben werden: `#GL_EMISSION`, `#GL_AMBIENT`, `#GL_DIFFUSE`, `#GL_SPECULAR` und `#GL_AMBIENT_AND_DIFFUSE`. Der Anfangswert ist `#GL_AMBIENT_AND_DIFFUSE`.

Um `#GL_COLOR_MATERIAL` zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_COLOR_MATERIAL` auf. `#GL_COLOR_MATERIAL` ist anfangs deaktiviert.

`gl.ColorMaterial()` ermöglicht es, eine Untermenge von Materialparametern für jeden Scheitel nur mit dem Befehl `gl.Color()` zu ändern, ohne `gl.Material()` aufzurufen. Wenn nur eine solche Teilmenge von Parametern für jeden Scheitel angegeben werden soll, ist der Aufruf von `gl.ColorMaterial()` besser als der Aufruf von `gl.Material()`.

Rufen Sie `gl.ColorMaterial()` vor dem Aktivieren von `#GL_COLOR_MATERIAL` auf.

Durch den Aufruf von `gl.DrawElements()` oder `gl.DrawArrays()` kann die aktuelle Farbe unbestimmt bleiben, wenn das Farb-Feld aktiviert ist. Wenn `gl.ColorMaterial()` aktiviert ist, während die aktuelle Farbe unbestimmt ist, ist der durch `face` und `mode` angegebene Status des Beleuchtungsmaterials ebenfalls unbestimmt.

Wenn die GL-Version 1.1 oder höher ist und `#GL_COLOR_MATERIAL` aktiviert ist, wirken sich ausgewertete Farbwerte auf die Ergebnisse der Beleuchtungsgleichung so aus, als ob die aktuelle Farbe geändert würde, aber der Beleuchtungsparametern der aktuellen Farbe wird nicht geändert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`face` gibt an, ob nur Vorder-, nur Rückseite- oder beide Seiten-Materialparameter die aktuelle Farbe beeinflussen sollen

`mode` gibt an, welche von mehreren Materialparametern die aktuelle Farbe beeinflussen (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `face` oder der Modus kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ColorMaterial()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_COLOR_MATERIAL`

`gl.Get()` mit dem Argument `#GL_COLOR_MATERIAL_PARAMETER`

`gl.Get()` mit dem Argument `#GL_COLOR_MATERIAL_FACE`

6.21 gl.ColorPointer

BEZEICHNUNG

gl.ColorPointer – definiert ein Feld von Farben

ÜBERSICHT

```
gl.ColorPointer(colorArray[, size])
```

BESCHREIBUNG

`gl.ColorPointer()` gibt ein Feld von Farbkomponenten an, die beim Rendern verwendet werden sollen. `colorArray` kann entweder eine eindimensionale Tabelle sein, die aus einer beliebigen Anzahl von aufeinanderfolgenden Farbwerten besteht, oder eine zweidimensionale Tabelle, die aus einer beliebigen Anzahl von Untertabellen besteht, die jeweils 3 oder 4 Farbwerte enthalten. Wenn `colorArray` eine eindimensionale Tabelle ist, müssen

Sie auch das optionale Argument `size` übergeben, um die Größe jeder Farbkomponente in `colorArray` zu definieren. `size` muss entweder 3 oder 4 sein. Wenn `colorArray` eine zweidimensionale Tabelle ist, wird `size` automatisch durch die Anzahl der Elemente in der ersten Untertabelle bestimmt, die entweder 3 oder 4 sein muss.

Beachten Sie bei der Verwendung einer zweidimensionalen Tabelle, dass die Anzahl der Farbwerte in jeder Untertabelle konstant sein muss. In den einzelnen Untertabellen dürfen keine unterschiedlichen Farbwerte verwendet werden. Die Anzahl der Farbwerte wird durch die Anzahl der Elemente in der ersten Untertabelle definiert und alle folgenden Untertabellen müssen die gleiche Anzahl an Farbwerten verwenden.

Wenn Sie `Nil` in `colorArray` übergeben, wird der Farbfeldpuffer freigegeben, aber er wird nicht aus OpenGL entfernt. Sie müssen dies manuell tun, z.B. indem Sie das Farbfeld deaktivieren oder ein neues definieren.

Um das Farbfeld zu aktivieren und zu deaktivieren, rufen Sie `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_COLOR_ARRAY` auf. Wenn es aktiviert ist, wird das Farbfeld verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Das Farbfeld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.ColorPointer()` ist zwischen `gl.Begin()` und `gl.End()` nicht erlaubt, aber ein Fehler kann oder kann nicht erzeugt werden. Wenn kein Fehler generiert wird, ist die Operation nicht definiert.

`gl.ColorPointer()` wird typischerweise auf der Klient-Seite implementiert.

Farbfeld-Parameter sind Klientseitig und werden daher nicht von `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt. Verwenden Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`colorArray`
: ein- oder zweidimensionale Tabelle mit Farbwerten oder `Nil` (siehe oben)

`size`
: optional: Größe jeder Farbkomponente; muss entweder 3 oder 4 sein und wird nur mit eindimensionalen Tabellen verwendet (siehe oben)

FEHLER

`#GL_INVALID_VALUE` wird generiert, wenn die Größe nicht 3 oder 4 ist.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_COLOR_ARRAY`

`gl.Get()` mit dem Argument `#GL_COLOR_ARRAY_SIZE`

`gl.Get()` mit dem Argument `#GL_COLOR_ARRAY_TYPE`

`gl.Get()` mit dem Argument `#GL_COLOR_ARRAY_STRIDE`

`gl.Get()` mit dem Argument `#GL_COLOR_ARRAY_POINTER`

6.22 gl.CopyPixels

BEZEICHNUNG

gl.CopyPixels – kopiert Pixel in den Bildspeicher

ÜBERSICHT

```
gl.CopyPixels(x, y, width, height, type)
```

BESCHREIBUNG

`gl.CopyPixels()` kopiert ein bildschirmausgerichtetes Rechteck von Pixeln von dem angegebenen Bildpufferspeicherposition (Framebufferposition) in einen Bereich relativ zur aktuellen Rasterposition. Seine Operation ist nur dann gut definiert, wenn der gesamte Pixelquellenbereich innerhalb des exponierten Teils des Fensters liegt. Ergebnisse von Kopien außerhalb des Fensters oder von nicht exponierten Bereichen des Fensters sind hardwareabhängig und nicht definiert.

`x` und `y` gibt die Fensterkoordinaten der unteren linken Ecke des zu kopierenden rechteckigen Bereichs an. `width` und `height` geben die Abmessungen des rechteckigen Bereichs an, der kopiert werden soll. Sowohl die Breite als auch die Höhe dürfen nicht negativ sein.

Mehrere Parameter steuern die Verarbeitung der Pixeldaten während des Kopierens. Diese Parameter werden mit drei Befehlen festgelegt: `gl.PixelTransfer()`, `gl.PixelMap()` und `gl.PixelZoom()`. Diese Referenzseite beschreibt die Auswirkungen der meisten, aber nicht aller Parameter von `gl.CopyPixels()` auf die von diesen drei Befehlen angegebenen Parameter.

`gl.CopyPixels()` kopiert Werte von jedem Pixel mit der unteren linken Ecke bei (x+i, y+j) für 0 <= i < Breite und 0 <= j < Höhe. Dieses Pixel wird als das i-te Pixel in der j-ten Reihe bezeichnet. Pixel werden in Reihenreihenfolge von der niedrigsten in die höchste Zeile kopiert, von links nach rechts in jeder Zeile.

`type` gibt an, ob Farb-, Tiefen- oder Schablonendaten kopiert werden sollen. Die Details der Übertragung für jeden Datentyp lauten wie folgt:

`#GL_COLOR`

Indizes oder RGBA-Farben werden aus dem aktuell als Lese-Quellfarbpuffer angegebenen Puffer gelesen. Siehe Abschnitt 6.121 [gl.ReadBuffer], Seite 188, für Details. Wenn sich GL im Farbindexmodus befindet, wird jeder Index, der aus diesem Puffer gelesen wird, in ein Festkommaformat mit einer nicht spezifizierten Anzahl von Bits rechts vom Binärpunkt konvertiert. Jeder Index wird dann um #GL_INDEX_SHIFT-Bits nach links verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_INDEX_SHIFT` negativ ist, wird die Verschiebung nach rechts vorgenommen. In beiden Fällen füllen Null-Bits ansonsten nicht definierte Bitstellen im Ergebnis. Wenn `#GL_MAP_COLOR True` ist, wird der Index durch den Wert ersetzt, auf den er in der Nachschlagetabelle `#GL_PIXEL_MAP_I_TO_I` verweist. Unabhängig davon, ob der Nachschlageaustausch des Indexes durchgeführt wird oder nicht, wird der ganzzahlige Teil des Index dann mit 2^b-1 AND-verknüpft, wobei b die Anzahl der Bits in einem Farbindexpuffer ist.

Wenn sich GL im RGBA-Modus befindet, werden die Rot-, Grün-, Blau- und Alpha-Komponenten jedes gelesenen Pixels in ein internes Gleitkommafor-

mat mit unbestimmter Genauigkeit konvertiert. Bei der Konvertierung wird der Wert der größten darstellbaren Komponente auf 1.0 und der Komponentenwert auf 0 bis 0.0 abgebildet. Die resultierenden Gleitkomma-Farbwerte werden dann mit `#GL_c_SCALE` multipliziert und zu `#GL_c_BIAS` hinzugefügt, wobei c für die jeweiligen Farbkomponenten ROT, GRÜN, BLAU und ALPHA ist. Die Ergebnisse werden auf den Bereich [0,1] festgelegt. Wenn `#GL_MAP_COLOR True` ist, wird jede Farbkomponente durch die Größe der Nachschlagetabelle `#GL_PIXEL_MAP_c_TO_c` skaliert und dann durch den Wert ersetzt, auf den sie in dieser Tabelle verweist. c ist R, G, B oder A.

Wenn die Erweiterung ARB_imaging unterstützt wird, können die Farbwerte zusätzlich durch Farbtabellen-Nachschlagewerke, Farbmatrixtransformationen und Faltungsfilter verarbeitet werden.

GL wandelt dann die resultierenden Indizes oder RGBA-Farben in Fragmente um, indem er die aktuellen Rasterpositionskoordinaten- und Texturkoordinaten an jedes Pixel anfügt und dann Fensterkoordinaten (xr+i, yr+j) zuweist, wobei (xr, yr) die aktuelle Rasterposition ist und das Pixel war das i-te Pixel in der j-ten Reihe. Diese Pixelfragmente werden dann genauso behandelt wie die Fragmente, die durch Rasterung von Punkten, Linien oder Polygonen erzeugt werden. Textur-Abbildungen, Nebel und alle Fragment-Operationen werden angewendet, bevor die Fragmente in den Bildpuffer geschrieben werden.

`#GL_DEPTH`

Tiefenwerte werden aus dem Tiefenpuffer gelesen und direkt in ein internes Gleitkommaformat mit unbestimmter Genauigkeit konvertiert. Der resultierende Gleitkommatiefenwert wird dann mit `#GL_DEPTH_SCALE` multipliziert und zu `#GL_DEPTH_BIAS` hinzugefügt. Das Ergebnis wird auf den Bereich [0,1] festgelegt.

GL wandelt dann die resultierenden Tiefenkomponenten in Fragmente um, indem er die aktuelle Rasterpositionsfarbe oder Farbindex- und Texturkoordinaten an jedes Pixel anfügt und dann Fensterkoordinaten (xr+i, yr+j) zuweist, wobei (xr, yr) die aktuelle Rasterposition ist und das Pixel war das i-te Pixel in der j-ten Reihe. Diese Pixelfragmente werden dann genauso behandelt wie die Fragmente, die durch Rasterung von Punkten, Linien oder Polygonen erzeugt werden. Textur-Abbildungen, Nebel und alle Fragment-Operationen werden angewendet, bevor die Fragmente in den Bildpuffer geschrieben werden.

`#GL_STENCIL`

Schablonenindizes werden aus dem Schablonenpuffer gelesen und in ein internes Festkommaformat mit einer unbestimmten Anzahl von Bits rechts vom Binärpunkt umgewandelt. Jeder Festkomma-Index wird dann um #GL_INDEX_SHIFT-Bits nach links verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_INDEX_SHIFT` negativ ist, wird die Verschiebung nach rechts vorgenommen. In beiden Fällen füllen Null-Bits ansonsten nicht spezifizierte Bitstellen im Ergebnis. Wenn `#GL_MAP_STENCIL True` ist, wird der Index durch den Wert ersetzt,

auf den er in der Nachschlagetabelle #GL_PIXEL_MAP_S_TO_S verweist. Unabhängig davon, ob der Nachschlageaustausch des Indexes durchgeführt wird oder nicht, wird der ganzzahlige Teil des Index dann mit 2^b-1 AND-verknüpft, wobei b die Anzahl der Bits im Schablonenpuffer ist. Die resultierenden Schablonenindizes werden dann in den Schablonenpuffer geschrieben, so dass der Index, der von der i-ten Stelle der j-ten Reihe gelesen wird, in den Ort (xr+i, yr+j) geschrieben wird, wobei (xr, yr) die aktuelle Rasterposition ist. Nur der Pixelbesitztest, der Scherentest und die Schablonen-Schreibmaske beeinflussen diese Schreiboperationen.

Die bisher beschriebene Rasterung setzt Pixelzoomfaktoren von 1.0 voraus. Wenn gl.PixelZoom() verwendet wird, um die Zoomfaktorfaktoren x und y zu ändern, werden die Pixel wie folgt in Fragmente konvertiert. Wenn (xr, yr) die aktuelle Rasterposition ist und ein gegebenes Pixel an der i-ten Stelle in der j-ten Reihe des Quellpixelrechtecks liegt, dann werden Fragmente für Pixel erzeugt, deren Mitten in dem Rechteck mit Ecken bei

```
(xr + zoomx_i, yr + zoomy_j)
```

und

```
(xr + zoomx_(i + 1), yr + zoomy_(j + 1))
```

ligen. Dabei ist zoomx der Wert von #GL_ZOOM_X und zoomy ist der Wert von #GL_ZOOM_Y.

Die mit gl.PixelStore() angegebenen Modi haben keinen Einfluss auf die Operation des Befehls gl.CopyPixels().

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

x	gibt die x-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an
y	gibt die y-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an
width	definiert die Abmessungen des rechteckigen Bereichs der zu kopierenden Pixel; beide müssen nicht negativ sein
height	definiert die Abmessungen des rechteckigen Bereichs der zu kopierenden Pixel; beide müssen nicht negativ sein
type	gibt an, ob Farbwerte, Tiefenwerte oder Schablonenwerte kopiert werden sollen; die symbolischen Konstanten #GL_COLOR, #GL_DEPTH und #GL_STENCIL werden akzeptiert

FEHLER

#GL_INVALID_ENUM wird generiert, wenn der Typ kein akzeptierter Wert ist.

#GL_INVALID_VALUE wird generiert, wenn Breite oder Höhe negativ sind.

#GL_INVALID_OPERATION wird generiert, wenn der Typ #GL_DEPTH ist und kein Tiefenpuffer vorhanden ist.

#GL_INVALID_OPERATION wird generiert, wenn der Typ #GL_STENCIL ist und kein Schablonenpuffer vorhanden ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.CopyPixels()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.23 gl.CopyTexImage

BEZEICHNUNG

gl.CopyTexImage – kopiert Pixel in ein Texturbild

ÜBERSICHT

```
gl.CopyTexImage(level, internalformat, border, x, y, width[, height])
```

BESCHREIBUNG

`gl.CopyTexImage()` definiert ein ein- oder zweidimensionales Texturbild mit Pixeln aus dem aktuellen `#GL_READ_BUFFER`. Wenn das optionale Argument `height` weggelassen wird, wird eine eindimensionale Textur definiert, andernfalls wird eine zweidimensionale Textur definiert.

Das bildschirmausgerichtete Pixelrechteck mit der unteren linken Ecke bei (x, y) und mit einer Breite von Breite+2*Rand und einer Höhe von Höhe+2*Rand definiert das Texturfeld auf dem Mipmap-Level, welches durch `level` definiert ist.

`internalformat` definiert das interne Format des Texturfeldes. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. Beachten Sie, dass im Gegensatz zu `gl.TexImage1D()` und `gl.TexImage2D()` die Werte 1, 2, 3 und 4 nicht von dem Parameter `internalformat` mit `gl.CopyTexImage()` unterstützt werden.

Die Pixel im Rechteck werden genau so verarbeitet, als wäre `gl.CopyPixels()` aufgerufen worden, aber der Prozess stoppt kurz vor der endgültigen Konvertierung. An diesem Punkt werden alle Pixelkomponentenwerte auf den Bereich [0,1] festgelegt und dann in das interne Format der Textur zum Speichern in dem Texel-Feld umgewandelt.

Die Pixelordnung ist derart, dass niedrigere x- und y-Bildschirmkoordinaten niedrigeren s- und t-Texturkoordinaten entsprechen.

Wenn eines der Pixel innerhalb des angegebenen Rechtecks des aktuellen `#GL_READ_BUFFER` außerhalb des Fensters liegt, das dem aktuellen Renderkontext zugeordnet ist, sind die für diese Pixel erhaltenen Werte nicht definiert.

Die Texturierung hat im Farbindexmodus keine Wirkung.

Ein Bild mit einer Höhe oder Breite von 0 zeigt eine `NULL-Textur` an.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level` gibt die Detaillierungsstufe an; Level 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild

`internalformat`
definiert das interne Format der Textur; muss eine der Pixelformatkonstanten sein (siehe oben)

`border` legt die Breite des Rahmens fest; muss entweder 0 oder 1 sein

`x` gibt die x-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an

`y` gibt die y-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an

`width` legt die Breite des Texturbildes fest. Muss 0 oder 2^n+2*Rand für eine ganze Zahl n sein

`height` optional: Gibt die Höhe des Texturbildes an. Muss 0 oder 2^n+2*Rand für eine ganze Zahl n sein (voreingestellt ist 1)

FEHLER

`#GL_INVALID_VALUE` wird generiert, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` wird generiert werden, wenn `level` größer als log2(max) ist, wobei `max` der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird generiert, wenn `internalformat` kein zulässiger Wert ist.

`#GL_INVALID_VALUE` wird generiert, wenn `width` kleiner als 0 oder größer als 2 + `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird generiert, wenn Not-Power-of-Two-Texturen nicht unterstützt werden und die Breite nicht 2^n+2*Rand für einen ganzzahligen Wert von `n` dargestellt werden kann.

`#GL_INVALID_VALUE` wird generiert, wenn der Rand nicht 0 oder 1 ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.CopyTexImage()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_2D` oder `#GL_TEXTURE_1D`

6.24 gl.CopyTexSubImage

BEZEICHNUNG

gl.CopyTexSubImage – kopiert ein zweidimensionales Textur-Teilbild

ÜBERSICHT

```
gl.CopyTexSubImage(level, x, y, xoffset, width[, yoffset, height])
```

BESCHREIBUNG

`gl.CopyTexSubImage()` ersetzt einen rechteckigen Teil eines ein- oder zweidimensionalen Texturbildes durch Pixel aus dem aktuellen `#GL_READ_BUFFER` (statt aus dem Hauptspeicher, wie es bei `gl.TexSubImage2D()` der Fall ist. Wenn die letzten beiden Argumente weggelassen werden, wird ein rechteckiger Abschnitt eines eindimensionalen Texturbildes ersetzt, andernfalls ist ein zweidimensionales Texturbild das Ziel.

Das bildschirmausgerichtete Pixelrechteck mit der linken, unteren Ecke bei (x, y), mit der Breite `width` und der Höhe `height` ersetzt die Pixel des Texturfeldes mit den `x`-Indizes

von `xoffset` bis und mit `xoffset + width` - 1 und `y-Indizes` von `yoffset` bis und mit `yoffset + height` - 1, auf der durch `level` festgelegten Mipmap-Ebene.

Die Pixel im Rechteck werden genau so verarbeitet, als wäre `gl.CopyPixels()` aufgerufen worden, aber der Prozess stoppt kurz vor der endgültigen Konvertierung. An diesem Punkt werden alle Pixelkomponentenwerte auf den Bereich [0,1] festgelegt und dann in das interne Format der Textur zum Speichern in dem Texel-Feld umgewandelt.

Das Zielrechteck im Texturfeld darf keine Texel außerhalb des Texturfelder enthalten, wie es ursprünglich festgelegt wurde. Es ist kein Fehler, eine Subtextur mit der Breite oder Höhe null anzugeben, aber eine solche Angabe hat keine Auswirkung.

Wenn eines der Pixel innerhalb des angegebenen Rechtecks des aktuellen `#GL_READ_BUFFER` außerhalb des Lesefensters liegt, das dem aktuellen Renderkontext zugeordnet ist, sind die für diese Pixel erhaltenen Werte nicht definiert.

Es wird keine Änderung an den Parametern `internalformat`, `width`, `height`, oder `border` der angegebenen Texturfeldern oder an Texelwerten außerhalb der angegebenen Teilregion vorgenommen.

Die Texturierung hat im Farbindexmodus keine Wirkung.

Die Modi `gl.PixelStore()` und `gl.PixelTransfer()` beeinflussen Texturbilder genau so, wie sich `gl.DrawPixels()` auswirkt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level`	gibt die Detaillierungsstufe an; Level 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild
`x`	gibt die x-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an
`y`	gibt die y-Koordinate der unteren linken Ecke des rechteckigen Bereichs der zu kopierenden Pixel an
`xoffset`	gibt einen Texel-Versatz in der x-Richtung innerhalb des Texturfeldes an
`yoffset`	optional: Gibt einen Texel-Versatz in der y-Richtung innerhalb des Texturfeldes an
`width`	legt die Breite des Textur-Teilbildes fest
`height`	optional: Gibt die Höhe des Textur-Teilbilds an

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn das Texturfeld nicht durch eine vorherige `gl.TexImage2D()` oder `gl.CopyTexImage()` -Operation definiert wurde.

`#GL_INVALID_VALUE` wird generiert, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` kann generiert werden, wenn level `>` log2(max), wobei `max` der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird generiert, wenn xoffset `<`-b, xoffset `+` width`>` w-b, yoffset `<`-b oder yoffset `+` height`>` h -b ist, wobei `w` die `#GL_TEXTURE_WIDTH` ist, `h` die `#GL_TEXTURE_HEIGHT` und `b` `#GL_TEXTURE_BORDER` des Texturbildes ist, das geändert wird. Beachten Sie, dass `w` und `h` die doppelte Rahmenbreite enthalten.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.CopyTexSubImage()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_2D` oder `#GL_TEXTURE_1D`

6.25 gl.CullFace

BEZEICHNUNG

gl.CullFace – gibt an, ob nach vorne oder nach hinten gerichtete Flächen ausgewählt werden können

ÜBERSICHT

`gl.CullFace(mode)`

BESCHREIBUNG

`gl.CullFace()` gibt an, ob Vorder- oder Rückseitenflächen (wie in `mode` angegeben) ausgewählt werden, wenn die Flächenausblendung aktiviert ist. Flächenausblendung ist zunächst deaktiviert. Um die Flächenausblendung zu aktivieren und zu deaktivieren, rufen Sie die Befehle `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_CULL_FACE` auf. Zu den Flächen gehören Dreiecke, Vierecke, Polygone und Rechtecke.

`gl.FrontFace()` gibt an, welche der Flächen im Uhrzeigersinn und Gegenuhrzeigersinn nach vorne und nach hinten weisen. Siehe Abschnitt 6.53 [gl.FrontFace], Seite 86, für Details.

Wenn der Modus `#GL_FRONT_AND_BACK` ist, werden keine Flächen dargestellt, aber andere Grundmuster wie Punkte und Linien werden gezeichnet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt an, ob nach vorne oder nach hinten gerichtete Flächen aussortiert werden; die symbolischen Konstanten `#GL_FRONT`, `#GL_BACK` und `#GL_FRONT_AND_BACK` werden akzeptiert; der Anfangswert ist `#GL_BACK`

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.CullFace()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_CULL_FACE`

`gl.Get()` mit dem Argument `#GL_CULL_FACE_MODE`

6.26 gl.DeleteLists

BEZEICHNUNG

gl.DeleteLists – löscht eine zusammenhängende Gruppe von Display-Listen

ÜBERSICHT

`gl.DeleteLists(list, range)`

BESCHREIBUNG

`gl.DeleteLists()` bewirkt, dass eine zusammenhängende Gruppe von Display-Listen gelöscht wird. `list` ist der Name der ersten zu löschenden Display-Liste und `range` ist die Anzahl der zu löschenden Display-Listen. Alle Display-Listen d mit der Liste <= d <= list + range - 1 werden gelöscht.

Alle Speicherorte, die den angegebenen Display-Listen zugewiesen sind, werden freigegeben und die Namen sind zu einem späteren Zeitpunkt für die Wiederverwendung verfügbar. Namen innerhalb des Bereichs, denen keine Display-Liste zugeordnet ist, werden ignoriert. Wenn der Bereich 0 ist, passiert nichts.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`list` gibt den Ganzzahlnamen der ersten zu löschenden Display-Liste an

`range` gibt die Anzahl der zu löschenden Display-Listen an

FEHLER

`#GL_INVALID_VALUE` wird generiert, wenn der Bereich negativ ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.DeleteLists()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.27 gl.DeleteTextures

BEZEICHNUNG

gl.DeleteTextures – löscht die angegeben Texturen

ÜBERSICHT

`gl.DeleteTextures(texturesArray)`

BESCHREIBUNG

`gl.DeleteTextures()` löscht alle Texturen, die in der Tabelle `texturesArray` übergeben wurden. Nachdem eine Textur gelöscht wurde, hat sie keine Inhalte oder Dimensionalität mehr und ihr Name ist zur Wiederverwendung frei (zum Beispiel durch `gl.GenTextures()`). Wenn eine Textur, die gerade gebunden ist, gelöscht wird, wird die Bindung auf 0 zurückgesetzt (die Standardtextur).

`gl.DeleteTextures()` ignoriert implizit Nullen und Namen, die nicht mit vorhandenen Texturen übereinstimmen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`texturesArray`
gibt ein Feld von Texturen an, die gelöscht werden sollen

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn glDeleteTextures zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN
`gl.IsTexture()`

6.28 gl.DepthFunc

BEZEICHNUNG
gl.DepthFunc – gibt den Wert an, der für Tiefenpuffervergleiche verwendet wird

ÜBERSICHT
`gl.DepthFunc(func)`

BESCHREIBUNG
`gl.DepthFunc()` gibt die Funktion an, mit der jeder eingehende Pixeltiefenwert mit dem im Tiefenpuffer vorhandenen Tiefenwert verglichen wird. Der Vergleich wird nur durchgeführt, wenn der Tiefentest aktiviert ist. (Siehe `gl.Enable()` und `gl.Disable()` mit `#GL_DEPTH_TEST`)

`func` legt die Bedingungen fest, unter denen das Pixel gezeichnet wird. Die Vergleichsfunktionen sind wie folgt:

`#GL_NEVER`
: Besteht den Tiefentest niemals.

`#GL_LESS`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert kleiner als der gespeicherte Tiefenwert ist.

`#GL_EQUAL`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert gleich dem gespeicherten Tiefenwert ist.

`#GL_LEQUAL`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert kleiner oder gleich dem gespeicherten Tiefenwert ist.

`#GL_GREATER`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert größer als der gespeicherte Tiefenwert ist.

`#GL_NOTEQUAL`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert nicht gleich dem gespeicherten Tiefenwert ist.

`#GL_GEQUAL`
: Besteht den Tiefentest, wenn der eingehende Tiefenwert größer oder gleich dem gespeicherten Tiefenwert ist.

`#GL_ALWAYS`
: Besteht den Tiefentest immer.

Der Anfangswert von `func` ist `#GL_LESS`. Anfänglich ist der Tiefentest deaktiviert. Wenn der Tiefentest deaktiviert ist oder wenn kein Tiefenpuffer existiert, ist es so, als würde der Tiefentest immer bestanden.

Selbst wenn der Tiefenpuffer existiert und die Tiefenmaske nicht Null ist, wird der Tiefenpuffer nicht aktualisiert, wenn der Tiefentest deaktiviert ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`func` gibt die Tiefentestfunktion an (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `func` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.DepthFunc()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_DEPTH_FUNC`

`gl.IsEnabled()` mit dem Argument `#GL_DEPTH_TEST`

6.29 gl.DepthMask

BEZEICHNUNG

gl.DepthMask – aktiviert oder deaktiviert das Schreiben in den Tiefenpuffer

ÜBERSICHT

```
gl.DepthMask(flag)
```

BESCHREIBUNG

`gl.DepthMask()` gibt an, ob der Tiefenpuffer zum Schreiben aktiviert ist. Wenn das Flag `#GL_FALSE` ist, ist das Schreiben von Tiefenpuffer deaktiviert. Andernfalls ist es aktiviert. Anfänglich ist das Schreiben von Tiefenpuffer aktiviert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`flag` gibt an, ob der Tiefenpuffer zum Schreiben aktiviert ist; Wenn das Flag `#GL_FALSE` ist, ist das Schreiben von Tiefenpuffern deaktiviert, andernfalls ist es aktiviert; Anfangs ist das Schreiben des Tiefenpuffers aktiviert (`#GL_TRUE`)

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.DepthMask()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_DEPTH_WRITEMASK`

6.30 gl.DepthRange

BEZEICHNUNG

gl.DepthRange – gibt der Zuordnung von Tiefenwerten von normalisierten Gerätekoordinaten zu Fensterkoordinaten an

ÜBERSICHT

```
gl.DepthRange(zNear, zFar)
```

BESCHREIBUNG

Nach dem Abschneiden und Teilen durch w reichen die Tiefenkoordinaten von -1 bis 1, was der nahen und fernen Schnittebene entspricht. `gl.DepthRange()` gibt eine lineare Abbildung der normalisierten Tiefenkoordinaten in diesem Bereich auf Fenstertiefenkoordinaten an. Unabhängig von der tatsächlichen Implementierung des Tiefenpuffers werden die Werte für die Fensterkoordinatentiefe so behandelt, als ob sie von 0 bis 1 reichen (wie Farbkomponenten). Daher werden die von `gl.DepthRange()` akzeptierten Werte in diesen Bereich festgelegt, bevor sie akzeptiert werden.

Die Einstellung von (0,1) bildet die nahe Ebene auf 0 und die ferne Ebene auf 1 ab. Mit dieser Abbildung wird der Tiefenpufferbereich vollständig ausgenutzt.

Es ist nicht notwendig, dass `nearVal` kleiner als `farVal` ist. Umkehrabbildungen wie nearVal= 1 und farVal= 0 sind akzeptabel.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`zNear` gibt die Abbildung der nahen Schnittebene auf Fensterkoordinaten an; der Anfangswert ist 0

`zFar` gibt die Abbildung der fernen Schnittebene auf Fensterkoordinaten an; der Anfangswert ist 1

FEHLER

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.DepthRange()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_DEPTH_RANGE`

6.31 gl.Disable

BEZEICHNUNG

gl.Disable – deaktiviert die serverseitigen GL-Fähigkeiten

ÜBERSICHT

```
gl.Disable(cap)
```

BESCHREIBUNG

`gl.Disable()` deaktiviert verschiedene Fähigkeiten. Verwenden Sie `gl.IsEnabled()` oder `gl.Get()`, um die aktuelle Einstellung einer beliebigen Fähigkeit zu ermitteln. Der Anfangswert mit Ausnahme von `#GL_DITHER` ist `#GL_FALSE` (bei `#GL_DITHER` ist `#GL_TRUE`).

`gl.Disable()` verwendet ein einzelnes Argument, `cap`, das einen der folgenden Werte annehmen kann:

`#GL_ALPHA_TEST`
Wenn aktiviert, wird ein Alpha-Test durchgeführt. Siehe Abschnitt 6.2 [gl.AlphaFunc], Seite 25, für Details.

`#GL_AUTO_NORMAL`
: Wenn diese Option aktiviert ist, generieren Sie Normale-Vektoren, wenn entweder `#GL_MAP2_VERTEX_3` oder `#GL_MAP2_VERTEX_4` zum Generieren von Scheiteln verwendet wird. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_BLEND`
: Wenn diese Option aktiviert ist, mischen Sie die berechneten Fragmentfarbwerte mit den Werten in den Farbpuffern. Siehe Abschnitt 6.8 [gl.BlendFunc], Seite 33, für Details.

`#GL_CLIP_PLANEi`
: Wenn diese Option aktiviert ist, schneiden Sie die Geometrie gegen die benutzerdefinierte Schnittfläche ab. Siehe Abschnitt 6.17 [gl.ClipPlane], Seite 41, für Details.

`#GL_COLOR_LOGIC_OP`
: Wenn diese Option aktiviert ist, wenden Sie die aktuell ausgewählte logische Operation auf die berechneten Fragment- und Farbpufferwerte an. Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_COLOR_MATERIAL`
: Wenn aktiviert, können ein oder mehrere Materialparameter die aktuelle Farbe benutzen. Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

`#GL_CULL_FACE`
: Wenn aktiviert, werden Polygone basierend auf ihrer Wicklung in Fensterkoordinaten ausgeschlossen. Siehe Abschnitt 6.25 [gl.CullFace], Seite 52, für Details.

`#GL_DEPTH_TEST`
: Wenn diese Option aktiviert ist, werden Tiefenteste durchgeführt und der Tiefenpuffer aktualisiert. Beachten Sie, dass selbst wenn der Tiefenpuffer vorhanden ist und die Tiefenmaske nicht Null ist, der Tiefenpuffer nicht aktualisiert wird, wenn der Tiefentest deaktiviert ist. Siehe Abschnitt 6.28 [gl.DepthFunc], Seite 54, für Details. Siehe Abschnitt 6.30 [gl.DepthRange], Seite 55, für Details.

`#GL_DITHER`
: Wenn diese Option aktiviert ist, werden Farbkomponenten oder Indizes gedithert, bevor sie in den Farbpuffer geschrieben werden.

`#GL_FOG` Wenn diese Option aktiviert ist und kein Fragment-Shader aktiv ist, wird eine Nebelfarbe in die Farbe nach dem Texturieren gemischt. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

`#GL_INDEX_LOGIC_OP`
: Wenn diese Option aktiviert ist, wird die aktuell ausgewählte logische Operation auf die Index- und Farbpufferindizes angewendet. Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_LIGHTi`
: Wenn aktiviert, wird die Lichtquelle i in die Auswertung der Beleuchtungsgleichung einbezogen. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details.

`#GL_LIGHTING`
: Wenn diese Option aktiviert ist und kein Vertex-Shader aktiv ist, werden die aktuellen Beleuchtungsparameter verwendet, um die Farbe oder den Index des Scheitelpunkts zu berechnen. Andernfalls verknüpfen Sie einfach die aktuelle Farbe oder den aktuellen Index mit jedem Scheitelpunkt. Siehe Abschnitt 6.95 [gl.Material], Seite 152, für Details. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details.

`#GL_LINE_SMOOTH`
: Wenn aktiviert, werden Linien mit korrekter Filterung gezeichnet; andernfalls aliasierte Linien. Siehe Abschnitt 6.87 [gl.LineWidth], Seite 141, für Details.

`#GL_LINE_STIPPLE`
: Wenn aktiviert, werden beim Zeichnen von Linien das aktuelle Linienmuster verwendet. Siehe Abschnitt 6.86 [gl.LineStipple], Seite 140, für Details.

`#GL_MAP1_COLOR_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` RGBA-Werte. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_INDEX`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` Farbindizes. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_NORMAL`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` Normale. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_1`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_2`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` t-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_3`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-, t- und r-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-, t-, r- und q-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_VERTEX_3`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` x-, y- und z-Scheitelpunktkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_VERTEX_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` homogene x-, y- , z- und w-Scheitelpunktkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_COLOR_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` RGBA-Werte. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_INDEX`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` Farbindizes. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_NORMAL`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` Normale. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_1`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_2`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s- und t-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_3`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-, t- und r-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` s-, t-, r- und q-Texturkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_3`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` x-, y- und z-Scheitelpunktkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_4`
: Wenn aktiviert, erzeugen `gl.EvalCoord()`, `gl.EvalMesh()` und `gl.EvalPoint()` homogene x-, y- , z- und w-Scheitelpunktkoordinaten. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_NORMALIZE`
: Wenn diese Option aktiviert ist und kein Vertex-Shader aktiv ist, werden Normale-Vektoren nach der Transformation und vor der Beleuchtung auf die Einheitslänge normalisiert. Siehe Abschnitt 6.99 [gl.Normal], Seite 157, für Details. Siehe Abschnitt 6.100 [gl.NormalPointer], Seite 158, für Details.

`#GL_POINT_SMOOTH`
: Wenn aktiviert, werden Punkte mit der richtigen Filterung gezeichnet, andernfalls Alias-Punkte. Siehe Abschnitt 6.107 [gl.PointSize], Seite 170, für Details.

`#GL_POLYGON_OFFSET_FILL`
: Wenn diese Option aktiviert ist und das Polygon im #GL_FILL-Modus gerendert wird, wird den Tiefenwerten der Fragmente eines Polygons ein Versatz hinzugefügt, bevor der Tiefentest durchgeführt wird. Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_OFFSET_LINE`
: Wenn aktiviert und falls das Polygon im #GL_LINE-Modus gerendert wird, wird den Tiefenwerten der Fragmente eines Polygons ein Versatz hinzugefügt, bevor der Tiefentest durchgeführt wird. Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_OFFSET_POINT`
: Wenn aktiviert, wird den Tiefenwerten der Fragmente eines Polygons ein Versatz hinzugefügt, bevor der Tiefentest durchgeführt wird, wenn das Polygon im GL_POINT-Modus gerendert wird. Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_SMOOTH`
: Wenn aktiviert, werden Polygone mit geeigneter Filterung gezeichnet. Andernfalls Alias-Polygone. Für korrekte antialiasierte Polygone wird ein Alpha-Puffer benötigt und die Polygone müssen von vorne nach hinten sortiert werden.

`#GL_POLYGON_STIPPLE`
: Wenn aktiviert, wird das aktuelle Polygon-Punktierungsmuster beim Rendern von Polygonen verwenden. Siehe Abschnitt 6.110 [gl.PolygonStipple], Seite 174, für Details.

`#GL_SCISSOR_TEST`
: Wenn diese Option aktiviert ist, werden Fragmente verworfen, die sich außerhalb des Scherenrechtecks befinden. Siehe Abschnitt 6.128 [gl.Scissor], Seite 197, für Details.

`#GL_STENCIL_TEST`
: Wenn diese Option aktiviert ist, wird ein Schablonentest durchführt und aktualisiert den Schablonenpuffer. Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details.

`#GL_TEXTURE_1D`
: Wenn diese Option aktiviert ist und kein Fragment-Shader aktiviert ist, wird eindimensionale Texturierung ausgeführt (es sei denn, es ist auch eine zwei- oder dreidimensionale oder Würfelkarten-Texturierung aktiviert). Siehe Abschnitt 6.139 [gl.TexImage1D], Seite 210, für Details.

`#GL_TEXTURE_2D`
: Wenn diese Option aktiviert ist und kein Fragment-Shader aktiviert ist, wird eine zweidimensionale Texturierung ausgeführt (sofern nicht auch die dreidimensionale oder Würfelkarten-Texturierung aktiviert ist). Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für Details.

`#GL_TEXTURE_GEN_Q`
: Wenn diese Option aktiviert ist und kein Scheitel-Shader aktiv ist, wird die q-Texturkoordinate mithilfe der mit `gl.TexGen()` definierten Texturgenerierungsfunktion berechnet. Andernfalls wird die aktuelle q-Texturkoordinate verwendet. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_R`
: Wenn diese Option aktiviert ist und kein Scheitel-Shader aktiv ist, wird die r-Texturkoordinate mithilfe der mit `gl.TexGen()` definierten Texturgenerierungsfunktion berechnet. Andernfalls wird die aktuelle r-Texturkoordinate verwendet. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_S`
: Wenn diese Option aktiviert ist und kein Scheitel-Shader aktiv ist, wird die s-Texturkoordinate mithilfe der mit `gl.TexGen()` definierten Texturgenerierungsfunktion berechnet. Andernfalls wird die aktuelle s-Texturkoordinate verwendet. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_T`
: Wenn diese Option aktiviert ist und kein Scheitel-Shader aktiv ist, wird die t-Texturkoordinate mithilfe der mit `gl.TexGen()` definierten Texturgenerierungsfunktion berechnet. Andernfalls wird die aktuelle t-Texturkoordinate verwendet. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`cap` gibt eine symbolische Konstante an, die eine GL-Fähigkeit anzeigt

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `cap` nicht einer der zuvor aufgeführten Werte ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.Enable()` oder `gl.Disable()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()`

`gl.Get()`

6.32 gl.DisableClientState

BEZEICHNUNG

gl.DisableClientState – deaktiviert clientseitige Funktionalitäten

ÜBERSICHT

```
gl.DisableClientState(cap)
```

BESCHREIBUNG

`gl.DisableClientState()` deaktiviert einzelne klientseitige Funktionalitäten. Standardmäßig sind alle klientseitigen Funktionalitäten deaktiviert. `gl.DisableClientState()` verwendet nur das einzelne Argument `cap`, das einen der folgenden Werte annehmen kann:

`#GL_COLOR_ARRAY`
: Wenn diese Option aktiviert ist, wird der Farb-Array zum Schreiben aktiviert und beim Rendern verwendet, wenn `gl.ArrayElement()`, `gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.21 [gl.ColorPointer], Seite 44, für Details.

`#GL_EDGE_FLAG_ARRAY`
: Wenn aktiviert, wird der Rand-Flag-Array zum Schreiben aktiviert und beim Rendern verwendet, wenn `gl.ArrayElement()`, `gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.39 [gl.EdgeFlagPointer], Seite 73, für Details.

`#GL_INDEX_ARRAY`
: Wenn aktiviert, wird der Index-Array zum Schreiben aktiviert und beim Rendern verwendet, wenn `gl.ArrayElement()`, `gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.78 [gl.IndexPointer], Seite 128, für Details.

`#GL_NORMAL_ARRAY`
: Wenn aktiviert, wird das Normalen-Array zum Schreiben aktiviert und beim Rendern verwendet, wenn `gl.ArrayElement()`, `gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.100 [gl.NormalPointer], Seite 158, für Details.

`#GL_TEXTURE_COORD_ARRAY`
: Wenn diese Option aktiviert ist, wird das Texturkoordinaten-Array zum Schreiben beim Rendern verwendet, wenn `gl.ArrayElement()`,

`gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.135 [gl.TexCoordPointer], Seite 205, für Details.

`#GL_VERTEX_ARRAY`
: Wenn aktiviert, wird das Scheitel-Array zum Schreiben aktiviert und beim Rendern verwendet, wenn `gl.ArrayElement()`, `gl.DrawArrays()` oder `gl.DrawElements()` aufgerufen wird. Siehe Abschnitt 6.147 [gl.VertexPointer], Seite 226, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`array` definiert die Option, welche zu deaktivieren ist (siehe oben für unterstützte Konstanten)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `cap` kein akzeptierter Wert ist.

`gl.DisableClientState()` ist nicht erlaubt zwischen der Ausführung von `gl.Begin()` und dem entsprechenden `gl.End()`, aber es kann ein Fehler auftreten oder auch nicht. Wenn kein Fehler generiert wird, ist das Verhalten nicht definiert.

6.33 gl.DrawArrays

BEZEICHNUNG

gl.DrawArrays – rendert Grundelement aus Felddaten

ÜBERSICHT

```
gl.DrawArrays(mode, first, count)
```

BESCHREIBUNG

`gl.DrawArrays()` gibt mehrere geometrische Grundelement mit sehr wenigen Unterprogrammaufrufen an. Anstatt eine GL-Prozedur aufzurufen, um jede einzelne Ecke, Normale, Texturkoordinate, Kantenmarkierung oder Farbe zu übergeben, können Sie separate Felder aus Scheiteln, Normalen und Farben vordefinieren und daraus eine Folge von Grundelemente mit einem einzigen Aufruf von `gl.DrawArrays()` erstellen.

Wenn `gl.DrawArrays()` aufgerufen wird, verwendet `count` sequentielle Elemente von jedem aktivierten Feld, um eine Folge von geometrischen Grundelementen zu konstruieren, die zuerst mit Elementen beginnen. `mode` gibt an, welche Art von Grundelemente konstruiert werden und wie die Feld-Elemente diese Grundelementen konstruieren. Wenn `#GL_VERTEX_ARRAY` nicht aktiviert ist, werden keine geometrischen Grundelemente generiert. `mode` kann auf die symbolischen Konstanten `#GL_POINTS`, `#GL_LINE_STRIP`, `#GL_LINE_LOOP`, `#GL_LINES`, `#GL_TRIANGLE_STRIP`, `#GL_TRIANGLE_FAN`, `#GL_TRIANGLES`, `#GL_QUAD_STRIP`, `#GL_QUADS`, oder `#GL_POLYGON` gesetzt werden.

Scheitel-Attribute, die von `gl.DrawArrays()` geändert werden, haben nach der Rückgabe von `gl.DrawArrays()` einen nicht angegebenen Wert. Wenn beispielsweise GL_COLOR_ARRAY aktiviert ist, ist der Wert der aktuellen Farbe nicht definiert, nachdem `gl.DrawArrays()` ausgeführt wurde. Attribute, die nicht geändert werden, bleiben also definiert.

`gl.DrawArrays()` ist in Display-Listen enthalten. Wenn `gl.DrawArrays()` in eine Display-Liste eingegeben wird, werden die erforderlichen Felddaten (die durch die Feldzeiger und Aktivierung bestimmt werden) ebenfalls in die Display-Liste aufgenommen. Da die Feldzeiger und -Ereignisse Klientseitig sind, wirken sich ihre Werte bei der Erstellung der Listen auf Display-Listen aus, nicht jedoch bei der Ausführung der Listen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt an, welche Art von Grundelemente gerendert werden sollen (siehe oben)

`first` gibt den Startindex in den aktivierten Feldern an

`count` gibt die Anzahl der zu rendernden Indizes an

FEHLER

`#GL_INVALID_ENUM00` wird generiert, wenn der Modus kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird generiert, wenn die Anzahl negativ ist.

`#GL_INVALID_OPERATION` wird generiert, wenn ein Pufferobjektname ungleich Null an ein aktiviertes Feld gebunden ist und der Datenspeicher des Pufferobjekts derzeit zugeordnet ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.DrawArrays()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.34 gl.DrawBuffer

BEZEICHNUNG

gl.DrawBuffer – gibt an, in welche Farbpuffer gezeichnet werden soll

ÜBERSICHT

```
gl.DrawBuffer(mode)
```

BESCHREIBUNG

Wenn Farben in den Frame-Puffer geschrieben werden, werden sie in die durch `gl.DrawBuffer()` angegebenen Farbpuffer geschrieben. Die folgenden Konstanten können im `mode` übergeben werden:

`#GL_NONE` Es wird in keine Farbpuffer geschrieben.

`#GL_FRONT_LEFT`
Es wird nur in den vorderen linken Farbpuffer geschrieben.

`#GL_FRONT_RIGHT`
Es wird nur in den vorderen rechten Farbpuffer geschrieben.

`#GL_BACK_LEFT`
Es wird nur in den hinteren linken Farbpuffer geschrieben.

`#GL_BACK_RIGHT`
Es wird nur in den hinteren rechten Farbpuffer geschrieben.

`#GL_FRONT`
: Es wird nur in die Farbpuffer vorne links und vorne rechts geschrieben. Wenn es keinen vorderen rechten Farbpuffer gibt, wird nur in den vorderen linken Farbpuffer geschrieben.

`#GL_BACK` Es wird nur in die Farbpuffer hinten links und hinten rechts geschrieben. Wenn es keinen hinteren rechten Farbpuffer gibt, wird nur in den hinteren linken Farbpuffer geschrieben.

`#GL_LEFT` Es wird nur in die Farbpuffer vorne links und hinten links geschrieben. Wenn es keinen hinteren linken Farbpuffer gibt, wird nur in den vorderen linken Farbpuffer geschrieben.

`#GL_RIGHT`
: Es wird nur in die Farbpuffer vorne rechts und hinten rechts geschrieben. Wenn es keinen hinteren rechten Farbpuffer gibt, wird nur in den vorderen rechten Farbpuffer geschrieben.

`#GL_FRONT_AND_BACK`
: Alle vorderen und hinteren Farbpuffer (vorne links, vorne rechts, hinten links, hinten rechts) werden beschrieben. Wenn es keine hinteren Farbpuffer gibt, wird nur die vorderen linken und vorderen rechten Farbpuffer geschrieben. Wenn es keine rechten Farbpuffer gibt, wird nur in die vorderen linken und hinteren linken Farbpuffer geschrieben. Wenn es keine rechten oder hinteren Farbpuffer gibt, wird nur in den vorderen linken Farbpuffer geschrieben.

`#GL_AUXi` Es wird nur in den Hilfsfarbpuffer `i` geschrieben, wobei `i` zwischen 0 und dem Wert von `#GL_AUX_BUFFERS` minus 1 liegt. Beachten Sie, dass `#GL_AUX_BUFFERS` nicht die Obergrenze ist; verwenden Sie `gl.Get()`, um die Anzahl der verfügbaren Aux-Puffer abzufragen. Es ist immer der Fall, dass `#GL_AUXi` = `#GL_AUX0 + i` ist.

Wenn mehr als ein Farbpuffer zum Zeichnen ausgewählt wird, werden Misch- oder logische Operationen berechnet sowie unabhängig für jeden Farbpuffer angewendet und können zu unterschiedlichen Ergebnissen in jedem Puffer führen.

Monoskopische Kontexte beinhalten nur linke Puffer, und stereoskopische Kontexte beinhalten sowohl linke als auch rechte Puffer. Ebenso beinhalten Einfach-gepufferte Zusammenhänge nur Vordere-Puffer und doppelt-gepufferte Zusammenhänge sowohl Vordere- als auch hintere-Puffer. Der Kontext wird bei der GL-Initialisierung ausgewählt.

Der Initialwert ist `#GL_FRONT` für Einfach-gepufferte Zusammenhänge und `#GL_BACK` für doppelt-gepufferte Zusammenhänge.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` spezifiziert bis zu vier Farbpuffer, in die gezeichnet werden soll (siehe oben).

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn der Modus kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn keiner der durch den Modus angegebenen Puffer existiert.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.DrawBuffer()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_DRAW_BUFFER`

`gl.Get()` mit dem Argument `#GL_AUX_BUFFERS`

6.35 gl.DrawElements

BEZEICHNUNG

gl.DrawElements – rendert Grundelement aus Felddaten

ÜBERSICHT

```
gl.DrawElements(mode, indicesArray)
```

BESCHREIBUNG

`gl.DrawElements()` definiert mehrere geometrische Grundelemente mit sehr wenigen Aufrufen von Unterprogrammen. Anstatt einen GL-Befehl aufzurufen, um jeden einzelnen Knoten, Normale, Texturkoordinate, Kantenmarkierung oder Farbe zu übergeben, können Sie separate Felder von Knoten, Normalen usw. vorgeben und diese verwenden, um eine Sequenz von Grundelementen mit einem einzigen Aufruf von `gl.DrawElements()` zu erstellen.

Wenn `gl.DrawElements()` aufgerufen wird, liest es sequentielle Elemente aus einem aktivierten Feld und erstellt eine Sequenz von geometrischen Grundelementen. `mode` gibt an, welche Art von Grundelemente konstruiert sind und wie die Feld-Elemente diese Grundelemente konstruieren. `mode` kann auf die symbolischen Konstanten `#GL_POINTS`, `#GL_LINE_STRIP`, `#GL_LINE_LOOP`, `#GL_LINES`, `#GL_TRIANGLE_STRIP`, `#GL_TRIANGLE_FAN`, `#GL_TRIANGLES`, `#GL_QUAD_STRIP`, `#GL_QUADS` und `#GL_POLYGON` gesetzt werden. Wenn mehr als ein Feld aktiviert ist, wird jedes verwendet. Wenn `#GL_VERTEX_ARRAY` nicht aktiviert ist, werden keine geometrischen Grundelemente konstruiert.

Vertex-Attribute, die durch `gl.DrawElements()` geändert werden, haben nach der Rückgabe von `gl.DrawElements()` einen nicht definierten Wert. Wenn beispielsweise `#GL_COLOR_ARRAY` aktiviert ist, ist der Wert der aktuellen Farbe nach der Ausführung von `gl.DrawElements()` undefiniert. Attribute, die nicht geändert werden, behalten ihre bisherigen Werte bei.

`gl.DrawElements()` ist in den Display-Listen enthalten. Wenn `gl.DrawElements()` in eine Display-Liste eingetragen wird, werden auch die notwendigen Felddaten (bestimmt durch die Feldzeiger und Aktivierungen) in die Display-Liste eingetragen. Da sich die Feldzeiger und Aktivierungen im Klient-seitigen Status befinden, wirken sich ihre Werte auf die Display-Listen beim Erstellen der Listen und nicht beim Ausführen der Listen aus.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt an, welche Art von Grundelemente gerendert werden sollen (siehe oben)

`indicesArray`
: definiert ein Feld, in dem die Indizes gespeichert sind; die Indizes in diesem Feld werden als Werte vom Typ `#GL_UNSIGNED_INT` behandelt.

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn der Modus kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn ein Name eines Pufferobjekts ungleich Null an ein aktiviertes Feld oder das Element-Feld gebunden ist und der Datenspeicher des Pufferobjekts aktuell abgebildet wird.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.DrawElements()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.36 gl.DrawPixels

BEZEICHNUNG

gl.DrawPixels – schreibt einen Block von Pixeln in den Rahmenpuffer

ÜBERSICHT

```
gl.DrawPixels(width, height, format, pixelsArray)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.DrawPixelsRaw()`, mit der Ausnahme, dass die Pixeldaten nicht als Rohspeicherpuffer übergeben werden, sondern als Tabelle, die width*height Anzahl der Elemente enthält, die jeweils ein Pixel beschreiben. Dies ist natürlich nicht so effizient wie die Verwendung von Rohspeicherpuffern, da die Pixeldaten der Tabelle zuerst in einen Rohspeicherpuffer kopiert werden müssen.

Beachten Sie, dass `gl.DrawPixels()` Daten vom Typ `#GL_FLOAT` innerhalb der Tabelle `pixelsArray` erwartet.

Siehe Abschnitt 6.37 [gl.DrawPixelsRaw], Seite 68, für weitere Details zu den von diesem Befehl akzeptierten Parametern.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`width`
: gibt die Breite des Pixelrechtecks an, das in den Rahmenpuffer geschrieben werden soll

`height`
: gibt die Höhe des Pixelrechtecks an, das in den Rahmenpuffer geschrieben werden soll

`format`
: gibt das Format der Pixeldaten an (siehe oben für unterstützte Formate)

`pixelsArray`
: definiert ein Feld, das die Pixeldaten enthält; Daten in diesem Feld werden behandelt als `#GL_FLOAT`

6.37 gl.DrawPixelsRaw

BEZEICHNUNG

gl.DrawPixelsRaw – schreibt einen Block von Pixeln in den Rahmenpuffer

ÜBERSICHT

```
gl.DrawPixelsRaw(width, height, format, type, pixels)
```

BESCHREIBUNG

`gl.DrawPixelsRaw()` liest Pixeldaten aus dem Speicher und schreibt sie in den Rahmenpuffer relativ zur aktuellen Rasterposition, vorausgesetzt, die Rasterposition ist gültig. Verwenden Sie `gl.RasterPos()` zum Einstellen der aktuellen Rasterposition; verwenden Sie `gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION_VALID`, um festzustellen, ob die angegebene Rasterposition gültig ist und `gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION` zur Abfrage der Rasterposition.

Mehrere Parameter definieren die Kodierung von Pixeldaten im Speicher und steuern die Verarbeitung der Pixeldaten, bevor sie in den Rahmenpuffer gestellt werden. Diese Parameter werden mit vier Befehlen eingestellt: `gl.PixelStore()`, `gl.PixelTransfer()`, `gl.PixelMap()` und `gl.PixelZoom()`. Diese Referenzseite beschreibt die Auswirkungen auf `gl.DrawPixelsRaw()` von vielen, aber nicht allen der durch diese vier Befehle angegebenen Parameter.

Daten werden von `pixels` als eine Folge von vorzeichenbehafteten (signiert) oder vorzeichenlosen (unsigniert) Bytes, vorzeichenbehafteten oder vorzeichenlosen Shorts, vorzeichenbehafteten oder vorzeichenlosen Ganzzahlen oder Gleitkommawerten mit einfacher Genauigkeit gelesen. Je nach `type` kann der `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT` sein. Jedes dieser Bytes, Shorts, Ganzzahlen oder Gleitkommazahlen-Werte wird je nach Format als eine Farb- oder Tiefenkomponente oder ein Index interpretiert. Indizes werden immer individuell behandelt. Farbkomponenten werden als Gruppen von einem, zwei, drei oder vier Werten behandelt, wiederum basierend auf `format`. Beide einzelne Indizes und Gruppen von Komponenten werden als Pixel bezeichnet. Wenn der Typ `#GL_BITMAP` ist, müssen die Daten vorzeichenlos sein. Bytes und das Format muss entweder `#GL_COLOR_INDEX` oder `#GL_STENCIL_INDEX` sein. Jedes vorzeichenlose Byte wird als acht 1-Bit-Pixel behandelt, mit Bitreihenfolge bestimmt durch `#GL_UNPACK_LSB_FIRST`. (Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.)

`width` * `height` Pixel werden aus dem Speicher gelesen, beginnend mit der Position `pixels`. Standardmäßig werden diese Pixel von benachbarten Speicherplätzen entnommen, mit der Ausnahme, dass nach dem Lesen aller Pixelbreite der Lesezeiger auf die nächste 4-Byte-Grenze bewegt wird. Die Ausrichtung der Vier-Byte-Zeile wird durch `gl.PixelStore()` mit dem Argument `#GL_UNPACK_ALIGNMENT` angegeben und es kann auf ein, zwei, vier oder acht Bytes gesetzt werden. Andere Pixelspeicherparameter geben unterschiedliche Lesezeigerfortschritte an, sowohl vor dem Lesen des ersten Pixels als auch nach dem Lesen aller Pixelbreite. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

Die aus dem Speicher gelesenen Pixel der `width` * `height` werden jeweils auf die gleiche Weise bearbeitet, basierend auf den Werten mehrerer Parameter, die durch `gl.PixelTransfer()` und `gl.PixelMap()` definiert sind. Die Details dieser Operationen sowie der Zielpuffer, in den die Pixel gezeichnet werden, sind spezifisch für das Format

der Pixel, wie durch `format` festgelegt. `format` kann einen von 13 symbolischen Werten annehmen:

`#GL_COLOR_INDEX`

Jeder Pixel ist ein Einzelwert, ein Farbindex. Es wird in das Festkommaformat konvertiert, mit einer unbestimmten Anzahl von Bits rechts vom Binärpunkt, unabhängig vom Speicherdatentyp. Gleitkommawerte werden in echte Festkommawerte umgewandelt.

Jeder Festkommaindex wird dann um #GL_INDEX_SHIFT-Bits nach links verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_INDEX_SHIFT` negativ ist, ist die Verschiebung nach rechts. In beiden Fällen füllen Nullbits ansonsten nicht spezifizierte Bitpositionen im Ergebnis.

Befindet sich GL im RGBA-Modus, wird der resultierende Index konvertiert zu einem RGBA Pixel mit Hilfe der `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B` und `#GL_PIXEL_MAP_I_TO_A` Tabellen. Wenn sich GL im Farbindexmodus befindet und `#GL_MAP_COLOR True` ist, wird der Index durch den Wert ersetzt, auf den er in der Lookup-Tabelle `#GL_PIXEL_MAP_I_I_TO_I` verweist. Unabhängig davon, ob die Ersetzung des Index durchgeführt wird oder nicht, wird der ganzzahlige Teil des Index dann mit 2^b-1 hinzugefügt, wobei `b` die Anzahl der Bits in einem Farbindexpuffer ist.

GL konvertiert dann die resultierenden Indizes oder RGBA-Farben in Fragmente, indem die aktuellen Rasterpositions-Z-Koordinaten und Texturkoordinaten an jedes Pixel anfügt und dann dem n-ten Fragment x- und y-Fensterkoordinaten zuordnet, so dass

```
xn = xr + n % width
yn = yr + n / width
```

ist, wobei (xr,yr) die aktuelle Rasterposition angibt. Diese Pixelfragmente werden dann genauso behandelt wie die Fragmente, die durch das Rastern von Punkten, Linien oder Polygonen erzeugt werden. Texturabbildung, Nebel und alle Fragmentoperationen werden angewendet, bevor die Fragmente in den Rahmenpuffer geschrieben werden.

`#GL_STENCIL_INDEX`

Jedes Pixel ist ein Einzelwert, ein Schablonenindex. Es wird in das Festkommaformat konvertiert, mit einer unbestimmten Anzahl von Bits rechts vom Binärpunkt, unabhängig vom Speicherdatentyp. Gleitkommawerte werden in echte Festkommawerte umgewandelt.

Jeder Festkommaindex wird dann um #GL_INDEX_SHIFT-Bits nach links verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_INDEX_SHIFT` negativ ist, wird die Verschiebung nach rechts vorgenommen. In beiden Fällen füllen Nullbits ansonsten nicht die definierte Bitpositionen das Ergebnis auf. Wenn `#GL_MAP_STENCIL True` ist, wird der Index durch den Wert ersetzt, auf den er in der Lookup-Tabelle `#GL_PIXEL_MAP_S_TO_S` verweist. Unabhängig davon, ob die Ersetzung des Index durchgeführt wird oder nicht, wird der ganzzahlige Teil des Index dann mit 2^b-1 hinzugefügt, wobei `b` die

Anzahl der Bits im Schablonenpuffer ist. Die resultierenden Schablonenindizes werden dann in die Schablonenpuffer geschrieben, so dass der n-te Index in die Position geschrieben wird.

