FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Seminararbeit im Rahmen des Bachelorstudiengangs Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 - Medizintechnik und Technomathematik

Seminarthema:

Entwicklung eines logischen GKS Gerätetreibers für die wxWidgets-Klassenbibliothek

---

Jülich, 21. Dezember 2011

Ingo Heimbach

Diese Arbeit wurde betreut von:
Prof. Dr. rer. nat. Jobst Hoffmann
Josef Heinen

Forschungszentrum Jülich GmbH

Peter Grünberg Institut
Jülich Centre for Neutron Science

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Entwicklung eines logischen GKS Gerätetreibers
für die wxWidgets-Klassenbibliothek

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Name: Ingo Heimbach

Jülich, den 21. Dezember 2011

Unterschrift der Studentin / des Studenten

Die Wissenschaftler des Peter Grünberg Instituts/Jülich Centre for Neutron Science untersuchen Form und Dynamik von Materialen wie Polymeren, Zusammenlagerungen großer Moleküle und biologischen Zellen sowie die elektronischen Eigenschaften von Festkörpern. Sowohl in den experimentellen als auch in den theoretischen Instituten werden zahlreiche Visualisierungs-Systeme zur Darstellung der Mess- oder Simulationsergebnisse benutzt. Dabei kommt bei der Entwicklung der Benutzeroberflächen häufig das wxWidgets-Toolkit zum Einsatz.
Den Schwerpunkt dieser Seminararbeit bildet die Erstellung eines logischen Gerätetreibers für ein im Hause entwickeltes Grafisches Kernsystem (GKS), um die Visualisierung in den o. g. Anwendungen zu vereinheitlichen und zu vereinfachen. Das Modul soll sowohl in vorhandenen GUI Applikationen als gewöhnliche Oberflächenkomponente als auch in lose gekoppelten und über das Netzwerk verteilten Anwendungen nutzbar sein.
Im Rahmen einer Bachelorarbeit kann die Aufgabenstellung in Analogie zum GKS-Konzept um die Erstellung einer Meta-Ebene erweitert werden, welche die Funktionen und Eigenschaften verschiedener graphischer Benutzeroberflächen (Qt, wxWidgets, GTK) abstrahiert. Ein besonderes Augenmerk ist hierbei auf den möglichen Einsatz in modernen objekt-orientierten Interpretern (Python) zu legen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Motivation

Visualisierungssysteme finden im wissenschaftlichen Umfeld weite Verbreitung und große Anwendung, so auch im Peter Grünberg Institut und Jülich Centre for Neutron Science. Sie bieten die Möglichkeit verschiedene, zum Teil sehr komplexe Vektorgrafiken zu visualisieren, die beispielsweise unterstützend zur Auswertung von Messergebnissen oder zur Darstellung physikalischer Zusammenhänge genutzt werden können. Viele dieser Systeme basieren auf einer Implementierung eines Grafischen Kernsystems (GKS), welches unabhängig vom verwendeten Ausgabegerät ist. GKS bietet verschiedene Ausgabemöglichkeiten, die von Postscripttreibern über Web-Anwendungen bis hin zu grafischen Benutzeroberflächen[1] reichen.
Bei der Entwicklung von GUI-Anwendungen erfreut sich das wxWidgets-Toolkit großer Beliebtheit, da es eine einfache plattformübergreifende Entwicklung grafischer Applikationen ermöglicht, die auf vielen verbreiteten Plattformen (wie Linux, Mac OS X, Windows) ein natives *Look and Feel* bieten.
Aufgrund der oben aufgeführten Punkte und der Tatsache, dass bis zu Beginn dieser Seminararbeit noch kein GKS-Treiber für wxWidgets existiert, erschien es sinnvoll diese Lücke mit einer geeigneten Implementierung zu schließen. Bisher konnten Grafiken des GKS ausschließlich unter Verwendung von Drittsoftware innerhalb von wxWidgets-Applikationen genutzt werden. Diese Lösungen stützen sich dabei auf andere bereits implementierte Subsysteme, die Rastergrafiken (wie *Portable Network Graphics*) erstellen und diese dann mit passenden wxWidgets-Anzeigeelementen in die jeweilige GUI importieren. Durch diesen Umweg werden allerdings Systemressourcen unnötig belastet bzw. die Vorteile von Vektorgrafiken (schnellere Reaktionszeiten bei Auflösungsänderungen und beliebige Größenskalierbarkeit) gehen vollkommen verloren. Außerdem steht auf diese Weise kein wiederverwendbarer und einfach einzubindender Code zur Verfügung, der in beliebigen Anwendungen, sowohl als gewöhnliches *Widget* als auch als Stand-Alone-Anwendung, verwendet werden kann und sich zusätzlich auf das flexible Client-Server-Prinzip eines GKS stützt. All diese hier beschriebenen Problemfelder sollen durch diese Seminararbeit ausgeräumt werden.

[1] im Englischen auch *Graphical User Interface*, kurz GUI, genannt

2 Grundlagen

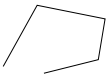
In den folgenden Unterkapiteln werden die Grundlagen für das GKS, wxWidgets und die verwendeten Programmiertechniken vermittelt, die eine essentielle Rolle für die Entwicklung des GKS-Anzeigetreibers in wxWidgets gespielt haben.

2.1 GKS

GKS ist die Kurzform für ***G****rafisches* ***K****ern****s****ystem* und stellt eine internationale Industrienorm dar, die vorgibt, welche Schnittstellen und Leistungen ein GKS zu bieten hat [7, S. 1]. Somit gibt es nicht **das** GKS, sondern es existieren verschiedenste Implementierungen, die jedoch alle kompatibel zu der ISO-Norm und somit untereinander austauschbar sind [7, S. V].

GKS ist für die Darstellung von zweidimensionalen Grafiken (vor allen Dingen Vektorgrafiken) konzipiert und bietet eine abstrahierte Schnittstelle, um als Universalbindeglied zwischen Anwendungen und Ausgabegeräten fungieren zu können. Hiermit wird es ermöglicht, jedes GKS-kompatible Anzeigegerät mit jeder GKS-kompatiblen Anwendung nutzen zu können, ohne dass hardwareabhängiger Code in den Programmen verwendet werden müsste[2] [7, S. 1].

GKS stellt allerdings nur elementare Grafikoperationen bereit. Komplexere Funktionen werden nicht direkt vom GKS unterstützt und müssen daher durch Primitive zusammengesetzt werden. Zu diesen grundlegenden Grafikobjekten gehören [7, S .2f.]:

Polylines	Aneinanderreihung verschiedener Geradenstücke	
Polymarker	Symbole	
Text	Textausgabe mit verschiedenen Schriftarten und Attributen	Beispieltext
Fill Area	Füllung einer geschlossenen Fläche mit Farbe, einer Schraffur oder einer Musterung	
Cell Array	Einfügen von Farbmatrizen (ähnlich Rastergrafiken)	

[2] Dieser Ansatz wird auch *Device Independent Device Dependent* (DIDD) genannt.

