Bachelorarbeit im Rahmen des Studiengangs
Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 - Medizintechnik und Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Entwicklung eines logischen GKS Gerätetreibers auf der Basis von OpenGL

Jülich, 16. Juli 2013

Jörg Winkler

Eigenhändigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

(Ort, Datum)	(Unterschrift)

Diese Arbeit wurde betreut von:

1. Prüfer:	Prof. Ulrich Stegelmann
2. Prüfer:	Josef Heinen

Sie wurde angefertigt in der Forschungszentrum Jülich GmbH
im Peter Grünberg Institut / Jülich Centre for Neutron Science.

Die Wissenschaftler am Peter Grünberg Institut / Jülich Centre for Neutron Science untersuchen in Experimenten und Simulationen Form und Dynamik von Materialien wie Polymeren, Zusammenlagerungen großer Moleküle und biologischer Zellen sowie die elektronischen Eigenschaften von Festkörpern. Für die Präsentation der in diesem Zusammenhang anfallenden Forschungsergebnisse in Vorträgen und Veröffentlichungen werden häufig 2D- und 3D-Darstellungen von Mess- und Simulationsergebnissen in Echtzeit benötigt.

Den Schwerpunkt dieser Bachelorarbeit bildet die Entwicklung eines logischen Gerätetreibers für ein im Hause entwickeltes Grafisches Kernsystem (GKS) auf der Basis der Open Graphics Library (OpenGL). Das Modul ist so aufzubauen, dass ein beliebiger OpenGL Kontext zur Darstellung von 2D-Grafikprimitiven des GKS genutzt werden kann. Die Erstellung eines OpenGL Kontextes ist losgelöst vom Treiber zu betrachten und durch ein Toolkit (z. B. GLUT oder GLFW) zu realisieren. Für die Darstellung von Texten ist die in der Seminararbeit [Win12] untersuchte FreeType-Bibliothek zu verwenden, welche das geräteunabhängige Rasterisieren von Texten ermöglicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

In wissenschaftlichen Arbeiten, Präsentationen und Berichten werden zur Visualisierung und Analyse komplexer Zusammenhänge häufig zwei- und dreidimensionale Grafiken verwendet. Einerseits können durch zweidimensionale Diagramme, Funktionsgraphen und Beschriftungen mathematische Zusammenhänge dargestellt werden. Andererseits bieten räumliche Darstellungen ein besseres Bild von der Wirklichkeit und können mehr Informationen enthalten, beispielsweise bei der Untersuchung von Molekülstrukturen.

Im PGI/JCNS basiert die Entwicklung entsprechender Visualisierungssoftware oft auf einem Grafischen Kernsystem (GKS), welches als Bindeglied zwischen den Programmen und verschiedenen Ausgabegeräten dient. Sogenannte GKS-Treiber übernehmen dabei die Kommunikation zwischen den Anwendungen und den verschiedenen Ausgabekomponenten. Die Treiber setzen die Darstellung der Grafikprimitiven gerätespezifisch um und ermöglichen somit die Ausgabe der Daten in eine Datei, auf einen Bildschirm oder mit einem Drucker.

Die Anzeige von dreidimensionalen Objekten erfolgt aktuell mit der im Hause entwickelten GR3-Bibliothek [Rhi12], die sich als Erweiterung zwischen Anwendung und GKS bzw. Ausgabegerät befindet. Zur Darstellung nutzt sie entweder die zweidimensionalen Darstellungsmöglichkeiten des GKS oder ein OpenGL[1]-basiertes Fenster. Eine interaktive Darstellung, in der sowohl zwei- als auch dreidimensionale Objekte gemeinsam enthalten sind, ist mit dem GKS bisher nicht möglich. Diese Tatsache motiviert die Entwicklung eines logischen Gerätetreibers, welcher die Grafikprimitiven des GKS auf Basis von OpenGL darstellt.

Ein weiterer Vorteil eines OpenGL-Gerätetreibers ist die Vereinfachung der Erzeugung weiterer Gerätetreiber. So können andere Implementierungen ein Fenster mit OpenGL-Kontext erstellen und den in dieser Arbeit entwickelten OpenGL-Treiber zur Darstellung der Primitiven benutzen. OpenGL ist plattformunabhängig, es fehlen jedoch Funktionen zum Darstellen von Texten. Da auch die Darstellung der Schriften unabhängig vom Gerät erfolgen soll, wird hierzu die in der Seminararbeit [Win12] untersuchte FreeType-Bibliothek verwendet.

[1] *Open Graphics Library*, wird in Kapitel 4.1 beschrieben

2. Grafisches Kernsystem

Dieses Kapitel beschäftigt sich mit dem Aufbau und den Eigenschaften eines Grafischen Kernsystems, die für die Erstellung des logischen Gerätetreibers relevant sind. Dabei wird insbesondere auf das im PGI/JCNS verwendete Grafiksystem eingegangen, damit der Treiber in die umgebende Software-Infrastruktur eingeordnet werden kann.

2.1. Aufbau

Das Grafische Kernsystem (GKS) wird im PGI/JCNS zur Grafikerzeugung in verschiedenen Visualisierungsanwendungen eingesetzt. Die Schnittstelle für die Anwendungen ist durch einen ISO-Standard vorgegeben und stellt Methoden für elementare Grafikoperationen auf zweidimensionalen Vektorgrafiken bereit. Grundlegende Grafikobjekte des GKS sind Linienzüge, Markersymbole, Text, Füllflächen und Pixmaps. Somit dient das GKS als Bindeglied zwischen Anwendungen und Ausgabegeräten wie Druckern, GUI-Toolkits, Bilddateien oder sonstigen Grafikformaten [Hei11, S. 2].

In Abbildung 2.1 ist die grundlegende Struktur dargestellt. Die Kommunikation zwischen Grafiksystem (blau) und Ausgabekomponenten wird von logischen GKS-Treibern (violett) umgesetzt. Im Rahmen dieser Arbeit wurde der in rot dargestellte OpenGL-Gerätetreiber entwickelt.

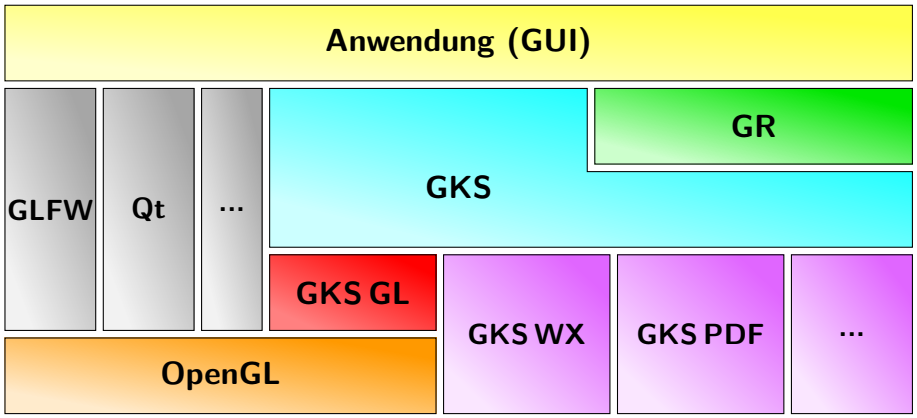

Abbildung 2.1.: Darstellung der GKS-Umgebung

Zwischen den Client-Anwendungen und dem GKS befindet sich ein weiterer (optionaler) Baustein: Die **Gr**aphics Library (GR) ist ein im PGI/JCNS entwickeltes Framework, welches die Funktionalität des GKS erweitert. Sie ermöglicht u. a. die Darstellung von Koordinatensystemen, geometrischen Formen, Füllflächen, Pfeilen, Gittern und Texten mit zugehörigen Darstellungseigenschaften.

Eine ebenfalls im PGI/JCNS entwickelte GR3-Bibliothek erweitert die Funktionalität auf dreidimensionale Objekte. Aktuell werden die dreidimensionalen Objekte mit GR3 in zweidimensionale Primitiven umgewandelt und mit GR gezeichnet, oder alternativ werden dreidimensionale Objekte in einem separaten OpenGL-basierten Fenster gezeichnet [Rhi12]. In Abbildung 2.2 ist dargestellt, wie mit dem OpenGL-Gerätetreiber zwei- und dreidimensionale Grafiken in einem Fenster realisiert werden können. Dazu wird für den GKS GL-Treiber und GR3 derselbe OpenGL-Kontext verwendet.

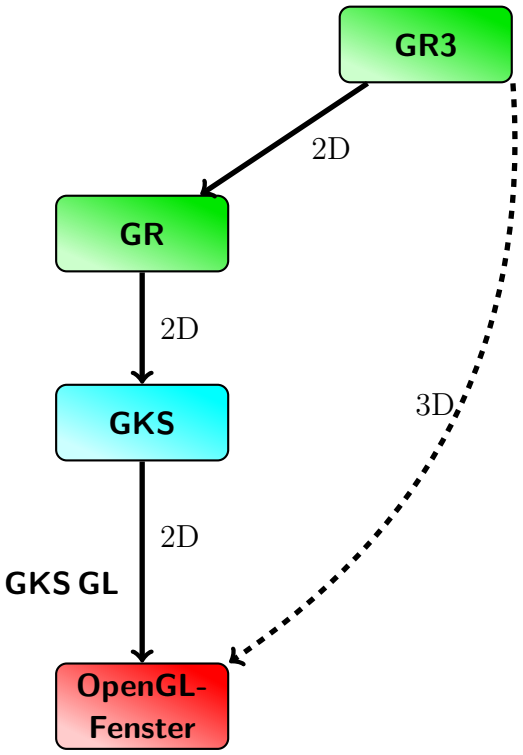

Abbildung 2.2.: Darstellung von 2D und 3D Grafik in einem Fenster

Für die Kombination der 2D- und 3D-Inhalte muss jedoch zunächst geklärt werden, wie die Elemente gemeinsam dargestellt werden können. Es bietet sich beispielsweise an, die zweidimensionalen Objekte auf eine Oberfläche im Raum zu projizieren.

2.2. Ausgabeprimitiven und ihre Attribute

Die Abbildung 2.3 zeigt die im GKS vorhandenen Primitiven und deren Attribute. Hierbei handelt es sich um geräteunabhängige Grafikoperationen, die in den Gerätetreibern gerätespezifisch umgesetzt werden.

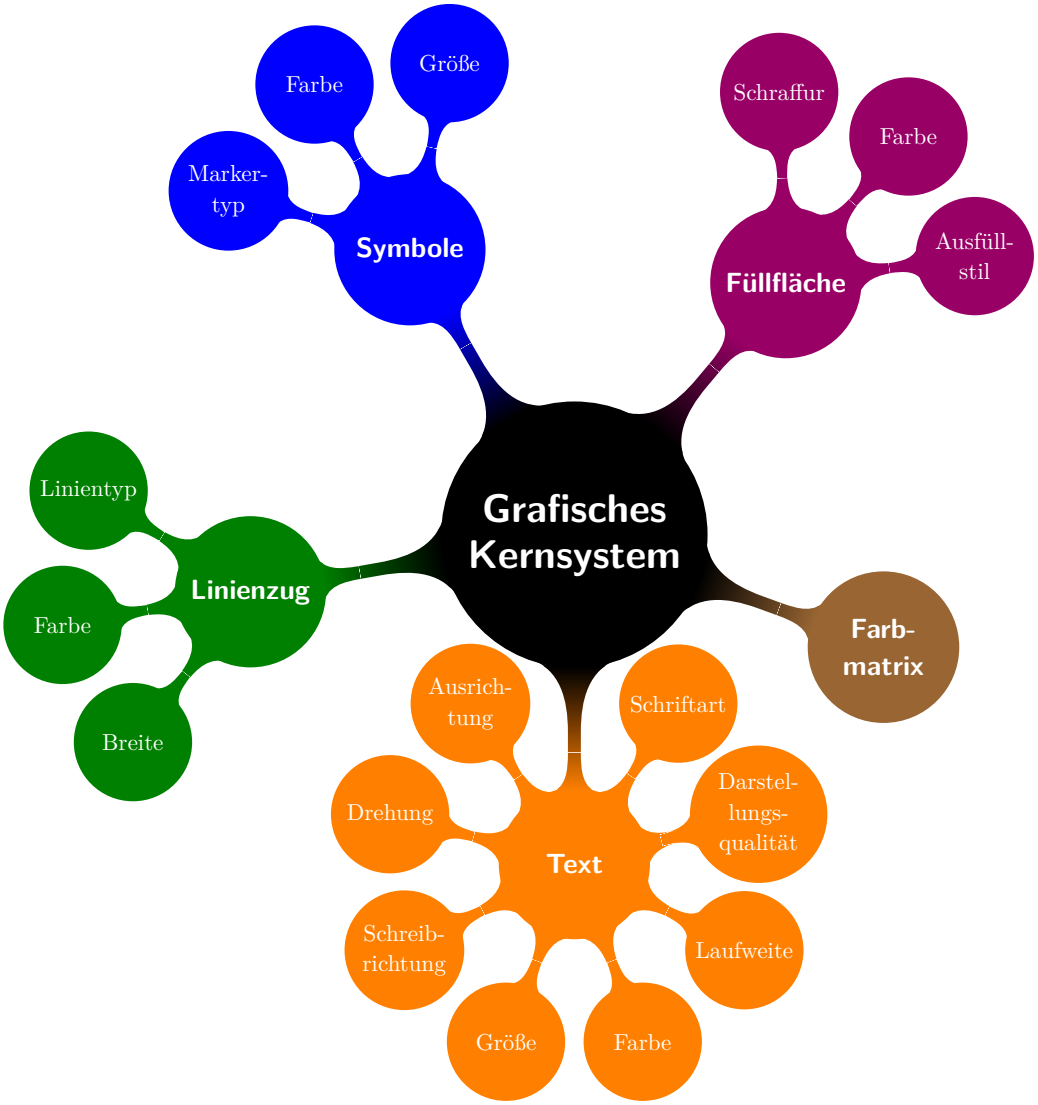

Abbildung 2.3.: Primitiven im GKS und ihre Attribute

Sämtliche Attribute werden in sogenannten **Statuslisten** gespeichert. Ein logischer Gerätetreiber muss die Eigenschaften aus der Statusliste abfragen, interpretieren und gerätespezifisch realisieren. Sowohl GKS, als auch GR verfügen über ändernde Methoden (*setter*), damit Anwendungen die Attributwerte gezielt verändern können [BB86].

Farben werden in den Statuslisten des GKS als Index in einer Farbtabelle gespeichert. Es existieren unterschiedliche Farbattribute für Text, Linien, Marker und Füllflächen. Mit der Funktion `gks_inq_rgb(index, r, g, b)` können die Werte für rot, grün und blau (*RGB-Werte*) unter Angabe des Index ausgelesen werden. Die ausgelesenen Werte sind Gleitkommazahlen $\in [0; 1]$ und stellen den Rot-, Grün- und Blau-Anteil der spezifizierten Farbe dar.

Die Ausgabeprimitiven und deren Attribute, sowie die Parameter für Transformationen werden im GKS in Form einer sogenannten **Displayliste** an die Gerätetreiber weitergeleitet. Der Aufbau einer Displayliste ist in Abbildung 2.4 dargestellt. Im Speicher stellt sie eine zusammenhängende Struktur dar, deren Kopf die Gesamtlänge der Daten enthält, gefolgt von hintereinander abgelegten Funktionsblöcken. Jeder Funktionsblock beschreibt einen Funktionsaufruf und besteht aus drei Komponenten:

1. Gesamtlänge des Funktionsblockes
2. Funktionsdeskriptor
3. Argumentliste

Der Funktionsdeskriptor ist eine Ganzzahl, die eindeutig einer auszuführenden Funktion zugeordnet werden kann. Die Argumentliste ist von der Funktion abhängig und enthält die Parameter für die Ausführung der Funktion [Hei11, S. 2f.].