```
xn = xr + n % width
yn = yr + n / width
```

wobei (xr,yr) die aktuelle Rasterposition ist. Nur der Pixelbesitztest, der Scherentest und die Schablonen-Schreibmaske beeinflussen diese Schreibvorgänge.

`#GL_DEPTH_COMPONENT`
: Jedes Pixel ist eine Komponente mit einer einzigen Tiefe. Gleitkommadaten werden direkt und mit unbestimmter Genauigkeit in ein internes Gleitkommaformat umgewandelt. Der resultierende Gleitkomma-Tiefenwert wird dann mit `#GL_DEPTH_SCALE` multipliziert und zu `#GL_DEPTH_BIAS` hinzugefügt. Das Ergebnis wird auf den Bereich [0,1] festgelegt.

  GL konvertiert dann die resultierenden Tiefenkomponenten in Fragmente, indem er die aktuelle Rasterpositionsfarbe oder den Farbindex und die Texturkoordinaten an jedes Pixel anfügt und dann dem n-ten Fragment x- und y-Fensterkoordinaten zuordnet, so dass

  ```
  xn = xr + n % width
  yn = yr + n / width
  ```

  ist, wobei (xr,yr) die aktuelle Rasterposition angibt. Diese Pixelfragmente werden dann genauso behandelt wie die Fragmente, die durch das Rastern von Punkten, Linien oder Polygonen erzeugt werden. Texturabbildung, Nebel und alle Fragmentoperationen werden angewendet, bevor die Fragmente in den Rahmenpuffer geschrieben werden.

`#GL_RGBA`
: Jedes Pixel ist eine Vier-Komponenten-Gruppe: Bei `#GL_RGBA` ist die rote Komponente zuerst, gefolgt von grün, dann von blau und am Schluss von alpha. Gleitkommawerte werden mit unbestimmter Genauigkeit direkt in ein internes Gleitkommaformat umgewandelt. Die resultierenden Fließkomma-Farbwerte werden dann mit `#GL_c_SCALE` multipliziert und zu `#GL_c_BIAS` addiert, wobei c ROT, GRÜN, BLAU oder ALPHA für die jeweiligen Farbkomponenten ist. Die Ergebnisse werden in den Bereich [0,1] festgelegt.

  Wenn `#GL_MAP_COLOR True` ist, wird jede Farbkomponente um die Größe der Ersetzungstabelle `#GL_PIXEL_MAP_c_TO_c` skaliert und dann durch den Wert ersetzt, auf den sie in dieser Tabelle verweist. c ist R, G, B bzw. A.

  GL konvertiert dann die resultierenden RGBA-Farben in Fragmente, indem die aktuellen Koordinaten der Rasterposition z und Texturkoordinaten an jedes Pixel anhängt und dann dem n-ten Fragment x- und y-Fensterkoordinaten zuordnet, so dass

  ```
  xn = xr + n % width
  yn = yr + n / width
  ```

  ist, wobei (xr,yr) die aktuelle Rasterposition angibt. Diese Pixelfragmente werden dann genauso behandelt wie die Fragmente, die durch das Rastern von Punkten, Linien oder Polygonen erzeugt werden. Texturabbildung, Ne-

bel und alle Fragmentoperationen werden angewendet, bevor die Fragmente in den Rahmenpuffer geschrieben werden.

`#GL_RED` Jedes Pixel ist eine einzelne rote Komponente. Diese Komponente wird in das interne Gleitkommaformat konvertiert, so wie es die Rotkomponente eines RGBA-Pixels ist. Es wird dann in ein RGBA-Pixel umgewandelt, wobei Grün und Blau auf 0 und Alpha auf 1 gesetzt werden. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_GREEN`
Jedes Pixel ist eine einzelne grüne Komponente. Diese Komponente wird in das interne Gleitkommaformat umgewandelt, wie die Grüne Komponente eines RGBA-Pixels. Es wird dann in ein RGBA-Pixel umgewandelt, wobei Rot und Blau auf 0 und Alpha auf 1 gesetzt werden. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_BLUE` Jedes Pixel ist eine einzelne blaue Komponente. Diese Komponente wird in das interne Gleitkommaformat umgewandelt, so wie die Blaue Komponente eines RGBA-Pixels. Es wird dann in ein RGBA-Pixel umgewandelt, wobei Rot und Grün auf 0 und Alpha auf 1 gesetzt werden. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_ALPHA`
Jedes Pixel ist eine einzelne Alpha-Komponente. Diese Komponente wird in das interne Gleitkommaformat konvertiert, wie es auch die Alpha Komponente eines RGBA-Pixels ist. Es wird dann in ein RGBA-Pixel umgewandelt, wobei Rot, Grün und Blau auf 0 gesetzt werden, und das Pixel wird nach dieser Konvertierung so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_RGB` Jedes Pixel ist eine Drei-Komponenten-Gruppe: zuerst rot, dann grün, dann blau. Jede Komponente wird in das interne Gleitkommaformat umgewandelt, wie die roten, grünen und blauen Komponenten eines RGBA-Pixels. Das Farbtripel wird in ein RGBA-Pixel umgewandelt, wobei Alpha auf 1 gesetzt ist. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_LUMINANCE`
Jedes Pixel ist eine einzelne Luminanzkomponente. Diese Komponente wird in das interne Gleitkommaformat konvertiert, so wie es die Rotkomponente eines RGBA-Pixels ist. Es wird dann in ein RGBA-Pixel umgewandelt, wobei Rot, Grün und Blau auf den konvertierten Helligkeitswert und Alpha auf 1 gesetzt werden. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

`#GL_LUMINANCE_ALPHA`
Jedes Pixel ist eine Zwei-Komponenten-Gruppe: Zuerst die Leuchtdichte, dann Alpha. Die beiden Komponenten werden in das interne Gleitkomma-

format umgewandelt, so wie der Rotanteil eines RGBA-Pixels. Sie werden dann in ein RGBA-Pixel umgewandelt, wobei Rot, Grün und Blau auf den konvertierten Luminanzwert und Alpha auf den konvertierten Alpha-Wert eingestellt werden. Nach dieser Konvertierung wird das Pixel so behandelt, als ob es als RGBA-Pixel gelesen worden wäre.

Die bisher beschriebene Rasterung geht von einem Pixelzoomfaktoren von 1 aus, wenn `gl.PixelZoom()` verwendet wird, um die Zoomfaktoren x und y zu ändern, werden Pixel wie folgt in Fragmente umgewandelt. Wenn (xr,yr) die aktuelle Rasterposition ist und sich ein bestimmtes Pixel in der n-ten Spalte und m-ten Zeile des Pixelrechtecks befindet, dann werden Fragmente für Pixel erzeugt, deren Mittelpunkt im Rechteck mit Ecken bei

```
(xr + zoomx_n, yr + zoomy_m)
```

und

```
(xr + zoomx_(n + 1), yr + zoomy_(m + 1))
```

ist, wobei `zoomx` der Wert von `#GL_ZOOM_X` und `zoomy` der Wert von `#GL_ZOOM_Y` angibt.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 6.36 [gl.DrawPixels], Seite 67, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`width` gibt die Breite des Pixelrechtecks an, das in den Rahmenpuffer geschrieben werden soll

`height` gibt die Höhe des Pixelrechtecks an, das in den Rahmenpuffer geschrieben werden soll

`format` gibt das Format der Pixeldaten an (siehe oben für unterstützte Formate)

`type` gibt den Datentyp der Pixeldaten an (siehe oben)

`pixels` definiert einen Zeiger auf die Pixeldaten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` oder `type` nicht zu den akzeptierten Werten gehört.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type #GL_BITMAP` ist und `format` nicht `#GL_COLOR_INDEX` oder `#GL_STENCIL_INDEX` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `width` oder `height` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format #GL_STENCIL_INDEX` ist und es keinen Schablonenpuffer gibt.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format #GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE`, oder `#GL_LUMINANCE_ALPHA` ist und GL sich im Farbindexmodus befindet.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.DrawPixelsRaw()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION_VALID`

6.38 gl.EdgeFlag

BEZEICHNUNG

gl.EdgeFlag – kennzeichnet Kanten entweder als Rand- oder Innenkante

ÜBERSICHT

```
gl.EdgeFlag(flag)
```

BESCHREIBUNG

Jeder Scheitelpunkt eines Polygons, eines separaten Dreiecks oder eines separaten Vierecks, der zwischen einem `gl.Begin()` und `gl.End()` angegeben ist, wird als Anfang einer Randkante oder einer Innenkante markiert. Wenn das aktuelle Kantenflag `True` ist und wenn der Knoten angegeben wird, wird dieser Knoten als Anfang einer Randkante markiert. Andernfalls wird der Knoten als Anfang einer Innenkante markiert. `gl.EdgeFlag()` setzt das Kantenflagbit auf `#GL_TRUE`, wenn das Flag `#GL_TRUE` ist und andernfalls auf `#GL_FALSE`. Der Initialwert ist `#GL_TRUE`.

Die Eckpunkte von verbundenen Dreiecken und verbundenen Vierecken werden immer als Randkante markiert, unabhängig vom Wert des Kantenflags.

Rand- und Innenkanteflags an Eckpunkten sind nur dann signifikant, wenn `#GL_POLYGON_MODE` auf `#GL_POINT` oder `#GL_LINE` gesetzt sind. Siehe Abschnitt 6.108 [gl.PolygonMode], Seite 172, für Details.

Das aktuelle Kantenflag kann jederzeit aktualisiert werden. Insbesondere kann `gl.EdgeFlag()` zwischen einem Aufruf von `gl.Begin()` und `gl.End()` aufgerufen werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`flag`	gibt den aktuellen Wert des Kantenflags an (entweder `#GL_TRUE` oder `#GL_FALSE`)

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_EDGE_FLAG`

6.39 gl.EdgeFlagPointer

BEZEICHNUNG

gl.EdgeFlagPointer – definiert ein Feld von Kanten-Flags

ÜBERSICHT

```
gl.EdgeFlagPointer(flagsArray)
```

BESCHREIBUNG

`gl.EdgeFlagPointer()` gibt ein Feld von Booleschen Kantenflags an, die beim Rendern verwendet werden sollen. Wenn Sie `Nil` in `flagsArray` übergeben, wird der Kanten-Flag-Feld-Puffer freigegeben, aber er wird nicht aus OpenGL entfernt. Dies muss manuell erfolgen, z.B. durch Deaktivieren des Kanten-Flag-Felds oder durch Definieren eines neuen.

Wenn ein Kanten-Flag-Feld angegeben wird, wird es zusätzlich zur aktuellen Scheitelpunkt-Feld-Pufferobjektbindung als klientseitiger Status gespeichert.

Um das Kanten-Flag-Feld zu aktivieren und zu deaktivieren, rufen Sie `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_EDGE_FLAG_ARRAY` auf. Wenn aktiviert, wird das Kanten-Flag-Feld verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Kantenflags werden für verschachtelte Scheitelpunkt-Feld-Formate nicht unterstützt. Siehe Abschnitt 6.80 [gl.InterleavedArrays], Seite 130, für Details.

Das Kanten-Flag-Feld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()`, oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.EdgeFlagPointer()` ist zwischen der Ausführung von `gl.Begin()` und `gl.End()` nicht erlaubt, aber es kann ein Fehler auftreten oder auch nicht. Wenn kein Fehler erzeugt wird, ist der Vorgang undefiniert.

`gl.EdgeFlagPointer()` wird typischerweise auf der Klient-Seite implementiert.

Kanten-Flag-Feld-Parameter sind klientseitige Zustände und werden daher nicht von `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt. Benutzen Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`flagsArray`
: definiert eine Tabelle, die ein Feld von Kantenflags oder `Nil` enthält

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_EDGE_FLAG_ARRAY`

`gl.Get()` mit dem Argument `#GL_EDGE_FLAG_ARRAY_POINTER`

6.40 gl.Enable

BEZEICHNUNG

gl.Enable – aktiviert serverseitige GL-Fähigkeiten

ÜBERSICHT

`gl.Enable(cap)`

BESCHREIBUNG

`gl.Enable()` ermöglicht verschiedene Fähigkeiten. Verwenden Sie `gl.IsEnabled()` oder `gl.Get()`, um die aktuelle Einstellung einer beliebigen Fähigkeit zu bestimmen. Der

Anfangswert für jede Fähigkeit mit Ausnahme von `#GL_DITHER` ist `#GL_FALSE`. Der Initialwert für `#GL_DITHER` ist `#GL_TRUE`.

Siehe Abschnitt 6.31 [gl.Disable], Seite 56, für eine Liste der unterstützten Funktionen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`cap` definiert eine symbolische Konstante, die eine GL-Fähigkeit anzeigt

6.41 gl.EnableClientState

BEZEICHNUNG

gl.EnableClientState – aktiviert klientseitige Fähigkeiten

ÜBERSICHT

```
gl.EnableClientState(cap)
```

BESCHREIBUNG

`gl.EnableClientState()` ermöglicht individuelle klientseitige Fähigkeiten. Standardmäßig sind alle klientseitigen Fähigkeiten deaktiviert. `gl.EnableClientState()` hat ein einziges Argument: `cap`. Siehe Abschnitt 6.32 [gl.DisableClientState], Seite 62, für eine Liste der unterstützten Fähigkeiten.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`cap` gibt die Fähigkeit an, die aktiviert werden soll

6.42 gl.End

BEZEICHNUNG

gl.End – begrenzt die Scheitelpunkte eines oder einer Gruppe ähnlicher Grafikgrundelemente

ÜBERSICHT

```
gl.End()
```

BESCHREIBUNG

Siehe Abschnitt 6.5 [gl.Begin], Seite 28, für Details.

EINGABEN

keine

6.43 gl.EndList

BEZEICHNUNG

gl.EndList – ersetzt eine Display-Liste

ÜBERSICHT

```
gl.EndList()
```

BESCHREIBUNG

Siehe Abschnitt 6.98 [gl.NewList], Seite 155, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.EndList()` ohne vorhergehende `gl.NewList()` aufgerufen wird.

6.44 gl.EvalCoord

BEZEICHNUNG

gl.EvalCoord – wertet aktivierte ein- und zweidimensionalen Karten aus

ÜBERSICHT

```
gl.EvalCoord(u[, v])
```

BESCHREIBUNG

`gl.EvalCoord()` wertet aktivierte ein- oder zweidimensionale Karten bei den Argument `u` oder `u` und `v` aus. Um eine Karte zu definieren, rufen Sie `gl.Map()` auf; um sie zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` auf.

Wenn der Befehl `gl.EvalCoord()` ausgegeben wird, werden alle aktuell aktivierten Karten der angegebenen Dimension ausgewertet. Dann ist es für jede aktivierte Karte so, als ob der entsprechende GL-Befehl mit dem berechneten Wert ausgegeben worden wäre. Das heißt, wenn `#GL_MAP1_INDEX` oder `#GL_MAP2_INDEX` aktiviert ist, wird ein `gl.Index()` simuliert. Wenn `#GL_MAP1_COLOR_4` oder `#GL_MAP2_COLOR_4` aktiviert ist, wird ein `gl.Color()` simuliert. Wenn `#GL_MAP1_NORMAL` oder `#GL_MAP2_NORMAL` aktiviert ist, wird ein normaler Vektor erzeugt, und wenn eine der Konstanten `#GL_MAP1_TEXTURE_COORD_1`, `#GL_MAP1_TEXTURE_COORD_2`, `#GL_MAP1_TEXTURE_COORD_3`, `#GL_MAP1_TEXTURE_COORD_4` oder auch die Konstanten `#GL_MAP2_TEXTURE_COORD_1`, `#GL_MAP2_TEXTURE_COORD_2`, `#GL_MAP2_TEXTURE_COORD_3` oder `#GL_MAP2_TEXTURE_COORD_4` aktiviert ist, dann simuliert GL ein entsprechender Befehl `gl.TexCoord()`.

Die Farb-, Farbindex-, Normale- und Texturkoordinatenwerte werden jedes mal errechnet, wenn die Berechunung für diese aktiviert wurde (ansonsten werden die aktuellen Werte benutzt). Diese berechneten Werte überschreiben allerdings nicht die aktuellen Farb-, Farbindex-, Normale- oder Texturkoordiantenwerte. Also, wenn `gl.Vertex()`-Befehle mit gl.EvalCoord()-Befehlen durchsetzt sind, wobei die Farb-, Normal- und Texturkoordinatenwerte den `gl.Vertex()`-Befehle zugeordnet sind, nicht von den Werten der gl.EvalCoord()-Befehle beeinflusst, sondern nur von den letzten `gl.Color()`, `gl.Index()`, `gl.Normal()` und `gl.TexCoord()` Befehlen.

Für Karten, die nicht aktiviert sind, werden keine Befehle ausgegeben. Wenn für eine bestimmte Dimension mehr als eine Texturauswertung aktiviert ist (z.B. `#GL_MAP2_TEXTURE_COORD_1` und `#GL_MAP2_TEXTURE_COORD_2`), dann wird nur die Auswertung der Karte durchgeführt, die die größere Anzahl von Koordinaten ergibt (in diesem Fall

#GL_MAP2_TEXTURE_COORD_2). #GL_MAP1_VERTEX_4 überschreibt #GL_MAP1_VERTEX_3 und #GL_MAP2_VERTEX_4 überschreibt #GL_MAP2_VERTEX_3 auf die gleiche Weise. Wenn für die angegebene Dimension weder eine Drei- noch eine Vierkomponenten-Scheitelkarte aktiviert ist, wird die Option beim gl.EvalCoord()-Befehl ignoriert.

Wenn Sie die automatische Normalen-Generierung aktiviert haben, indem Sie gl.Enable() mit dem Argument #GL_AUTO_NORMAL aufrufen, erzeugt gl.EvalCoord() Oberflächennormale analytisch, unabhängig vom Inhalt oder der Aktivierung der #GL_MAP2_NORMAL-Karte. Wenn die automatische Normalen-Generierung deaktiviert ist, wird die entsprechende Normalen-Karte #GL_MAP2_NORMAL, falls aktiviert, verwendet, um eine Normale zu erzeugen. Wenn weder die automatische Normalen-Generierung noch eine Normalen-Karte aktiviert ist, wird für den gl.EvalCoord()-Befehl keine Normalen generiert.

Alternativ können Sie auch eine Tabelle mit einer oder zwei Domänenkoordinaten an gl.EvalCoord() übergeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

u	gibt einen Wert an, der die Domänekoordinate u zur Basisfunktion ist, die in einem vorherigen Befehl gl.Map() definiert wurde
v	optional: Gibt einen Wert an, der die Domänekoordinate v zur Basisfunktion ist, die in einem vorherigen Befehl gl.Map() definiert wurde

VERBUNDENE GET-OPERATIONEN

gl.IsEnabled() mit dem Argument #GL_MAP1_VERTEX_3
gl.IsEnabled() mit dem Argument #GL_MAP1_VERTEX_4
gl.IsEnabled() mit dem Argument #GL_MAP1_INDEX
gl.IsEnabled() mit dem Argument #GL_MAP1_COLOR_4
gl.IsEnabled() mit dem Argument #GL_MAP1_NORMAL
gl.IsEnabled() mit dem Argument #GL_MAP1_TEXTURE_COORD_1
gl.IsEnabled() mit dem Argument #GL_MAP1_TEXTURE_COORD_2
gl.IsEnabled() mit dem Argument #GL_MAP1_TEXTURE_COORD_3
gl.IsEnabled() mit dem Argument #GL_MAP1_TEXTURE_COORD_4
gl.IsEnabled() mit dem Argument #GL_MAP2_VERTEX_3
gl.IsEnabled() mit dem Argument #GL_MAP2_VERTEX_4
gl.IsEnabled() mit dem Argument #GL_MAP2_INDEX
gl.IsEnabled() mit dem Argument #GL_MAP2_COLOR_4
gl.IsEnabled() mit dem Argument #GL_MAP2_NORMAL
gl.IsEnabled() mit dem Argument #GL_MAP2_TEXTURE_COORD_1
gl.IsEnabled() mit dem Argument #GL_MAP2_TEXTURE_COORD_2
gl.IsEnabled() mit dem Argument #GL_MAP2_TEXTURE_COORD_3
gl.IsEnabled() mit dem Argument #GL_MAP2_TEXTURE_COORD_4
gl.IsEnabled() mit dem Argument #GL_AUTO_NORMAL
gl.GetMap()

6.45 gl.EvalMesh

BEZEICHNUNG

gl.EvalMesh – berechnet ein- oder zweidimensionale Gitter von Punkten oder Linien

ÜBERSICHT

```
gl.EvalMesh(mode, i1, i2[, j1, j2])
```

BESCHREIBUNG

Mit diesem Befehl kann ein ein- oder zweidimensionales Gitter von Punkten oder Linien berechnet werden. Wenn Sie die letzten beiden Parameter weglassen, wird ein eindimensionales Gitter, andernfalls ein zweidimensionales berechnet.

`gl.MapGrid()` und `gl.EvalMesh()` werden parallel verwendet, um eine Reihe von gleichmäßig verteilten Kartendomänenwerten effizient zu erzeugen und auszuwerten. `gl.EvalMesh()` durchläuft die ganzzahligen Domänen eines ein- oder zweidimensionalen Gittters, dessen Bereich der Domäne der durch `gl.Map()` angegebene Karten ist. `mode` bestimmt, ob die resultierenden Knoten als Punkte, Linien oder gefüllte Polygone verbunden werden (letzteres wird nur für zweidimensionale Gitter unterstützt). Im eindimensionalen Fall wird `gl.EvalMesh()` das Gitter so erzeugen, als ob der folgende Fragment-Code ausgeführt wurde:

```
gl.Begin(type)
For Local i = i1 To i2 Do gl.EvalCoord(i*du+u1)
gl.End()
```

wobei du = (u2-u1)/n und n, u1 und u2 die Argumente für den letzten `gl.MapGrid()` Befehl sind. `type` ist `#GL_POINTS`, wenn der Mode `#GL_POINT` ist oder `#GL_LINES`, wenn der Mode `#GL_LINE` ist. Die einzige absolute numerische Anforderung ist, dass wenn i = n, der aus i*du+u1 berechnete Wert genau u2 ist.

Im zweidimensionalen Fall, `gl.EvalMesh()`, führen Sie

```
du = (u2-u1)/n
dv = (v2-v1)/m,
```

aus, wobei n, u1, u2, m, v1 und v2 die Argumente für den letzten `gl.MapGrid()` Befehl sind. Wenn der Mode dann `#GL_FILL` ist, ist der Befehl `gl.EvalMesh()` äquivalent zu:

```
For Local j = j1 To j2 - 1
   gl.Begin(#GL_QUAD_STRIP)
   For Local i = i1 To i2
      gl.EvalCoord(i*du+u1, j*dv+v1)
      gl.EvalCoord(i*du+u1, (j+1)*dv+v1)
   Next
   gl.End()
Next
```

Wenn der Mode `#GL_LINE` ist, dann ist ein Aufruf von `gl.EvalMesh()` äquivalent zu:

```
For Local j = j1 To j2
   gl.Begin(#GL_LINE_STRIP)
   For Local i = i1 To i2
      gl.EvalCoord(i*du+u1, j*dv+v1)
   Next
```

```
      gl.End()
   Next

   For Local i = i1 To i2
      gl.Begin(#GL_LINE_STRIP)
      For Local j = j1 To j2
         gl.EvalCoord(i*du+u1, j*dv+v1)
      Next
      gl.End()
   Next
```

Und schließlich, wenn der Mode `#GL_POINT` ist, dann ist ein Aufruf von `gl.EvalMesh()` äquivalent zu:

```
gl.Begin(#GL_POINTS)
For Local j = j1 To j2
   For Local i = i1 To i2
      gl.EvalCoord(i*du+u1, j*dv+v1)
   Next
Next
gl.End()
```

In allen drei Fällen sind die einzigen absoluten numerischen Anforderungen, dass wenn i = n, dann der aus i*du+u1 berechnete Wert genau u2 ist, und wenn j = m, dann ist der aus j*dv+v1 berechnete Wert genau v2.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode`	gibt an, ob ein Gitter aus Punkten, Linien oder Polygonen berechnet werden soll; symbolische Konstanten `#GL_POINT`, `#GL_LINE` und bei einem zweidimensionalen Gitter wird `#GL_FILL` akzeptiert.
`i1`	gibt den ersten ganzzahligen Wert für die Gitter-Domänenvariable i an
`i2`	gibt den letzten ganzzahligen Wert für die Gitter-Domänenvariable i an
`j1`	optional: Gibt den ersten ganzzahligen Wert für die Gitter-Domänenvariable j an
`j2`	optional: Gibt den letzten ganzzahligen Wert für die Gitter-Domänenvariable j an

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.EvalMesh()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_SEGMENTS`

6.46 gl.EvalPoint

BEZEICHNUNG

gl.EvalPoint – erzeugt und wertet einen einzelnen Punktes in einem Gitter aus

ÜBERSICHT

```
gl.EvalPoint(i[, j])
```

BESCHREIBUNG

`gl.MapGrid()` und `gl.EvalMesh()` werden parallel verwendet, um eine Reihe von gleichmäßig verteilten Kartendomänenwerten effizient zu erzeugen und auszuwerten. `gl.EvalPoint()` kann verwendet werden, um einen einzelnen Gitterpunkt im gleichen Rasterbereich auszuwerten, der von `gl.EvalMesh()` durchquert wird. Der Aufruf von `gl.EvalPoint()` mit einem einzigen Argument ist gleichbedeutend mit dem Aufruf von

```
gl.EvalCoord(i*du+u1)
```

wobei

```
du = (u2-u1)/n
```

und n, u1 und u2 die Argumente für den letzten `gl.MapGrid()` Befehl sind. Die einzige absolute numerische Anforderung ist, dass wenn i = n, der aus i*du+u1 berechnete Wert genau u2 ist.

Im zweidimensionalen Fall, lassen `gl.EvalPoint()`,

```
du = (u2-u1)/n
dv = (v2-v1)/m
```

zu, wobei n, u1, u2, m, v1 und v2 die Argumente für den letzten `gl.MapGrid()`-Befehl sind. Dann ist der Befehl `gl.EvalPoint()` äquivalent zum Aufruf von

```
gl.EvalCoord(i*du+u1, j*dv+v1)
```

Die einzigen absoluten numerischen Anforderungen sind, dass wenn i = n ist, dann der aus i*du+u1 berechnete Wert genau u2 und wenn dann j = m ist, ist der aus j*dv+v1 berechnete Wert genau v2.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`i`	gibt den ganzzahligen Wert für die Gitterdomänenvariable i an
`j`	optional: gibt den ganzzahligen Wert für die Gitterdomänenvariable j an

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_SEGMENTS`

6.47 gl.FeedbackBuffer

BEZEICHNUNG

gl.FeedbackBuffer – steuert den Feedback-Modus

ÜBERSICHT

```
buffer = gl.FeedbackBuffer(size, type)
```

BESCHREIBUNG

Der Befehl `gl.FeedbackBuffer()` steuert die Rückmeldung. Feedback ist wie die Auswahl ein GL-Modus. Der Modus wird durch Aufruf von `gl.RenderMode()` mit `#GL_FEEDBACK` ausgewählt. Wenn sich GL im Feedback-Modus befindet, werden durch die Rasterung keine Pixel erzeugt. Stattdessen werden Informationen über Grundelemente, die gerastert worden wären, mit Hilfe des GL an die Anwendung zurückgegeben.

`gl.FeedbackBuffer()` hat zwei Argumente: `size` gibt die Größe des Feldes an, das in Elementen von `#GL_FLOAT` zurückgegeben werden soll. `type` ist eine symbolische Konstante, die die Informationen beschreibt, die für jeden Knoten zurückgegeben werden. `gl.FeedbackBuffer()` muss ausgegeben werden, bevor der Feedback-Modus aktiviert wird (durch Aufruf von `gl.RenderMode()` mit dem Argument `#GL_FEEDBACK`). Das Setzen von `#GL_FEEDBACK` ohne Einrichtung des Feedback-Puffers oder der Aufruf von `gl.FeedbackBuffer()`, während sich GL im Feedback-Modus befindet, ist ein Fehler.

Wenn `gl.RenderMode()` im Feedback-Modus aufgerufen wird, gibt er die Anzahl der Einträge im Feedback-Feld zurück und setzt den Feedback-Feldzeiger auf die Basis des Feedback-Puffers zurück. Der zurückgegebene Wert überschreitet nie die Größe. Wenn die Feedback-Daten mehr Platz benötigten, als im Puffer verfügbar war, gibt `gl.RenderMode()` einen negativen Wert zurück. Um GL aus dem Feedback-Modus zu nehmen, rufen Sie `gl.RenderMode()` mit einem Parameter-Wert anders als `#GL_FEEDBACK` auf.

Im Feedback-Modus erzeugt jedes Grundelement, Bitmap oder Pixelrechteck, das gerastert werden soll, einen Block von Werten, die in das Feedback-Feld kopiert werden. Wenn dies dazu führen würde, dass die Anzahl der Einträge das Maximum übersteigt, wird der Block teilweise so geschrieben, um das Feld zu füllen (wenn überhaupt noch Platz vorhanden ist) und ein Überlauf-Flag gesetzt. Jeder Block beginnt mit einem Code, der den Grundelement-Typ angibt, gefolgt von Werten, die die Knoten des Grundelements und die zugehörigen Daten beschreiben. Einträge werden auch für Bitmaps und Pixelrechtecke geschrieben. Die Rückmeldung erfolgt nach dem Polygonauslesen und `gl.PolygonMode()`. Die Interpretation von Polygonen hat stattgefunden, so dass Polygone, die gerendert werden, nicht im Feedback-Puffer zurückgegeben werden. Sie kann auch auftreten, wenn Polygone mit mehr als drei Kanten in Dreiecke zerlegt werden, wenn die GL-Implementierung Polygone rendert, indem man diese Zerlegung durchführt.

Der Befehl `gl.PassThrough()` kann verwendet werden, um eine Markierung in den Feedback-Puffer einzufügen. Siehe Abschnitt 6.102 [gl.PassThrough], Seite 160, für Details.

Es folgt die Grammatik für die Blöcke von Werten, die in den Feedback-Puffer geschrieben werden. Jedes Grundelement wird mit einem eindeutigen Identifizierungswert angezeigt, gefolgt von einer bestimmten Anzahl von Knoten. Polygon-Einträge enthalten einen ganzzahligen Wert, der angibt, wie viele Knoten folgen. Ein Knoten wird je nach Typ als eine bestimmte Anzahl von Gleitkommawerten zurückgegeben. Farben werden als vier Werte im RGBA-Modus und ein Wert im Farbindex-Modus zurückgegeben.

Die Koordinaten der Feedback-Eckpunkte sind als Fensterkoordinaten angegeben, außer w, das als Clip-Koordinaten zurückgegeben wird. Die Feedback-Farben leuchten, wenn

die Beleuchtung aktiviert ist. Feedback-Texturkoordinaten werden erzeugt, wenn die Generierung von Texturkoordinaten aktiviert ist. Sie werden immer von der Texturmatrix transformiert.

`gl.FeedbackBuffer()` wird bei Verwendung in einer Display-Liste nicht in die Display-Liste kompiliert, sondern sofort ausgeführt.

Bitte beachten Sie, dass `gl.FeedbackBuffer()` nur die Texturkoordinate der Textureinheit `#GL_TEXTURE0` zurückgibt.

Um einen von diesem Befehl zugewiesenen Puffer freizugeben, rufen Sie den Befehl `gl.FreeFeedbackBuffer()` auf. Siehe Abschnitt 6.51 [gl.FreeFeedbackBuffer], Seite 85, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`size` gibt die maximale Anzahl der Werte an, die zurückgegeben werden sollen

`type` gibt eine symbolische Konstante an, die die Informationen beschreibt, die für jeden Knoten zurückgegeben werden; `#GL_2D`, `#GL_3D`, `#GL_3D_COLOR`, `#GL_3D_COLOR_TEXTURE` und `#GL_4D_COLOR_TEXTURE` werden akzeptiert

RÜCKGABEWERTE

`buffer` Zeiger auf den Feedback-Puffer

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `size` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.FeedbackBuffer()` aufgerufen wird, während der Rendermodus `#GL_FEEDBACK` ist oder wenn `gl.RenderMode()` mit dem Argument `#GL_FEEDBACK` aufgerufen wird, bevor `gl.FeedbackBuffer()` mindestens einmal aufgerufen wurde.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.FeedbackBuffer()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_RENDER_MODE`

`gl.Get()` mit dem Argument `#GL_FEEDBACK_BUFFER_POINTER`

`gl.Get()` mit dem Argument `#GL_FEEDBACK_BUFFER_SIZE`

`gl.Get()` mit dem Argument `#GL_FEEDBACK_BUFFER_TYPE`

6.48 gl.Finish

BEZEICHNUNG

gl.Finish – blockiert, bis alle GL-Ausführungen abgeschlossen sind

ÜBERSICHT

```
gl.Finish()
```

BESCHREIBUNG

`gl.Finish()` wird est dann beendet, wenn alle zuvor aufgerufenen GL-Befehle vollständig abgearbeitet sind, die den GL-Status, den Verbindungszustand und den Inhalt des Rahmenpuffers ändern.

`gl.Finish()` erfordert eine Verbindung zum Server.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Finish()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.49 gl.Flush

BEZEICHNUNG

gl.Flush – erzwingt die Ausführung von GL-Befehlen in begrenzter Zeit

ÜBERSICHT

`gl.Flush()`

BESCHREIBUNG

Verschiedene GL-Implementierungen puffern Befehle an mehreren verschiedenen Stellen, darunter Netzwerkpuffer und der Grafikbeschleuniger selbst. glFlush leert alle diese Puffer, so dass alle erteilten Befehle so schnell ausgeführt werden, wie sie von der eigentlichen Rendering-Engine akzeptiert werden. Obwohl diese Ausführung nicht in einem bestimmten Zeitraum abgeschlossen werden kann, wird sie in begrenzter Zeit abgeschlossen.

Da jedes GL-Programm über ein Netzwerk oder auf einem Beschleuniger ausgeführt werden kann, der Befehle puffert, sollten alle Programme `gl.Flush()` aufrufen, wenn sie damit rechnen, dass alle ihre zuvor ausgegebenen Befehle abgeschlossen sind. Rufen Sie beispielsweise `gl.Flush()` auf, bevor Sie auf Benutzereingaben warten, die vom erzeugten Bild abhängen.

`gl.Flush()` wartet nicht, bis die Ausführung aller zuvor ausgegebenen GL-Befehle abgeschlossen sind. Somit wird die Anwendung fortgesetzt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Flush()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.50 gl.Fog

BEZEICHNUNG

gl.Fog – gibt die Nebelparameter an

ÜBERSICHT

```
gl.Fog(pname, param)
```

BESCHREIBUNG

Nebel ist zunächst deaktiviert. Wenn aktiviert, wirkt sich Nebel auf gerasterte Geometrien, Bitmaps und Pixelblöcke aus, nicht aber auf Löschvorgänge. Um Nebel zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_FOG` auf.

`gl.Fog()` weist dem durch `pname` angegebenen Nebelparameter den oder die Werte in Paramentern zu. Die folgenden Werte werden für `pname` akzeptiert:

`#GL_FOG_MODE`
: `param` ist ein einzelner Gleitkommawert, der die Gleichung angibt, die zur Berechnung des Nebelblendfaktors f zu verwenden ist. Drei symbolische Konstanten werden akzeptiert: `#GL_LINEAR`, `#GL_EXP` und `#GL_EXP2`. Die Gleichungen, die diesen symbolischen Konstanten entsprechen, sind im Folgenden definiert. Der anfängliche Nebelmodus ist `#GL_EXP`.

`#GL_FOG_DENSITY`
: `param` ist ein einzelner Gleitkommawert, der die Dichte angibt, die in beiden exponentiellen Nebelgleichungen verwendete Nebeldichte. Es werden nur nichtnegative Dichten akzeptiert. Die anfängliche Nebeldichte ist 1.

`#GL_FOG_START`
: `param` ist ein einzelner Gleitkommawert, der den Start, den in der linearen Nebelgleichung verwendeten Nahbereich, angibt. Der anfängliche Nahbereich ist 0.

`#GL_FOG_END`
: `param` ist ein einzelner Gleitkommawert, der das Ende, den in der linearen Nebelgleichung verwendeten Fernbereich, angibt. Der anfängliche Fernbereich ist 1.

`#GL_FOG_INDEX`
: `param` ist ein einzelner Gleitkommawert, der den Nebelfarbindex i f angibt. Der anfängliche Nebelindex ist 0.

`#GL_FOG_COLOR`
: `param` muss eine Tabelle mit vier Gleitkommawerten sein, die die Nebelfarbe Cf angeben. Alle Farbkomponenten werden in den Bereich [0,1] festgelegt. Die anfängliche Nebelfarbe ist (0, 0, 0, 0).

Fog mischt eine Nebelfarbe mit der Farbe jedes gerasterten Pixelfragments nach der Texturierung unter Verwendung eines Mischfaktors f. Der Faktor f wird je nach Nebelmodus auf eine von drei Arten berechnet. c ist der Abstand in Augenkoordinaten vom Ursprung bis zum vernebelten Fragment. Die Gleichung für `#GL_LINEAR` Nebel lautet wie folgt:

```
f = (end - c) / (end - start)
```

Die Gleichung für `#GL_EXP` Nebel lautet:

```
f = e^(-density*c)
```

Die Gleichung für `#GL_EXP2` Nebel lautet:

```
f = e^(-density*c)^2
```

Unabhängig vom Nebelmodus wird f nach der Berechnung in den Bereich [0,1] festgelegt. Wenn sich GL dann im RGBA-Farbmodus befindet, werden die roten, grünen und blauen Farben des Fragments durch Cr ersetzt-

```
Cr' = f*Cr+(1-f)*Cf
```

Fog beeinflusst nicht die Alpha-Komponente eines Fragments.

Im Farbindexmodus wird der Farbindex ir des Fragments ersetzt durch

```
ir' = f*ir+(1-f)*if
```

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt einen einzeln Nebelparameter an; `#GL_FOG_MODE`, `#GL_FOG_DENSITY`, `#GL_FOG_START`, `#GL_FOG_END`, `#GL_FOG_INDEX` und `#GL_FOG_COLOR` werden akzeptiert

`param` gibt den Wert an, auf den `pname` gesetzt wird

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist, oder wenn `pname` `#GL_FOG_MODE` ist und `param` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `pname` `#GL_FOG_DENSITY` ist und `param` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Fog()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_FOG`

`gl.Get()` mit dem Argument `#GL_FOG_COLOR`

`gl.Get()` mit dem Argument `#GL_FOG_INDEX`

`gl.Get()` mit dem Argument `#GL_FOG_DENSITY`

`gl.Get()` mit dem Argument `#GL_FOG_START`

`gl.Get()` mit dem Argument `#GL_FOG_END`

`gl.Get()` mit dem Argument `#GL_FOG_MODE`

6.51 gl.FreeFeedbackBuffer

BEZEICHNUNG

gl.FreeFeedbackBuffer – löscht den Inhalt des Feedback-Moduspuffer

ÜBERSICHT

```
gl.FreeFeedbackBuffer(buffer)
```

BESCHREIBUNG

Dieser Befehl löscht den Inhalt des Puffers, der von `gl.FeedbackBuffer()` zugewiesen wurde. Siehe Abschnitt 6.47 [gl.FeedbackBuffer], Seite 80, für Details.

Beachten Sie, dass dieser Befehl den Puffer nicht von GL trennt. Dies müssen Sie manuell tun, z.B. durch ändern des Rendermodus mit Hilfe von `gl.RenderMode()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`buffer` ein Puffer, der durch `gl.FeedbackBuffer()` zugewiesen wurde

6.52 gl.FreeSelectBuffer

BEZEICHNUNG

gl.FreeSelectBuffer – löscht den Inhalt eines frei auswählbaren Puffer-Modus

ÜBERSICHT

```
gl.FreeSelectBuffer(buffer)
```

BESCHREIBUNG

Dieser Befehl löscht den Ihnalt eines Puffers, der von `gl.SelectBuffer()` zugewiesen wurde. Siehe Abschnitt 6.129 [gl.SelectBuffer], Seite 198, für Details.

Beachten Sie, dass dieser Befehl den Puffer nicht von GL trennt. Dies müssen Sie manuell tun, z.B. durch ändern des Rendermodus mit Hilfe von `gl.RenderMode()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`buffer` ein Puffer, der durch `gl.SelectBuffer()` zugewiesen wurde

6.53 gl.FrontFace

BEZEICHNUNG

gl.FrontFace – definiert nach vorne und hinten gerichteten Polygone

ÜBERSICHT

```
gl.FrontFace(mode)
```

BESCHREIBUNG

In einer Szene, die ausschließlich aus undurchsichtigen, geschlossenen Oberflächen besteht, sind nach hinten gerichtete Polygone nie sichtbar. Die Eliminierung dieser unsichtbaren Polygone hat den offensichtlichen Vorteil, dass die Darstellung des Bildes beschleunigt wird. Um die Eliminierung von nach hinten gerichteten Polygonen zu aktivieren und zu deaktivieren, rufen Sie den Befehl `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_CULL_FACE` auf.

Die Projektion eines Polygons auf Fensterkoordinaten soll eine Wicklung im Uhrzeigersinn aufweisen, wenn sich ein imaginäres Objekt, das dem Weg von seinem ersten Knoten, seinem zweiten Knoten usw. bis zu seinem letzten Knoten und schließlich zurück zu seinem ersten Knoten folgt, im Uhrzeigersinn um das Innere des Polygons bewegt. Die Wicklung des Polygons gilt als gegen den Uhrzeigersinn, wenn sich das imaginäre Objekt, das dem gleichen Weg folgt, gegen den Uhrzeigersinn um das Innere des Polygons bewegt. `gl.FrontFace()` gibt an, ob Polygone mit einer Drehung im Uhrzeigersinn in Fensterkoordinaten oder einer Drehung gegen den Uhrzeigersinn in Fensterkoordinaten als nach vorne gerichtet betrachtet werden. Wenn Sie `#GL_CCW` an den Modus übergeben, werden Polygone gegen den Uhrzeigersinn als nach vorne gerichtet ausgewählt; `#GL_CW`

wählt Polygone im Uhrzeigersinn als nach vorne gerichtet. Standardmäßig werden Polygone gegen den Uhrzeigersinn als nach vorne gerichtet betrachtet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt die Ausrichtung der nach vorne gerichteten Polygone an; `#GL_CW` und `#GL_CCW` werden akzeptiert; der Initialwert ist `#GL_CCW`

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.FrontFace()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_FRONT_FACE`

6.54 gl.Frustum

BEZEICHNUNG

gl.Frustum – multipliziert die aktuelle Matrix mit einer perspektivischen Matrix

ÜBERSICHT

```
gl.Frustum(left, right, bottom, top, zNear, zFar)
```

BESCHREIBUNG

`gl.Frustum()` beschreibt eine perspektivische Matrix, die eine perspektivische Projektion erzeugt. (left,bottom,-zNear) und (right,top,-zNear) geben die Punkte auf der nahen Ausschnittebene an, die auf die linke bzw. rechte untere und obere Ecke des Fensters abgebildet sind, vorausgesetzt, das Auge befindet sich bei (0, 0, 0, 0). -zFar gibt die Position der entfernten Ausschnittebene an. Sowohl `zNear` als auch `zFar` müssen positiv sein. Konsultieren Sie eine OpenGL-Referenz für die entsprechende Matrix.

Die aktuelle Matrix wird mit dieser Matrix multipliziert, wobei das Ergebnis die aktuelle Matrix ersetzt. Das heißt, wenn M die aktuelle Matrix und F die perspektivische Matrix ist, dann wird M durch M*F ersetzt.

Benutzem Sie `gl.PushMatrix()` und `gl.PopMatrix()` um den aktuellen Matrixstapel zu speichern und wiederherzustellen.

Die Genauigkeit des Tiefenpuffers wird durch die für `zNear` und `zFar` angegebenen Werte beeinflusst. Je größer das Verhältnis von weit zu nah ist, desto weniger effektiv ist der Tiefenpuffer bei der Unterscheidung zwischen Oberflächen, die nahe beieinander liegen. Wenn

```
r = zFar / zNear
```

gehen ungefähr ld(r) Bits der Genauigkeit des Tiefenpuffers verloren. Da r sich der Unendlichkeit nähert, wenn `zNear` sich Null nähert, darf `zNear` niemals auf Null gesetzt werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`left` gibt die Koordinate für die linke vertikale Ausschnittebene an

`right` gibt die Koordinate für die rechte vertikale Ausschnittebene an

`bottom` gibt die Koordinate für die untere horizontale Ausschnittebene an

`top` gibt die Koordinate für die obere horizontale Ausschnittebene an

`zNear` gibt den Abstand zur Ausschnittebene mit niedriger Tiefe an; muss positiv sein.

`zFar` gibt den Abstand zur Ausschnittebene mit großer Tiefe an; er muss positiv sein.

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `zNear` oder `zFar`, oder wenn left = right, oder bottom = top, oder zNear = zFar nicht positiv ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Frustum()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.55 gl.GenLists

BEZEICHNUNG

gl.GenLists – erzeugt einen zusammenhängenden Satz von leeren Display-Listen

ÜBERSICHT

```
num = gl.GenLists(range)
```

BESCHREIBUNG

`gl.GenLists()` hat ein Argument: `range`. Es wird eine ganze Zahl n zurückgegeben, so dass `range` zusammenhängende leere Display-Listen, genannt n, n + 1 , ... , n + Bereich - 1 erstellt werden. Wenn `range` gleich 0 ist, wenn keine Gruppe von Bereichsnamen verfügbar ist oder wenn ein Fehler auftritt, werden keine Display-Listen erzeugt und 0 wird zurückgegeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`range` gibt die Anzahl der zusammenhängenden leeren Display-Listen an, die erzeugt werden sollen.

RÜCKGABEWERTE

`num` Name der ersten leeren Display-Liste

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `range` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GenLists()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN
`gl.IsList()`

6.56 gl.GenTextures

BEZEICHNUNG
gl.GenTextures – generiert Texturnamen

ÜBERSICHT
```
texturesArray = gl.GenTextures(n)
```

BESCHREIBUNG
`gl.GenTextures()` generiert `n` Texturnamen und gibt sie in der Tabelle `texturesArray` zurück. Es gibt keine Garantie, dass die Namen eine zusammenhängende Menge von ganzen Zahlen bilden; es wird jedoch garantiert, dass keiner der zurückgegebenen Namen unmittelbar vor dem Aufruf von `gl.GenTextures()` verwendet wurde.

Die erzeugten Texturen haben keine Dimensionalität; sie nehmen die Dimensionalität des Texturziels an, an das sie zuerst gebunden werden. Siehe Abschnitt 6.6 [gl.BindTexture], Seite 30, für Details.

Texturnamen, die von einem `gl.GenTextures()` Aufruf zurückgegeben werden, werden von nachfolgenden Aufrufen nicht generiert, ausser sie sind zuerst mit `gl.DeleteTextures()` gelöscht worden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`n` gibt die Anzahl der zu generierenden Texturnamen an

RÜCKGABEWERTE

`texturesArray`
Tabelle mit `n` Anzahl Texturnamen

FEHLER
`#GL_INVALID_VALUE` wird erzeugt, wenn `n` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GenTextures()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN
`gl.IsTexture()`

6.57 gl.Get

BEZEICHNUNG
gl.Get – liefert den Wert oder die Werte eines ausgewählten Parameters zurück

ÜBERSICHT
```
param, ... = gl.Get(pname)
```

BESCHREIBUNG

Dieser Befehl gibt Werte für einfache Zustandsvariablen in GL zurück. `pname` ist eine symbolische Konstante, die die zurückzugebende Zustandsvariable angibt. Beachten sie dabei, dass viele der booleschen Parameter auch leichter mit `gl.IsEnabled()` abgefragt werden können. Ausserdem können Sie auch den Befehl `gl.GetArray()` verwenden, der die Werte eines ausgewählten Parameters als ein Feld zurück gibt. Die folgenden symbolischen Konstanten werden durch `pname` akzeptiert:

`#GL_ACCUM_ALPHA_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der Alpha-Bitebenen im Akkumulationspuffer.

`#GL_ACCUM_BLUE_BIT`
: `param` gibt einen Wert zurück: Die Anzahl der blauen Bitebenen im Akkumulationspuffer.

`#GL_ACCUM_CLEAR_VALUE`
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und Alpha-Werte, die zum Löschen des Akkumulationspuffers verwendet werden. Siehe Abschnitt 6.12 [gl.ClearAccum], Seite 38, für Details.

`#GL_ACCUM_GREEN_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der grünen Bitebenen im Akkumulationspuffer.

`#GL_ACCUM_RED_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der roten Bitebenen im Akkumulationspuffer.

`#GL_ALPHA_BIAS`
: `param` gibt einen Wert zurück: Den Alpha-Tendenzfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_ALPHA_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der Alpha-Bitebenen in jedem Farbpuffer.

`#GL_ALPHA_SCALE`
: `param` gibt einen Wert zurück: Den Alpha-Skalierungsfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_ALPHA_TEST`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Alpha-Tests von Fragmenten aktiviert sind. Siehe Abschnitt 6.2 [gl.AlphaFunc], Seite 25, für Details.

`#GL_ALPHA_TEST_FUNC`
: `param` gibt einen Wert zurück: Den symbolischen Namen der Alpha-Test-Funktion. Siehe Abschnitt 6.2 [gl.AlphaFunc], Seite 25, für Details.

`#GL_ALPHA_TEST_REF`
: `param` gibt einen Wert zurück: Den Referenzwert für den Alpha-Test. Siehe Abschnitt 6.2 [gl.AlphaFunc], Seite 25, für Details.

`#GL_ATTRIB_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die Tiefe des Attributstapels. Wenn der Stapel leer ist, wird Null zurückgegeben. Siehe Abschnitt 6.116 [gl.PushAttrib], Seite 179, für Details.

`#GL_AUTO_NORMAL`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Kartenauswertung automatisch Oberflächennormalen erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_AUX_BUFFERS`
: `param` gibt einen Wert zurück: Die Anzahl der Hilfsfarbpuffer.

`#GL_BLEND`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob das Mischen aktiviert ist. Siehe Abschnitt 6.8 [gl.BlendFunc], Seite 33, für Details.

`#GL_BLEND_DST`
: `param` gibt einen Wert zurück: Die symbolische Konstante, die die Zielmischfunktion identifiziert. Siehe Abschnitt 6.8 [gl.BlendFunc], Seite 33, für Details.

`#GL_BLEND_SRC`
: `param` gibt einen Wert zurück: Die symbolische Konstante, die die Quellmischfunktion identifiziert. Siehe Abschnitt 6.8 [gl.BlendFunc], Seite 33, für Details.

`#GL_BLUE_BIAS`
: `param` gibt einen Wert zurück: Den blauen Vorspannungsfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_BLUE_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der blauen Bitebenen in jedem Farbpuffer.

`#GL_BLUE_SCALE`
: `param` gibt einen Wert zurück: Den blauen Skalierungsfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_CLIP_PLANEi`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die angegebene Ausschnittebene aktiviert ist. Siehe Abschnitt 6.17 [gl.ClipPlane], Seite 41, für Details.

`#GL_COLOR_CLEAR_VALUE`
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und Alpha-Werte, die zum Löschen der Farbpuffer verwendet werden. Siehe Abschnitt 6.13 [gl.ClearColor], Seite 38, für Details.

`#GL_COLOR_MATERIAL`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob ein oder mehrere Materialparameter sich auf die aktuelle Farbe auswirkt. Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

`#GL_COLOR_MATERIAL_FACE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Materialien einen auf die aktuelle Farbe auswirkenden Parameter haben. Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

`#GL_COLOR_MATERIAL_PARAMETER`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Materialparameter sich auf die aktuelle Farbe auswirkt. Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

`#GL_COLOR_WRITEMASK`
: `param` gibt vier boolesche Werte zurück: Die roten, grünen, blauen und Alpha-Schreibbefehle für die Farbpuffer. Siehe Abschnitt 6.19 [gl.ColorMask], Seite 42, für Details.

`#GL_CULL_FACE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob das Polygon-Auslesen aktiviert ist. Siehe Abschnitt 6.25 [gl.CullFace], Seite 52, für Details.

`#GL_CULL_FACE_MODE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Polygonflächen geholt werden sollen. Siehe Abschnitt 6.25 [gl.CullFace], Seite 52, für Details.

`#GL_CURRENT_COLOR`
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und Alpha-Werte der aktuellen Farbe. Siehe Abschnitt 6.18 [gl.Color], Seite 42, für Details.

`#GL_CURRENT_INDEX`
: `param` gibt einen Wert zurück: Den aktuellen Farbindex. Siehe Abschnitt 6.76 [gl.Index], Seite 127, für Details.

`#GL_CURRENT_NORMAL`
: `param` gibt drei Werte zurück: Die x-, y- und z-Werte der aktuellen Normalen. Siehe Abschnitt 6.99 [gl.Normal], Seite 157, für Details.

`#GL_CURRENT_RASTER_COLOR`
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und Alpha-Werte der aktuellen Rasterposition. Siehe Abschnitt 6.120 [gl.RasterPos], Seite 186, für Details.

`#GL_CURRENT_RASTER_INDEX`
: `param` gibt einen Wert zurück: Den Farbindex der aktuellen Rasterposition. Siehe Abschnitt 6.120 [gl.RasterPos], Seite 186, für Details.

`#GL_CURRENT_RASTER_POSITION`
: `param` gibt vier Werte zurück: Die x-, y-, z- und w-Komponenten der aktuellen Rasterposition. x, y und z befinden sich in Fensterkoordinaten und

w in Clip-Koordinaten. Siehe Abschnitt 6.120 [gl.RasterPos], Seite 186, für Details.

`#GL_CURRENT_RASTER_TEXTURE_COORDS`
: `param` gibt vier Werte zurück: Die aktuellen Koordinaten der Rastertextur s, t, r und q. Siehe Abschnitt 6.120 [gl.RasterPos], Seite 186, für Details. Siehe Abschnitt 6.134 [gl.TexCoord], Seite 204, für Details.

`#GL_CURRENT_RASTER_POSITION_VALID`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die aktuelle Rasterposition gültig ist. Siehe Abschnitt 6.120 [gl.RasterPos], Seite 186, für Details.

`#GL_CURRENT_TEXTURE_COORDS`
: `param` gibt vier Werte zurück: Die aktuellen Texturkoordinaten s, t, r und q. Siehe Abschnitt 6.134 [gl.TexCoord], Seite 204, für Details.

`#GL_DEPTH_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der Bitebenen im Tiefenpuffer.

`#GL_DEPTH_CLEAR_VALUE`
: `param` gibt einen Wert zurück: Den Wert, der zum Löschen des Tiefenpuffers verwendet wird. Siehe Abschnitt 6.14 [gl.ClearDepth], Seite 39, für Details.

`#GL_DEPTH_FUNC`
: `param` gibt einen Wert zurück: Die symbolische Konstante, die die Tiefentestsfunktion anzeigt. Siehe Abschnitt 6.28 [gl.DepthFunc], Seite 54, für Details.

`#GL_DEPTH_RANGE`
: `param` gibt zwei Werte zurück: Die nahen und fernen Abbildungsgrenzen für den Tiefenpuffer. Siehe Abschnitt 6.30 [gl.DepthRange], Seite 55, für Details.

`#GL_DEPTH_WRITEMASK`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob der Tiefenpuffer zum Schreiben aktiviert ist. Siehe Abschnitt 6.29 [gl.DepthMask], Seite 55, für Details.

`#GL_DOUBLEBUFFER`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob eine doppelte Pufferung unterstützt wird.

`#GL_DRAW_BUFFER`
: `param` gibt einen Wert zurückzurück: Dies ist eine symbolische Konstante, die angibt, zu welchen Puffern gezeichnet wird. Siehe Abschnitt 6.34 [gl.DrawBuffer], Seite 64, für Details.

`#GL_EDGE_FLAG`
: `param` gibt eine einzelne boolesche Wertangabe zurück, ob das aktuelle Flankenflag `True` oder `False` ist. Siehe Abschnitt 6.38 [gl.EdgeFlag], Seite 73, für Details.

`#GL_FOG` `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Fogging aktiviert ist. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_COLOR
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und Alpha-Komponenten der Nebelfarbe. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_DENSITY
: `param` gibt einen Wert zurück: Den Parameter der Nebeldichte. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_END
: `param` gibt einen Wert zurück: Den Endfaktor für die lineare Nebelgleichung. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_HINT
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die den Modus des Nebelhinweises anzeigt. Siehe Abschnitt 6.75 [gl.Hint], Seite 126, für Details.

#GL_FOG_INDEX
: `param` gibt einen Wert zurück: Den Nebelfarbenindex. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_MODE
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Nebelgleichung ausgewählt ist. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FOG_START
: `param` gibt einen Wert zurück: Den Startfaktor für die lineare Nebelgleichung. Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

#GL_FRONT_FACE
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, ob die Polygonwicklung im Uhrzeigersinn oder gegen den Uhrzeigersinn als nach vorne gerichtet behandelt wird. Siehe Abschnitt 6.53 [gl.FrontFace], Seite 86, für Details.

#GL_GREEN_BIAS
: `param` gibt einen Wert zurück: Den grünen Vorspannungsfaktor, der bei der Pixelübertragung verwendet wird.

#GL_GREEN_BITS
: `param` gibt einen Wert zurück: Die Anzahl der grünen Bitebenen in jedem Farbpuffer.

#GL_GREEN_SCALE
: `param` gibt einen Wert zurück: Den grünen Skalierungsfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

#GL_INDEX_BITS
: `param` gibt einen Wert zurück: Die Anzahl der Bitebenen in jedem Farbindexpuffer.

`#GL_INDEX_CLEAR_VALUE`
: `param` gibt einen Wert zurück: Den Farbindex, der zum Löschen der Farbindexpuffer verwendet wird. Siehe Abschnitt 6.15 [gl.ClearIndex], Seite 40, für Details.

`#GL_INDEX_MODE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob sich GL im Farbindexmodus (`True`) oder im RGBA-Modus (`False`) befindet.

`#GL_INDEX_OFFSET`
: `param` gibt einen Wert zurück: Den Versatz, der bei der Pixelübertragung zu Farb- und Schablonenindizes hinzugefügt wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_INDEX_SHIFT`
: `param` gibt einen Wert zurück: Den Betrag, um den sich die Farb- und Schablonenindizes während der Pixelübertragung verschieben. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_INDEX_WRITEMASK`
: `param` gibt einen Wert zurück: Dies ist eine Maske, die angibt, welche Bitebenen jedes Farbindexpuffers geschrieben werden können. Siehe Abschnitt 6.77 [gl.IndexMask], Seite 128, für Details.

`#GL_LIGHTi`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die angegebene Leuchte aktiviert ist. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details.

`#GL_LIGHTING`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Beleuchtung aktiviert ist. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details.

`#GL_LIGHT_MODEL_AMBIENT`
: `param` gibt vier Werte zurück: Die roten, grünen, blauen und alpha-Komponenten der Umgebungsintensität der gesamten Szene. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details.

`#GL_LIGHT_MODEL_LOCAL_VIEWER`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Berechnungen der Spiegelreflexion den Betrachter als lokal zur Szene behandeln. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details.

`#GL_LIGHT_MODEL_TWO_SIDE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob separate Materialien zur Berechnung der Beleuchtung für nach vorne und hinten gerichtete Polygone verwendet werden. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details.

`#GL_LINE_SMOOTH`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Antialiasing-Funktion für Linien aktiviert ist. Siehe Abschnitt 6.87 [gl.LineWidth], Seite 141, für Details.

`#GL_LINE_STIPPLE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob das Punktieren von Linien aktiviert ist. Siehe Abschnitt 6.86 [gl.LineStipple], Seite 140, für Details.

`#GL_LINE_STIPPLE_PATTERN`
: `param` gibt einen Wert zurück: Das 16-Bit-Linienpunktiermuster. Siehe Abschnitt 6.86 [gl.LineStipple], Seite 140, für Details.

`#GL_LINE_STIPPLE_REPEAT`
: `param` gibt einen Wert zurück: Den Wiederholungsfaktor für das Punktieren von Linien. Siehe Abschnitt 6.86 [gl.LineStipple], Seite 140, für Details.

`#GL_LINE_WIDTH`
: `param` gibt einen Wert zurück: Die Linienbreite die mit `gl.LineWidth()` angegeben wird.

`#GL_LINE_WIDTH_GRANULARITY`
: `param` gibt einen Wert zurück: Die Breitendifferenz zwischen benachbarten unterstützten Breiten für antialisierte Linien. Siehe Abschnitt 6.87 [gl.LineWidth], Seite 141, für Details.

`#GL_LINE_WIDTH_RANGE`
: `param` gibt zwei Werte zurück: Die kleinste und die größte unterstützte Breite für Antialias-Linien. Siehe Abschnitt 6.87 [gl.LineWidth], Seite 141, für Details.

`#GL_LIST_BASE`
: `param` gibt einen Wert zurück: Der BasisVersatz von allen Namen in Feldern, die mit `gl.CallLists()` hinzugefügt werden. Siehe Abschnitt 6.88 [gl.ListBase], Seite 143, für Details.

`#GL_LIST_INDEX`
: `param` gibt einen Wert zurück: Den Namen der Display-Liste, die sich gerade im Aufbau befindet. Null wird zurückgegeben, wenn sich derzeit keine Display-Liste im Aufbau befindet. Siehe Abschnitt 6.98 [gl.NewList], Seite 155, für Details.

`#GL_LIST_MODE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die den Baumodus der aktuell erstellten Display-Liste anzeigt. Siehe Abschnitt 6.98 [gl.NewList], Seite 155, für Details.

`#GL_LOGIC_OP`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Fragmentindizes mit Hilfe einer logischen Operation in den Rahmenbuffer zusammengeführt werden. Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_LOGIC_OP_MODE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die den gewählten logischen Betriebsmodus anzeigt. Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_MAP1_COLOR_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung Farben erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_GRID_DOMAIN`
: `param` gibt zwei Werte zurück: Die Endpunkte der Rasterdomäne der 1D-Karte. Siehe Abschnitt 6.94 [gl.MapGrid], Seite 150, für Details.

`#GL_MAP1_GRID_SEGMENTS`
: `param` gibt einen Wert zurück: Die Anzahl der Partitionen in der Rasterdomäne der 1D-Karte. Siehe Abschnitt 6.94 [gl.MapGrid], Seite 150, für Details.

`#GL_MAP1_INDEX`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung Farbindizes erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_NORMAL`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung Normale erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_1`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 1D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_2`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 2D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_3`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 3D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 4D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_VERTEX_3`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 3D-Scheitelpunkt-Koordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_VERTEX_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Auswertung 4D-Scheitelpunkt-Koordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_COLOR_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung Farben erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_GRID_DOMAIN`
: `param` gibt vier Werte zurück: Die Endpunkte der i- und j-Rasterdomänen der 2D-Karte. Siehe Abschnitt 6.94 [gl.MapGrid], Seite 150, für Details.

`#GL_MAP2_GRID_SEGMENTS`
: `param` gibt zwei Werte zurück: Die Anzahl der Partitionen in den i- und j-Gitterbereichen der 2D-Karte. Siehe Abschnitt 6.94 [gl.MapGrid], Seite 150, für Details.

`#GL_MAP2_INDEX`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung Farbindizes erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_NORMAL`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung Normale erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_1`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 1D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_2`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 2D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_3`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 3D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 4D-Texturkoordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_3`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 3D-Scheitelpunkt-Koordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_4`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Auswertung 4D-Scheitelpunkt-Koordinaten erzeugt. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP_COLOR`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Farben und Farbindizes durch Tabellensuche bei der Pixelübertragung ersetzt werden sollen. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_MAP_STENCIL`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Schablonenindizes durch Tabellensuche bei der Pixelübertragung ersetzt werden sollen. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_MATRIX_MODE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welcher Matrixstapel gerade das Ziel aller Matrixoperationen ist. Siehe Abschnitt 6.96 [glMatrixMode], Seite 154, für Details.

`#GL_MAX_ATTRIB_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die maximal unterstützte Tiefe des Attributstapels. Siehe Abschnitt 6.116 [gl.PushAttrib], Seite 179, für Details.

`#GL_MAX_CLIP_PLANES`
: `param` gibt einen Wert zurück: Die maximale Anzahl der von der Anwendung definierten Ausschnittebenen. Siehe Abschnitt 6.17 [gl.ClipPlane], Seite 41, für Details.

`#GL_MAX_EVAL_ORDER`
: `param` gibt einen Wert zurück: Die maximale Gleichungsreihenfolge, unterstützt durch 1D- und 2D-Evaluatoren. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAX_LIGHTS`
: `param` gibt einen Wert zurück: Die maximale Anzahl der Leuchten. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details.

`#GL_MAX_LIST_NESTING`
: `param` gibt einen Wert zurück: Die maximale Rekursionstiefe, die während der Traversierung der Display-Liste erlaubt ist. Siehe Abschnitt 6.9 [gl.CallList], Seite 35, für Details.

`#GL_MAX_MODELVIEW_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die maximal unterstützte Tiefe des Modellansicht-Matrixstapel. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

`#GL_MAX_NAME_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die maximal unterstützte Tiefe des Selektionsnamenstapels. Siehe Abschnitt 6.119 [gl.PushName], Seite 185, für Details.

#GL_MAX_PIXEL_MAP_TABLE
: param gibt einen Wert zurück: Die maximal unterstützte Größe einer glPixelMap Transfer-Tabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

#GL_MAX_PROJECTION_STACK_DEPTH
: param gibt einen Wert zurück: Die maximal unterstützte Tiefe des Projektionsmatrix-Stapels. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

#GL_MAX_TEXTURE_SIZE
: param gibt einen Wert zurück: Die maximale Breite oder Höhe eines Texturbildes (ohne Grenzen). Siehe Abschnitt 6.139 [gl.TexImage1D], Seite 210, für Details. Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für Details.

#GL_MAX_TEXTURE_STACK_DEPTH
: param gibt einen Wert zurück: Die maximal unterstützte Tiefe des Texturmatrixstapels. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

#GL_MAX_VIEWPORT_DIMS
: param gibt zwei Werte zurück: Die maximal unterstützte Breite und Höhe des Ansichtsfensters. Siehe Abschnitt 6.148 [gl.Viewport], Seite 227, für Details.

#GL_MODELVIEW_MATRIX
: param gibt sechzehn Werte zurück: Die Modelansicht-Matrix oben auf dem Modelansicht-Matrix-Stapel. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

#GL_MODELVIEW_STACK_DEPTH
: param gibt einen Wert zurück: Die Anzahl der Matrizen auf dem Matrixstapel der Modellansicht. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

#GL_NAME_STACK_DEPTH
: param gibt einen Wert zurück: Die Anzahl der Namen auf dem Selektionsnamenstapel. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

#GL_NORMALIZE
: param gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Normale automatisch auf die Einheitslänge skaliert werden, nachdem sie in Augenkoordinaten umgewandelt wurden. Siehe Abschnitt 6.99 [gl.Normal], Seite 157, für Details.

#GL_PACK_ALIGNMENT
: param gibt einen Wert zurück: Die Byte-Ausrichtung, die zum Schreiben von Pixeldaten in den Speicher verwendet wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

#GL_PACK_LSB_FIRST
: param gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Einzelbit-Pixel, die in den Speicher geschrieben werden, zuerst in das niederwertigste Bit jedes vorzeichenlosen Bytes geschrieben werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_PACK_ROW_LENGTH`
: `param` gibt einen Wert zurück: Die Zeilenlänge, die zum Schreiben von Pixeldaten in den Speicher verwendet wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_PACK_SKIP_PIXELS`
: `param` gibt einen Wert zurück: Die Anzahl der Pixelpositionen, die übersprungen wurden, bevor das erste Pixel in den Speicher geschrieben wurde. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_PACK_SKIP_ROWS`
: `param` gibt einen Wert zurück: Die Anzahl der Zeilen von Pixelpositionen, die übersprungen wurden, bevor das erste Pixel in den Speicher geschrieben wurde. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_PACK_SWAP_BYTES`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Bytes von Zwei-Byte- und Vier-Byte-Pixelindizes und Komponenten vertauscht werden, bevor sie in den Speicher geschrieben werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_PIXEL_MAP_A_TO_A_SIZE`
: `param` gibt einen Wert in der Größe der Alpha-zu-Alpha-Pixeltransfertabelle zurück. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_B_TO_B_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Transfertabelle von blau nach blauen Pixel. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_G_TO_G_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Transfertabelle von grün nach grünem Pixel. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_I_TO_A_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Index-zu-Alpha-Pixel-Transfertabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_I_TO_B_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Index-zu-Blau-Pixel-Transfertabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_I_TO_G_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Index-zu-Grün-Pixel-Transfertabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_I_TO_I_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Index-zu-Index-Pixel-Transfertabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_I_TO_R_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Index-zu-Rot-Pixel-Transfertabelle. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_R_TO_R_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Transfertabelle von rot nach rotem Pixel. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_PIXEL_MAP_S_TO_S_SIZE`
: `param` gibt einen Wert zurück: Die Größe der Transfertabelle von Schablone zu Schablone Pixel. Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

`#GL_POINT_SIZE`
: `param` gibt einen Wert zurück: Die Punktgröße wie durch `gl.PointSize()` angegeben.

`#GL_POINT_SIZE_GRANULARITY`
: `param` gibt einen Wert zurück: Die Größendifferenz zwischen benachbarten unterstützten Größen für Antialiasing-Punkte. Siehe Abschnitt 6.107 [gl.PointSize], Seite 170, für Details.

`#GL_POINT_SIZE_RANGE`
: `param` gibt zwei Werte zurück: Die kleinste und die größte unterstützte Größe für Antialiasing-Punkte. Siehe Abschnitt 6.107 [gl.PointSize], Seite 170, für Details.

`#GL_POINT_SMOOTH`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Antialiasierung von Punkten aktiviert ist. Siehe Abschnitt 6.107 [gl.PointSize], Seite 170, für Details.

`#GL_POLYGON_MODE`
: `param` gibt zwei Werte zurück: Symbolische Konstanten, die angeben, ob nach vorne gerichtete und nach hinten gerichtete Polygone als Punkte, Linien oder gefüllte Polygone gerastert werden. Siehe Abschnitt 6.108 [gl.PolygonMode], Seite 172, für Details.

`#GL_POLYGON_SMOOTH`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Antialiasing-Funktion für Polygone aktiviert ist. Siehe Abschnitt 6.108 [gl.PolygonMode], Seite 172, für Details.

`#GL_POLYGON_STIPPLE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob das Punktieren von Polygonen aktiviert ist. Siehe Abschnitt 6.110 [gl.PolygonStipple], Seite 174, für Details.

`#GL_PROJECTION_MATRIX`
: `param` gibt sechzehn Werte zurück: Die Projektionsmatrix auf der Oberseite des Projektionsmatrix-Stapels. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

`#GL_PROJECTION_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die Anzahl der Matrizen auf dem Projektionsmatrix-Stapel. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

`#GL_READ_BUFFER`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welcher Farbpuffer zum Lesen ausgewählt ist. Siehe Abschnitt 6.122 [gl.ReadPixels], Seite 189, für Details. Siehe Abschnitt 6.1 [gl.Accum], Seite 23, für Details.

`#GL_RED_BIAS`
: `param` gibt einen Wert zurück: Den roten Tendenzfaktor, der bei der Pixelübertragung verwendet wird.

`#GL_RED_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der roten Bitebenen in jedem Farbpuffer.

`#GL_RED_SCALE`
: `param` gibt einen Wert zurück: Den roten Skalierungsfaktor, der bei der Pixelübertragung verwendet wird. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_RENDER_MODE`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, ob sich GL im Render-, Selektierungs- oder Feedback-Modus befindet. Siehe Abschnitt 6.125 [gl.RenderMode], Seite 193, für Details.

`#GL_RGBA_MODE`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob sich GL im RGBA-Modus (`True`) oder im Farbindex-Modus (`False`) befindet. Siehe Abschnitt 6.18 [gl.Color], Seite 42, für Details.

`#GL_SCISSOR_BOX`
: `param` gibt vier Werte zurück: Die x- und y-Fensterkoordinaten der Scherenbox, gefolgt von ihrer Breite und Höhe. Siehe Abschnitt 6.128 [gl.Scissor], Seite 197, für Details.

`#GL_SCISSOR_TEST`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob das Schneiden aktiviert ist. Siehe Abschnitt 6.128 [glScisscor], Seite 197, für Details.

`#GL_SHADE_MODEL`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, ob der Schattierungs-Modus Einfach- (Flat) ist oder einen Farbverlauf (Smooth) hat. Siehe Abschnitt 6.130 [gl.ShadeModel], Seite 199, für Details.

`#GL_STENCIL_BITS`
: `param` gibt einen Wert zurück: Die Anzahl der Bitebenen im Schablonenpuffer.

`#GL_STENCIL_CLEAR_VALUE`
: `param` gibt einen Wert zurück: Den Index, auf dem die Schablonen-Bitplanes gelöscht werden. Siehe Abschnitt 6.16 [gl.ClearStencil], Seite 40, für Details.

`#GL_STENCIL_FAIL`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Aktion ausgeführt wird, wenn der Schablonentest fehlschlägt. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details.

`#GL_STENCIL_FUNC`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, mit welcher Funktion der Schablonenbezugswert mit dem Schablonenpufferwert verglichen wird. Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details.

`#GL_STENCIL_PASS_DEPTH_FAIL`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Aktion ausgeführt wird, wenn der Schablonentest besteht, aber der Tiefentest fehlschlägt. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details.

`#GL_STENCIL_PASS_DEPTH_PASS`
: `param` gibt einen Wert zurück: Dies ist eine symbolische Konstante, die angibt, welche Aktion ausgeführt wird, wenn der Schablonentest sowie der Tiefentest bestanden wird. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details.

`#GL_STENCIL_REF`
: `param` gibt einen Wert zurück: Den Referenzwert, der mit dem Inhalt des Schablonenpuffers verglichen wird. Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details.

`#GL_STENCIL_TEST`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Schablonentest von Fragmenten aktiviert ist. Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details. Siehe Abschnitt 6.133 [glStencilOp], Seite 203, für Details.

`#GL_STENCIL_VALUE_MASK`
: `param` gibt einen Wert zurück: Die Maske, mit der sowohl der Schablonenbezugswert als auch der Schablonenpufferwert maskiert werden, bevor sie verglichen werden. Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details.

`#GL_STENCIL_WRITEMASK`
: `param` gibt einen Wert zurück: Die Maske, die das Schreiben der Schablonen-Bitebenen steuert. Siehe Abschnitt 6.132 [gl.StencilMask], Seite 202, für Details.

`#GL_STEREO`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Stereo-Puffer (links und rechts) unterstützt werden.

`#GL_SUBPIXEL_BITS`
: `param` gibt einen Wert zurück: Dies ist eine Schätzung der Anzahl der Bits der Subpixelauflösung, die verwendet werden, um gerasterte Geometrie in Fensterkoordinaten zu positionieren.

`#GL_TEXTURE_1D`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 1D-Texturzuordnung aktiviert ist. Siehe Abschnitt 6.139 [gl.TexImage1D], Seite 210, für Details.

`#GL_TEXTURE_2D`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die 2D-Texturzuordnung aktiviert ist. Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für Details.

`#GL_TEXTURE_GEN_S`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die automatische Generierung der S-Texturkoordinate aktiviert ist. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_T`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die automatische Generierung der T-Texturkoordinate aktiviert ist. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_R`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die automatische Generierung der R-Texturkoordinate aktiviert ist. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_Q`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die automatische Generierung der Q-Texturkoordinate aktiviert ist. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_MATRIX`
: `param` gibt sechzehn Werte zurück: Die Texturmatrix oben auf dem Stapel der Texturmatrix. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

`#GL_TEXTURE_STACK_DEPTH`
: `param` gibt einen Wert zurück: Die Anzahl der Matrizen auf dem Texturmatrix-Stapel. Siehe Abschnitt 6.118 [gl.PushMatrix], Seite 185, für Details.

`#GL_UNPACK_ALIGNMENT`
: `param` gibt einen Wert zurück: Die Byte-Ausrichtung, die zum Lesen von Pixeldaten aus dem Speicher verwendet wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_LSB_FIRST`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob Einzelbit-Pixel, die aus dem Speicher gelesen werden, zuerst aus dem

niederwertigsten Bit eines jeden vorzeichenlosen Bytes gelesen werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_ROW_LENGTH`
: `param` gibt einen Wert zurück: Die Zeilenlänge, die zum Lesen von Pixeldaten aus dem Speicher verwendet wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_SKIP_IMAGES`
: `param` gibt einen Wert zurück: Die Anzahl der Bilder, die übersprungen wurden, bevor das erste (3D-)Pixel aus dem Speicher gelesen wurde. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_SKIP_PIXELS`
: `param` gibt einen Wert zurück: Damit wird die Anzahl der Pixelpositionen übersprungen, bevor das erste Pixel aus dem Speicher gelesen wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_SKIP_ROWS`
: `param` gibt einen Wert zurück: Die Anzahl der Zeilen von Pixelpositionen, die übersprungen wurden, bevor das erste Pixel aus dem Speicher gelesen wurde. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_UNPACK_SWAP_BYTES`
: `param` gibt einen einzelnen booleschen Wert zurück: Der gibt an, ob die Bytes von Zwei-Byte- und Vier-Byte-Pixelindizes und Komponenten nach dem Lesen aus dem Speicher vertauscht werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

`#GL_VIEWPORT`
: `param` gibt vier Werte zurück: Die x- und y-Fensterkoordinaten des Ansichtsfensters, gefolgt von seiner Breite und Höhe. Siehe Abschnitt 6.148 [gl.Viewport], Seite 227, für Details.

`#GL_ZOOM_X`
: `param` gibt einen Wert zurück: Den x-Pixel-Zoomfaktor. Siehe Abschnitt 6.106 [gl.PixelZoom], Seite 170, für Details.

`#GL_ZOOM_Y`
: `param` gibt einen Wert zurück: Den y-Pixel-Zoomfaktor. Siehe Abschnitt 6.106 [gl.PixelZoom], Seite 170, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt den zurückzugebenden Parameterwert an (siehe oben für unterstützte Konstanten).

RÜCKGABEWERTE

`param` Wert des angegebenen Parameters

`...` zusätzliche Rückgabewerte je nach `pname`

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Get()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.58 gl.GetArray

BEZEICHNUNG

gl.GetArray – gibt den Wert oder die Werte eines ausgewählten Parameters als ein Feld zurück

ÜBERSICHT