2.1.1 GLI und GR

Sollen weitere Funktionen, wie z. B. logarithmische Koordinatensysteme, Histogramme u. Ä. für den Anwender zur Verfügung stehen, so müssen weitere Softwareschichten (s. Abb. 1) über dem GKS angesiedelt werden, die die Erstellung dieser Grafiken aus GKS Primitiven übernehmen. Im Forschungszentrum Jülich wird genau diese Aufgabe von der *GLI*- oder *GR*-Software übernommen [8, Kap. 1, S. 3] [9, S .VII].
GLI steht dabei für *Graphics Language Interpreter* und bietet eine Interpreterschnittstelle zu einigen mächtigen Subsystemen wie dem GUS (*Graphics Utility System*), welches mit einfachen Befehlen komplexe 2D- und 3D-Grafiken auf GKS-Basis erzeugen kann [8, Kap. 1, S. 5]. Da die GUS-Bibliothek jedoch an traditionelle Programmierumgebungen angelehnt ist, war mit der Einführung moderner, interpretierter Sprachen eine Überarbeitung notwendig, um weiterhin eine einfache und auch moderne Schnittstelle zu den Grafikroutinen bereitstellen zu können. So ist die GR-Bibliothek entstanden [???].

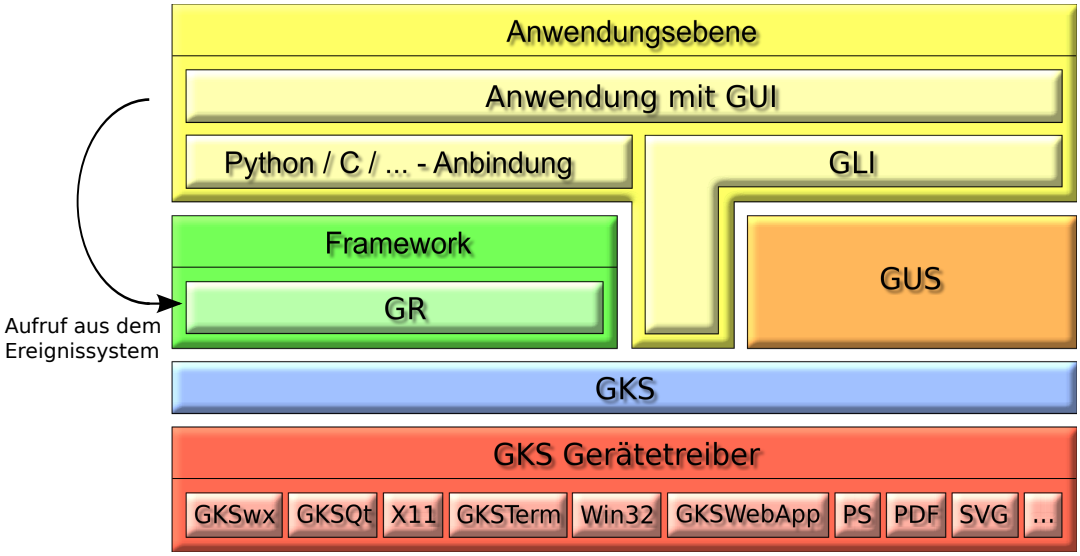


Abbildung 1: Vereinfachte Darstellung der GKS-Struktur [???]

2.1.2 Workstationkonzept

Die Abstraktion der GKS-Ausgabeschnittstelle wird durch das *Workstationkonzept* erreicht, welches alle relevanten Eigenschaften eines Ausgabegeräts in einer *Workstation Description Table* zusammenfasst [7, S. 38f.]. Die eigentlichen Zeicheninstruktionen werden in der Regel zusätzlich in Form einer Displayliste an die Ausgabegeräte gereicht.
Der Kopf einer Displayliste enthält die Gesamtlänge der Daten (ohne die Längenangabe), gefolgt von hintereinander abgelegten Unterlisten, die jeweils einen Befehl beschreiben (s. Abb. 2 auf S. 4). Eine solche Subliste besteht hierbei wiederum aus einer Längenangabe, einem Befehlscode, der eindeutig auf den auszuführenden Befehl abgebildet werden kann und einer Liste von Befehlsparametern des Kommandos (wie z. B. einer Punktliste, wenn

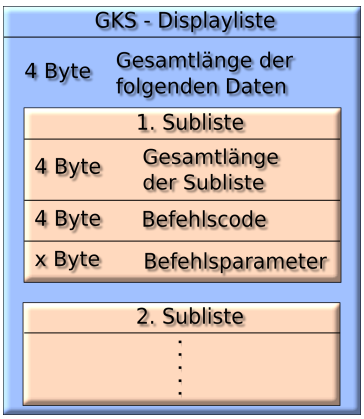


Abbildung 2: Aufbau einer GKS-Displayliste

ein Linienzug gezeichnet werden soll). Wie die Parameter aufgebaut sind, hängt von dem gesendeten Befehl ab [11, S. 9f.].

2.1.3 Koordinatentransformation

Neben dem *Workstationkonzept* spezifiziert die GKS-Norm eine zweistufige Koordinatentransformation, um dem Anwender einen einheitlichen Koordinatenraum zur Verfügung zu stellen. Dies ermöglicht dem Anwender ein beliebiges Koordinatensystem zu definieren, auf das sich die Grafikbefehle des Anwenders beziehen. Die eingegebenen Koordinaten werden dann intern auf einen vorher ebenfalls festgelegten Ausschnitt eines einheitlichen Koordinatenraums umgerechnet (in *Normalized Device Coordinates*). Ein definierbarer Teil dieses Raums wird dann schließlich auf die entsprechenden Gerätekoordinaten transformiert (s. Abb. 3) [7, S. 49ff.].

Im Rahmen dieser Seminararbeit kann nur ein kurzer Einblick in die umfangreiche Funktionalität eines GKS gegeben werden. Für tiefer gehende Informationen sei an dieser Stelle auf weiterführende Literatur ([7, 8, 9, 10]) verwiesen.

2.2 wxWidgets-Toolkit

Das wxWidgets-Toolkit ist eine objektorientierte Bibliothek zur plattformübergreifenden Entwicklung von grafischen Benutzeroberflächen auf sehr vielen Plattformen, wie z. B. Windows, Mac OS X, Linux und anderen unixartigen Systemen [1, S. 1f.].
Ins Leben gerufen wurde wxWidgets von Julian Smart an der Universität Edinburgh im Jahr 1992, um die Entwicklung einer selbstgeschriebenen Anwendung parallel sowohl auf Windows als auch auf Sun-Systemen vorantreiben zu können. Hierbei entstand die erste Version von wxWidgets, die rasch auch von anderen Programmierern verwendet wurde und so eine anwachsende Community fand, die sich dem Voranschreiten der Entwicklung der Bibliothek annahm [1, S. 5ff.]. Heute[3] ist wxWidgets in der Version