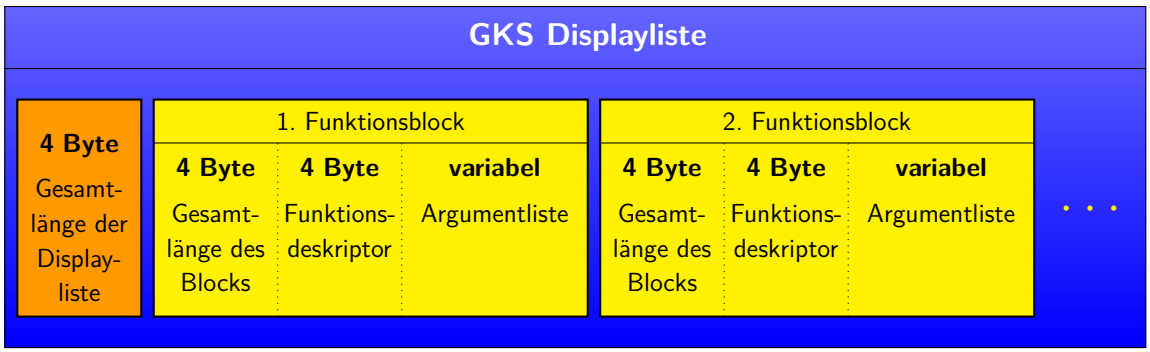

Abbildung 2.4.: Aufbau einer GKS Displayliste

Die gesamte Struktur wird an den GKS-Gerätetreiber gereicht, der die Funktionsblöcke sequenziell ausliest und verarbeitet. Zu Beginn der Verarbeitung wird jeweils die Statusliste mit den globalen Attributwerten übertragen.

2.3. Koordinatentransformationen

Das GKS arbeitet mit unterschiedlichen Koordinatensystemen, um die Darstellung an die verschiedenen Anzeigegeräte anzupassen:

WC Welt-, Anwendungskoordinaten (*world coordinates*)

NDC normierte Gerätekoordinaten (*normalised device coordinates*)

DC Fensterkoordinaten (*device coordinates*)

Die Koordinatensysteme sind in Abbildung 2.5 veranschaulicht. Für die Umwandlung der Koordinaten werden die folgenden Makros verwendet:

```
1 WC_to_NDC(x[i], y[i], gkss->cntnr, xn, yn);
2 seg_xform(&xn, &yn);
3 NDC_to_DC(xn, yn, xd, yd);
```

Im ersten Schritt werden alle Anwendungskoordinaten x_i, y_i in ein normiertes System überführt, wobei `cntnr` eine ganzzahlige Transformierungs-Nummer aus der Statusliste ist, weil es mehrere Normierungstransformationen geben kann. Anschließend wird mit dem Makro `seg_xform` eine von der Anwendung vorgegebene Transformation auf die normierten Koordinaten ausgeführt. So kann ein Segment innerhalb des Fensters gedreht, skaliert oder verschoben werden. Im letzten Schritt werden die Koordinaten in die Pixel-Koordinaten des Ausgabefensters (*viewport*) überführt, sodass die x-Koordinaten zwischen 0 und Fensterbreite und die y-Koordinaten zwischen 0 und Fensterhöhe liegen.

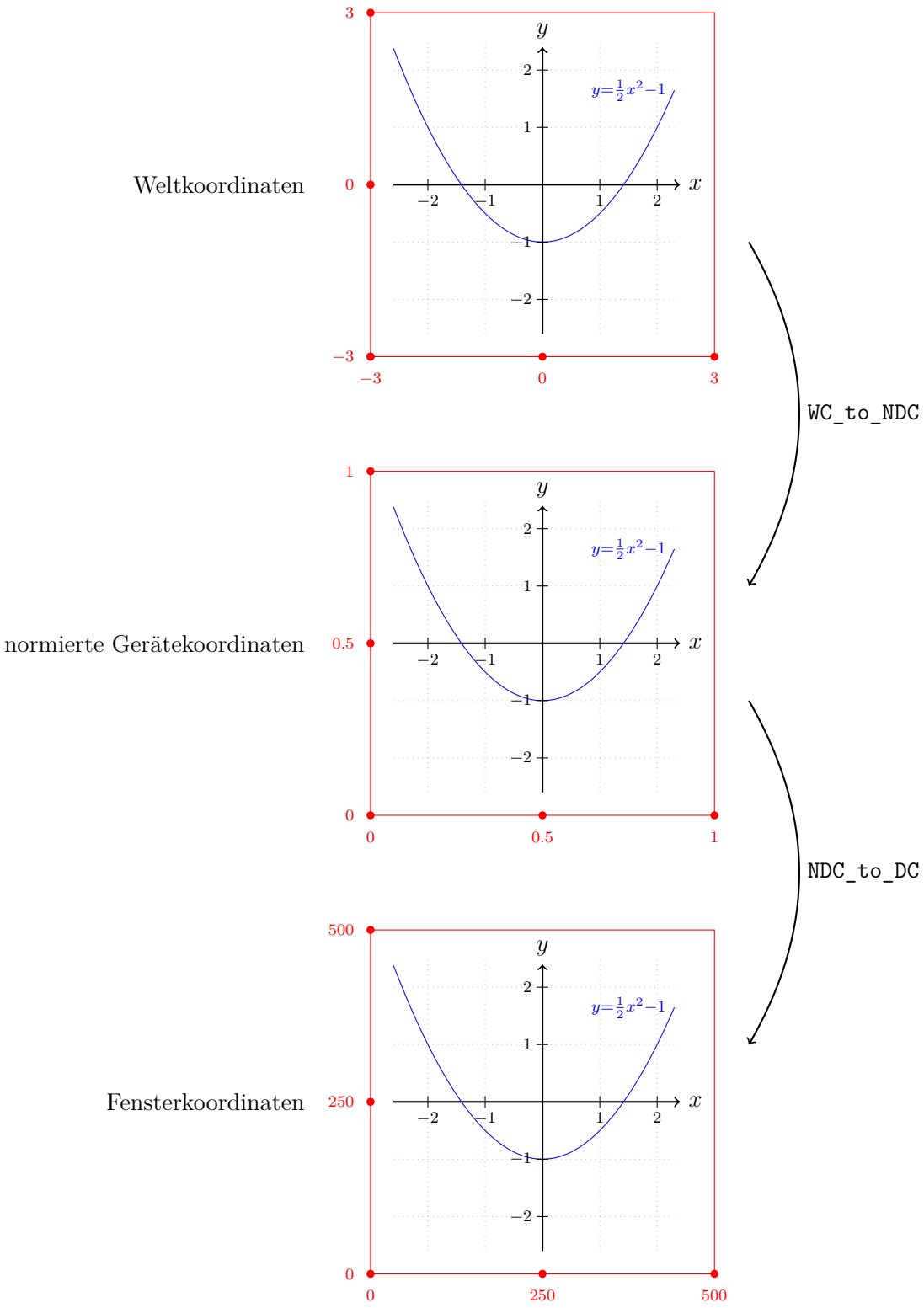

Abbildung 2.5.: Koordinatentransformationen

3. OpenGL-Kontext mit GLFW

Um mit OpenGL zeichnen zu können, muss für den Gerätetreiber zunächst ein OpenGL-Kontext erzeugt werden. Dieser enthält Informationen über ein Fenster, in dem die Inhalte dargestellt werden. In dieser Arbeit wurde GLFW verwendet, alternativ könnten aber auch GLUT oder Qt[1] benutzt werden. Die folgenden Abschnitte beschreiben die Möglichkeiten von GLFW und die Realisierung im OpenGL-Treiber.

3.1. Eigenschaften von GLFW

GLFW (*OpenGL Framework*) ist eine freie Bibliothek, die die Erzeugung und Verwaltung von Fenstern in verschiedenen Betriebssystemen abstrahiert. Die Bibliothek steht für Microsoft Windows, Linux, Apple Mac OS X und andere unixartige Betriebssysteme zur Verfügung und bietet eine einheitliche Schnittstelle für die Erzeugung von OpenGL-Umgebungen [glf]. Weiterhin besteht die Möglichkeit, Eingaben von Tastatur und Maus abzufragen, was in Kapitel 3.3 genauer erläutert wird.

Im Einzelnen beinhaltet der Funktionsumfang von GLFW die folgenden Aspekte:

- Open Source, freie Lizenz
- einfach zu benutzende Schnittstelle
- Erzeugung von einem Fenster mit OpenGL-Kontext in nur zwei Funktionsaufrufen
- unterstützt mehrere Bildschirme, hohe Auflösungen und Gammakorrekturen
- Kopieren und Einfügen aus der Zwischenablage
- Eingaben können abgefragt oder mit sog. Callbacks übergeben werden
- Unicode-Kodierung UTF-8
- statisches und dynamisches Linken
- umfangreiche Dokumentation

[1] weitere Informationen sind auf den entsprechenden Homepages zu finden:
`http://www.glfw.org/`
`http://www.opengl.org/resources/libraries/glut/`
`http://qt-project.org/`

3.1.1. Veränderungen in GLFW Version 3

Am 12. Juni 2013 wurde GLFW in der Version 3 veröffentlicht. Allerdings ist die dritte Version nicht abwärtskompatibel zu Version 2. Es wurden beispielsweise die Funktionen für das *Threading* entfernt, sodass für die Steuerung von parallel laufenden Prozessen nun andere Bibliotheken verwendet werden müssen. Gleichzeitig wurde die Thread-Sicherheit von GLFW verbessert. Die wichtigsten Neuerungen gegenüber Version 2 sind [glm]:

- Unterstützung von mehreren Fenstern und Bildschirmen
- Kopieren und Einfügen von Text über die Zwischenablage (*clipboard*)
- verbesserte Fehlerberichte
- UTF-8 Kodierung für alle Zeichenketten
- mehr Funktionen für die Ereignisverarbeitung

Für den OpenGL-Kontext im GKS wurde bezüglich der zu verwendenden Version von GLFW noch keine Entscheidung getroffen. Die weitere Entwicklung der Versionen 2 und 3 soll zunächst weiter verfolgt werden.

3.1.2. Unterschiede zwischen GLFW und GLUT

GLUT (*OpenGL Utility Toolkit*) implementiert ebenfalls eine einfache Schnittstelle, um kleinere OpenGL-Programme in einem Fenster darzustellen. GLUT ist jedoch eine proprietäre Software, deshalb ist in dieser Arbeit stets die freie Alternative *FreeGLUT* gemeint.

Der grundlegende Unterschied zwischen GLFW und FreeGLUT besteht in der Art, wie die *Window Loop* gesteuert wird. Dabei handelt es sich um eine Schleife, die so lange läuft, wie ein Anwendungsfenster geöffnet ist. In dieser Schleife wird ständig kontrolliert, ob Ereignisse zu verarbeiten sind. Im Unterschied zu GLFW implementiert und verwaltet FreeGLUT die *Window Loop* selbst, sodass Entwickler weniger Einfluss darauf nehmen können, wie Ereignisse verarbeitet werden. Bei Verwendung von GLFW liegt die Steuerung der *Window Loop* bei den Anwendungen [gls].

3.2. Erzeugung des Kontextes im OpenGL-Treiber

Für die Erzeugung eines OpenGL-Kontextes muss zuerst die GLFW-Bibliothek initialisiert werden. Dadurch wird die Verwendung der Bibliothek vorbereitet, ein Fenster existiert jedoch noch nicht (Zeile 1 im Listing rechts). Anschließend wird ein Fenster als OpenGL-Kontext erzeugt, wobei die im Listing angegebenen Parameter die

folgenden Fensterattribute einstellen (Zeile 2):

1-2 die Fenstergröße wird aus einer Statusliste des GKS gelesen und gesetzt

3-5 die Farben rot, grün und blau werden jeweils durch 8 Bit repräsentiert

6-8 ein alpha-, depth- und stencil-Buffer werden nicht erzeugt

9 die Ausgabe erfolgt in einem Fenster (im Gegensatz zum Vollbildmodus)

```
1 glfwInit();
2 glfwOpenWindow(p->width, p->height, 8, 8, 8, 0, 0, 0, GLFW_WINDOW);
```

Alle nun folgenden OpenGL-Befehle werden in diesem Fenster umgesetzt. Schließlich wird das Fenster wieder geschlossen und die GLFW-Bibliothek beendet:

```
1 glfwCloseWindow();
2 glfwTerminate();
```

3.3. Ereignissteuerung

Für die Behandlung von Ereignissen wie Tastatureingaben oder Mausklicks muss die GLFW-Prozedur `glfwPollEvents` aufgerufen werden, wenn GLFW 3 ausgeführt wird. In GLFW 2 wird die Prozedur beim *Wechsel des Framebuffers*[2] implizit aufgerufen und die Ereignisse werden verarbeitet. Der implizite Aufruf lässt sich jedoch mit dem Befehl `glfwDisable(GLFW_AUTO_POLL_EVENTS)` verhindern [gld].

Bei der Verwendung von GLFW im GKS trifft eine prozedurale Ereignisverarbeitung auf eine ereignisgesteuerte. Für jede Benutzereingabe oder Programmaktion (z. B. Wechsel des Framebuffers) wird vom Betriebssystem ein Ereignis erzeugt und entsprechend an GLFW oder das aufrufende Programm (z. B. Python) geleitet. Jedoch kann nur eines der beiden die Ereignisse abrufen und verarbeiten. Deshalb bekommt der *Window Manager* des Betriebssystems von einem Fenster keine Rückmeldungen mehr: Liegt der Fokus im Python-Terminal, so ruft GLFW seine Ereignisse eine Zeit lang nicht ab, gibt keine entsprechende Rückmeldung an das OpenGL-System und es entsteht ein Stau, sodass das Betriebssystem das Fenster als *beschäftigt* oder *nicht reagierend* behandelt. Dieses Problem kann beispielsweise durch die Verwendung von zwei parallelen Prozessen gelöst werden, in denen jeweils ständig Ereignisse abgefragt werden. Es wird in dieser Arbeit nicht weiter behandelt.

[2] Der Wechsel des Framebuffers wird mit einem Aufruf der Methode `glfwSwapBuffers` hervorgerufen. Ein Framebuffer ist ein Speicherbereich, in dem das Monitorbild gespeichert ist. Oft gibt es einen Front- und einen Backbuffer, sodass auf dem zweiten bereits das nächste Monitorbild erzeugt wird, während der erste das aktuelle Bild anzeigt.

4. OpenGL-Treiber

Nachdem in den vorherigen Kapiteln die Umgebung des GKS GL-Treibers behandelt wurde, beschäftigt sich dieses Kapitel mit seiner Realisierung. Zunächst wird eine kurze Einführung in OpenGL gegeben, in der das Framework und seine Eigenschaften vorgestellt werden. In den darauf folgenden Abschnitten wird die Implementierung des OpenGL-basierten logischen GKS-Treibers beschrieben und erörtert. Der Quellcode der beschriebenen Funktionen befindet sich im Anhang A.

4.1. OpenGL

Bei der *Open Graphics Library* handelt es sich um die Spezifikation einer Schnittstelle zur Erzeugung von zwei- und dreidimensionaler Computergrafik. Sie stellt eine allgemeine und plattformunabhängige Abstraktion der Grafikhardware dar und ermöglicht eine effiziente Erzeugung von 3D-Grafiken in Echtzeit, indem bestimmte Berechnungen direkt auf dem Grafikprozessor (GPU) erfolgen[1]. OpenGL- Implementierungen sind ein fester Bestandteil der Betriebssysteme Windows, Mac OS X und vieler Linux-Distributionen, sodass dafür meist keine zusätzliche Installation erforderlich ist. Außerdem gibt es Implementierungen, die in Verbindung mit speziellen Grafikkartentreibern bereitgestellt werden [Rhi11]. Die *Mesa 3D*-Bibliothek stellt eine weitere Alternative für Linux-Systeme dar; sie ist eine freie Grafikbibliothek, die ebenfalls die OpenGL-Spezifikation umsetzt und entweder rein softwarebasiert oder hardwarebeschleunigt[2] eingesetzt werden kann [m3d].