```
paramsArray = gl.GetArray(pname)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.Get()`, außer dass die Werte als Feld zurückgegeben werden. Siehe Abschnitt 6.57 [gl.Get], Seite 89, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt den Parameterwert an, der zurückgegeben werden soll.

RÜCKGABEWERTE

`paramsArray`
Parameterwerte in einem Feld

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn gl.Get() zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.59 gl.GetClipPlane

BEZEICHNUNG

gl.GetClipPlane – gibt die Koeffizienten der angegebenen Ausschnittebene zurück

ÜBERSICHT

```
equationArray = gl.GetClipPlane(plane)
```

BESCHREIBUNG

`gl.GetClipPlane()` gibt in Gleichung die vier Koeffizienten der Ebenengleichung für die Ebene zurück.

Es ist immer so, dass `#GL_CLIP_PLANEi` = `#GL_CLIP_PLANE0 +` i ist.

Wird ein Fehler erzeugt, wird der Inhalt der Gleichung nicht geändert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`plane` gibt eine Ausschnittebene an; die Anzahl der Ausschnittebenen hängt von der Implementierung ab, aber es werden mindestens sechs Ausschnittebenen unterstützt; sie werden durch symbolische Namen von `#GL_CLIP_PLANEi` identifiziert, wobei `i` von 0 bis zum Wert von #GL_MAX_CLIP_PLANES-1 reicht.

RÜCKGABEWERTE

`equationArray`
Tabelle mit vier Doppelpräzisionswerten, die die Koeffizienten der Ebenengleichung der Ebene in Augenkoordinaten sind; der Anfangswert ist (0, 0, 0, 0)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `plane` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetClipPlane()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.60 gl.GetError

BEZEICHNUNG

gl.GetError – gibt Fehlerinformationen zurück

ÜBERSICHT

```
error = gl.GetError()
```

BESCHREIBUNG

`gl.GetError()` gibt den Wert des Fehlerflags zurück. Jedem erkennbaren Fehler sind ein Zahlencode und ein symbolischer Name zugeordnet. Im Fehlerfall wird das Fehlerflag auf den entsprechenden Fehlercodewert gesetzt. Es werden keine weiteren Fehler aufgezeichnet, bis `gl.GetError()` aufgerufen, der Fehlercode zurückgegeben und das Flag auf `#GL_NO_ERROR` zurückgesetzt wird. Wenn ein Aufruf von `gl.GetError()` `#GL_NO_ERROR` zurückgibt, gab es seit dem letzten Aufruf von `gl.GetError()` oder seit der Initialisierung der GL keinen erkennbaren Fehler.

Um verteilte Implementierungen zu ermöglichen, kann es mehrere Fehlerflags geben. Wenn ein einzelnes Fehlerflag einen Fehler aufgezeichnet hat, wird der Wert dieses Flags zurückgegeben und dieses Flag wird auf `#GL_NO_ERROR` zurückgesetzt, wenn `gl.GetError()` aufgerufen wird. Wenn mehr als ein Flag einen Fehler aufgezeichnet hat, gibt `gl.GetError()` einen beliebigen Fehler-Flagwert zurück und löscht ihn. Daher sollte `gl.GetError()` immer in einer Schleife aufgerufen werden, bis es `#GL_NO_ERROR` zurückgibt, wenn alle Fehlerflags zurückgesetzt werden sollen.

Zunächst werden alle Fehlerflags auf `#GL_NO_ERROR` gesetzt.

Die folgenden Fehler sind derzeit definiert:

`#GL_NO_ERROR`
Es wurde kein Fehler festgestellt. Der Wert dieser symbolischen Konstante ist garantiert 0.

`#GL_INVALID_ENUM`
: Für ein aufgezähltes Argument wird ein inakzeptabler Wert angegeben. Der fehlerhafte Befehl wird ignoriert und hat keine andere Nebeneffekte, als das Fehlerflags zu setzen.

`#GL_INVALID_VALUE`
: Ein numerisches Argument liegt außerhalb des Bereichs. Der fehlerhafte Befehl wird ignoriert und hat keine andere Nebeneffekte, als das Fehlerflags zu setzen.

`#GL_INVALID_OPERATION`
: Die angegebene Operation ist im aktuellen Status nicht erlaubt. Der fehlerhafte Befehl wird ignoriert und hat keine andere Nebeneffekte, als das Fehlerflags zu setzen.

`#GL_STACK_OVERFLOW`
: Dieser Befehl würde einen Stapelüberlauf verursachen. Der fehlerhafte Befehl wird ignoriert und hat keine andere Nebenwirkung, als das Fehlerflags zu setzen.

`#GL_STACK_UNDERFLOW`
: Dieser Befehl würde einen Stapelunterlauf verursachen. Der fehlerhafte Befehl wird ignoriert und hat keine andere Nebenwirkung, als das Fehlerflags zu setzen.

`#GL_OUT_OF_MEMORY`
: Es ist nicht genügend Speicherplatz vorhanden, um den Befehl auszuführen. Der Status von GL ist undefiniert, außer dem Status der Fehlerflags, nachdem dieser Fehler aufgezeichnet wurde.

Wenn ein Fehlerflag gesetzt ist, sind die Ergebnisse einer GL-Operation nur dann undefiniert, wenn `#GL_OUT_OF_MEMORY` aufgetreten ist. In allen anderen Fällen wird der Befehl, der den Fehler erzeugt, ignoriert und hat keinen Einfluss auf den GL-Status- oder Rahmenpufferinhalt. Wenn der erzeugende Befehl einen Wert zurückgibt, gibt er 0 zurück. Wenn `gl.GetError()` selbst einen Fehler erzeugt, gibt er 0 zurück.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

RÜCKGABEWERTE

`error` Wert des GL-Fehlerflags

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetError()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird. In diesem Fall gibt `gl.GetError()` 0 zurück.

6.61 gl.GetLight

BEZEICHNUNG

gl.GetLight – gibt die Lichtquellen-Parameterwerten zurück

ÜBERSICHT

```
paramsArray = gl.GetLight(light, pname)
```

BESCHREIBUNG

`gl.GetLight()` gibt in `paramsArray` den Wert oder die Werte eines Lichtquellenparameters zurück. `light` nennt das Licht und ist ein symbolischer Name der Form `#GL_LIGHTi`, wobei `i` von 0 bis zum Wert von `#GL_MAX_LIGHTS` - 1 reicht. `#GL_MAX_LIGHTS` ist eine implementierungsabhängige Konstante, die größer oder gleich acht ist. `pname` gibt einen von zehn Lichtquellenparametern an, wiederum durch einen symbolischen Namen.

Die folgenden Parameter sind definiert:

`#GL_AMBIENT`
: Liefert vier Gleitkommawerte, die die Umgebungsintensität der Lichtquelle darstellen. Der Anfangswert ist (0, 0, 0, 1).

`#GL_DIFFUSE`
: Liefert vier Gleitkommawerte, die die diffuse Intensität der Lichtquelle darstellen. Der Anfangswert für `#GL_LIGHT0` ist (1, 1, 1, 1); für andere Lichtquelle ist der Anfangswert (0, 0, 0, 0).

`#GL_SPECULAR`
: Liefert vier Gleitkommawerte, die die Spiegelintensität der Lichtquelle darstellen. Der Anfangswert für `#GL_LIGHT0` ist (1, 1, 1, 1); für andere Lichtquelle ist der Anfangswert (0, 0, 0, 0).

`#GL_POSITION`
: Liefert vier Gleitkommawerte, die die Position der Lichtquelle darstellen. Die zurückgegebenen Werte sind diejenigen, die in Augenkoordinaten gepflegt sind. Sie sind nicht gleich den Werten, die mit `gl.Light()` angegeben wurden, es sei denn, die Modellansichtsmatrix war zu dem Zeitpunkt identisch als `gl.Light()` aufgerufen wurde. Der Anfangswert ist (0, 0, 1, 0).

`#GL_SPOT_DIRECTION`
: Liefert drei Gleitkommawerte, die die Richtung der Lichtquelle darstellen. Die zurückgegebenen Werte sind diejenigen, die in Augenkoordinaten gepflegt sind. Sie sind nicht gleich den Werten, die mit `gl.Light()` angegeben wurden, es sei denn, die Modellansichtsmatrix war zum Zeitpunkt identisch als `gl.Light()` aufgerufen wurde. Obwohl die Spotrichtung normiert wird, bevor sie in der Beleuchtungsgleichung verwendet wird, sind die zurückgegebenen Werte die transformierten Versionen der angegebene Werte vor der Normalisierung. Der Anfangswert ist (0,0,-1).

`#GL_SPOT_EXPONENT`
: Liefert einen einzelnen Gleitkommawert, der den Punkt-Exponenten des Lichts darstellt. Der Anfangswert ist 0.

`#GL_SPOT_CUTOFF`
: Liefert einen einzelnen Gleitkommawert, der den Schnittwinkel des Lichts repräsentiert. Der Anfangswert ist 180.

`#GL_CONSTANT_ATTENUATION`
: Liefert einen einzelnen Gleitkommawert, der die konstante (nicht entfernungsabhängige) Dämpfung des Lichts darstellt. Der Anfangswert ist 1.

`#GL_LINEAR_ATTENUATION`
: Liefert einen einzelnen Gleitkommawert, der die lineare Dämpfung des Lichts darstellt. Der Anfangswert ist 0.

`#GL_QUADRATIC_ATTENUATION`
: Liefert einen einzelnen Gleitkommawert, der die quadratische Dämpfung des Lichts darstellt. Der Anfangswert ist 0.

Es ist immer der Fall, dass `#GL_LIGHT` i = `#GL_LIGHT0 +` i ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`light` gibt eine Lichtquelle an; die Anzahl der möglichen Lichtquellen hängt von der Implementierung ab, aber es werden mindestens acht Lichtquellen unterstützt; sie werden durch symbolische Namen der Form `#GL_LIGHTi` identifiziert, wobei `i` von 0 bis zum Wert von #GL_MAX_LIGHTS-1 geht

`pname` gibt einen Lichtquellenparameter für `light` an (siehe oben für mögliche Parameter).

RÜCKGABEWERTE

`paramsArray`
: Tabelle mit den angeforderten Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `light` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetLight()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.62 gl.GetMap

BEZEICHNUNG

gl.GetMap – gibt die Parameter des Evaluators zurück

ÜBERSICHT

```
vArray = gl.GetMap(target, query)
```

BESCHREIBUNG

`gl.Map()` definiert einen Evaluator. `gl.GetMap()` gibt Evaluatoren-Parameter zurück. `target` wählt eine Karte aus und `query` wählt einen bestimmten Parameter aus. Die folgenden Werte werden derzeit für `target` unterstützt:

```
#GL_MAP1_COLOR_4
#GL_MAP1_INDEX
#GL_MAP1_NORMAL
#GL_MAP1_TEXTURE_COORD_1
#GL_MAP1_TEXTURE_COORD_2
#GL_MAP1_TEXTURE_COORD_3,
#GL_MAP1_TEXTURE_COORD_4
#GL_MAP1_VERTEX_3
```

```
#GL_MAP1_VERTEX_4
#GL_MAP2_COLOR_4
#GL_MAP2_INDEX
#GL_MAP2_NORMAL
#GL_MAP2_TEXTURE_COORD_1
#GL_MAP2_TEXTURE_COORD_2
#GL_MAP2_TEXTURE_COORD_3
#GL_MAP2_TEXTURE_COORD_4
#GL_MAP2_VERTEX_3
#GL_MAP2_VERTEX_4
```

Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`query` kann die folgenden Werte annehmen:

`#GL_COEFF`
: v gibt die Kontrollpunkte für die Evaluatoren-Funktion zurück. Eindimensionale Evaluatoren geben Order-Kontrollpunkte zurück und zweidimensionale Evaluatoren geben uorder*vorder Kontrollpunkte zurück. Jeder Kontrollpunkt besteht aus einem, zwei, drei oder vier doppelpräzisen Gleitkommawerten. GL gibt zweidimensionale Kontrollpunkte in zeilenweiser Reihenfolge zurück und erhöht den Uorder-Index schnell und den Vorderindex nach jeder Zeile.

`#GL_ORDER`
: v gibt die Reihenfolge der Evaluatoren-Funktion zurück. Eindimensionale Evaluatoren liefern einen Einzelwert: order. Der Anfangswert ist 1. Zweidimensionale Evaluatoren geben zwei Werte zurück: uorder und vorder. Der Anfangswert ist (1,1).

`#GL_DOMAIN`
: v gibt die linearen u- und v-Zuordnungs-Parameter zurück. Eindimensionale Evaluatoren liefern zwei Werte, u1 und u2, wie durch `gl.Map()` angegeben. Zweidimensionale Evaluatoren geben vier Werte (u1, u2, v1 und v2) zurück, wie in der Datei `gl.Map()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt den symbolischen Namen einer Karte an (siehe oben für mögliche Werte).

`query` gibt an, welcher Parameter zurückgegeben werden soll (siehe oben für mögliche Werte).

RÜCKGABEWERTE

`vArray` Tabelle mit den angeforderten Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `either` Ziel oder `query` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetMap()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.63 gl.GetMaterial

BEZEICHNUNG

gl.GetMaterial – gibt Material-Parameter zurück

ÜBERSICHT

```
paramsArray = gl.GetMaterial(face, pname)
```

BESCHREIBUNG

`gl.GetMaterial()` gibt eine Tabelle zurück, die den Wert oder die Werte des Parameters `pname` des Materials `face` enthält. Die folgenden sechs Parameter können in `pname` übergeben werden:

`#GL_AMBIENT`
: Liefert vier Gleitkommawerte, die die Umgebungsreflexion des Materials darstellen. Der Anfangswert ist (0.2, 0.2, 0.2, 1.0)

`#GL_DIFFUSE`
: Liefert vier Gleitkommawerte, die den diffusen Reflexionsgrad des Materials darstellen. Der Anfangswert ist (0.8, 0.8, 0.8, 1.0).

`#GL_SPECULAR`
: Liefert vier Gleitkommawerte, die den Spiegelreflexionsgrad des Materials darstellen. Der Anfangswert ist (0, 0, 0, 1).

`#GL_EMISSION`
: Liefert vier Gleitkommawerte, die die emittierte Lichtintensität des Materials darstellen. Der Anfangswert ist (0, 0, 0, 1).

`#GL_SHININESS`
: Liefert einen Gleitkommawert, der den Spiegelexponenten des Materials darstellt. Der Anfangswert ist 0.

`#GL_COLOR_INDEXES`
: Liefert drei Gleitkommawerte, die den Umgebungs-, Diffus- und Spiegelindex des Materials darstellen. Diese Indizes werden nur für die Farbindexbeleuchtung verwendet (alle anderen Parameter werden nur für die RGBA-Beleuchtung verwendet).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`face` gibt an, welches der beiden Materialien abgefragt wird; `#GL_FRONT` oder `#GL_BACK` werden akzeptiert und repräsentieren das Vorder- bzw. Rückseitenmaterial

`pname` gibt den zurückzugebenden Materialparameter an (siehe oben für mögliche Werte)

RÜCKGABEWERTE

`paramsArray`
: Tabelle mit den angeforderten Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `face` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetMaterial()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.64 gl.GetPixelMap

BEZEICHNUNG

gl.GetPixelMap – gibt die angegebene Pixelkarte zurück

ÜBERSICHT

```
valuesArray = gl.GetPixelMap(map)
```

BESCHREIBUNG

`gl.GetPixelMap()` gibt den Inhalt der in `map` angegebenen Pixelkarte zurück. Dies kann eine der folgenden Konstanten sein:

```
#GL_PIXEL_MAP_I_TO_I
#GL_PIXEL_MAP_S_TO_S
#GL_PIXEL_MAP_I_TO_R
#GL_PIXEL_MAP_I_TO_G
#GL_PIXEL_MAP_I_TO_B
#GL_PIXEL_MAP_I_TO_A
#GL_PIXEL_MAP_R_TO_R
#GL_PIXEL_MAP_G_TO_G
#GL_PIXEL_MAP_B_TO_B
#GL_PIXEL_MAP_A_TO_A
```

Siehe Abschnitt 6.103 [gl.PixelMap], Seite 161, für Details.

Pixelkarten werden bei der Ausführung von `gl.ReadPixels()`, `gl.DrawPixels()`, `gl.CopyPixels()`, `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage1D()`, `gl.TexSubImage2D()`, `gl.CopyTexImage()` und `gl.CopyTexSubImage()` benutzt, um Farbindizes, Schablonenindizes, Farbkomponenten und Tiefenkomponenten auf andere Werte abzubilden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`map`
: gibt den Namen der zurückzugebenden Pixelkarte an (siehe oben für mögliche Werte)

RÜCKGABEWERTE

`valuesArray`
: Tabelle mit dem Inhalt der Pixelkarte

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `map` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetPixelMap()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_I_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_S_TO_S_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_R_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_G_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_B_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_A_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_R_TO_R_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_G_TO_G_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_B_TO_B_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_A_TO_A_SIZE`

`gl.Get()` mit dem Argument `#GL_MAX_PIXEL_MAP_TABLE`

6.65 gl.GetPointer

BEZEICHNUNG

gl.GetPointer – gibt die Werte des angegebenen Zeigers zurück

ÜBERSICHT

```
valuesArray = gl.GetPointer(pname, n)
```

BESCHREIBUNG

`gl.GetPointer()` gibt Elemente zurück, die von einem GL-Zeiger gelesen wurden. `pname` ist eine symbolische Konstante, die den zu verwendenden Zeiger angibt und `n` gibt an, wie viele Elemente gelesen und zurückgegeben werden sollen. `pname` kann auf die folgenden Werte gesetzt werden:

```
#GL_COLOR_ARRAY_POINTER
#GL_EDGE_FLAG_ARRAY_POINTER
#GL_FEEDBACK_BUFFER_POINTER
#GL_INDEX_ARRAY_POINTER
#GL_NORMAL_ARRAY_POINTER
#GL_SELECTION_BUFFER_POINTER
#GL_TEXTURE_COORD_ARRAY_POINTER
#GL_VERTEX_ARRAY_POINTER
```

Die Zeiger haben alle Klient-seitige Zustände.

Der Anfangswert für jeden Zeiger ist `NULL`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt das abzufragende Feld oder den Pufferzeiger an (siehe oben für mögliche Werte).

`n` Anzahl der Elemente, die vom Zeiger gelesen werden sollen

RÜCKGABEWERTE

`valuesArray`
Tabelle mit `n`-Einträgen, die vom jeweiligen Zeiger gelesen werden

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

6.66 gl.GetPolygonStipple

BEZEICHNUNG

gl.GetPolygonStipple – gibt das Polygon-Punktemuster zurück

ÜBERSICHT

```
maskArray = gl.GetPolygonStipple()
```

BESCHREIBUNG

`gl.GetPolygonStipple()` kehrt zum Muster eines 32*32 Polygon-Punktemusters zurück. Das Muster wird so in den Speicher gepackt, als ob `gl.ReadPixels()` mit einer Höhe und Breite von 32, einem Typ von `#GL_BITMAP` und einem Format von `#GL_COLOR_INDEX` aufgerufen und das Punktmuster in einem internen 32 * 32 Muster Farbindexpuffer gespeichert worden wäre. Im Gegensatz zu `gl.ReadPixels()`, werden jedoch Pixelübertragungsvorgänge (Shift, Versatz, Pixelmap) nicht auf das zurückgegebene Punkteditorbild angewendet. Da `#GL_BITMAP` nur 1 Bit pro Pixel verwendet, wird die Tabelle zurückgegeben, die durch diese Funktion immer genau 128 Elemente haben wird und die 8 Pixel pro Tabellenelement enthalten.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

RÜCKGABEWERTE

`maskArray`
Tabelle, die das Punktemuster enthält; der Anfangswert ist bei allen 1

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetPolygonStipple()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.67 gl.GetSelectBuffer

BEZEICHNUNG

gl.GetSelectBuffer – gibt den Wert aus dem Auswahlpuffer zurück

ÜBERSICHT

```
value = gl.GetSelectBuffer(buffer, index)
```

BESCHREIBUNG

Dieser Befehl kann verwendet werden, um den Wert beim Index `index` im Auswahlpuffer zu lesen, der im `buffer` übergeben wird. Dieser Puffer muss durch `gl.SelectBuffer()` zugewiesen worden sein. Werte werden als vorzeichenlose vier Byte Ganzzahlen ab Index 0 gelesen.

Siehe Abschnitt 6.129 [gl.SelectBuffer], Seite 198, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`buffer` Speicherpuffer, der durch `gl.SelectBuffer()` zugewiesen wird

`index` Index des zu lesenden Wertes (ab Index 0)

RÜCKGABEWERTE

`value` Wert am angegebenen Index

6.68 gl.GetString

BEZEICHNUNG

gl.GetString – gibt eine Zeichenkette zurück, die die aktuelle GL-Anbindung beschreibt

ÜBERSICHT

```
string = gl.GetString(name)
```

BESCHREIBUNG

`gl.GetString()` gibt einen Zeiger auf eine statische Zeichenkette zurück, die einen Aspekt der aktuellen GL-Anbindung beschreibt. Der Name kann eine der folgenden Konstanten sein:

`#GL_VENDOR`
: Liefert den Hersteller, der für diese GL-Implementierung verantwortlich ist. Dieser Name ändert sich nicht von Veröffentlichung zu Veröffentlichung.

`#GL_RENDERER`
: Liefert den Namen des Renderers. Dieser Name ist typischerweise spezifisch für eine bestimmte Konfiguration einer Hardwareplattform. Er ändert sich nicht von Veröffentlichung zu Veröffentlichung.

`#GL_VERSION`
: Liefert eine Versions- oder Veröffentlichungs-Nummer.

`#GL_EXTENSIONS`
: Liefert eine durch Leerzeichen getrennte Liste der unterstützten Erweiterungen für GL.

Da GL keine Abfragen nach den Leistungsmerkmalen einer Implementierung beinhaltet, werden einige Anwendungen geschrieben, um bekannte Plattformen zu erkennen und ihre GL-Nutzung basierend auf den bekannten Leistungsmerkmalen dieser Plattformen zu ändern. Die Zeichenketten `#GL_VENDOR` und `#GL_RENDERER` zusammen geben eine Plattform eindeutig an. Sie wechseln nicht von Veröffentlichung zu Veröffentlichung und sollten von Plattform-Erkennungsalgorithmen verwendet werden.

Einige Anwendungen wollen Funktionen nutzen, die nicht zum Standard-GL gehören. Diese Funktionen können als Erweiterung des Standard-GL implementiert werden. Die Zeichenkette `#GL_EXTENSIONS` ist eine durch Leerzeichen getrennte Liste der unterstützten GL-Erweiterungen. (Erweiterungsnamen enthalten nie ein Leerzeichen.)

Die Zeichenkette `#GL_VERSION` beginnt mit einer Versionsnummer. Die Versionsnummer verwendet eines dieser Formen:

```
<major_number>.<minor_number>
<major_number>.<minor_number>.<release_number>
```

Anbieterspezifische Informationen können der Versionsnummer folgen. Sein Format hängt von der Implementierung ab, aber ein Leerzeichen trennt immer die Versionsnummer und die herstellerspezifischen Informationen.

Der Klient und der Server können verschiedene Versionen oder Erweiterungen unterstützen. `gl.GetString()` gibt immer eine kompatible Versionsnummer oder eine Liste von Erweiterungen zurück. Die Veröffentlichungs-Nummer beschreibt immer den Server.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`name` gibt eine symbolische Konstante an (siehe oben für mögliche Werte)

RÜCKGABEWERTE

`string` Zeichenkette, die die aktuelle GL-Anbindung beschreibt

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `name` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetString()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.69 gl.GetTexEnv

BEZEICHNUNG

gl.GetTexEnv – gibt Textur-Umgebungsparameter zurück

ÜBERSICHT

```
paramsArray = gl.GetTexEnv(pname)
```

BESCHREIBUNG

`gl.GetTexEnv()` gibt eine Tabelle mit ausgewählten Werten einer Texturumgebung zurück, die mit `gl.TexEnv()` angegeben wurde. `pname` benennt einen bestimmten Parameter der Texturumgebung. Die beiden Parameter sind wie folgt:

`#GL_TEXTURE_ENV_MODE`
: Liefert einen einwertigen Textur-Umgebungsmodus, eine symbolische Konstante.

`#GL_TEXTURE_ENV_COLOR`
: Liefert vier Gleitkommawerte, die die Farbe der Texturumgebung darstellen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt den symbolischen Namen eines Textur-Umgebungsparameters an (siehe oben für mögliche Werte)

RÜCKGABEWERTE

`paramsArray`
Tabelle mit den angeforderten Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexEnv()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.70 gl.GetTexGen

BEZEICHNUNG

gl.GetTexGen – gibt Textur-Koordinaten-Generierungsparameter zurück

ÜBERSICHT

```
paramsArray = gl.GetTexGen(coord, pname)
```

BESCHREIBUNG

`gl.GetTexGen()` gibt eine Tabelle mit ausgewählten Parametern eines Textur-Koordinaten-Generierungsparameters zurück, die mit `gl.TexGen()` angegeben wurde. `coord` benennt eine der Texturkoordinaten (s, t, r, q) unter Verwendung der symbolischen Konstanten `#GL_S`, `#GL_T`, `#GL_R` oder `#GL_Q`.

`pname` gibt einen von drei symbolischen Namen an:

`#GL_TEXTURE_GEN_MODE`
Liefert die Funktion zur Erzeugung von Einzelwert-Texturen, eine symbolische Konstante. Der Initialwert ist `#GL_EYEYE_LINEAR`.

`#GL_OBJECT_PLANE`
Dies gibt die vier Ebenengleichungskoeffizienten zurück, die die lineare Koordinatengenerierung des Objekts angibt.

`#GL_EYE_PLANE`
Liefert die vier Ebenengleichungskoeffizienten, die die Erzeugung von linearen Augenkoordinaten festlegen. Die zurückgegebenen Werte sind diejenigen, die in Augenkoordinaten gepflegt sind. Sie sind nicht gleich den mit `gl.TexGen()` angegebenen Werten, es sei denn, die Modelansicht-Matrix war identisch, wenn `gl.TexGen()` aufgerufen wurde.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`coord` gibt eine Texturkoordinate an

`pname` gibt den symbolischen Namen der zurückzugebenden Werte an

RÜCKGABEWERTE

`paramsArray`
Tabelle mit den angeforderten Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `coord` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexGen()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.71 gl.GetTexImage

BEZEICHNUNG

gl.GetTexImage – gibt ein Texturbild zurück

ÜBERSICHT

```
pixelsArray = gl.GetTexImage(target, level, format)
```

BESCHREIBUNG

`gl.GetTexImage()` gibt die Pixel eines Texturbildes zurück. Eindimensionale Texturen werden in einer eindimensionalen Tabelle zurückgegeben, während zweidimensionale Texturen in einer Tabelle zurückgegeben werden, die Untertabellen für alle Zeilen der Textur enthält. Die Pixel werden als Werte vom Typ `#GL_FLOAT` zurückgegeben. `target` gibt an, ob das gewünschte Texturbild eines ist, das durch `gl.TexImage1D()` (`#GL_TEXTURE_1D`) oder `gl.TexImage2D()` (`#GL_TEXTURE_2D`) angegeben wird. `level` gibt die Detaillierungsstufe des gewünschten Bildes an. `format` gibt das Format des gewünschten Bildfeldes an. Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für eine Beschreibung der zulässigen Werte für den Formatparameter..

Um die Funktionsweise von `gl.GetTexImage()` zu verstehen, betrachten Sie das ausgewählte interne Vierkomponenten-Texturbild als einen RGBA-Farbpuffer in der Größe des Bildes. Die Semantik von `gl.GetTexImage()` ist dann identisch mit `gl.ReadPixels()`, mit der Ausnahme, dass keine Pixelübertragungsoperationen durchgeführt werden, wenn sie mit dem gleichen Format und Typ aufgerufen werden, wobei x und y auf 0 gesetzt sind, die Breite auf die Breite des Texturbildes (einschließlich Rand, wenn einer angegeben wurde) und die Höhe auf 1 für 1D-Bilder oder auf die Höhe des Texturbildes (einschließlich Rand, wenn einer angegeben wurde) für 2D-Bilder eingestellt ist. Da das interne Texturbild ein RGBA-Bild ist, werden die Pixelformate `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX` und `#GL_DEPTH_COMPONENT` sowie der Pixeltyp `#GL_BITMAP` nicht akzeptiert.

Wenn das ausgewählte Texturbild keine vier Komponenten enthält, werden die folgenden Zuordnungen angewendet: Einkomponenten-Texturen werden als RGBA-Puffer mit Rot auf den Einkomponentenwert, Grün auf 0, Blau auf 0 und Alpha auf 1 gesetzt. Zweikomponenten-Texturen werden als RGBA-Puffer mit Rot auf den Wert der Komponente 0, Alpha auf den Wert der Komponente 1 und Grün und Blau auf 0 gesetzt. 3-Komponenten-Texturen werden schließlich als RGBA-Puffer mit Rot auf Komponente 0, Grün auf Komponente 1, Blau auf Komponente 2 und Alpha auf 1 gesetzt.

Wenn Sie eine fein abgestimmte Kontrolle über den Pixeltyp haben möchten oder wenn Sie wollen, dass die Pixel in einen Speicherpuffer anstelle einer Tabelle geschrieben werden, können Sie den Befehl `gl.GetTexImageRaw()` verwenden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt an, welche Textur erhalten werden soll (muss `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` sein)

`level` gibt die Detaillierungsstufe des gewünschten Bildes an; Stufe 0 ist die Basis-Bildebene; Stufe n ist das n-te Mipmap-Reduktionsbild

`format` gibt ein Pixelformat für die zurückgegebenen Daten an; die unterstützten Formate sind `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` und `#GL_LUMINANCE_ALPHA`

RÜCKGABEWERTE

`pixelsArray`
Tabelle mit den Rohpixeln

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` oder `format` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` kleiner als Null oder größer als ld(max) ist, wobei `max` der Rückgabewert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexImage()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_WIDTH`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_HEIGHT`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_BORDER`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_COMPONENTS`

`gl.Get()` mit dem Argumente `#GL_PACK_ALIGNMENT` und andere

6.72 gl.GetTexImageRaw

BEZEICHNUNG

gl.GetTexImageRaw – gibt ein Texturbild zurück

ÜBERSICHT

```
gl.GetTexImageRaw(target, level, format, type, pixels)
```

BESCHREIBUNG

`gl.GetTexImageRaw()` schreibt die Pixel eines Texturbildes in `pixels`. Dies muss ein Speicherpuffer sein, der von dem Hollywood-Befehl `AllocMem()` zugewiesen und von `GetMemPointer()` zurückgegeben wird. Um die erforderliche Größe der `pixels` zu bestimmen, verwenden Sie `gl.GetTexLevelParameter()`, um die Abmessungen des internen Texturbildes zu bestimmen und skalieren dann die erforderliche Anzahl von Pixeln

durch den für jedes Pixel erforderlichen Speicherplatz, basierend auf `format` und `type`. Achten Sie darauf, dass Sie die Pixelspeicherparameter berücksichtigen, insbesondere `#GL_PACK_ALIGNMENT`.

Die unterstützten Werte für `format` sind `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` und `#GL_LUMINANCE_ALPHA`.

Unterstützte Datentypen für `type` sind `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` und `#GL_FLOAT`.

Die Pixel werden als Werte vom Typ `type` in den Speicherpuffer geschrieben. `target` gibt an, ob das gewünschte Texturbild eines ist, das durch `gl.TexImage1D()` (`#GL_TEXTURE_1D`) oder `gl.TexImage2D()` (`#GL_TEXTURE_2D`) angegeben wird. `level` gibt die Detaillierungsstufe des gewünschten Bildes an. `format` gibt das Format des gewünschten Bildfeldes an. Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für eine Beschreibung der zulässigen Werte für den Formatparameter.

Um die Funktionsweise von `gl.GetTexImageRaw()` zu verstehen, betrachten Sie das ausgewählte interne Vierkomponenten-Texturbild als einen RGBA-Farbpuffer in der Größe des Bildes. Die Semantik von `gl.GetTexImageRaw()` ist dann identisch mit der von `gl.ReadPixels()`, mit der Ausnahme, dass keine Pixelübertragungsoperationen durchgeführt werden, wenn sie mit dem gleichen Format und Typ aufgerufen werden, wobei x und y auf 0 gesetzt sind, die Breite auf die Breite des Texturbildes (einschließlich Rand, wenn einer angegeben wurde) und die Höhe auf 1 für 1D-Bilder oder auf die Höhe des Texturbildes (einschließlich Rand, wenn einer angegeben wurde) eingestellt wird für 2D-Bilder. Da das interne Texturbild ein RGBA-Bild ist, werden die Pixelformate `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX` und `#GL_DEPTH_COMPONENT` sowie der Pixeltyp `#GL_BITMAP` nicht akzeptiert.

Wenn das ausgewählte Texturbild keine vier Komponenten enthält, werden die folgenden Zuordnungen angewendet: Einkomponenten-Texturen werden als RGBA-Puffer mit Rot auf den Einkomponentenwert, Grün auf 0, Blau auf 0 und Alpha auf 1 gesetzt. Zweikomponenten-Texturen werden als RGBA-Puffer mit Rot auf den Wert der Komponente 0, Alpha auf den Wert der Komponente 1 und Grün und Blau auf 0 gesetzt. 3-Komponenten-Texturen werden schließlich als RGBA-Puffer mit Rot auf Komponente 0, Grün auf Komponente 1, Blau auf Komponente 2 und Alpha auf 1 gesetzt.

Wenn Sie möchten, dass die Pixel in einer Tabelle anstelle eines Speicherpuffers zurückgegeben werden, können Sie den Befehl `gl.GetTexImage()` verwenden. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt an, welche Textur erhalten werden soll (muss `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` sein)

`level` gibt die Detaillierungsstufe des gewünschten Bildes an; Stufe 0 ist die Basis-Bildebene; Stufe n ist das n-te Mipmap-Reduktionsbild

`format` gibt ein Pixelformat für die zurückgegebenen Daten an (siehe oben)

`type` gibt einen Pixeltyp für die zurückgegebenen Daten an (siehe oben)

`pixels` zeigt auf einen Speicherpuffer, um die Pixel in den Speicher zu schreiben

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target`, `type` oder `format` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` kleiner als Null oder größer als ld(max) ist, wobei `max` der Rückgabewert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexImageRaw()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_WIDTH`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_HEIGHT`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_BORDER`

`gl.GetTexLevelParameter()` mit dem Argument `#GL_TEXTURE_COMPONENTS`

`gl.Get()` mit dem Argumente `#GL_PACK_ALIGNMENT` und andere

6.73 gl.GetTexLevelParameter

BEZEICHNUNG

gl.GetTexLevelParameter – gibt Texturparameterwerte für einen bestimmten Detaillierungsgrad zurück

ÜBERSICHT

```
param = gl.GetTexLevelParameter(target, level, pname)
```

BESCHREIBUNG

`gl.GetTexLevelParameter()` gibt Texturparameterwerte für einen bestimmten Detailstufenwert zurück, der als `level` angegeben ist. `target` definiert die Zieltextur, entweder `#GL_TEXTURE_1D`, `#GL_TEXTURE_2D`, `#GL_PROXY_TEXTURE_1D` oder `#GL_PROXY_TEXTURE_2D`.

`#GL_MAX_TEXTURE_SIZE` ist nicht wirklich beschreibend genug. Sie muss das größte quadratische Texturbild melden, das mit Mipmaps und Rändern aufgenommen werden kann, aber eine lange dünne Textur oder eine Textur ohne Mipmaps und Ränder kann leicht in den Texturspeicher passen. Die Proxy-Ziele ermöglichen es dem Benutzer, genauer abzufragen, ob GL eine Textur einer bestimmten Konfiguration aufnehmen kann. Wenn die Textur nicht untergebracht werden kann, werden die Texturzustandsvariablen, die mit `gl.GetTexLevelParameter()` abgefragt werden können, auf 0 gesetzt. Die Option Textur kann angepasst werden, die Werte für den Texturzustand werden so gesetzt, wie sie für ein Nicht-Proxy-Ziel gesetzt würden.

`pname` gibt den Texturparameter an, dessen Wert oder Werte zurückgegeben werden. Die akzeptierten Parameternamen lauten wie folgt:

`#GL_TEXTURE_WIDTH`

Der Parameter gibt einen Einzelwert zurück: Die Breite des Texturbildes. Dieser Wert beinhaltet den Rand des Texturbildes. Der Anfangswert ist 0.

`#GL_TEXTURE_HEIGHT`
: Der Parameter gibt einen Einzelwert zurück: Die Höhe des Texturbildes. Dieser Wert beinhaltet den Rand des Texturbildes. Der Anfangswert ist 0.

`#GL_TEXTURE_DEPTH`
: Der Parameter gibt einen Einzelwert zurück: Die Tiefe des Texturbildes. Dieser Wert beinhaltet den Rand des Texturbildes. Der Anfangswert ist 0.

`#GL_TEXTURE_INTERNAL_FORMAT`
: Der Parameter gibt einen Einzelwert mit dem internen Format des Texturbildes zurück.

`#GL_TEXTURE_BORDER`
: Der Parameter gibt einen Einzelwert zurück: Die Breite in Pixel des Randes des Texturbildes. Der Anfangswert ist 0.

`#GL_TEXTURE_XXX_SIZE`
: Die interne Speicherauflösung einer einzelnen Komponente (XXX kann RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, DEPTH sein). Die vom GL gewählte Auflösung entspricht weitgehend der vom Benutzer geforderten Auflösung mit dem Komponentenargument `gl.TexImage1D()`, `gl.TexImage2D()` und `gl.CopyTexImage()`. Der Anfangswert ist 0.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt den symbolischen Namen der Zieltextur an, entweder `#GL_TEXTURE_1D`, `#GL_TEXTURE_2D`, `#GL_PROXY_TEXTURE_1D` oder `#GL_PROXY_TEXTURE_2D`

`level` gibt die Detaillierungsstufe des gewünschten Bildes an; Level 0 ist die Basisbildebene; Level n ist das n-te Mipmap-Reduktionsbild

`pname` gibt den symbolischen Namen eines Texturparameters an (siehe oben für mögliche Werte)

RÜCKGABEWERTE

`param` angeforderte Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird generiert, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` kann erzeugt werden, wenn `level` größer als ld(max) ist, wobei max der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexLevelParameter()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.74 gl.GetTexParameter

BEZEICHNUNG

gl.GetTexParameter – gibt Texturparameterwerte zurück

ÜBERSICHT

```
param = gl.GetTexParameter(target, pname)
```

BESCHREIBUNG

`gl.GetTexParameter()` gibt den Wert oder die Werte des Texturparameters zurück, der als `pname` angegeben ist. `target` definiert die Zieltextur, entweder `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D`, um eine ein- oder zweidimensionale Texturierung festzulegen. `pname` akzeptiert die gleichen symbolischen Konstanten wie `gl.TexParameter()` mit den gleichen Interpretationen:

`#GL_TEXTURE_MAG_FILTER`
: Liefert den Einzel-Wert-Texturvergrößerungsfilter: Eine symbolische Konstante. Der Anfangswert ist `#GL_LINEAR`.

`#GL_TEXTURE_MIN_FILTER`
: Liefert den Einzel-Wert-Textur-Minifikationsfilter: Eine symbolische Konstante. Der Anfangswert ist `#GL_NEAREST_MIPMAP_LINEAR`.

`#GL_TEXTURE_WRAP_S`
: Liefert die Einzel-Wert-Wrapping-Funktion für Texturkoordinaten s: Eine symbolische Konstante. Der Anfangswert ist `#GL_REPEAT`.

`#GL_TEXTURE_WRAP_T`
: Liefert die Einzel-Wert-Wrapping-Funktion für die Texturkoordinate t: Eine symbolische Konstante. Der Anfangswert ist `#GL_REPEAT`.

`#GL_TEXTURE_BORDER_COLOR`
: Liefert vier Gleitkommazahlen, die die RGBA-Farbe des Texturrandes umfassen. Der Anfangswert ist (0, 0, 0, 0).

`#GL_TEXTURE_PRIORITY`
: Liefert die Residenzpriorität der Zieltextur (oder der daran gebundenen Textur). Der Anfangswert ist 1. Siehe Abschnitt 6.115 [gl.PrioritizeTextures], Seite 178, für Details.

`#GL_TEXTURE_RESIDENT`
: Liefert den Residenzstatus der Zieltextur. Wenn der in params zurückgegebene Wert `#GL_TRUE` ist, wird die Textur im Texturspeicher gespeichert. Siehe Abschnitt 6.3 [gl.AreTexturesResident], Seite 26, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt den symbolischen Namen der Zieltextur an; `#GL_TEXTURE_1D` und `#GL_TEXTURE_2D` werden akzeptiert

`pname` gibt den symbolischen Namen eines Texturparameters an (siehe oben für unterstützte Werte)

RÜCKGABEWERTE

`param` angeforderte Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.GetTexParameter()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.75 gl.Hint

BEZEICHNUNG

gl.Hint – setzt implementierungsspezifische Hinweise

ÜBERSICHT

```
gl.Hint(target, mode)
```

BESCHREIBUNG

Bestimmte Aspekte des GL-Verhaltens können mit Hinweisen gesteuert werden, wenn es Interpretationsspielraum gibt. Ein Hinweis wird mit zwei Argumenten angegeben. `target` ist eine symbolische Konstante, die das zu kontrollierende Verhalten angibt und `mode` ist eine weitere symbolische Konstante, die das gewünschte Verhalten angibt. Der Initialwert für jedes Ziel ist `#GL_DONT_CARE`. `mode` kann eine der folgenden symbolischen Konstanten sein:

`#GL_FASTEST`
: Es sollte die effizienteste Option gewählt werden.

`#GL_NICEST`
: Es sollte die korrekteste oder qualitativ hochwertigste Option gewählt werden.

`#GL_DONT_CARE`
: Keine Einstellung.

Obwohl die Implementierungsaspekte, die angedeutet werden können, gut definiert sind, hängt die Interpretation der Hinweise von der Implementierung ab. Die Hinweisaspekte, die mit `target` angegeben werden können sowie die vorgeschlagene Semantik sind wie folgt:

`#GL_FOG_HINT`
: Zeigt die Genauigkeit der Nebelberechnung an. Wenn die Nebelberechnung pro Pixel von der GL-Implementierung nicht effizient unterstützt wird, kann der Hinweis auf `#GL_DONT_CARE` oder `#GL_FASTEST` zu einer Berechnung der Nebeleffekte pro Scheitelpunkt führen.

`#GL_LINE_SMOOTH_HINT`
: Zeigt die Abtastqualität von Antialiasing-Linien an. Wenn eine größere Filterfunktion angewendet wird, kann der Hinweis `#GL_NICEST` dazu führen, dass während der Rasterung mehr Pixelfragmente erzeugt werden.

`#GL_PERSPECTIVE_CORRECTION_HINT`
: Zeigt die Qualität der Interpolation von Farbe, Texturkoordinate und Nebelkoordinate an. Wenn die perspektivkorrigierte Parameterinterpolation von der GL-Implementierung nicht effizient unterstützt wird, kann der Hinweis `#GL_DONT_CARE` oder `#GL_FASTEST` zu einer einfachen linearen Interpolation von Farben und/oder Texturkoordinaten führen.

`#GL_POINT_SMOOTH_HINT`
: Zeigt die Qualität von Antialias-Punkten an. Wenn eine größere Filterfunktion angewendet wird, kann der Hinweis `#GL_NICEST` dazu führen, dass während der Rasterung mehr Pixelfragmente erzeugt werden.

`#GL_POLYGON_SMOOTH_HINT`
: Zeigt die Abtastqualität von Antialiasing-Polygonen an. Der Hinweis `#GL_NICEST` kann dazu führen, dass bei der Rasterung mehr Pixelfragmente erzeugt werden, wenn eine größere Filterfunktion angewendet wird.

Die Interpretation von Hinweisen hängt von der Implementierung ab. Einige Implementierungen ignorieren die Einstellungen von `gl.Hint()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` ist eine symbolische Konstante, die das zu kontrollierende Verhalten angibt (siehe oben für mögliche Werte)

`mode` gibt eine symbolische Konstante an, die das gewünschte Verhalten angibt; `#GL_FASTEST`, `#GL_NICEST` und `#GL_DONT_CARE` werden akzeptiert

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` oder `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Hint()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.76 gl.Index

BEZEICHNUNG

gl.Index – stellt den aktuellen Farbindex ein

ÜBERSICHT

```
gl.Index(c)
```

BESCHREIBUNG

`gl.Index()` aktualisiert den aktuellen (ein Wert) Farbindex. Er braucht ein Argument, der neue Werte für den aktuellen Farbindex benötigt.

Der aktuelle Index wird als Fließkommawert gespeichert. Der Anfangswert ist 1.

Indexwerte außerhalb des darstellbaren Bereichs des Farbindexpuffers werden nicht festgelegt. Bevor ein Index jedoch gedithert (falls aktiviert) und in den Rahmenpuffer geschrieben wird, wird er in das Festkomma-Format umgewandelt. Alle Bits im ganzzahligen Teil des resultierenden Festkommawertes, die nicht den Bits im Rahmenpuffer entsprechen, werden ausgeblendet.

Der aktuelle Index kann jederzeit aktualisiert werden. Insbesondere kann `gl.Index()` zwischen einem Aufruf von `gl.Begin ()` und `gl.End ()` aufgerufen werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`c` gibt den neuen Wert für den aktuellen Farbindex an

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_INDEX`

6.77 gl.IndexMask

BEZEICHNUNG

gl.IndexMask – steuert das Schreiben einzelner Bits in den Farbindexpuffer

ÜBERSICHT

```
gl.IndexMask(mask)
```

BESCHREIBUNG

`gl.IndexMask()` steuert das Schreiben einzelner Bits in den Farbindexpuffer. Die niederwertigsten n Bits der Maske, wobei n die Anzahl der Bits in einem Farbindexpuffer ist, geben eine Maske an. Wenn eine 1 (eins) in der Maske erscheint, ist es möglich, in das entsprechende Bit im Farbindexpuffer (oder in die Puffer) zu schreiben. Erscheint eine 0 (Null), ist das entsprechende Bit schreibgeschützt.

Diese Maske wird nur im Farbindexmodus verwendet und betrifft nur die aktuell zum Schreiben ausgewählten Puffer. Siehe Abschnitt 6.34 [gl.DrawBuffer], Seite 64, für Details.. Zunächst sind alle Bits zum Schreiben freigegeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mask` gibt eine Bitmaske an, um das Schreiben einzelner Bits in die Farbindexpuffer zu aktivieren und zu deaktivieren; zunächst hat die Maske nur Einsen (alle Bits können geändert werden).

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.IndexMask()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_INDEX_WRITEMASK`

6.78 gl.IndexPointer

BEZEICHNUNG

gl.IndexPointer – definiert ein Feld von Farbindizes

ÜBERSICHT

```
gl.IndexPointer(indexArray)
```

BESCHREIBUNG

`gl.IndexPointer()` gibt ein Feld von Farbindizes an, die beim Rendern verwendet werden sollen. `indexArray` muss ein Feld sein, das eine Anzahl von Gleitkommawerten enthält, die Farbindizes beschreiben.

Wenn Sie `Nil` in `indexArray` übergeben, wird der Inhalt des Farbindex-Feld-Puffer gelöscht, aber er wird nicht aus OpenGL entfernt. Dies muss manuell erfolgen, z.B. durch Deaktivierung des Farbindex-Feldes oder durch Definition eines neuen Feldes.

Wenn ein Farbindex-Feld angegeben wird, wird es zusätzlich zur aktuellen Scheitelpunkt-Feld-Pufferobjektbindung als klientseitiger Status gespeichert.

Um das Farbindex-Feld zu aktivieren und zu deaktivieren, rufen Sie den Befehl `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_INDEX_ARRAY` auf. Wenn aktiviert, wird das Farbindex-Feld verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()`, oder `gl.ArrayElement()` aufgerufen wird.

Farbindizes werden für verschachtelte Scheitelpunkt-Feld-Formate nicht unterstützt. Siehe Abschnitt 6.80 [gl.InterleavedArrays], Seite 130, für Details.

Das Farbindex-Feld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.IndexPointer()` ist zwischen `gl.Begin()` und `gl.End()` nicht erlaubt. Es kann ein Fehler auftreten oder aber auch nicht. Wenn kein Fehler erzeugt wird, ist der Vorgang undefiniert.

`gl.IndexPointer()` wird typischerweise auf der Klient-Seite implementiert.

Farbindex-Feld-Parameter sind klientseitig und werden daher nicht gespeichert oder durch `gl.PushAttrib()` und `gl.PopAttrib()` wiederhergestellt. Benutzen Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`indexArray`
: Feld von Farbindizes oder `Nil` (siehe oben)

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_INDEX_ARRAY`

`gl.Get()` mit dem Argument `#GL_INDEX_ARRAY_TYPE`

`gl.Get()` mit dem Argument `#GL_INDEX_ARRAY_STRIDE`

`gl.GetPointer()` mit dem Argument `#GL_INDEX_ARRAY_POINTER`

6.79 gl.InitNames

BEZEICHNUNG

gl.InitNames – initialisiert den Namenstapel

ÜBERSICHT

`gl.InitNames()`

BESCHREIBUNG

Der Namensstapel wird im Auswahlmodus verwendet, um eine eindeutige Identifizierung von Renderingbefehlen zu ermöglichen. Er besteht aus einem geordneten Satz vorzeichenloser Ganzzahlen. `gl.InitNames()` bewirkt, dass der Namensstapel in seinen leeren Standardzustand initialisiert wird.

Der Namenstapel ist immer leer, wenn der Rendermodus nicht `#GL_SELECT` ist. Der Aufruf von `gl.InitNames()`, wenn der Rendermodus nicht `#GL_SELECT` ist, wird ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.InitNames()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_NAME_STACK_DEPTH`

6.80 gl.InterleavedArrays

BEZEICHNUNG

gl.InterleavedArrays – gibt verschachtelte Felder an und aktiviert sie gleichzeitig

ÜBERSICHT

```
gl.InterleavedArrays(format, stride, data)
```

BESCHREIBUNG

`gl.InterleavedArrays()` ermöglicht es Ihnen, individuelle Farb-, Normal-, Textur- und Scheitelpunkt-Felder anzugeben und zu aktivieren, deren Elemente Teil eines größeren Aggregat-Feld-Element hat. Bei einigen Implementierungen ist dies effizienter als die separate Angabe der Felder.