[3] Stand Dezember 2011

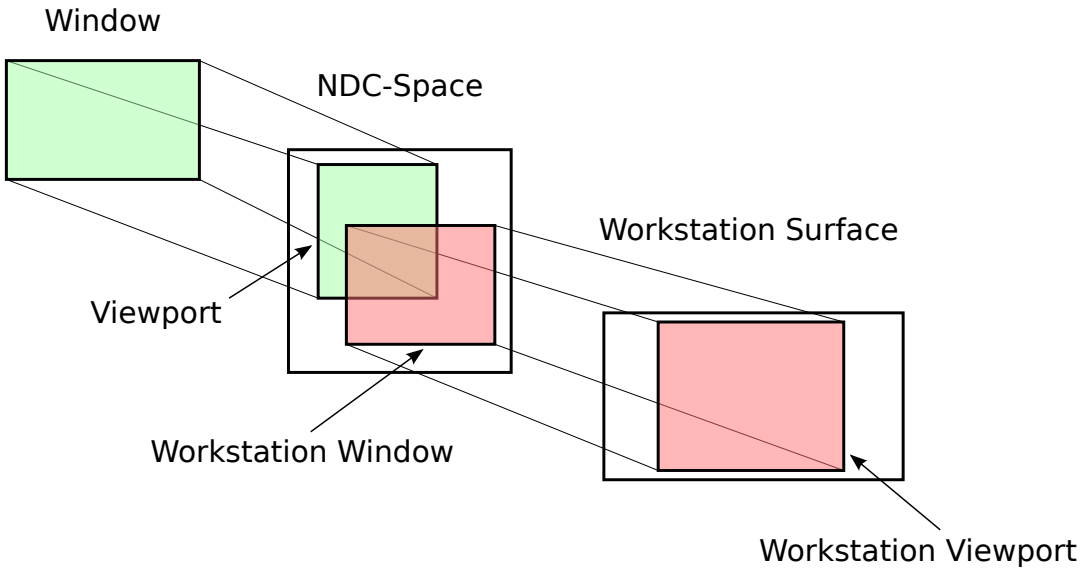


Abbildung 3: Ablauf der 2-Wege-Transformation des GKS

Abbildung 4: Auflistung der verschiedenen wxWidgets-Portierungen [1, S. 8f.]

2.9.3 verfügbar [2], unter einer modifizierten LGPL[4] für die freie Verwendung lizenziert und bietet mittlerweile viel mehr als reine GUI-Programmierung. Die Zielsetzung von wxWidgets ist es, nicht nur plattformübergreifende Programmierung von Benutzeroberflächen zu ermöglichen, sondern allgemein die Entwicklung plattformunabhängiger Anwendungen zu vereinfachen. Aus diesem Grund bietet wxWidgets zusätzlich eine Reihe von häufig benötigten Funktionalitäten, wie Netzwerk- und Threadprogrammierung oder beispielsweise Bildverarbeitung, sodass Entwickler in vielen Fällen Code schreiben können, der ohne zusätzliche Anpassungen auf allen Plattformen funktioniert (s. Abb. 4), für die wxWidgets verfügbar ist [1, S. 1][2].
Die wxWidgets-Bibliothek ist in C++ programmiert und richtet sich primär an C++-Entwickler, jedoch existieren auch *Wrapper*[5] für andere Programmiersprachen wie Python, Perl oder Javascript [1, S. XXVI].
Eine der größten Stärken von wxWidgets verglichen mit anderen GUI-Toolkits, mit denen ebenfalls plattformübergreifender Code erstellt werden kann (wie z. B. Qt oder GTK+), liegt darin, dass wxWidgets möglichst immer auf die systemspezifischen Fra-

[4] wxWidgets steht momentan unter der *wxWindows-Lizenz*, die sowohl die Entwicklung quelloffener als auch properitärer Software ermöglicht [3].
[5] *Wrapper* stellen eine Hülle für Software dar, mit der Bibliotheken beispielsweise um eine Schnittstelle für eine Sprache erweitert werden können, für die sie ursprünglich nicht geschrieben wurde.

meworks zugreift, um die genutzten *Widgets*[6] im nativen *Look and Feel* des verwendeten Systems darzustellen (s. Abb. 5). Viele andere Bibliotheken versuchen das Aussehen des jeweiligen Systems nachzuahmen. Im Gegensatz hierzu verwendet wxWidgets tatsächlich die Komponenten des Systems, auf dem das Programm ausgeführt wird und Anwender bemerken in vielen Fällen gar nicht, dass die Anwendung nicht speziell für ihr System entworfen wurde [1, S. 2ff.].
Mit wxWidgets entwickelte Oberflächen sind immer hierarchisch aufgebaut. Dieses Konzept orientiert sich an der hierarchischen Natur klassischer GUIs und sorgt für baumartige Abhängigkeiten zwischen den einzelnen *Widgets*. Da sich Teilbäume in einer Hierarchie immer wie eigenständige Bäume verhalten, können Teile einer grafischen Oberfläche so leicht als selbstständiges *Widget* gruppiert werden, welches in mehreren Anwendungen wiederverwendet werden kann [S. 40f.][1].

2.2.1 Eventsystem

Klassischerweise werden Algorithmen zur Lösung eines Problems mit dem imperativen Programmierparadigma umgesetzt. Das bedeutet, dass der Programmierer in einer imperativen Programmiersprache (wie z. B. Fortran, C oder Pascal) beschreibt, *wie* das Problem zu lösen ist. Es wird also eine konkrete Abfolge von Befehlen vorgegeben, die von dem Computer sequentiell abzuarbeiten ist.
Im Zusammenhang mit der Entwicklung grafischer Bedienoberflächen stellt man fest, dass die imperative Programmierung nicht immer das geeignete Mittel zur Lösungsfindung darstellt. Benutzeroberflächen werden dafür entworfen, um mit dem späteren Anwender in einen Dialog zu treten, indem sie Eingaben in Form von Tastatureingaben und Mausklicks entgegennehmen. Intern muss sich die Anwendung jedoch auch noch um Verwaltungsarbeiten kümmern, die beispielsweise dazu dienen das Fenster in regelmäßigen Abständen neu zu zeichnen, um die Anzeige stets auf dem aktuellen Stand zu halten. Dabei erhält der Anwender den Eindruck, dass all diese Aufgaben vollkommen parallel ausgeführt würden.
Aus diesem Grund bauen alle modernen Bibliotheken zur GUI-Programmierung (auch wxWidgets) nicht auf dem imperativen, sondern auf dem ereignisgesteuerten Programmierparadigma auf, welches dem passiven Charakter einer grafischen Benutzeroberfläche gerecht wird. Speziell in wxWidgets werden Ereignisse wie Mausklicks als *Events* modelliert, welche von *Eventhandlern* verarbeitet werden (das kann z. B. das Hauptfenster der Anwendung sein oder auch Unterkomponenten, wie ein Knopf, auf dem ein Mausklick ausgeführt wurde). Der *Eventhandler* überprüft, ob zuvor für den generierten Ereignistyp eine *Callback*-Routine[7] registriert wurde. Ist dies der Fall, so wird diese mit dem dazugehörigen Ereignisobjekt aufgerufen, damit innerhalb der Routine genauere Informationen über das Ereignis bereitstehen, sodass beispielsweise nicht nur bekannt ist, dass mit der Maus geklickt wurde, sondern auch an welcher Position der Benutzerober-

[6] *Widgets* sind in wxWidgets die einzelnen GUI-Komponenten wie Textboxen oder Knöpfe.
[7] Callback-Routinen sind Funktionen, die einer anderen Funktion üblicherweise als Funktionszeiger übergeben werden, um sie innerhalb eines Algorithmus als austauschbare Teilfunktionalität verwenden zu können.