Bei der Erzeugung der Grafiken werden die Zeichenobjekte in vektorieller Beschreibung an die OpenGL-Implementierung übergeben und in mehreren Schritten in eine Rastergrafik umgewandelt. Die einzelnen Schritte werden durch eine sogenannte Rendering-Pipeline beschrieben, die wie folgt zusammengesetzt ist [Shr08]:

1. Das zu zeichnende Objekt wird durch seine Eckpunkte (*Vertices*) und deren Eigenschaften beschrieben.

[1] Berechnungen auf der GPU entlasten den Hauptprozessor des Rechners. Aus diesem Grund werden sie als *hardwarebeschleunigt* bezeichnet.

[2] Gallium3D ist eine Schnittstelle für die Realisierung von hardwarebeschleunigten Grafikausgaben und ist Teil von Mesa 3D

2. Die Koordinaten werden in normalisierte Gerätekoordinaten übertragen (siehe Abbildung 4.2 auf der nächsten Seite).
3. Die Vertices werden zu grafischen Primitiven zusammengesetzt.
4. Die Primitiven werden in eine Rastergrafik umgewandelt.

Im entwickelten GKS GL-Treiber wird dazu aus Kompatibilitätsgründen das Verfahren der *Fixed Function Pipeline* verwendet, bei dem die Schritte von der Eingabe bis zu der fertigen Grafik fest im Grafikkartentreiber implementiert sind. Die Darstellung (z. B. Transformationen, Beleuchtung) wird durch Parameter gesteuert. Dieses Verfahren ist in Abbildung 4.1 visualisiert.

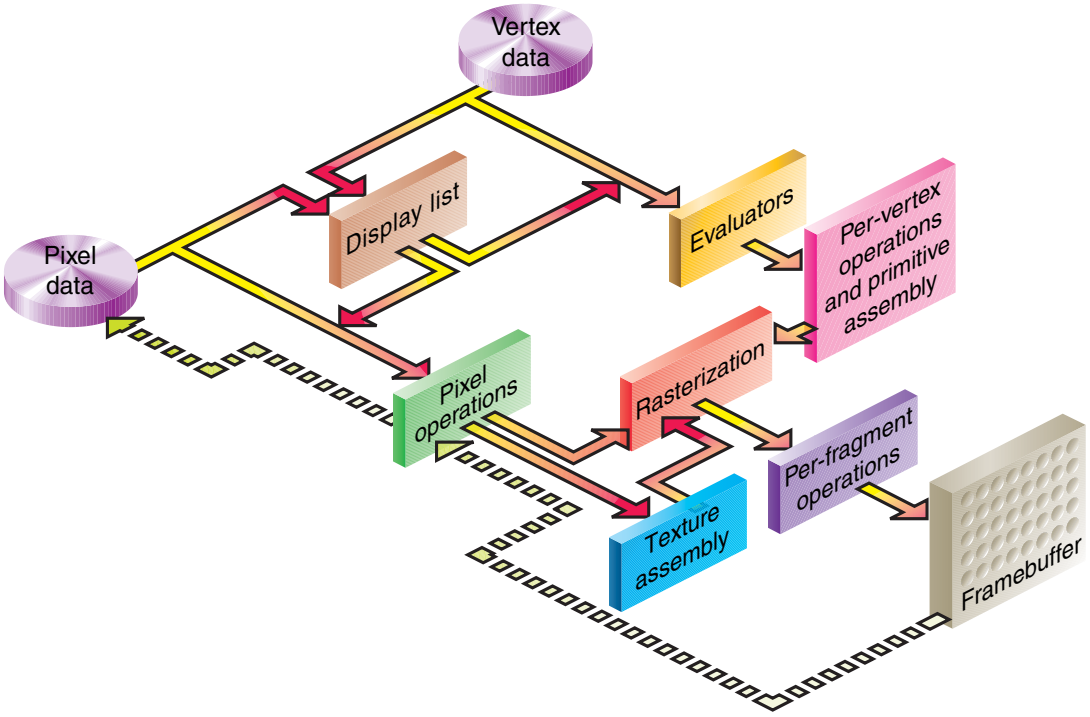

Abbildung 4.1.: OpenGL Rendering Pipeline [Shr08, S. 11]

Eine alternatives, moderneres Verfahren stellt die *Shaderbasierte Pipeline* dar, bei der bestimmte Funktionen vom Anwendungsentwickler in einer speziellen Programmiersprache[3] selbst implementiert werden. Für die Anforderungen des GKS ist die Fixed Function Pipeline jedoch ausreichend, weil die benötigte Funktionalität mit entsprechenden OpenGL-Funktionen erreicht werden kann. Außerdem wird das neue Verfahren erst seit OpenGL Version 2 unterstützt, weshalb es im OpenGL-Treiber nicht angewendet wird.

[3]bei dieser Programmiersprache handelt es sich um die OpenGL Shading Language (GLSL)

Der sichtbare Bildausschnitt (*viewport*) wird in OpenGL standardmäßig als ein System gesetzt, dessen Nullpunkt in der Fenstermitte liegt und dessen Ränder die Koordinaten -1 bzw. 1 haben (vgl. Abbildung 4.2). Die resultierenden Koordinaten werden *normalisierte Koordinaten* genannt. Für die Umrechnung wird die sogenannte *ModelView*-Matrix verwendet, die an dieser Stelle für den GKS GL-Treiber angegeben wird:

$$\begin{pmatrix} \frac{2}{Breite} & 0 & 0 & -1 \\ 0 & \frac{-2}{Höhe} & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Elemente auf der Diagonalen skalieren die Breite und die Höhe, die ersten beiden Werte der 4. Spalte verschieben die Koordinaten in x- bzw. y-Richtung.

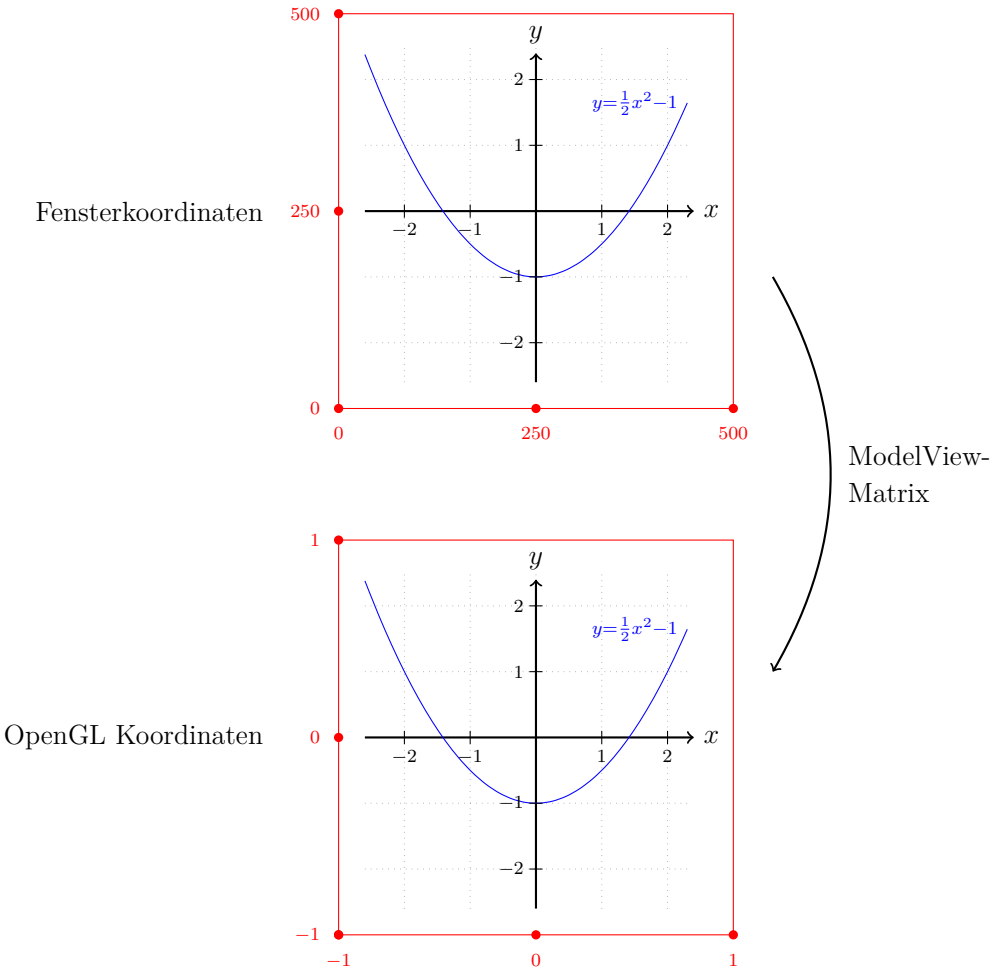

Abbildung 4.2.: Koordinatentransformation mit der ModelView-Matrix

4.2. Zerlegung eines Polygons in Dreiecke

Bei der Darstellung von Oberflächen werden diese zunächst in Dreiecksgitter zerlegt (*trianguliert*), weil die Grafikhardware für die Darstellung von Dreiecken optimiert ist. Auch bei der OpenGL-Primitive `GL_POLYGON` wird intern eine Zerlegung in Dreiecke vorgenommen. Die Primitive unterstützt allerdings nur konvexe Polygone, während das GKS auch konkave Polygone erzeugen kann. Die Triangulierung von konkaven Polygonen muss demnach im Treiber erfolgen: Für die Erzeugung eines Dreiecksgitters aus beliebigen Polygonen wird in diesem Abschnitt ein Algorithmus vorgestellt, mit dem die Zerlegung einfach und schnell zu implementieren ist.

Zunächst werden zu jedem Eckpunkt des Polygons zwei horizontale Strahlen gebildet, einer verläuft nach links und der andere nach rechts. Die Strahlen enden, sobald sie auf den Rand des Polygons treffen – liegt ein Strahl vollständig außerhalb des Polygons, gibt es keinen Schnittpunkt und der Strahl wird verworfen. In Abbildung 4.3a ist ein zu unterteilendes Polygon abgebildet; die Strahlen sind als gestrichelte Linien dargestellt.

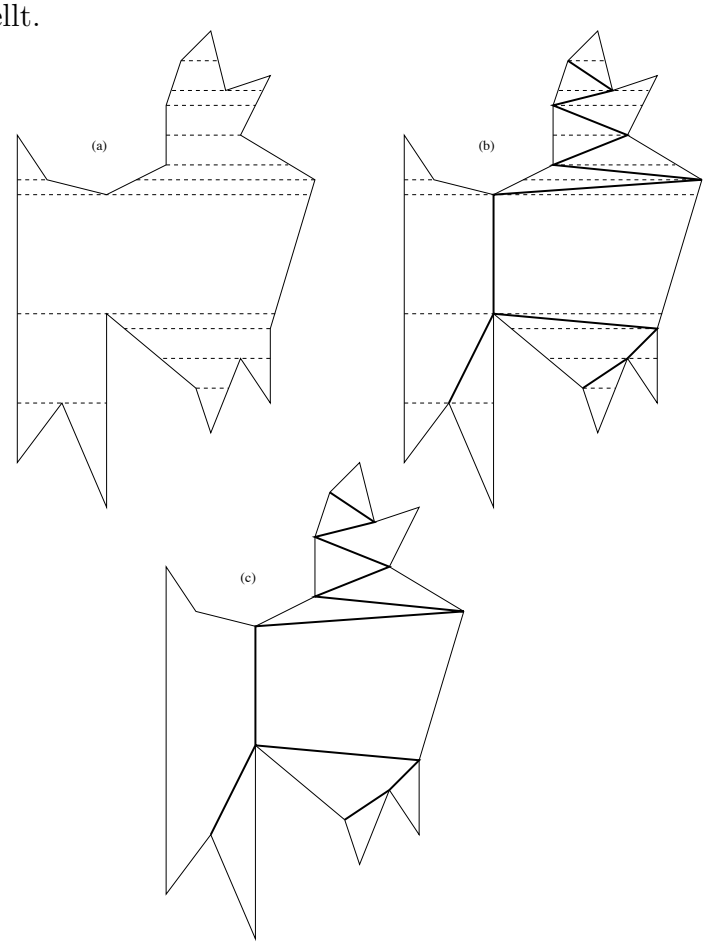

Abbildung 4.3.: Zerlegung eines Polygons [Sei91]

Mit den Strahlen sind im Inneren des Polygons zahlreiche Trapeze und Dreiecke entstanden. Im nächsten Schritt wird für jedes Trapez geprüft, ob es zwei Eckpunkte des Polygons enthält. Wenn diese auf verschiedenen Seiten des Trapezes liegen, werden die Eckpunkte verbunden, sodass aus dem Trapez zwei Dreiecke oder zwei kleinere Trapeze entstehen. Die neuen Verbindungen sind in Abbildung 4.3b in Form dicker Linien eingezeichnet.

Für den letzten Schritt werden nur noch die Verbindungen aus dem vorherigen Schritt und das ursprüngliche Polygon betrachtet. Abbildung 4.3c zeigt die entstandenen inneren Polygone – sie können durch Abschneiden von konvexen Ecken einfach in Dreiecke zerlegt werden [Sei91].

4.3. Umsetzung der Ausgabeprimitiven

4.3.1. Linienzüge

Ein Linienzug (*Polyline*) ist eine Verknüpfung von mehreren Strecken und wird durch mindestens zwei Stützpunkte definiert. Eine **einfache Linie** ist ein Sonderfall des Linienzuges mit genau zwei Stützpunkten und muss nicht gesondert behandelt werden. Im GKS haben Linienzüge die folgenden Attribute:

- Linienfarbe
- Linientyp
- Linienbreite

Die **Linienfarbe** wird mit der Methode `glColor3fv` gesetzt, wobei als Parameter die RGB-Werte als Vektor mit drei Gleitkommazahlen gemäß Kapitel 2.2 übergeben werden.

Für die **Linienbreite** wird die Methode `glLineWidth` verwendet. Da die Linienbreite in der Statusliste bereits als Fließkommazahl vorliegt und eine analoge Bedeutung hat, muss sie nicht weiter umgewandelt werden.

Eine **gestrichelte Linie** wird durch ein Linienmuster spezifiziert, welches für die Länge der Linie laufend wiederholt wird. Das Linienmuster gibt an, welche Pixel gezeichnet werden und welche nicht. Diese Information liegt in Form eines Bitmusters vor.

In OpenGL steht für die Spezifikation des Linienmusters zwar die Methode `glLineStipple` zur Verfügung, allerdings verarbeitet diese ein 16 Bit-Muster und einen Streckungsfaktor, um den jedes Element des Musters vervielfacht wird. Somit ist diese Methode sehr unflexibel in Bezug auf die Definition von beliebigen Mustern. Deshalb wurden die meisten Strichmuster des GKS auf 16 Bit angepasst, wie in Abbildung 4.4 gezeigt ist.