`data` muss ein Zeiger auf einen Rohdatenspeicher sein, der von dem Hollywood-Befehl `AllocMem()` zugewiesen wird und die relevanten Felddaten enthält. Verwenden Sie den Hollywood-Befehl `GetMemPointer()`, um die Rohzeiger-Adresse von Speicherblöcken zu erhalten, die von `AllocMem()` zugewiesen wurden. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Wenn `stride` gleich 0 ist, werden die Aggregatelemente nacheinander gespeichert. Andernfalls treten `stride` Bytes zwischen dem Anfang eines Aggregat-Feld-Elements und dem Anfang des nächsten Aggregat-Feld-Elements auf.

`format` dient als Schlüssel, der die Extraktion einzelner Felder aus dem Aggregat-Feld beschreibt. Wenn `format` ein T enthält, dann werden Texturkoordinaten aus dem verschachtelten Feld extrahiert. Wenn ein C vorhanden ist, werden Farbwerte extrahiert. Wenn ein N vorhanden ist, werden normale Koordinaten extrahiert. Scheitelpunkt-Koordinaten werden immer extrahiert. Die Ziffern 2, 3 und 4 geben an, wie viele Werte extrahiert werden. F zeigt an, dass Werte als Fließkommazahlen

extrahiert werden. Farben können auch als 4 unsignierte Bytes extrahiert werden, wenn 4UB dem C folgt. Wenn eine Farbe als 4 unsignierte Bytes extrahiert wird, befindet sich das nachfolgende Scheitelpunkt-Feld-Element an der ersten möglichen Fließkomma-Adresse. Die folgenden symbolischen Konstanten werden für `format` erkannt:

```
#GL_V2F
#GL_V3F
#GL_C4UB_V2F
#GL_C4UB_V3F
#GL_C3F_V3F
#GL_N3F_V3F
#GL_C4F_N3F_V3F
#GL_T2F_V3F
#GL_T4F_V4F
#GL_T2F_C4UB_V3F
#GL_T2F_C3F_V3F
#GL_T2F_N3F_V3F
#GL_T2F_C4F_N3F_V3F
#GL_T4F_C4F_N3F_V4F
```

Wenn `gl.InterleavedArrays()` beim kompilieren einer Display-Liste aufgerufen wird, wird sie nicht in die Liste kompiliert, sondern sofort ausgeführt.

Die Ausführung von `gl.InterleavedArrays()` ist zwischen `gl.Begin()` und `gl.End()` nicht erlaubt. Es kann ein Fehler auftreten oder aber auch nicht. Wenn kein Fehler erzeugt wird, ist der Vorgang undefiniert.

`gl.InterleavedArrays()` wird typischerweise auf der Klient-Seite implementiert.

Scheitelpunkt-Feld-Parameter sind klientseitig und werden daher nicht durch `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt. Verwenden Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`format` gibt den Typ des zu aktivierenden Feldes an (siehe oben für unterstützte Formate)

`stride` gibt den Versatz in Bytes zwischen den einzelnen Aggregat-Feld-Elementen an

`data` Rohspeicherzeiger mit Daten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` kein akzeptierter Wert ist.

6.81 gl.IsEnabled

BEZEICHNUNG

gl.IsEnabled – prüft, ob eine OpenGL-Fähigkeit aktiviert ist

ÜBERSICHT

```
bool = gl.IsEnabled(cap)
```

BESCHREIBUNG

`gl.IsEnabled()` gibt `#GL_TRUE` zurück, wenn `cap` eine aktivierte Fähigkeit, ndernfalls `#GL_FALSE`. Zunächst sind alle Fähigkeiten außer `#GL_DITHER` deaktiviert; `#GL_DITHER` ist zunächst aktiviert.

Die folgenden Fähigkeiten werden für `cap` akzeptiert:

`#GL_ALPHA_TEST`
: Siehe Abschnitt 6.2 [gl.AlphaFunc], Seite 25, für Details.

`#GL_AUTO_NORMAL`
: Siehe Abschnitt 6.44 [gl.EvalCoord], Seite 76, für Details.

`#GL_BLEND`
: Siehe Abschnitt 6.8 [gl.BlendFunc], Seite 33, für Details. Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_CLIP_PLANEi`
: Siehe Abschnitt 6.17 [gl.ClipPlane], Seite 41, für Details.

`#GL_COLOR_ARRAY`
: Siehe Abschnitt 6.21 [gl.ColorPointer], Seite 44, für Details.

`#GL_COLOR_LOGIC_OP`
: Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_COLOR_MATERIAL`
: Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

`#GL_CULL_FACE`
: Siehe Abschnitt 6.25 [gl.CullFace], Seite 52, für Details.

`#GL_DEPTH_TEST`
: Siehe Abschnitt 6.28 [gl.DepthFunc], Seite 54, für Details. Siehe Abschnitt 6.30 [gl.DepthRange], Seite 55, für Details.

`#GL_DITHER`
: Siehe Abschnitt 6.40 [gl.Enable], Seite 74, für Details.

`#GL_EDGE_FLAG_ARRAY`
: Siehe Abschnitt 6.39 [gl.EdgeFlagPointer], Seite 73, für Details.

`#GL_FOG` Siehe Abschnitt 6.50 [gl.Fog], Seite 83, für Details.

`#GL_INDEX_ARRAY`
: Siehe Abschnitt 6.78 [gl.IndexPointer], Seite 128, für Details.

`#GL_INDEX_LOGIC_OP`
: Siehe Abschnitt 6.92 [gl.LogicOp], Seite 145, für Details.

`#GL_LIGHTi`
: Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details.

`#GL_LIGHTING`
: Siehe Abschnitt 6.95 [gl.Material], Seite 152, für Details. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details. Siehe Abschnitt 6.84 [gl.Light], Seite 136, für Details.

`#GL_LINE_SMOOTH`
: Siehe Abschnitt 6.87 [gl.LineWidth], Seite 141, für Details.

`#GL_LINE_STIPPLE`
: Siehe Abschnitt 6.86 [gl.LineStipple], Seite 140, für Details.

`#GL_MAP1_COLOR_4`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_INDEX`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_NORMAL`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_1`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_2`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_3`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP1_TEXTURE_COORD_4`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_COLOR_4`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_INDEX`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_NORMAL`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_1`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_2`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_3`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_TEXTURE_COORD_4`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_3`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_MAP2_VERTEX_4`
: Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details.

`#GL_NORMAL_ARRAY`
: Siehe Abschnitt 6.100 [gl.NormalPointer], Seite 158, für Details.

`#GL_NORMALIZE`
: Siehe Abschnitt 6.99 [gl.Normal], Seite 157, für Details.

`#GL_POINT_SMOOTH`
: Siehe Abschnitt 6.107 [gl.PointSize], Seite 170, für Details.

`#GL_POLYGON_SMOOTH`
: Siehe Abschnitt 6.108 [gl.PolygonMode], Seite 172, für Details.

`#GL_POLYGON_OFFSET_FILL`
: Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_OFFSET_LINE`
: Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_OFFSET_POINT`
: Siehe Abschnitt 6.109 [gl.PolygonOffset], Seite 173, für Details.

`#GL_POLYGON_STIPPLE`
: Siehe Abschnitt 6.110 [gl.PolygonStipple], Seite 174, für Details.

`#GL_RESCALE_NORMAL`
: Siehe Abschnitt 6.99 [gl.Normal], Seite 157, für Details.

`#GL_SCISSOR_TEST`
: Siehe Abschnitt 6.128 [gl.Scissor], Seite 197, für Details.

`#GL_STENCIL_TEST`
: Siehe Abschnitt 6.131 [gl.StencilFunc], Seite 201, für Details. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details.

`#GL_TEXTURE_1D`
: Siehe Abschnitt 6.139 [gl.TexImage1D], Seite 210, für Details.

`#GL_TEXTURE_2D`
: Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für Details.

`#GL_TEXTURE_COORD_ARRAY`
: Siehe Abschnitt 6.135 [gl.TexCoordPointer], Seite 205, für Details.

`#GL_TEXTURE_GEN_Q`
: Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_R`
: Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_S`
: Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_TEXTURE_GEN_T`
: Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details.

`#GL_VERTEX_ARRAY`
: Siehe Abschnitt 6.147 [gl.VertexPointer], Seite 226, für Details.

Wenn ein Fehler generiert wird, gibt `gl.IsEnabled()` 0 zurück.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`cap` gibt eine symbolische Konstante an, die eine GL-Fähigkeit anzeigt

RÜCKGABEWERTE

`bool` `#GL_TRUE` oder `#GL_FALSE`

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `cap` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.IsEnabled()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.82 gl.IsList

BEZEICHNUNG

gl.IsList – bestimmt, ob ein Name mit einer Display-Liste übereinstimmt

ÜBERSICHT

```
bool = gl.IsList(list)
```

BESCHREIBUNG

`gl.IsList()` gibt `#GL_TRUE` zurück, wenn `list` der Name einer Display-Liste ist, `#GL_FALSE`, wenn nicht oder wenn ein Fehler auftritt.

Ein Name, der von `gl.GenLists()` zurückgegeben wird, aber noch nicht mit einer Display-Liste mit dem Befehl `gl.NewList()` verknüpft wurde, ist nicht der Name einer Display-Liste.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`list` gibt einen Namen für eine potentielle Display-Liste an

RÜCKGABEWERTE

`bool` `#GL_TRUE` oder `#GL_FALSE`

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.IsList()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.83 gl.IsTexture

BEZEICHNUNG

gl.IsTexture – bestimmt, ob ein Name einer Textur entspricht

ÜBERSICHT

```
bool = gl.IsTexture(texture)
```

BESCHREIBUNG

`gl.IsTexture()` gibt `#GL_TRUE` zurück, wenn `texture` aktuell der Name einer Textur ist. Wenn `texture` Null ist oder ein Wert ungleich Null ist, der derzeit nicht der Name einer Textur ist sowie wenn ein Fehler auftritt, gibt `gl.IsTexture()` `#GL_FALSE` zurück.

Ein Name, der von `gl.GenTextures()` zurückgegeben wird, aber noch nicht mit dem Befehl `gl.BindTexture()` mit einer Textur verknüpft wurde, ist nicht der Name einer Textur.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`texture` gibt einen Wert an, der der Name einer Textur sein kann

RÜCKGABEWERTE

`bool` `#GL_TRUE` oder `#GL_FALSE`

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.IsTexture()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.84 gl.Light

BEZEICHNUNG

gl.Light – stellt die Lichtquellenparameter ein

ÜBERSICHT

```
gl.Light(light, pname, param)
```

BESCHREIBUNG

`gl.Light()` setzt die Werte der einzelnen Lichtquellenparameter. `light` bezeichnet das Licht und ist ein symbolischer Name der Form `#GL_LIGHTi`, wobei `i` von 0 bis zum Wert von `#GL_MAX_LIGHTS` - 1 reicht. `pname` gibt einen von zehn Lichtquellenparametern an, wiederum durch einen symbolischen Namen. `param` ist entweder ein einzelner Gleitkommawert oder eine Tabelle, die mehrere Gleitkommawerte enthält. Dies hängt vom Parameter `pname` ab.

Um die Lichtberechnung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` auf mit dem Argument `#GL_LIGHTING`. Die Beleuchtung ist zunächst deaktiviert. Wenn sie aktiviert ist, tragen aktivierte Lichtquellen zur Lichtberechnung bei. Die Lichtquelle i wird mit `gl.Enable()` und `gl.Disable()` zusammen mit dem Argument `#GL_LIGHTi` aktiviert und deaktiviert.

Die zehn Lichtparameter sind wie folgt:

`#GL_AMBIENT`

: `param` muss vier Gleitkommawerte enthalten, die die RGBA-Umgebungsintensität des Lichts angeben. Die anfängliche Umgebungshelligkeit ist (0, 0, 0, 1).

`#GL_DIFFUSE`

: `param` muss vier Gleitkommawerte enthalten, die die diffuse RGBA-Intensität des Lichts angeben. Der Anfangswert für `#GL_LIGHT0` ist (1, 1, 1, 1); für andere Lichtquellen ist der Anfangswert (0, 0, 0, 1).

`#GL_SPECULAR`

`param` muss vier Gleitkommawerte enthalten, die die RGBA-Spiegelintensität des Lichts angeben. Der Anfangswert für `#GL_LIGHT0` ist (1, 1, 1, 1); für andere Lichtquellen ist der Anfangswert (0, 0, 0, 1).

`#GL_POSITION`

`param` muss vier Gleitkommawerte enthalten, die die Position des Lichts in homogenen Objektkoordinaten angeben. Die Position wird durch die Modellsichtmatrix transformiert, wenn `gl.Light()` aufgerufen wird (als wäre es ein Punkt), wird sie in Augenkoordinaten gespeichert. Wenn die w-Komponente der Position 0 ist, wird das Licht als gerichtete Quelle behandelt. Bei der Berechnung von diffuser und spiegelnder Beleuchtung wird die Richtung des Lichts, nicht aber seine tatsächliche Position berücksichtigt und die Dämpfung ist deaktiviert. Andernfalls basieren die Berechnungen der diffusen und spiegelnden Beleuchtung auf der tatsächlichen Position des Lichts in den Augenkoordinaten und die Dämpfung ist aktiviert. Die Ausgangsposition ist (0, 0, 1, 0); somit ist die Ausgangslichtquelle ausgerichtet, parallel zur und in Richtung der -z-Achse.

`#GL_SPOT_DIRECTION`

`param` muss drei Gleitkommawerte enthalten, die die Richtung des Lichts in homogenen Objektkoordinaten angeben. Die Spotrichtung wird beim Aufruf von `gl.Light()` durch die oberen 3x3 der Modellsichtmatrix transformiert und in Augenkoordinaten gespeichert. Es ist nur dann von Bedeutung, wenn `#GL_SPOT_CUTOFF` nicht 180 ist, was es ursprünglich ist. Die Anfangsrichtung ist (0, 0, -1).

`#GL_SPOT_EXPONENT`

`param` muss ein einzelner Gleitkommawert sein, der die Intensitätsverteilung des Lichts angibt. Es werden nur Werte im Bereich (0, 128) akzeptiert. Die effektive Lichtintensität wird durch den Kosinus des Winkels zwischen der Richtung des Lichts und der Richtung vom Licht zum beleuchteten Scheitelpunkt gedämpft, erhöht um die Kraft des Punkt-Exponenten. Höhere Punkt-Exponenten führen somit zu einer fokussierteren Lichtquelle, unabhängig vom Punktabschaltwinkel (siehe `#GL_SPOT_CUTOFF`, nächster Absatz). Der anfängliche Punkt-Exponent ist 0, was zu einer gleichmäßigen Lichtverteilung führt.

`#GL_SPOT_CUTOFF`

`param` muss ein einzelner Gleitkommawert sein, der den maximalen Streuwinkel einer Lichtquelle angibt. Es werden nur Werte im Bereich (0, 90) und der Sonderwert 180 akzeptiert. Wenn der Winkel zwischen der Richtung des Lichts und der Richtung vom Licht zum beleuchteten Scheitelpunkt größer ist als der Punktabschaltungswinkel, ist das Licht vollständig verdeckt. Ansonsten wird seine Intensität durch den Punkt-Exponenten und die Dämpfungsfaktoren gesteuert. Die anfängliche Punktabschaltung beträgt 180, was zu einer gleichmäßigen Lichtverteilung führt.

`#GL_CONSTANT_ATTENUATION`
: `param` muss ein einzelner Gleitkommawert sein, der einen der drei Lichtdämpfungsfaktoren angibt. Es werden nur positive Werte akzeptiert. Wenn das Licht nicht ausgerichtet, sondern positionell ist, wird seine Intensität durch den Kehrwert der Summe aus dem konstanten Faktor, dem linearen Faktor mal dem Abstand zwischen dem Licht und dem beleuchteten Scheitelpunkt und dem quadratischen Faktor mal dem Quadrat des gleichen Abstands abgeschwächt. Die anfänglichen Dämpfungsfaktoren sind (1, 0, 0), was zu keiner Dämpfung führt.

`#GL_LINEAR_ATTENUATION`
: Siehe Dokumentation von `#GL_CONSTANT_ATTENUATION` oben.

`#GL_QUADRATIC_ATTENUATION`
: Siehe Dokumentation von `#GL_CONSTANT_ATTENUATION` oben.

Es ist immer der Fall, dass `#GL_LIGHTi` = `#GL_LIGHT0 +` i ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`light` gibt eine Lichtquelle an (siehe oben)

`pname` gibt einen einzelwertigen Lichtquellenparameter fürs Licht an (siehe oben)

`param` einen einzelnen Gleitkommawert oder eine Tabelle mit mehreren Gleitkommawerten (abhängig vom Parameter `pname`, siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird generiert, wenn entweder `light` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn ein Punkt-Exponentenwert außerhalb des Bereichs (0, 128) angegeben ist, oder wenn eine Punktabschaltung außerhalb des Bereichs (0, 90) (außer dem Sonderwert 180) angegeben ist, oder wenn ein negativer Dämpfungsfaktor angegeben ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Light()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetLight()`

`gl.IsEnabled()` mit dem Argument `#GL_LIGHTING`

6.85 gl.LightModel

BEZEICHNUNG

gl.LightModel – stellt den Parameter des Beleuchtungsmodells ein

ÜBERSICHT

```
gl.LightModel(pname, param)
```

BESCHREIBUNG

`gl.LightModel()` setzt den Parameter des Beleuchtungsmodells. `pname` benennt einen Parameter und `param` gibt den neuen Wert an. Es gibt drei Parameter für das Beleuchtungsmodell:

`#GL_LIGHT_MODEL_AMBIENT`
: `param` muss vier Gleitkommawerte enthalten, die die Umgebungs-RGBA-Intensität der gesamten Szene angeben. Die anfängliche Intensität der Umgebungsszene ist (0.2, 0.2, 0.2, 1.0).

`#GL_LIGHT_MODEL_LOCAL_VIEWER`
: `param` muss ein einzelner Gleitkommawert sein, der angibt, wie spiegelnde Reflexionswinkel berechnet werden. Wenn `param` gleich 0 ist, nehmen spiegelnde Reflexionswinkel die Blickrichtung parallel zur und in Richtung der -z Achse an, unabhängig von der Position des Scheitelpunktes in Augenkoordinaten. Andernfalls werden Spiegelreflexionen aus dem Ursprung des Augenkoordinatensystems berechnet. Der Anfangswert ist 0.

`#GL_LIGHT_MODEL_TWO_SIDE`
: `param` muss ein einzelner Gleitkommawert sein, der angibt, ob ein- oder zweiseitige Lichtberechnungen für Polygone durchgeführt werden. Sie hat keinen Einfluss auf die Lichtberechnungen für Punkte, Linien oder Bitmaps. Wenn `param` gleich 0 ist, wird eine einseitige Beleuchtung angegeben und es werden nur die Materialparameter der Vorderseite in der Beleuchtungsgleichung verwendet. Andernfalls ist eine zweiseitige Beleuchtung vorgesehen. In diesem Fall werden die Scheitelpunkte von nach hinten gerichteten Polygonen mit den Parametern des nach hinten gerichteten Materials beleuchtet und ihre Normalen umgekehrt, bevor die Beleuchtungsgleichung ausgewertet wird. Eckpunkte von nach vorne gerichteten Polygonen sind immer unter Verwendung der vorderen Materialparameter beleuchtet, ohne Änderung ihrer Normalen. Der Anfangswert ist 0.

Im RGBA-Modus ist die beleuchtete Farbe eines Scheitelpunktes die Summe aus der Materialemissionsintensität, dem Produkt aus dem Materialumgebungsreflexionsgrad und der Vollbildumgebungsintensität des Beleuchtungsmodells sowie dem Beitrag jeder aktivierten Lichtquelle. Jede Lichtquelle trägt die Summe von drei Teilen bei: Ambient, diffus und specular. Der Beitrag der Umgebungslichtquelle ergibt sich aus dem Produkt aus dem Material Umgebungsreflexionsgrad und der Umgebungsintensität des Lichts. Der Beitrag der diffusen Lichtquelle ist das Produkt aus der diffusen Material-Reflexion, der diffusen Intensität des Lichts und dem Punktprodukt der Normalität des Scheitels mit dem normierten Vektor vom Scheitelpunkt zur Lichtquelle. Der Beitrag der spiegelnden Lichtquelle ist das Produkt aus dem Materialspiegelreflexionsgrad, der Spiegelintensität des Lichts und dem Punktprodukt der normierten Scheitel-Augen- und Scheitel-Licht-Vektoren, womit die Kraft des Glanzes des Materials erhöht werden. Alle drei Lichtquellenbeiträge werden gleichmäßig gedämpft, basierend auf dem Abstand vom Scheitelpunkt zur Lichtquelle und der Lichtquellenrichtung, dem Spreizexponenten und dem Spreizwinkel. Alle Punktprodukte werden durch 0 ersetzt, wenn sie zu einem negativen Wert berechnet werden.

Die Alpha-Komponente der resultierenden leuchtenden Farbe wird auf den Alpha-Wert des Materials diffuser Reflexionsgrad eingestellt.

Im Farbindexmodus reicht der Wert des beleuchteten Index eines Scheitels von der Umgebung bis zu den an `gl.Material()` übergebenen Spiegelungswerten mit `#GL_COLOR_INDEXES`. Diffuse und spiegelnde Koeffizienten, berechnet mit einer (.30, .59, .11) Gewichtung der Lichtfarben, des Glanzes des Materials und den gleichen Reflexions- und Dämpfungsgleichungen wie im RGBA-Fall, bestimmen, wie weit über der Umgebung der resultierende Index liegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt einen einzelwertigen Lichtmodellparameter an (siehe oben)

`param` einen einzelnen Gleitkommawert oder eine Tabelle mit mehreren Gleitkommawerten (abhängig vom Parameter `pname`, siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LightModel()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_LIGHT_MODEL_AMBIENT`

`gl.Get()` mit dem Argument `#GL_LIGHT_MODEL_LOCAL_VIEWER`

`gl.Get()` mit dem Argument `#GL_LIGHT_MODEL_TWO_SIDE`

`gl.IsEnabled()` mit dem Argument `#GL_LIGHTING`

6.86 gl.LineStipple

BEZEICHNUNG

gl.LineStipple – setzt das Linienrastermuster

ÜBERSICHT

```
gl.LineStipple(factor, pattern)
```

BESCHREIBUNG

Das Punktieren von Linien maskiert bestimmte Fragmente, die durch die Rasterung entstehen; diese Fragmente werden nicht gezeichnet. Die Maskierung wird durch die Verwendung von drei Parametern erreicht: Dem 16-Bit-Zeilen-Punktmuster `pattern`, dem Wiederholungszahlfaktor `factor` und einem ganzzahligen Rasterzähler s.

Der Zähler s wird auf 0 zurückgesetzt, wenn `gl.Begin()` aufgerufen wird und vor jedem Liniensegment wird eine

```
gl.Begin(#GL_LINES)
gl.End()
```

Sequenz erzeugt. Sie wird erhöht, nachdem jedes Fragment eines aliasierten Liniensegments mit Einheitsbreite erzeugt wurde oder nachdem jedes i-Fragment eines i-Breiten-

Liniensegments erzeugt wurde. Die mit der Anzahl s verbundenen i-Fragmente werden ausgeblendet, wenn

```
pattern bit s factor % 16
```

0 ist, ansonsten werden diese Fragmente an den Rahmenpuffer gesendet. Bit-Null des Musters ist das niederwertigste Bit.

Antialiasing-Linien werden als eine Folge von 1*breiten Rechtecken zum Zwecke des Punktierens behandelt. Ob das Rechteck s gerastert ist oder nicht, hängt von der für aliasierte Linien beschriebenen Fragmentregel ab, die Rechtecke statt Gruppen von Fragmenten zählt.

Um die Zeilenzuordnung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` auf, mit dem Argument `#GL_LINE_STIPPLE`. Wenn aktiviert, wird das Linienpunktemuster wie oben beschrieben angewendet. Wenn deaktiviert, ist es, als ob alle Muster auf 1 gesetzt wären. Zunächst ist die Linienpunktierung deaktiviert.

Alternativ können Sie auch eine Zeichenkette aus 16 Zeichen übergeben, die in `pattern` entweder 0 oder 1 ist, z.B. `"1111000011110000"`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`factor` gibt einen Multiplikator für jedes Bit im Zeilen-Punktmuster an; wenn der Faktor beispielsweise 3 ist, wird jedes Bit im Muster dreimal verwendet, bevor das nächste Bit im Muster verwendet wird; der Faktor wird auf den Bereich [1, 256] festgelegt und ist standardmäßig auf 1 eingestellt

`pattern` gibt eine 16-Bit Ganzzahl an, deren Bitmuster bestimmt, welche Fragmente einer Linie gezeichnet werden, wenn die Linie gerastert wird; Bit Null wird zuerst verwendet; Das Standardmuster besteht nur aus Einsen

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LineStipple()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_LINE_STIPPLE_PATTERN`

`gl.Get()` mit dem Argument `#GL_LINE_STIPPLE_REPEAT`

`gl.IsEnabled()` mit dem Argument `#GL_LINE_STIPPLE`

6.87 gl.LineWidth

BEZEICHNUNG

gl.LineWidth – setzt die Breite der gerasterten Linien

ÜBERSICHT

`gl.LineWidth(width)`

BESCHREIBUNG

`gl.LineWidth()` gibt die gerasterte Breite sowohl von aliasierten als auch von antialisierten Linien an. Die Verwendung einer anderen Linienbreite als 1 hat unterschiedliche Auswirkungen, je nachdem, ob die Linien-Antialiasing-Funktion aktiviert ist. Um die Linienantialiasing zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()`

mit dem Argument #GL_LINE_SMOOTH auf. Die Linienantialiasing-Funktion ist zunächst deaktiviert.

Wenn die Linienantialiasing-Funktion deaktiviert ist, wird die tatsächliche Breite durch Rundung der gelieferten Breite auf die nächste Ganzzahl bestimmt (wenn die Rundung zum Wert 0 führt, ist es, als ob die Linienbreite 1 wäre). Wenn delta x >= delta y ist, werden in jeder gerasterten Spalte i Pixel gefüllt, wobei i der gerundete Wert der Breite ist. Andernfalls werden in jeder Zeile, die gerastert wird, i Pixel gefüllt.

Wenn Antialiasing aktiviert ist, erzeugt die Linienrasterung für jedes Pixelquadrat ein Fragment, das den innerhalb des Rechtecks liegenden Bereich schneidet. Die Breite ist gleich der aktuellen Linienbreite, dessen Länge gleich der tatsächlichen Länge der Linie ist und das auf das mathematische Liniensegment zentriert ist. Der Abdeckungswert für jedes Fragment ist der Fensterkoordinatenbereich des Schnittpunktes des rechteckigen Bereichs mit dem entsprechenden Pixelquadrat. Dieser Wert wird gespeichert und im letzten Schritt der Rasterung verwendet.

Nicht alle Breiten können unterstützt werden, wenn die Linienantialiasing-Funktion aktiviert ist. Wenn eine nicht unterstützte Breite gewünscht wird, wird die nächstgelegene unterstützte Breite verwendet. Nur die Breite 1 wird garantiert unterstützt, andere sind von der Implementierung abhängig. Ebenso gibt es einen Bereich für aliasierte Linienbreiten. Um den Bereich der unterstützten Breiten und die Größendifferenz zwischen den unterstützten Breiten innerhalb des Bereichs abzufragen, rufen Sie den Befehl gl.Get() mit dem Argumente #GL_LINE_WIDTH_RANGE und #GL_LINE_WIDTH_GRANULARITY auf.

Die durch gl.LineWidth() angegebene Linienbreite wird immer zurückgegeben, wenn #GL_LINE_WIDTH abgefragt wird. Das Festlegen und Runden für aliasierte und antialiatisierte Linien hat keinen Einfluss auf den vorgegebenen Wert.

Die nicht-antialisierte Linienbreite kann auf ein implementierungsabhängiges Maximum festgelegt werden. Obwohl dieses Maximum nicht abgefragt werden kann, darf es nicht kleiner sein als der Maximalwert für antialiatisierte Linien, gerundet auf den nächsten ganzzahligen Wert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`width` gibt die Breite der gerasterten Linien an; der Anfangswert ist 1

FEHLER

#GL_INVALID_VALUE wird erzeugt, wenn `width` kleiner oder gleich 0 ist.

#GL_INVALID_OPERATION wird erzeugt, wenn gl.LineWidth() zwischen gl.Begin() und gl.End() ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

gl.Get() mit dem Argument #GL_LINE_WIDTH

gl.Get() mit dem Argument #GL_LINE_WIDTH_RANGE

gl.Get() mit dem Argument #GL_LINE_WIDTH_GRANULARITY

gl.IsEnabled() mit dem Argument #GL_LINE_SMOOTH

6.88 gl.ListBase

BEZEICHNUNG

gl.ListBase – setzt die Basis der Display-Listen für gl.CallLists()

ÜBERSICHT

`gl.ListBase(base)`

BESCHREIBUNG

`gl.CallLists()` gibt ein Feld von Versätzen an. Displaylistennamen werden erzeugt, indem zu jedem Versatz eine Basis hinzugefügt wird. Namen, die auf gültige Display-Listen verweisen, werden ausgeführt; die anderen werden ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`base` gibt einen ganzzahligen Versatz an, der zu den Versätzen von `gl.CallLists()` hinzugefügt wird, um Displaylistennamen zu erzeugen; der Anfangswert ist 0

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ListBase()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_LIST_BASE`

6.89 gl.LoadIdentity

BEZEICHNUNG

gl.LoadIdentity – ersetzt die aktuelle Matrix durch die Identitätsmatrix

ÜBERSICHT

`gl.LoadIdentity()`

BESCHREIBUNG

gl.LoadIdentity) ersetzt die aktuelle Matrix durch die Identitätsmatrix. Es ist semantisch äquivalent zum Aufruf von `gl.LoadMatrix()` mit der Identitätsmatrix, aber in einigen Fällen ist es effizienter.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LoadIdentity()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.90 gl.LoadMatrix

BEZEICHNUNG

gl.LoadMatrix – ersetzt die aktuelle Matrix durch die angegebene Matrix

ÜBERSICHT

```
gl.LoadMatrix(mArray)
```

BESCHREIBUNG

`gl.LoadMatrix()` ersetzt die aktuelle Matrix durch diejenige, deren Elemente in `mArray` angegeben sind. Die aktuelle Matrix ist die Projektionsmatrix, Modellansichtsmatrix oder Texturmatrix, abhängig vom aktuellen Matrixmodus. Siehe Abschnitt 6.96 [gl.MatrixMode], Seite 154, für Details.. `MArray` muss seine Werte in der Reihenfolge der Hauptspalten speichern.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mArray` gibt ein Feld mit 16 aufeinanderfolgenden Werten an, die als Elemente einer 4*4 Hauptspalten-Matrix verwendet werden

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LoadMatrix()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.91 gl.LoadName

BEZEICHNUNG

gl.LoadName – lädet einen Namen in den Namensstapel

ÜBERSICHT

```
gl.LoadName(name)
```

BESCHREIBUNG

Der Namensstapel wird im Auswahlmodus verwendet, um eine eindeutige Identifizierung von Renderingbefehlen zu ermöglichen. Sie besteht aus einem geordneten Satz von vorzeichenlosen Ganzzahlen und ist zunächst leer.

`gl.LoadName()` bewirkt, dass `name` den Wert oben im Namensstapel ersetzt.

Der Namensstapel ist immer leer, wenn der Rendermodus nicht `#GL_SELECT` ist. Aufrufe von `gl.LoadName()`, wenn der Rendermodus nicht `#GL_SELECT` ist, werden ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`name` gibt einen Namen an, der den obersten Wert auf dem Namensstapel ersetzt

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LoadName()` aufgerufen wird, während der Namensstapel leer ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LoadName()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_NAME_STACK_DEPTH`

6.92 gl.LogicOp

BEZEICHNUNG

gl.LogicOp – setzt die logische Pixeloperation für die Farbindexwiedergabe

ÜBERSICHT

```
gl.LogicOp(opcode)
```

BESCHREIBUNG

`gl.LogicOp()` gibt eine logische Operation an, die, wenn sie aktiviert ist, zwischen dem eingehenden Farbindex oder der RGBA-Farbe und dem Farbindex oder der RGBA-Farbe an der entsprechenden Stelle im Rahmenpuffer angewendet wird. Um die logische Verknüpfung zu aktivieren oder zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` über die symbolische Konstante `#GL_COLOR_LOGIC_OP` auf für den RGBA-Modus oder `#GL_INDEX_LOGIC_OP` für den Farbindex-Modus. Der Anfangswert ist für beide Vorgänge deaktiviert.

`#GL_CLEAR`
0

`#GL_SET` 1

`#GL_COPY` s

`#GL_COPY_INVERTED`
~s

`#GL_NOOP` d

`#GL_INVERT`
~d

`#GL_AND` s & d

`#GL_NAND` ~(s & d)

`#GL_OR` s | d

`#GL_NOR` ~(s | d)

`#GL_XOR` s ^ d

`#GL_EQUIV`
~(s ^ d)

`#GL_AND_REVERSE`
s & ~d

`#GL_AND_INVERTED`
~s & d

`#GL_OR_REVERSE`
s | ~d

`#GL_OR_INVERTED`
~s | d

`opcode` ist eine symbolische Konstante, die aus der obigen Liste ausgewählt wurde. In der Erklärung der logischen Operationen stellt `s` den eingehenden Farbindex und `d` den Index im Rahmenpuffer dar. Es werden Standard-C-Sprachoperatoren verwendet. Wie diese bitweisen Operatoren vermuten lassen, wird die logische Operation unabhängig voneinander auf jedes Bitpaar der Quell- und Zielindizes oder Farben angewendet.

Logische Operationen mit dem Farbindex werden immer unterstützt. RGBA logische Operationen werden nur unterstützt, wenn die GL-Version 1.1 oder höher ist.

Wenn mehr als ein RGBA-Farbpuffer oder Indexpuffer zum Zeichnen aktiviert ist, werden logische Operationen für jeden aktivierten Puffer separat ausgeführt, wobei für den Zielwert der Inhalt dieses Puffers verwendet wird. Siehe Abschnitt 6.34 [gl.DrawBuffer], Seite 64, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`opcode` gibt eine symbolische Konstante ein, die eine logische Operation auswählt (siehe oben für unterstützte Konstanten)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `opcode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.LogicOp()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_LOGIC_OP_MODE`.

`gl.IsEnabled()` mit dem Argument `#GL_COLOR_LOGIC_OP` oder `#GL_INDEX_LOGIC_OP`.

6.93 gl.Map

BEZEICHNUNG

gl.Map – definiert einen ein- oder zweidimensionalen Evaluator

ÜBERSICHT

```
gl.Map(target, u1, u2, pointsArray)
gl.Map(target, u1, u2, v1, v2, pointsArray)
```

BESCHREIBUNG

Evaluatoren bieten eine Möglichkeit, polynome oder rationale polynome Abbildungen zu verwenden, um Knoten, Normalen, Texturkoordinaten und Farben zu erzeugen. Die von einem Evaluator erzeugten Werte werden in weitere Stufen der GL-Verarbeitung geschickt, als ob sie mit `gl.Vertex()`, `gl.Normal()`, `gl.TexCoord()` und `gl.Color()` Befehle präsentiert worden wären, mit der Ausnahme, dass die erzeugten Werte nicht die aktuelle Normale, Texturkoordinaten oder Farben aktualisieren.

Alle polynomialen oder rationalen Polynom-Splines beliebigen Grades (bis zu dem von der GL-Implementierung unterstützten maximalen Grad) können mit Hilfe von Evaluatoren beschrieben werden. Dazu gehören fast alle Splines, die in der Computergrafik verwendet werden: B-Splines, Bézier-Kurve , Hermite-Splines und so weiter.

`gl.Map()` wird verwendet, um die Basis zu definieren und festzulegen, welche Art von Werten erzeugt werden. Einmal definiert, kann eine Karte durch Aufruf von `gl.Enable()` und `gl.Disable()` mit dem Kartennamen aktiviert und deaktiviert werden, einem der neun vordefinierten Werte für das Ziel, die nachfolgend beschrieben werden. `gl.EvalCoord()` wertet die eindimensionalen Karten aus, die aktiviert sind. Wenn `gl.EvalCoord()` stellt einen Wert u oder Werte u und v dar, die Bernstein-Funktionen werden mit u^ und v^ ausgewertet, wobei

```
u^ = (u - u1) / (u2 - u1)
v^ = (v - v1) / (v2 - v1)
```

ist.

`target` ist eine symbolische Konstante, die angibt, welche Art von Kontrollpunkten in `pointsArray` bereitgestellt werden und welche Ergebnisse bei der Auswertung der Karte erzeugt werden. Im eindimensionalen Modus kann es einen der folgenden neun vordefinierten Werte annehmen:

`#GL_MAP1_VERTEX_3`
: Jeder Kontrollpunkt besteht aus drei Fließkommawerten, die x, y und z darstellen. Interne `gl.Vertex()` Befehle werden bei der Auswertung der Karte generiert.

`#GL_MAP1_VERTEX_4`
: Jeder Kontrollpunkt besteht aus vier Gleitkommawerten, die x, y, z und w darstellen. Interne `gl.Vertex()` Befehle werden bei der Auswertung der Karte generiert.

`#GL_MAP1_INDEX`
: Jeder Kontrollpunkt ist ein einzelner Gleitkommawert, der einen Farbindex darstellt. Interne `gl.Index()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber der aktuelle Index wird nicht mit dem Wert dieses `gl.Index()` Befehls aktualisiert.

`#GL_MAP1_COLOR_4`
: Jeder Kontrollpunkt besteht aus vier Fließkommawerten, die Rot, Grün, Blau und Alpha darstellen. Interne `gl.Color()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuelle Farbe wird nicht mit dem Wert dieses `gl.Color()` Befehls aktualisiert.

`#GL_MAP1_NORMAL`
: Jeder Kontrollpunkt besteht aus drei Gleitkommawerten, die die x-, y- und z-Komponenten eines Normalen-Vektors darstellen. Interne `gl.Normal()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuelle Normale wird nicht mit dem Wert dieses `gl.Normal()` Befehls aktualisiert.

`#GL_MAP1_TEXTURE_COORD_1`
: Jeder Kontrollpunkt ist ein einzelner Gleitkommawert, der die Texturkoordinate s darstellt. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP1_TEXTURE_COORD_2`
: Jeder Kontrollpunkt besteht aus zwei Gleitkommawerten, die die Texturkoordinaten s und t darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP1_TEXTURE_COORD_3`
: Jeder Kontrollpunkt besteht aus drei Fließkommawerten, die die Texturkoordinaten s, t und r darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP1_TEXTURE_COORD_4`
: Jeder Kontrollpunkt besteht aus vier Gleitkommawerten, die die Texturkoordinaten s, t, r und q darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

Im zweidimensionalen Modus werden die folgenden vordefinierten Konstanten unterstützt:

`#GL_MAP2_VERTEX_3`
: Jeder Kontrollpunkt besteht aus drei Fließkommawerten, die x, y und z darstellen. Interne `gl.Vertex()` Befehle werden bei der Auswertung der Karte generiert.

`#GL_MAP2_VERTEX_4`
: Jeder Kontrollpunkt besteht aus vier Gleitkommawerten, die x, y, z und w darstellen. Interne `gl.Vertex()` Befehle werden bei der Auswertung der Karte generiert.

`#GL_MAP2_INDEX`
: Jeder Kontrollpunkt ist ein einzelner Gleitkommawert, der einen Farbindex darstellt. Interne `gl.Index()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber der aktuelle Index wird nicht mit dem Wert dieses `gl.Index()` Befehls aktualisiert.

`#GL_MAP2_COLOR_4`
: Jeder Kontrollpunkt besteht aus vier Fließkommawerten, die Rot, Grün, Blau und Alpha darstellen. Interne `gl.Color()` Befehle werden generiert,

wenn die Karte ausgewertet wird, aber die aktuelle Farbe wird nicht mit dem Wert dieses `gl.Color()` Befehls aktualisiert.

`#GL_MAP2_NORMAL`
: Jeder Kontrollpunkt besteht aus drei Gleitkommawerten, die die x-, y- und z-Komponenten eines Normalen-Vektors darstellen. Interne `gl.Normal()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuelle Normale wird nicht mit dem Wert dieses `gl.Normal()` Befehls aktualisiert.

`#GL_MAP2_TEXTURE_COORD_1`
: Jeder Kontrollpunkt ist ein einzelner Gleitkommawert, der die Texturkoordinate s darstellt. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP2_TEXTURE_COORD_2`
: Jeder Kontrollpunkt besteht aus zwei Gleitkommawerten, die die Texturkoordinaten s und t darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP2_TEXTURE_COORD_3`
: Jeder Kontrollpunkt besteht aus drei Fließkommawerten, die die Texturkoordinaten s, t und r darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

`#GL_MAP2_TEXTURE_COORD_4`
: Jeder Kontrollpunkt besteht aus vier Gleitkommawerten, die die Texturkoordinaten s, t, r und q darstellen. Interne `gl.TexCoord()` Befehle werden generiert, wenn die Karte ausgewertet wird, aber die aktuellen Texturkoordinaten werden nicht mit dem Wert dieses `gl.TexCoord()` Befehls aktualisiert.

Zunächst ist `#GL_AUTO_NORMAL` aktiviert. Wenn `#GL_AUTO_NORMAL` aktiviert ist, werden Normalen-Vektoren erzeugt, wenn entweder `#GL_MAP2_VERTEX_3` oder `#GL_MAP2_VERTEX_4` zum Erzeugen von Knoten verwendet wird.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt die Art der Werte an, die vom Evaluator erzeugt werden (siehe oben)

`u1` gibt eine lineare Zuordnung von u an, wie sie in `gl.EvalCoord()` dargestellt wird, zu u^, der Variablen, die durch die in diesem Befehl angegebenen Gleichungen bewertet wird

`u2` gibt eine lineare Zuordnung von u an, wie sie in `gl.EvalCoord()` dargestellt wird, zu u^, der Variablen, die durch die in diesem Befehl angegebenen Gleichungen bewertet wird

`v1` gibt eine lineare Zuordnung von v an, wie sie in `gl.EvalCoord()` dargestellt wird, zu v^, einer der beiden Variablen, die durch die in diesem Befehl angegebenen Gleichungen bewertet werden

`v2` gibt eine lineare Zuordnung von v an, wie sie in `gl.EvalCoord()` dargestellt wird, zu v^, einer der beiden Variablen, die durch die in diesem Befehl angegebenen Gleichungen bewertet werden

`pointsArray`
gibt eine Tabelle mit einer Anzahl von Kontrollpunkten an (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `u1` gleich `u2` ist, oder wenn `v1` gleich `v2` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Map()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetMap()`

`gl.Get()` mit dem Argument `#GL_MAX_EVAL_ORDER`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_VERTEX_3`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_VERTEX_4`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_INDEX`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_COLOR_4`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_NORMAL`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_TEXTURE_COORD_1`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_TEXTURE_COORD_2`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_TEXTURE_COORD_3`

`gl.IsEnabled()` mit dem Argument `#GL_MAP1_TEXTURE_COORD_4`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_VERTEX_3`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_VERTEX_4`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_INDEX`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_COLOR_4`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_NORMAL`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_TEXTURE_COORD_1`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_TEXTURE_COORD_2`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_TEXTURE_COORD_3`

`gl.IsEnabled()` mit dem Argument `#GL_MAP2_TEXTURE_COORD_4`

6.94 gl.MapGrid

BEZEICHNUNG

gl.MapGrid – definiert ein ein- oder zweidimensionales Gitter

ÜBERSICHT

```
gl.MapGrid(un, u1, u2[, vn, v1, v2])
```

BESCHREIBUNG

`gl.MapGrid()` und `gl.EvalMesh()` werden zusammen verwendet, um eine Reihe von gleichmäßig verteilten Werten für die Kartendomäne effizient zu erzeugen und auszuwerten. `gl.EvalMesh()` durchläuft die ganzzahlige Domäne eines ein- oder zweidimensionalen Gitters, dessen Bereich die Domäne der durch `gl.Map()` spezifizierten Bewertungskarten ist.

`gl.MapGrid()` gibt die linearen Gitterzuordnungen zwischen den ganzzahligen Rasterkoordinaten i (oder i und j) und den Kartenkoordinaten u (oder u und v) der Gleitkommaauswertung an. Siehe Abschnitt 6.93 [gl.Map], Seite 146, für Details, wie u- und v-Koordinaten ausgewertet werden.

Im eindimensionalen Modus spezifiziert `gl.MapGrid()` eine einzelne lineare Zuordnung, so dass die ganzzahlige Gitterkoordinate 0 genau auf u1 abbildet und die ganzzahlige Gitterkoordinate un genau auf u2 abbildet. Alle anderen ganzzahligen Gitterkoordinaten i werden so abgebildet, dass

```
u = i(u2 - u1) / un + u1
```

ist.

Im zweidimensionalen Modus gibt `gl.MapGrid()` zwei solche linearen Zuordnungen an. Eine bildet die ganzzahlige Gitterkoordinate i = 0 genau auf u1 ab und die ganzzahlige Gitterkoordinate i = un genau auf u2. Die andere bildet die ganzzahlige Gitterkoordinate j = 0 genau auf v1 und die ganzzahlige Gitterkoordinate j = vn genau auf v2 ab, die anderen ganzzahligen Gitterkoordinaten i und j werden so abgebildet, dass

```
u = i(u2 - u1) / un + u1
v = j(v2 - v1) / vn + v1
```

ist.

Die von gl.MapGrid () angegebenen Zuordnungen werden von `gl.EvalMesh()` und `gl.EvalPoint()` identisch verwendet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`un`	gibt die Anzahl der Partitionen im Rasterbereichsintervall[u1, u2] an; muss positiv sein
`u1`	gibt die Zuordnungen für ganzzahlige Rasterdomänenwerte i = 0 und i = un an
`u2`	gibt die Zuordnungen für ganzzahlige Rasterdomänenwerte i = 0 und i = un an
`vn`	optional: Gibt die Anzahl der Partitionen im Rasterbereichsintervall[v1, v2] an
`v1`	optional: Gibt die Zuordnungen für ganzzahlige Rasterdomänenwerte j = 0 und j = vn an
`v2`	optional: Gibt die Zuordnungen für ganzzahlige Rasterdomänenwerte j = 0 und j = vn an

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `un` oder `vn` nicht positiv ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.MapGrid()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_DOMAIN`

`gl.Get()` mit dem Argument `#GL_MAP1_GRID_SEGMENTS`

`gl.Get()` mit dem Argument `#GL_MAP2_GRID_SEGMENTS`

6.95 gl.Material

BEZEICHNUNG

gl.Material – setzt Materialparameter für das Beleuchtungsmodell

ÜBERSICHT

```
gl.Material(face, pname, param)
```

BESCHREIBUNG

`gl.Material()` weist den Materialparametern Werte zu. Es gibt zwei aufeinander abgestimmte Einstellungen von Materialparametern. Eine, die nach vorne gerichtete, wird verwendet, um Punkte, Linien, Bitmaps und alle Polygone (wenn die zweiseitige Beleuchtung deaktiviert ist) oder nur nach vorne gerichtete Polygone (wenn die zweiseitige Beleuchtung aktiviert ist) zu schattieren. Die andere Einstellung, nach hinten gerichtet, wird nur dann verwendet, wenn eine zweiseitige Beleuchtung aktiviert ist, um nach hinten gerichtete Polygone zu ermöglichen. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für Details zur ein- und zweiseitigen Lichtberechnung.

`gl.Material()` nimmt drei Argumente an. Das erste, `face`, gibt an, ob die `#GL_FRONT` Materialien, die `#GL_BACK` Materialien oder beide `#GL_FRONT_AND_BACK` Materialien modifiziert werden. Der zweite, `pname`, gibt an, welcher von mehreren Parametern in einer oder beiden Einstellungen geändert wird. Der dritte, `param`, gibt an, welcher Wert oder welche Werte dem angegebenen Parameter zugewiesen werden. Es kann ein einzelner Gleitkommawert oder eine Tabelle mit mehreren Gleitkommawerten sein.

Materialparameter werden in der Beleuchtungsgleichung verwendet, die optional auf jeden Knoten angewendet wird. Die Gleichung wird auf der Referenzseite `gl.LightModel()` erläutert. Die Parameter, die mit `gl.Material()` angegeben werden können und ihre Interpretation durch die Lichtgleichung sind wie folgt:

`#GL_AMBIENT`

> `param` muss eine Tabelle mit vier Gleitkommawerten sein, die den Umgebungs-RGBA-Reflexionsgrad des Materials angeben. Die anfängliche Umgebungsreflexion für sowohl nach vorne als auch nach hinten gerichtete Materialien beträgt (0.2, 0.2, 0.2, 1.0).

`#GL_DIFFUSE`

> `param` muss eine Tabelle mit vier Gleitkommawerten sein, die den diffusen RGBA-Reflexionsgrad des Materials angeben. Die anfängliche diffuse Reflexion für sowohl vordere als auch hintere Materialien ist (0.8, 0.8, 0.8, 1.0).

`#GL_SPECULAR`
: `param` muss eine Tabelle mit vier Gleitkommawerten sein, die den spiegelnden RGBA-Reflexionsgrad des Materials angeben. Die anfängliche Spiegelreflexion für sowohl vordere als auch hintere Materialien ist (0, 0, 0, 1).

`#GL_EMISSION`
: `param` muss eine Tabelle mit vier Gleitkommawerten sein, die die RGBA-emittierte Lichtintensität des Materials angeben. Die anfängliche Emissionsintensität für sowohl nach vorne als auch nach hinten gerichtete Materialien beträgt (0, 0, 0, 1).

`#GL_SHININESS`
: `param` muss ein einzelner Gleitkommawert sein, der den RGBA-Spiegelexponenten des Materials angibt. Es werden nur Werte im Bereich (0,128) akzeptiert. Der anfängliche Spiegelexponent für sowohl nach vorne als auch nach hinten gerichtete Materialien ist 0.

`#GL_AMBIENT_AND_DIFFUSE`
: Entspricht dem zweimaligen Aufruf von `gl.Material()` mit den gleichen Parameterwerten, einmal mit `#GL_AMBIENT` und einmal mit `#GL_DIFFUSE`.

`#GL_COLOR_INDEXES`
: `param` muss eine Tabelle mit drei Gleitkommawerten sein, die die Farbindizes für Umgebungs-, Diffus- und Spiegelbeleuchtung angibt. Diese drei Werte und `#GL_SHINESS` sind die einzigen Materialwerte, die von der Beleuchtungsgleichung im Farbindexmodus verwendet werden. Siehe Abschnitt 6.85 [gl.LightModel], Seite 138, für eine Behandlung über Farbindexbeleuchtung.

Die Materialparameter können jederzeit aktualisiert werden. Insbesondere kann `gl.Material()` zwischen einem Aufruf von `gl.Begin()` und `gl.End()` aufgerufen werden. Soll jedoch nur ein einziger Materialparameter pro Knoten geändert werden, so ist `gl.ColorMaterial()` bevorzugt gegenüber `gl.Material()` zu wählen. Siehe Abschnitt 6.20 [gl.ColorMaterial], Seite 43, für Details.

Während die Parameter Umgebung-, Diffus-, Spiegel- und Emissionsmaterial alle Alpha-Komponenten aufweisen, wird bei der Lichtberechnung nur die diffuse Alpha-Komponente verwendet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`face` gibt an, welche Fläche(n) aktualisiert werden sollen; muss eine der folgenden sein: `#GL_FRONT`, `#GL_BACK`, oder `#GL_FRONT_AND_BACK`

`pname` gibt den Materialparameter der Fläche(n) an, die aktualisiert wird/werden (siehe oben)

`param` einen Gleitkommawert (oder eine Tabelle mit mehreren Gleitkommawerten), auf den `pname` gesetzt wird

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn entweder `face` oder `pname` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn ein Spiegelexponent außerhalb des Bereichs (0,128) angegeben wird.

VERBUNDENE GET-OPERATIONEN
`gl.GetMaterial()`

6.96 gl.MatrixMode

BEZEICHNUNG
gl.MatrixMode – gibt an, welche Matrix die aktuelle Matrix ist

ÜBERSICHT
`gl.MatrixMode(mode)`

BESCHREIBUNG
`gl.MatrixMode()` setzt den aktuellen Matrixmodus. `mode` kann einen von drei Werten annehmen:

`#GL_MODELVIEW`
: Wendet nachfolgende Matrixoperationen auf den Matrixstapel der Modellansicht an.

`#GL_PROJECTION`
: Wendet nachfolgende Matrixoperationen auf den Projektionsmatrix-Stapel an.

`#GL_TEXTURE`
: Wendet nachfolgende Matrixoperationen auf den Texturmatrix-Stapel an.

Um herauszufinden, welcher Matrixstapel derzeit das Ziel aller Matrixoperationen ist, rufen Sie `gl.Get()` mit dem Argument `#GL_MATRIX_MODE` auf. Der Initialwert ist `#GL_MODELVIEW`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt an, welcher Matrixstapel das Ziel für nachfolgende Matrixoperationen ist (siehe oben)

FEHLER
`#GL_INVALID_ENUM` wird generiert, wenn `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.MatrixMode()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN
`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

6.97 gl.MultMatrix

BEZEICHNUNG

gl.MultMatrix – multipliziert die aktuelle Matrix mit der angegebenen Matrix

ÜBERSICHT

```
gl.MultMatrix(mArray)
```

BESCHREIBUNG

`gl.MultMatrix()` multipliziert die aktuelle Matrix mit der durch `mArray` angegebenen und ersetzt die aktuelle Matrix durch das Produkt.

Die aktuelle Matrix wird durch den aktuellen Matrixmodus bestimmt. Siehe Abschnitt 6.96 [gl.MatrixMode], Seite 154, für Details. Es ist entweder die Projektionsmatrix, die Modellansichtsmatrix oder die Texturmatrix.

Während die Elemente der Matrix mit doppelter Genauigkeit angegeben sind, kann GL diese Werte mit weniger als einer Genauigkeit speichern oder bearbeiten.

In vielen Computersprachen werden 4x4-Felder in zeilenweiser Reihenfolge dargestellt. Die soeben beschriebenen Transformationen stellen diese Matrizen in der Reihenfolge von Spalte zu Spalte dar. Die Reihenfolge der Multiplikation ist wichtig. Wenn beispielsweise die aktuelle Transformation eine Drehung ist und `gl.MultMatrix()` mit einer Übersetzungsmatrix aufgerufen wird, erfolgt die Übersetzung direkt auf den zu transformierenden Koordinaten, während die Drehung auf den Ergebnissen dieser Verschiebung erfolgt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mArray` Tabelle mit 16 aufeinanderfolgenden Werten, die als Elemente einer 4x4-Hauptspalten-Matrix verwendet werden

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn glMultMatrix zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.98 gl.NewList

BEZEICHNUNG

gl.NewList – erstellt oder ersetzt eine Display-Liste

ÜBERSICHT

```
gl.NewList(list, mode)
```

BESCHREIBUNG

Display-Listen sind Gruppen von GL-Befehlen, die zur späteren Ausführung gespeichert wurden. Display-Listen werden mit `gl.NewList()` erstellt. Alle nachfolgenden Befehle werden in der Ausgabeliste in der angegebenen Reihenfolge abgelegt, bis `gl.EndList()` aufgerufen wird.

`gl.NewList()` hat zwei Argumente. Das erste Argument, `list`, ist eine positive ganze Zahl, die zum eindeutigen Namen für die Display-Liste wird. Namen können mit `gl.GenLists()` erstellt und reserviert werden und auf Eindeutigkeit mit `gl.IsList()` getestet werden. Das zweite Argument, `mode`, ist eine symbolische Konstante, die einen von zwei Werten annehmen kann:

`#GL_COMPILE`
: Befehle werden nur kompiliert.

`#GL_COMPILE_AND_EXECUTE`
: Befehle werden ausgeführt, wenn sie in die Display-Liste aufgenommen werden.

Bestimmte Befehle werden nicht in die Display-Liste kompiliert, sondern sofort ausgeführt, unabhängig davon, welcher Display-Listenmodus gerade aktiv ist. Diese Befehle sind `gl.AreTexturesResident()`, `gl.ColorPointer()`, `gl.DeleteLists()`, `gl.DeleteTextures()` und `gl.DisableClientState()`, `gl.EdgeFlagPointer()`, `gl.EnableClientState()` und `gl.FeedbackBuffer()`, `gl.Finish()`, `gl.Flush()`, `gl.GenLists()`, `gl.GenTextures()`, `gl.IndexPointer()`, `gl.InterleavedArrays()`, `gl.IsEnabled()` und auch `gl.IsList()`, `gl.IsTexture()`, `gl.NormalPointer()`, `gl.PopClientAttrib()` und schließlich auch `gl.PixelStore()`, `gl.PushClientAttrib()`, `gl.ReadPixels()`, `gl.RenderMode()`, `gl.SelectBuffer()`, `gl.TexCoordPointer()`, `gl.VertexPointer()` und alle `gl.Get()` Befehle.

Wenn `gl.EndList()` angetroffen wird, wird die Definition der Display-Liste durch die Zuordnung der Liste zur eindeutigen Namensliste (angegeben im Befehl `gl.NewList()`) vervollständigt. Wenn bereits eine Display-Liste mit Namensliste existiert, wird diese nur ersetzt, wenn `gl.EndList()` aufgerufen wird.

`gl.CallList()` und `gl.CallLists()` kann in Display-Listen eingetragen werden. Befehle in der Display-Liste oder Listen, die durch `gl.CallList()` oder `gl.CallLists()` ausgeführt werden sind nicht in der zu erstellenden Display-Liste enthalten, auch wenn der Listenerstellungsmodus `#GL_COMPILE_AND_EXECUTE` ist.

Eine Display-Liste ist nur eine Gruppe von Befehlen und Argumenten, so dass bei der Ausführung der Liste Fehler generiert werden, die durch Befehle in einer Display-Liste erzeugt werden. Wenn die Liste im Modus `#GL_COMPILE` erstellt wird, werden Fehler erst bei der Ausführung der Liste generiert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`list` gibt den Namen der Display-Liste als Ganzzahl an

`mode` gibt den Kompiliermodus an, der für die Kompilierung `#GL_COMPILE` oder `#GL_COMPILE_AND_EXECUTE` sein kann

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `list` 0 ist.

`#GL_INVALID_ENUM` wird generiert, wenn `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.EndList()` ohne vorherige `gl.NewList()` aufgerufen wird, oder wenn `gl.NewList()` aufgerufen wird, während eine Display-Liste definiert wird.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.NewList()` oder `gl.EndList()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

`#GL_OUT_OF_MEMORY` wird erzeugt, wenn nicht genügend Speicherplatz vorhanden ist, um die Display-Liste zu erstellen. Wenn die GL-Version 1.1 oder höher ist, wird keine Änderung am vorherigen Inhalt der Display-Liste vorgenommen, falls vorhanden und es wird keine andere Änderung am GL-Status vorgenommen. (Es ist, als ob kein Versuch unternommen worden wäre, die neue Display-Liste zu erstellen).

VERBUNDENE GET-OPERATIONEN

`gl.IsList()`

`gl.Get()` mit dem Argument `#GL_LIST_INDEX`

`gl.Get()` mit dem Argument `#GL_LIST_MODE`

6.99 gl.Normal

BEZEICHNUNG

gl.Normal – setzt den aktuell gültigen Normale-Vektor

ÜBERSICHT

```
gl.Normal(nx, ny, nz)
```

BESCHREIBUNG

Die aktuelle Normale wird bei Ausgabe von `gl.Normal()` auf die angegebenen Gleitkommakoordinaten gesetzt. Der Anfangswert der aktuellen Normale ist der Einheitenvektor (0, 0, 1). Alternativ kann `gl.Normal()` auch mit einem einzigen Tabellenelement aufgerufen werden, das die Normale Koordinaten x, y, z enthält.

Normale, die mit `gl.Normal()` angegeben werden, müssen keine Einheitenlänge haben. Wenn `#GL_NORMALIZE` aktiviert ist, werden Normale beliebiger Länge, die mit `gl.Normal()` angegeben wurden, nach der Transformation normalisiert. Wenn `#GL_RESCALE_NORMAL` aktiviert ist, werden Normale durch einen Skalierungsfaktor skaliert, der aus der Modelansicht-Matrix abgeleitet wird. `#GL_RESCALE_NORMAL` setzt voraus, dass die ursprünglich angegebenen Normalen von einheitlicher Länge waren und dass die Modellansichtmatrix nur einheitliche Skalen für korrekte Ergebnisse enthält. Um die Normalisierung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` auf, mit entweder `#GL_NORMALIZE` oder `#GL_RESCALE_NORMAL`. Die Normalisierung ist zunächst deaktiviert.

Die aktuelle Normale kann jederzeit aktualisiert werden. Insbesondere kann `gl.Normal()` zwischen einem Aufruf von `gl.Begin()` und von `gl.End()` aufgerufen werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`nx` gibt die x-Koordinate der neuen aktuellen Normalen an

`ny` gibt die y-Koordinate der neuen aktuellen Normalen an

`nz` gibt die z-Koordinate der neuen aktuellen Normalen an

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_NORMAL`

`gl.IsEnabled()` mit dem Argument `#GL_NORMALIZE`

`gl.IsEnabled()` mit dem Argument `#GL_RESCALE_NORMAL`

6.100 gl.NormalPointer

BEZEICHNUNG

gl.NormalPointer – definiert eine Reihe von Normalen

ÜBERSICHT

```
gl.NormalPointer(normalArray[, type])
```

BESCHREIBUNG

`gl.NormalPointer()` gibt ein Feld von Normalen an, die beim Rendern verwendet werden sollen. `normalArray` kann entweder eine eindimensionale Tabelle sein, die aus einer beliebigen Anzahl aufeinanderfolgender Normalen besteht, oder eine zweidimensionale Tabelle, die aus einer beliebigen Anzahl von Untertabellen besteht, die jeweils eine Normale enthält. Jede Normale muss x-, y- und z-Koordinaten enthalten, die als Gleitkommawerte angegeben sind. Wenn `normalArray` eine eindimensionale Tabelle ist, müssen Sie das optionale Argument `type` auf `#GL_TRUE` setzen. Wenn `normalArray` eine zweidimensionale Tabelle ist, kann das optionale Argument `type` weglassen oder auf `#GL_FALSE` gesetzt werden.

Wenn Sie `Nil` in `normalArray` übergeben, wird der Inhalt der Normalen-Feld-Puffer gelöscht, aber er wird nicht aus OpenGL entfernt. Dies muss manuell erfolgen, z.B. durch Deaktivierung des Normalen-Feldes oder durch Definition eines neuen Feldes.

Um das normale Feld zu aktivieren und zu deaktivieren, rufen Sie den Befehl `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_NORMAL_ARRAY` auf. Wenn aktiviert, wird das Normalen-Feld verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()`, oder `gl.ArrayElement()` aufgerufen wird.

Das Normalen-Feld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()`, oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.NormalPointer()` ist zwischen der Ausführung von `gl.Begin()` und `gl.End()` nicht erlaubt. Dabei kann ein Fehler auftreten oder auch nicht. Wenn kein Fehler erzeugt wird, ist der Vorgang undefiniert.

`gl.NormalPointer()` wird typischerweise auf der Klient-Seite implementiert.