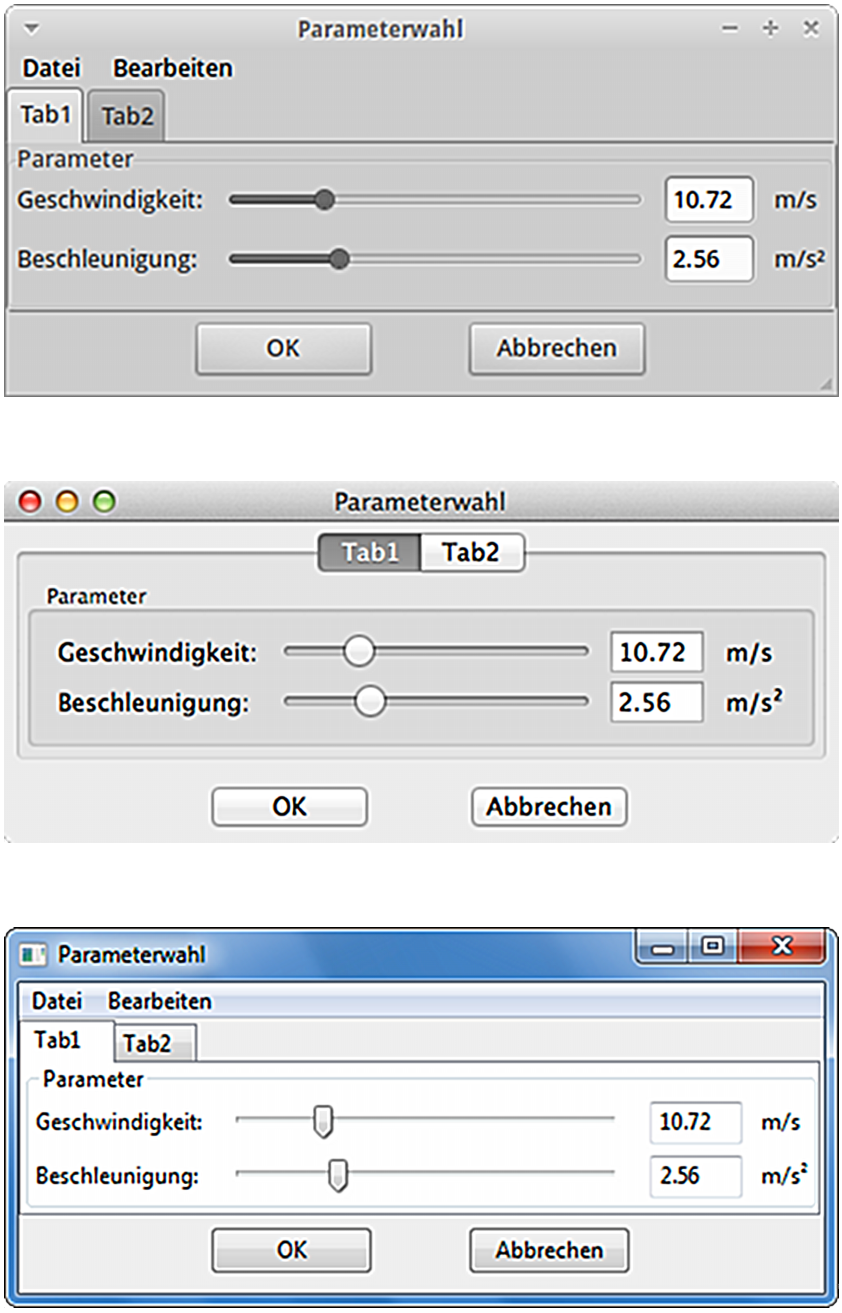


Abbildung 5: Beispielanwendung mit wxWidgets auf verschiedenen Plattformen (oben: Xubuntu mit GTK+, mittig: Mac OS X Lion, unten: Windows 7)

fläche. Sollte keine Funktion registriert sein, so wird das Ereignis je nach Typ an den nächsten verfügbaren *Eventhandler* weitergegeben [1, S. 25ff.].
Das bereits vorgestellte GKS basiert auf einem rein imperativen Ansatz. Der Benutzer übergibt eine Eingabe, die nach einer sequentiellen Verarbeitung wieder ausgegeben wird (*EVA*-Prinzip [14]). Der Anwender kann also nur bedingt durch die Wahl der Eingabedaten Einfluss auf den Programmablauf nehmen, der ansonsten fest durch die Codierungsreihenfolge der internen GKS-Anweisungen bestimmt ist. In diesem Fall ist die Anwendung daher die bestimmende Komponente und kann auch als *Master* bezeichnet werden. Bei einem GUI-basierten Programm bestimmt hingegen der Benutzer selber hauptsächlich den Ablauf der Anwendung, da sie kontinuierlich auf neue Eingaben reagiert (hier kann der Anwender als *Master* angesehen werden). Ein Ziel dieser Seminararbeit ist es daher, diese beiden recht unterschiedlichen Ansätze in einem Programm zu vereinen.
Konkret sieht die Eventverarbeitung des wxWidgets-Toolkits (s. a. Abb. 6) folgendermaßen aus [1, S. 613ff]:

1. *Eventhandler* können untereinander mit Zeigern als doppelt verkettete Liste organisiert sein. In den meisten Fällen liegt keine Verkettung vor, sollte es aber dennoch so sein, dann werden zunächst nacheinander alle *Handler* der Liste angesprochen, bis einer gefunden wird, bei dem eine passende *Callback*-Routine registriert wurde.

2. In wxWidgets werden zwei verschiedene Arten von *Events* eingesetzt:

   - **gewöhnliche Events**, die nur auf der aktuellen Hierarchiestufe verarbeitet werden. Diese werden nur innerhalb der verketteten Liste des *Eventhandlers* verarbeitet, der zur aktuellen Hierarchiestufe gehört.

   - **propagierende Events**, welche zusätzlich zu den normalen *Events* in der Hierarchie heraufgereicht werden und auch dort alle Listen von *Eventhandlern* durchlaufen.

3. Sollte kein passender *Eventhandler* gefunden worden sein, so wird das Ereignis (sowohl gewöhnliche als auch propagierende) zusätzlich an das globale *Application*-Objekt gegeben, welches in jeder wxWidgets-Anwendung existiert. Dessen *Eventhandler* kann dann evtl. noch auf das *Event* reagieren.

Da wxWidgets in C++ programmiert ist, ist das Eventsystem intern natürlich auch mit imperativen Alogorithmen realisiert. Das globale *Application*-Objekt verfügt über eine *Mainloop*, eine Schleife, welche auftretende Ereignisse an die *Eventhandler* verteilt und die registrierten Callback-Routinen aufruft [1, S. 25ff.]. Da all dies jedoch durch die wxWidgets-Bibliothek gekapselt wird, kann man sich als Programmierer rein auf die Ereignissteuerung konzentrieren, es sei denn, man möchte zusätzlich das Paradigma der Parallelprogrammierung nutzen, welches auch von wxWidgets unterstützt wird. Hierauf wird aber noch genauer im folgenden Unterkapitel eingegangen.