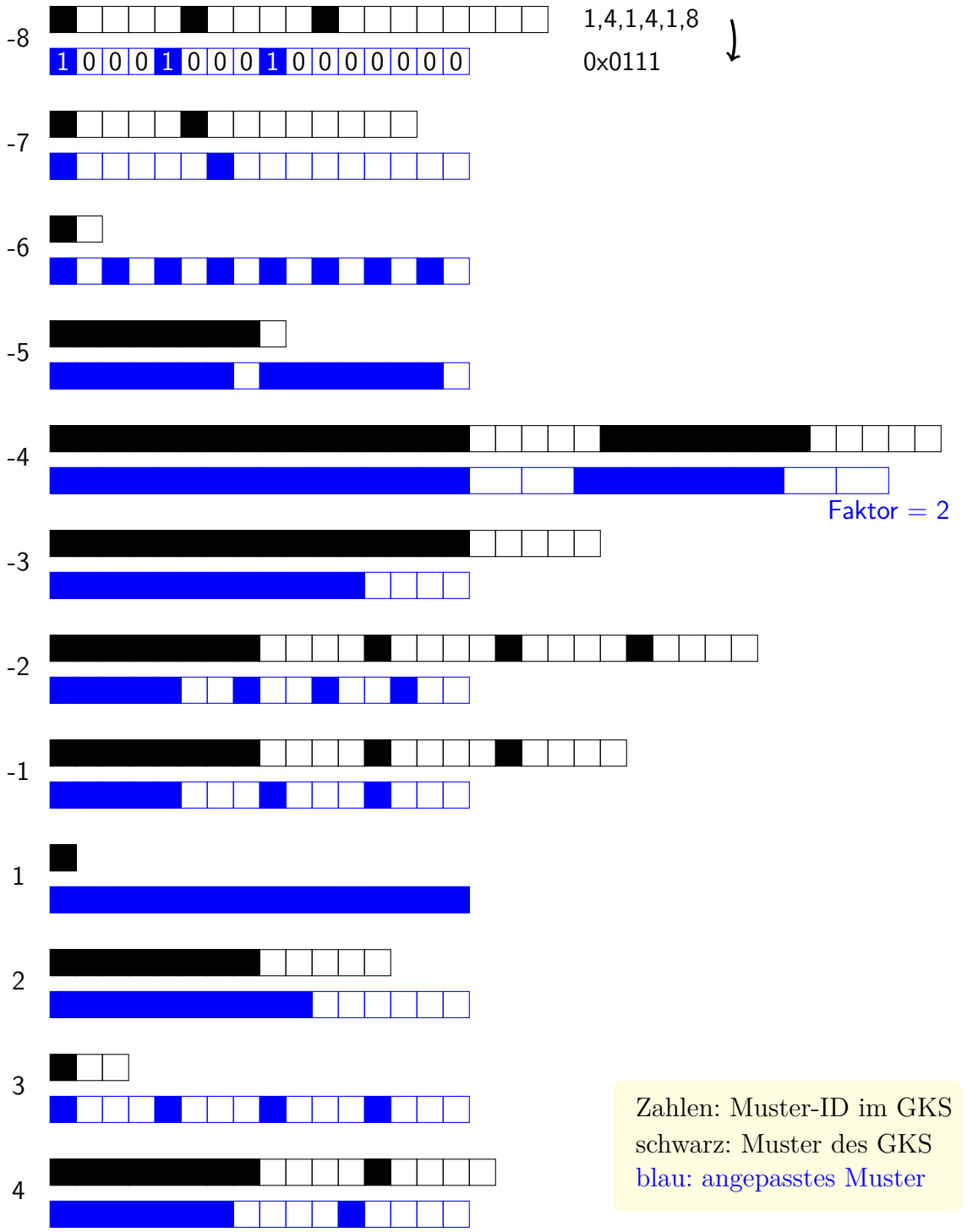

Abbildung 4.4.: Muster zur Erzeugung von Linien

Die **Stützpunkte** bestehen jeweils aus einer x- und einer y-Koordinate. Die x- und y-Koordinaten liegen separat in zwei Feldern vor, d. h. für einen Stützpunkt werden dieselben Indices in beiden Feldern verwendet. In einer Schleife über alle Stützpunkte werden diese zunächst mit GKS-Funktionen transformiert, sodass aus den Weltkoordinaten Fensterkoordinaten entstehen (vgl. Kapitel 2.3). Letztere werden anschließend als Eckpunkte (*vertices*) an OpenGL-Prozeduren übergeben, damit sie gezeichnet werden. Zum Zeichnen wird die Primitive auf `GL_LINE_STRIP` gesetzt.

4.3.2. Füllflächen

Für Füllflächen gibt es drei verschiedene Stile:

- nur Rahmen, Inhalt transparent
- mit einer Farbe gefüllt
- mit einer Schraffur in einer Farbe gefüllt

Jede Füllfläche wird durch ein Polygon begrenzt, dessen Eckpunkte analog zum Linienzug in zwei Feldern vorliegen. Die Koordinaten des Polygons werden mit GKS-Funktionen (vgl. Kapitel 2.3) in Fensterkoordinaten transformiert und anschließend in einen *Vertex-Buffer* geschrieben. Bei einem Buffer (dt. *Puffer*) handelt es sich um einen Speicherbereich, der später in einem Vorgang in den Grafikspeicher kopiert wird. Dadurch wird die Laufzeit im Vergleich zu einzelnen Funktionsaufrufen für jeden Eckpunkt verbessert. Ein Puffer muss in OpenGL zunächst aktiviert werden; anschließend wird er mit den Werten eines Feldes gefüllt, zum Grafikchip übertragen und sein Inhalt kann schließlich mit der Methode `glDrawArrays` gezeichnet werden. Im Folgenden wird die individuelle Vorgehensweise bei der Darstellung der einzelnen Füllstile erläutert.

Bei einer **einfarbigen Füllung** wird die Füllfarbe als Zeichenfarbe gesetzt. Als Zeichenprimitive wird `GL_POLYGON` verwendet, sodass der Flächeninhalt mit der zuvor gesetzten Zeichenfarbe gefärbt wird.

Wenn **keine Füllung** gezeichnet werden soll, wird nur der Rand des Polygons gezeichnet. Hierzu wird die Zeichenprimitive auf `GL_LINE_LOOP` gesetzt.

Eine **Schraffur** im GKS ist eine 8×8-Matrix mit Informationen, welche Pixel ein- oder ausgeschaltet sind. Dabei gelten wie beim Linientyp periodische Randbedingungen, d. h. ein Muster wird in jede Richtung durch Wiederholung fortgesetzt.

Wenn das Polygon mit einem Muster gefüllt werden soll, muss dessen Index zunächst aus der Statusliste des GKS abgefragt werden (`fl_style`), anschließend wird das Muster in ein Feld übertragen und ausgewertet. Die Modulo-Operationen in Zeile 6 des folgenden Listings bewirken eine Verschiebung des Musters.

```
int parray[33], i, j;
GLubyte bitmap[8][8];
gks_inq_pattern_array(fl_style, parray);
for (i = 0; i < 8; i++) {
  for (j = 0; j < 8; j++) {
    bitmap[(j+7)%8][(i+7)%8] = (parray[j%parray[0]+1]>>i) & 0x01 ?
       0 : 255;
  }
}
```

Die entstandene Matrix wird auf eine Textur übertragen, mit der das zu zeichnende Polygon ausgefüllt wird. Mit der folgenden Textur-Matrix wird die Textur der Größe 8×8 normiert, indem x- und y-Komponenten auf ein Achtel skaliert werden:

$$\begin{pmatrix} \frac{1}{8} & 0 & 0 & 0 \\ 0 & \frac{1}{8} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Bei der Füllung mit einer Farbe oder einer Schraffur wird ein Polygon als Begrenzung für die Fläche angegeben. Hierbei können auch konkave Polygone vorkommen, jedoch werden von der OpenGL-Primitive `GL_POLYGON` nur konvexe Polygone unterstützt. Deshalb muss das Polygon zunächst trianguliert werden; ein Algorithmus dafür wurde in Abschnitt 4.2 vorgestellt.

4.3.3. Marker-Symbole

Die Symbole sind hauptsächlich aus den bisher behandelten Primitiven Linienzug und Füllflächen zusammengesetzt, zusätzlich werden Kreise benötigt. In Abbildung 4.5 werden die GKS-Symbole dargestellt.

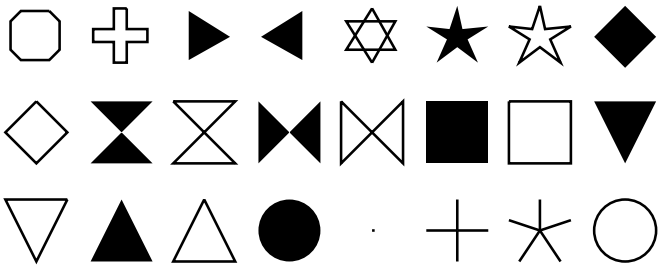

Abbildung 4.5.: GKS Marker-Symbole

Die Zeichenprimitiven mit Koordinaten für ein Symbol liegen in den übrigen GKS-Treibern in Form von Listen vor. So ist der Stern beispielsweise ein Polygon mit 11 Eckpunkten. Dieses Polygon ist jedoch nicht konvex, sodass zunächst ein Dreiecksgitter erzeugt werden muss.

Die Entwicklung dieses Dreiecksmusters ist hierbei am besten zu realisieren, indem der Mittelpunkt des Sterns gewählt wird und jeweils zwischen zwei benachbarten Außenpunkten und dem Mittelpunkt ein Dreieck gezeichnet wird. In Abbildung 4.6 wird dieses Dreiecksgitter dargestellt. Bei Verwendung der Primitive `GL_TRIANGLE_FAN` wird in der Koordinatenliste lediglich der Mittelpunkt ergänzt, somit sind nur minimale Änderungen an der Liste erforderlich.

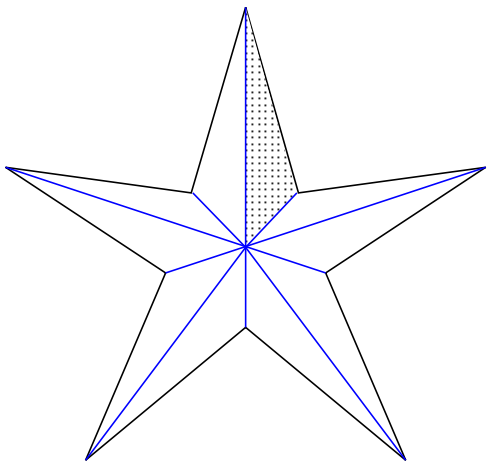

Abbildung 4.6.: Dreiecksgitter für den Stern

Für die Darstellung der Kreise muss eine Approximation durch einen Linienzug bzw. durch ein Dreiecksgitter erfolgen. Für einen Kreisbogen sind folgende Daten gegeben:

- Startwinkel α
- Endwinkel β
- Kreismittelpunkt (x, y)
- Kreisradius r

Das folgende Listing zeigt die Vorgehensweise am Beispiel eines ausgefüllten Kreisbogens:

```
1 float angle, c, s, xr, yr, tmp;
2 int num_segments = 4 * r;
3 angle = (marker[mtype][pc+2] - marker[mtype][pc+1]) * M_PI/180;
4 c = cosf(angle / (num_segments - 1));
5 s = sinf(angle / (num_segments - 1));
```

```
6  xr = r * cosf(marker[mtype][pc + 1]);
7  yr = r * sinf(marker[mtype][pc + 1]);
8
9  glBegin(GL_TRIANGLE_FAN);
10 for (i = 0; i < num_segments; i++) {
11   glVertex2f(x + xr, y + yr);
12   tmp = xr;
13   xr = c * xr  - s * yr;
14   yr = s * tmp + c * yr;
15 }
16 glEnd();
```

Die Anzahl der zu verwendenden Stützpunkte n für die Kreisapproximation muss mit wachsendem Radius linear größer werden. Sie wurde durch Tests auf den vierfachen Radius festgelegt, weil dies ein guter Kompromiss zwischen Darstellungsqualität und Rechenaufwand ist. Der zu zeichnende Rotationswinkel berechnet sich als Differenz aus Start- und Endwinkel $(\beta - \alpha)$, im Falle des Kreises beträgt er 360°. Für die Verwendung in Sinus- und Kosinusoperationen wird der Rotationswinkel mit dem Faktor $\frac{\pi}{180}$ in das Bogenmaß umgerechnet (Zeile 3).

Die Variablen x und y stellen den (konstanten) Kreismittelpunkt dar, xr und yr repräsentieren jeweils die Koordinaten des aktuellen Stützpunktes. Zwischen je zwei Stützpunkten wird ein Kreissegment aufgespannt, wie es in Abbildung 4.7 an einem Beispiel veranschaulicht ist. Der Winkel eines Kreissegments beträgt somit $\delta = \frac{\beta - \alpha}{n-1}$.

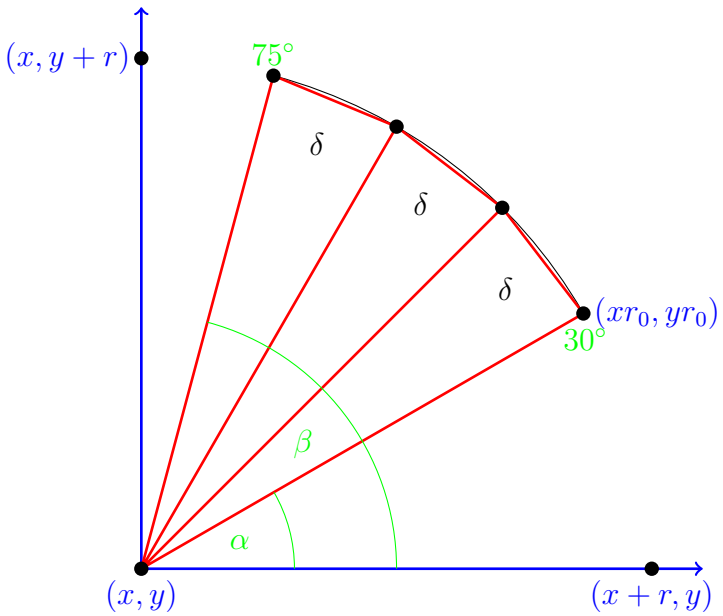

Abbildung 4.7.: Kreisapproximation durch Dreiecksgitter

Damit bei der Berechnung der Stützpunkte nicht laufend trigonometrische Funktionen ausgewertet werden müssen, wurde ein iterativer Algorithmus mit einer Rotationsmatrix verwendet. Der erste Stützpunkt ergibt sich aus dem Startwinkel (vgl. Zeilen 6-7):

$$\begin{pmatrix} xr_0 \\ yr_0 \end{pmatrix} = r \cdot \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix}$$

Die benötigten Werte für die Rotationsmatrix können vor der Iteration berechnet werden, sodass innerhalb der Iteration nur noch Multiplikationen und Additionen bzw. Subtraktionen im Rahmen eines Matrix-Vektor-Produktes benötigt werden (Zeilen 4-5 und 12-14):

$$\begin{pmatrix} xr_{k+1} \\ yr_{k+1} \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \cdot \begin{pmatrix} xr_k \\ yr_k \end{pmatrix} \text{ mit } c = \cos\delta \text{ und } s = \sin\delta$$

Der Nachteil dieses Einschrittverfahrens besteht darin, dass durch Gleitkommaoperationen verursachte Rundungsfehler in die nachfolgenden Iterationsschritte übernommen werden. Somit kann sich der Fehler weiter verstärken. Diese Fehler sind allerdings so klein, dass sie mit der verwendeten Anzahl Stützpunkte auf dem Bildschirm nicht sichtbar sind, weil sie bei der Diskretisierung auf Bildschirmpixel wieder ausgelöscht werden.

4.3.4. Rastergrafiken (Cellarray)

Eine Rastergrafik wird als Textur auf der Zeichenfläche dargestellt. Mit den Parametern $x_{\min}$, $x_{\max}$, $y_{\min}$ und $y_{\max}$ wird der Bereich festgelegt, in den die Grafik eingepasst werden soll. Dabei wird sie ggfs. gestreckt oder gestaucht, wenn das Seitenverhältnis der Grafik nicht mit dem des Bereiches übereinstimmt. Für den Fall, dass eine *min*-Koordinate größer als eine *max*-Koordinate ist, wird die Grafik entsprechend gespiegelt.