Normalen-Feld-Parameter sind Klient-Seitige Zustände und werden daher nicht durch `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt . Verwenden Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`normalArray`
: ein- oder zweidimensionale Tabelle mit Normalenwerten oder `Nil` (siehe oben)

`type`
: optional: `#GL_TRUE` wenn die Tabelle in `normalArray` eine eindimensionale Tabelle ist, sonst `#GL_FALSE` (Voreingestellt ist `#GL_FALSE`)

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_NORMAL_ARRAY`

`gl.Get()` mit dem Argument `#GL_NORMAL_ARRAY_TYPE`

`gl.Get()` mit dem Argument `#GL_NORMAL_ARRAY_STRIDE`

`gl.GetPointer()` mit dem Argument `#GL_NORMAL_ARRAY_POINTER`

6.101 gl.Ortho

BEZEICHNUNG

gl.Ortho – multipliziert die aktuelle Matrix mit einer orthographischen Matrix

ÜBERSICHT

```
gl.Ortho(left, right, bottom, top, zNear, zFar)
```

BESCHREIBUNG

`gl.Ortho()` beschreibt eine Transformation, die eine Parallelprojektion erzeugt. Die aktuelle Matrix (siehe `gl.MatrixMode()` für Details) wird mit dieser Matrix multipliziert und das Ergebnis ersetzt die aktuelle Matrix, als ob `gl.MultMatrix()` mit der folgenden Matrix als Argument aufgerufen worden wäre:

```
A 0 0 tx
0 B 0 ty
0 0 C tz
0 0 0 1
```

wobei

```
A = 2 / (right - left)
B = 2 / (top - bottom)
C = -2 / (far  - near)
tx = -(right + left) / (right  - left)
ty = -(top + bottom) / (top  - bottom)
tz = -(zFar + zNear) / (zFar  - zNear)
```

ist.

Typischerweise ist der Matrixmodus `#GL_PROJECTION` und (`left`, `bottom`, `-zNear`) und (`right`, `top`, `-zNear`) geben die Punkte auf der nahen Ausschnittebene an, die auf die untere linke bzw. obere rechte Ecke des Fensters abgebildet sind, vorausgesetzt, dass sich das Auge bei (0, 0, 0) befindet. `-zFar` gibt die Position der entfernten Ausschnittebene an. Sowohl `zNear` als auch `zFar` können entweder positiv oder negativ sein.

Verwenden Sie `gl.PushMatrix()` und `gl.PopMatrix()` um den aktuellen Matrixstapel zu speichern und wiederherzustellen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`left`	gibt die Koordinate für die linke vertikale Ausschnittebene an
`right`	gibt die Koordinate für die rechte vertikale Ausschnittebene an
`bottom`	gibt die Koordinate für die untere horizontale Ausschnittebene an
`top`	gibt die Koordinate für die obere horizontale Ausschnittebene an
`zNear`	gibt den Abstand zur näheren Tiefenausschnittebene an; dieser Wert ist negativ, wenn die Ebene hinter dem Betrachter liegen soll
`zFar`	gibt den Abstand zur weiter entfernten Tiefenausschnittebene an; dieser Wert ist negativ, wenn die Ebene hinter dem Betrachter liegen soll

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn left = right, oder bottom = top, oder zNear = zFar ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Ortho()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.102 gl.PassThrough

BEZEICHNUNG

gl.PassThrough – setzt eine Markierung in den Feedback-Puffer

ÜBERSICHT

```
gl.PassThrough(token)
```

BESCHREIBUNG

Feedback ist ein GL-Rendermodus. Der Modus wird durch Aufruf von `gl.RenderMode()` mit `#GL_FEEDBACK` ausgewählt. Wenn sich GL im Feedback-Modus befindet, werden durch die Rasterung keine Pixel erzeugt. Stattdessen werden Informationen über Grundmuster, die gerastert worden wären, mit Hilfe von GL an die Anwendung zurückgegeben. Siehe Abschnitt 6.47 [gl.FeedbackBuffer], Seite 80, für eine Beschreibung des Feedback-Puffers und der darin enthaltenen Werte.

`gl.PassThrough()` fügt eine benutzerdefinierte Markierung in den Feedback-Puffer ein, wenn er im Feedback-Modus ausgeführt wird. `token` wird zurückgegeben, als wäre es ein Grundmuster; es wird mit einem eigenen eindeutigen Identifikationswert angezeigt: `#GL_PASS_THROUGH_TOKEN`. Die Reihenfolge der gl.PassThrough()-Befehle in Bezug auf die Angabe von Grafikgrundmustern bleibt erhalten.

`gl.PassThrough()` wird ignoriert, wenn sich GL nicht im Feedback-Modus befindet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`token` gibt einen Markierungswert an, der in den Feedback-Puffer nach einem `#GL_PASS_THROUGH_TOKEN` ausgegeben wird

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PassThrough()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_RENDER_MODE`

6.103 gl.PixelMap

BEZEICHNUNG

gl.PixelMap – richtet Pixel-Transferkarten ein

ÜBERSICHT

```
gl.PixelMap(map, mapsize, valuesArray)
```

BESCHREIBUNG

`gl.PixelMap()` richtet Transfertabellen oder Karten ein, die verwendet werden von `gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.DrawPixels()`, `gl.ReadPixels()`, `gl.TexImage()` und auch `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()` und `gl.TexSubImage2D()`. Die Verwendung dieser Karten ist auf der Referenzseite `gl.PixelTransfer()` vollständig beschrieben und teilweise in den Referenzseiten für die Befehle Pixel und Texturbild. Nur das Angeben der Karten ist auf dieser Referenzseite beschrieben.

`map` ist ein symbolischer Kartenname, der eine von zehn zu erstellenden Karten angibt. `valuesArray` ist eine Tabelle, die ein Feld von Werten für den angegebenen Kartennamen enthält. Die zehn Karten sind wie folgt:

`#GL_PIXEL_MAP_I_TO_I`
: Ordnet Farbindizes Farbindizes zu.

`#GL_PIXEL_MAP_S_TO_S`
: Ordnet Schablonenindizes Schablonenindizes zu.

`#GL_PIXEL_MAP_I_TO_R`
: Ordnet Farbindizes roten Komponenten zu.

`#GL_PIXEL_MAP_I_TO_G`
: Ordnet Farbindizes grünen Komponenten zu.

`#GL_PIXEL_MAP_I_TO_B`
: Ordnet Farbindizes blauen Komponenten zu.

`#GL_PIXEL_MAP_I_TO_A`
: Ordnet Farbindizes Alpha-Komponenten zu.

`#GL_PIXEL_MAP_R_TO_R`
: Ordnet rote Komponenten den roten Komponenten zu.

`#GL_PIXEL_MAP_G_TO_G`
: Ordnet grüne Komponenten den grünen Komponenten zu.

`#GL_PIXEL_MAP_B_TO_B`
: Ordnet blaue Komponenten den blauen Komponenten zu.

`#GL_PIXEL_MAP_A_TO_A`
: Ordnet Alpha-Komponenten Alpha-Komponenten zu.

Die Einträge in einer Karte werden als Gleitkommazahlen angegeben. Karten, die Farbkomponentenwerte speichern (alle mit Ausnahme von `#GL_PIXEL_MAP_I_TO_I` und `#GL_PIXEL_MAP_S_TO_S`), behalten ihre Werte im Gleitkommaformat, mit nicht angegebener Fangvorrichtung und Exponentengrößen. Die durch `gl.PixelMap()` angegebenen Gleitkommawerte werden direkt in das interne Gleitkommaformat dieser Karten konvertiert und dann in den Bereich [0,1] festgelegt.

Karten, die Indizes speichern, `#GL_PIXEL_MAP_I_TO_I` und `#GL_PIXEL_MAP_S_TO_S`, behalten ihre Werte im Festkommaformat, mit einer unbestimmten Anzahl von Bits rechts neben dem Binärpunkt. Gleitkommawerte, die durch `gl.PixelMap()` angegeben werden, werden direkt in das interne Festkommaformat dieser Karten umgewandelt.

Die folgende Tabelle zeigt die anfänglichen Größen und Werte für jede der Karten. Karten, die entweder durch Farb- oder Schablonenindizes gekennzeichnet sind, müssen für einige mapsize = 2^n haben oder die Ergebnisse sind undefiniert. Die maximal zulässige Größe für jede Map hängt von der Implementierung ab und kann durch Aufruf von `gl.Get()` mit dem Argument `#GL_MAX_PIXEL_MAP_TABLE` bestimmt werden. Das einzelne Maximum gilt für alle Karten; es beträgt mindestens 32.

Karte	Lookup-Index	Lookup-Wert	Def Grösse	Def Wert
#GL_PIXEL_MAP_I_TO_I	Farbindex	Farbindex	1	0
#GL_PIXEL_MAP_S_TO_S	Schablonenindex	Schablonenindex	1	0
#GL_PIXEL_MAP_I_TO_R	Farbindex	R	1	0
#GL_PIXEL_MAP_I_TO_G	Farbindex	G	1	0
#GL_PIXEL_MAP_I_TO_B	Farbindex	B	1	0
#GL_PIXEL_MAP_I_TO_A	Farbindex	A	1	0
#GL_PIXEL_MAP_R_TO_R	R	R	1	0
#GL_PIXEL_MAP_G_TO_G	G	G	1	0
#GL_PIXEL_MAP_B_TO_B	B	B	1	0
#GL_PIXEL_MAP_A_TO_A	A	A	1	0

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`map` gibt einen symbolischen Kartennamen an (siehe oben für unterstützte Namen)

`mapsize` gibt die Größe der zu definierenden Karte an.

`valuesArray`
: gibt eine Tabelle an, die ein Feld von Werten enthält

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `map` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn mapsize kleiner als eins oder größer als `#GL_MAX_PIXEL_MAP_TABLE` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `map` `#GL_PIXEL_MAP_I_TO_I`, `#GL_PIXEL_MAP_S_TO_S`, `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B`, oder `#GL_PIXEL_MAP_I_TO_A` und die Kartengröße keine Zweierpotenz ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PixelMap()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetPixelMap()`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_I_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_S_TO_S_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_R_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_G_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_B_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_I_TO_A_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_R_TO_R_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_G_TO_G_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_B_TO_B_SIZE`

`gl.Get()` mit dem Argument `#GL_PIXEL_MAP_A_TO_A_SIZE`

`gl.Get()` mit dem Argument `#GL_MAX_PIXEL_MAP_TABLE`

6.104 gl.PixelStore

BEZEICHNUNG

gl.PixelStore – stellt den Pixelspeichermodi ein

ÜBERSICHT

```
gl.PixelStore(pname, param)
```

BESCHREIBUNG

`gl.PixelStore()` setzt Pixelspeichermodi, die den Betrieb der nachfolgenden `gl.DrawPixels()` und `gl.ReadPixels()` beeinflussen, sowie das Entpacken von Polygon-Punktemustern, Bitmaps und Texturmuster. Siehe Abschnitt 6.110 [gl.PolygonStipple], Seite 174, für Details. Siehe Abschnitt 6.7 [gl.Bitmap], Seite 31, für Details. Siehe Abschnitt 6.138 [gl.TexImage], Seite 209, für Details.

`pname` ist eine symbolische Konstante, die den zu setzenden Parameter angibt und `param` ist der neue Wert. Sechs der zwölf Speicherparameter beeinflussen, wie Pixeldaten an den Klientspeicher zurückgegeben werden. Sie sind wie folgt:

`#GL_PACK_SWAP_BYTES`

Wenn `True`, wird die Byte-Ordnung für Multibyte-Farbkomponenten, Tiefenkomponenten, Farbindizes oder Schablonenindizes umgekehrt. Das heißt, wenn eine 4-Byte-Komponente aus den Bytes b0, b1, b2, b3 besteht, wird sie im Speicher als b3, b2, b1, b0 gespeichert, wenn `#GL_PACK_SWAP_BYTES`

`True` ist. `#GL_PACK_SWAP_BYTES` hat keinen Einfluss auf die Speicherordnung der Komponenten innerhalb eines Pixels, sondern nur auf die Reihenfolge der Bytes innerhalb der Komponenten oder Indizes. So werden beispielsweise die drei Komponenten eines Formats `#GL_RGB` immer mit roten ersten, grünen zweiten und blauen dritten Pixeln gespeichert, unabhängig vom Wert von `#GL_PACK_SWAP_BYTES`.

`#GL_PACK_LSB_FIRST`
: Wenn `True`, werden Bits innerhalb eines Bytes von niederwertig bis höchstwertig angeordnet; andernfalls ist das erste Bit in jedem Byte das bedeutendste. Dieser Parameter ist nur für Bitmap-Daten von Bedeutung.

`#GL_PACK_ROW_LENGTH`
: Wenn größer als 0, definiert `#GL_PACK_ROW_LENGTH` die Anzahl der Pixel in einer Zeile. Wenn das erste Pixel einer Reihe an der Position p im Speicher platziert wird, dann wird die Position des ersten Pixels der nächsten Reihe durch Überspringen einer bestimmten Anzahl von Komponenten oder Indizes erhalten. Siehe in einem OpenGL-Referenzhandbuch für Einzelheiten.

`#GL_PACK_SKIP_PIXELS`
: Dieser Wert wird dem Programmierer zur Verfügung gestellt; er bietet keine Funktionalität, die nicht durch einfaches Erhöhen des an `gl.ReadPixels()` übergebenen Zeigers dupliziert werden kann. Das Setzen von `#GL_PACK_SKIP_PIXELS` auf i entspricht dem Inkrementieren des Zeigers um `in` Komponenten oder Indizes, wobei n die Anzahl der Komponenten oder Indizes in jedem Pixel ist.

`#GL_PACK_SKIP_ROWS`
: Dieser Wert wird dem Programmierer zur Verfügung gestellt; er bietet keine Funktionalität, die nicht durch einfaches Erhöhen des an `gl.ReadPixels()` übergebenen Zeigers dupliziert werden kann. Das Setzen von `#GL_PACK_SKIP_ROWS` auf j entspricht dem Inkrementieren des Zeigers um `jm` Komponenten oder Indizes, wobei m die Anzahl der Komponenten oder Indizes pro Zeile ist, wie sie gerade im Abschnitt `#GL_PACK_ROW_LENGTH` berechnet wurde.

`#GL_PACK_ALIGNMENT`
: Gibt die Ausrichtungsanforderungen für den Anfang jeder Pixelzeile im Speicher an. Die zulässigen Werte sind 1 (Byte-Ausrichtung), 2 (Zeilen, die auf geradzahlige Bytes ausgerichtet sind), 4 (Wort-Ausrichtung) und 8 (Zeilen, die an Doppelwortgrenzen beginnen).

Die anderen sechs der zwölf Speicherparameter beeinflussen, wie Pixeldaten aus dem Klientspeicher gelesen werden. Diese Werte sind dann von Bedeutung für die Befehle `gl.DrawPixels()`, `gl.TexImage()` und weiterhin auch für `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()`, `gl.TexSubImage2D()`, `gl.Bitmap()` und `gl.PolygonStipple()`. Sie sind wie folgt:

`#GL_UNPACK_SWAP_BYTES`
: Wenn `True`, wird die Byte-Ordnung für Multibyte-Farbkomponenten, Tiefenkomponenten, Farbindizes oder Schablonenindizes umgekehrt. Das heißt,

wenn eine 4-Byte-Komponente aus Bytes b0, b1, b2, b3 besteht, wird sie als b3, b2, b1, b1, b0 aus dem Speicher genommen, wenn `#GL_UNPACK_SWAP_BYTES` `True` ist. `#GL_UNPACK_SWAP_BYTES` hat keinen Einfluss auf die Speicherordnung der Komponenten innerhalb eines Pixels, sondern nur auf die Reihenfolge der Bytes innerhalb der Komponenten oder Indizes. So werden beispielsweise die drei Komponenten eines Pixels im Format `#GL_RGB` immer mit rotem ersten, grünen zweiten und blauen dritten Pixel gespeichert, unabhängig vom Wert von `#GL_UNPACK_SWAP_BYTES`.

`#GL_UNPACK_LSB_FIRST`

Wenn `True`, werden Bits innerhalb eines Bytes von niederwertig bis höchstwertig angeordnet; andernfalls ist das erste Bit in jedem Byte das bedeutendste. Dies ist nur für Bitmap-Daten relevant.

`#GL_UNPACK_ROW_LENGTH`

Wenn größer als 0, definiert `#GL_UNPACK_ROW_LENGTH` die Anzahl der Pixel in einer Zeile. Wenn das erste Pixel einer Reihe an der Position p im Speicher platziert wird, dann wird die Position des ersten Pixels der nächsten Reihe durch Überspringen einer bestimmten Anzahl von Komponenten oder Indizes erhalten. Siehe OpenGL-Referenzhandbuch für Einzelheiten.

`#GL_UNPACK_SKIP_PIXELS`

Dieser Wert wird dem Programmierer als Annehmlichkeit zur Verfügung gestellt; er bietet keine Funktionalität, die nicht durch Inkrementieren des übergebenen Zeigers durch `gl.DrawPixels()`, oder mit `gl.TexImage()`, oder auch mit `gl.TexImage1D()` und `gl.TexImage2D()`, zusätzlich auch mit `gl.TexSubImage()`, oder mit `gl.TexSubImage1D()` und `gl.TexSubImage2D()`, `gl.Bitmap()`, oder mit `gl.PolygonStipple()` dupliziert werden kann. Das Setzen von `#GL_UNPACK_SKIP_PIXELS` auf i entspricht dem Inkrementieren des Zeigers um `in` Komponenten oder Indizes, wobei n die Anzahl der Komponenten oder Indizes in jedem Pixel ist.

`#GL_UNPACK_SKIP_ROWS`

Dieser Wert wird dem Programmierer als Annehmlichkeit zur Verfügung gestellt; er bietet keine Funktionalität, die nicht durch Inkrementieren des übergebenen Zeigers durch `gl.DrawPixels()`, oder mit `gl.TexImage()`, oder auch mit `gl.TexImage1D()` und `gl.TexImage2D()`, zusätzlich auch mit `gl.TexSubImage()`, oder mit `gl.TexSubImage1D()` und `gl.TexSubImage2D()`, `gl.Bitmap()`, oder mit `gl.PolygonStipple()` dupliziert werden kann. Das Setzen von `#GL_UNPACK_SKIP_ROWS` auf j entspricht dem Inkrementieren des Zeigers um `jk-Komponenten` oder -Indizes, wobei k die Anzahl der Komponenten oder -Indizes pro Zeile ist, wie gerade im Abschnitt `#GL_UNPACK_ROW_LENGTH` berechnet.

`#GL_UNPACK_ALIGNMENT`

Gibt die Ausrichtungsanforderungen für den Anfang jeder Pixelzeile im Speicher an. Die zulässigen Werte sind 1 (Byte-Ausrichtung), 2 (Zeilen, die auf geradzahlige Bytes ausgerichtet sind), 4 (Wort-Ausrichtung) und 8 (Zeilen, die an Doppelwortgrenzen beginnen).

Die folgende Tabelle enthält den Typ, den Anfangswert und den Bereich der gültigen Werte für jeden Speicherparameter, der mit `gl.PixelStore()` eingestellt werden kann.

```
pname                  |  Type   | Default | Valid Range
-------------------------------------------------------
#GL_PACK_SWAP_BYTES    | boolean | false   | true or false
#GL_PACK_LSB_FIRST     | boolean | false   | true or false
#GL_PACK_ROW_LENGTH    | integer | 0       | [0,oo)
#GL_PACK_SKIP_ROWS     | integer | 0       | [0,oo)
#GL_PACK_SKIP_PIXELS   | integer | 0       | [0,oo)
#GL_PACK_ALIGNMENT     | integer | 4       | 1, 2, 4, or 8
#GL_UNPACK_SWAP_BYTES | boolean | false   | true or false
#GL_UNPACK_LSB_FIRST   | boolean | false   | true or false
#GL_UNPACK_ROW_LENGTH | integer | 0       | [0,oo)
#GL_UNPACK_SKIP_ROWS   | integer | 0       | [0,oo)
#GL_UNPACK_SKIP_PIXELS| integer | 0       | [0,oo)
#GL_UNPACK_ALIGNMENT   | integer | 4       | 1, 2, 4, or 8
```

Die Pixelspeichermodi, die wirksam sind, wenn `gl.DrawPixels()`, `gl.ReadPixels()` oder `gl.TexImage()`, `gl.TexImage1D()` oder `gl.TexImage2D()`, oder `gl.TexSubImage()`, `gl.TexSubImage1D()`, oder auch GL's `gl.TexSubImage2D()` oder `gl.Bitmap()`, oder `gl.PolygonStipple()` in eine Display-Liste gestellt werden, die die Interpretation von Speicherdaten steuert. Die Pixelspeichermodi, die beim Ausführen einer Display-Liste wirksam werden, sind nicht signifikant.

Pixelspeichermodi sind Klient-Zustände und mit `gl.PushClientAttrib()` und `gl.PopClientAttrib()` angestoßen und wiederhergestellt werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname` gibt den symbolischen Namen des zu setzenden Parameters an (siehe oben für mögliche Modi)

`param` gibt den Wert an, auf den `pname` gesetzt ist

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn eine negative Zeilenlänge, Pixel-Sprung oder Zeilen-Sprung-Wert angegeben ist, oder wenn die Ausrichtung von 1, 2, 4 oder 8 verschieden ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PixelStore()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_PACK_SWAP_BYTES`

`gl.Get()` mit dem Argument `#GL_PACK_LSB_FIRST`

`gl.Get()` mit dem Argument `#GL_PACK_ROW_LENGTH`

`gl.Get()` mit dem Argument `#GL_PACK_SKIP_ROWS`

`gl.Get()` mit dem Argument `#GL_PACK_SKIP_PIXELS`

`gl.Get()` mit dem Argument `#GL_PACK_ALIGNMENT`

`gl.Get()` mit dem Argument `#GL_UNPACK_SWAP_BYTES`
`gl.Get()` mit dem Argument `#GL_UNPACK_LSB_FIRST`
`gl.Get()` mit dem Argument `#GL_UNPACK_ROW_LENGTH`
`gl.Get()` mit dem Argument `#GL_UNPACK_SKIP_ROWS`
`gl.Get()` mit dem Argument `#GL_UNPACK_SKIP_PIXELS`
`gl.Get()` mit dem Argument `#GL_UNPACK_ALIGNMENT`

6.105 gl.PixelTransfer

BEZEICHNUNG

gl.PixelTransfer – stellt die Pixelübertragungsmodi ein

ÜBERSICHT

```
gl.PixelTransfer(pname, param)
```

BESCHREIBUNG

`gl.PixelTransfer()` setzt Pixel-Transfer-Modi, die den Betrieb der nachfolgenden Systeme beeinflussen: `gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.DrawPixels()`, `gl.ReadPixels()`, `gl.TexImage()`, `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()` und `gl.TexSubImage2D()`. Die Algorithmen, die durch Pixelübertragungsmodi angegeben werden, arbeiten mit Pixeln, nachdem sie aus dem Rahmenpuffer mit `gl.CopyPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()`, `gl.ReadPixels()` gelesen wurden oder falls sie aus dem Klientspeicher mit `gl.DrawPixels()`, `gl.TexImage()`, `gl.TexImage1D()`, `gl.TexImage2D()`, `gl.TexSubImage()`, `gl.TexSubImage1D()` und `gl.TexSubImage2D()` entpackt wurden. Pixeltransferoperationen erfolgen in der gleichen Reihenfolge und auf die gleiche Weise, unabhängig von dem Befehl, der zur Pixeloperation geführt hat. Pixelspeichermodi steuern das Entpacken von Pixeln, die aus dem Klientspeicher gelesen werden und das Packen von Pixeln, die wieder in den Klientspeicher geschrieben werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

Pixeltransferoperationen behandeln vier grundlegende Pixeltypen: Farbe, Farbindex, Tiefe und Schablone. Farbpixel bestehen aus vier Gleitkommawerten mit nicht angegebener Fangvorrichtung und Exponentialgrößen, skaliert so, dass 0 die Nullintensität und 1 die volle Intensität darstellt. Farbindizes bestehen aus einem einzigen Festpunktwert mit unbestimmter Genauigkeit rechts vom Binärpunkt. Tiefenpixel umfassen einen einzelnen Gleitkommawert mit nicht angegebener Fangvorrichtung und Exponentengrößen, skaliert so, dass 0,0 den minimalen Tiefenpufferwert und 1,0 den maximalen Tiefenpufferwert darstellt. Schließlich bestehen Schablonenpixel aus einem einzigen Festkommawert mit nicht angegebener Genauigkeit rechts neben dem Binärpunkt.

Die Pixelübertragungsvorgänge, die bei den vier grundlegenden Pixeltypen durchgeführt werden, sind wie folgt:

Color — Jede der vier Farbkomponenten wird mit einem Skalierungsfaktor multipliziert und dann zu einem Biasfaktor addiert. Das heißt, die rote Komponente

wird mit #GL_RED_SCALE multipliziert, dann zu #GL_RED_BIAS hinzugefügt; die grüne Komponente wird mit #GL_GREEN_SCALE multipliziert und dann zu #GL_GREEN_BIAS hinzugefügt; die blaue Komponente wird mit #GL_BLUE_SCALE multipliziert und dann zu #GL_BLUE_BIAS hinzugefügt; und die Alpha-Komponente wird mit #GL_ALPHA_SCALE multipliziert und dann zu #GL_ALPHA_BIAS hinzugefügt. Nachdem alle vier Farbkomponenten skaliert und voreingestellt sind, wird jede einzelne auf den Bereich [0, 1] festgelegt. Alle Farb-, Skalierungs- und Voreinstellungswerte werden mit gl.PixelTransfer() angegeben. Wenn #GL_MAP_COLOR True ist, wird jede Farbkomponente um die Größe der entsprechenden Farb-zu-Farb-Karte skaliert und dann durch den Inhalt dieser Karte ersetzt, die von der skalierten Komponente indiziert wird. Das heißt, die rote Komponente wird durch #GL_PIXEL_MAP_R_TO_R_R_SIZE skaliert und dann durch den Inhalt von #GL_PIXEL_MAP_R_TO_TO_R ersetzt, der von selbst indiziert wird. Die grüne Komponente wird durch #GL_PIXEL_MAP_G_TO_G_G_SIZE skaliert und dann durch den Inhalt von #GL_PIXEL_MAP_G_TO_G ersetzt, der von selbst indiziert wird. Die blaue Komponente wird durch #GL_PIXEL_MAP_B_TO_B_B_SIZE skaliert und dann durch den Inhalt von #GL_PIXEL_MAP_B_TO_B ersetzt, das von selbst indiziert wird. Und die Alpha-Komponente wird durch #GL_PIXEL_MAP_A_TO_A_A_SIZE skaliert und dann durch den Inhalt von #GL_PIXEL_MAP_A_TO_A ersetzt, der von selbst indiziert wird. Alle aus den Karten entnommenen Komponenten werden dann in den Bereich [0, 1] festgelegt. #GL_MAP_COLOR wird mit gl.PixelTransfer() angegeben. Der Inhalt der verschiedenen Karten wird durch die Angabe von gl.PixelMap() bestimmt.

Color index

Jeder Farbindex wird um #GL_INDEX_SHIFT-Bits nach links verschoben; alle Bits, die über die Anzahl der vom Festkommaindex getragenen Bruchteilen hinausgehen, werden mit Nullen gefüllt. Wenn #GL_INDEX_SHIFT negativ ist, ist die Verschiebung nach rechts, auch hier wird mit Nullen gefüllt. Dann wird #GL_INDEX_OFFSET dem Index hinzugefügt. #GL_INDEX_SHIFT und #GL_INDEX_OFFSET werden mit gl.PixelTransfer() angegeben.

Von diesem Punkt an variiert die Operation je nach dem erforderlichen Format der resultierenden Pixel. Wenn die resultierenden Pixel in einen Farbindexpuffer geschrieben werden sollen oder wenn sie im Format #GL_COLOR_INDEX in den Klient-Speicher zurückgelesen werden, werden die Pixel weiterhin als Indizes behandelt. Wenn #GL_MAP_COLOR True ist, wird jeder Index durch 2^n - 1 maskiert, wobei n #GL_PIXEL_MAP_I_TO_I_I_SI_SIZE ist, ersetzt durch den Inhalt von #GL_PIXEL_MAP_I_TO_I, indiziert durch den maskierten Wert. #GL_MAP_COLOR wird mit gl.PixelTransfer() angegeben. Der Inhalt der Indexkarte wird mit glPixelMap festgelegt.

Wenn die resultierenden Pixel in einen RGBA-Farbpuffer geschrieben werden sollen oder wenn sie in einem anderen Format als #GL_COLOR_INDEX in den Klient-Speicher zurückgelesen werden, werden die Pixel von Indizes in Farben umgewandelt, indem auf die vier Karten #GL_PIXEL_MAP_I_TO_R, #GL_PIXEL_MAP_I_TO_G, #GL_PIXEL_I_TO_G, #GL_PIXEL_MAP_I_I_TO_B

und #GL_PIXEL_MAP_I_TO_A Bezug genommen wird. Vor der Dereferenzierung wird der Index durch 2^n - 1 maskiert, wobei n #GL_PIXEL_MAP_I_TO_R_SIZE für die rote Karte, #GL_PIXEL_MAP_I_I_TO_G_SIZE für die grüne Karte, #GL_PIXEL_MAP_I_TO_B_SIZE für die blaue Karte und #GL_PIXEL_MAP_I_TO_A_SIZE für die Alpha-Karte ist. Alle aus den Karten entnommenen Komponenten werden dann in den Bereich [0, 1] festgelegt. Der Inhalt der vier Karten wird mit gl.PixelMap() festgelegt.

Depth — Jeder Tiefenwert wird mit #GL_DEPTH_SCALE multipliziert, zu #GL_DEPTH_BIAS addiert und dann auf den Bereich [0, 1] festgelegt.

Stencil — Jeder Index wird um #GL_INDEX_SHIFT Bits verschoben genau wie ein Farbindex und dann zu #GL_INDEX_OFFSET hinzugefügt. Wenn #GL_MAP_STENCIL True ist, wird jeder Index durch 2^n - 1 maskiert, wobei n #GL_PIXEL_MAP_S_TO_S_SIZE ist, dann ersetzt durch den Inhalt von #GL_PIXEL_MAP_S_TO_S_TO_S, indiziert durch den maskierten Wert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

pname — gibt den symbolischen Namen des zu setzenden Pixelübertragungsparameters an (siehe oben)

param — gibt den Wert an, auf den pname gesetzt ist

FEHLER

#GL_INVALID_ENUM wird erzeugt, wenn pname kein akzeptierter Wert ist.

#GL_INVALID_OPERATION wird erzeugt, wenn gl.PixelTransfer() zwischen gl.Begin() und gl.End() ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

gl.Get() mit dem Argument #GL_MAP_COLOR

gl.Get() mit dem Argument #GL_MAP_STENCIL

gl.Get() mit dem Argument #GL_INDEX_SHIFT

gl.Get() mit dem Argument #GL_INDEX_OFFSET

gl.Get() mit dem Argument #GL_RED_SCALE

gl.Get() mit dem Argument #GL_RED_BIAS

gl.Get() mit dem Argument #GL_GREEN_SCALE

gl.Get() mit dem Argument #GL_GREEN_BIAS

gl.Get() mit dem Argument #GL_BLUE_SCALE

gl.Get() mit dem Argument #GL_BLUE_BIAS

gl.Get() mit dem Argument #GL_ALPHA_SCALE

gl.Get() mit dem Argument #GL_ALPHA_BIAS

gl.Get() mit dem Argument #GL_DEPTH_SCALE

gl.Get() mit dem Argument #GL_DEPTH_BIAS

6.106 gl.PixelZoom

BEZEICHNUNG

gl.PixelZoom – gibt den Pixel-Zoomfaktoren an

ÜBERSICHT

```
gl.PixelZoom(xfactor, yfactor)
```

BESCHREIBUNG

`gl.PixelZoom()` gibt Werte für die Zoomfaktoren x und y an. Während der Ausführung von `gl.DrawPixels()` oder `gl.CopyPixels()`, wenn (xr,yr) die aktuelle Rasterposition ist und sich ein bestimmtes Element in der m-ten Zeile und n-ten Spalte des Pixelrechtecks befindet, dann sind Pixel, deren Zentren im Rechteck mit Ecken sind

```
(xr+n*xfactor, yr+m*yfactor)
(xr+(n+1)*xfactor,yr+(m+1)*yfactor)
```

Kandidaten für eine Nachfolge. Jedes Pixel, dessen Mittelpunkt am unteren oder linken Rand dieses rechteckigen Bereichs liegt, wird ebenfalls geändert.

Pixel-Zoomfaktoren sind nicht auf positive Werte beschränkt. Negative Zoomfaktoren spiegeln das resultierende Bild über die aktuelle Rasterposition wider.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`xfactor` gibt den x-Zoomfaktor für Pixel-Schreiboperationen an

`yfactor` gibt den y-Zoomfaktor für Pixel-Schreiboperationen an

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn glPixelZoom zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ZOOM_X`

`gl.Get()` mit dem Argument `#GL_ZOOM_Y`

6.107 gl.PointSize

BEZEICHNUNG

gl.PointSize – gibt den Durchmesser der gerasterten Punkte an

ÜBERSICHT

```
gl.PointSize(size)
```

BESCHREIBUNG

`gl.PointSize()` gibt den gerasterten Durchmesser von aliasierten und antialiatisierten Punkten an. Die Verwendung einer anderen Punktgröße als 1 hat unterschiedliche Auswirkungen, je nachdem, ob die Punkt-Antialiasing-Funktion aktiviert ist. Um Punktantialiasing zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_POINT_SMOOTH` auf. Die Punktantialiasing-Funktion ist zunächst deaktiviert.

Wenn die Punktantialiasing-Funktion deaktiviert ist, wird die tatsächliche Größe durch Rundung der gelieferten Größe auf die nächste Ganzzahl bestimmt (wenn die Rundung zu dem Wert 0 führt, ist es, als ob die Punktgröße 1 wäre). Wenn die gerundete Größe ungerade ist, dann wird der Mittelpunkt (x,y) des Pixelfragments, das den Punkt repräsentiert, wie folgt berechnet

```
(xw + 0.5, yw + 0.5)
```

wobei w-Indizes Fensterkoordinaten angeben. Alle Pixel, die innerhalb des quadratischen Rasters der abgerundeten Größe liegen und auf (x,y) zentriert sind, bilden das Fragment. Wenn die Größe gleich ist, ist der Mittelpunkt

```
(xw + 0.5, yw + 0.5)
```

und die Zentren des gerasterten Fragments sind die halb ganzzahligen Fensterkoordinaten innerhalb des Quadrats der abgerundeten Größe, zentriert auf (x,y). Allen Pixelfragmenten, die beim Rastern eines nicht-antialisierten Punktes erzeugt werden, werden die gleichen zugehörigen Daten zugewiesen, die des dem Punkt entsprechenden Scheitelpunktes.

Wenn Antialiasing aktiviert ist, erzeugt die Punktrasterung für jedes Pixelquadrat ein Fragment, das den Bereich schneidet, der innerhalb des Kreises liegt und einen Durchmesser hat, der gleich der aktuellen Punktgröße ist und auf den Punkt (xw,yw) zentriert ist. Der Abdeckungswert für jedes Fragment ist der Fensterkoordinatenbereich des Schnittpunktes des kreisförmigen Bereichs mit dem entsprechenden Pixelquadrat. Dieser Wert wird gespeichert und im letzten Schritt der Rasterung verwendet. Die jedem Fragment zugeordneten Daten sind die Daten, die dem zu rasternden Punkt zugeordnet sind.

Nicht alle Größen werden unterstützt, wenn die Punktantialiasing-Funktion aktiviert ist. Wenn eine nicht unterstützte Größe angefordert wird, wird die nächstgelegene unterstützte Größe verwendet. Nur die Größe 1 wird garantiert unterstützt, andere sind von der Implementierung abhängig. Um den Bereich der unterstützten Größen und die Größendifferenz zwischen den unterstützten Größen innerhalb des Bereichs abzufragen, rufen Sie `gl.Get()` mit den Argumenten `#GL_POINT_SIZE_RANGE` und `#GL_POINT_SIZE_GRANULARITY` auf.

Die durch `gl.PointSize()` angegebene Punktgröße wird immer zurückgegeben, wenn `#GL_POINT_SIZE` abgefragt wird. Das Befestigen und Runden für aliasierte und antialiatisierte Punkte hat keinen Einfluss auf den angegebenen Wert.

Eine nicht-antialisierte Punktgröße kann auf ein implementierungsabhängiges Maximum festgelegt werden. Obwohl dieses Maximum nicht abgefragt werden kann, darf es nicht kleiner sein als der Maximalwert für Antialiaspunkte, gerundet auf den nächsten ganzzahligen Wert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`size` gibt den Durchmesser der gerasterten Punkte an; der Anfangswert ist 1

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `size` kleiner oder gleich 0 ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PointSize()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_POINT_SIZE`

`gl.Get()` mit dem Argument `#GL_POINT_SIZE_RANGE`

`gl.Get()` mit dem Argument `#GL_POINT_SIZE_GRANULARITY`

`gl.IsEnabled()` mit dem Argument `#GL_POINT_SMOOTH`

6.108 gl.PolygonMode

BEZEICHNUNG

gl.PolygonMode – wählt einen Polygon-Rastermodus aus

ÜBERSICHT

```
gl.PolygonMode(face, mode)
```

BESCHREIBUNG

`gl.PolygonMode()` steuert die Interpretation von Polygonen für die Rasterung. `face` beschreibt, welcher Polygonmodus gilt: Nach vorne gerichtete Polygone (`#GL_FRONT`), nach hinten gerichtete Polygone (`#GL_BACK`) oder beides (`#GL_FRONT_AND_BACK`). Der Polygonmodus wirkt sich nur auf die endgültige Rasterung von Polygonen aus. Insbesondere werden die Eckpunkte eines Polygons beleuchtet und das Polygon abgeschnitten und eventuell ausgelesen, bevor diese Modi angewendet werden.

Es sind drei Modi definiert, die im `mode` festgelegt werden können:

`#GL_POINT`
: Polygonknoten, die als Anfang einer Begrenzungskante markiert sind, werden als Punkte gezeichnet. Punktattribute wie `#GL_POINT_SIZE` und `#GL_POINT_SMOOTH` steuern die Rasterung der Punkte. Andere Polygonrasterungsattribute als `#GL_POLYGON_MODE` haben keine Auswirkung.

`#GL_LINE`
: Begrenzungskanten des Polygons werden als Liniensegmente gezeichnet. Sie werden als verbundene Liniensegmente für das Linienpunktieren behandelt; der Linienpunktierungszähler und das Muster werden zwischen den Segmenten nicht zurückgesetzt (siehe `gl.LineStipple()`). Linienattribute wie `#GL_LINE_WIDTH` und `#GL_LINE_SMOOTH` steuern die Rasterung der Linien. Andere Polygonrasterungsattribute als `#GL_POLYGON_MODE` haben keine Auswirkung.

`#GL_FILL`
: Das Innere des Polygons wird ausgefüllt. Polygonattribute wie `#GL_POLYGON_STIPPLE` und `#GL_POLYGON_SMOOTH` steuern die Rasterung des Polygons.

Der Initialwert ist `#GL_FILL` für nach vorne und hinten gerichtete Polygone.

Eckpunkte werden als Grenze oder nicht begrenzt mit einem Kantenflag markiert. Kantenflags werden intern vom GL erzeugt, wenn er Polygone zerlegt; sie können explizit mit `gl.EdgeFlag()` gesetzt werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`face` gibt die Polygone an, für die der Modus gilt (siehe oben)

`mode` legt fest, wie Polygone gerastert werden sollen (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn entweder `face` oder `mode` kein akzeptierter Wert ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PolygonMode()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_POLYGON_MODE`

6.109 gl.PolygonOffset

BEZEICHNUNG

gl.PolygonOffset – stellt die Skala und die Einheiten zur Berechnung der Tiefenwerte ein

ÜBERSICHT

```
gl.PolygonOffset(factor, units)
```

BESCHREIBUNG

Wenn `#GL_POLYGON_OFFSET_FILL`, `#GL_POLYGON_OFFSET_LINE` oder `#GL_POLYGON_OFFSET_POINT` aktiviert ist, wird der Tiefenwert jedes Fragments nach der Interpolation aus den Tiefenwerten der entsprechenden Knoten verschoben. Der Wert des Versatzes ist Faktor * delta(Z) + r * Einheiten, wobei delta(Z) ein Maß für die Tiefenänderung in Bezug auf die Bildschirmfläche des Polygons ist und r der kleinste Wert ist, der garantiert einen lösbaren Versatz für eine gegebene Implementierung erzeugt. Der Versatz wird addiert, bevor der Tiefentest durchgeführt wird und bevor der Wert in den Tiefenpuffer geschrieben wird.

`gl.PolygonOffset()` ist nützlich für das Rendern von Bildern mit verdeckten Linien, für das Aufbringen von Bildern auf Oberflächen und für das Rendern von Festkörpern mit hervorgehobenen Kanten.

`gl.PolygonOffset()` hat keinen Einfluss auf die Tiefenkoordinaten, die im Feedback-Puffer platziert sind.

`gl.PolygonOffset()` hat keinen Einfluss auf die Auswahl (Modus).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`factor` gibt einen Skalierungsfaktor an, der verwendet wird, um einen variablen Tiefenversatz für jedes Polygon zu erzeugen; der Anfangswert ist 0

`units` wird mit einem implementierungsspezifischen Wert multipliziert, um einen konstanten Tiefenversatz zu erzeugen; der Anfangswert ist 0

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PolygonOffset()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_POLYGON_OFFSET_FILL`, `#GL_POLYGON_OFFSET_LINE`, oder `#GL_POLYGON_OFFSET_POINT`.

`gl.Get()` mit dem Argument `#GL_POLYGON_OFFSET_FACTOR` oder `#GL_POLYGON_OFFSET_UNITS`.

6.110 gl.PolygonStipple

BEZEICHNUNG

gl.PolygonStipple – setzt das Polygon-Punktemuster

ÜBERSICHT

```
gl.PolygonStipple(maskArray)
```

BESCHREIBUNG

Das Polygon-Punktieren, wie das Linien-Punktieren (siehe `gl.LineStipple()`), maskiert bestimmte durch die Rasterung erzeugte Fragmente und erzeugt ein Muster. Das Punktieren ist unabhängig von der Polygon-Antialiasing-Funktion.

`maskArray` ist eine Tabelle mit einem 32*32 Punktemuster, das als monochrome Bitmap gespeichert ist und nur 1 Bit pro Pixel verwendet. Die Bitmap wird in einer Tabelle übergeben, die aus Teilen von 8 Pixeln besteht, die in ein Byte gepackt sind. Für ein 32*32 Punktemuster müssen Sie also eine Tabelle übergeben, die 128 Byte-Elemente mit jeweils 8 Pixeln enthält. Dies kann entweder eine eindimensionale Tabelle mit 128 Byte-Einträgen oder eine zweidimensionale Tabelle mit 32 Untertabellen mit je 4 Byte-Einträgen sein (diese 4 Byte-Einträge beschreiben eine Zeile von je 32 Pixeln). Die Daten werden in einem zusammenhängenden Speicherblock ohne Ausfüllung oder spezielle Ausrichtungen an GL übergeben, so dass keine exotischen Einstellungen mit `gl.PixelStore()` vorgenommen werden. Sie sind aktiv, da `gl.PolygonStipple()` erwartet, dass die Musterdaten im Speicher gespeichert werden, genau wie die Pixeldaten, die an `gl.DrawPixels()` geliefert werden mit Höhe und Breite gleich 32, einem Pixelformat von `#GL_COLOR_INDEX` und Datentyp von `#GL_BITMAP`. Das heißt, das Punktmuster wird als 32x32-Feld von 1-Bit-Farbindizes dargestellt, die in unsignierten Bytes gepackt sind. `gl.PixelStore()` Parameter wie `#GL_UNPACK_SWAP_BYTES` und `#GL_UNPACK_LSB_FIRST` beeinflussen die Zusammensetzung der Bits zu einem Punktmuster. Pixelübertragungsvorgänge (Shift, Versatz, Pixel-Map) werden nicht auf das Punktbild angewendet.

Um das Polygon-Punktieren zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_POLYGON_STIPPLE` auf. Das Punktieren von Polygonen ist zunächst deaktiviert. Wenn es aktiviert ist, wird ein gerastertes Polygonfragment mit den Fensterkoordinaten xw und yw an die nächste Stufe von GL gesendet, wenn und nur wenn das (xw%32)-te Bit in der (yw%32)-ten Zeile des Punktemusters 1 (eins) ist. Wenn das Polygon-Punktieren deaktiviert ist, ist es so, als ob das Punktemuster aus allen 1's besteht.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`maskArray`
: gibt eine Tabelle an, die ein 32x32-Punktemuster enthält

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PolygonStipple()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetPolygonStipple()`

`gl.IsEnabled()` mit dem Argument `#GL_POLYGON_STIPPLE`

6.111 gl.PopAttrib

BEZEICHNUNG

gl.PopAttrib – öffnet den Server-Attributstapel

ÜBERSICHT

`gl.PopAttrib()`

BESCHREIBUNG

`gl.PopAttrib()` stellt die Werte der Zustandsvariablen wieder her, die mit dem letzten Befehl `gl.PushAttrib()` gespeichert wurden. Die nicht gespeicherten bleiben unverändert. Siehe Abschnitt 6.116 [gl.PushAttrib], Seite 179, für eine Liste der unterstützten Zustandsvariablen.

Es ist ein Fehler, Attribute von einem leeren Stapel zu entfernen. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_STACK_UNDERFLOW` wird erzeugt, wenn `gl.PopAttrib()` aufgerufen wird, während der Attributstabel leer ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PopAttrib()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_ATTRIB_STACK_DEPTH`

6.112 gl.PopClientAttrib

BEZEICHNUNG

gl.PopClientAttrib – öffnet den Klient-Attributstapel

ÜBERSICHT

`gl.PopClientAttrib()`

BESCHREIBUNG

`gl.PopClientAttrib()` stellt die Werte der Klient-Zustandsvariablen wieder her, die mit dem letzten `gl.PushClientAttrib()` gespeichert wurden. Die nicht gespeicherten bleiben unverändert.

Siehe Abschnitt 6.117 [gl.PushClientAttrib], Seite 184, für eine Liste der unterstützten Klient-Zustandsvariablen.

Es ist ein Fehler, Attribute von einem leeren Stapel zu entfernen. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_STACK_UNDERFLOW` wird erzeugt, wenn `gl.PopClientAttrib()` aufgerufen wird, während der Attributstapel leer ist.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_CLIENT_ATTRIB_STACK_DEPTH`

6.113 gl.PopMatrix

BEZEICHNUNG

gl.PopMatrix – öffnet den aktuellen Matrixstapel

ÜBERSICHT

`gl.PopMatrix()`

BESCHREIBUNG

`gl.PopMatrix()` öffnet den aktuellen Matrixstapel und ersetzt die aktuelle Matrix durch die darunter liegende Matrix auf dem Stapel.

Zunächst enthält jeder der Stapel eine Matrix, eine Identitätsmatrix.

Es ist ein Fehler, einen Matrixstapel zu öffnen, der nur eine einzige Matrix enthält. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_STACK_UNDERFLOW` wird erzeugt, wenn `gl.PopMatrix()` aufgerufen wird, während der aktuelle Matrixstapel nur eine einzige Matrix enthält.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PopMatrix()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_PROJECTION_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_TEXTURE_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_MODELVIEW_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_PROJECTION_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_TEXTURE_STACK_DEPTH`

6.114 gl.PopName

BEZEICHNUNG

gl.PopName – öffnet den Namensstapel

ÜBERSICHT

`gl.PopName()`

BESCHREIBUNG

Der Namensstapel wird im Auswahlmodus verwendet, um eine eindeutige Identifizierung von Renderingbefehlen zu ermöglichen. Er besteht aus einem geordneten Satz von vorzeichenlosen Ganzzahlen und ist zunächst leer.

`gl.PopName()` gibt einen Namen oben vom Stapel ab. Die maximale Tiefe des Namensstapels ist implementierungsabhängig; rufen Sie `#GL_MAX_NAME_STACK_DEPTH` auf, um den Wert für eine bestimmte Implementierung herauszufinden.

Es ist ein Fehler, einen Namen von einem leeren Stapel zu entfernen. Es ist auch ein Fehler, den Namensstapel zwischen der Ausführung von `gl.Begin()` und `gl.End()` zu manipulieren. In jedem dieser Fälle wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Der Namensstapel ist immer leer, wenn der Rendermodus nicht `#GL_SELECT` ist und Aufrufe von `gl.PopName()` werden dann ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

FEHLER

`#GL_STACK_UNDERFLOW` wird erzeugt, wenn `gl.PopName()` aufgerufen wird, während der Namensstapel leer ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PopName()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_NAME_STACK_DEPTH`

6.115 gl.PrioritizeTextures

BEZEICHNUNG

gl.PrioritizeTextures – stellt die Priorität der Textur-Residenz ein

ÜBERSICHT

```
gl.PrioritizeTextures(texturesArray, prioritiesArray)
```

BESCHREIBUNG

`gl.PrioritizeTextures()` weist die in `prioritiesArray` angegebenen Texturprioritäten den in `texturesArray` genannten Texturen zu.

GL etabliert einen "Arbeitsumfang" von Texturen, die sich im Texturspeicher befinden. Diese Texturen können viel effizienter an ein Texturziel gebunden werden als Texturen, die nicht resident sind. Durch die Angabe einer Priorität für jede Textur ermöglicht `gl.PrioritizeTextures()` Anwendungen, die die GL-Implementierung bei der Bestimmung, welche Texturen resident sein sollen, zu leiten.

Die in `prioritiesArray` angegebenen Prioritäten werden vor der Zuweisung auf den Bereich (0,1) festgelegt. 0 zeigt die niedrigste Priorität an; Texturen mit der Priorität 0 sind am wenigsten wahrscheinlich resident. 1 zeigt die höchste Priorität an; Texturen mit Priorität 1 sind am ehesten resident. Es wird jedoch nicht garantiert, dass Texturen bis zu ihrer Verwendung resident sind.

`gl.PrioritizeTextures()` ignoriert stillschweigend Versuche, die Textur 0 oder einen beliebigen Texturnamen zu priorisieren, der nicht mit einer bestehenden Textur übereinstimmt.

`gl.PrioritizeTextures()` verlangt nicht, dass eine der von Texturen angegebenen Texturen an ein Texturziel gebunden wird. `gl.TexParameter()` kann auch verwendet werden, um die Priorität einer Textur festzulegen, aber nur, wenn die Textur aktuell gebunden ist. Dies ist die einzige Möglichkeit, die Priorität einer Standardtextur festzulegen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`texturesArray`
: gibt ein Feld an, das die Namen der zu priorisierenden Texturen enthält

`prioritiesArray`
: gibt ein Feld an, das die Texturprioritäten enthält; eine Priorität, die in einem Element der Prioritäten angegeben ist, gilt für die Textur, die durch das entsprechende Element der Texturen benannt ist

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PrioritizeTextures()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexParameter()` mit dem Parametername `#GL_TEXTURE_PRIORITY` ruft die Priorität einer aktuell gebundenen Textur ab.

6.116 gl.PushAttrib

BEZEICHNUNG

gl.PushAttrib – verschiebt den Server-Attributstapel

ÜBERSICHT

`gl.PushAttrib(mask)`

BESCHREIBUNG

`gl.PushAttrib()` verwendet eine Maske als Argument, die angibt, welche Gruppen von Statusvariablen im Attributstapel gespeichert werden sollen. Symbolische Konstanten werden verwendet, um Bits in der Maske zu setzen. Die Maske wird typischerweise durch O-Ring aus mehreren dieser Konstanten zusammengesetzt. Die spezielle Maske `#GL_ALL_ATTRIB_BITS` kann verwendet werden, um alle stapelbaren Zustände zu speichern.

Die symbolischen Maskenkonstanten und der damit verbundene GL-Status sind wie folgt:

`#GL_ACCUM_BUFFER_BIT`
: Löschwert des Akkumulationspuffer

`#GL_COLOR_BUFFER_BIT`
: `#GL_ALPHA_TEST` Freigabebit
: Alpha-Testfunktion und Referenzwert
: `#GL_BLEND` Freigabebit
: Zusammenführung von Quell- und Zielfunktionen
: Konstante Mischfarbe
: Mischungsgleichung
: `#GL_DITHER` Freigabebit
: `#GL_DRAW_BUFFER` Einstellung
: `#GL_COLOR_LOGIC_OP` Freigabebit
: `#GL_INDEX_LOGIC_OP` Freigabebit
: Logik-Op-Funktion
: Farbmodus und Indexmodus Werte löschen
: Farbmodus- und Indexmodus-Schreibmasken

`#GL_CURRENT_BIT`
: Aktuelle RGBA-Farbe
: Aktueller Farbindex
: Aktueller Standardvektor
: Aktuelle Texturkoordinaten
: Aktuelle Rasterposition
: `#GL_CURRENT_RASTER_POSITION_VALID` Flag
: RGBA-Farbe, die der aktuellen Rasterposition zugeordnet ist
: Farbindex zugeordnet zur aktuellen Rasterposition
: Texturkoordinaten, die der aktuellen Rasterposition zugeordnet sind
: `#GL_EDGE_FLAG` Flag

#GL_DEPTH_BUFFER_BIT
- #GL_DEPTH_TEST Freigabebit
- Tiefenpuffer-Testfunktion
- Tiefenpuffer Löschwert
- #GL_DEPTH_WRITEMASK Freigabebit

#GL_ENABLE_BIT
- #GL_ALPHA_TEST Flag
- #GL_AUTO_NORMAL Flag
- #GL_BLEND Flag
- Aktivierungsbits für die benutzerdefinierbaren Ausschnittebenen
- #GL_COLOR_MATERIAL
- #GL_CULL_FACE Flag
- #GL_DEPTH_TEST Flag
- #GL_DITHER Flag
- #GL_FOG Flag
- #GL_LIGHTi wobei 0 .le. i < #GL_MAX_LIGHTS
- #GL_LIGHTING Flag
- #GL_LINE_SMOOTH Flag
- #GL_LINE_STIPPLE Flag
- #GL_COLOR_LOGIC_OP Flag
- #GL_INDEX_LOGIC_OP Flag
- #GL_MAP1_x wobei x ein Kartentyp ist
- #GL_MAP2_x wobei x ein Kartentyp ist
- #GL_NORMALIZE Flag
- #GL_POINT_SMOOTH Flag
- #GL_POLYGON_OFFSET_LINE Flag
- #GL_POLYGON_OFFSET_FILL Flag
- #GL_POLYGON_OFFSET_POINT Flag
- #GL_POLYGON_SMOOTH Flag
- #GL_POLYGON_STIPPLE Flag
- #GL_SCISSOR_TEST Flag
- #GL_STENCIL_TEST Flag
- #GL_TEXTURE_1D Flag
- #GL_TEXTURE_2D Flag
- Flags #GL_TEXTURE_GEN_x wobei x ein S, T, R, oder Q ist

#GL_EVAL_BIT
- #GL_MAP1_x Aktivierungsbits, wobei x ein Kartentyp ist
- #GL_MAP2_x Aktivierungsbits, wobei x ein Kartentyp ist
- 1D Gitterendpunkte und -teilungen

2D Gitterendpunkte und -teilungen
#GL_AUTO_NORMAL Freigabebit

#GL_FOG_BIT
#GL_FOG Freigabebit
Nebelfarbe
Nebeldichte
Linearer Nebelstart
Lineares Nebelende
NebelIndex
#GL_FOG_MODE Wert

#GL_HINT_BIT
#GL_PERSPECTIVE_CORRECTION_HINT Einstellung
#GL_POINT_SMOOTH_HINT Einstellung
#GL_LINE_SMOOTH_HINT Einstellung
#GL_POLYGON_SMOOTH_HINT Einstellung
#GL_FOG_HINT Einstellung

#GL_LIGHTING_BIT
#GL_COLOR_MATERIAL Freigabebit
#GL_COLOR_MATERIAL_FACE Wert
Farbmaterialparameter, die der aktuellen Farbe folgen
Farbe der Umgebungsszene
#GL_LIGHT_MODEL_LOCAL_VIEWER Wert
#GL_LIGHT_MODEL_TWO_SIDE Einstellung
#GL_LIGHTING Freigabebit
Freigabebit für jede Leuchte
Umgebungs-, Diffus- und Spiegelintensität für jedes Licht
Richtung, Position, Exponent und Cutoff-Winkel für jede Leuchte
Konstante, lineare und quadratische Dämpfungsfaktoren für jedes Licht
Umgebungs-, diffuse, spiegelnde und emissive Farbe für jedes Material
Umgebungs-, Diffus- und Spiegel-Farbindizes für jedes Material
Spiegelexponent für jedes Material
#GL_SHADE_MODEL Einstellung

#GL_LINE_BIT
#GL_LINE_SMOOTH Flag
#GL_LINE_STIPPLE Freigabebit
Linien-Punktemuster und Wiederholungszähler
Linienbreite

#GL_LIST_BIT
#GL_LIST_BASE Einstellung

`#GL_PIXEL_MODE_BIT`
: `#GL_RED_BIAS` und `#GL_RED_SCALE` Einstellungen
: `#GL_GREEN_BIAS` und `#GL_GREEN_SCALE` Werte
: `#GL_BLUE_BIAS` und `#GL_BLUE_SCALE` Werte
: `#GL_ALPHA_BIAS` und `#GL_ALPHA_SCALE` Werte
: `#GL_DEPTH_BIAS` und `#GL_DEPTH_SCALE` Werte
: `#GL_INDEX_OFFSET` und `#GL_INDEX_SHIFT` Werte
: `#GL_MAP_COLOR` und `#GL_MAP_STENCIL` Flags
: `#GL_ZOOM_X` und `#GL_ZOOM_Y` Faktoren
: `#GL_READ_BUFFER` Einstellung

`#GL_POINT_BIT`
: `#GL_POINT_SMOOTH` Flag Punktgröße

`#GL_POLYGON_BIT`
: `#GL_CULL_FACE` Freigabebit
: `#GL_CULL_FACE_MODE` Wert
: `#GL_FRONT_FACE` Indikator
: `#GL_POLYGON_MODE` Einstellung
: `#GL_POLYGON_SMOOTH` Flag
: `#GL_POLYGON_STIPPLE` Freigabebit
: `#GL_POLYGON_OFFSET_FILL` Flag
: `#GL_POLYGON_OFFSET_LINE` Flag
: `#GL_POLYGON_OFFSET_POINT` Flag
: `#GL_POLYGON_OFFSET_FACTOR`
: `#GL_POLYGON_OFFSET_UNITS`

`#GL_POLYGON_STIPPLE_BIT`
: Polygon-Punktierungsbild

`#GL_SCISSOR_BIT`
: `#GL_SCISSOR_TEST` Flag
: Scherenkasten

`#GL_STENCIL_BUFFER_BIT`
: `#GL_STENCIL_TEST` Freigabebit
: Schablonenfunktion und Referenzwert
: Schablonenwertmaske
: Schablonenfehler, Durchlauf und Tiefenpuffer Durchlaufaktionen
: Schablonenpuffer Löschwert
: Schablonenpuffer Schreibmaske

`#GL_TEXTURE_BIT`
: Aktivieren von Bits für die vier Texturkoordinaten
: Rahmenfarbe für jedes Texturbild

Verkleinerungsfunktion für jedes Texturbild

Vergrößerungsfunktion für jedes Texturbild

Texturkoordinaten und Wrap-Modus für jedes Texturbild

Farbe und Modus für jede Texturumgebung

Freigabebits `#GL_TEXTURE_GEN_x`, x ist S, T, R und Q

`#GL_TEXTURE_GEN_MODE` Einstellung für S, T, R und Q

`gl.TexGen()` Ebenengleichungen für S, T, R und Q

Aktuelle Texturbindungen (z.B., `#GL_TEXTURE_2D_BINDING`)

`#GL_TRANSFORM_BIT`

Koeffizienten der sechs Ausschnittebenen

Aktivierungsbits für die benutzerdefinierbaren Ausschnittebenen

`#GL_MATRIX_MODE` Wert

`#GL_NORMALIZE` Flag

`#GL_VIEWPORT_BIT`

Tiefenbereich (nah und fern)

Ansichtsfenster Ursprung und Umfang

Es ist ein Fehler, Attribute auf einen vollen Stapel zu schieben. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Zunächst ist der Attributstabel leer.

Nicht alle Werte für den GL-Status können auf dem Attributstabel gespeichert werden. So können beispielsweise den Status des Rendermodus sowie den Status von Auswahl und Feedback nicht gespeichert werden. Der Client-Status muss mit `gl.PushClientAttrib()` gespeichert werden.

Die Tiefe des Attributstapels hängt von der Implementierung ab, muss aber mindestens 16 betragen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mask` spezifiziert eine Maske, die angibt, welche Attribute gespeichert werden sollen (siehe oben)

FEHLER

`#GL_STACK_OVERFLOW` wird erzeugt, wenn `gl.PushAttrib()` aufgerufen wird, während der Attributstapel voll ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PushAttrib()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_ATTRIB_STACK_DEPTH`

6.117 gl.PushClientAttrib

BEZEICHNUNG

gl.PushClientAttrib – verschiebt den Klient-Attributstapel

ÜBERSICHT

`gl.PushClientAttrib(mask)`

BESCHREIBUNG

`gl.PushClientAttrib()` verwendet eine Maske als Argument, die angibt, welche Gruppen von Klient-Zustandsvariablen auf dem Klient-Attributstapel gespeichert werden sollen. Symbolische Konstanten werden verwendet, um Bits in der Maske zu setzen. Die `mask` wird typischerweise durch die Angabe der bitweisen oder mehrerer dieser Konstanten zusammen aufgebaut. Die spezielle Maske `#GL_CLIENT_ALL_ATTRIB_BITS` kann verwendet werden, um alle stapelbaren Klient-Status zu speichern.

Die symbolischen Maskenkonstanten und der damit verbundene GL-Klient-Status sind wie folgt:

`#GL_CLIENT_PIXEL_STORE_BIT`
: Pixel-Speichermodi

`#GL_CLIENT_VERTEX_ARRAY_BIT`
: Scheitelpunkt-Felder (und -Aktivierungen)

Es ist ein Fehler, Attribute auf einen vollständigen Klient-Attributstapel zu verschieben. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Zunächst ist der Klient-Attributstapel leer.

Nicht alle Werte für den GL-Klient-Status können auf dem Attribut Stapel gespeichert werden. So können beispielsweise der Auswahl- und Feedback-Status nicht gespeichert werden.

Die Tiefe des Attributstapels hängt von der Implementierung ab, muss aber mindestens 16 betragen.

Verwenden Sie gl.PushAttrib(), um den Verschiebe-Status zu erhalten, der auf dem Server gehalten wird. Nur Pixelspeichermodi und Scheitelpunkt-Feld-Zustände können mit `gl.PushClientAttrib()` verschoben werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mask` spezifiziert eine Maske, die angibt, welche Attribute gespeichert werden sollen (siehe oben)

FEHLER

`#GL_STACK_OVERFLOW` wird erzeugt, wenn `gl.PushClientAttrib()` aufgerufen wird, während der Attributstapel voll ist.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_ATTRIB_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_CLIENT_ATTRIB_STACK_DEPTH`

6.118 gl.PushMatrix

BEZEICHNUNG

gl.PushMatrix – verschiebt den aktuellen Matrixstapel

ÜBERSICHT

`gl.PushMatrix()`

BESCHREIBUNG

Für jeden der Matrixmodi gibt es einen Stapel von Matrizen. Im #GL_MODELVIEW-Modus beträgt die Stapeltiefe mindestens 32. In den anderen Modi, `#GL_COLOR`, `#GL_PROJECTION` und `#GL_TEXTURE`, beträgt die Tiefe mindestens 2, wobei die aktuelle Matrix in jedem Modus die Matrix oben auf dem Stapel für diesen Modus ist.