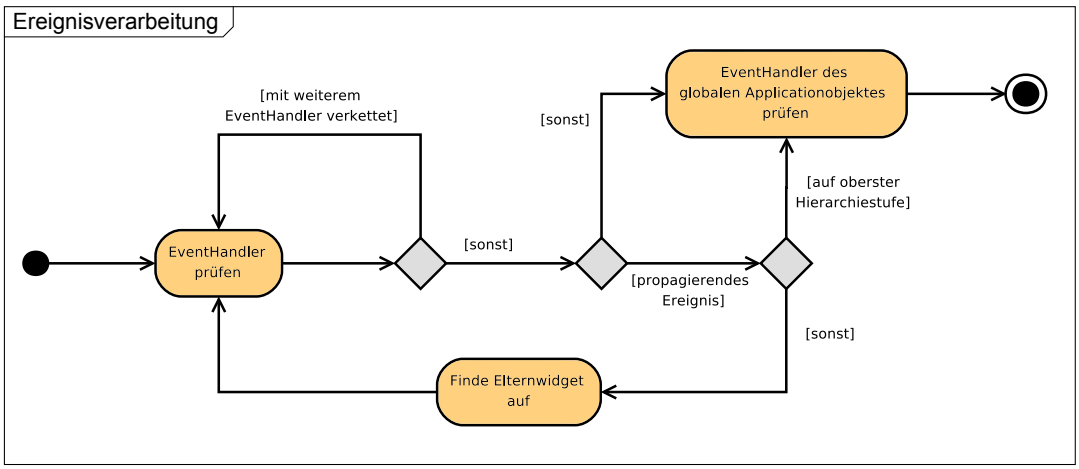


Abbildung 6: Aktivitätsdiagramm zur Eventverarbeitung in wxWidgets [1, S. 614]

2.2.2 Netzwerkprogrammierung mit wxWidgets

Wie bereits angesprochen, ist wxWidgets mehr als eine reine GUI-Bibliothek und bietet noch wesentlich mehr Funktionen. So ist auch eine plattformunabhängige Programmierung von Netzwerkanwendungen mit wxWidgets möglich. Die Bibliothek stellt für diesen Zweck verschiedene Klassen bereit, die zum einen das reine Senden und Empfangen von Datenpaketen über das TCP- und UDP-Protokoll ermöglichen. Zum anderen bietet wxWidgets jedoch auch darauf aufbauende Klassen, die ganze HTTP- oder FTP-Dienste bereitstellen [1, S. 463ff., 479]. Die Ausführungen dieser Arbeit beschäftigen sich jedoch nur mit den Basis-TCP-Verbindungen der Serverseite, wie sie auch in der späteren Realisierung des GKSwx-Treibers benötigt werden. Die Bibliothek unterstützt zwei Arten der Netzwerkprogrammierung [1, S. 464]:

- **die klassische thread-basierte Lösung** (Abb. 7a auf S. 10). Ein *Socket*serverobjekt[8] lauscht in einem eigenen Thread auf neue ankommende Verbindungen und erzeugt neue Threads, sobald sich Clienten mit dem Server verbinden wollen. Diese neu erstellten Threads bearbeiten dann die Anfrage des Clienten und beenden sich selber, sobald die Verbindung geschlossen wurde.

- **die ereignisbasierte Netzwerkprogrammierung** (Abb. 7b auf S. 11). In diesem Fall wird die gesamte Netzwerkkommunikation über das Ereignissystem von wxWidgets abgewickelt. Der *Socket*server meldet sich bei einem beliebigen *Eventhandler* an, der informiert wird, sobald eine Verbindungsanfrage am Server anliegt. Dieser kann dann mit einer zu implementierenden *Callback*-Funktion auf die Verbindungsanfrage reagieren, in der die Verbindung akzeptiert und ein neues

[8] Ein *Socket* stellt einen vom Betriebssystem bereitgestellten Kommunikationsendpunkt dar, der in Programmen verwendet werden kann, um über ein Netzwerk Daten mit anderen Programmen auszutauschen. Ein *Socketserver* ist ebenfalls ein *Socket*, der allerdings nicht den Datenaustausch, sondern die Verbindungsannahme regelt.

*Socket*objekt erstellt wird, welches die Verbindung zum Clienten repräsentiert. Dieses wird wiederum bei einem *Eventhandler* angemeldet, der dann entsprechend auf Ereignisse, wie „Daten erhalten“ oder „Verbindung geschlossen“ reagieren kann.

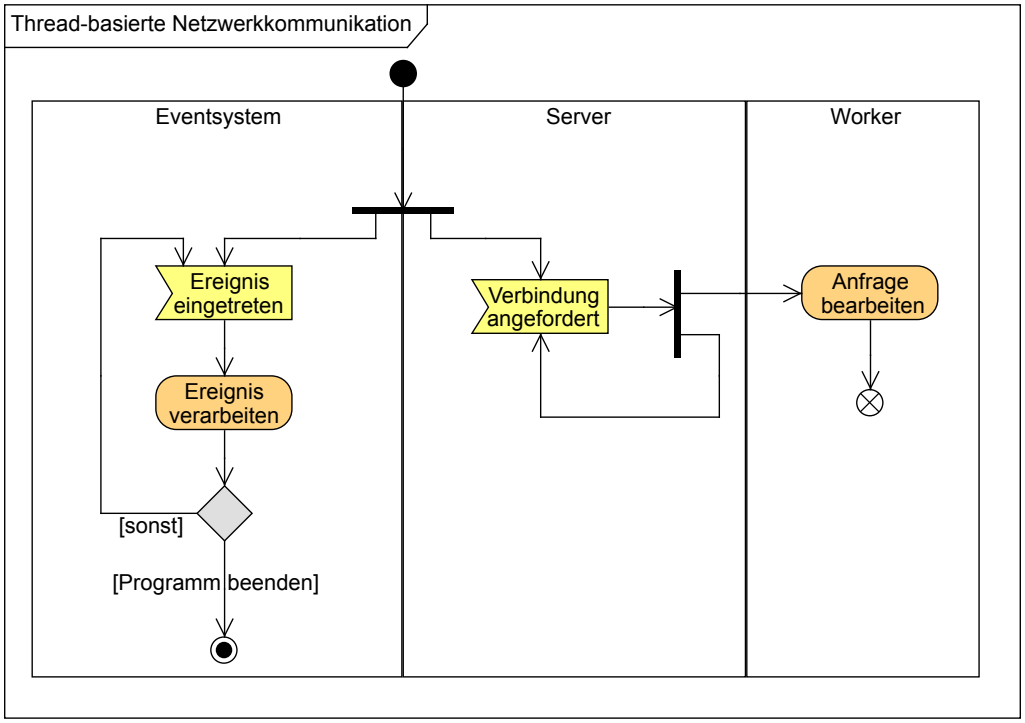


Abbildung 7a: Threadprogrammierung im Netzwerkbereich mit wxWidgets

Die ereignisbasierte Variante kommt vollständig ohne zusätzliche Threads aus, arbeitet rein sequentiell und vermeidet somit im Gegensatz zur thread-basierten Lösung, dass kritische Bereiche für gemeinsame Daten oder Synchronisierungen eingerichtet werden müssten. Außerdem sind so keine *Deadlocks*[9] möglich.
Trotz der dargestellten Vorteile ist auch die Verwendung eines thread-basierten Ansatzes berechtigt, da aufgrund der sequentiellen Ausführung der ereignisbasierten Variante andere Problemfelder entstehen, von der die thread-basierte Programmierung nicht betroffen ist. Um dies vollständig nachvollziehen zu können, sind jedoch noch weitere Grundlagen nötig.
Mit wxWidgets lässt sich über Attribute genau einstellen, ob die Socketkommunikation *blockierend* oder *nicht blockierend* sein soll. *Blockierend* bedeutet klassischerweise in diesem Zusammenhang, dass der sequentielle Ablauf des Programms mit dem Aufruf einer Kommunikationsmethode wie *Read* solange unterbrochen wird, bis die entsprechende Operation (gesamte Datenmenge lesen) vollständig abgeschlossen wurde. *Nicht*

[9] Deadlocks (oder sog. Verklemmungen) entstehen, wenn zwei Threads aufeinander warten und somit das Programm in einen unendlichen Wartezustand gerät.