Zunächst werden die vier Koordinaten des Bereiches in Gerätekoordinaten umgewandelt, d. h. sie nehmen nun ganzzahlige Pixelwerte an. Anschließend werden Position und Größe des Bereiches berechnet:

$$\begin{aligned} x &= \min(x_{\min}, x_{\max}) \\ y &= \min(y_{\min}, y_{\max}) \\ \text{width} &= |x_{\max} - x_{\min}| \\ \text{height} &= |y_{\max} - y_{\min}| \end{aligned}$$

Die Farbwerte der Rastergrafik werden mit dem Zeiger *colia* referenziert und die Parameter *dx* und *dy* geben die Größe der Farbmatrix in x- und y-Richtung an. Im Folgenden wird die Farbmatrix in zwei verschachtelten Schleifen durchlaufen und die gelesenen Werte werden in das RGBA-Format[4] gebracht, das von OpenGL verarbeitet werden kann. Wenn die Grafik in einer oder zwei Richtungen gespiegelt werden muss, dann wird die Farbmatrix rückwärts durchlaufen (vgl. Zeilen 3-6 im folgenden Listing).

Der Parameter *true_color* enthält einen Wahrheitswert, der festlegt, ob jeder Wert der Farbmatrix direkt als RGBA-Kombination oder als Index in einer Farbtabelle interpretiert werden kann. Im ersten Fall werden die 32 Bit eines Integers in jeweils 8 Bit für rot, grün, blau und alpha aufgeteilt, wobei Werte $\in [0; 255]$ resultieren. Diese werden mittels Division durch 255 in eine Fließkommazahl umgerechnet und in die Bitmap eingetragen (vgl. Zeilen 8-12).

Falls es sich bei den Werten der Farbmatrix nicht um RGBA-Werte handelt, sind sie als Farbindex des GKS zu interpretieren (vgl. Kapitel 2.2, dritter Absatz). In diesem Fall werden die benötigten Werte aus dem entsprechenden Feld ausgelesen und in die Bitmap eingetragen (vgl. Zeilen 14-18).

```
1  int i, j, k, ix, iy, rgb, index;
2  GLfloat bitmap[dx][dy][4];
3  for (i = 0; i < dx; i++) {
4    ix = (x1 > x2) ? dx - i - 1 : i;
5    for (j = 0; j < dy; j++) {
6      iy = (y1 < y2) ? dy - j - 1 : j;
7      if (true_color) {
8        rgb = colia[ix * dy + iy];
9        bitmap[i][j][0] = (rgb & 0xff) / 255.;
10       bitmap[i][j][1] = ((rgb & 0xff00) >> 8) / 255.;
11       bitmap[i][j][2] = ((rgb & 0xff0000) >> 16) / 255.;
12       bitmap[i][j][3] = 1.0;//TODO: ((rgb & 0xff000000) >> 24)/255.;
13     } else {
14       index = Color8Bit(colia[ix*dy+iy]);
15       for (k = 0; k < 3; k++) {
16         bitmap[i][j][k] = p->rgb[index][k];
17       }
18       bitmap[i][j][3] = 1.0;
19     }
20   }
21 }
```

Die aktuelle Implementierung des GKS unterstützt noch keine alpha-Werte, deshalb werden diese Werte standardmäßig auf 1.0 gesetzt (Zeilen 12, 18).

[4]Das RGBA-Format besteht aus vier Zahlenwerten, von denen die ersten drei den Rot-, Grün- und Blau-Anteil einer Farbe darstellen (RGB-Werte, vgl. Kapitel 2.2, dritter Absatz). Der vierte Wert ist der Alpha-Wert, der die Transparenz der Farbe festlegt.

Die besetzte Bitmap wird analog zu den Füllflächen auf eine Textur übertragen. Um die Programmstruktur zu vereinfachen und ein Buffer-Objekt zu sparen, wurden sowohl für Texturkoordinaten als auch für Vertices dieselben Koordinaten verwendet, nämlich die Eckpunkte eines normalisierten Systems: `0,0, 1,0, 0,1, 1,1`

Die zu Beginn dieses Abschnittes beschriebene Positionierung der Textur im Fenster wird nun von der *ModelView*-Matrix übernommen:

$$\begin{pmatrix} \frac{2*\text{width}}{\text{Breite}} & 0 & 0 & -1+\frac{2*\text{x}}{\text{width}} \\ 0 & \frac{-2*\text{height}}{\text{Höhe}} & 0 & 1-\frac{2*\text{y}}{\text{height}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Bei *Breite* und *Höhe* handelt es sich wie bei der Matrix in Kapitel 4.1 um die Fensterbreite bzw. -höhe. Die Unterschiede zu dieser Matrix sind in blau dargestellt. Die Variablen *width*, *height*, x und y entsprechen der Beschreibung zu Beginn dieses Abschnittes.

4.3.5. Text

Für die Darstellung von Schrift existieren in OpenGL keine Funktionen, sodass dafür zusätzliche Hilfsfunktionen benötigt werden. Diese haben die Aufgabe, einen Text in eine Rastergrafik umzuwandeln, welche anschließend mittels einer Textur dargestellt werden kann.

Für das Rendern von Text wurde in der Seminararbeit [Win12] bereits die FreeType-Bibliothek untersucht mit dem Ergebnis, dass sie für eine plattformunabhängige und effiziente Textdarstellung sehr gut geeignet ist. Aus diesem Grund wurde ein Modul entwickelt, welches auf der Basis von FreeType Rastergrafiken erstellt. Mit diesem Modul beschäftigt sich Kapitel 5 dieser Arbeit.

Für die Darstellung der Rastergrafik, die nun den Text enthält, könnte theoretisch die *Cellarray*-Routine verwendet werden. Da das Modul jedoch die Rastergrafik bereits als Feld von einzelnen RGBA-Bytes liefert, ist die im vorherigen Abschnitt beschriebene Erstellung der Bitmap überflüssig. Zudem können mit einer eigenen Darstellungsroutine Optimierungen für die Textdarstellung eingesetzt werden. Somit hat die neu implementierte Routine folgende Unterschiede zur `cellarray`-Routine:

- Die Koordinatenumrechnungen und die Berechnung von Position und Größe der Grafik fallen weg, weil das Modul bereits die korrekten Daten liefert.

- Das Durchlaufen der Rastergrafik mit geschachtelten Schleifen entfällt, da bereits eine für OpenGL lesbare Bitmap vorliegt und keine Integerwerte in Bytes aufgeteilt werden müssen. Gleichzeitig wird auf die Abfragen verzichtet, die die Rastergrafik spiegeln, falls $max < min$ gilt.
- Der Verzerrungsfilter für die Textur wurde von `GL_NEAREST` auf `GL_LINEAR` geändert. Der Verzerrungsfilter legt fest, wie die Pixel approximiert werden sollen, wenn sie nicht deckungsgleich zu den Fensterpixeln sind. Bei einem Bild wird die Farbe des am nächsten liegenden Pixels verwendet (schnelle Methode); ein Text wird jedoch besser dargestellt, wenn die Farbe aus den benachbarten Pixeln linear interpoliert wird (rechenintensivere Methode). Im OpenGL-Treiber tritt bei normaler Verwendung keine Verzerrung auf, da im FreeType-Modul eine deckungsgleiche Rastergrafik erzeugt wird.

Wenn die Darstellungsqualität für Text auf `STROKE` gesetzt ist oder FreeType auf dem System nicht verfügbar ist, wird mit Linien und Füllflächen eine alternative Schriftdarstellung emuliert. Details zu dieser Emulation kann [Win12] entnommen werden.

5. Textdarstellung mit FreeType

Für eine geräteunabhängige Darstellung von Schrift wurde ein Modul entwickelt, das die Funktionen von FreeType für die Verwendung in OpenGL-Treibern erweitert und vereinfacht. Der erste Abschnitt stellt die FreeType-Bibliothek kurz vor, anschließend werden Details zur Schnittstelle und zur Implementierung behandelt.

5.1. FreeType

FreeType ist eine plattformunabhängige, freie Programmbibliothek, mit der einfach und einheitlich auf Schriftartdateien zugegriffen werden kann. Mit der Freetype-Bibliothek lassen sich Glyphen von hoher Qualität erzeugen, die anschließend beispielsweise als OpenGL-Textur im GKS GL-Treiber dargestellt werden können.

Die Bibliothek wird seit 1995 von David Turner entwickelt. Am 19. Juni 2013 wurde die aktuelle Version 2.5 veröffentlicht. FreeType ist unabhängig von anderer Software – es wird lediglich ein ANSI-C Compiler benötigt, sodass die Bibliothek auf jedem Computer installiert werden kann [ftp].

5.2. Schnittstelle des Moduls

Das entwickelte Modul stellt folgende Schnittstelle zur Verfügung:

```
int gks_ft_init (void);
```

Diese Methode initialisiert das Modul, indem sie die FreeType-Bibliothek lädt und Fehlerzustände abfängt, falls FreeType nicht installiert ist.

```
int gks_ft_terminate (void);
```

Mit dieser Methode wird der von FreeType reservierte Speicher wieder freigegeben.

```
unsigned char *gks_ft_get_bitmap (int *x, int *y, int *width, int
   *height, gks_state_list_t gkss, const char *text, int length);
```

Die Routine erzeugt aus dem übergebenen Text eine Grauwert-Bitmap und liefert sie zurück. Zudem wird aus der gegebenen Position (x, y) die neue Position berechnet, die der eingestellten Textausrichtung entspricht. Über die Parameter `width` und `height` wird die Breite und Höhe der erzeugten Bitmap bereitgestellt.

Das Format der Bitmap ist ein Feld von Grauwerten. Jeder Grauwert wird in einem Byte durch eine Zahl zwischen 0 (weiß) und 255 (schwarz) dargestellt. Die zweidimensionale Struktur der Grauwerte kann aus dem Feld mithilfe von Breite und Höhe der Bitmap wiederhergestellt werden. Die Farbgebung wird vom OpenGL-Treiber übernommen.

```
int *gks_ft_render (int *x, int *y, int *width, int *height,
   gks_state_list_t gkss, const char *text, int length);
```

Die Routine erzeugt mit einem Aufruf von `gks_ft_get_bitmap` eine Grauwert-Bitmap und erstellt in einer Schleife über die Werte der Bitmap eine neue farbige Rastergrafik. Diese hat das in Kapitel 4.3.4 beschriebene RGBA-Format.

Im OpenGL-Treiber wird die Routine nicht verwendet, da die Farbgebung mit OpenGL-Funktionen effizienter ist. Weil das FreeType-Modul unabhängig von jeglichen Gerätetreibern entwickelt wurde, stellt diese Routine eine allgemeine Möglichkeit für die Erzeugung von farbigem Text bereit.

5.3. Einzelheiten zur Implementierung

Dieser Abschnitt beschreibt Details zur Implementierung des FreeType-Moduls, die nicht aus der Untersuchung der Bibliothek in [Win12] hervorgehen. Der entsprechende Quellcode befindet sich im Anhang B.

Den Routinen zur Erzeugung einer Rastergrafik wird jeweils die GKS-Statusliste *gkss* übergeben, wie aus den Listings des vorherigen Abschnittes ersichtlich ist. Diese Statusliste enthält alle für den OpenGL-Treiber nötigen Attribute, unter anderem auch die der Textdarstellung. Auf diese Weise kann die Implementierung des Moduls direkt auf die Attributwerte zugreifen, ohne dass eine aufwändige Parameterübergabe erfolgen muss. Als Konsequenz ist das FreeType-Modul nun an das GKS gebunden und kann nicht mehr unabhängig für andere Software verwendet werden.

Ein Großteil der Funktionalität konnte ohne großen Aufwand mit den FreeType-Funktionen umgesetzt werden. In den folgenden Unterabschnitten wird auf einige Funktionen eingegangen, die hauptsächlich im Modul bearbeitet werden.

5.3.1. Textrotation

Die Rotation von Text wird durch das GKS-Attribut `chup` festgelegt. Bei diesem Attribut handelt es sich um den Zeichenaufwärtsvektor. In Richtung dieses Vektors werden die Zeichen ausgerichtet; im normalen horizontalen Text zeigt er senkrecht nach oben. Mit dem Vektor wird wie folgt eine Rotationsmatrix berechnet:

$$M = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \qquad \text{mit } \alpha = \operatorname{atan2}(y, x) - \frac{\pi}{2}$$

Die Funktion atan2 berücksichtigt bei der Berechnung des Rotationswinkels α den Quadranten, in dem sich der Vektor befindet. Da der Winkel mit der atan2-Funktion von der positiven x-Achse aus gemessen wird und die zu messende Drehung von der positiven y-Achse ausgeht, muss ein Winkel von $\frac{\pi}{2}$ subtrahiert werden [Win12].

FreeType verwendet stets Festkommazahlen, deshalb müssen die Werte der Matrix jeweils mit 2^{20} multipliziert werden [ftt], wie der Quellcodeauszug zeigt. Im letzten Schritt wird die Rotationsmatrix gesetzt, sodass alle Metriken mit dieser Drehung berechnet werden.

```
1 FT_Matrix rotation;
2 float angle = atan2f(gkss->chup[1], gkss->chup[0]) - M_PI / 2;
3 rotation.xx =  cosf(angle) * 0x10000L;
4 rotation.xy = -sinf(angle) * 0x10000L;
5 rotation.yx =  sinf(angle) * 0x10000L;
6 rotation.yy =  cosf(angle) * 0x10000L;
7 FT_Set_Transform(face, &rotation, NULL);
```

5.3.2. Textausrichtung

Die Positionierung der Glyphen und die Ausrichtung des gesamten Textes wird nicht von der FreeType-Bibliothek übernommen. Vielmehr bietet sie einen einfachen Zugriff auf die erforderlichen Metriken der einzelnen Glyphen, sodass damit der Text positioniert werden kann. Einige der Metriken sind in Abbildung 5.1 dargestellt.

Positionierung einer einzelnen Glyphe im horizontalen Text

Eine Glyphe wird stets von der Grundlinie aus positioniert, im Folgenden wird dieser Positionierungspunkt analog zum Quellcode *pen* genannt; in Abbildung 5.1 ist er jedoch mit *origin* bezeichnet. Der Vektor, der von *pen* auf die tatsächliche Position der Glyphe zeigt, wird *bearing* genannt.

Im ersten Schritt wird die x-Komponente des *bearing*-Vektors auf die Breite des Leerraumes links von der Glyphe gesetzt (in Abb. 5.1 mit *bearingX* bezeichnet). Anschließend wird der Vektor mit Hilfe der Rotationsmatrix in Schreibrichtung gedreht.

Im zweiten Schritt wird der *pen* in Richtung des *bearing*-Vektors verschoben, sodass er sich auf der Grundlinie an der Position befindet, wo der linke Rand der Glyphe die Grundlinie schneidet.