`gl.PushMatrix()` verschiebt den aktuellen Matrixstapel um eins nach unten und dupliziert die aktuelle Matrix. Das heißt, nach einem gl.PushMatrix()-Aufruf ist die Matrix oben auf dem Stapel identisch mit der darunter.

Zunächst enthält jeder der Stapel eine Matrix, eine Identitätsmatrix.

Es ist ein Fehler, einen vollen Matrixstapel zu verschieben. In diesem Fall wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

Keine

FEHLER

`#GL_STACK_OVERFLOW` wird erzeugt, wenn `gl.PushMatrix()` aufgerufen wird, während der aktuelle Matrixstapel voll ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PushMatrix()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_PROJECTION_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_TEXTURE_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_MODELVIEW_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_PROJECTION_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_TEXTURE_STACK_DEPTH`

6.119 gl.PushName

BEZEICHNUNG

gl.PushName – verschiebt den Namensstapel

ÜBERSICHT

```
gl.PushName(name)
```

BESCHREIBUNG

Der Namensstapel wird im Auswahlmodus verwendet, um eine eindeutige Identifizierung von Renderingbefehlen zu ermöglichen. Er besteht aus einem geordneten Satz von vorzeichenlosen Ganzzahlen und ist zunächst leer.

`gl.PushName()` bewirkt, dass der Name auf den Namensstapel geschoben wird.

Die maximale Tiefe des Namensstapels ist implementierungsabhängig; rufen Sie `#GL_MAX_NAME_STACK_DEPTH` auf, um den Wert für eine bestimmte Implementierung herauszufinden.

Es ist ein Fehler, einen Namen auf einen vollen Stapel zu schieben. Es ist auch ein Fehler, den Namensstapel zwischen der Ausführung von `gl.Begin()` und `gl.End()` zu manipulieren. In jedem dieser Fälle wird das Fehlerflag gesetzt und es wird keine weitere Änderung des GL-Zustandes vorgenommen.

Der Namensstapel ist immer leer, wenn der Rendermodus nicht `#GL_SELECT` ist. Aufrufe von `gl.PushName()`, wenn der Rendermodus nicht `#GL_SELECT` ist, werden ignoriert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`name` gibt einen Namen an, der auf den Namensstapel geschoben wird

FEHLER

`#GL_STACK_OVERFLOW` wird erzeugt, wenn `gl.PushName()` aufgerufen wird, während der Namensstapel voll ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.PushName()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_MAX_NAME_STACK_DEPTH`

6.120 gl.RasterPos

BEZEICHNUNG

gl.RasterPos – gibt die Rasterposition für Pixeloperationen an

ÜBERSICHT

```
gl.RasterPos(x, y[, z, w])
```

BESCHREIBUNG

GL hält eine 3D-Position in Fensterkoordinaten ein. Diese Position, die so genannte Rasterposition, wird verwendet, um Pixel- und Bitmap-Schreiboperationen zu positionieren. Sie wird mit Subpixelgenauigkeit verwaltet. Siehe Abschnitt 6.7 [gl.Bitmap], Seite 31, für Details. Siehe Abschnitt 6.36 [gl.DrawPixels], Seite 67, für Details. Siehe Abschnitt 6.22 [gl.CopyPixels], Seite 46, für Details.

Die aktuelle Rasterposition besteht aus drei Fensterkoordinaten (x, y, z), einem Ausschnitt-Koordinatenwert (w), einem Augenkoordinatenabstand, einem gültigen Bit

und den zugehörigen Farbdaten und Texturkoordinaten. Die w-Koordinate ist eine Ausschnitt-Koordinate, da w nicht auf Fensterkoordinaten projiziert wird.

Die von `gl.RasterPos()` dargestellten Objektkoordinaten werden wie die eines `gl.Vertex()` Befehl behandelt: Sie werden durch die aktuellen Modellansichten und Projektionsmatrizen transformiert und an die Ausschnitt-Phase übergeben. Wenn der Knoten nicht geholt wird, dann wird er projiziert und auf Fensterkoordinaten skaliert, die zur neuen aktuellen Rasterposition werden und das Flag `#GL_CURRENT_RASTER_POSITION_VALID` wird gesetzt. Wenn der Knoten gelesen wird, dann wird das gültige Bit gelöscht und die aktuelle Rasterposition und die zugehörigen Farb- und Texturkoordinaten sind undefiniert.

Die aktuelle Rasterposition enthält auch einige zugehörige Farbdaten und Texturkoordinaten. Wenn die Beleuchtung aktiviert ist, wird `#GL_CURRENT_RASTER_COLOR` (im RGBA-Modus) oder `#GL_CURRENT_RASTER_INDEX` (im Farbindex-Modus) auf die von der Lichtberechnung erzeugte Farbe gesetzt (siehe `gl.Light()`, `gl.LightModel()` und `gl.ShadeModel()`). Wenn die Beleuchtung deaktiviert ist, wird die aktuelle Farbe (im RGBA-Modus, Zustandsvariable `#GL_CURRENT_COLOR`) oder der Farbindex (im Farbindex-Modus, Zustandsvariable `#GL_CURRENT_INDEX`) verwendet, um die aktuelle Rasterfarbe zu aktualisieren.

Ebenso wird `#GL_CURRENT_RASTER_TEXTURE_COORDS` in Abhängigkeit von `#GL_CURRENT_TEXTURE_COORDS` aktualisiert, basierend auf der Texturmatrix und den Funktionen zur Texturerzeugung. Siehe Abschnitt 6.137 [gl.TexGen], Seite 207, für Details. Schließlich ersetzt der Abstand vom Ursprung des Augenkoordinatensystems zum Scheitelpunkt, der nur durch die Modellsichtmatrix transformiert wird, `#GL_CURRENT_RASTER_DISTANCE`.

Zunächst ist die aktuelle Rasterposition (0, 0, 0, 1), der aktuelle Rasterabstand ist 0, das gültige Bit ist gesetzt, die zugehörige RGBA-Farbe ist (1, 1, 1, 1), der zugehörige Farbindex ist 1 und die zugehörigen Texturkoordinaten sind (0, 0, 0, 1). Im RGBA-Modus ist `#GL_CURRENT_RASTER_INDEX` immer 1; im Farbindex-Modus behält die aktuelle Raster-RGBA-Farbe immer ihren Anfangswert.

Die Rasterposition wird geändert durch `gl.RasterPos()` und `gl.Bitmap()`.

Wenn die Koordinaten der Rasterposition ungültig sind, werden Zeichenbefehle, die auf der Rasterposition basieren, ignoriert (d.h. sie führen nicht zu Änderungen im GL-Status).

Der Aufruf von `gl.DrawElements()` kann die aktuelle Farbe oder den Index unbestimmt lassen. Wenn `gl.RasterPos()` ausgeführt wird, während die aktuelle Farbe oder der aktuelle Index unbestimmt ist, bleibt die aktuelle Rasterfarbe oder der aktuelle Rasterindex unbestimmt.

Alternativ kann `gl.RasterPos()` auch mit einem einzigen Tabellenargument aufgerufen werden, das zwei bis vier Koordinaten enthält, die als neue Rasterposition festgelegt werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`	gibt die x-Objektkoordinaten für die Rasterposition an
`y`	gibt die y-Objektkoordinaten für die Rasterposition an

`z` optional: Gibt die Koordinaten des z-Objekts für die Rasterposition an (Standard ist 0)

`w` optional: Gibt die Koordinaten des w-Objekts für die Rasterposition an (Standard ist 1)

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.RasterPos()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_POSITION_VALID`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_DISTANCE`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_COLOR`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_INDEX`

`gl.Get()` mit dem Argument `#GL_CURRENT_RASTER_TEXTURE_COORDS`

6.121 gl.ReadBuffer

BEZEICHNUNG

gl.ReadBuffer – wählt eine Farbpufferquelle für Pixel aus

ÜBERSICHT

`gl.ReadBuffer(mode)`

BESCHREIBUNG

`gl.ReadBuffer()` gibt einen Farbpuffer als Quelle für nachfolgende `gl.ReadPixels()`, `gl.CopyTexImage()`, `gl.CopyTexSubImage()` und `gl.CopyPixels()` Befehle an. `mode` akzeptiert einen von zwölf oder mehr vordefinierten Werten. `#GL_AUX0` bis `#GL_AUX3` sind immer definiert. In einem vollständig konfigurierten System benennen `#GL_FRONT`, `#GL_LEFT` und `#GL_FRONT_LEFT` alle den vorderen linken Puffer, `#GL_FRONT_RIGHT` und `#GL_RIGHT` den vorderen rechten Puffer und `#GL_BACK_LEFT` und `#GL_BACK` den hinteren linken Puffer.

Nicht stereoskopische Doppelpuffer-Konfigurationen haben nur einen vorderen linken und einen hinteren linken Puffer. Einzelgepufferte Konfigurationen haben einen vorderen linken und einen vorderen rechten Puffer bei Stereo und nur einen vorderen linken Puffer bei Nichtstereo. Es ist ein Fehler, einen nicht existierenden Puffer für `gl.ReadBuffer()` anzugeben.

`mode` ist zunächst `#GL_FRONT` in Einzel-Puffer-Konfigurationen und `#GL_BACK` in Doppel-Puffer-Konfigurationen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt einen Farbpuffer an (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `mode` nicht einer der zwölf (oder mehr) akzeptierten Werte ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `mode` einen Puffer angibt, der nicht existiert.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ReadBuffer()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_READ_BUFFER`

6.122 gl.ReadPixels

BEZEICHNUNG

gl.ReadPixels – liest einen Pixelblock aus dem Rahmenpuffer

ÜBERSICHT

```
pixelsArray = gl.ReadPixels(x, y, width, height, format)
```

BESCHREIBUNG

`gl.ReadPixels()` gibt Pixeldaten aus dem Rahmenpuffer zurück, beginnend mit dem Pixel, dessen linke untere Ecke sich an der Position (x, y) in einer Tabelle befindet. Mehrere Parameter steuern die Verarbeitung der Pixeldaten, bevor sie in die Tabelle gestellt werden. Diese Parameter werden mit drei Befehlen eingestellt: `gl.PixelStore()`, `gl.PixelTransfer()` und `gl.PixelMap()`. Diese Referenzseite beschreibt die Auswirkungen auf `gl.ReadPixels()` der meisten, aber nicht aller durch diese drei Befehle angegebenen Parameter.

`gl.ReadPixels()` gibt Werte von jedem Pixel mit der linken unteren Ecke bei (x + i, y + j) für 0 <= i < Breite und 0 <= j < Höhe zurück. Dieses Pixel soll das i-te Pixel in der j-ten Zeile sein. Die Pixel werden in Zeilenfolge von der niedrigsten bis zur höchsten Zeile zurückgegeben, von links nach rechts in jeder Zeile.

`gl.ReadPixels()` verwendet immer den Typ `#GL_FLOAT`, um die Pixel aus dem Rahmenpuffer zu lesen. Für eine fein abgestimmte Kontrolle über den Datentyp der Pixeldaten können Sie stattdessen `gl.ReadPixelsRaw()` verwenden. Siehe Abschnitt 6.123 [gl.ReadPixelsRaw], Seite 192, für Details.

`format` gibt das Format für die zurückgegebenen Pixelwerte an; akzeptierte Werte sind:

`#GL_COLOR_INDEX`
: Farbindizes werden aus dem durch gl.ReadBuffer() ausgewählten Farbpuffer gelesen. Jeder Index wird in einen Festpunkt umgewandelt, je nach Wert und Vorzeichen von `#GL_INDEX_SHIFT` nach links oder rechts verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_MAP_COLOR` `#GL_TRUE` ist, werden Indizes durch ihre Zuordnungen in der Tabelle `#GL_PIXEL_MAP_I_TO_I` ersetzt.

`#GL_STENCIL_INDEX`
: Schablonenwerte werden aus dem Schablonenpuffer gelesen. Jeder Index wird in einen Festpunkt umgewandelt, je nach Wert und Vorzeichen von `#GL_INDEX_SHIFT` nach links oder rechts verschoben und zu `#GL_INDEX_OFFSET` hinzugefügt. Wenn `#GL_MAP_STENCIL` `#GL_TRUE` ist, werden Indizes durch ihre Zuordnungen in der Tabelle `#GL_PIXEL_MAP_S_TO_S` ersetzt.

#GL_DEPTH_COMPONENT
: Tiefenwerte werden aus dem Tiefenpuffer gelesen. Jede Komponente wird in ein Gleitkomma umgewandelt, so dass der minimale Tiefenwert auf 0 und der maximale Wert auf 1 abgebildet wird. Jede Komponente wird dann mit #GL_DEPTH_SCALE multipliziert, zu #GL_DEPTH_BIAS hinzugefügt und schließlich in den Bereich (0,1) festgelegt.

#GL_RED
: Die Verarbeitung unterscheidet sich je nachdem, ob Farbpuffer Farbindizes oder RGBA-Farbkomponenten speichern. Wenn Farbindizes gespeichert sind, werden sie aus dem durch gl.ReadBuffer() ausgewählten Farbpuffer gelesen. Jeder Index wird in einen Festpunkt umgewandelt, je nach Wert und Vorzeichen von #GL_INDEX_SHIFT nach links oder rechts verschoben und zu #GL_INDEX_OFFSET hinzugefügt. Indizes werden dann durch die roten, grünen, blauen und alpha-Werte ersetzt, die durch die Indizierung der Tabellen #GL_PIXEL_MAP_I_TO_R, #GL_PIXEL_MAP_I_TO_G, #GL_PIXEL_MAP_I_TO_B und #GL_PIXEL_MAP_I_TO_A erhalten werden. Jede Tabelle muss die Größe 2^n haben, aber n kann für verschiedene Tabellen unterschiedlich sein. Bevor ein Index verwendet wird, um einen Wert in einer Tabelle der Größe 2^n nachzuschlagen, muss er gegen 2^n - 1 maskiert werden.

  Wenn RGBA-Farbkomponenten in den Farbpuffern gespeichert sind, werden sie aus dem durch gl.ReadBuffer() ausgewählten Farbpuffer gelesen. Jede Farbkomponente wird in ein Gleitkomma umgewandelt, so dass die Nullintensität auf 0.0 und die volle Intensität auf 1.0 abgebildet wird. Jede Komponente wird dann mit #GL_c_SCALE multipliziert und zu #GL_c_BIAS hinzugefügt, wobei c ROT, GRÜN, BLAU oder ALPHA ist. Wenn #GL_MAP_COLOR #GL_TRUE ist, wird schließlich jede Komponente auf den Bereich (0,1) festgelegt, auf die Größe der entsprechenden Tabelle skaliert und dann durch ihre Zuordnung in der Tabelle #GL_PIXEL_MAP_c_TO_c, wobei c R, G, B oder A ist.

  Nicht benötigte Daten werden dann verworfen. So entsorgt beispielsweise #GL_RED die Grün-, Blau- und Alpha-Komponenten, während #GL_RGB nur die Alpha-Komponente entsorgt. #GL_LUMINANCE berechnet einen Einkomponentenwert als Summe der roten, grünen und blauen Komponenten und #GL_LUMINANCE_ALPHA macht dasselbe, während Alpha als zweiter Wert beibehalten wird. Die Endwerte werden auf den Bereich (0,1) festgelegt.

#GL_GREEN
: Siehe oben in #GL_RED.

#GL_BLUE
: Siehe oben in #GL_RED.

#GL_ALPHA
: Siehe oben in #GL_RED.

#GL_RGB
: Siehe oben in #GL_RED.

#GL_RGBA
: Siehe oben in #GL_RED.

#GL_LUMINANCE
: Siehe oben in #GL_RED.

`#GL_LUMINANCE_ALPHA`
: Siehe oben in `#GL_RED`.

Die gerade beschriebenen Verschiebungs-, Skalierungs-, Verzerrungs- und Lookup-Faktoren werden alle durch `gl.PixelTransfer()` angegeben. Die Inhalte der Lookup-Tabelle selbst werden durch die Angabe von `gl.PixelMap()` angeben.

Die Rückgabewerte werden wie folgt in die Tabelle eingetragen. Wenn `format` `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX`, `#GL_DEPTH_COMPONENT`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA` oder `#GL_LUMINANCE` ist, wird ein einzelner Fließkommawert zurückgegeben. `#GL_RGB` gibt drei Werte zurück, `#GL_RGBA` gibt vier Werte zurück und `#GL_LUMINANCE_ALPHA` gibt zwei Werte für jedes Pixel zurück, wobei alle Werte, die einem einzelnen Pixel entsprechen, zusammenhängenden Platz in den Daten belegen. Speicherparameter werden eingestellt durch `gl.PixelStore()`, wie z.B. `#GL_PACK_LSB_FIRST` und `#GL_PACK_SWAP_BYTES`, beeinflussen die Art und Weise, wie Daten in den Speicher geschrieben werden. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

Werte für Pixel, die außerhalb des Fensters liegen, das mit dem aktuellen GL-Kontext verbunden ist, sind undefiniert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`	gibt die linke Koordinate eines rechteckigen Blocks von Pixeln an
`y`	gibt die untere Koordinate eines rechteckigen Blocks von Pixeln an
`width`	Breite des Pixelrechtecks
`height`	Höhe des Pixelrechtecks
`format`	Format der Pixeldaten (siehe oben)

RÜCKGABEWERTE

`pixelsArray`
: eine Tabelle mit den Pixeldaten

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `width` oder `height` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format` `#GL_COLOR_INDEX` ist und die Farbpuffer RGBA-Farbkomponenten speichern.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format` `#GL_STENCIL_INDEX` ist und es keinen Schablonenpuffer gibt.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format` `#GL_DEPTH_COMPONENT` ist und es keinen Tiefenpuffer gibt.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ReadPixels()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_INDEX_MODE`

6.123 gl.ReadPixelsRaw

BEZEICHNUNG

gl.ReadPixelsRaw – liest einen Pixelblock aus dem Rahmenpuffer

ÜBERSICHT

```
gl.ReadPixelsRaw(x, y, width, height, format, type, pixels)
```

BESCHREIBUNG

Dies entspricht `gl.ReadPixels()`, gibt aber die Pixel nicht in einer Tabelle zurück, sondern werden stattdessen direkt in einen Speicherblock geschrieben, der in `pixels` übergeben werden muss. Dies muss ein Speicherpuffer sein, der von dem Hollywood-Befehl `AllocMem()` zugewiesen und von `GetMemPointer()` zurückgegeben wird. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Siehe Abschnitt 6.122 [gl.ReadPixels], Seite 189, für eine Liste der unterstützten Typen für den Formatparameter.

Zusätzlich können Sie mit `gl.ReadPixelsRaw()` auch den Datentyp definieren, der beim Lesen von Pixeln aus dem Rahmenpuffer verwendet werden soll. `type` kann die folgenden Werte annehmen: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT`. `gl.ReadPixels()` verwendet immer `#GL_FLOAT`. Mit `gl.ReadPixelsRaw()` können Sie diesen Parameter an Ihre spezifischen Bedürfnisse anpassen.

Siehe Abschnitt 6.122 [gl.ReadPixels], Seite 189, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`	gibt die linke Koordinate eines rechteckigen Blocks von Pixeln an
`y`	gibt die untere Koordinate eines rechteckigen Blocks von Pixeln an
`width`	Breite des Pixelrechtecks
`height`	Höhe des Pixelrechtecks
`format`	Format der Pixeldaten (siehe oben)
`type`	gibt den Datentyp der Pixeldaten an (siehe oben)
`pixels`	Zeiger auf einen Speicherpuffer, um die Pixel in den Speicher zu schreiben

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` oder `type` kein akzeptierter Wert ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `width` oder `height` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format #GL_COLOR_INDEX` ist und die Farbpuffer RGBA-Farbkomponenten speichern.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format #GL_STENCIL_INDEX` ist und es keinen Schablonenpuffer gibt.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `format #GL_DEPTH_COMPONENT` ist und es keinen Tiefenpuffer gibt.

`#GL_INVALID_OPERATION` wird erzeugt, wenn gl.ReadPixels() zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN
`gl.Get()` mit dem Argument `#GL_INDEX_MODE`

6.124 gl.Rect

BEZEICHNUNG
gl.Rect – zeichnet ein Rechteck

ÜBERSICHT
```
gl.Rect(x1, y1, x2, y2)
```

BESCHREIBUNG
`gl.Rect()` unterstützt die effiziente Spezifikation von Rechtecken als zwei Eckpunkte. Jeder Rechteckbefehl nimmt vier Argumente an, die als zwei aufeinanderfolgende Paare von (x,y) Koordinaten organisiert sind. Alternativ können Sie auch zwei Tabellen mit jeweils (x,y) Koordinaten an `gl.Rect()` übergeben. Das resultierende Rechteck ist in der Ebene mit z = 0 definiert.

gl.Rect(x1, y1, x2, y2) ist genau gleichbedeutend mit der folgenden Sequenz:

```
gl.Begin(#GL_POLYGON)
gl.Vertex(x1, y1)
gl.Vertex(x2, y1)
gl.Vertex(x2, y2)
gl.Vertex(x1, y2)
gl.End()
```

Beachten Sie, dass wenn sich der zweite Knoten über und rechts vom ersten Knoten befindet, das Rechteck mit einer Wicklung gegen den Uhrzeigersinn konstruiert ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x1`	gibt einen Scheitelpunkt des Rechtecks an
`y1`	gibt einen Scheitelpunkt des Rechtecks an
`x2`	gibt den gegenüberliegenden Scheitelpunkt des Rechtecks an
`y2`	gibt den gegenüberliegenden Scheitelpunkt des Rechtecks an

FEHLER
`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Rect()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

6.125 gl.RenderMode

BEZEICHNUNG
gl.RenderMode – stellt den Rasterisierungsmodus ein

ÜBERSICHT
```
r = gl.RenderMode(mode)
```

BESCHREIBUNG

`gl.RenderMode()` setzt den Rasterisierungsmodus. Es braucht ein Argument, `mode`, das einen von drei vordefinierten Werten annehmen kann:

`#GL_RENDER`
: Rendermodus. Grundmuster werden gerastert und erzeugen Pixelfragmente, die in den Rahmenbuffer geschrieben werden. Dies ist der Normalmodus und auch der Standardmodus.

`#GL_SELECT`
: Auswahlmodus. Es werden keine Pixelfragmente erzeugt und es wird keine Änderung am Inhalt des Rahmenpuffers vorgenommen. Stattdessen wird ein Datensatz der Namen von Grundmustern, die gezeichnet worden wären, wenn der Rendermodus `#GL_RENDER` gewesen wäre, in einem Auswahl-Puffer zurückgegeben, der vor dem Eintritt in den Auswahlmodus erstellt werden muss. Siehe Abschnitt 6.129 [gl.SelectBuffer], Seite 198, für Details.

`#GL_FEEDBACK`
: Rückmelde-Modus. Es werden keine Pixelfragmente erzeugt und es wird keine Änderung am Inhalt des Rahmenpuffers vorgenommen. Stattdessen werden die Koordinaten und Attribute von Knoten, die gezeichnet worden wären, wenn der Rendermodus `#GL_RENDER` gewesen wäre, in einem Rückmelde-Puffer zurückgegeben, der erstellt werden muss, bevor der Rückmelde-Modus aufgerufen wird. Siehe Abschnitt 6.47 [gl.FeedbackBuffer], Seite 80, für Details.

Der Rückgabewert von `gl.RenderMode()` wird durch den Rendermodus beim Aufruf von `gl.RenderMode()` und nicht durch `mode` bestimmt. Die zurückgegebenen Werte für die drei Rendermodi lauten wie folgt:

`#GL_RENDER`
: 0.

`#GL_SELECT`
: Die Anzahl der Treffer, die in den Auswahl-Puffer übertragen werden.

`#GL_FEEDBACK`
: Die Anzahl der Werte (nicht Eckpunkte), die in den Rückmelde-Puffer übertragen werden.

Siehe Abschnitt 6.129 [gl.SelectBuffer], Seite 198, für weitere Details zur Auswahloperation.

Siehe Abschnitt 6.47 [gl.FeedbackBuffer], Seite 80, für weitere Details zum Rückmelde-Betrieb.

Wenn ein Fehler erzeugt wird, gibt `gl.RenderMode()` 0 zurück, unabhängig vom aktuellen Rendermodus.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt den Rasterisierungsmodus an; der Initialwert ist `#GL_RENDER` (siehe oben).

RÜCKGABEWERTE

`r` Rückgabewert (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `mode` nicht einer der drei akzeptierten Werte ist.

`#GL_INVALID_OPERATION` wird generiert, wenn `gl.SelectBuffer()` aufgerufen wird, während der Render-Modus `#GL_SELECT` ist, oder wenn `gl.RenderMode()` mit dem Argument `#GL_SELECT` aufgerufen wird bevor `gl.SelectBuffer()` mindestens einmal aufgerufen wurde.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.SelectBuffer()` aufgerufen wird, während der Rendermodus `#GL_SELECT` ist, oder wenn `gl.RenderMode()` mit dem Argument `#GL_SELECT` aufgerufen wird, bevor `gl.SelectBuffer()` mindestens einmal aufgerufen wurde.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.FeedbackBuffer()` aufgerufen wird, während der Rendermodus `#GL_FEEDBACK` ist, oder wenn `gl.RenderMode()` mit dem Argument `#GL_FEEDBACK` aufgerufen wird, bevor `gl.FeedbackBuffer()` mindestens einmal aufgerufen wurde.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.RenderMode()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_RENDER_MODE`

6.126 gl.Rotate

BEZEICHNUNG

gl.Rotate – multipliziert die aktuelle Matrix mit einer Rotationsmatrix

ÜBERSICHT

```
gl.Rotate(angle, x, y, z)
```

BESCHREIBUNG

`gl.Rotate()` erzeugt eine Drehung der Winkelgrade um den Vektor (x,y,z). Die aktuelle Matrix (siehe `gl.MatrixMode()`) wird mit einer Rotationsmatrix multipliziert, wobei das Produkt die aktuelle Matrix ersetzt.

Wenn der Matrixmodus entweder `#GL_MODELVIEW` oder `#GL_PROJECTION` ist, werden alle nach dem Aufruf von `gl.Rotate()` gezeichneten Objekte gedreht.

Verwenden Sie `gl.PushMatrix()` und `gl.PopMatrix()`, um das nicht skalierte Koordinatensystem zu speichern und wiederherzustellen.

Diese Drehung folgt der Rechtshandregel, d.h. wenn der Vektor (x,y,z) auf den Benutzer zeigt, erfolgt die Drehung gegen den Uhrzeigersinn.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`angle` gibt den Drehwinkel in Grad an

`x` gibt die x-Koordinate eines Vektors an

y	gibt die y-Koordinate eines Vektors an
z	gibt die z-Koordinate eines Vektors an

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Rotate()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.127 gl.Scale

BEZEICHNUNG

gl.Scale – multipliziert die aktuelle Matrix mit einer allgemeinen Skalierungsmatrix

ÜBERSICHT

```
gl.Scale(x, y, z)
```

BESCHREIBUNG

`gl.Scale()` erzeugt eine ungleichmäßige Skalierung entlang der x-, y- und z-Achse. Die drei Parameter geben den gewünschten Skalierungsfaktor entlang jeder der drei Achsen an.

Die aktuelle Matrix (siehe `gl.MatrixMode()`) wird mit dieser Skalenmatrix multipliziert und das Produkt ersetzt die aktuelle Matrix, als würde `gl.MultMatrix()` mit der folgenden Matrix als Argument aufgerufen:

```
x 0 0 0
0 y 0 0
0 0 z 0
0 0 0 1
```

Wenn der Matrixmodus entweder `#GL_MODELVIEW` oder `#GL_PROJECTION` ist, werden alle nach dem Aufruf von `gl.Scale()` gezeichneten Objekte skaliert.

Verwenden Sie `gl.PushMatrix()` und `gl.PopMatrix()`, um das nicht skalierte Koordinatensystem zu speichern und wiederherzustellen.

Wenn andere Skalierungsfaktoren als 1 auf die Modellsichtmatrix angewendet werden und die Beleuchtung aktiviert ist, erscheint die Beleuchtung oft falsch. In diesem Fall aktivieren Sie die automatische Normalisierung von Normalen, indem Sie `gl.Enable()` mit dem Argument `#GL_NORMALIZE` aufrufen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

x	legt den Skalierungsfaktor entlang der x-Achse fest
y	legt den Skalierungsfaktor entlang der y-Achse fest

`z` legt den Skalierungsfaktor entlang der z-Achse fest

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Scale()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.128 gl.Scissor

BEZEICHNUNG

gl.Scissor – definiert die Scherenbox

ÜBERSICHT

```
gl.Scissor(x, y, width, height)
```

BESCHREIBUNG

`gl.Scissor()` definiert ein Rechteck, die so genannte Scherenbox, in Fensterkoordinaten. Die ersten beiden Argumente, `x` und `y`, geben die linke untere Ecke des Feldes an. `width` und `height` spezifizieren die Breite und Höhe der Box.

Um den Scherentest zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit Argument `#GL_SCISSOR_TEST` auf. Der Test ist zunächst deaktiviert. Während der Test aktiviert ist, können nur Pixel, die sich in der Scherenbox befinden, durch Zeichnungsbefehle verändert werden. Fensterkoordinaten haben ganzzahlige Werte an den gemeinsamen Ecken von Rahmenpufferpixeln. gl.Scissor(0,0,1,1) erlaubt die Änderung nur des unteren linken Pixels im Fenster und gl.Scissor(0,0,0,0) erlaubt keine Änderung von Pixeln im Fenster.

Wenn der Scherentest deaktiviert ist, ist es, als ob die Scherenbox das gesamte Fenster umfasst.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x` gibt die linke Ecke der Scherenbox an; zunächst 0

`y` gibt die untere Ecke der Scherenbox an; zunächst 0

`width` gibt die Breite der Scherenbox an

`height` gibt die Höhe der Scherenbox an

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `width` oder `height` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Scissor()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_SCISSOR_BOX`

`gl.IsEnabled()` mit dem Argument `#GL_SCISSOR_TEST`

6.129 gl.SelectBuffer

BEZEICHNUNG

gl.SelectBuffer – richtet einen Puffer für die Werte des Auswahlmodus ein

ÜBERSICHT

```
buffer = gl.SelectBuffer(size)
```

BESCHREIBUNG

`gl.SelectBuffer()` weist einen Speicherpuffer der im Argument `size` angegebenen Größe zu und gibt einen Zeiger auf diesen Puffer zurück. Werte aus dem Namensstapel werden in diesen Puffer geschrieben (siehe `gl.InitNames()`, `gl.LoadName()`, `gl.PushName()`), wenn der Rendering-Modus `#GL_SELECT` ist. Siehe Abschnitt 6.125 [gl.RenderMode], Seite 193, für Details. `gl.SelectBuffer()` muss ausgegeben werden, bevor der Auswahlmodus aktiviert wird, und darf nicht ausgegeben werden, wenn der Rendering-Modus `#GL_SELECT` aktiviert ist.

Sie können Werte aus dem von `gl.SelectBuffer()` zurückgegebenen Speicherpuffer lesen, indem Sie `gl.GetSelectBuffer()` aufrufen, oder den direkten Zugriff auf den Puffer mit den Hollywood-Befehlen `GetMemPointer()` und `Peek()` vornehmen. Beim direkten Zugriff auf den Puffer ist zu beachten, dass die ersten vier Bytes im Puffer die Größe des Selektionspuffers in Bytes enthalten.

Ein Programmierer kann mit Hilfe der Auswahl bestimmen, welche Grundelemente in einen Bereich eines Fensters gezeichnet werden. Dieser Bereich wird durch die aktuelle Modellansicht und perspektivische Matrizen definiert.

Im Auswahlmodus werden keine Pixelfragmente aus der Rasterung erzeugt. Wenn stattdessen ein Grundelement oder eine Rasterposition das durch den Betrachtungsrahmen und die benutzerdefinierten Ausschnittebene definierte Ausschnitt-Volumen schneidet, führt dieses Grundelement zu einem Selektionstreffer (bei Polygonen tritt kein Treffer auf, wenn das Polygon ausgewählt wird). Wenn eine Änderung am Namensstapel vorgenommen oder wenn `gl.RenderMode()` aufgerufen wird, wird ein Trefferprotokoll in den Puffer kopiert, wenn seit dem letzten solchen Ereignis Treffer aufgetreten sind (Namensstapeländerung oder `gl.RenderMode()` Aufruf). Der Trefferdatensatz besteht aus der Anzahl der Namen im Namensstapel zum Zeitpunkt des Ereignisses, gefolgt von den minimalen und maximalen Tiefenwerten aller Knoten, die seit dem vorherigen Ereignis getroffen wurden, gefolgt von den Inhalten des Namensstapels, zuerst der untere Name.

Tiefenwerte (die im Bereich von [0,1] liegen) werden mit 2^32 - 1 multipliziert, bevor sie in das Trefferprotokoll aufgenommen werden.

Ein interner Index im Puffer wird bei jedem Aufruf des Auswahlmodus auf 0 zurückgesetzt. Jedes Mal, wenn ein Trefferdatensatz in den Puffer kopiert wird, erhöht sich der Index, um auf die Zelle unmittelbar nach dem Ende des Namensblocks zu verweisen, d.h. in die nächste verfügbare Zelle, wenn der Trefferdatensatz größer als die Anzahl der verbleibenden Positionen im Puffer ist und es werden so viele Daten

wie möglich kopiert sowie das Überlauf-Flag gesetzt. Wenn der Namensstapel beim Kopieren eines Treffer-Datensatzes leer ist, besteht dieser Datensatz aus 0, gefolgt von den minimalen und maximalen Tiefenwerten.

Rufen Sie zum Verlassen des Auswahlmodus `gl.RenderMode()` mit einem anderen Argument als `#GL_SELECT` auf. Immer wenn `gl.RenderMode()` aufgerufen wird, während der Rendermodus `#GL_SELECT` ist, wird die Anzahl der in den Puffer kopierten Treffer zurückgegeben, das Überlauf-Flag und der Auswahlpufferzeiger zurückgesetzt und initialisiert den Namensstapel als leer. Wenn das Überlauf-Flag gesetzt war, als `gl.RenderMode()` aufgerufen wurde, wird eine negative Trefferanzahl zurückgegeben.

Der Inhalt des Puffers ist undefiniert, bis `gl.RenderMode()` mit einem anderen Argument als `#GL_SELECT` aufgerufen wird.

`gl.Begin()` / `gl.End()` Grundelemente und Aufrufe von `gl.RasterPos()` können zu Treffern führen.

Um einen von dieser Funktion zugewiesenen Puffer freizugeben, rufen Sie den Befehl `gl.FreeSelectBuffer()` auf. Siehe Abschnitt 6.52 [gl.FreeSelectBuffer], Seite 86, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`size` gibt die Größe des Puffers in Bytes an

RÜCKGABEWERTE

`buffer` Speicherpuffer zur Verwendung im Auswahlmodus

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `size` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.SelectBuffer()` aufgerufen wird, während der Rendermodus `#GL_SELECT` ist, oder wenn `gl.RenderMode()` mit dem Argument `#GL_SELECT` aufgerufen wird, bevor `gl.SelectBuffer()` mindestens einmal aufgerufen wird.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.SelectBuffer()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_NAME_STACK_DEPTH`

`gl.Get()` mit dem Argument `#GL_SELECTION_BUFFER_SIZE`

`gl.GetPointer()` mit dem Argument `#GL_SELECTION_BUFFER_POINTER`

6.130 gl.ShadeModel

BEZEICHNUNG

gl.ShadeModel – wählt eine Einfach- oder eine Farbverlauf-Schattierung aus

ÜBERSICHT

```
gl.ShadeModel(mode)
```

BESCHREIBUNG

GL-Grundmuster können entweder eine Einfach- (Flat) oder eine Farbverlauf-Schattierung (Smooth) aufweisen. Die standardmäßige Farbverlauf-Schattierung bewirkt, dass die berechneten Farben von Knoten interpoliert werden, während das Grundmuster gerastert wird, wobei jedem resultierenden Pixelfragment typischerweise unterschiedliche Farben zugewiesen werden. Einfach-Schattierung wählt die berechnete Farbe von nur einem Knoten aus und ordnet sie allen Pixelfragmenten zu, die durch das Rastern eines einzelnen Grundmusters erzeugt werden. In beiden Fällen ist die berechnete Farbe eines Knoten das Ergebnis der Beleuchtung, wenn die Beleuchtung aktiviert ist, oder es ist die aktuelle Farbe zum Zeitpunkt der Angabe des Knoten, wenn die Beleuchtung deaktiviert ist.

Einfach- (Flat) und Farbverlauf-Schattierung (Smooth) sind für Punkte nicht zu erkennen. Beginnend mit `gl.Begin()` werden Knoten und Grundmuster von 1 gezählt und ausgegeben. GL gibt jedem einfach schattierten Liniensegment i die berechnete Farbe von Knoten i + 1, seinem zweiten Knoten. Mit ähnlicher Zählung von 1 gibt GL jedem einfach schattierten Polygon die berechnete Farbe des in der folgenden Tabelle aufgeführten Scheitels. Dies ist der letzte Knoten, an dem das Polygon in allen Fällen außer einzelnen Polygonen angegeben wird, wobei der erste Knoten die einfach schattierte Farbe angibt.

```
Grundmuster Polygontyp i     |  Vertex
-------------------------------------
Einfaches Polygon (i == 1)   |     1
Dreiecksstreifen             |   i + 2
Dreieckslüfter               |   i + 2
Unabhängiges Dreieck         |    3i
Vierfachstreifen             |  2i + 2
Unabhängiges Viereck         |    4i
```

Einfach- (Flat) und Farbverlauf-Schattierung (Smooth) werden von `gl.ShadeModel()` angegeben, wobei der Modus auf `#GL_FLAT` bzw. `#GL_SMOOTH` gesetzt ist.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mode` gibt einen symbolischen Wert an, der eine Schattierungstechnik repräsentiert; akzeptierte Werte sind `#GL_FLAT` und `#GL_SMOOTH`; der Initialwert ist `#GL_SMOOTH`.

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `mode` ein anderer Wert als `#GL_FLAT` oder `#GL_SMOOTH` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.ShadeModel()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_SHADE_MODEL`

6.131 gl.StencilFunc

BEZEICHNUNG

gl.StencilFunc – stellt die Funktion und den Referenzwert für den Schablonentest ein

ÜBERSICHT

```
gl.StencilFunc(func, ref, mask)
```

BESCHREIBUNG

Schablonieren, wie z.B. Tiefenpufferung, aktiviert und deaktiviert das Zeichnen auf Pro-Pixel-Basis. Schablonenebenen werden zunächst mit Hilfe von GL-Zeichnungsgrundmuster gezeichnet, dann werden Geometrie und Bilder mit Hilfe der Schablonenebenen gerendert, um Teile des Bildschirms auszublenden. Schablonieren wird typischerweise in Multipass-Rendering-Algorithmen verwendet, um Spezialeffekte wie Dekor, Konturen und konstruktive Solide-Geometrie-Rendering zu erzielen.

Der Schablonentest eliminiert bedingt ein Pixel basierend auf dem Ergebnis eines Vergleichs zwischen dem Referenzwert und dem Wert im Schablonenpuffer. Um den Test zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_STENCIL_TEST` auf. Um Aktionen basierend auf dem Ergebnis des Schablonentests festzulegen, rufen Sie den Befehl `gl.StencilOp()` auf.

`func` ist eine symbolische Konstante, die die Vergleichsfunktion der Schablone bestimmt. `func` akzeptiert einen von acht Werten, die in der folgenden Liste dargestellt wird. `ref` ist ein ganzzahliger Referenzwert, der im Schablonenvergleich verwendet wird. Er wird auf den Bereich (0,2^n-1) festgelegt, wobei n die Anzahl der Bitplanes im Schablonenpuffer sind. `mask` wird bitweise mit dem Referenzwert und dem Schablonen AND-Wert gespeichert, wobei die AND-Werte am Vergleich teilnehmen.

Wenn `stencil` den Wert repräsentiert, der in der entsprechenden Schablonenpufferposition gespeichert ist, zeigt die folgende Liste die Wirkung jeder Vergleichsfunktion, die durch `func` angegeben werden kann. Nur wenn der Vergleich gelingt, wird der Pixel an die nächste Stufe des Rasterisierungsprozesses übergeben. Siehe Abschnitt 6.133 [gl.StencilOp], Seite 203, für Details. Alle Tests behandeln Schablonenwerte als vorzeichenlose ganze Zahlen im Bereich (0,2^n-1), wobei n die Anzahl der Bitebenen im Schablonenpuffer ist.

Die folgenden Werte werden von `func` akzeptiert:

`#GL_NEVER`
: Fällt immer aus.

`#GL_LESS`
: Wird übergeben, wenn (ref & mask) < (stencil & mask) ist.

`#GL_LEQUAL`
: Wird übergeben, wenn (ref & mask) <= (Stencil & mask) ist.

`#GL_GREATER`
: Wird übergeben, wenn (ref & mask) > (stencil & mask) ist.

`#GL_GEQUAL`
: Wird übergeben, wenn (ref & mask) >= (stencil & mask) ist.

`GL_EQUAL`
: Wird übergeben, wenn (ref & mask) = (stencil & mask) ist.

`GL_NOTEQUAL`
: Wird übergeben, wenn (ref & mask) != (stencil & mask) ist.

`#GL_ALWAYS`
: Besteht immer.

Zunächst ist der Schablonentest deaktiviert. Wenn es keinen Schablonenpuffer gibt, kann keine Schablonenmodifikation stattfinden und es ist, als ob der Schablonentest immer bestanden wird.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`func` gibt die Testfunktion an; der Initialwert ist `#GL_ALWAYS` (siehe oben)

`ref` gibt den Referenzwert für den Schablonentest an; der Anfangswert ist 0

`mask` gibt eine Maske an, die sowohl mit dem Referenzwert als auch mit dem gespeicherten Schablonenwert AND-verknüpft ist, wenn der Test abgeschlossen ist; der Anfangswert ist bei allen 1's

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `func` nicht einer der acht akzeptierten Werte ist

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.StencilFunc()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_STENCIL_FUNC`, `#GL_STENCIL_VALUE_MASK`, `#GL_STENCIL_REF` oder `#GL_STENCIL_BITS`

`gl.IsEnabled()` mit dem Argument `#GL_STENCIL_TEST`

6.132 gl.StencilMask

BEZEICHNUNG

gl.StencilMask – steuert das Schreiben einzelner Bits in den Schablonenebenen

ÜBERSICHT

```
gl.StencilMask(mask)
```

BESCHREIBUNG

`gl.StencilMask()` steuert das Schreiben einzelner Bits in den Schablonenebenen. Die niederwertigsten n Bits der Maske, wobei n die Anzahl der Bits im Schablonenpuffer ist, geben eine Maske an. Wenn eine 1 in der Maske erscheint, ist es möglich, in das entsprechende Bit im Schablonenpuffer zu schreiben. Erscheint eine 0, ist das entsprechende Bit schreibgeschützt. Zunächst sind alle Bits zum Schreiben freigegeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`mask` gibt eine Bitmaske an, um das Schreiben einzelner Bits in den Schablonenebenen zu aktivieren und zu deaktivieren; zunächst ist die Maske bei allen 1's

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.StencilMask()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_STENCIL_WRITEMASK`, `#GL_STENCIL_BACK_WRITEMASK`, oder `#GL_STENCIL_BITS`

6.133 gl.StencilOp

BEZEICHNUNG

gl.StencilOp – legt Schablonentest-Aktionen fest

ÜBERSICHT

```
gl.StencilOp(fail, zfail, zpass)
```

BESCHREIBUNG

Schablonieren, wie z.B. Tiefenpufferung, aktiviert und deaktiviert das Zeichnen auf Pro-Pixel-Basis. Sie zeichnen mit GL-Zeichnungsgrundmustern in die Schablonenebenen, rendern dann Geometrie und Bilder und verwenden die Schablonenebenen, um Teile des Bildschirms auszublenden. Schablonieren wird typischerweise in Multipass-Rendering-Algorithmen verwendet, um Spezialeffekte wie Dekor, Konturen und konstruktive Solide-Geometrie-Rendering zu erzielen.

Der Schablonentest eliminiert bedingt ein Pixel basierend auf dem Ergebnis eines Vergleichs zwischen dem Wert im Schablonenpuffer und einem Referenzwert. Um den Test zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_STENCIL_TEST` auf; um es zu steuern, rufen Sie den Befehl `gl.StencilFunc()` auf.

`gl.StencilOp()` nimmt drei Argumente an, die anzeigen, was mit dem gespeicherten Schablonenwert passiert, während das Schablonieren aktiviert ist. Wenn der Schablonentest fehlschlägt, wird keine Änderung an den Farb- oder Tiefenpuffern des Pixels vorgenommen und `fail` gibt an, was mit dem Inhalt des Schablonenpuffer passiert. Die folgenden acht Aktionen sind möglich.

`#GL_KEEP` Behält den aktuellen Wert bei.

`#GL_ZERO` Setzt den Wert des Schablonenpuffers auf 0.

`#GL_REPLACE`
: Setzt den Wert des Schablonenpuffers auf ref, wie durch `gl.StencilFunc()` angegeben.

`#GL_INCR` Erhöht den aktuellen Schablonenpufferwert. Festgelegt auf den maximal darstellbaren vorzeichenlosen Wert.

`#GL_DECR` Verringert den aktuellen Schablonenpufferwert. Festgelegt auf 0.

`#GL_INVERT`
: Bitweise wird der aktuelle Schablonenpufferwert invertiert.

Schablonenpufferwerte werden als vorzeichenlose Ganzzahlen behandelt. Beim Inkrementieren und Dekrementieren werden die Werte auf 0 und 2^n - 1 festgelegt, wobei n der Wert ist, der bei der Abfrage von `#GL_STENCIL_BITS` zurückgegeben wird.

Die beiden anderen Argumente zu `gl.StencilOp()` spezifizieren Schablonenpufferaktionen, die davon abhängen, ob nachfolgende Tiefenpuffer-Tests erfolgreich (`zpass`) oder fehlgeschlagen (`zfail`) sind. Siehe Abschnitt 6.28 [gl.DepthFunc], Seite 54, für Details. Die Aktionen werden mit den gleichen acht symbolischen Konstanten wie `fail` angegeben. Beachten Sie, dass `zfail` ignoriert wird, wenn es keinen Tiefenpuffer gibt oder wenn der Tiefenpuffer nicht aktiviert ist. In diesen Fällen geben `fail` und `zpass` die Schablonenaktion an, wenn der Schablonentest fehlschlägt bzw. besteht.

Zunächst ist der Schablonentest deaktiviert. Wenn es keinen Schablonenpuffer gibt, kann keine Schablonenmodifikation stattfinden und es ist, als ob der Schablonentest immer bestanden würde, unabhängig von einem Aufruf von `gl.StencilOp()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`fail` gibt die Aktion an, die durchgeführt werden soll, wenn der Schablonentest fehlschlägt; der Initialwert ist `#GL_KEEP` (siehe oben)

`zfail` gibt die Schablonenaktion an, wenn der Schablonentest bestanden wird, aber der Tiefenpuffer-Test fehlschlägt; `zfail` akzeptiert die gleichen symbolischen Konstanten wie `fail`; der Anfangswert ist `#GL_KEEP`

`zpass` gibt die Schablonenaktion an, wenn sowohl der Schablonentest als auch die Tiefenpufferprüfung bestanden wird, oder wenn der Schablonentest bestanden wird und entweder kein Tiefenpuffer oder keine Tiefenpufferprüfung aktiviert ist; `zpass` akzeptiert die gleichen symbolischen Konstanten wie `fail`; der Anfangswert ist `#GL_KEEP`

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `fail`, `zfail` oder `zpass` ein anderer Wert als die acht definierten konstanten Werte ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.StencilOp()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit einem der folgenden Argumente: `#GL_STENCIL_FAIL`, `#GL_STENCIL_PASS_DEPTH_PASS`, `#GL_STENCIL_PASS_DEPTH_FAIL`, `#GL_STENCIL_BACK_FAIL` oder `#GL_STENCIL_BITS`

`gl.IsEnabled()` mit dem Argument `#GL_STENCIL_TEST`

6.134 gl.TexCoord

BEZEICHNUNG

gl.TexCoord – setzt die aktuellen Texturkoordinaten

ÜBERSICHT

```
gl.TexCoord(s[, t, r, q])
```

BESCHREIBUNG

`gl.TexCoord()` gibt Texturkoordinaten in einer, zwei, drei oder vier Dimension an.

Die aktuellen Texturkoordinaten sind Teil der Daten, die jedem Knoten und der aktuellen Rasterposition zugeordnet sind. Zunächst sind die Werte für s, t, r und q (0, 0, 0, 1).

Die aktuellen Texturkoordinaten können jederzeit aktualisiert werden. Insbesondere kann `gl.TexCoord()` zwischen einem Aufruf von `gl.Begin()` und `gl.End()` aufgerufen werden.

Alternativ können Sie auch eine Tabelle mit ein bis vier Koordinaten an `gl.TexCoord()` übergeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`s`	gibt die Texturkoordinate s an
`t`	optional: Gibt die Koordinate der t-Textur an (Standardwert ist 0)
`r`	optional: Gibt die Koordinate der r-Textur an (Standardwert ist 0)
`q`	optional: Gibt die q-Texturkoordinate an (Standard ist 1)

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_CURRENT_TEXTURE_COORDS`

6.135 gl.TexCoordPointer

BEZEICHNUNG

gl.TexCoordPointer – definiert ein Feld von Texturkoordinaten

ÜBERSICHT

```
gl.TexCoordPointer(vArray[, size])
```

BESCHREIBUNG

`gl.TexCoordPointer()` gibt ein Feld von Texturkoordinaten an, die beim Rendern verwendet werden sollen. `vArray` kann entweder eine eindimensionale Tabelle sein, die aus einer beliebigen Anzahl von aufeinanderfolgenden Texturkoordinaten besteht, oder eine zweidimensionale Tabelle, die aus einer beliebigen Anzahl von Untertabellen besteht, die jeweils 1 bis 4 Texturkoordinaten enthalten. Wenn `vArray` eine eindimensionale Tabelle ist, müssen Sie auch das optionale Argument `size` übergeben, um die Anzahl der Texturkoordinaten pro Feld-Element zu definieren. Wenn `vArray` eine zweidimensionale Tabelle ist, wird `size` automatisch durch die Anzahl der Elemente in der ersten Untertabelle bestimmt, die ebenfalls im Bereich von 1 bis 4 liegen müssen.

Bei der Verwendung einer zweidimensionalen Tabelle ist zu beachten, dass die Anzahl der Texturkoordinaten in jeder Untertabelle konstant sein muss. Es ist nicht erlaubt, in den einzelnen Untertabellen eine unterschiedliche Anzahl von Texturkoordinaten zu verwenden. Die Anzahl der Texturkoordinaten wird durch die Anzahl der Elemente in der ersten Untertabelle definiert und alle folgenden Untertabellen müssen die gleiche Anzahl von Koordinaten verwenden.

Wenn Sie `Nil` in `vArray` übergeben, wird der Inhalt des Texturkoordinaten-Feld-Puffers gelöscht, aber er wird nicht aus OpenGL entfernt. Dies muss manuell erfolgen, z.B. durch Deaktivierung des Texturkoordinaten-Feldes oder durch Definition eines neuen Feldes.

Um ein Texturkoordinaten-Feld zu aktivieren und zu deaktivieren, rufen Sie den Befehl `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY` auf. Wenn aktiviert, wird das Texturkoordinaten-Feld verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Jedes Texturkoordinaten-Feld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.TexCoordPointer()` ist zwischen der Ausführung von `gl.Begin()` und `gl.End()` nicht erlaubt. Es kann ein Fehler auftreten oder auch nicht. Wenn kein Fehler erzeugt wird, ist die Operation undefiniert.

`gl.TexCoordPointer()` wird typischerweise auf der Klient-Seite implementiert.

Texturkoordinaten-Feld-Parameter sind klientseitig und werden daher nicht durch `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt. Benutzen Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`vArray` ein- oder zweidimensionale Tabelle mit Texturkoordinaten oder `Nil` (siehe oben)

`size` optional: Texturkoordinaten pro Feld-Element; muss zwischen 1 und 4 liegen und wird nur bei eindimensionalen Tabellen (siehe oben) verwendet

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `size` nicht 1, 2, 3 oder 4 ist.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY`

`gl.Get()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY_SIZE`

`gl.Get()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY_TYPE`

`gl.Get()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY_STRIDE`

`gl.GetPointer()` mit dem Argument `#GL_TEXTURE_COORD_ARRAY_POINTER`

6.136 gl.TexEnv

BEZEICHNUNG

gl.TexEnv – stellt Textur-Umgebungsparameter ein

ÜBERSICHT

```
gl.TexEnv(pname, param)
```

BESCHREIBUNG

Eine Texturumgebung legt fest, wie Texturwerte interpretiert werden, wenn ein Fragment texturiert wird. `pname` kann entweder `#GL_TEXTURE_ENV_MODE` oder `#GL_TEXTURE_ENV_COLOR` sein. Wenn `pname` `#GL_TEXTURE_ENV_MODE` ist, dann muss `param` der symbolische

Name einer Texturfunktion sein. Es können vier Texturfunktionen angegeben werden: `#GL_MODULATE`, `#GL_DECAL`, `#GL_BLEND` und `#GL_REPLACE`.

Eine Texturfunktion wirkt auf das zu texturierende Fragment mit dem für das Fragment geltenden Texturbildwert (siehe `gl.TexParameter()`) und erzeugt eine RGBA-Farbe für dieses Fragment. In einem OpenGL-Referenzhandbuch finden Sie Informationen darüber, wie die RGBA-Farbe für jede der drei wählbaren Texturfunktionen erzeugt wird.

Wenn `pname #GL_TEXTURE_ENV_COLOR` ist, muss `param` eine Tabelle sein, die ein Feld enthält, das eine RGBA-Farbe enthält, die aus vier Fließkommawerten besteht.

`#GL_TEXTURE_ENV_MODE` ist standardmäßig auf `#GL_MODULATE` und `#GL_TEXTURE_ENV_COLOR` ist standardmäßig auf (0, 0, 0, 0) gesetzt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`pname`	gibt den symbolischen Namen eines Textur-Umgebungsparameters an (siehe oben)
`param`	Tabelle oder Einzelwert unter Angabe des Parameters

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` nicht einer der akzeptierten definierten Werte ist oder wenn `param` einen definierten konstanten Wert haben sollte (basierend auf dem Wert von `pname`) und dies nicht hat.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexEnv()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexEnv()`

6.137 gl.TexGen

BEZEICHNUNG

gl.TexGen – kontrolliert die Erzeugung von Texturkoordinaten

ÜBERSICHT

```
gl.TexGen(coord, pname, param)
```

BESCHREIBUNG

`gl.TexGen()` wählt eine Texturkoordinatengenerierungs-Funktion aus oder liefert Koeffizienten für eine der Funktionen. `coord` nennt eine der Texturkoordinaten (s, t, r, q); es muss eines der Symbole `#GL_S`, `#GL_T`, `#GL_R` oder `#GL_Q` sein. `pname` muss eine von drei symbolischen Konstanten sein: `#GL_TEXTURE_GEN_MODE`, `#GL_OBJECT_PLANE` oder `#GL_EYE_PLANE`. Wenn `pname #GL_TEXTURE_GEN_MODE` ist, dann wählt `param` eine der drei Modi `#GL_OBJECT_LINEAR`, `#GL_EYE_LINEAR` oder `#GL_SPHERE_MAP` aus. Wenn `pname` entweder `#GL_OBJECT_PLANE` oder `#GL_EYE_PLANE` ist, enthält `param` Koeffizienten für die entsprechende Texturierungsfunktion.

Wenn die Texturerzeugungsfunktion `#GL_OBJECT_LINEAR` ist, wird die Funktion

```
g = p1x0 + p2y0 + p3z0 + p4w0
```

verwendet, wobei g der berechnete Wert für die in `coord` genannte Koordinate ist, p1, p2, p3 und p4 die vier in `param` gelieferten Werte und x0, y0, z0 und w0 die Objektkoordinaten des Knoten sind. Dieser Befehl kann z.B. verwendet werden, um das Gelände mit Hilfe des Meeresspiegels als Bezugsebene (definiert durch p1, p2, p3 und p4) zu strukturieren. Die Höhe eines Geländepunktes wird durch die Koordinatengenerierung `#GL_OBJECT_LINEAR` als Abstand vom Meeresspiegel berechnet; diese Höhe kann dann verwendet werden, um das Texturbild zu indizieren, um weißen Schnee auf Gipfeln und grünes Gras auf dessen Ausläufern abzubilden.

Wenn die Texturerzeugungsfunktion `#GL_EYE_LINEAR` ist, wird die Funktion

```
g = p1'x0 + p2'y0 + p3'z0 + p4'w0
```

verwendet, wobei

```
(p1' p2' p3' p4') = ( p1 p2 p3 p4) M^-1
```

und x0, y0, z0 und w0 die Augenkoordinaten des Scheitelpunkts sind, p1, p2, p3 und p4 sind die in `param` angegebenen Werte und M ist die Modellansichtsmatrix, wenn `gl.TexGen()` aufgerufen wird. Wenn M schlecht konditioniert oder singulär ist, können die von der resultierenden Funktion erzeugten Texturkoordinaten ungenau oder undefiniert sein.

Beachten Sie, dass die Werte in `param` eine Bezugsebene in Augenkoordinaten definieren. Die Modellansichtsmatrix, die auf sie angewendet wird, ist möglicherweise nicht die gleiche, die bei der Transformation der Polygonknoten wirksam ist. Dieser Befehl erstellt ein Feld von Texturkoordinaten, das dynamische Konturlinien auf sich bewegenden Objekten erzeugen kann.