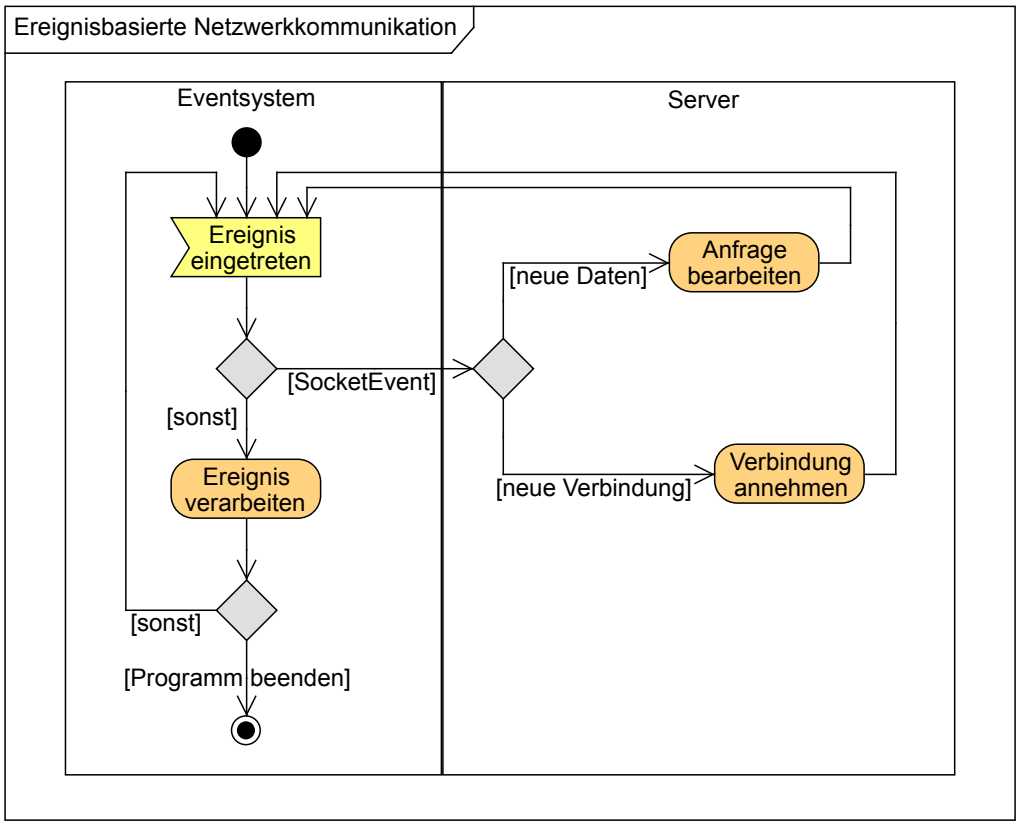


Abbildung 7b: Ereignisprogrammierung im Netzwerkbereich mit wxWidgets

blockierend bedeutet hingegen, dass nur die Daten verarbeitet werden, die bereits im Empfangspuffer der Verbindung warten und die Funktion nachfolgend direkt wieder zur aufrufenden Umgebung zurückkehrt, um den flüssigen sequentiellen Ablauf nicht zu unterbrechen [1, S. 474].
Im Zusammenhang mit wxWidgets fällt die Definition jedoch etwas anders aus, denn es existieren folgende Attribute für eine Socketverbindung [1, S. 473ff.]:

Waitall Socketoperationen sind blockierend und unterbrechen den sequentiellen Ablauf der *Eventhandler*-Routine. Andere Ereignisse der grafischen Oberfläche werden dennoch weiterhin verarbeitet, sodass die GUI bedienbar bleibt. Dies wird realisiert, indem ständig zwischen der Socketoperation und der Verarbeitung ankommender Ereignisse hin und her gesprungen wird, was jedoch zu einer hohen CPU-Auslastung führt.

Nowait Socketoperationen sind nicht blockierend und kehren sofort zurück, auch wenn gar keine Daten verarbeitet werden konnten.

None Diese Variante stellt eine Mischung aus den beiden oben genannten dar, da

zumindest ein Teil der Daten verarbeitet wird, bevor der Aufruf zurückkehrt[10].

Block *Block* kann mit den oben genannten Attributen kombiniert werden und bewirkt, dass Socketoperationen tatsächlich am Stück ausgeführt werden, ohne dass sie von anderen Ereignisverarbeitungen unterbrochen würden. Dies führt zu einer geringen CPU-Auslastung, aber sorgt dafür, dass die GUI für den Zeitraum der Socketoperation nicht mehr bedient werden kann.

Möchte man die Netzwerkroutinen in wxWidgets ausschließlich über das *Eventhandling*-System realisieren, so ist es nicht möglich alle Daten am Stück vollständig zu lesen und gleichzeitig die Oberfläche nicht zu blockieren bzw. die CPU-Auslastung gering zu halten. Baut man alternativ auf einer Umsetzung mit Threads auf, so muss auf keinen der Punkte verzichtet werden [1, S. 475]. Es hängt also vom Anwendungszweck ab, ob Threads verwendet werden müssen oder ob auch die ereignisgesteuerte Variante einsetzbar und evtl. aufgrund der zuvor genannten Vorteile vorzuziehen ist.

2.2.3 Zeichnen mit wxWidgets

Die wxWidgets-Bibliothek bietet umfangreiche Möglichkeiten um auf *Widgets* mit verschiedensten grafischen Konstrukten zu zeichnen [1, S. 131]. In den meisten Fällen ist es erwünscht Grafiken auf einer leeren Fläche zu erstellen (so als würde man mit einem weißen Blatt Papier beginnen). Für diese Zwecke kann die Klasse `wxPanel` genutzt werden, die direkt nach der Instanziierung nur einen leeren Platzhalter beliebiger Größe darstellt[11].
Das Zeichnen der Grafiken selber wird in der wxWidgets-Bibliothek durch das Konzept der *Device Contexte* (entsprechende Klasse: `wxDC`) gekapselt. Die Klasse `wxDC` stellt eine abstrakte Basis dar, die vorgibt, welche Funktionalitäten ein *Device Context* leisten muss [1, S. 131]. Neben verschiedenen Routinen zum Zeichnen und Setzen von Attributen beinhaltet ein *Device Context* Informationen über ein logisches Koordinatensystem, auf welches sich alle angebotenen Grafikroutinen beziehen. Die Transformation auf das Zielkoordinatensystem (z.B. des Computerbildschirms) wird durch den *Device Context* gekapselt und führt daher zu der Unabhängigkeit vom konkret genutzten Ausgabegerät [1, S. 132]. Ein zentrales Konzept der wxWidgets-Entwickler ist daher die einfache Austauschbarkeit verschiedener *Device Contexte*, sodass Code für verschiedenste grafische Ausgabegeräte wiederverwendet werden kann. Im optimalen Fall führt dies dazu, dass grafikerzeugender Code in Funktionen verpackt wird, die als Übergabeparameter einen allgemeinen *Context* (`wxDC`) erwarten, welcher jedes Objekt einer konkreten Implementierung annehmen kann. Der Programmierer hat dann vor dem Aufruf nur noch dafür Sorge zu tragen, dass das logische Koordinatensystem mit den korrekten Parametern versehen wurde.