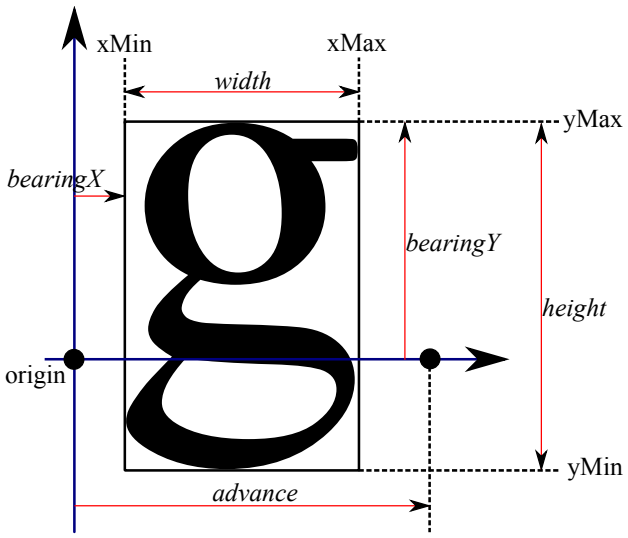

Abbildung 5.1.: Metrikdaten einer Glyphe [ftd]

In einem dritten Schritt wird der *bearing*-Vektor auf die relative Position des *pen* innerhalb der rechteckigen Glyph-Bitmap gesetzt. Mit Hilfe der absoluten *pen*-Position und der relativen Position innerhalb der Bitmap kann die Bitmap schließlich positioniert werden.

```
1  FT_Vector *bearing;
2  bearing->x = face->glyph->metrics.horiBearingX;
3  bearing->y = 0;
4  if (bearing->x != 0) FT_Vector_Transform(bearing, rotation);
5  pen->x -= bearing->x;
6  pen->y -= bearing->y;
7  bearing->x = 64 * face->glyph->bitmap_left;
8  bearing->y = 64 * face->glyph->bitmap_top;
      [...]
9  pos_x = ( pen.x + bearing.x - bb.xMin) / 64;
10 pos_y = (-pen.y - bearing.y + bb.yMax) / 64;
```

Ausrichtung des gesamten Textes

Zur Ausrichtung des gesamten Textes gemäß den GKS-Attributen muss das Rendern der einzelnen Glyphen insgesamt zweimal geschehen: Im ersten Durchlauf wird die sogenannte *Bounding Box* des gesamten Textes berechnet, dabei handelt es sich um den äußeren rechteckigen Rahmen, der um den gerenderten Text gelegt werden kann. Die *Bounding Box* legt die Größe des zu reservierenden Speicherbereiches fest. Erst beim zweiten Rendern werden die Grauwerte der Pixel wirklich in die Bitmap eingetragen.

Die Positionierung der *Bounding Box* erfolgt mit dem Vektor *align*. Die vertikale Ausrichtung erfolgt in Abhängigkeit von der Höhe des Textes *chh*, wie dem folgenden Listing entnommen werden kann:

```
1  align.x = align.y = 0;
2  if (valign != GKS_K_TEXT_VALIGN_BASE) {
3    align.y = gkss->chh * windowheight * 64;
4    FT_Vector_Transform(&align, &rotation);
5    if (valign == GKS_K_TEXT_VALIGN_HALF) {
6      align.x *= 0.5;
7      align.y *= 0.5;
8    } else if (valign == GKS_K_TEXT_VALIGN_TOP) {
9      align.x *= 1.2;
10     align.y *= 1.2;
11   } else if (valign == GKS_K_TEXT_VALIGN_BOTTOM) {
12     align.x *= -0.2;
13     align.y *= -0.2;
14   }
15 }
```

Für die horizontale Ausrichtung wird der Weißraum rechts von dem letzten Buchstaben aufwendig herausgerechnet, damit ein rechtsbündig ausgerichteter Text pixelgenau an der Positionierungslinie endet (Zeilen 2-9). Bei einer rechtsbündigen Ausrichtung wird *align* um die letzte *pen*-Position (ohne Weißraum) erhöht, bei einer zentrierten Ausrichtung ist der Vektor nur halb so lang.

```
1  if (!vertical && halign != GKS_K_TEXT_HALIGN_LEFT) {
2    FT_Vector right;
3    right.x = face->glyph->metrics.width +
       face->glyph->metrics.horiBearingX;
4    right.y = 0;
5    if (right.x != 0) {
6      FT_Vector_Transform(&right, &rotation);
7    }
8    pen.x += right.x - face->glyph->advance.x;
9    pen.y += right.y - face->glyph->advance.y;
10   if (halign == GKS_K_TEXT_HALIGN_CENTER) {
11     align.x += pen.x / 2;
12     align.y += pen.y / 2;
13   } else if (halign == GKS_K_TEXT_HALIGN_RIGHT) {
14     align.x += pen.x;
15     align.y += pen.y;
16   }
17 }
```

Schließlich wird die Position der Rastergrafik (x, y) um den Ausrichtungsvektor verschoben.

```
1  *x += (bb.xMin - align.x) / 64;
2  *y += (bb.yMin - align.y) / 64;
```

6. Zusammenfassung und Ausblick

Als Ergebnis steht nun ein funktionsfähiger logischer Gerätetreiber zur Verfügung. Mit ihm wurde die Möglichkeit geschaffen, GKS-Grafiken in einem OpenGL-basierten Fenster darzustellen. Des Weiteren wurde ein Modul auf der Basis von FreeType entwickelt, mit dem sich Texte plattformunabhängig rendern und darstellen lassen.

Bei der Entwicklung des Treibers wurden die Muster zur Erzeugung von Linien angepasst, sodass sie den Originalen möglichst gleichen. Dies ist durch die Beschränkung auf ein 16 Bit-Muster seitens OpenGL begründet. Eine Alternative wäre die Emulation der Linien mit Texturen, allerdings ist dies sehr aufwendig.

Die Füllflächen können derzeit nur von konvexen Polygonen begrenzt werden, da die entsprechende OpenGL-Primitive keine konkaven Polygone unterstützt. Ein Algorithmus zur Lösung des Problems wurde aufgezeigt und wird noch umgesetzt.

Der entwickelte Treiber legt die Grundlage für die Darstellung von zwei- und dreidimensionaler Grafik in einem Fenster: Dreidimensionale Objekte können mit GR3 bereits in einem OpenGL-Fenster dargestellt werden und mit GKS GL wird die GKS-Ausgabe ebenfalls in einem solchen Fenster dargestellt. Um die Inhalte zu kombinieren, muss ein Konzept dafür entworfen werden, in welcher Weise die 2D-Objekte in einer 3D-Grafik dargestellt werden können: Welche Objekte kommen in den Vordergrund? Werden 2D-Objekte auf Oberflächen im Raum projiziert? Welche Position und Größe haben diese Oberflächen dann und sind sie transparent? Wie wird Text so positioniert, dass er lesbar bleibt?

Mit dem OpenGL-Treiber wird zudem die Entwicklung von weiteren GKS- Gerätetreibern für die GUI-Toolkits einfacher, die OpenGL unterstützen (z. B. Qt). Diese Treiber können einen GL-Kontext erzeugen und die Implementierung des OpenGL-Treibers verwenden. Auf die Einzelheiten der Treiberentwicklung, wie beispielsweise das Zeichnen der einzelnen Primitiven, muss dadurch nicht mehr eingegangen werden.

Schließlich wird auch die Implementierung der Textfunktionen erheblich erleichtert, weil das entwickelte FreeType-Modul allen Treibern zur Verfügung steht. Es rendert aus einer Zeichenkette plattformunabhängig eine Rastergrafik, die dann mit der entsprechenden *Cellarray*-Methode dargestellt werden kann.

Literaturverzeichnis

[BB86] Bechlars, Jörg und Buhtz, Rainer: *GKS in der Praxis*. Springer-Verlag Berlin, Heidelberg, 1986, ISBN 9783540161394.

[ftd] *The design of FreeType 2*.
`http://www.freetype.org/freetype2/docs/design/design-2.html`
(zuletzt geprüft am 03.07.2013).

[ftp] *The FreeType Project*.
`http://freetype.org/freetype2/index.html`
(zuletzt geprüft am 03.07.2013).

[ftt] *FreeType 2 Tutorial*.
`http://freetype.org/freetype2/docs/tutorial/step2.html`
(zuletzt geprüft am 03.07.2013).

[gld] *GLFW Documentation. Window handling*.
`http://www.glfw.org/docs/3.0/group__window.html`
(zuletzt geprüft am 03.07.2013).

[glf] *GLFW Homepage*.
`http://glfw.org`
(zuletzt geprüft am 03.07.2013).

[glm] *Moving from GLFW 2 to 3*.
`http://www.glfw.org/docs/3.0/moving.html`
(zuletzt geprüft am 03.07.2013).

[gls] *Unofficial OpenGL Software Development Kit. GLFW*.
`http://glsdk.sourceforge.net/docs/html/group__module__glfw.html`
(zuletzt geprüft am 03.07.2013).

[Hei11] Heimbach, Ingo: *Entwicklung eines logischen GKS Gerätetreibers für die wxWidgets-Klassenbibliothek*. Forschungszentrum Jülich GmbH, 2011.

[m3d] *The Mesa 3D Graphics Library*.
`http://mesa3d.org/intro.html`
(zuletzt geprüft am 03.07.2013).

[Rhi11] Rhiem, Florian: *Integration OpenGL-basierter Visualisierungs-Techniken in 2D-Grafiksystemen*. Forschungszentrum Jülich GmbH, 2011.

[Rhi12] Rhiem, Florian: *Integration von 3D-Visualisierungstechniken in 2D-Grafiksystemen.* Forschungszentrum Jülich GmbH, 2012.

[SBH96] Schumacher, Dr. H., Busch, M. und Heinen, J.: *2D- und 3D-Visualisierung mit den Graphiksystemen GR, GLI, IDL und AVS – Ein Überblick.* Forschungszentrum Jülich GmbH, Institut für Festkörperforschung, 1996.

[Sei91] Seidel, Raimund: *A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons.* Comput. Geom. Theory Appl, 1:51–64, 1991.

[Shr08] Shreiner, Dave: *OpenGL Programming Guide. Sixth Edition.* Addison Wesley, 2008.

[Win12] Winkler, Jörg: *Rasterisierung von Vektor-Schriften auf der Basis der FreeType Software.* Forschungszentrum Jülich GmbH, 2012.

Anhang

A. Quellcode des OpenGL-Treibers

A.1. Linienzüge

```
1  static
2  void line_routine(int num_points, float *x, float *y, int linetype, int tnr)
3  {
4    int i;
5    float xn, yn, xd, yd;
6    const float modelview_matrix[16] = {
7      2.0f/p->width, 0,                0, -1,
8      0,             -2.0f/p->height, 0, 1,
9      0,             0,                1, 0,
10     0,             0,                0, 1
11   };
12
13   glMatrixMode(GL_MODELVIEW);
14   glLoadTransposeMatrixf(modelview_matrix);
15   glBegin(GL_LINE_STRIP);
16   for (i = 0; i < num_points; ++i) {
17     WC_to_NDC(x[i], y[i], gkss->cntnr, xn, yn);
18     seg_xform(&xn, &yn);
19     NDC_to_DC(xn, yn, xd, yd);
20     glVertex2f(xd,yd);
21   }
22   glEnd();
23   glLoadIdentity();
24 }
25
26 static
27 void polyline(int num_points, float *x, float *y)
28 {
29   static GLushort pattern[13] = {
30     0x0111, 0x0041, 0x5555, 0x7F7F, 0x3CFF, 0x0FFF, 0x249F, 0x111F,
31     0xFFFF,
32     0xFFFF, 0x03FF, 0x1111, 0x087F
33   };
34   static GLint factor[13] = {
35     1, 1, 1, 1, 2, 1, 1, 1,
36     1,
37     1, 1, 1, 1
38   };
39   int ln_type, ln_color;
40   float ln_width;
41
42   ln_type  = gkss->asf[0] ? gkss->ltype  : gkss->lindex;
43   ln_width = gkss->asf[1] ? gkss->lwidth : 1;
44   ln_color = gkss->asf[2] ? Color8Bit(gkss->plcoli) : 1;
45
46   if (gkss->version > 4)
47     ln_width *= p->height / 500.0;
48   ln_width = max(1, nint(ln_width));
49
50   glLineWidth(ln_width);
51   glLineStipple(nint(ln_width * factor[ln_type + 8]), pattern[ln_type + 8]);
```

```
52    glEnable(GL_LINE_STIPPLE);
53    glColor3fv(p->rgb[ln_color]);
54
55    line_routine(num_points, x, y, ln_type, gkss->cntnr);
56
57    glColor3f(0, 0, 0);
58    glDisable(GL_LINE_STIPPLE);
59    glLineWidth(1.0);
60  }
```

A.2. Füllflächen

```
static
void fill_routine (int n, float *px, float *py, int tnr)
{
  int fl_inter, fl_style, i, j, ln_width;
  GLfloat vertices[2*n];
  GLuint texture = 0;
  int parray[33];
  GLubyte bitmap[8][8];
  static GLuint buffer = 0;
  GLboolean draw_pattern = 0;
  float x, y;

  const float modelview_matrix[16] = {
    2.0f/p->width, 0,               0, -1,
    0,             -2.0f/p->height, 0, 1,
    0,             0,               1, 0,
    0,             0,               0, 1
  };
  const float texcoord_matrix[16] = {
    1./8.,  0,      0, 0,
    0,      1./8.,  0, 0,
    0,      0,      1, 0,
    0,      0,      0, 1
  };

  for (i = 0; i < n; i++) {
    WC_to_NDC(px[i], py[i], gkss->cntnr, x, y);
    seg_xform(&x, &y);
    NDC_to_DC(x, y, vertices[2*i], vertices[2*i+1]);
  }

  fl_inter = gkss->asf[10] ? gkss->ints : predef_ints[gkss->findex - 1];

  ln_width = (gkss->version > 4) ? max(1, nint(p->height / 500.0)) : 1;
  glLineWidth(ln_width);

  draw_pattern = (fl_inter == GKS_K_INTSTYLE_PATTERN ||
                  fl_inter == GKS_K_INTSTYLE_HATCH);

  if (draw_pattern) {
    fl_style = gkss->asf[11] ? gkss->styli : predef_styli[gkss->findex - 1];
    if (fl_inter == GKS_K_INTSTYLE_HATCH) fl_style += HATCH_STYLE;
    if (fl_style >= PATTERNS) fl_style = 1;
    gks_inq_pattern_array(fl_style, parray);

    for (i = 0; i < 8; i++) {
      for (j = 0; j < 8; j++) {
        bitmap[(j + 7) % 8][(i + 7) % 8] =
          (parray[(j % parray[0]) + 1] >> i) & 0x01 ? 0 : 255;
      }
    }

    glGenTextures(1, &texture);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_ALPHA, 8, 8, 0, GL_ALPHA,
                 GL_UNSIGNED_BYTE, bitmap);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glEnable(GL_TEXTURE_2D);
    glEnable(GL_BLEND);
```

```
64      glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
65    }
66
67    glMatrixMode(GL_MODELVIEW);
68    glLoadTransposeMatrixf(modelview_matrix);
69    if (draw_pattern) {
70      glMatrixMode(GL_TEXTURE);
71      glLoadTransposeMatrixf(texcoord_matrix);
72      glEnableClientState(GL_TEXTURE_COORD_ARRAY);
73    }
74    glEnableClientState(GL_VERTEX_ARRAY);
75    if (!buffer) {
76      glGenBuffers(1, &buffer);
77      glBindBuffer(GL_ARRAY_BUFFER, buffer);
78    }
79    glBindBuffer(GL_ARRAY_BUFFER, buffer);
80    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
81    glVertexPointer(2, GL_FLOAT, 0, 0);
82    if (draw_pattern) {
83      glTexCoordPointer(2, GL_FLOAT, 0, 0);
84      glDrawArrays(GL_POLYGON, 0, n);
85      glDisable(GL_TEXTURE_2D);
86      glDeleteTextures(1, &texture);
87      glDisable(GL_BLEND);
88    } else if (fl_inter == GKS_K_INTSTYLE_HOLLOW) {
89      glDrawArrays(GL_LINE_LOOP, 0, n);
90    } else if (fl_inter == GKS_K_INTSTYLE_SOLID) {
91      glDrawArrays(GL_POLYGON, 0, n);
92    }
93    glLoadIdentity();
94  }
95
96  static
97  void fillarea(int n, float *px, float *py)
98  {
99    int fl_color;
100
101   fl_color = gkss->asf[12] ? Color8Bit(gkss->facoli) : 1;
102   glColor3fv(p->rgb[fl_color]);
103
104   fill_routine(n, px, py, gkss->cntnr);
105
106   glColor3f(0, 0, 0);
107 }
```