Wenn `pname` `#GL_SPHERE_MAP` und `coord` entweder `#GL_S` oder `#GL_T` ist, werden s und t Texturkoordinaten wie folgt generiert: u ist dann der Einheitenvektor, der vom Nullpunkt auf den Polygonknoten zeigt (in Augenkoordinaten). n' wird dann die aktuelle Normalität nach der Transformation in Augenkoordinaten sein. Dabei soll f = (fx fy fz)^T der Reflexionsvektor sein, so daß

```
f = u - 2n' n'^Tu
```

schliesslich m = 2 sqrt(fx^2 + fy^2 + (fz + 1)^2) ist. Dann werden die Werte für die Texturkoordinaten s und t wie folgt zugewiesen

```
s = fx/m + 1/2
t = fy/m + 1/2
```

Um eine Texturkoordinatengenerierungs-Funktion zu aktivieren oder zu deaktivieren, rufen Sie `gl.Enable()` oder `gl.Disable()` auf, mit einem der symbolischen Texturkoordinatennamen (`#GL_TEXTURE_GEN_S`, `#GL_TEXTURE_GEN_T`, `#GL_TEXTURE_GEN_R` oder `#GL_TEXTURE_GEN_Q`) als Argument. Wenn aktiviert, wird die angegebene Texturkoordinate gemäß der Generierungsfunktion berechnet, die dieser Koordinate zugeordnet ist. Wenn deaktiviert, übernehmen nachfolgende Eckpunkte die angegebene Texturkoordinate aus dem aktuellen Satz von Texturkoordinaten. Zunächst sind alle Funktionen zur Texturerzeugung auf `#GL_EYE_LINEAR` gesetzt und deaktiviert. Beide s-Ebenengleichungen sind (1, 0, 0, 0), beide t-Ebenengleichungen (0, 1, 0, 0) und alle r- und q-Ebenengleichungen (0, 0, 0, 0).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`coord` gibt eine Texturkoordinate an; muss eine der folgenden sein: `#GL_S`, `#GL_T`, `#GL_R` oder `#GL_Q`

`pname` gibt den symbolischen Namen der Texturkoordinaten-Erzeugungsfunktion oder der Funktionsparameter an (siehe oben)

`param` Einzelwert oder Tabelle mit Parametern für `pname` (siehe oben)

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `coord` oder `pname` kein akzeptierter definierter Wert ist oder wenn `pname` `#GL_TEXTURE_GEN_MODE` und `param` kein akzeptierter definierter Wert ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `pname` `#GL_TEXTURE_GEN_MODE`, `param` `#GL_SPHERE_MAP` ist und `coord` entweder `#GL_R` oder `#GL_Q` beinhaltet.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexGen()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexGen()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_GEN_S`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_GEN_T`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_GEN_R`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_GEN_Q`

6.138 gl.TexImage

BEZEICHNUNG

gl.TexImage – gibt ein ein- oder zweidimensionales Texturbild an

ÜBERSICHT

```
gl.TexImage(level, internalformat, format, type, pixels[, border])
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.TexImage1D()` und `gl.TexImage2D()`, mit der Ausnahme, dass die Pixeldaten nicht in einem Rohspeicherpuffer übergeben werden, sondern als Tabelle, die für jede Pixelreihe eine Untertabelle enthält. Dies ist natürlich nicht so effizient wie die Verwendung von Rohspeicherpuffern, da die Pixeldaten der Tabelle zuerst in einen Rohspeicherpuffer kopiert werden müssen.

Breite und Höhe der Textur werden automatisch durch das Layout der Tabelle in `pixels` bestimmt. Wenn es nur eine Untertabelle innerhalb von `pixels` gibt, definiert `gl.TexImage()` eine Textur vom Typ `#GL_TEXTURE_1D`. Wenn es mehrere Untertabellen innerhalb von `pixels` gibt, wird `#GL_TEXTURE_2D` verwendet.

Beachten Sie, dass nur `#GL_FLOAT` und `#GL_UNSIGNED_BYTE` derzeit für `type` unterstützt werden und `internalformat` nur `#GL_RGB`, `#GL_RGBA`, `#GL_ALPHA`, `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_DEPTH_COMPONENT` und die Werte 1, 2, 3 und 4 akzeptiert.

Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für weitere Details zu den von diesem Befehl akzeptierten Parametern.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level` gibt die Anzahl der Detailstufen an; Level 0 ist die Basis-Bildebene, Level n ist das n-te Mipmap-Reduktionsbild

`internalformat`
gibt die Anzahl der Farbkomponenten in der Textur an; muss 1, 2, 3 oder 4 oder eine symbolische Konstante sein (siehe oben)

`format` gibt das Format der Pixeldaten an (siehe oben)

`type` gibt den Datentyp der Pixeldaten an (siehe oben)

`pixels` spezifiziert eine zweidimensionale Tabelle an, die Pixeldaten enthält

`border` optional: gibt die Breite des Rahmens an (Standard ist 0)

6.139 gl.TexImage1D

BEZEICHNUNG

gl.TexImage1D – gibt ein eindimensionales Texturbild an

ÜBERSICHT

```
gl.TexImage1D(level, internalformat, width, border, format, type, pixels)
```

BESCHREIBUNG

Texturierung bildet einen Teil eines bestimmten Texturbildes auf jedes grafische Grundmuster ab, für das die Texturierung aktiviert ist. Um die eindimensionale Texturierung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_TEXTURE_1D` auf.

Texturbilder werden mit `gl.TexImage1D()` definiert. Die Argumente beschreiben die Parameter des Texturbildes, wie Breite, Breite des Rahmens, Anzahl der Detailstufen (siehe `gl.TexParameter()`), sowie die interne Auflösung und das Format, das zum Speichern des Bildes verwendet wird. Die letzten drei Argumente beschreiben, wie das Bild im Speicher dargestellt wird; sie sind identisch mit den Pixelformaten für `gl.DrawPixels()`.

Daten werden von `pixels` als eine Folge von vorzeichenbehafteten oder vorzeichenlose Bytes, Shorts oder Longs oder einfach präzisen Gleitkommawerten gelesen, je nach `type`. Diese Werte werden je nach `format` in Gruppen von einem, zwei, drei oder vier Werten zu Elementen zusammengefasst. Wenn der Typ `#GL_BITMAP` ist, werden die Daten als eine Zeichenkette von vorzeichenlose Bytes betrachtet (und das Format muss `#GL_COLOR_INDEX` sein). Jedes Datenbyte wird als acht 1-Bit-Elemente behandelt, wobei die Bitreihenfolge durch `#GL_UNPACK_LSB_FIRST` bestimmt wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

Das erste Element entspricht dem linken Ende des Texturfeldes. Nachfolgende Elemente gehen von links nach rechts durch die restlichen Texel im Textur-Feld. Das letzte Element entspricht dem rechten Ende des Texturfeldes.

`format` bestimmt die Zusammensetzung jedes Elements in Pixeln. Es kann einen von neun symbolischen Werten annehmen:

`#GL_COLOR_INDEX`
: Jedes Element ist ein Einzelwert, ein Farbindex. GL wandelt ihn in einen Festpunkt um (mit einer unbestimmten Anzahl von Nullbits rechts vom Binärpunkt), verschiebt ihn je nach Wert und Vorzeichen von `#GL_INDEX_SHIFT` nach links oder rechts und ergänzt ihn zu `#GL_INDEX_OFFSET` (siehe `gl.PixelTransfer()`). Der resultierende Index wird mit Hilfe der Tabellen `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B` und `#GL_PIXEL_MAP_I_TO_A` in eine Reihe von Farbkomponenten umgewandelt und in den Bereich [0,1] festgelegt.

`#GL_RED`
: Jedes Element ist eine einzelne rote Komponente. GL wandelt es in ein Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Grün sowie Blau und 1 zu Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_GREEN`
: Jedes Element ist eine einzelne grüne Komponente. GL wandelt es in ein Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot sowie Blau und 1 zu Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_BLUE`
: Jedes Element ist eine einzelne blaue Komponente. GL wandelt es in ein Fließkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot sowie Grün und 1 zu Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_ALPHA`
: Jedes Element ist eine einzelne Alpha-Komponente. GL wandelt es in ein Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot, Grün und Blau anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_RGB`
: Jedes Element ist ein dreifaches RGB. GL wandelt es in ein Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_RGBA`
: Jedes Element enthält alle vier Komponenten. Jede Komponente wird mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_LUMINANCE`
: Jedes Element ist ein einzelner Leuchtdichtewert. GL wandelt es in Fließkomma um und fügt es dann zu einem RGBA-Element zusammen, indem er den Leuchtdichtewert dreimal für Rot, Grün und Blau repliziert und 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_LUMINANCE_ALPHA`
: Jedes Element ist ein Leuchtdichte/Alpha-Paar. GL wandelt es in Fließkomma um und fügt es dann zu einem RGBA-Element zusammen, indem er den Leuchtdichtewert dreimal für Rot, Grün und Blau repliziert. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_DEPTH_COMPONENT`
: Jedes Element ist eine einzelne Tiefenkomponente. Es wird in Gleitkomma umgewandelt, dann mit dem vorzeichenbehafteten Skalierungsfaktor `#GL_DEPTH_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_DEPTH_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

Wenn eine Anwendung die Textur in einer bestimmten Auflösung oder in einem bestimmten Format speichern möchte, kann sie die Auflösung und das Format mit `internalformat` anfordern. `internalformat` gibt das interne Format des Textur-Feldes an. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. GL wird eine interne Darstellung wählen, die der vom `internalformat` geforderten genau entspricht, aber möglicherweise nicht genau übereinstimmt. (Die durch `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_RGB` und `#GL_RGBA` angegebenen Darstellungen müssen genau übereinstimmen. Die Zahlenwerte 1, 2, 3 und 4 können auch verwendet werden, um die obigen Darstellungen anzugeben.)

Ein Einkomponenten-Texturbild verwendet nur die Rotkomponente der aus Pixeln extrahierten RGBA-Farbe, ein Zweikomponentenbild das R und die A Werte, ein Dreikomponentenbild die Werte R, G und B und ein Vierkomponentenbild verwendet alle RGBA-Komponenten.

Die Texturierung hat im Farbindexmodus keine Wirkung.

Das Texturbild kann durch die gleichen Datenformate dargestellt werden wie die Pixel in einem `gl.DrawPixels()`, nur dass `#GL_STENCIL_INDEX` und `#GL_DEPTH_COMPONENT` nicht verwendet werden können. `gl.PixelStore()` und `gl.PixelTransfer()` Modi beeinflussen Texturbilder genau so, wie es `gl.DrawPixels()` beeinflusst.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 6.138 [gl.TexImage], Seite 209, für Details.

Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level` gibt die Detaillierungsstufe an; Level 0 ist die Basis-Bildebene, Ebene n ist das n-te Mipmap-Reduktionsbild

`internalformat`
gibt die Anzahl der Farbkomponenten in der Textur an; muss 1, 2, 3 oder 4 oder eine symbolische Konstante sein (siehe oben)

`width` gibt die Breite des Texturbildes an; muss 2^n + 2*border für eine ganze Zahl n sein; alle Implementierungen unterstützen Texturbilder, die mindestens 64 Texel breit sind

`border` gibt die Breite des Rahmens an; muss entweder 0 oder 1 sein

`format` gibt das Format der Pixeldaten an (siehe oben)

`type` gibt den Datentyp der Pixeldaten an (siehe oben)

`pixels` gibt einen Zeiger auf die Bilddaten im Speicher an

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` keine akzeptierte Formatkonstante ist. Andere Formatkonstanten als `#GL_STENCIL_INDEX` werden akzeptiert.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` keine Typkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` `#GL_BITMAP` und `format` nicht `#GL_COLOR_INDEX` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` größer als log2max ist, wobei max der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `internalformat` nicht 1, 2, 3, 4 oder eine der akzeptierten symbolischen Auflösungs- und Formatkonstanten ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `width` oder `height` kleiner als 0 oder größer als 2 + `#GL_MAX_TEXTURE_SIZE` ist oder wenn beide nicht als 2k + 2*Border für einen ganzzahligen Wert von k dargestellt werden können.

`#GL_INVALID_VALUE` wird erzeugt, wenn `border` nicht 0 oder 1 ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexImage1D()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_1D`

6.140 gl.TexImage2D

BEZEICHNUNG

gl.TexImage2D – gibt ein zweidimensionales Texturbild an

ÜBERSICHT

```
gl.TexImage2D(level, internalformat, w, h, border, format, type, pixels)
```

BESCHREIBUNG

Texturierungskarte bildet einen Teil eines bestimmten Texturbildes auf jedes grafische Grundmuster ab, für das die Texturierung aktiviert ist. Um die zweidimensionale Texturierung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_TEXTURE_2D` auf.

Um Texturbilder zu definieren, rufen Sie `gl.TexImage2D()` auf. Die Argumente beschreiben die Parameter des Texturbildes, wie Höhe, Breite, Breite des Rahmens, Detaillierungsstufe (siehe `gl.TexParameter()`) und Anzahl der bereitgestellten Farbkomponenten. Die letzten drei Argumente beschreiben, wie das Bild im Speicher dargestellt wird; sie sind identisch mit den für glDrawPixels verwendeten Pixelformaten.

Daten werden von `pixels` als eine Folge von vorzeichenbehafteten oder vorzeichenlose Bytes, Shorts oder Longs oder einfach präzisen Gleitkommawerten gelesen, abhängig vom `type`, der `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT`, `#GL_INT` und `#GL_FLOAT` sein kann. Diese Werte werden je nach `format` in Gruppen von einem, zwei, drei oder vier Werten zu Elementen zusammengefasst. Wenn der Typ `#GL_BITMAP` ist, werden die Daten als eine Zeichenkette von unsignierten Bytes betrachtet (`format` muss `#GL_COLOR_INDEX` sein). Jedes Datenbyte wird als acht 1-Bit-Elemente behandelt, wobei die Bitreihenfolge durch `#GL_UNPACK_LSB_FIRST` festgelegt wird. Siehe Abschnitt 6.104 [gl.PixelStore], Seite 163, für Details.

Das erste Element entspricht der linken unteren Ecke des Texturbildes. Nachfolgende Elemente gehen von links nach rechts durch die restlichen Texel in der untersten Zeile des Texturbildes und dann in den nacheinander höheren Zeilen des Texturbildes. Das letzte Element entspricht der oberen rechten Ecke des Texturbildes.

`format` bestimmt die Zusammensetzung jedes Elements in Pixeln. Es kann einen von neun symbolischen Werten annehmen:

`#GL_COLOR_INDEX`
: Jedes Element ist ein Einzelwert, ein Farbindex. GL wandelt ihn in einen Festpunkt um (mit einer unbestimmten Anzahl von Nullbits rechts vom Binärpunkt), verschiebt ihn je nach Wert und Vorzeichen von `#GL_INDEX_SHIFT` nach links oder rechts und ergänzt ihn zu `#GL_INDEX_OFFSET`. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details. Der resultierende Index wird mit Hilfe der Tabellen `#GL_PIXEL_MAP_I_TO_R`, `#GL_PIXEL_MAP_I_TO_G`, `#GL_PIXEL_MAP_I_TO_B` und `#GL_PIXEL_MAP_I_TO_A` in eine Reihe von Farbkomponenten umgewandelt und in den Bereich [0,1] festgelegt.

`#GL_RED`
: Jedes Element ist eine einzelne rote Komponente. GL wandelt es in Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Grün sowie Blau und 1 für Alpha anfügt. Jede Komponente wird dann mit

dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_GREEN`

Jedes Element ist eine einzelne grüne Komponente. GL wandelt es in Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot sowie Blau und 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_BLUE` Jedes Element ist eine einzelne blaue Komponente. GL wandelt es in Fließkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot sowie Grün und 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_ALPHA`

Jedes Element ist eine einzelne Alpha-Komponente. GL wandelt es in Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 0 für Rot, Grün und Blau anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_RGB` Jedes Element ist ein dreifaches RGB. GL wandelt es in Gleitkomma um und fügt es zu einem RGBA-Element zusammen, indem er 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_RGBA` Jedes Element enthält alle vier Komponenten. Jede Komponente wird mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_LUMINANCE`

Jedes Element ist ein einzelner Leuchtdichtewert. GL wandelt es in Fließkomma um und fügt es dann zu einem RGBA-Element zusammen, indem er den Leuchtdichtewert dreimal für Rot, Grün und Blau repliziert und 1 für Alpha anfügt. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_LUMINANCE_ALPHA`
: Jedes Element ist ein Leuchtdichte/Alpha-Paar. GL wandelt es in Fließkomma um und fügt es dann zu einem RGBA-Element zusammen, indem er den Leuchtdichtewert dreimal für Rot, Grün und Blau repliziert. Jede Komponente wird dann mit dem vorzeichenbehafteten Skalenfaktor `#GL_c_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_c_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

`#GL_DEPTH_COMPONENT`
: Jedes Element ist eine einzelne Tiefenkomponente. Es wird in Gleitkomma umgewandelt, dann mit dem vorzeichenbehafteten Skalierungsfaktor `#GL_DEPTH_SCALE` multipliziert, zur vorzeichenbehafteten Ausrichtung `#GL_DEPTH_BIAS` addiert und in den Bereich [0, 1] festgelegt. Siehe Abschnitt 6.105 [gl.PixelTransfer], Seite 167, für Details.

Wenn eine Anwendung die Textur in einer bestimmten Auflösung oder in einem bestimmten Format speichern möchte, kann sie die Auflösung und das Format mit `internalformat` anfordern. `internalformat` gibt das interne Format des Textur-Feldes an. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. GL wird eine interne Darstellung wählen, die der vom `internalformat` geforderten genau entspricht, aber möglicherweise nicht genau übereinstimmt. (Die durch `#GL_LUMINANCE`, `#GL_LUMINANCE_ALPHA`, `#GL_RGB` und `#GL_RGBA` angegebenen Darstellungen müssen genau übereinstimmen. Die Zahlenwerte 1, 2, 3, und 4 können auch verwendet werden, um die obigen Darstellungen anzugeben.)

Ein Einkomponenten-Texturbild verwendet nur die Rotkomponente der aus Pixeln extrahierten RGBA-Farbe, ein Zweikomponentenbild das R und die A Werte, ein Dreikomponentenbild die Werte R, G und B und ein Vierkomponentenbild verwendet alle RGBA-Komponenten.

Die Texturierung hat im Farbindexmodus keine Wirkung.

Das Texturbild kann durch die gleichen Datenformate dargestellt werden wie die Pixel in einem `gl.DrawPixels()`, nur dass `#GL_STENCIL_INDEX` und `#GL_DEPTH_COMPONENT` nicht verwendet werden können. `gl.PixelStore()` und `gl.PixelTransfer()` Modi beeinflussen Texturbilder genau so, wie sie `gl.DrawPixels()` beeinflussen.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 6.138 [gl.TexImage], Seite 209, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Details zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level`
: gibt die Detailstufenzahl an; Level 0 ist die Basis-Bildebene, Level n ist das n-te Mipmap-Reduktionsbild

`internalformat`
: gibt die Anzahl der Farbkomponenten in der Textur an; muss 1, 2, 3 oder 4 oder eine symbolische Konstante sein (siehe oben)

`w` gibt die Breite des Texturbildes an; muss 2^n + 2*border für eine ganze Zahl n sein; alle Implementierungen unterstützen Texturbilder, die mindestens 64 Texel breit sind

`h` gibt die Höhe des Texturbildes an; muss 2^m + 2*border für eine ganze Zahl m sein; alle Implementierungen unterstützen Texturbilder, die mindestens 64 Texel hoch sind

`border` gibt die Breite des Rahmens an; muss entweder 0 oder 1 sein

`format` gibt das Format der Pixeldaten an (siehe oben)

`type` gibt den Datentyp der Pixeldaten an (siehe oben)

`pixels` gibt einen Zeiger auf die Bilddaten im Speicher an

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` keine akzeptierte Formatkonstante ist. Andere Formatkonstanten als `#GL_STENCIL_INDEX` werden akzeptiert.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` keine Typkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type #GL_BITMAP` ist und `format` nicht `#GL_COLOR_INDEX` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `level` größer als log2max ist, wobei max der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `internalformat` nicht 1, 2, 3, 4 oder eine der akzeptierten symbolischen Auflösungs- und Formatkonstanten ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn `width` oder `height` kleiner als 0 oder größer als 2 + `#GL_MAX_TEXTURE_SIZE` ist oder wenn entweder 2k + 2*border nicht für einen ganzzahligen Wert von k dargestellt werden kann.

`#GL_INVALID_VALUE` wird erzeugt, wenn `border` nicht 0 oder 1 ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexImage2D()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_2D`

6.141 gl.TexParameter

BEZEICHNUNG

gl.TexParameter – stellt Texturparameter ein

ÜBERSICHT

```
gl.TexParameter(target, pname, param)
```

BESCHREIBUNG

Texturzuordnung ist eine Technik, die ein Bild auf die Oberfläche eines Objekts aufbringt, als wäre das Bild eine Dekor- oder Zellophan-Schrumpffolie. Das Bild wird im

Texturraum mit einem Koordinatensystem (s, t) erstellt. Eine Textur ist ein ein- oder zweidimensionales Bild und eine Reihe von Parametern, die bestimmen, wie Muster aus dem Bild abgeleitet werden.

`gl.TexParameter()` weist dem als `pname` angegebenen Texturparameter den oder die Werte in `param` zu. `target` definiert die Zieltextur, entweder `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D`. Die folgenden Symbole werden im `pname` akzeptiert:

`#GL_TEXTURE_MIN_FILTER`

Die Texturbearbeitungsfunktion wird immer dann verwendet, wenn das zu texturierende Pixel auf einen Bereich mit mehr als einem Textelement abgebildet wird. Es gibt sechs definierte Minifizierungsfunktionen. Zwei von ihnen verwenden das nächstgelegene eine oder die nächstgelegenen vier Textelemente, um den Texturwert zu berechnen. Die anderen vier verwenden Mipmaps.

Eine Mipmap ist ein geordneter Satz von Feldern, die das gleiche Bild bei immer niedrigeren Auflösungen darstellen: 2^a für 1D Mipmaps und 2^a*2^b für 2D Mipmaps.

Wenn eine 2D-Textur beispielsweise die Abmessungen 2^m*2^n hat, gibt es max(m, n) + 1 Mipmaps. Das erste Mipmap ist die ursprüngliche Textur mit den Abmessungen 2^m*2^n. Jede nachfolgende Mipmap hat die Abmessungen 2^k-1*2^l-1, wobei 2^k*2^l die Abmessungen der vorherigen Mipmap sind, bis entweder k=0 oder l=0. An diesem Punkt haben die nachfolgenden Mipmaps die Abmessungen 1*2^l-1 oder 2^k-1*1 bis zur endgültigen Mipmap, die die Abmessungen 1x1 hat. Um die Mipmaps zu definieren, rufen Sie `gl.TexImage1D()`, `gl.TexImage2D()` oder `gl.CopyTexImage()` mit dem Level-Argument auf, das die Reihenfolge der Mipmaps angibt. Level 0 ist die ursprüngliche Textur; Level max(m, n) ist die letzte 1x1 Mipmap. `param` liefert eine Funktion zur Verkleinerung der Textur als eine der folgenden:

`#GL_NEAREST`

Liefert den Wert des Texturelements, das am nächsten (in Manhattan-Metrik) zur Mitte des zu texturierenden Pixels liegt.

`#GL_LINEAR`

Liefert den bewerteten Mittelwert der vier Texturelemente, die der Mitte des zu texturierenden Pixels am nächsten liegen. Diese können Randtexturelemente beinhalten, abhängig von den Werten von `#GL_TEXTURE_WRAP_S` und `#GL_TEXTURE_WRAP_T` sowie von der genauen Zuordnung.

`#GL_NEAREST_MIPMAP_NEAREST`

Wählt die Mipmap, die der Größe des zu texturierenden Pixels am nächsten kommt und verwendet das Kriterium `#GL_NEAREST` (das Texturelement, das der Mitte des Pixels am nächsten liegt), um einen Texturwert zu erzeugen.

`#GL_LINEAR_MIPMAP_NEAREST`

Wählt die Mipmap, die der Größe des zu texturierenden Pixels am nächsten kommt und verwendet das Kriterium `#GL_LINEAR`

(ein bewerteter Durchschnitt der vier Texturelemente, die der Mitte des Pixels am nächsten liegen), um einen Texturwert zu erzeugen.

`#GL_NEAREST_MIPMAP_LINEAR`
: Wählt die beiden Mipmaps aus, die der Größe des zu texturierenden Pixels am nächsten kommen und verwendet das Kriterium `#GL_NEAREST` (das Texturelement, das der Mitte des Pixels am nächsten liegt), um aus jedem Mipmap einen Texturwert zu erzeugen. Der endgültige Texturwert ist ein bewerteter Durchschnitt dieser beiden Werte.

`#GL_LINEAR_MIPMAP_LINEAR`
: Wählt die beiden Mipmaps aus, die der Größe des zu texturierenden Pixels am nächsten kommen und verwendet das Kriterium `#GL_LINEAR` (ein bewerteter Durchschnitt der vier Texturelemente, die der Mitte des Pixels am nächsten liegen), um aus jeder Mipmap einen Texturwert zu erzeugen. Der endgültige Texturwert ist ein bewerteter Durchschnitt dieser beiden Werte.

Da im Verkleinerungsprozess mehr Texturelemente gesampelt werden, werden weniger Aliasing-Artefakte sichtbar. Während die Verkleinerungsfunktionen `#GL_NEAREST` und `#GL_LINEAR` schneller sind als die anderen vier, da sie nur ein oder vier Texturelemente abtasten, um den Texturwert des gerenderten Pixels zu bestimmen, können sie aber Moiré-Muster oder unregelmäßige Übergänge erzeugen. Der Initialwert von `#GL_TEXTURE_MIN_FILTER` ist `#GL_NEAREST_MIPMAP_LINEAR`.

`#GL_TEXTURE_MAG_FILTER`
: Die Texturvergrößerungsfunktion wird verwendet, wenn das zu texturierende Pixel auf einen Bereich kleiner oder gleich einem Textelement abgebildet wird. Es stellt die Texturvergrößerungsfunktion entweder auf `#GL_NEAREST` oder `#GL_LINEAR` (siehe unten). `#GL_NEAREST` ist im Allgemeinen schneller als `#GL_LINEAR`, aber es kann strukturierte Bilder mit schärferen Kanten erzeugen, da der Übergang zwischen den Texturelementen nicht so glatt ist. Der Initialwert von `#GL_TEXTURE_MAG_FILTER` ist `#GL_LINEAR`.

  `#GL_NEAREST`
  : Liefert den Wert des Texturelements, das am nächsten (in Manhattan-Metrik) zur Mitte des zu texturierenden Pixels liegt.

  `#GL_LINEAR`
  : Liefert den bewerteten Mittelwert der vier Texturelemente, die der Mitte des zu texturierenden Pixels am nächsten liegen. Diese können Randtexturelemente beinhalten, abhängig von den Werten von `#GL_TEXTURE_WRAP_S` und `#GL_TEXTURE_WRAP_T` sowie von der genauen Zuordnung.

`#GL_TEXTURE_WRAP_S`
: Setzt den Wrap-Parameter für die Texturkoordinate s auf `#GL_CLAMP` oder `#GL_REPEAT`. `#GL_CLAMP` bewirkt, dass die s-Koordinaten auf den Bereich [0, 1] festgelegt werden und ist nützlich, um Wrapping-Artefakte beim zuordnen eines einzelnen Bildes auf ein Objekt zu verhindern. `#GL_REPEAT` bewirkt, dass der ganzzahlige Teil der s-Koordinate ignoriert wird; GL verwendet nur den Bruchteil und erzeugt so ein sich wiederholendes Muster. Auf Randtexturelemente wird nur zugegriffen, wenn das Wrapping auf `#GL_CLAMP` gesetzt ist. Zunächst wird `#GL_TEXTURE_WRAP_S` auf `#GL_REPEAT` gesetzt.

`#GL_TEXTURE_WRAP_T`
: Setzt den Wrap-Parameter für die Texturkoordinate t auf `#GL_CLAMP` oder `#GL_REPEAT`. Siehe die Erläuterung unter `#GL_TEXTURE_WRAP_S`. Zunächst wird `#GL_TEXTURE_WRAP_T` auf `#GL_REPEAT` gesetzt.

`#GL_TEXTURE_WRAP_R_EXT`
: Setzt den Wrap-Parameter für die Texturkoordinate r auf `#GL_CLAMP` oder `#GL_REPEAT`. Siehe die Erläuterung unter `#GL_TEXTURE_WRAP_S`. Zunächst wird `#GL_TEXTURE_WRAP_R_EXT` auf `#GL_REPEAT` gesetzt.

`#GL_TEXTURE_BORDER_COLOR`
: Setzt eine Rahmenfarbe. `param` muss eine Tabelle mit vier Gleitkommawerten sein, die die RGBA-Farbe des Texturrandes umfassen. Zunächst ist die Randfarbe (0, 0, 0, 0).

`#GL_TEXTURE_PRIORITY`
: Gibt die Priorität der Texturresidenz der aktuell gebundenen Textur an. Die zulässigen Werte liegen im Bereich [0, 1]. Siehe Abschnitt 6.115 [gl.PrioritizeTextures], Seite 178, für Details.

Angenommen, ein Programm hat die Texturierung aktiviert (durch Aufruf von `gl.Enable()` mit dem Argument `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D`) und hat `#GL_TEXTURE_MIN_FILTER` auf eine der Funktionen gesetzt, die eine Mipmap erfordert. Wenn entweder die Dimensionen der aktuell definierten Texturbilder (mit vorherigen Aufrufen von `gl.TexImage1D()`, `gl.TexImage2D()` oder `gl.CopyTexImage()`) nicht der richtigen Reihenfolge für Mipmaps (siehe oben) folgen, oder es sind weniger Texturbilder definiert als benötigt werden, oder der Satz von Texturbildern hat eine unterschiedliche Anzahl von Texturkomponenten, dann ist es, als ob die Texturzuordnung deaktiviert wäre.

Die lineare Filterung greift nur in 2D-Texturen auf die vier nächstgelegenen Textelemente zu. In 1D-Texturen greift die lineare Filterung auf die beiden nächstgelegenen Textelemente zu.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`target` gibt die Zieltextur an, die entweder `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` sein muss

`pname` gibt den symbolischen Namen eines Texturparameters an (siehe oben)

`param` spezifiziert einen Einzelwert oder eine Tabelle, die den Wert für `pname` enthält

FEHLER

`#GL_INVALID_ENUM` wird erzeugt, wenn `target` oder `pname` nicht einer der akzeptierten definierten Werte ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `param` einen definierten konstanten Wert haben sollte (basierend auf dem Wert von `pname`) und es nicht hat.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexParameter()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexParameter()`

`gl.GetTexLevelParameter()`

6.142 gl.TexSubImage

BEZEICHNUNG

gl.TexSubImage – gibt ein ein- oder zweidimensionales Textur-Teilbild an

ÜBERSICHT

```
gl.TexSubImage(level, format, type, pixels, xoffset[, yoffset])
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.TexSubImage1D()` und `gl.TexSubImage2D()`, mit der Ausnahme, dass die Pixeldaten nicht in einem Rohspeicherpuffer übergeben werden, sondern als Tabelle, die für jede Pixelreihe eine Untertabelle enthält. Dies ist natürlich nicht so effizient wie die Verwendung von Rohspeicherpuffern, da die Pixeldaten der Tabelle zuerst in einen Rohspeicherpuffer kopiert werden müssen.

Breite und Höhe der Textur werden automatisch durch das Layout der Tabelle in `pixels` bestimmt. Wenn es nur eine Untertabelle innerhalb von `pixels` gibt, definiert `gl.TexSubImage()` eine Textur vom Typ `#GL_TEXTURE_1D`. Wenn es mehrere Untertabellen innerhalb von `pixels` gibt, wird `#GL_TEXTURE_2D` verwendet.

Beachten Sie, dass derzeit nur `#GL_FLOAT` und `#GL_UNSIGNED_BYTE` für `type` unterstützt werden.

Siehe Abschnitt 6.144 [gl.TexSubImage2D], Seite 223, für weitere Einzelheiten zu den von diesem Befehl akzeptierten Parametern.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level`	gibt die Detaillierungsstufe an; Stufe 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild
`format`	gibt das Format der Pixeldaten an (siehe oben)
`type`	gibt den Datentyp der Pixeldaten an (siehe oben)
`pixels`	spezifiziert eine ein- oder zweidimensionale mit Pixeldaten
`xoffset`	gibt einen Texel-Versatz in der x-Richtung innerhalb des Textur-Feldes an
`yoffset`	optional: Gibt einen Texel-Versatz in der y-Richtung innerhalb des Textur-Feldes an (nur erforderlich für zweidimensionale Texturen)

6.143 gl.TexSubImage1D

BEZEICHNUNG

gl.TexSubImage1D – gibt ein eindimensionales Textur-Teilbild an

ÜBERSICHT

```
gl.TexSubImage1D(level, xoffset, width, format, type, pixelsUserData)
```

BESCHREIBUNG

Texturierung bildet einen Teil eines bestimmten Texturbildes auf jedes grafische Grundmuster ab, für das die Texturierung aktiviert ist. Um die eindimensionale Texturierung zu aktivieren oder zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_TEXTURE_1D` auf.

`gl.TexSubImage1D()` definiert eine zusammenhängende Subregion eines vorhandenen eindimensionalen Texturbildes neu. Die durch Pixel referenzierten Texel ersetzen den Teil des vorhandenen Texturfeldes durch X-Indizes xoffset und xoffset + width - 1, inklusive. Dieser Bereich darf keine Texel außerhalb des Bereichs des Texturfeldes enthalten, wie er ursprünglich angegeben wurde. Es ist kein Fehler, eine Subtextur mit der Breite 0 anzugeben, aber eine solche Spezifikation hat keine Wirkung.

Die Texturierung hat im Farbindexmodus keine Wirkung.

`gl.PixelStore()` und `gl.PixelTransfer()` Modi beeinflussen Texturbilder genau so, wie `gl.DrawPixels()` sie beeinflussen.

Siehe Abschnitt 6.139 [gl.TexImage1D], Seite 210, für weitere Einzelheiten zu den von dieserm Befehl akzeptierten Parametern.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 6.142 [gl.TexSubImage], Seite 221, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level`	gibt die Detaillierungsstufe an; Stufe 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild
`xoffset`	gibt einen Texel-Versatz in der x-Richtung innerhalb des Textur-Feldes an
`width`	legt die Breite des Textur-Unterbildes fest
`format`	gibt das Format der Pixeldaten an (siehe oben)
`type`	gibt den Datentyp der Pixeldaten an (siehe oben)
`pixels`	gibt einen Zeiger auf die Bilddaten im Speicher an

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn das Textur-Feld nicht durch ein vorherige `gl.TexImage1D()` Operation definiert wurde.

`#GL_INVALID_VALUE` wird generiert, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` kann erzeugt werden, wenn `level` größer als log2max ist, wobei max der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn xoffset `<` -b, oder wenn (xoffset `+` width) `>` (w - b) ist, wobei w die `#GL_TEXTURE_WIDTH` und b die Breite des `#GL_TEXTURE_BORDER` des zu modifizierenden Texturbildes ist. Beachten Sie, dass w die doppelte Randbreite beinhaltet.

`#GL_INVALID_VALUE` wird erzeugt, wenn `width` kleiner als 0 ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` keine akzeptierte Formatkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` keine Typkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` `#GL_BITMAP` und das Format nicht `#GL_COLOR_INDEX` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexSubImage1D()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_1D`

6.144 gl.TexSubImage2D

BEZEICHNUNG

gl.TexSubImage2D – gibt ein zweidimensionales Textur-Teilbild an

ÜBERSICHT

```
gl.TexSubImage2D(level, xoff, yoff, w, h, format, type, pixels)
```

BESCHREIBUNG

Texturierung bildet einen Teil eines bestimmten Texturbildes auf jedes grafische Grundmuster ab, für das die Texturierung aktiviert ist. Um die zweidimensionale Texturierung zu aktivieren und zu deaktivieren, rufen Sie `gl.Enable()` und `gl.Disable()` mit dem Argument `#GL_TEXTURE_2D` auf.

`gl.TexSubImage2D()` definiert einen zusammenhängenden Subbereich eines vorhandenen zweidimensionalen Texturbildes neu. Die durch `pixels` referenzierten Texel ersetzen den Teil des vorhandenen Texturfeldes durch X-Indizes `xoffset` und xoffset `+` width - 1, inklusive und Y-Indizes `yoffset` und yoffset `+` height - 1, inklusive. Dieser Bereich darf keine Texel außerhalb des Bereichs des Texturfeldes enthalten, wie er ursprünglich angegeben wurde. Es ist kein Fehler, eine Subtextur mit einer Breite oder Höhe von Null anzugeben, aber eine solche Angabe hat keine Wirkung.

Die Texturierung hat im Farbindexmodus keine Wirkung.

`gl.PixelStore()` und `gl.PixelTransfer()` Modi beeinflussen Texturbilder genau so, wie `gl.DrawPixels()` sie beeinflussen.

Siehe Abschnitt 6.140 [gl.TexImage2D], Seite 214, für weitere Einzelheiten zu den von dieserm Befehl akzeptierten Parametern.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 6.142 [gl.TexSubImage], Seite 221, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`level`	gibt die Detaillierungsstufe an; Stufe 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild
`xoffset`	gibt einen Texel-Versatz in der x-Richtung innerhalb des Textur-Feldes an
`yoffset`	gibt einen Texel-Versatz in y-Richtung innerhalb des Textur-Feldes an
`width`	legt die Breite des Textur-Unterbildes fest
`height`	gibt die Höhe des Textur-Subibildes an
`format`	gibt das Format der Pixeldaten an (siehe oben)
`type`	gibt den Datentyp der Pixeldaten an (siehe oben)
`pixels`	gibt einen Zeiger auf die Bilddaten im Speicher an

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn das Textur-Feld nicht durch ein vorherige `gl.TexImage2D()` Operation definiert wurde.

`#GL_INVALID_VALUE` wird generiert, wenn `level` kleiner als 0 ist.

`#GL_INVALID_VALUE` kann erzeugt werden, wenn `level` größer als log2max ist, wobei max der zurückgegebene Wert von `#GL_MAX_TEXTURE_SIZE` ist.

`#GL_INVALID_VALUE` wird erzeugt, wenn xoffset `<` -b, (xoffset `+` width) `>` (w - b), yoffset `<` -b, oder (yoffset `+` height) `>` (h - b), wobei w die `#GL_TEXTURE_WIDTH` ist, h die `#GL_TEXTURE_HEIGHT` ist und b die Randbreite des zu modifizierenden Texturbildes ist. Beachten Sie, dass w und h die doppelte Randbreite beinhalten.

`#GL_INVALID_VALUE` wird erzeugt, wenn `width` oder `height` kleiner als 0 ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `format` keine akzeptierte Formatkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type` keine Typkonstante ist.

`#GL_INVALID_ENUM` wird erzeugt, wenn `type #GL_BITMAP` ist und `format` nicht `#GL_COLOR_INDEX` ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.TexSubImage2D()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.GetTexImage()`

`gl.IsEnabled()` mit dem Argument `#GL_TEXTURE_2D`

6.145 gl.Translate

BEZEICHNUNG

gl.Translate – multipliziert die aktuelle Matrix mit einer Übersetzungsmatrix

ÜBERSICHT

```
gl.Translate(x, y, z)
```

BESCHREIBUNG

`gl.Translate()` erzeugt eine Verschiebung von (x,y,z). Die aktuelle Matrix (siehe `gl.MatrixMode()`) wird mit dieser Verschiebungsmatrix multipliziert, wobei das Produkt die aktuelle Matrix ersetzt, als ob `gl.MultMatrix()` mit der folgenden Matrix für ihr Argument aufgerufen wurde:

```
1 0 0 x
0 1 0 y
0 0 1 z
0 0 0 1
```

Wenn der Matrixmodus entweder `#GL_MODELVIEW` oder `#GL_PROJECTION` ist, werden alle Objekte, die nach einem Aufruf von `gl.Translate()` gezeichnet wurden, verschoben.

Verwenden Sie `gl.PushMatrix()` und `gl.PopMatrix()`, um das nicht verschobene Koordinatensystem zu speichern und wiederherzustellen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`	gibt die x-Koordinate eines Verschiebungsvektors an
`y`	gibt die y-Koordinate eines Verschiebungsvektors an
`z`	gibt die z-Koordinate eines Verschiebungsvektors an

FEHLER

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Translate()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_MATRIX_MODE`

`gl.Get()` mit dem Argument `#GL_MODELVIEW_MATRIX`

`gl.Get()` mit dem Argument `#GL_PROJECTION_MATRIX`

`gl.Get()` mit dem Argument `#GL_TEXTURE_MATRIX`

6.146 gl.Vertex

BEZEICHNUNG

gl.Vertex – gibt einen Knoten an

ÜBERSICHT

```
gl.Vertex(x, y[, z, w])
```

BESCHREIBUNG

`gl.Vertex()` wird innerhalb von `gl.Begin()` / `gl.End()` Paare verwendet, um Punkt-, Linien- und Polygonknoten anzugeben. Die aktuelle Farb-, Normalen-, Textur- und Nebelkoordinate sind dem Knoten zugeordnet, wenn `gl.Vertex()` aufgerufen wird.

Wenn nur `x` und `y` angegeben sind, ist `z` standardmäßig auf 0 und `w` standardmäßig auf 1 eingestellt. Wenn x, y und z angegeben sind, ist `w` standardmäßig auf 1 eingestellt.

Alternativ können Sie diesem Befehl auch eine Tabelle mit zwei bis vier Knotenkoordinaten übergeben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`	gibt die x-Koordinate eines Knoten an
`y`	gibt die y-Koordinate eines Knoten an
`z`	optional: Gibt die z-Koordinate eines Knoten an (Standard ist 0)
`w`	optional: Gibt die w-Koordinate eines Knoten an (Standard ist 1)

6.147 gl.VertexPointer

BEZEICHNUNG

gl.VertexPointer – definiert ein Feld von Scheitelpunktdaten

ÜBERSICHT

```
gl.VertexPointer(vertexArray[, size])
```

BESCHREIBUNG

`gl.VertexPointer()` gibt ein Feld von Vertex-Koordinaten an, die beim Rendern verwendet werden sollen. `vertexArray` kann entweder eine eindimensionale Tabelle sein, die aus einer beliebigen Anzahl von aufeinanderfolgenden Knotenkoordinaten besteht, oder eine zweidimensionale Tabelle, die aus einer beliebigen Anzahl von Untertabellen besteht, die jeweils 2 bis 4 Texturkoordinaten enthalten. Wenn `vertexArray` eine eindimensionale Tabelle ist, müssen Sie auch das optionale Argument `size` übergeben, um die Anzahl der Knotenkoordinaten pro Feld-Element zu definieren. Wenn `vertexArray` eine zweidimensionale Tabelle ist, wird `size` automatisch durch die Anzahl der Elemente in der ersten Untertabelle bestimmt, die ebenfalls im Bereich von 2 bis 4 liegen müssen.

Bei der Verwendung einer zweidimensionalen Tabelle ist zu beachten, dass die Anzahl der Knotenkoordinaten in jeder Untertabelle konstant sein muss. Es ist nicht erlaubt, in den einzelnen Untertabellen eine unterschiedliche Anzahl von Knotenkoordinaten zu verwenden. Die Anzahl der Knotenkoordinaten wird durch die Anzahl der Elemente in der ersten Untertabelle definiert und alle folgenden Untertabellen müssen die gleiche Anzahl von Koordinaten verwenden.

Wenn Sie `Nil` in `vertexArray` übergeben, wird der Inhalt des Vertex-Koordinaten-Feld-Puffer gelöscht, aber er wird nicht aus OpenGL entfernt. Dies muss manuell erfolgen, z.B. durch Deaktivieren des Vertex-Koordinaten-Feldes oder durch Definieren eines neuen.

Um ein Vertex-Feld zu aktivieren und zu deaktivieren, rufen Sie den Befehl `gl.EnableClientState()` und `gl.DisableClientState()` mit dem Argument `#GL_VERTEX_ARRAY` auf. Wenn aktiviert, wird Vertex-Array verwendet, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Das Vertex-Feld ist zunächst deaktiviert und wird nicht aufgerufen, wenn `gl.DrawArrays()`, `gl.DrawElements()` oder `gl.ArrayElement()` aufgerufen wird.

Die Ausführung von `gl.VertexPointer()` ist zwischen der Ausführung von `gl.Begin()` und `gl.End()` nicht erlaubt. Dabei kann ein Fehler auftreten oder auch nicht. Wenn kein Fehler erzeugt wird, ist der Vorgang undefiniert.

`gl.VertexPointer()` wird typischerweise auf der Klient-Seite implementiert.

Vertex-Feld-Parameter sind klient-seitige Zustände und werden daher nicht durch `gl.PushAttrib()` und `gl.PopAttrib()` gespeichert oder wiederhergestellt. Benutzen Sie stattdessen `gl.PushClientAttrib()` und `gl.PopClientAttrib()`.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`vertexArray`
: ein- oder zweidimensionale Tabelle mit Knotenkoordinaten oder `Nil` (siehe oben)

`size`
: optional: Vertexkoordinaten pro Feld-Element; muss zwischen 2 und 4 liegen und wird nur bei eindimensionalen Tabellen (siehe oben) verwendet

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn `size` nicht 2, 3, oder 4 ist.

VERBUNDENE GET-OPERATIONEN

`gl.IsEnabled()` mit dem Argument `#GL_VERTEX_ARRAY`

`gl.Get()` mit dem Argument `#GL_VERTEX_ARRAY_SIZE`

`gl.Get()` mit dem Argument `#GL_VERTEX_ARRAY_TYPE`

`gl.Get()` mit dem Argument `#GL_VERTEX_ARRAY_STRIDE`

`gl.GetPointer()` mit dem Argument `#GL_VERTEX_ARRAY_POINTER`

6.148 gl.Viewport

BEZEICHNUNG

gl.Viewport – setzt das Ansichtsfenster

ÜBERSICHT

```
gl.Viewport(x, y, width, height)
```

BESCHREIBUNG

`gl.Viewport()` gibt die affine Transformation von x und y von normierten Gerätekoordinaten in Fensterkoordinaten an. Wenn (xnd, ynd) normalisierte Gerätekoordinaten sind, dann werden die Fensterkoordinaten (xw, yw) wie folgt berechnet:

```
xw = (xnd + 1) * (width / 2) + x
yw = (ynd + 1) * (height / 2) + y
```

Die Breite und Höhe des Ansichtsfensters werden automatisch auf einen von der Implementierung abhängigen Bereich festgelegt. Um diesen Bereich abzufragen, rufen Sie den Befehl `gl.Get()` mit dem Argument `#GL_MAX_VIEWPORT_DIMS` auf.

Wenn ein GL-Kontext zum ersten Mal an ein Fenster angehängt wird, werden Breite und Höhe auf die Abmessungen dieses Fensters festgelegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x`
: gibt die linke Ecke des Ansichtsrechtecks in Pixeln an; der Anfangswert ist 0

`y`	gibt die untere Ecke des Ansichtsrechtecks in Pixeln an; der Anfangswert ist 0
`width`	gibt die Breite des Ansichtsfensters an
`height`	gibt die Höhe des Ansichtsfensters an

FEHLER

`#GL_INVALID_VALUE` wird erzeugt, wenn entweder `width` oder `height` negativ ist.

`#GL_INVALID_OPERATION` wird erzeugt, wenn `gl.Viewport()` zwischen `gl.Begin()` und `gl.End()` ausgeführt wird.

VERBUNDENE GET-OPERATIONEN

`gl.Get()` mit dem Argument `#GL_VIEWPORT`

`gl.Get()` mit dem Argument `#GL_MAX_VIEWPORT_DIMS`

7 GLU Referenz

7.1 glu.Build1DMipmaps

BEZEICHNUNG

glu.Build1DMipmaps – erstellt eine 1D Mipmap

ÜBERSICHT

```
error = glu.Build1DMipmaps(internalformat, width, format, type, pixels)
```

BESCHREIBUNG

`glu.Build1DMipmaps()` erstellt eine Reihe von vorgefilterten 1D-Texturkarten mit abnehmender Auflösung (Mipmap genannt). Mipmaps können so verwendet werden, dass Texturen nicht aliasiert erscheinen.

Ein Rückgabewert von 0 bedeutet Erfolg. Andernfalls wird ein GLU-Fehlercode zurückgegeben. Siehe Abschnitt 7.5 [glu.ErrorString], Seite 234, für Details.

`glu.Build1DMipmaps()` prüft zunächst, ob die Breite der Daten eine Potenz von 2 ist, andernfalls skaliert es eine Kopie von `pixels` (aufwärts oder abwärts) auf die nächstgelegene Potenz von 2. Diese Kopie dient als Grundlage für nachfolgende Mipmapping-Vorgänge. Wenn beispielsweise `width` 57 ist, skaliert eine Kopie von `pixels` auf 64, bevor ein Mipmapping stattfindet (wenn `width` genau zwischen 2er Potenzen liegt, wird die Kopie von `pixels` nach oben skaliert).

Wenn die GL-Version 1.1 oder höher ist, verwendet `glu.Build1DMipmaps()` Proxy-Texturen (siehe `gl.TexImage1D()`), um zu bestimmen, ob die Implementierung die gewünschte Textur im Texturspeicher speichern kann. Wenn nicht genügend Platz vorhanden ist, wird `width` halbiert (und wieder halbiert), bis sie passt.

Als nächstes erstellt `glu.Build1DMipmaps()` eine Reihe von Mipmap-Ebenen; es halbiert eine Kopie von `pixels` (oder eine skalierte Version von `pixels`, falls erforderlich), bis Größe 1 erreicht ist. Auf jeder Ebene ist jedes Stück im halbierten Bild ein Durchschnitt der entsprechenden zwei Stücke im größeren Bild.

`gl.TexImage1D()` wird aufgerufen, um jedes dieser Bilder nach Ebenen zu laden. Wenn `width` eine Potenz von 2 ist, die in die Implementierung passt, ist Ebene 0 eine Kopie von `pixels` und die höchste Ebene ist log2(width). Wenn z.B. `width` 64 ist, werden folgende Bilder erstellt: 64x1, 32x1, 16x1, 8x1, 4x1, 2x1 und 1x1. Diese entsprechen den Stufen 0 bis 6.

`internalformat` gibt das interne Format des Texturbildes an. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. Dies kann auch einer der Sonderwerte 1, 2, 3 oder 4 sein.

`format` muss eine der folgenden Konstanten sein: `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` oder `#GL_LUMINANCE_ALPHA`

`type` muss eine der folgenden sein: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT`.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist

natürlich langsamer. Siehe Abschnitt 7.4 [glu.BuildMipmaps], Seite 233, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`internalformat`
: gibt das interne Format der Textur an; muss eine der Pixelformatkonstanten sein (siehe oben)

`width`
: gibt die Breite des Texturbildes an

`format`
: gibt das Format der Pixeldaten an (siehe oben)

`type`
: gibt den Datentyp für `pixels` an (siehe oben)

`pixels`
: gibt einen Zeiger auf die Bilddaten im Speicher an

RÜCKGABEWERTE

`error`
: Fehlercode oder 0 für den Erfolg

7.2 glu.Build2DMipmaps

BEZEICHNUNG

glu.Build2DMipmaps – erstellt eine 2D-Mipmap

ÜBERSICHT

```
error = glu.Build2DMipmaps(iformat, width, height, format, type, pixels)
```

BESCHREIBUNG

`glu.Build2DMipmaps()` erstellt eine Reihe von vorgefilterten 2D-Texturkarten mit abnehmender Auflösung. Mipmaps können so verwendet werden, dass Texturen nicht aliasiert erscheinen.

Ein Rückgabewert von 0 bedeutet Erfolg. Andernfalls wird ein GLU-Fehlercode zurückgegeben. Siehe Abschnitt 7.5 [glu.ErrorString], Seite 234, für Details.

`glu.Build2DMipmaps()` prüft zunächst, ob `width` und `height` von `pixels` beide eine Potenzen von 2 haben. `glu.Build2DMipmaps()` skaliert eine Kopie von `pixels` auf die nächste Potenz von 2 und dient dann als Basis für nachfolgende Mipmapping-Operationen. Wenn beispielsweise `width` 57 und `height` 23 ist, dann skaliert eine Kopie von `pixels` auf 64 bzw. auf 16, bevor ein Mipmapping stattfindet (wenn `width` oder `height` genau zwischen den Potenzen von 2 liegt, wird die Kopie der Daten nach oben skaliert).

Wenn die GL-Version 1.1 oder höher ist, verwendet `glu.Build2DMipmaps()` dann Proxy-Texturen (siehe `gl.TexImage2D()`), um festzustellen, ob genügend Platz für die gewünschte Textur in der Implementierung vorhanden ist. Wenn nicht, wird die `width` halbiert (und wieder halbiert), bis sie passt.

`glu.Build2DMipmaps()` verwendet dann Proxy-Texturen (siehe `gl.TexImage2D()`), um zu bestimmen, ob die Implementierung die gewünschte Textur im Texturspeicher speichern kann. Wenn nicht, werden beide Dimensionen kontinuierlich halbiert, bis es passt.

Anschließend erstellt `glu.Build2DMipmaps()` eine Reihe von Bildern; es halbiert eine Kopie von `type` (oder eine skalierte Version von `type`, falls erforderlich) entlang beider Dimensionen, bis die Größe 1x1 erreicht ist. Auf jeder Ebene ist jedes Texel in der halbierten Mipmap ein Durchschnitt der entsprechenden vier Texel in der größeren Mipmap. (Im Falle von rechteckigen Bildern führt die Halbierung der Bilder wiederholt zu einer n*1 oder 1*n Konfiguration. Hier werden stattdessen zwei Texel gemittelt.)

`gl.TexImage2D()` wird aufgerufen, um jedes dieser Bilder nach Ebenen zu laden. Wenn `width` und `height` beide Potenzen von 2 sind, die in die Implementierung passen, ist Ebene 0 eine Kopie von `pixels` und die höchste Ebene ist log2(max(width, height)). Wenn beispielsweise `width` 64 und `height` 16 ist, werden folgende Mipmaps erstellt: 64x16, 32x8, 16x4, 8x2, 4x1, 2x1 und 1x1. Diese entsprechen den Stufen 0 bis 6.

`iformat` legt das interne Format des Texturbildes fest. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. Dies kann auch einer der Sonderwerte 1, 2, 3 oder 4 sein.

`format` muss eine der folgenden Konstanten sein: `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` oder `#GL_LUMINANCE_ALPHA`.

`type` muss eine der folgenden sein: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT`.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 7.4 [glu.BuildMipmaps], Seite 233, für Details. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`iformat`	gibt das interne Format der Textur an; muss eine der Pixelformatkonstanten sein (siehe oben)
`width`	gibt die Breite des Texturbildes an
`height`	gibt die Höhe des Texturbildes an
`format`	gibt das Format der Pixeldaten an (siehe oben)
`type`	gibt den Datentyp für `pixels` an (siehe oben)
`pixels`	gibt einen Zeiger auf die Bilddaten im Speicher an

RÜCKGABEWERTE

`error`	Fehlercode oder 0 für den Erfolg

7.3 glu.Build3DMipmaps

BEZEICHNUNG

glu.Build3DMipmaps – erstellt eine 3D-Mipmap

ÜBERSICHT

```
error = glu.Build3DMipmaps(ifmt, width, height, depth, fmt, type, data)
```

BESCHREIBUNG

`glu.Build3DMipmaps()` erstellt eine Reihe von vorgefilterten 3D-Texturkarten mit abnehmender Auflösung, genannt Mipmap. Dies wird für das Antialiasing von texturierten Grundmuster verwendet.

Ein Rückgabewert von Null bedeutet Erfolg, ansonsten wird ein GLU-Fehlercode zurückgegeben. Siehe Abschnitt 7.5 [glu.ErrorString], Seite 234, für Details.

Zuerst werden die `width`, `height` und `depth` der Daten überprüft, um festzustellen, ob sie eine Potenz von 2 haben. Andernfalls wird eine Kopie der Daten erstellt und auf die nächstgelegene Potenz von 2 hoch- oder herunterskaliert (wenn `width`, `height` und `depth` genau zwischen 2 liegt, dann wird die Kopie der Daten nach oben skaliert). Diese Kopie wird für nachfolgenden beschriebene Mipmapping-Vorgänge verwendet. Wenn zum Beispiel die Breite 57, die Höhe 23 und die Tiefe 24 ist, dann skaliert eine Kopie der Daten bis zu 64 in der Breite, bis zu 16 in der Höhe und bis zu 32 in der Tiefe, bevor das Mipmapping stattfindet.

Anschließend werden Proxy-Texturen (siehe gl.TexImage3D()) verwendet, um festzustellen, ob die Implementierung zur gewünschten Textur passen kann. Wenn nicht, werden alle drei Dimensionen kontinuierlich halbiert, bis sie passen.

Als nächstes wird eine Reihe von Mipmap-Ebenen aufgebaut, indem eine Kopie der Daten in der Hälfte entlang aller drei Dimensionen dezimiert wird, bis die Größe 1x1x1 erreicht ist. Auf jeder Ebene ist jedes Texel in der halbierten Mipmap-Ebene ein Durchschnitt der entsprechenden acht Texel in der größeren Mipmap-Ebene. (Wenn genau eine der Dimensionen 1 ist, werden vier Texel gemittelt. Wenn genau zwei der Dimensionen 1 sind, werden zwei Texel gemittelt.)

gl.TexImage3D() wird aufgerufen, um jede dieser Mipmap-Ebenen zu laden. Level 0 ist eine Kopie von Daten. Die höchste Ebene ist log2(max(width,height,depth)). Wenn beispielsweise `width` 64, `height` 16 und `depth` 32 ist und die Implementierung eine Textur dieser Größe speichern kann, werden folgende Mipmap-Level aufgebaut: 64x16x32, 32x8x16, 16x4x8, 8x2x4, 4x1x2, 2x1x1 und 1x1x1. Diese entsprechen den Stufen 0 bis 6.

`ifmt` gibt das interne Format des Texturbildes an. Siehe Abschnitt 3.12 [Interne Pixelformate], Seite 16, für Details. Dies kann auch einer der Sonderwerte 1, 2, 3 oder 4 sein.

`fmt` muss eine der folgenden Konstanten sein: `#GL_COLOR_INDEX`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` oder `#GL_LUMINANCE_ALPHA`.

`type` muss eine der folgenden sein: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT`.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Siehe Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`ifmt`	gibt das interne Format der Textur an; muss eine der Pixelformatkonstanten sein (siehe oben)

`width` gibt die Breite des Texturbildes an

`height` gibt die Höhe des Texturbildes an

`depth` gibt die Tiefe des Texturbildes an

`fmt` gibt das Format der Pixeldaten an (siehe oben)

`type` gibt den Datentyp für `pixels` an (siehe oben)

`data` gibt einen Zeiger auf die Bilddaten im Speicher an

RÜCKGABEWERTE

`error` Fehlercode oder 0 für den Erfolg

7.4 glu.BuildMipmaps

BEZEICHNUNG

glu.BuildMipmaps – erstellt 1D- oder 2D-Mipmaps

ÜBERSICHT

```
error = glu.BuildMipmaps(internalformat, format, pixels)
```

BESCHREIBUNG

Dieser Befehl entspricht `glu.Build1DMipmaps()` und `gl.Build2DMipmaps()`, mit der Ausnahme, dass die Pixeldaten nicht in einem Rohspeicherpuffer übergeben werden, sondern als Tabelle, die für jede Pixelreihe eine Untertabelle enthält. Dies ist natürlich nicht so effizient wie die Verwendung von Rohspeicherpuffern, da die Pixeldaten der Tabelle zuerst in einen Rohspeicherpuffer kopiert werden müssen.

Breite und Höhe der Textur werden automatisch durch das Layout der Tabelle in `pixels` bestimmt. Wenn es nur eine Untertabelle innerhalb von `pixels` gibt, erstellt gl.BuildMipmaps() eine 1D-Mipmap. Wenn es mehrere Untertabellen innerhalb von `pixels` gibt, werden 2D-Mipmaps erstellt.

Beachten Sie, dass `#GL_UNSIGNED_BYTE` derzeit der einzige unterstützte Datentyp ist. `glu.BuildMipmaps()` erwartet, dass alle Elemente in `pixels` den Datentyp `#GL_UNSIGNED_BYTE` verwenden.

Siehe Abschnitt 7.2 [glu.Build2DMipmaps], Seite 230, für weitere Einzelheiten zu den von dieserm Befehl akzeptierten Parametern.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`internalFormat` gibt das interne Format der Textur an; muss eine der Pixelformatkonstanten sein (siehe oben)

`format` gibt das Format der Pixeldaten an (siehe oben)

`pixels` spezifiziert eine Tabelle, die die Pixeldaten enthält

RÜCKGABEWERTE

`error` Fehlercode oder 0 für den Erfolg

7.5 glu.ErrorString

BEZEICHNUNG

glu.ErrorString – erzeugt eine Fehlerzeichenfolge aus einem GL- oder GLU-Fehlercode

ÜBERSICHT

```
string = glu.ErrorString(errorCode)
```

BESCHREIBUNG

`glu.ErrorString()` erzeugt eine Fehlerzeichenkette aus einem GL- oder GLU-Fehlercode. Die Zeichenkette ist im ISO Latin 1 Format. glu.ErrorString(`#GL_OUT_OF_MEMORY`) gibt beispielsweise die Zeichenkette `"Out of memory"` zurück.

Die Standard-GLU-Fehlercodes sind `#GLU_INVALID_ENUM`, `#GLU_INVALID_VALUE` und `#GLU_OUT_OF_MEMORY`. Bestimmte andere GLU-Befehle können spezielle Fehlercodes durch Callback-Aktionen zurückgeben. Siehe Abschnitt 6.60 [gl.GetError], Seite 108, für die Liste der GL-Fehlercodes.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`errorCode`
: gibt einen GL oder GLU-Fehlercode an

RÜCKGABEWERTE

`string` Fehlerzeichenkette

7.6 glu.GetString

BEZEICHNUNG

glu.GetString – gibt eine Zeichenkette zurück, die die GLU-Version oder GLU-Erweiterungen beschreibt

ÜBERSICHT

```
string = glu.GetString(name)
```

BESCHREIBUNG

`glu.GetString()` gibt eine Zeichenkette zurück, die die GLU-Version oder die unterstützten GLU-Erweiterungen beschreibt.

Die Versionsnummer hat eines der folgenden Formate:

```
<major_number>.<minor_number>
<major_number>.<minor_number>.<release_number>
```

Die Versionszeichenkette hat die folgende Form:

```
<version number><space><vendor-specific information>
```

Lieferantenspezifische Informationen sind optional. Format und Inhalt sind abhängig von der Implementierung.

Die Standard-GLU enthält einen Basissatz von Befehlen und Fähigkeiten. Wenn ein Unternehmen oder eine Unternehmensgruppe andere Merkmale unterstützen möchte, können diese als Erweiterungen der GLU aufgenommen werden. Wenn `name #GLU_EXTENSIONS` ist, dann gibt `glu.GetString()` eine durch Leerzeichen getrennte Liste von

Namen der unterstützten GLU-Erweiterungen zurück. (Erweiterungsnamen enthalten keine Leerzeichen.)

Bitte beachten Sie, dass `glu.GetString()` nur Informationen über GLU-Erweiterungen zurückgibt. Rufen Sie `gl.GetString()` auf, um eine Liste der GL-Erweiterungen zu erhalten.

`glu.GetString()` ist eine Initialisierungsroutine. Aufruf nach einem `gl.NewList()` führt zu einem undefinierten Verhalten.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`name` gibt eine symbolische Konstante an, entweder `#GLU_VERSION`, oder `#GLU_EXTENSIONS`

RÜCKGABEWERTE

`string` Zeichenkette, die die GLU-Version oder die unterstützten GLU-Erweiterungen beschreibt

7.7 glu.LookAt

BEZEICHNUNG

glu.LookAt – definiert eine Ansichtstransformation

ÜBERSICHT

```
glu.LookAt(Ex, Ey, Ez, Lx, Ly, Lz, Ux, Uy, Uz)
```

BESCHREIBUNG

`glu.LookAt()` erzeugt eine Betrachtungsmatrix, die von einem Augenpunkt, einem Referenzpunkt, der die Mitte der Szene anzeigt und einem UP-Vektor abgeleitet ist.

Die Matrix bildet den Bezugspunkt auf die negative z-Achse und den Augenpunkt auf den Ausgangspunt ab. Bei Verwendung einer typischen Projektionsmatrix wird daher die Mitte der Szene auf die Mitte des Ansichtsfensters abgebildet. Ebenso wird die durch den auf die Betrachtungsebene projizierten UP-Vektor beschriebene Richtung auf die positive y-Achse abgebildet, so dass sie im Ansichtsfenster nach oben zeigt. Der UP-Vektor darf nicht parallel zur Sichtlinie vom Augenpunkt zum Referenzpunkt verlaufen.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`Ex` gibt die x-Position des Augenpunktes an

`Ey` gibt die y-Position des Augenpunktes an

`Ez` gibt die z-Position des Augenpunktes an

`Lx` gibt die x-Position des Referenzpunktes an

`Ly` gibt die y-Position des Referenzpunktes an

`Lz` gibt die z-Position des Referenzpunktes an

`Ux` gibt die x-Richtung des Aufwärtsvektors an

`Uy` gibt die y-Richtung des Aufwärtsvektors an

`Uz` gibt die z-Richtung des Aufwärtsvektors an

7.8 glu.NewNurbsRenderer

BEZEICHNUNG

glu.NewNurbsRenderer – legt ein NURBS-Objekt an

ÜBERSICHT

```
nurb = glu.NewNurbsRenderer()
```

BESCHREIBUNG

`glu.NewNurbsRenderer()` erstellt und gibt einen Zeiger auf ein neues NURBS-Objekt zurück. Auf dieses Objekt muss beim Aufruf der NURBS-Rendering- und Steuerungsfunktionen verwiesen werden.

NURBS-Objekte werden vom Speicherbereiniger von Hollywood automatisch gelöscht, wenn sie nicht mehr verwendet werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

RÜCKGABEWERTE

`nurb` ein neues NURBS-Objekt

7.9 glu.NewQuadric

BEZEICHNUNG

glu.NewQuadric – erstellt ein quadrisches Objekt

ÜBERSICHT

```
quad = glu.NewQuadric()
```

BESCHREIBUNG

`glu.NewQuadric()` erstellt und gibt ein neues quadrisches Objekt zurück. Auf dieses Objekt muss beim Aufruf der Quadrics-Rendering- und Steuerungsfunktionen verwiesen werden.

Quadrics-Objekte werden vom Speicherbereiniger von Hollywood automatisch gelöscht, wenn sie nicht mehr verwendet werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

RÜCKGABEWERTE

`quad` ein neues quadrisches Objekt

7.10 glu.Ortho2D

BEZEICHNUNG

glu.Ortho2D – definiert eine 2D orthographische Projektionsmatrix

ÜBERSICHT

```
glu.Ortho2D(left, right, bottom, top)
```

BESCHREIBUNG

`glu.Ortho2D()` richtet einen zweidimensionalen orthographischen Betrachtungsbereich ein. Dies entspricht dem Aufruf von `gl.Ortho()` mit near = -1 und far = 1.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`left`	gibt die Koordinate für die linken vertikalen Beschneidungsebenen an
`right`	gibt die Koordinate für die rechten vertikalen Beschneidungsebenen an
`bottom`	gibt die Koordinaten für die unteren horizontalen Beschneidungsebenen an
`top`	gibt die Koordinaten für die oberen horizontalen Beschneidungsebenen an

7.11 glu.Perspective

BEZEICHNUNG

glu.Perspective – baut eine perspektivische Projektionsmatrix auf

ÜBERSICHT

```
glu.Perspective(fovy, aspect, near, far)
```

BESCHREIBUNG

`glu.Perspective()` spezifiziert ein Sichtfeld in das Weltkoordinatensystem. Im Allgemeinen sollte das Seitenverhältnis in `glu.Perspective()` dem Seitenverhältnis des zugehörigen Ansichtsfensters entsprechen. `Aspekt` = 2,0 bedeutet beispielsweise, dass der Blickwinkel des Betrachters in x doppelt so breit ist wie in y. Ist das Ansichtsfenster doppelt so breit wie hoch, zeigt es das Bild verzerrungsfrei an.

Die von `glu.Perspective()` erzeugte Matrix wird mit der aktuellen Matrix multipliziert, als ob `gl.MultMatrix()` mit der erzeugten Matrix aufgerufen worden wäre. Um die perspektivische Matrix stattdessen auf den aktuellen Matrixstapel zu laden, setzen Sie dem Aufruf von `glu.Perspective()` den Aufruf von `gl.LoadIdentity()` vorraus.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`fovy`	gibt den Sichtfeldwinkel in Grad in y-Richtung an
`aspect`	gibt das Seitenverhältnis an, das das Sichtfeld in x-Richtung bestimmt; das Seitenverhältnis ist das Verhältnis von x (Breite) zu y (Höhe)
`near`	gibt den Abstand vom Betrachter zur nahen Schnittebene an (immer positiv)
`far`	gibt den Abstand vom Betrachter zur entfernten Schnitt-Ebene an (immer positiv)

7.12 glu.PickMatrix

BEZEICHNUNG

glu.PickMatrix – definiert einen Bereich im Ansichtsfensters

ÜBERSICHT

```
glu.PickMatrix(x, y, deltax, deltay, viewportArray)
```

BESCHREIBUNG

`glu.PickMatrix()` erstellt eine Projektionsmatrix, mit der die Zeichnung auf einen kleinen Bereich des Ansichtsfensters beschränkt werden kann. Dies ist in der Regel nützlich, um festzustellen, welche Objekte in der Nähe des Cursors gezeichnet werden. Verwenden Sie `glu.PickMatrix()`, um die Zeichnung auf einen kleinen Bereich um den Cursor herum zu beschränken. Wechseln Sie dann in den Auswahlmodus (mit `gl.RenderMode()`) und rerendern Sie die Szene. Alle Grundmuster, die in die Nähe des Cursors gezeichnet worden wären, werden identifiziert und im Auswahlpuffer gespeichert.