[10] Dieser Modus entspricht genau einem blockierenden Aufruf einer C-Socketroutine mit festgelegtem Buffer.

[11] `wxPanel` ist eigentlich ein Container, der zur logischen Gruppierung von Widgets benutzt wird [1, S. 72]. Die Verwendung als Zeichenfläche ist nur ein weiterer Einsatzzweck.

Als konkrete Implementierung der DeviceContexts bietet wxWidgets schon zahlreiche vorgefertigte Klassen [1, S. 133]. Die wichtigsten seien hier kurz aufgelistet:

ClientDC Der *ClientDC* ist der Standard-DC, wenn direkt auf Widgets gezeichnet werden soll. Auf den meisten Plattformen erfolgt die Ausgabe direkt, es gibt jedoch auch Systeme, auf denen die Zeichenbefehle zunächst auf einem internen Buffer ausgeführt werden, sodass das Ergebnis der Operationen zusammenhängend auf dem Bildschirm erscheint (dies ist in der Regel unter Mac OS X der Fall).

PaintDC Dieser DC bietet dieselbe Funktionalität wie der *ClientDC*, aber ist speziell für den Einsatz in Eventroutinen des *PaintEvents* ausgelegt. Was dies genau bedeutet und warum dies notwendig ist, wird im Abschnitt 2.2.3.1 (S. 14) genauer erläutert.

MemoryDC *MemoryDCs* setzen die aufgerufenen Operationen nicht auf dem Bildschirm, sondern auf Bitmap-Grafiken[12] um. Auf diese Weise ist es zum einen möglich erzeugte Grafiken nachträglich mit Filtern zu manipulieren und dauerhaft in Dateien zu speichern und zum anderen einen Grafikbuffer auf Systemen zu implementieren, die dies nicht standardmässig intern anbieten (s. Abschnitt 2.2.3.2 auf S. 15).

GCDC *GCDC* steht für *Graphics Context Device Context* und kann genutzt werden, um einen *Graphics Context* vor andere DCs zu schalten, sodass deren Funktionsumfang erweitert wird [2]. Im Rahmen dieser Seminararbeit wird diese Klasse noch im Zusammenhang mit *Antialiasing* vorgestellt.

Die Basisklasse *wxDC* erzwingt die Implementierung zahlreicher Grafikroutinen. Für diese Seminararbeit sind die folgenden relevant [1, S. 148ff.][2]:

DrawPoint Zeichnet einen Punkt an die angegebene Position. Zu beachten ist allerdings, dass ein Punkt nicht auf allen Systemen einem Pixel entspricht, selbst wenn das logische Koordinatensystem mit dem Gerätekoordinatensystem identisch ist.

DrawLines Zeichnet einen Linienzug basierend auf einer angegebenen Punktliste.

DrawPolygon Erstellt wie *DrawLines* einen Linienzug und verbindet zusätzlich den ersten und den letzten Punkt miteinander. Außerdem kann die so entstandene geschlossene Fläche optional mit einer Farbe oder Schraffur gefüllt werden.

DrawEllipticArc Zeichnet den spezifizierten Teil einer Ellipse.

[12] Bitmap-Grafiken sind rechteckige Rastergrafiken, die typischerweise zu jedem Pixel die Farbinformationen im RGB-Format speichern.

DrawText Stellt den angegebenen Text mit der zuvor eingestellten Schriftart dar.

DrawRotatedText Wie *DrawText*, nur dass zusätzlich ein Drehwinkel angegeben wird, um den der zu erzeugende Text rotiert wird.

DrawBitmap Zeichnet eine Rastergrafik an eine bestimmte Position. Diese Methode wird benötigt, um *Double Buffering* zu realisieren.

Wie in Grafikprogrammen kann der zu verwendende „Stift“ und der „Pinsel“ eingestellt werden, sodass Linienstärke, -musterung und Vorder- bzw. Hintergrundfarbe variiert werden können [1, S. 142ff.].

2.2.3.1 Integration von Grafikroutinen in die Eventqueue

Die Fensterverwaltungssysteme der verschiedenen Plattformen generieren unter bestimmten Bedingungen (beispielsweise wenn ein Fenster von einem anderen Fenster verdeckt und anschließend in den Vordergrund geholt wird) *PaintEvents*, die die Anwendung veranlassen den Fensterinhalt neuzuzeichnen und somit die Anzeige aufzufrischen[13] [1, S. 41]. Möchte man nun die Grafik eines *Widgets* mittels eines *Device Contexts* modifizieren, so muss daher der dafür notwendige Code in diese Auffrischung der Anwendung integriert werden. Wie bereits im Abschnitt 2.2.1 (S. 6) über wxWidgets-*Events* beschrieben, kann hierzu eine *Callback*-Routine registriert werden, welche ausgeführt wird, sobald das zu modifizierende *Widget* von einem *PaintEvent* erreicht wird. Diese Routine beinhaltet dann den Grafikcode des Programmierers. Zu beachten ist, dass innerhalb einer solchen Funktion immer ein *PaintDC* verwendet werden muss. *ClientDCs* sind nur außerhalb des Auffrischungsvorgangs erlaubt. Dies hat zwei Gründe [1, S. 135ff.]:

1. Die Erstellung eines *PaintDCs* teilt dem globalen *Application*-Objekt mit, dass die Routine im Folgenden den gewünschten Grafikcode auf dem *Widget* ausführt und danach die Auffrischung für dieses *Widget* beendet ist. Würde kein *PaintDC* erstellt, so würde das Neuzeichnen als nicht abgeschlossen angesehen werden und es würden immer weiter *PaintEvents* generiert.

2. Um die Auffrischung zu beschleunigen, wird analysiert, welche Anwendungbereiche seit der letzten Aktualisierung überhaupt modifiziert wurden (beispielsweise indem ein anderes Fenster auf die Anwendung bewegt wurde), sodass nur die veränderten Ausschnitte auf den neuesten Stand gebracht werden. Diese Information ist automatisch in *PaintDCs* integriert und wird auch vollautomatisch verwendet.

[13] Diese Aktualisierung wird nur durchgeführt, wenn sie notwendig ist und ist nicht mit regelmäßigen Bilderneuerung zu verwechseln, die beispielsweise bei Röhrenmonitoren notwendig ist, um die Bilddarstellung vorzunehmen.