A.3. Marker-Symbole

```
static
void draw_marker(float xn, float yn, int mtype, float mscale, int mcolor)
{
  static int marker[26][57] = {
    { 5, 9, -4, 7, 4, 7, 7, 4, 7, -4,    /* omark */
      4, -7, -4, -7, -7, -4, -7, 4,
      -4, 7, 3, 9, -4, 7, 4, 7, 7, 4,
      7, -4, 4, -7, -4, -7, -7, -4,
      -7, 4, -4, 7, 0 },
    { 5, 13, -2, 8, 2, 8, 2, 2, 8, 2,    /* hollow plus */
      8, -2, 2, -2, 2, -8, -2, -8,
      -2, -2, -8, -2, -8, 2, -2, 2,
      -2, 8, 3, 13, -2, 8, 2, 8,
      2, 2, 8, 2, 8, -2, 2, -2, 2, -8,
      -2, -8, -2, -2, -8, -2, -8, 2,
      -2, 2, -2, 8, 0 },
    { 4, 4, -8, 0, 4, 7, 4, -7,          /* solid triangle right */
      -8, 0, 0 },
    { 4, 4, 8, 0, -4, -7, -4, 7,         /* solid triangle left */
      8, 0, 0 },
    { 5, 4, 0, 8, 7, -4, -7, -4, 0, 8,   /* triangle up down */
      5, 4, 0, -8, -7, 4, 7, 4, 0, -8,
      3, 4, 0, 8, 7, -4, -7, -4, 0, 8,
      3, 4, 0, -8, -7, 4, 7, 4, 0, -8,
      0 },
    { 4, 11, 0, 9, 2, 2, 9, 3, 3, -1,    /* solid star */
      6, -8, 0, -3, -6, -8, -3, -1,
      -9, 3, -2, 2, 0, 9, 0 },
    { 5, 11, 0, 9, 2, 2, 9, 3, 3, -1,    /* hollow star */
      6, -8, 0, -3, -6, -8, -3, -1,
      -9, 3, -2, 2, 0, 9,
      3, 11, 0, 9, 2, 2, 9, 3, 3, -1,
      6, -8, 0, -3, -6, -8, -3, -1,
      -9, 3, -2, 2, 0, 9, 0 },
    { 4, 5, 0, 9, 9, 0, 0, -9, -9, 0,    /* solid diamond */
      0, 9, 0 },
    { 5, 5, 0, 9, 9, 0, 0, -9, -9, 0,    /* hollow diamond */
      0, 9, 3, 5, 0, 9, 9, 0, 0, -9,
      -9, 0, 0, 9, 0 },
    { 4, 5, 9, 9, -9, -9, 9, -9, -9, 9, /* solid hourglass */
      9, 9, 0 },
    { 5, 5, 9, 9, -9, -9, 9, -9, -9, 9, /* hollow hourglass */
      9, 9, 3, 5, 9, 9, -9, -9, 9, -9,
      -9, 9, 9, 9, 0 },
    { 4, 5, 9, 9, 9, -9, -9, 9, -9, -9, /* solid bowtie */
      9, 9, 0 },
    { 5, 5, 9, 9, 9, -9, -9, 9, -9, -9, /* hollow bowtie */
      9, 9, 3, 5, 9, 9, 9, -9, -9, 9,
      -9, -9, 9, 9, 0 },
    { 4, 5, 9, 9, 9, -9, -9, -9, -9, 9, /* solid square */
      9, 9, 0 },
    { 5, 5, 9, 9, 9, -9, -9, -9, -9, 9, /* hollow square */
      9, 9, 3, 5, 9, 9, 9, -9, -9, -9,
      -9, 9, 9, 9, 0 },
    { 4, 4, -9, 9, 9, 9, 0, -9, -9, 9,  /* solid triangle down */
      0 },
    { 5, 4, -9, 9, 9, 9, 0, -9, -9, 9,  /* hollow triangle down */
      3, 4, -9, 9, 9, 9, 0, -9, -9, 9,
      0 },
    { 4, 4, 0, 9, 9, -9, -9, -9, 0, 9,  /* solid triangle up */
      0 },
    { 5, 4, 0, 9, 9, -9, -9, -9, 0, 9,  /* hollow triangle up */
      3, 4, 0, 9, 9, -9, -9, -9, 0, 9, 0 },
```

```
64      { 7, 0, 360, 0 },                   /* solid circle */
65      { 0 },                              /* not used */
66      { 1, 0 },                           /* dot */
67      { 2, 0, 0, 0, 9, 2, 0, 0, 9, 0,     /* plus */
68        2, 0, 0, 0, -9, 2, 0, 0, -9, 0,
69        0 },
70      { 2, 0, 0, 0, 9, 2, 0, 0, 9, 3,     /* asterisk */
71        2, 0, 0, 6, -9, 2, 0, 0, -6, -9,
72        2, 0, 0, -9, 3, 0 },
73      { 8, 0, 360, 6, 0, 360, 0 },        /* circle */
74      { 2, 0, 0, 9, 9, 2, 0, 0, 9, -9,    /* diagonal cross */
75        2, 0, 0, -9, -9, 2, 0, 0, -9, 9,
76        0 }
77    };
78    static int is_concav[26] = {
79      0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
80      0,
81      0, 0, 0, 0, 0
82    };
83    const float modelview_matrix[16] = {
84      2.0f/p->width, 0,                 0, -1,
85      0,             -2.0f/p->height,   0, 1,
86      0,             0,                 1, 0,
87      0,             0,                 0, 1
88    };
89
90    int r, i, num_segments;
91    int pc, op;
92    float scale, x, y, xr, yr, angle, c, s, tmp;
93
94    if (gkss->version > 4)
95      mscale *= p->height / 500.0;
96    r = (int) (3 * mscale);
97    scale = mscale / 3.0;
98
99    xr = r;
100   yr = 0;
101   seg_xform_rel(&xr, &yr);
102   r = nint(sqrt(xr * xr + yr * yr));
103
104   NDC_to_DC(xn, yn, x, y);
105
106   pc = 0;
107   mtype = (2 * r > 1) ? mtype + 20 : 21;
108
109   glMatrixMode(GL_MODELVIEW);
110   glLoadTransposeMatrixf(modelview_matrix);
111   glColor3fv(p->rgb[mcolor]);
112
113   do {
114     op = marker[mtype][pc];
115     switch (op) {
116       case 1:           /* point */
117         glBegin(GL_POINTS);
118         glVertex2f(x, y);
119         glEnd();
120         break;
121
122       case 2:           /* line */
123         glBegin(GL_LINES);
124         for (i = 0; i < 2; i++) {
125           xr = scale * marker[mtype][pc + 2 * i + 1];
126           yr = -scale * marker[mtype][pc + 2 * i + 2];
127           seg_xform_rel(&xr, &yr);
128           glVertex2f(x - xr, y + yr);
129         }
```

```
130           glEnd();
131           pc += 4;
132           break;
133
134         case 3:         /* polygon */
135         case 4:         /* filled polygon */
136         case 5:         /* hollow polygon */
137           if (op == 4) {
138             glBegin(GL_TRIANGLE_FAN);
139           } else if (op == 5) {
140             glColor3f(1, 1, 1);
141             glBegin(GL_TRIANGLE_FAN);
142           } else {
143             glBegin(GL_LINE_LOOP);
144           }
145           if (op != 3 && is_concav[mtype])
146             glVertex2f(x, y);
147           for (i = 0; i < marker[mtype][pc + 1]; i++) {
148             xr = scale * marker[mtype][pc + 2 + 2 * i];
149             yr = -scale * marker[mtype][pc + 3 + 2 * i];
150             seg_xform_rel(&xr, &yr);
151             glVertex2f(x - xr, y + yr);
152           }
153           glEnd();
154           if (op == 5) glColor3fv(p->rgb[mcolor]);
155           pc += 1 + 2 * marker[mtype][pc + 1];
156           break;
157
158         case 6:         /* arc */
159         case 7:         /* filled arc */
160         case 8:         /* hollow arc */
161           {
162             num_segments = 4 * r;
163             angle = (marker[mtype][pc + 2] - marker[mtype][pc + 1]) * M_PI/180;
164             c = cosf(angle / (num_segments - 1));
165             s = sinf(angle / (num_segments - 1));
166             xr = r * cosf(marker[mtype][pc + 1]);
167             yr = r * sinf(marker[mtype][pc + 1]);
168             if (op == 7) {
169               glBegin(GL_TRIANGLE_FAN);
170             } else if (op == 8) {
171               glColor3f(1, 1, 1);
172               glBegin(GL_TRIANGLE_FAN);
173             } else {
174               glBegin(GL_LINE_LOOP);
175             }
176             for (i = 0; i < num_segments; i++) {
177               glVertex2f(x + xr, y + yr);
178               tmp = xr;
179               xr = c * xr  - s * yr;
180               yr = s * tmp + c * yr;
181             }
182             glEnd();
183             if (op == 8) glColor3fv(p->rgb[mcolor]);
184           }
185           pc += 2;
186           break;
187         }
188
189         pc++;
190       }
191     while (op != 0);
192
193     glLoadIdentity();
194   }
195
```

```
196 static
197 void polymarker(int n, float *px, float *py)
198 {
199   int mk_type, mk_color;
200   float mk_size, ln_width, *clrt;
201   float x, y;
202   int i;
203
204   mk_type = gkss->asf[3] ? gkss->mtype : gkss->mindex;
205   mk_size = gkss->asf[4] ? gkss->mszsc : 1;
206   mk_color = gkss->asf[5] ? Color8Bit(gkss->pmcoli) : 1;
207
208   ln_width = (gkss->version > 4) ? max(1, nint(p->height / 500.0)) : 1;
209   glLineWidth(ln_width);
210
211   clrt = gkss->viewport[gkss->cntnr];
212
213   for (i = 0; i < n; i++) {
214     WC_to_NDC(px[i], py[i], gkss->cntnr, x, y);
215     seg_xform(&x, &y);
216     if (gkss->clip != GKS_K_CLIP ||
217         (x >= clrt[0] && x <= clrt[1] && y >= clrt[2] && y <= clrt[3])) {
218       draw_marker(x, y, mk_type, mk_size, mk_color);
219     }
220   }
221   glLineWidth(1.0);
222 }
```

A.4. Rastergrafiken

```
1  static
2  void cellarray(float xmin, float xmax, float ymin, float ymax,
3                 int dx, int dy, int dimx, int *colia, int true_color)
4  {
5    float x1, y1, x2, y2;
6    int x, y, width, height, ix, iy;
7    static GLuint buffer = 0;
8    GLuint texture = 0;
9    int i, j, k, index, rgb;
10   GLfloat bitmap[dx][dy][4];
11
12   WC_to_NDC(xmin, ymax, gkss->cntnr, x1, y1);
13   seg_xform(&x1, &y1);
14   NDC_to_DC(x1, y1, x1, y1);
15
16   WC_to_NDC(xmax, ymin, gkss->cntnr, x2, y2);
17   seg_xform(&x2, &y2);
18   NDC_to_DC(x2, y2, x2, y2);
19
20   x = (int) min(x1, x2);
21   y = (int) min(y1, y2);
22   width  = (int) fabs(x2 - x1) + 1;
23   height = (int) fabs(y2 - y1) + 1;
24
25   const float modelview_matrix[16] = {
26     2.0f*width/p->width, 0,                      0, 2.0f*x/p->width-1,
27     0,                   -2.0f*height/p->height, 0, -2.0f*y/p->height+1,
28     0,                   0,                      1, 0,
29     0,                   0,                      0, 1
30   };
31
32   for (i = 0; i < dx; i++) {
33     ix = (x1 > x2) ? dx - i - 1 : i;
34     for (j = 0; j < dy; j++) {
35       iy = (y1 < y2) ? dy - j - 1 : j;
36       if (true_color) {
37         rgb = colia[ix * dy + iy];
38         bitmap[i][j][0] = (rgb & 0xff) / 255.;
39         bitmap[i][j][1] = ((rgb & 0xff00) >> 8) / 255.;
40         bitmap[i][j][2] = ((rgb & 0xff0000) >> 16) / 255.;
41         bitmap[i][j][3] = 1.0; // TODO: ((rgb & 0xff000000) >> 24) / 255.;
42       } else {
43         index = Color8Bit(colia[ix*dy+iy]);
44         for (k = 0; k < 3; k++) {
45           bitmap[i][j][k] = p->rgb[index][k];
46         }
47         bitmap[i][j][3] = 1.0;
48       }
49     }
50   }
51   glEnable(GL_BLEND);
52   glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
53   glGenTextures(1, &texture);
54   glBindTexture(GL_TEXTURE_2D, texture);
55   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, dx, dy, 0, GL_RGBA, GL_FLOAT, bitmap);
56   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
57   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
58   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
59   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
60
61   glEnable(GL_TEXTURE_2D);
62   glMatrixMode(GL_MODELVIEW);
63   glLoadTransposeMatrixf(modelview_matrix);
```

```
64    glEnableClientState(GL_VERTEX_ARRAY);
65    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
66
67    if (!buffer) {
68      GLfloat vertices[8] = {0,0, 1,0, 0,1, 1,1};
69      glGenBuffers(1, &buffer);
70      glBindBuffer(GL_ARRAY_BUFFER, buffer);
71      glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
72    }
73    glBindBuffer(GL_ARRAY_BUFFER, buffer);
74    glVertexPointer(2, GL_FLOAT, 0, 0);
75    glBindBuffer(GL_ARRAY_BUFFER, buffer);
76    glTexCoordPointer(2, GL_FLOAT, 0, 0);
77    glColor3f(1,1,1);
78    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
79
80    glMatrixMode(GL_MODELVIEW);
81    glLoadIdentity();
82    glDisable(GL_TEXTURE_2D);
83    glDisable(GL_BLEND);
84    glDeleteTextures(1, &texture);
85  }
```

A.5. Text

```
#ifdef XFT

static
void gl_drawimage(int x, int y, int w, int h, unsigned char *bitmap)
{
  static GLuint buffers[2] = {0, 0};
  static GLuint texture = 0;
  int tx_color;

  const float modelview_matrix[16] = {
    2.0f*w/p->width, 0,                0, 2.0f*x/p->width-1,
    0,               2.0f*h/p->height, 0, 2.0f*y/p->height-1,
    0,               0,                1, 0,
    0,               0,                0, 1
  };

  tx_color = gkss->asf[9] ? Color8Bit(gkss->txcoli) : 1;

  if (!texture) {
    glGenTextures(1, &texture);
  }
  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
  glBindTexture(GL_TEXTURE_2D, texture);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_ALPHA, w, h, 0, GL_ALPHA, GL_UNSIGNED_BYTE,
               bitmap);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  glBindTexture(GL_TEXTURE_2D, texture);

  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_TEXTURE_2D);
  glColor3fv(p->rgb[tx_color]);

  glMatrixMode(GL_MODELVIEW);
  glLoadTransposeMatrixf(modelview_matrix);
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  if (!buffers[0]) {
    GLfloat vertices[] = {0,0, 1,0, 0,1, 1,1};
    GLint text_box[] = {0,1, 1,1, 0,0, 1,0};
    glGenBuffers(2, buffers);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(text_box), text_box, GL_STATIC_DRAW);
  }
  glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
  glVertexPointer(2, GL_FLOAT, 0, 0);
  glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
  glTexCoordPointer(2, GL_INT, 0, 0);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

  glLoadIdentity();
  glDisable(GL_TEXTURE_2D);
  glDisable(GL_BLEND);
  glDeleteTextures(1, &texture);
}

#endif
```

```
64
65  static
66  void text(float x_pos, float y_pos, int nchars, char *text)
67  {
68    int tx_color;
69  #ifdef XFT
70    unsigned char *bitmap;
71    int x, y, w, h;
72    int tx_prec = gkss->asf[6] ? gkss->txprec : predef_prec[gkss->tindex - 1];
73
74    if (tx_prec == GKS_K_TEXT_PRECISION_STROKE) {
75  #endif
76      tx_color = gkss->asf[9] ? Color8Bit(gkss->txcoli) : 1;
77      if (tx_color <= 0 || tx_color >= MAX_COLOR) tx_color = 1;
78
79      glColor3fv(p->rgb[tx_color]);
80      gks_emul_text(x_pos, y_pos, nchars, text, line_routine, fill_routine);
81      glColor3f(0,0,0);
82
83  #ifdef XFT
84    } else {
85      NDC_to_DC(x_pos, y_pos, x, y);
86      h = p->height;
87      y = p->height - y;  /* in FT y-axis is in up direction */
88      bitmap = gks_ft_get_bitmap(&x, &y, &w, &h, gkss, text, nchars);
89      if (bitmap != NULL) {
90        gl_drawimage(x, y, w, h, bitmap);
91        free(bitmap);
92      }
93    }
94  #endif
95  }
```