Die von `glu.PickMatrix()` erzeugte Matrix wird mit der aktuellen Matrix multipliziert, als ob `gl.MultMatrix()` wäre, wird mit der erzeugten Matrix aufgerufen. Um die generierte Kommissioniermatrix für die Kommissionierung effektiv zu nutzen, rufen Sie zuerst `gl.LoadIdentity()` auf, um eine Identitätsmatrix auf den perspektivischen Matrixstapel zu laden. Rufen Sie dann `glu.PickMatrix()` auf und anschließend einen Befehl (z.B. `glu.Perspective()`), um die perspektivische Matrix mit der Pick-Matrix zu multiplizieren.

Wenn Sie `glu.PickMatrix()` zur Auswahl von NURBS verwenden, achten Sie darauf, dass Sie die NURBS-Eigenschaft `#GLU_AUTO_LOAD_MATRIX` deaktivieren. Wenn `#GLU_AUTO_LOAD_MATRIX` nicht ausgeschaltet ist, dann wird jede gerenderte NURBS-Oberfläche mit der Pick-Matrix anders unterteilt als ohne die Pick-Matrix.

Die vier Fensterkoordinaten für `viewportArray` können einfach durch den Aufruf von `gl.Get()` mit `#GL_VIEWPORT` ermittelt werden. Siehe Abschnitt 6.57 [gl.Get], Seite 89, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`x` gibt den x-Mittelpunkt eines Bereichs in Fensterkoordinaten an

`y` gibt den y-Mittelpunkt eines Bereichs in Fensterkoordinaten an

`deltax` gibt die Breite des Bereichs in Fensterkoordinaten an

`deltay` gibt die Höhe des Bereichs in Fensterkoordinaten an

`viewportArray`
die vier Fensterkoordinaten des Ansichtsfensters in einer Tabelle

7.13 glu.Project

BEZEICHNUNG

glu.Project – ordnet Objektkoordinaten den Fensterkoordinaten zu

ÜBERSICHT

```
e,wx,wy,wz = glu.Project(objx, objy, objz, model, proj, view)
```

BESCHREIBUNG

`glu.Project()` transformiert die angegebenen Objektkoordinaten mit `model`, `proj` und `view` in Fensterkoordinaten. Das Ergebnis wird in `wx`, `wy` und `wz` gespeichert. Ein Rückgabewert von `#GL_TRUE` zeigt den Erfolg an, ein Rückgabewert von `#GL_FALSE` zeigt einen Fehler an.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`objx`	gibt die Objekt x-Koordinate an
`objy`	gibt die Objekt y-Koordinate an
`objz`	gibt die Objekt z-Koordinate an
`model`	spezifiziert die aktuelle Modellansichtsmatrix als Tabelle
`proj`	spezifiziert die aktuelle Projektionsmatrix als Tabelle
`view`	spezifiziert das aktuelle Ansichtsfenster als Tabelle

RÜCKGABEWERTE

`e`	Fehlercode
`wx`	berechnet die Fenster-x-Koordinate
`wy`	berechnet die Fenster-y-Koordinate
`wz`	berechnet die Fenster-z-Koordinate

7.14 glu.ScaleImage

BEZEICHNUNG

glu.ScaleImage – skaliert ein Bild auf eine beliebige Größe

ÜBERSICHT

```
err, pixelsOut = glu.ScaleImage(format, widthIn, heightIn, pixelsIn,
                     widthOut, heightOut)
```

BESCHREIBUNG

Dieser Befehl funktioniert genauso wie `glu.ScaleImageRaw()`, mit der Ausnahme, dass die Pixeldaten nicht als Rohspeicherpuffer übergeben und zurückgegeben werden, sondern als Tabelle, die die Anzahl der Elemente width*height*depth enthält, die jeweils ein Pixel beschreiben. Dies ist natürlich nicht so effizient wie die Verwendung von Rohdatenpuffern, da die Pixeldaten der Tabelle zuerst in einen Rohdatenpuffer kopiert und dann wieder auf eine Tabelle abgebildet werden müssen.

Siehe Abschnitt 7.15 [glu.ScaleImageRaw], Seite 240, für weitere Einzelheiten zu den von dieserm Befehl akzeptierten Parametern.

Beachten Sie, dass `glu.ScaleImage()` Daten vom Typ `#GL_FLOAT` innerhalb der Tabelle `pixelsIn` erwartet. `#GL_FLOAT` Pixeldaten werden ebenfalls in die Rückgabetabelle `pixelsOut` geschrieben.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`format` gibt das Format der Pixeldaten an (siehe oben)

`widthIn` gibt die Breite des Quellbildes an, das skaliert wird

`heightIn` gibt die Höhe des Quellbildes an, das skaliert wird

`pixelsIn` spezifiziert eine Tabelle, die die Pixeldaten des Quellbildes enthält

`widthOut` gibt die Breite des Zielbildes an

`heightOut`
gibt die Höhe des Zielbildes an

RÜCKGABEWERTE

`error` Fehlercode oder 0 für den Erfolg

`pixelsOut`
Tabelle mit den skalierten Bilddaten

7.15 glu.ScaleImageRaw

BEZEICHNUNG

glu.ScaleImageRaw – skaliert ein Bild auf eine beliebige Größe

ÜBERSICHT

```
error = glu.ScaleImageRaw(format, widthIn, heightIn, typeIn, pixelsIn,
           widthOut, heightOut, typeOut, pixelsOut)
```

BESCHREIBUNG

`glu.ScaleImageRaw()` skaliert ein Pixelbild mit den entsprechenden Pixelspeichermodi, um Daten aus dem Quellbild zu entpacken und Daten in das Zielbild zu packen.

Wenn Sie ein Bild verkleinern, verwendet `glu.ScaleImageRaw()` einen Boxfilter, um das Quellbild abzutasten und Pixel für das Zielbild zu erstellen. Wenn ein Bild vergrößert wird, werden die Pixel aus dem Quellbild linear interpoliert, um das Zielbild zu erzeugen.

`format` muss eine der folgenden Konstanten sein: `#GL_COLOR_INDEX`, `#GL_STENCIL_INDEX`, `#GL_DEPTH_COMPONENT`, `#GL_RED`, `#GL_GREEN`, `#GL_BLUE`, `#GL_ALPHA`, `#GL_RGB`, `#GL_RGBA`, `#GL_LUMINANCE` und `#GL_LUMINANCE_ALPHA`.

`typeIn` und `typeOut` muss eine der folgenden sein: `#GL_UNSIGNED_BYTE`, `#GL_BYTE`, `#GL_BITMAP`, `#GL_UNSIGNED_SHORT`, `#GL_SHORT`, `#GL_UNSIGNED_INT`, `#GL_INT` oder `#GL_FLOAT`.

Ein Rückgabewert von 0 bedeutet Erfolg. Andernfalls gibt glu.ScaleImage() einen GLU-Fehlercode zurück, der anzeigt, was das Problem ist. Siehe Abschnitt 7.5 [glu.ErrorString], Seite 234, für Details.

Bitte beachten Sie, dass dieser Befehl direkt mit Speicherzeigern arbeitet. Es gibt auch eine Version, die mit Tabellen anstelle von Speicherzeigern arbeitet, aber das ist natürlich langsamer. Siehe Abschnitt 7.14 [glu.ScaleImage], Seite 239, für Details. Siehe

Abschnitt 3.7 [Mit Zeigern arbeiten], Seite 13, für Einzelheiten zur Verwendung von Speicherzeigern mit Hollywood.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`format` gibt das Format der Pixeldaten an (siehe oben)

`widthIn` gibt die Breite des Quellbildes an, das skaliert wird

`heightIn` gibt die Höhe des Quellbildes an, das skaliert wird

`typeIn` gibt den Datentyp für `pixelsIn` an (siehe oben)

`pixelsIn` spezifiziert einen Zeiger auf das Quellbild

`widthOut` gibt die Breite des Zielbildes an

`heightOut` gibt die Höhe des Zielbildes an

`typeOut` gibt den Datentyp für `pixelsOut` an (siehe oben)

`pixelsOut` gibt einen Zeiger auf das Zielbild an

RÜCKGABEWERTE

`error` Fehlercode oder 0 für den Erfolg

7.16 glu.UnProject

BEZEICHNUNG

glu.UnProject – ordnet die Fensterkoordinaten den Objektkoordinaten zu

ÜBERSICHT

```
e,objx,objy,objz = glu.UnProject(winx, winy, winz, model, proj, view)
```

BESCHREIBUNG

`glu.UnProject()` bildet die angegebenen Fensterkoordinaten mit `model`, `proj` und `view` auf Objektkoordinaten ab. Das Ergebnis wird in `objx`, `objy` und `objz` gespeichert. Ein Rückgabewert von `#GL_TRUE` zeigt den Erfolg an; ein Rückgabewert von `#GL_FALSE` bedeutet Fehler.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`winx` gibt die x-Koordinate des Fensters an

`winy` gibt die y-Koordinate des Fensters an

`winz` gibt die z-Koordinate des Fensters an

`model` gibt die aktuelle Modellansichtsmatrix als Tabelle an

`proj` gibt die aktuelle Projektionsmatrix als Tabelle an

`view` gibt das aktuelle Ansichtsfenster als Tabelle an

RÜCKGABEWERTE

`e`	Fehlercode
`objx`	berechnet die Objekt x Koordinate
`objy`	berechnet die Objekt y Koordinate
`objz`	berechnet die Objekt z Koordinate

7.17 nurb:BeginCurve

BEZEICHNUNG

nurb:BeginCurve – makiert den Anfang einer NURBS-Kurvendefinition

ÜBERSICHT

`nurb:BeginCurve()`

BESCHREIBUNG

Verwenden Sie nurb:BeginCurve(), um den Anfang einer NURBS-Kurvendefinition zu markieren. Nachdem Sie nurb:BeginCurve() aufgerufen haben, rufen Sie ein oder mehrere `nurb:Curve()` auf, um die Attribute der Kurve zu definieren. Genau einer der Aufrufe von `nurb:Curve()` muss einen Kurventyp von `#GLU_MAP1_VERTEX_3` oder `#GLU_MAP1_VERTEX_4` haben. Um das Ende der NURBS-Kurvendefinition zu markieren, rufen Sie `nurb:EndCurve()` auf.

GL-Evaluatoren werden eingesetzt, um die NURBS-Kurve als eine Reihe von Liniensegmenten darzustellen. Der Status des Evaluators bleibt während des Renderings mit `gl.PushAttrib()` (`#GLU_EVAL_BIT`) und `gl.PopAttrib()` erhalten. Siehe Abschnitt 6.116 [gl.PushAttrib], Seite 179, für Einzelheiten darüber, welchen Status diese Aufrufe bewahren.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

7.18 nurb:BeginSurface

BEZEICHNUNG

nurb:BeginSurface – markiert den Anfang einer NURBS-Flächendefinition

ÜBERSICHT

`nurb:BeginSurface()`

BESCHREIBUNG

Verwenden Sie nurb:BeginSurface(), um den Anfang einer NURBS-Flächendefinition zu markieren. Nachdem Sie nurb:BeginSurface() aufgerufen haben, führen Sie einen oder mehrere Aufrufe mit `nurb:Surface()` durch, um die Attribute der Oberfläche zu definieren. Genau einer dieser Aufrufe von `nurb:Surface()` muss einen Oberflächentyp von `#GLU_MAP2_VERTEX_3` oder `#GLU_MAP2_VERTEX_4` haben. Um das Ende der NURBS-Flächendefinition zu markieren, rufen Sie `nurb:EndSurface()` auf.

Das Zuschneiden von NURBS-Oberflächen wird mit `nurb:BeginTrim()`, `nurb:PwlCurve()`, `nurb:Curve()` und `nurb:EndTrim()` unterstützt. Siehe Abschnitt 7.19 [nurb:BeginTrim], Seite 243, für Details.

GL-Evaluatoren werden eingesetzt, um die NURBS-Oberfläche als eine Reihe von Polygonen darzustellen. Der Status des Evaluators bleibt während des Renderings mit `gl.PushAttrib()` (`#GLU_EVAL_BIT`) und `gl.PopAttrib()` erhalten. Siehe Abschnitt 6.116 [gl.PushAttrib], Seite 179, für Einzelheiten darüber, welchen Status diese Aufrufe bewahren.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

7.19 nurb:BeginTrim

BEZEICHNUNG

nurb:BeginTrim – markiert den Anfang einer NURBS-Trimmschleifendefinition

ÜBERSICHT

`nurb:BeginTrim()`

BESCHREIBUNG

Verwenden Sie nurb:BeginTrim(), um den Anfang einer Trimmschleife zu markieren und `nurb:EndTrim()`, um das Ende einer Trimmschleife zu markieren. Eine Trimmschleife ist ein Satz von orientierten Kurvensegmenten (die eine geschlossene Kurve bilden), die die Grenzen einer NURBS-Oberfläche definieren. Diese Trimmschleifen nehmen Sie in die Definition einer NURBS-Oberfläche zwischen den Aufrufen von `nurb:BeginSurface()` und `nurb:EndSurface()` auf.

Die Definition einer NURBS-Oberfläche kann viele Trimmschleifen enthalten. Wenn Sie beispielsweise eine Definition für eine NURBS-Oberfläche geschrieben haben, die einem Rechteck mit ausgestanztem Loch ähnelt, würde die Definition zwei Trimmschleifen enthalten. Eine Schleife definiert den äußeren Rand des Rechtecks, die andere definiert das aus dem Rechteck ausgestanzten Loch. Die Definitionen jeder dieser Trimmschleifen werden von einer nurb:BeginTrim() / `nurb:EndTrim()` Paar eingeklammert.

Die Definition eines einzelnen geschlossenen Trimmkreislaufs kann aus mehreren Kurvensegmenten bestehen, die jeweils als stückweise lineare Kurve (siehe `nurb:PwlCurve()`) oder als einzelne NURBS-Kurve (siehe `nurb:Curve()`) oder als Kombination von beiden in beliebiger Reihenfolge beschrieben werden. Die einzigen Bibliotheksaufrufe, die in einer Trimmschleifendefinition (zwischen den Aufrufen von nurb:BeginTrim() und `nurb:EndTrim()`) auftreten können sind `nurb:PwlCurve()` und `nurb:Curve()`.

Der Bereich der angezeigten NURBS-Oberfläche ist der Bereich in der Domäne links neben der Trimmkurve mit zunehmendem Kurvenparameter. Somit befindet sich der zurückgehaltene Bereich der NURBS-Oberfläche innerhalb einer Trimmschleife im Gegenuhrzeigersinn und außerhalb einer Trimmschleife im Uhrzeigersinn. Für das zuvor genannte Rechteck verläuft die Trimmschleife für die Außenkante des Rechtecks gegen den Uhrzeigersinn, während die Trimmschleife für das ausgestanzte Loch im Uhrzeigersinn verläuft.

Wenn Sie mehr als eine Kurve verwenden, um einen einzelnen Trimmkreis zu definieren, müssen die Kurvensegmente einen geschlossenen Regelkreis bilden (d.h. der Endpunkt jeder Kurve muss der Startpunkt der nächsten Kurve sein und der Endpunkt der Endkurve muss der Startpunkt der ersten Kurve sein). Wenn die Endpunkte der Kurve ausreichend nah beieinander liegen, aber nicht genau übereinstimmen, werden sie zwangsläufig angepasst. Wenn die Endpunkte nicht ausreichend dicht sind, kommt es zu einem Fehler. Siehe Abschnitt 7.20 [nurb:Callback], Seite 244, für Details.

Wenn eine Trimmschleifendefinition mehrere Kurven enthält, muss die Richtung der Kurven konsistent sein (d.h. die Innenseite muss links von allen Kurven liegen). Verschachtelte Trimmschleifen sind legal, solange die Kurvenausrichtungen korrekt wechseln. Wenn sich die Trimmkurven selbst kreuzen oder sich schneiden, kommt es zu einem Fehler.

Wenn für eine NURBS-Oberfläche keine Trimminformationen angegeben werden, wird die gesamte Fläche gezeichnet.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

keine

7.20 nurb:Callback

BEZEICHNUNG

nurb:Callback – definiert eine Callback-Funktion für ein NURBS-Objekt

ÜBERSICHT

```
nurb:Callback(which, func)
```

BESCHREIBUNG

nurb:Callback() wird verwendet, um eine Callback-Funktion zu definieren, die von einem NURBS-Objekt verwendet werden soll. Wenn die angegebene Callback-Funktion bereits definiert ist, wird sie ersetzt. Wenn `func` `Nil` ist, dann wird diese Callback-Funktion nicht aufgerufen und die zugehörigen Daten, falls vorhanden, gehen verloren.

Abgesehen von einer Fehlercallback-Funktion werden diese Callback-Funktionen vom NURBS-Tessellator verwendet (wenn `#GLU_NURBS_MODE` auf `#GLU_NURBS_TESSELLATOR` gesetzt ist), um die aus der Tessellierung resultierenden OpenGL-Polygon-Grundmuster zurückzugeben. Die Fehlercallback-Funktion ist unabhängig davon wirksam, auf welchen Wert `#GLU_NURBS_MODE` gesetzt ist. Alle anderen Callback-Funktionen sind nur wirksam, wenn `#GLU_NURBS_MODE` auf `#GLU_NURBS_TESSELLATOR` gesetzt ist.

Alle Callback-Funktionen erhalten als ersten Parameter ein Zugriffsrecht für das NURBS-Objekt. Der zweite Parameter hängt von der in `which` angegebenen Callback-Art ab.

Die zulässigen Callback-Aktionen stellen sich wie folgt dar:

`#GLU_NURBS_BEGIN`

> Der Start-Callback zeigt den Beginn eines Grundmusters an. Die Funktion erhält ein ganzzahliges Argument, das eines der folgenden sein kann: `#GLU_LINES`, `#GLU_LINE_STRIP`, `#GLU_TRIANGLE_FAN`, `#GLU_TRIANGLE_STRIP`, `#GLU_TRIANGLES` oder `#GLU_QUAD_STRIP`. Die Standard-Callback-Funktion für den Beginn ist `Nil`.

`#GLU_NURBS_VERTEX`
: Der Vertex-Callback zeigt einen Scheitelpunkt des Grundmusters an. Die Koordinaten des Scheitels werden in einem Tabellenparameter als vier Fließkommazahlen übergeben. Alle erzeugten Knoten haben die Dimension 3, d.h. homogene Koordinaten wurden in affine Koordinaten umgewandelt. Die Standardfunktion für den Callback von Knoten ist `Nil`.

`#GLU_NURBS_NORMAL`
: Der Normalen-Callback wird aufgerufen, wenn der Normalen-Knoten erzeugt wird. Die Komponenten der Normalen werden in einem Tabellenparameter als drei Fließkommazahlen übergeben. Im Falle einer NURBS-Kurve ist die Callback-Funktion nur wirksam, wenn der Benutzer eine Normalen-Karte (`#GLU_MAP1_NORMAL`) bereitstellt. Im Falle einer NURBS-Oberfläche, wenn eine Normalen-Karte (`#GLU_MAP2_NORMAL`) bereitgestellt wird, wird der erzeugte Normalenwert aus der Normalen-Karte berechnet. Wenn keine Normalen-Karte zur Verfügung gestellt wird, wird eine Oberflächen-Normale auf eine ähnliche Weise berechnet wie für Evaluatoren beschrieben, wenn `#GLU_AUTO_NORMAL` aktiviert ist. Die Standardfunktion für den Normalen-Callback ist `Nil`.

`#GLU_NURBS_COLOR`
: Der Farb-Callback wird aufgerufen, wenn die Farbe eines Knoten erzeugt wird. Die Komponenten der Farbe werden in einem Tabellenparameter übergeben. Dieser Callback ist nur wirksam, wenn der Benutzer eine Farbkarte zur Verfügung stellt (`#GLU_MAP1_COLOR_4` oder `#GLU_MAP2_COLOR_4`). Die an diesen Callback übergebene Tabelle enthält vier Gleitkomma-Komponenten: R, G, B, A. Die Standardfunktion für den Farbrückruf ist `Nil`.

`#GLU_NURBS_TEXTURE_COORD`
: Der Textur-Callback wird aufgerufen, wenn die Texturkoordinaten eines Knoten erzeugt werden. Diese Koordinaten werden in einem Tabellenparameter übergeben. Die Anzahl der Texturkoordinaten kann 1, 2, 3 oder 4 betragen, je nachdem, welcher Typ von Texturkarte angegeben ist (`#GLU_MAP1_TEXTURE_COORD_1`, `#GLU_MAP1_TEXTURE_COORD_2`, `#GLU_MAP1_TEXTURE_COORD_3`, `#GLU_MAP1_TEXTURE_COORD_4`, `#GLU_MAP2_TEXTURE_COORD_1`, `#GLU_MAP2_TEXTURE_COORD_2`, `#GLU_MAP2_TEXTURE_COORD_3`, `#GLU_MAP2_TEXTURE_COORD_4`). Wenn keine Texturkarte angegeben ist, wird diese Callback-Funktion nicht aufgerufen. Die Standardfunktion für den Textur-Callback ist `Nil`.

`#GLU_NURBS_END`
: Der End-Callback wird am Ende eines Grundmusters aufgerufen. Die Standardfunktion für den End-Callback ist `Nil`. Dieser Callback empfängt keine Parameter außer einem Handler für das NURBS-Objekt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`which` gibt an, dass der Callback definiert wird (siehe oben)

`func` gibt die Funktion an, die der Callback aufruft

7.21 nurb:Curve

BEZEICHNUNG

nurb:Curve – definiert die Form einer NURBS-Kurve

ÜBERSICHT

```
nurb:Curve(knotsArray, controlArray, type)
```

BESCHREIBUNG

Verwenden Sie nurb:Curve(), um eine NURBS-Kurve zu beschreiben.

Wenn nurb:Curve() zwischen einem `nurb:BeginCurve()` / `nurb:EndCurve()` Paar erscheint, wird er verwendet, um eine zu rendernde Kurve zu beschreiben. Positions-, Textur- und Farbkoordinaten werden zugeordnet, indem sie jeweils als separate nurb:Curve() zwischen einem `nurb:BeginCurve()` / `nurb:EndCurve()` Paar dargestellt werden. Es kann nicht mehr als ein Aufruf von nurb:Curve() für jede der Farb-, Positions- und Texturdaten innerhalb eines einzigen `nurb:BeginCurve()` / `nurb:EndCurve()` Paar erfolgen. Es muss genau ein Aufruf zur Beschreibung der Position der Kurve erfolgen (eine Art `#GLU_MAP1_VERTEX_3` oder `#GLU_MAP1_VERTEX_4`).

Wenn nurb:Curve() zwischen einem nurb:BeginTrim() / `nurb:EndTrim()` Paar erscheint, wird er verwendet, um eine Trimmkurve auf einer NURBS-Oberfläche zu beschreiben. Wenn der Typ `#GLU_MAP1_TRIM_2` ist, dann beschreibt er eine Kurve im zweidimensionalen (u und v) Parameterraum. Wenn er `#GLU_MAP1_TRIM_3` ist, dann beschreibt er eine Kurve im zweidimensionalen homogenen (u, v und w) Parameterraum. Siehe Abschnitt 7.19 [nurb:BeginTrim], Seite 243, für weitere Diskussionen über das Trimmen von Kurven.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`knotsArray`
: gibt ein Feld von nicht abnehmenden Knotenwerten an

`controlArray`
: gibt ein Feld von Kontrollpunkten an; die Koordinaten müssen mit dem `type` übereinstimmen, der unten angegeben ist

`type` gibt den Typ der Kurve an (siehe oben)

7.22 nurb:EndCurve

BEZEICHNUNG

nurb:EndCurve – makiert das Ende einer NURBS-Kurvendefinition

ÜBERSICHT

```
nurb:EndCurve()
```

BESCHREIBUNG
nurb:EndCurve() markiert das Ende einer NURBS-Kurvendefinition. Siehe Abschnitt 7.17 [nurb:BeginCurve], Seite 242, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN
keine

7.23 nurb:EndSurface

BEZEICHNUNG
nurb:EndSurface – markiert das Ende einer NURBS-Flächendefinition

ÜBERSICHT
```
nurb:EndSurface()
```

BESCHREIBUNG
nurb:EndSurface() markiert das Ende einer NURBS-Flächendefinition. Siehe Abschnitt 7.18 [nurb:BeginSurface], Seite 242, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN
keine

7.24 nurb:EndTrim

BEZEICHNUNG
nurb:EndTrim – markiert das Ende einer Trimmschleifendefinition

ÜBERSICHT
```
nurb:EndTrim()
```

BESCHREIBUNG
nurb:EndTrim() markiert das Ende einer Trimmschleifendefinition. Siehe Abschnitt 7.19 [nurb:BeginTrim], Seite 243, für Details.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN
keine

7.25 nurb:GetProperty

BEZEICHNUNG
nurb:GetProperty – ermittelt eine NURBS-Eigenschaft

ÜBERSICHT
```
value = nurb:GetProperty(property)
```

BESCHREIBUNG

nurb:GetProperty() kann verwendet werden, um den Status einer NURBS-Eigenschaft abzurufen. Siehe Abschnitt 7.28 [nurb:SetProperty], Seite 249, für eine Liste der akzeptierten Eigenschaften.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`property` gibt die zu ermittelte Eigenschaft an (siehe oben)

RÜCKGABEWERTE

`value` Status der angegebenen Eigenschaft

7.26 nurb:LoadSamplingMatrices

BEZEICHNUNG

nurb:LoadSamplingMatrices – lädt NURBS Abtast- und Selektionsmatrizen

ÜBERSICHT

```
nurb:LoadSamplingMatrices(modelArray, perspectiveArray, viewArray)
```

BESCHREIBUNG

nurb:LoadSamplingMatrices() verwendet `modelArray`, `perspectiveArray` und `viewArray`, um die in `nurb` gespeicherten Abtast- und Selektionsmatrizen neu zu berechnen. Die Abtastmatrix bestimmt, wie fein eine NURBS-Kurve oder -Oberfläche tesselliert werden muss, um die Abtasttoleranz (bestimmt durch die Eigenschaft `#GLU_SAMPLING_TOLERANCE`) zu erfüllen. Die Selektionsmatrix wird verwendet, um zu entscheiden, ob eine NURBS-Kurve oder -Oberfläche vor dem Rendern entfernt werden soll (wenn die Eigenschaft `#GLU_CULLING` eingeschaltet ist).

nurb:LoadSamplingMatrices() ist nur erforderlich, wenn die Eigenschaft `#GLU_AUTO_LOAD_MATRIX` deaktiviert ist. Siehe Abschnitt 7.28 [nurb:SetProperty], Seite 249, für Details. Obwohl es bequem sein kann, die Eigenschaft `#GLU_AUTO_LOAD_MATRIX` eingeschaltet zu lassen, kann es zu einem Leistungsabfall kommen. (Eine Verbindung zum GL-Server ist erforderlich, um die aktuellen Werte der Modelnsicht-Matrix, Projektionsmatrix und des Ansichtsports zu erhalten.)

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`modelArray`
: gibt eine Tabelle an, die eine Modellansichtsmatrix enthält

`perspectiveArray`
: gibt eine Tabelle an, die eine Projektionsmatrix enthält

`viewArray`
: gibt eine Tabelle mit Ansichtenkoordinaten an

7.27 nurb:PwlCurve

BEZEICHNUNG

nurb:PwlCurve – beschreibt eine stückweise lineare NURBS-Trimmkurve

ÜBERSICHT

```
nurb:PwlCurve(dataArray, type)
```

BESCHREIBUNG

nurb:PwlCurve() beschreibt eine stückweise lineare Trimmkurve für eine NURBS-Oberfläche. Eine stückweise lineare Kurve besteht aus einer Liste aus Koordinaten von Punkten im Parameterraum für die zu trimmende NURBS-Fläche. Diese Punkte werden mit Liniensegmenten zu einer Kurve verbunden. Wenn die Kurve eine Annäherung an eine Kurve ist, die nicht stückweise linear ist, sollten die Punkte im Parameterraum so nah beieinander liegen, dass der resultierende Weg mit der in der Anwendung verwendeten Auflösung gekrümmt erscheint.

Wenn `type` `#GLU_MAP1_TRIM_2` ist, dann beschreibt er eine Kurve im zweidimensionalen (u und v) Parameterraum. Wenn es `#GLU_MAP1_TRIM_3` ist, dann beschreibt er eine Kurve im zweidimensionalen homogenen (u, v und w) Parameterraum. Siehe Abschnitt 7.19 [nurb:BeginTrim], Seite 243, für weitere Informationen über das Trimmen von Kurven.

Um eine Trimmkurve zu beschreiben, die den Konturen einer NURBS-Oberfläche eng folgt, rufen Sie `nurb:Curve()` auf.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`dataArray`
: gibt ein Feld an, das die Kurvenpunkte enthält

`type`
: gibt den Typ der Kurve an; muss entweder `#GLU_MAP1_TRIM_2` oder `#GLU_MAP1_TRIM_3` sein

7.28 nurb:SetProperty

BEZEICHNUNG

nurb:SetProperty – setzt eine NURBS-Eigenschaft

ÜBERSICHT

```
nurb:SetProperty(property, value)
```

BESCHREIBUNG

nurb:SetProperty() wird verwendet, um Eigenschaften zu steuern, die in einem NURBS-Objekt gespeichert sind. Diese Eigenschaften beeinflussen die Art und Weise, wie eine NURBS-Kurve dargestellt wird. Die akzeptierten Werte für `property` sind wie folgt:

`#GLU_NURBS_MODE`
: `value` sollte so eingestellt werden, dass es entweder `#GLU_NURBS_RENDERER` oder `#GLU_NURBS_TESSELLATOR` ist. Wenn es auf `#GLU_NURBS_RENDERER` eingestellt ist, werden NURBS-Objekte in OpenGL-Grundmuster tesselliert und zum Rendern an die Leitung gesendet. Wenn es auf

#GLU_NURBS_TESSELLATOR eingestellt ist, werden NURBS-Objekte in OpenGL-Grundmuster tesselliert, die Knoten, Normalen, Farben und/oder Texturen über eine Callback-Schnittstelle abgerufen (siehe nurb:Callback()). Dies ermöglicht es dem Benutzer, das tessellierte Ergebnisse zur weiteren Verarbeitung zwischenzuspeichern. Der Initialwert ist #GLU_NURBS_RENDERER.

#GLU_SAMPLING_METHOD

Gibt an, wie eine NURBS-Oberfläche tesselliert werden soll. value kann #GLU_PATH_LENGTH, #GLU_PARAMETRIC_ERROR, #GLU_DOMAIN_DISTANCE, #GLU_OBJECT_PATH_LENGTH oder #GLU_OBJECT_PARAMETRIC_ERROR sein. Wenn #GLU_PATH_LENGTH gesetzt ist, wird die Oberfläche so gerendert, dass die maximale Länge der Kanten der Tesselierungspolygone in Pixeln nicht größer ist als das, was durch #GLU_SAMPLING_TOLERANCE angegeben wird.

#GLU_PARAMETRIC_ERROR gibt an, dass die Oberfläche so gerendert wird, dass der durch #GLU_PARAMETRIC_TOLERANCE angegebene Wert den maximalen Abstand in Pixeln zwischen den Mosaikpolygonen und den von ihnen angenäherten Oberflächen beschreibt.

#GLU_DOMAIN_DISTANCE ermöglicht es dem Benutzer, in parametrischen Koordinaten anzugeben, wie viele Abtastpunkte pro Längeneinheit in u-, v-Richtung genommen werden.

#GLU_OBJECT_PATH_LENGTH ist ähnlich wie #GLU_PATH_LENGTH, nur dass die Ansicht unabhängig ist, d.h. die Oberfläche wird so gerendert, dass die maximale Länge der Kanten der Tesselierungspolygone im Objektraum nicht größer ist als das, was durch #GLU_SAMPLING_TOLERANCE angegeben wird.

#GLU_OBJECT_PARAMETRIC_ERROR ist ähnlich wie #GLU_PARAMETRIC_ERROR, mit der Ausnahme, dass die Ansicht unabhängig ist, d.h. die Oberfläche wird so gerendert, dass der durch #GLU_PARAMETRIC_TOLERANCE angegebene Wert den maximalen Abstand im Objektraum zwischen den Mosaikpolygonen und den von ihnen angenäherten Oberflächen beschreibt.

Der Initialwert von #GLU_SAMPLING_METHOD ist #GLU_PATH_LENGTH.

#GLU_SAMPLING_TOLERANCE

Gibt die maximale Länge in Pixel oder in Objektraumlängeneinheit an, die verwendet werden soll, wenn die Abtastmethode auf #GLU_PATH_LENGTH oder #GLU_OBJECT_PATH_LENGTH gesetzt ist. Der NURBS-Code ist bei der Darstellung einer Kurve oder Fläche konservativ, so dass die tatsächliche Länge etwas kürzer sein kann. Der Anfangswert beträgt 50,0 Pixel.

#GLU_PARAMETRIC_TOLERANCE

Gibt den maximalen Abstand in Pixel oder in Objektraumlängeneinheit an, der verwendet werden soll, wenn die Abtastmethode #GLU_PARAMETRIC_ERROR oder #GLU_OBJECT_PARAMETRIC_ERROR ist. Der Anfangswert ist 0,5.

#GLU_U_STEP

Gibt die Anzahl der Abtastpunkte pro Längeneinheit entlang der u-Achse in parametrischen Koordinaten an. Sie wird benötigt, wenn #GLU_SAMPLING_METHOD auf #GLU_DOMAIN_DISTANCE gesetzt ist. Der Anfangswert ist 100.

`#GLU_V_STEP`

Gibt die Anzahl der Abtastpunkte pro Längeneinheit entlang der v-Achse in parametrischer Koordinate an. Sie wird benötigt, wenn `#GLU_SAMPLING_METHOD` auf `#GLU_DOMAIN_DISTANCE` gesetzt ist. Der Anfangswert ist 100.

`#GLU_DISPLAY_MODE`

`value` kann auf `#GLU_OUTLINE_POLYGON`, `#GLU_FILL` oder `#GLU_OUTLINE_PATCH` gesetzt werden. Wenn `#GLU_NURBS_MODE` auf `#GLU_NURBS_RENDERER` gesetzt ist, definiert `value`, wie eine NURBS-Oberfläche gerendert werden soll. Wenn `value` auf `#GLU_FILL` gesetzt ist, wird die Oberflächen-Option als ein Satz von Polygonen dargestellt. Wenn der Wert auf `#GLU_OUTLINE_POLYGON` gesetzt ist, zeichnet die NURBS-Bibliothek nur die Umrisse der Polygone, die durch Tesselierung erzeugt wurden. Wenn der Wert auf `#GLU_OUTLINE_PATCH` gesetzt ist, werden nur die Umrisse vom Benutzer definierten Patches und Trimmkurven gezeichnet.

Wenn `#GLU_NURBS_MODE` auf `#GLU_NURBS_TESSELLATOR` gesetzt ist, definiert `value`, wie eine NURBS-Oberfläche tesselliert werden soll. Wenn `#GLU_DISPLAY_MODE` auf `#GLU_FILL` oder `#GLU_OUTLINE_POLYGON` gesetzt ist, wird die NURBS-Oberfläche in OpenGL-Dreieck-Grundmuster tesselliert, die durch Callback-Funktionen zurückgeholt werden können. Wenn `#GLU_DISPLAY_MODE` auf `#GLU_OUTLINE_PATCH` gesetzt ist, werden nur die Umrisse der Patches und Trimmkurven als eine Folge von Linienstreifen erzeugt, die über Callback-Funktionen abgerufen werden können.

Der Initialwert ist `#GLU_FILL`.

`#GLU_CULLING`

`value` ist ein boolescher Wert, der, wenn er auf `#GLU_TRUE` gesetzt ist, anzeigt, dass eine NURBS-Kurve vor der Tesselierung verworfen werden sollte, wenn ihre Kontrollpunkte außerhalb des aktuellen Ansichtsfensters liegen. Der Initialwert ist `#GLU_FALSE`.

`#GLU_AUTO_LOAD_MATRIX`

`value` ist ein boolescher Wert. Wenn er auf `#GLU_TRUE` gesetzt ist, lädt der NURBS-Code die Projektionsmatrix, die Modellansichtsmatrix und das Ansichtsfenster vom GL-Server herunter, um Sample- und Selektionsmatrizen für jede gerenderte NURBS-Kurve zu berechnen. Sample- und Selektionsmatrizen sind erforderlich, um die Tesselierung einer NURBS-Oberfläche in Liniensegmente oder Polygone zu bestimmen und eine NURBS-Oberfläche zu selektieren, wenn sie außerhalb des Darstellungsbereichs liegt.

Wenn dieser Modus auf `#GLU_FALSE` eingestellt ist, muss das Programm eine Projektionsmatrix, eine Modelansichts-Matrix, bereitstellen und ein Ansichtsfenster für den NURBS-Renderer, mit dem Sample- und Selektionsmatrizen konstruiert werden können. Dies kann mit dem Befehl `nurb:LoadSamplingMatrices()` geschehen. Dieser Modus ist zunächst auf `#GLU_TRUE` eingestellt. Das Ändern von `#GLU_TRUE` auf `#GLU_FALSE` hat keinen Einfluss auf die Sample- und Selektionsmatrizen bis `nurb:LoadSamplingMatrices()` aufgerufen wird.

Wenn `#GLU_AUTO_LOAD_MATRIX True` ist, kann es vorkommen, dass das Samplen und Selektieren falsch ausgeführt wird, wenn NURBS-Routinen in eine Display-Liste kompiliert werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`property` gibt die zu setzende Eigenschaft an (siehe oben)

`value` gibt den Wert der angegebenen Eigenschaft an (siehe oben)

7.29 nurb:Surface

BEZEICHNUNG

nurb:Surface – definiert die Form einer NURBS-Oberfläche

ÜBERSICHT

```
nurb:Surface(sKnotsArray, tKnotsArray, controlArray, type)
```

BESCHREIBUNG

Verwenden Sie nurb:Surface() innerhalb einer NURBS (Nicht-Uniforme Rationale B-Splines) Oberflächendefinition, um die Form einer NURBS-Oberfläche zu beschreiben (vor jedem Trimmen). Um den Anfang einer NURBS-Flächendefinition zu markieren, verwenden Sie den Befehl `nurb:BeginSurface()`. Um das Ende einer NURBS-Flächendefinition zu markieren, verwenden Sie den Befehl `nurb:EndSurface()`. Rufen Sie nurb:Surface() nur innerhalb einer NURBS-Flächendefinition auf.

Positions-, Textur- und Farbkoordinaten sind einer Oberfläche zugeordnet, indem sie jeweils als separate nurb:Surface() zwischen einem `nurb:BeginSurface()` / `nurb:EndSurface()` Paar dargestellt werden. Nicht mehr als ein Anruf von nurb:Surface() für jede der Farb-, Positions- und Texturdaten kann innerhalb eines einzigen `nurb:BeginSurface()` / `nurb:EndSurface()` Paar erstellt werden. Es muss genau ein Aufruf zur Beschreibung der Position der Oberfläche erfolgen (ein Typ von `#GLU_MAP2_VERTEX_3` oder `#GLU_MAP2_VERTEX_4`).

Eine NURBS-Oberfläche kann mit den Befehlen `nurb:Curve()` und `nurb:PwlCurve()` zwischen `nurb:BeginTrim()` und `nurb:EndTrim()` aufgerufen werden.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`sKnotsArray`
: gibt ein Feld von nicht abnehmenden Knotenwerten in der parametrischen u-Richtung an

`tKnotsArray`
: gibt ein Feld von nicht abnehmenden Knotenwerten in der parametrischen v-Richtung an

`controlArray`
: gibt ein Feld an, das Kontrollpunkte für die NURBS-Oberfläche enthält

`type` gibt den Typ der Oberfläche an; kann jeder der gültigen zweidimensionalen Evaluierertypen sein (z.B. `#GLU_MAP2_VERTEX_3` oder `#GLU_MAP2_COLOR_4`)

7.30 quad:Cylinder

BEZEICHNUNG

quad:Cylinder – zeichnet einen Zylinder

ÜBERSICHT

```
quad:Cylinder(base, top, height, slices, stacks)
```

BESCHREIBUNG

quad:Cylinder() zeichnet einen Zylinder, der entlang der z-Achse ausgerichtet ist. Der Boden des Zylinders ist bei z = 0 und die Oberseite bei z = height platziert. Ein Zylinder ist wie eine Kugel um die z-Achse in Scheiben und entlang der z-Achse in Stapel unterteilt.

Beachten Sie, dass, wenn der obere Wert auf 0,0 gesetzt ist, diese Routine einen Kegel erzeugt.

Wenn die Ausrichtung auf `#GLU_OUTSIDE` eingestellt ist (mit `quad:Orientation()`), dann zeigen alle erzeugten Normalen weg von der z-Achse. Andernfalls zeigen sie auf die z-Achse.

Wenn die Texturierung mit `quad:Texture()` eingeschaltet ist, dann werden Texturkoordinaten so erzeugt, dass t linear von 0,0 zu z = 0 bis 1,0 zu z = height reicht und s von 0,0 zu der +y-Achse, bis 0,25 bei der +x-Achse, bis 0,5 bei der -y-Achse, bis 0,75 bei der -x-Achse und zurück bis 1,0 bei der +y-Achse.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`base`	gibt den Radius des Zylinders bei z = 0 an
`top`	gibt den Radius des Zylinders bei z = height an
`height`	gibt die Höhe des Zylinders an
`slices`	gibt die Anzahl der Unterteilungen um die z-Achse an
`stacks`	gibt die Anzahl der Unterteilungen entlang der z-Achse an

7.31 quad:Disk

BEZEICHNUNG

quad:Disk – zeichnet eine Scheibe

ÜBERSICHT

```
quad:Disk(inner, outer, slices, loops)
```

BESCHREIBUNG

quad:Disk() rendert eine Scheibe auf der Ebene z = 0. Die Scheibe hat einen Radius von `outer` und enthält ein konzentrisches kreisförmiges Loch mit einem Radius von `inner`. Wenn `inner` 0 ist, dann wird kein Loch erzeugt. Die Scheibe wird um die z-Achse herum

in Scheiben (wie Pizzascheiben) und auch um die z-Achse in Ringe unterteilt (wie in `slices` und `loops` angegeben ist).

In Bezug auf die Ausrichtung wird die +z-Seite der Scheibe als "außen" betrachtet (siehe `quad:Orientation()`). Das bedeutet, dass, wenn die Ausrichtung auf `#GLU_OUTSIDE` gesetzt ist, alle Normalen, die erzeugt wurden, entlang der +z-Achse liegen. Andernfalls zeigen sie auf die Achse -z.

Wenn die Texturierung mit `quad:Texture()` eingeschaltet ist, werden Texturkoordinaten linear so erzeugt, dass bei r = outer der Wert bei(r, 0, 0) (1, 0.5) ist, bei (0, r, 0) ist es (0.5, 1), bei (-r, 0, 0) ist es (0, 0.5) und bei (0, -r, 0) ist es (0.5, 0).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`inner` gibt den Innenradius der Scheibe an (kann 0 sein)

`outer` gibt den äußeren Radius der Scheibe an

`slices` gibt die Anzahl der Unterteilungen um die z-Achse an

`loops` gibt die Anzahl der konzentrischen Ringe über den Ursprung an, in den die Scheibe unterteilt ist

7.32 quad:DrawStyle

BEZEICHNUNG

quad:DrawStyle – gibt den gewünschten Zeichenstil für Vierecke an

ÜBERSICHT

`quad:DrawStyle(draw)`

BESCHREIBUNG

quad:DrawStyle() gibt den Zeichenstil für Vierecke an, die mit `quad` gerendert wurden. Die zulässigen Werte sind wie folgt:

`#GLU_FILL`
: Vierecke werden mit Polygon-Grundmuster gerendert. Die Polygone werden im Gegenuhrzeigersinn in Bezug auf ihre Normalen (wie mit `quad:Orientation()`) definiert.

`#GLU_LINE`
: Vierecke werden als ein Satz von Linien dargestellt.

`#GLU_SILHOUETTE`
: Vierecke werden als ein Satz von Linien dargestellt, mit der Ausnahme, dass Kanten, die koplanare Flächen trennen, nicht gezeichnet werden.

`#GLU_POINT`
: Vierecke werden als eine Reihe von Punkten dargestellt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`draw` gibt den gewünschten Zeichenstil an (siehe oben)

7.33 quad:Normals

BEZEICHNUNG

quad:Normals – gibt an, welche Art von Normalen für Vierecke gewünscht werden

ÜBERSICHT

```
quad:Normals(normal)
```

BESCHREIBUNG

quad:Normals() gibt an, welche Art von Normalen für Vierecke gewünscht werden, die mit `quad` gerendert wurden. Die zulässigen Werte sind wie folgt:

`#GLU_NONE`
: Es werden keine Normalen generiert.

`#GLU_FLAT`
: Für jede Facette eines Vierecks wird eine Normale erzeugt.

`#GLU_SMOOTH`
: Für jeden Scheitelpunkt eines Vierecks wird eine Normale erzeugt. Dies ist der Anfangswert.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`normal` gibt die gewünschte Art von Normalen an (siehe oben)

7.34 quad:Orientation

BEZEICHNUNG

quad:Orientation – gibt die Innen-/Außenausrichtung für Vierecke an

ÜBERSICHT

```
quad:Orientation(orientation)
```

BESCHREIBUNG

quad:Orientation() gibt an, welche Art von Ausrichtung für Vierecke gewünscht wird, die mit `quad` gerendert wurden. Die Ausrichtungswerte sind wie folgt:

`#GLU_OUTSIDE`
: Vierecke werden mit nach außen gerichteten Normalen (dem Anfangswert) gezeichnet.

`#GLU_INSIDE`
: Die Vierecke sind mit nach innen gerichteten Normalen gezeichnet.

Beachten Sie, dass die Interpretation von außen und innen vom gezeichneten Viereck abhängt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`orientation`
: gibt die gewünschte Ausrichtung an (siehe oben)

7.35 quad:PartialDisk

BEZEICHNUNG

quad:PartialDisk – zeichnet den Bogen einer Scheibe

ÜBERSICHT

```
quad:PartialDisk(inner, outer, slices, loops, start, sweep)
```

BESCHREIBUNG

quad:PartialDisk() rendert eine Teilscheibe auf der Ebene z = 0. Eine Teilscheibe ist einer vollen Scheibe ähnlich, mit der Ausnahme, dass nur die Teilmenge der Scheibe von `start` bis `start + sweep` enthalten ist (wobei 0 Grad entlang der +y-Achse, 90 Grad entlang der +x-Achse, 180 Grad entlang der -y-Achse und 270 Grad entlang der -x-Achse liegen).

Die Teilscheibe hat einen Radius von `outer` und enthält eine konzentrische kreisförmige Bohrung mit einem Radius von `inner`. Wenn `inner` 0 ist, dann wird kein Loch erzeugt. Die Teilscheibe wird um die z-Achse herum in Scheiben (wie Pizzascheiben) um die z-Achse in Ringe unterteilt (wie durch `slices` und `loops` festgelegt).

In Bezug auf die Ausrichtung wird die +z-Seite der Teilscheibe als außerhalb betrachtet (siehe `quad:Orientation()`). Das bedeutet, dass wenn die Ausrichtung auf `#GLU_OUTSIDE` gesetzt ist, alle erzeugten Normalen entlang der +z-Achse liegen. Andernfalls zeigen sie auf die -z Achse.

Wenn die Texturierung mit `quad:Texture()` eingeschaltet ist, werden Texturkoordinaten linear so erzeugt, dass bei r = outer der Wert bei (r, 0, 0) (1.0, 0.5) ist, bei (0, r, 0) ist es (0.5, 1.0), bei (-r, 0, 0) ist es (0.0, 0.5) und bei (0, -r, 0) ist es (0.5, 0.0).

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`inner`	gibt den Innenradius der Teilscheibe an (kann 0 sein)
`outer`	gibt den Außenradius der Teilscheibe an
`slices`	gibt die Anzahl der Unterteilungen um die z-Achse an
`loops`	gibt die Anzahl der konzentrischen Ringe um den Ursprung an, in den die Teilscheibe unterteilt ist
`start`	gibt den Startwinkel des Scheibenabschnitts in Grad an
`sweep`	gibt den Schwenkwinkel des Scheibenabschnitts in Grad an

7.36 quad:Texture

BEZEICHNUNG

quad:Texture – gibt an, ob die Texturierung für Vierecke erwünscht ist

ÜBERSICHT

```
quad:Texture(texture)
```

BESCHREIBUNG

quad:Texture() gibt an, ob Texturkoordinaten für Vierecke generiert werden sollen, die mit `quad` gerendert wurden. Wenn der Wert der Textur `#GLU_TRUE` ist, dann werden Texturkoordinaten generiert und wenn die Textur `#GLU_FALSE` ist, dann nicht. Der Ausgangswert ist `#GLU_FALSE`.

Die Art und Weise, wie Texturkoordinaten erzeugt werden, hängt von den angegebenen Vierecken ab.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`texture` spezifiziert ein Flag, das angibt, ob Texturkoordinaten generiert werden sollen

7.37 quad:Sphere

BEZEICHNUNG

quad:Sphere – zeichnet eine Kugel

ÜBERSICHT

```
quad:Sphere(radius, slices, stacks)
```

BESCHREIBUNG

quad:Sphere() zeichnet eine Kugel mit dem angegebenen Radius, die um den Ursprung zentriert ist. Die Kugel wird um die z-Achse in Scheiben und entlang der z-Achse in Stapel unterteilt (ähnlich den Längen- und Breitengradlinien).

Wenn die Ausrichtung auf `#GLU_OUTSIDE` (mit `quad:Orientation()`) eingestellt ist, dann zeigen alle erzeugten Normalen weg vom Zentrum, Andernfalls auf das Zentrum der Kugel.

Wenn die Texturierung mit `quad:Texture()` eingeschaltet ist, dann werden Texturkoordinaten so erzeugt, dass t im Bereich von 0,0 bei z = -Radius bis 1,0 bei z = Radius (t nimmt linear entlang der Längslinien zu) und s im Bereich von 0,0 bei der +y-Achse, bis 0,25 bei der +x-Achse, bis 0,5 bei der -y-Achse, bis 0,75 bei der -x-Achse und zurück bis 1,0 bei der +y-Achse liegt.

Weitere Informationen finden Sie in einem OpenGL-Referenzhandbuch.

EINGABEN

`radius` gibt den Radius der Kugel an

`slices` gibt die Anzahl der Unterteilungen um die z-Achse an (ähnlich wie bei Längengradlinien)

`stacks` gibt die Anzahl der Unterteilungen entlang der z-Achse an (ähnlich wie bei den Breitengradlinien)

8 GLFW Referenz

8.1 glfw.GetJoystickAxes

BEZEICHNUNG

glfw.GetJoystickAxes – ermittelt die Zustände aller Joystickachsen (V1.1)

ÜBERSICHT

```
t, count = glfw.GetJoystickAxes(port)
```

BESCHREIBUNG

Dieser Befehl gibt die Werte aller Achsen des angegebenen Joysticks in einer Tabelle zurück. Jedes Element in der Tabelle ist ein Wert zwischen -1,0 und 1,0. Das Argument `port` muss auf eine gültige Joystickkennung zwischen `#GLFW_JOYSTICK_1` und `#GLFW_JOYSTICK_16` gesetzt werden.

EINGABEN

`port`	der Joystick-Port, der abgefragt wird

RÜCKGABEWERTE

`t`	Tabelle mit Achsdaten
`count`	Anzahl der Einträge in der Tabelle

8.2 glfw.GetJoystickButtons

BEZEICHNUNG

glfw.GetJoystickButtons – ruft die Zuständen aller Joystick-Tasten ab (V1.1)

ÜBERSICHT

```
t, count = glfw.GetJoystickButtons(port)
```

BESCHREIBUNG

Dieser Befehl gibt die Zustände aller Tasten des angegebenen Joysticks in einer Tabelle zurück. Jedes Element in der Tabelle ist entweder `True`, wenn die Taste gedrückt wird, oder `False`, wenn sie nicht gedrückt wird. Das Argument `port` muss auf eine gültige Joystickkennung zwischen `#GLFW_JOYSTICK_1` und `#GLFW_JOYSTICK_16` gesetzt werden.

EINGABEN

`port`	der Joystick-Port, der abgefragt wird

RÜCKGABEWERTE

`t`	Tabelle mit den Schaltflächenzuständen
`count`	Anzahl der Einträge in der Tabelle

8.3 glfw.GetJoystickName

BEZEICHNUNG

glfw.GetJoystickName – gibt den Namen des Joysticks zurück (V1.1)

ÜBERSICHT

```
name$ = glfw.GetJoystickName(port)
```

BESCHREIBUNG

Dieser Befehl gibt den Namen des angegebenen Joysticks zurück. Das Argument `port` muss auf eine gültige Joystickkennung zwischen `#GLFW_JOYSTICK_1` und `#GLFW_JOYSTICK_16gesetzt` werden.

EINGABEN

`port`	der Joystick-Port, der abgefragt wird

RÜCKGABEWERTE

`name$`	Name des Joysticks

8.4 glfw.JoystickPresent

BEZEICHNUNG

glfw.JoystickPresent – überprüft, ob sich ein Joystick am angegebenen Port befindet (V1.1)

ÜBERSICHT

```
bool = glfw.JoystickPresent(port)
```

BESCHREIBUNG

Dieser Befehl gibt zurück, ob der angegebene Joystick vorhanden ist. Das Argument `port` muss auf eine gültige Joystickkennung zwischen `#GLFW_JOYSTICK_1` und `#GLFW_JOYSTICK_16` gesetzt werden.

EINGABEN

`port`	der Joystick-Port, der abgefragt wird

RÜCKGABEWERTE

`bool`	`True` oder `False` abhängig davon, ob sich ein Joystick am angegebenen Port befindet

9 Hollywood bridge

9.1 gl.BitmapFromBrush

BEZEICHNUNG

gl.BitmapFromBrush – zeichnet eine Bitmap aus der Maske eines Pinsels

ÜBERSICHT

```
gl.BitmapFromBrush(xorig, yorig, xmove, ymove, id)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.Bitmap()`, mit der Ausnahme, dass die Pixeldaten aus der durch `id` angegebene Maske des Hollywood-Pinsels geholt werden. Wenn der in `id` angegebene Pinsel keine Maske hat, wird ein Fehler erzeugt.

Siehe Abschnitt 6.7 [gl.Bitmap], Seite 31, für Details.

EINGABEN

`xorig`	gibt die Position des x-Ursprungs im Bitmap-Bild an. Der Ursprung wird in der linken unteren Ecke der Bitmap gemessen, wobei rechts und oben die positiven Achsen sind
`yorig`	gibt die Position des y-Ursprungs im Bitmap-Bild an. Der Ursprung wird in der linken unteren Ecke der Bitmap gemessen, wobei rechts und oben die positiven Achsen sind
`xmove`	gibt den x-Versatz an, der der aktuellen Rasterposition hinzugefügt werden soll, nachdem die Bitmap gezeichnet wurde
`ymove`	gibt den y-Versatz an, der der aktuellen Rasterposition hinzugefügt werden soll, nachdem die Bitmap gezeichnet wurde
`id`	Identifikator eines Hollywood-Pinsels, der eine Maske hat

9.2 gl.DrawPixelsFromBrush

BEZEICHNUNG

gl.DrawPixelsFromBrush – zeichnet einen Hollywood-Pinsel in den Rahmen-Puffer

ÜBERSICHT

```
gl.DrawPixelsFromBrush(id)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.DrawPixels()`, mit der Ausnahme, dass die Pixeldaten aus dem in `id` angegebenen Hollywood-Pinsel geholt werden.

Siehe Abschnitt 6.36 [gl.DrawPixels], Seite 67, für Details.

EINGABEN

`id`	Identifikator eines Hollywood-Pinsels

9.3 gl.GetCurrentContext

BEZEICHNUNG

gl.GetCurrentContext – ruft den aktuellen OpenGL-Kontext ab

ÜBERSICHT

```
id = gl.GetCurrentContext()
```

BESCHREIBUNG

Dieser Befehl gibt den Identifikator des Displays zurück, deren OpenGL-Kontext der aktuelle ist.

Sie können `gl.SetCurrentContext()` verwenden, um den aktuellen GL-Kontext einzustellen.

EINGABEN

keine

RÜCKGABEWERTE

`id` Identifikator des Displays, deren OpenGL-Kontext der aktuelle ist

9.4 gl.GetTexImageToBrush

BEZEICHNUNG

gl.GetTexImageToBrush – gibt ein Texturbild als Pinsel zurück

ÜBERSICHT

```
[id] = gl.GetTexImageToBrush(target, level, id)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.GetTexImage()`, mit der Ausnahme, dass die Pixeldaten in den in `id` angegebenen Hollywood-Pinsel umgewandelt werden. Wenn es bereits einen Pinsel gibt, der den Identifikator `id` verwendet, wird er zuerst gelöscht. Wenn Sie `Nil` im Argument `id` angeben, wählt `gl.GetTexImageToBrush()` automatisch einen freien Identifikator für diesen Pinsel aus und gibt ihn zurück.

Siehe Abschnitt 6.71 [gl.GetTexImage], Seite 120, für Details.

EINGABEN

`target` gibt an, welche Texturart verwendet werden soll (muss `#GL_TEXTURE_1D` oder `#GL_TEXTURE_2D` sein)

`level` gibt die Detaillierungsstufe des gewünschten Bildes an; Stufe 0 ist die Basis-Bildebene; Stufe n ist das n-te Mipmap-Reduktionsbild

`id` id für den neuen Pinsel oder `Nil` für die automatische ID-Auswahl

RÜCKGABEWERTE

`id` optional: Identifikator des Pinsels; wird nur zurückgegeben, wenn Sie `Nil` im Argument `id` übergeben (siehe oben)

9.5 gl.ReadPixelsToBrush

BEZEICHNUNG

gl.ReadPixelsToBrush – liest Pixel aus dem Rahmenpuffer in einen Pinsel

ÜBERSICHT

```
[id] = gl.ReadPixelsToBrush(x, y, width, height, id)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.ReadPixels()`, mit der Ausnahme, dass die Pixeldaten in den durch `id` angegebenen Hollywood-Pinsel umgewandelt werden. Wenn es bereits einen Pinsel gibt, der den Identifikator `id` verwendet, wird er zuerst gelöscht. Wenn Sie `Nil` im Argument `id` angeben, wählt `gl.ReadPixelsToBrush()` automatisch einen freien Identifikator für diesen Pinsel aus und gibt ihn zurück.

Siehe Abschnitt 6.122 [gl.ReadPixels], Seite 189, für Details.

EINGABEN

`x`	gibt die linke Koordinate eines rechteckigen Blocks von Pixeln an
`y`	gibt die untere Koordinate eines rechteckigen Blocks von Pixeln an
`width`	Breite des Pixelrechtecks
`height`	Höhe des Pixelrechtecks
`id`	id für den neuen Pinsel oder `Nil` für die automatische ID-Auswahl

RÜCKGABEWERTE

`id`	optional: Identifikator des Pinsels; wird nur zurückgegeben, wenn Sie `Nil` in `id` übergeben (siehe oben)

9.6 gl.SetCurrentContext

BEZEICHNUNG

gl.SetCurrentContext – setzt den aktuellen OpenGL-Kontext

ÜBERSICHT

```
gl.SetCurrentContext(id)
```

BESCHREIBUNG

Dieser Befehl macht den OpenGL-Kontext dem in `id` angegebenen Display aktuell. Mit GL Galore behält jedes Hollywood-Display seinen eigenen GL-Kontext. Mit diesem Befehl können Sie den aktuellen GL-Kontext einstellen, den alle weiteren Aufrufe von OpenGL verwenden sollen.

Sie können `gl.GetCurrentContext()` verwenden, um den aktuellen GL-Kontext abzurufen.

EINGABEN

`id`	Identifikator eines Hollywood-Displays, dessen GL-Kontext aktuell gemacht werden soll

9.7 gl.TexImageFromBrush

BEZEICHNUNG

gl.TexImageFromBrush – gibt ein 1D- oder 2D-Texturbild aus dem Pinsel an

ÜBERSICHT

```
gl.TexImageFromBrush(level, id)
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.TexImage()`, holt aber die Pixeldaten von dem in `id` angegebenen Hollywood-Pinsel. Die zweidimensionale Texturierung wird automatisch verwendet, wenn der Pinsel mehr als eine Zeile hat.

Siehe Abschnitt 6.138 [gl.TexImage], Seite 209, für Details.

EINGABEN

`level`	gibt die Detailstufenzahl an; Level 0 ist die Basis-Bildebene, Level n ist das n-te Mipmap-Reduktionsbild
`id`	Identifikator eines Hollywood-Pinsels

9.8 gl.TexSubImageFromBrush

BEZEICHNUNG

gl.TexSubImageFromBrush – gibt ein 1D- oder 2D-Textur-Teilbild aus dem Pinsel an

ÜBERSICHT

```
gl.TexSubImageFromBrush(level, id, xoffset[, yoffset])
```

BESCHREIBUNG

Dieser Befehl entspricht `gl.TexSubImage()`, holt aber die Pixeldaten aus dem in `id` angegebenen Hollywood-Pinsel. Die zweidimensionale Texturierung wird automatisch verwendet, wenn der Pinsel mehr als eine Zeile hat.

Siehe Abschnitt 6.142 [gl.TexSubImage], Seite 221, für Details.

EINGABEN

`level`	gibt die Detaillierungsstufe an; Stufe 0 ist die Grundbildstufe; Level n ist das n-te Mipmap-Reduzierungsbild
`id`	Identifikator eines Hollywood-Pinsels
`xoffset`	gibt einen Texel-Versatz in der x-Richtung innerhalb des Textur-Feldes an
`yoffset`	Optional: Gibt einen Texel-Versatz in der y-Richtung innerhalb des Textur-Feldes an; nur für 2D-Texturen erforderlich

9.9 glu.BuildMipmapsFromBrush

BEZEICHNUNG

glu.BuildMipmapsFromBrush – erstellt 1D- oder 2D-Mipmaps aus einem Pinsel

ÜBERSICHT

```
error = glu.BuildMipmapsFromBrush(id)
```

BESCHREIBUNG

Dieser Befehl entspricht `glu.BuildMipmaps()`, holt aber die Pixeldaten aus dem in `id` angegebenen Hollywood-Pinsel. Zweidimensionale Mipmaps werden automatisch erstellt, wenn der Pinsel mehr als eine Zeile hat.

Siehe Abschnitt 7.4 [glu.BuildMipmaps], Seite 233, für Details.

EINGABEN

`id`	Identifikator eines Hollywood-Pinsels

RÜCKGABEWERTE

`error`	Fehlercode oder 0 für den Erfolg

Anhang A Licenses

A.1 LuaGL license

LuaGL is licensed under the terms of the MIT license reproduced below. This means that LuaGL is free software and can be used for both academic and commercial purposes at absolutely no cost.

A.2 GLFW license

A.3 SGI Free Software B license

A.4 LGPL license

can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that

uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an

application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than

a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies

directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Index

G

N

Q