B. Quellcode des FreeType-Moduls

B.1. Glyphe laden und Metriken auslesen

```
1  /* load a glyph into the slot and compute bearing */
2  static FT_Error set_glyph(FT_Face face, FT_UInt codepoint, FT_UInt *previous,
3                            FT_Vector *pen, FT_Bool vertical, FT_Matrix *rotation,
4                            FT_Vector *bearing, FT_Int halign) {
5    FT_Error error;
6    FT_UInt glyph_index;
7
8    glyph_index = FT_Get_Char_Index(face, codepoint);
9    if (FT_HAS_KERNING(face) && *previous && !vertical && glyph_index) {
10     FT_Vector delta;
11     FT_Get_Kerning(face, *previous, glyph_index, FT_KERNING_DEFAULT,
12                    &delta);
13     FT_Vector_Transform(&delta, rotation);
14     pen->x += delta.x;
15     pen->y += delta.y;
16   }
17   error = FT_Load_Glyph(face, glyph_index, vertical ?
18                         FT_LOAD_VERTICAL_LAYOUT : FT_LOAD_DEFAULT);
19   if (error) {
20     gks_perror("Glyph could not be loaded: %c", codepoint);
21     return 1;
22   }
23   error = FT_Render_Glyph(face->glyph, FT_RENDER_MODE_NORMAL);
24   if (error) {
25     gks_perror("Glyph could not be rendered: %c", codepoint);
26     return 1;
27   }
28   *previous = glyph_index;
29
30   bearing->x = face->glyph->metrics.horiBearingX;
31   bearing->y = 0;
32   if (vertical) {
33     if (halign == GKS_K_TEXT_HALIGN_RIGHT) {
34       bearing->x += face->glyph->metrics.width;
35     } else if (halign == GKS_K_TEXT_HALIGN_CENTER) {
36       bearing->x += face->glyph->metrics.width / 2;
37     }
38     if (bearing->x != 0) FT_Vector_Transform(bearing, rotation);
39     bearing->x = 64 * face->glyph->bitmap_left - bearing->x;
40     bearing->y = 64 * face->glyph->bitmap_top  - bearing->y;
41   } else {
42     if (bearing->x != 0) FT_Vector_Transform(bearing, rotation);
43     pen->x -= bearing->x;
44     pen->y -= bearing->y;
45     bearing->x = 64 * face->glyph->bitmap_left;
46     bearing->y = 64 * face->glyph->bitmap_top;
47   }
48   return 0;
49 }
```

B.2. Graustufen-Bitmap rendern

```
1  unsigned char *gks_ft_get_bitmap(int *x, int *y, int *width, int *height,
2                                   gks_state_list_t *gkss, const char *text, int length) {
3    FT_Face face;                  /* font face */
4    FT_Vector pen;                 /* glyph position */
5    FT_BBox bb;                    /* bounding box */
6    FT_Vector bearing;             /* individual glyph translation */
7    FT_UInt previous;              /* previous glyph index */
8    FT_Vector spacing;             /* amount of additional space between glyphs */
9    FT_ULong textheight;           /* textheight in FreeType convention */
10   FT_Error error;                /* error code */
11   FT_Matrix rotation;            /* text rotation matrix */
12   FT_UInt size;                  /* number of pixels of the bitmap */
13   FT_String *file;               /* concatenated font path */
14   const FT_String *font, *prefix; /* font file name and directory */
15   FT_UInt *unicode_string;       /* unicode text string */
16   FT_Int halign, valign;         /* alignment */
17   FT_Byte *mono_bitmap = NULL;   /* target for rendered text */
18   FT_Int num_glyphs;             /* number of glyphs */
19   FT_Vector align;
20   FT_Bitmap ftbitmap;
21   FT_UInt codepoint;
22   int i, j, k, textfont, dx, dy, value, pos_x, pos_y;
23   float angle;
24   const int windowheight = *height;
25   const int direction = (gkss->txp <= 3 && gkss->txp >= 0 ? gkss->txp : 0);
26   const FT_Bool vertical = (direction == GKS_K_TEXT_PATH_DOWN ||
27                             direction == GKS_K_TEXT_PATH_UP);
28   const FT_String *suffix_type1 = ".afm";
29
30   if (!init) gks_ft_init();
31
32   num_glyphs = length;
33   unicode_string = (FT_UInt *) malloc(length * sizeof(FT_UInt) + 1);
34   convert_text((FT_Bytes)text, unicode_string, &num_glyphs);
35
36   if (gkss->txal[0] != GKS_K_TEXT_HALIGN_NORMAL) {
37     halign = gkss->txal[0];
38   } else if (vertical) {
39     halign = GKS_K_TEXT_HALIGN_CENTER;
40   } else if (direction == GKS_K_TEXT_PATH_LEFT) {
41     halign = GKS_K_TEXT_HALIGN_RIGHT;
42   } else {
43     halign = GKS_K_TEXT_HALIGN_LEFT;
44   }
45   valign = gkss->txal[1];
46   if (valign != GKS_K_TEXT_VALIGN_NORMAL) {
47     valign = gkss->txal[1];
48   } else {
49     valign = GKS_K_TEXT_VALIGN_BASE;
50   }
51
52   textfont = abs(gkss->txfont);
53   if (textfont >= 101 && textfont <= 131)
54     textfont -= 100;
55   if (textfont <= 32) {
56     font = gks_font_list[map[textfont - 1] - 1];
57   } else {
58     gks_perror("Invalid font index: %d", gkss->txfont);
59     font = gks_font_list[0];
60   }
61   prefix = gks_getenv("GKS_FONTPATH");
62   if (prefix == NULL) {
63     prefix = GRDIR;
```

```
64    }
65    file = (FT_String *) malloc(strlen(prefix) + 9 + strlen(font) + 4 + 1);
66    strcpy(file, prefix);
67  #ifndef _WIN32
68    strcat(file, "/fonts/");
69  #else
70    strcat(fontdb, "\\FONTS\\");
71  #endif
72    strcat(file, font);
73    strcat(file, ".pfb");
74    error = FT_New_Face(library, file, 0, &face);
75    if (error == FT_Err_Unknown_File_Format) {
76      gks_perror("Unknown file format: %s", file);
77      return NULL;
78    } else if (error) {
79      gks_perror("Could not open font file: %s", file);
80      return NULL;
81    }
82    if (strcmp(FT_Get_X11_Font_Format(face), "Type 1") == 0) {
83      strcpy(file, prefix);
84  #ifndef _WIN32
85      strcat(file, "/fonts/");
86  #else
87      strcat(fontdb, "\\FONTS\\");
88  #endif
89      strcat(file, font);
90      strcat(file, suffix_type1);
91      FT_Attach_File(face, file);
92    }
93    free(file);
94
95    textheight = gkss->chh * windowheight * 64 / caps[map[textfont - 1] - 1];
96    error = FT_Set_Char_Size(face, textheight * gkss->chxp, textheight, 72, 72);
97    if (error) gks_perror("Cannot set text height");
98
99    if (gkss->chup[0] != 0.0 || gkss->chup[1] != 0.0) {
100     angle = atan2f(gkss->chup[1], gkss->chup[0]) - M_PI / 2;
101     rotation.xx =  cosf(angle) * 0x10000L;
102     rotation.xy = -sinf(angle) * 0x10000L;
103     rotation.yx =  sinf(angle) * 0x10000L;
104     rotation.yy =  cosf(angle) * 0x10000L;
105     FT_Set_Transform(face, &rotation, NULL);
106   } else {
107     FT_Set_Transform(face, NULL, NULL);
108   }
109
110   spacing.x = spacing.y = 0;
111   if (gkss->chsp != 0.0) {
112     error = FT_Load_Glyph(face, FT_Get_Char_Index(face, ' '),
113                           vertical ? FT_LOAD_VERTICAL_LAYOUT : FT_LOAD_DEFAULT);
114     if (!error) {
115       spacing.x = face->glyph->advance.x * gkss->chsp;
116       spacing.y = face->glyph->advance.y * gkss->chsp;
117     } else {
118       gks_perror("Cannot apply character spacing");
119     }
120   }
121
122   bb.xMin = bb.yMin = LONG_MAX;
123   bb.xMax = bb.yMax = LONG_MIN;
124   pen.x = pen.y = 0;
125   previous = 0;
126
127   for (i = 0; i < num_glyphs; i++) {
128     codepoint = unicode_string[direction == GKS_K_TEXT_PATH_LEFT ?
129                                (num_glyphs - 1 - i) : i];
```

```
130       error = set_glyph(face, codepoint, &previous, &pen, vertical, &rotation,
131                         &bearing, halign);
132       if (error) continue;
133
134       bb.xMin = min(bb.xMin, pen.x + bearing.x);
135       bb.xMax = max(bb.xMax, pen.x + bearing.x + 64 * face->glyph->bitmap.width);
136       bb.yMin = min(bb.yMin, pen.y + bearing.y - 64 * face->glyph->bitmap.rows);
137       bb.yMax = max(bb.yMax, pen.y + bearing.y);
138
139       if (direction == GKS_K_TEXT_PATH_DOWN) {
140         pen.x -= face->glyph->advance.x + spacing.x;
141         pen.y -= face->glyph->advance.y + spacing.y;
142       } else {
143         pen.x += face->glyph->advance.x + spacing.x;
144         pen.y += face->glyph->advance.y + spacing.y;
145       }
146     }
147     *width  = (int)((bb.xMax - bb.xMin) / 64);
148     *height = (int)((bb.yMax - bb.yMin) / 64);
149     if (bb.xMax <= bb.xMin || bb.yMax <= bb.yMin) {
150       gks_perror("Invalid bitmap size");
151       return NULL;
152     }
153     size = *width * *height;
154     mono_bitmap = (FT_Byte *) safe_realloc(mono_bitmap, size);
155     memset(mono_bitmap, 0, size);
156
157     pen.x = 0;
158     pen.y = 0;
159     previous = 0;
160
161     for (i = 0; i < num_glyphs; i++) {
162       bearing.x = bearing.y = 0;
163       codepoint = unicode_string[direction == GKS_K_TEXT_PATH_LEFT ?
164                                  (num_glyphs - 1 - i) : i];
165       error = set_glyph(face, codepoint, &previous, &pen, vertical, &rotation,
166                         &bearing, halign);
167       if (error) continue;
168
169       pos_x = ( pen.x + bearing.x - bb.xMin) / 64;
170       pos_y = (-pen.y - bearing.y + bb.yMax) / 64;
171       ftbitmap = face->glyph->bitmap;
172       for (j = 0; j < ftbitmap.rows; j++) {
173         for (k = 0; k < ftbitmap.width; k++) {
174           dx = k + pos_x;
175           dy = j + pos_y;
176           value = mono_bitmap[dy * *width + dx];
177           value += ftbitmap.buffer[j * ftbitmap.pitch + k];
178           if (value > 255) {
179             value = 255;
180           }
181           mono_bitmap[dy * *width + dx] = value;
182         }
183       }
184
185       if (direction == GKS_K_TEXT_PATH_DOWN) {
186         pen.x -= face->glyph->advance.x + spacing.x;
187         pen.y -= face->glyph->advance.y + spacing.y;
188       } else {
189         pen.x += face->glyph->advance.x + spacing.x;
190         pen.y += face->glyph->advance.y + spacing.y;
191       }
192     }
193
194
195
```

```
196   /* Alignment */
197   if (direction == GKS_K_TEXT_PATH_DOWN) {
198     pen.x += spacing.x;
199     pen.y += spacing.y;
200   } else {
201     pen.x -= spacing.x;
202     pen.y -= spacing.y;
203   }
204
205   align.x = align.y = 0;
206   if (valign != GKS_K_TEXT_VALIGN_BASE) {
207     align.y = gkss->chh * windowheight * 64;
208     FT_Vector_Transform(&align, &rotation);
209     if (valign == GKS_K_TEXT_VALIGN_HALF) {
210       align.x *= 0.5;
211       align.y *= 0.5;
212     } else if (valign == GKS_K_TEXT_VALIGN_TOP) {
213       align.x *= 1.2;
214       align.y *= 1.2;
215     } else if (valign == GKS_K_TEXT_VALIGN_BOTTOM) {
216       align.x *= -0.2;
217       align.y *= -0.2;
218     }
219   }
220
221   if (!vertical && halign != GKS_K_TEXT_HALIGN_LEFT) {
222     FT_Vector right;
223     right.x = face->glyph->metrics.width + face->glyph->metrics.horiBearingX;
224     right.y = 0;
225     if (right.x != 0) {
226       FT_Vector_Transform(&right, &rotation);
227     }
228     pen.x += right.x - face->glyph->advance.x;
229     pen.y += right.y - face->glyph->advance.y;
230     if (halign == GKS_K_TEXT_HALIGN_CENTER) {
231       align.x += pen.x / 2;
232       align.y += pen.y / 2;
233     } else if (halign == GKS_K_TEXT_HALIGN_RIGHT) {
234       align.x += pen.x;
235       align.y += pen.y;
236     }
237   }
238
239   *x += (bb.xMin - align.x) / 64;
240   *y += (bb.yMin - align.y) / 64;
241   return mono_bitmap;
242 }
```

B.3. RGBA-Bitmap erzeugen

```
1  /* create rgba bitmap and compute position */
2  int *gks_ft_render(int *x, int *y, int *width, int *height,
3                     gks_state_list_t *gkss, const char *text, int length) {
4    FT_Byte *rgba_bitmap = NULL;
5    float red, green, blue;
6    int tmp, size, i, j;
7    int color[3];
8    unsigned  char *mono_bitmap = gks_ft_get_bitmap(x, y, width, height, gkss, text,
         length);
9    gks_inq_rgb(gkss->txcoli, &red, &green, &blue);
10   color[0] = (int)(red * 255);
11   color[1] = (int)(green * 255);
12   color[2] = (int)(blue * 255);
13
14   size = *width * *height;
15   rgba_bitmap = (FT_Byte *) safe_realloc(rgba_bitmap, 4 * size);
16   memset(rgba_bitmap, 0, 4 * size);
17   for (i = 0; i < size; i++) {
18     for (j = 0; j < 3; j++) {
19       tmp = rgba_bitmap[4*i + j] + color[j] * mono_bitmap[i] / 255;
20       rgba_bitmap[4*i + j] = (FT_Byte) min(tmp, 255);
21     }
22     tmp = rgba_bitmap[4*i + 3] + mono_bitmap[i];
23     rgba_bitmap[4*i + 3] = (FT_Byte) min(tmp, 255);
24   }
25
26   free(mono_bitmap);
27   return (int *) rgba_bitmap;
28 }
```