2.2.3.2 Double Buffering

Zeichnet man direkt auf ein Widget unter Verwendung eines *ClientDCs* oder eines *PaintDCs*, so kann es zu ungewollten Flimmereffekten kommen, da das Bild Stück für Stück mit den einzelnen Grafikbefehlen aufgebaut wird. Dieser Effekt wird noch dadurch verstärkt, dass in wxWidgets standardmäßig immer zunächst das gesamte Widget mit einer Hintergrundfarbe ausgefüllt wird, bevor die eigentliche Grafikroutine aktiv wird [1, S. 134f.].
Aus diesem Grund verwendet man neben dem Grafikbuffer, der sowieso zur Anzeige der grafischen Oberfläche benötigt wird, einen zusätzlichen Buffer, auf dem zunächst alle Grafikoperationen ausgeführt werden, die später auf dem Bildschirm erscheinen sollen. Ist die Grafik im Buffer vollständig erstellt, so wird sie auf das *Widget* übertragen. Da so der schrittweise Aufbau der Anzeige vermieden wird, können keine Flimmereffekte mehr auftreten. Aufgrund der Verwendung von zwei Buffern wird diese Technik auch *Double Buffering* genannt [1, S. 136][12]. Dies kann in wxWidgets beispielsweise damit realisiert werden, dass alle Befehle zunächst intern auf einem *MemoryDC* ausgeführt werden, dessen Inhalt dann abschließend auf den *PaintDC* übertragen wird.
Wie eingangs erwähnt, weist wxWidgets die Besonderheit auf, dass das Löschen des Hintergrundes und das Zeichnen der Vordergrundgrafik getrennt durchgeführt werden. Der Kerngedanke des *Double Bufferings* ist es allerdings, die Grafik mit einer einzigen Operation auf der Anzeige sichtbar zu machen, da ansonsten keine flimmerfreie Änderung der Anzeige möglich ist. Deshalb muss in wxWidgets zusätzlich das Löschen des Hintergrundes deaktiviert werden, um *Double Buffering* korrekt realisieren zu können[1, S. 134].

2.2.3.3 Antialiasing

Computermonitore basieren auf einem Raster von quadratischen Pixeln mit begrenzter Auflösung, sodass Grafikteile, die auf den Pixelgrenzen liegen, einer Rundung unterzogen werden müssen und deshalb verfälscht wiedergegeben werden. Visuell fällt dies besonders bei feinen kontrastreichen Strukturen (wie beispielsweise Texten) ins Gewicht, bei denen oft deutliche Artefakte in Form von *Quantisierungseffekten*(„Treppenstufenbildung“) zu erkennen sind. Durch *Quantisierungseffekte* werden Linien, die als Geradenstück dargestellt werden sollten, durch Pixelanordnungen in Form einer Treppe visualiert, da eine kontinuierliche Darstellung der Linie in einem diskreten Gitter nicht möglich ist.
Auf der Basis des zuvor beschriebenen Problems wurden zahlreiche Algorithmen entwickelt, um eine sog. *Kantenglättung* (im Englischen *Antialiasing*) durchzuführen. Ziel der Algorithmen ist es, die harten Kontraste an Kanten von Grafiken abzuschwächen, sodass auch die sichtbaren Treppenstufenartefakte minimiert werden [13].
Je nach verwendeter Plattform können die *Device Contexts* schon eine automatische Kantenglättung enthalten. Auf den meisten Systemen ist dies nicht der Fall, jedoch liefert wxWidgets für diesen Zweck den *GCDC*, der vor einen beliebigen anderen *DC* geschaltet werden kann und die Möglichkeit bietet, gezielt *Antialising* zu aktivieren oder auch zu deaktivieren, wenn standardmäßig eine automatische Kantenglättung durch-

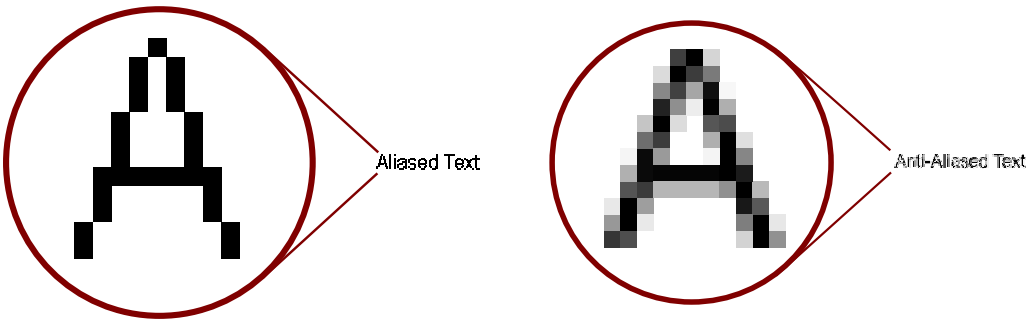


Abbildung 8: Vergleich von ungeglättetem und geglättetem Text

geführt wird (wie auf Mac OS X) [2].

	wxMSW	*wxGTK*	*wxMac*
automatisches *Double Buffering*	✗	✗	✓
automatisches *Antialiasing*	✗	✗	✓

Abbildung 9: Vergleich verschiedener *wxWidgets Ports* im Hinblick auf *Antialiasing* und *Double Buffering*

3 Realisierung

Das folgende Kapitel erläutert, wie der zu implementierende Gerätetreiber auf Basis der zuvor vorgestellten Grundlagen entwickelt wurde.

3.1 Vorgaben für die Entwicklung

Die Entwicklung des GKS-Gerätetreibers für wxWidgets war an einige Vorgaben gebunden. So sollte die fertige Implementierung prinzipiell sowohl als gewöhnliche GUI-Komponente (*Widget*) als auch als Stand-Alone-Anwendung nutzbar sein. Die Stand-Alone-Fassung des Programms erhält dabei die GKS-Displayliste über eine TCP-Verbindung und visualisiert die Daten anschließend in einem eigenständigen Fenster.
Um das Programm als Widget ohne Netzwerkkommunikation nutzen zu können, muss ein Plug-In[14] geschrieben werden, welches aus dem GKS heraus die passenden Grafikbefehle auf dem Widget aufruft. Die zusätzliche Entwicklung des Plug-Ins geht im Rahmen dieser Seminararbeit jedoch zu weit. Ziel der Arbeit ist es aber das *Widget* so aufzubauen, dass zu einem späteren Zeitpunkt die Plug-In-Funktionalität ohne große Änderungen eingebracht werden kann.
Die resultierende Grafikanzeige soll – wenn gewünscht – mit einem Kantenglättungsalgorithmus gefiltert werden können.
Als Orientierungshilfe und als Unterstützung im Umgang mit der GKS-Schnittstelle diente ein bereits vorliegender GKS-Gerätetreiber für das Qt4-Framework.

3.2 Softwarevoraussetzungen

Die wxWidgets-Klassenbibliothek stellt bereits Funktionalitäten zur Kantenglättung bereit, die jedoch erst in der aktuellen Entwicklungsversion 2.9.x vollständig zur Verfügung stehen [2]. Diese bildet daher zusammen mit der aktuellen GLIGKS Version 5 die Basis der im Folgenden beschriebenen Implementierung.

3.3 Interner Aufbau

Die GKSwx-Implementierung (s. Abb. 10) besteht aus vier Hauptklassen, die jeweils logisch zusammengehörige Funktionalitäten kapseln. Die Klassen werden hier im Folgenden vorgestellt:

GKSApp Die Implementierung der *Application*-Klasse ist zwingend von wxWidgets vorgegeben, wenn eine eigenständige grafische Benutzeroberfläche programmiert werden soll. Sie verwaltet die Instanziierung des Hauptfensters und beinhaltet die *Mainloop*, in welcher die schon beschriebene Verarbeitung der ausgelösten Ereignisse vorgenommen wird. wxWidgets

[14] *Plug-Ins* sind Programmteile, die die Funktionalität eines anderen Programms erweitern.

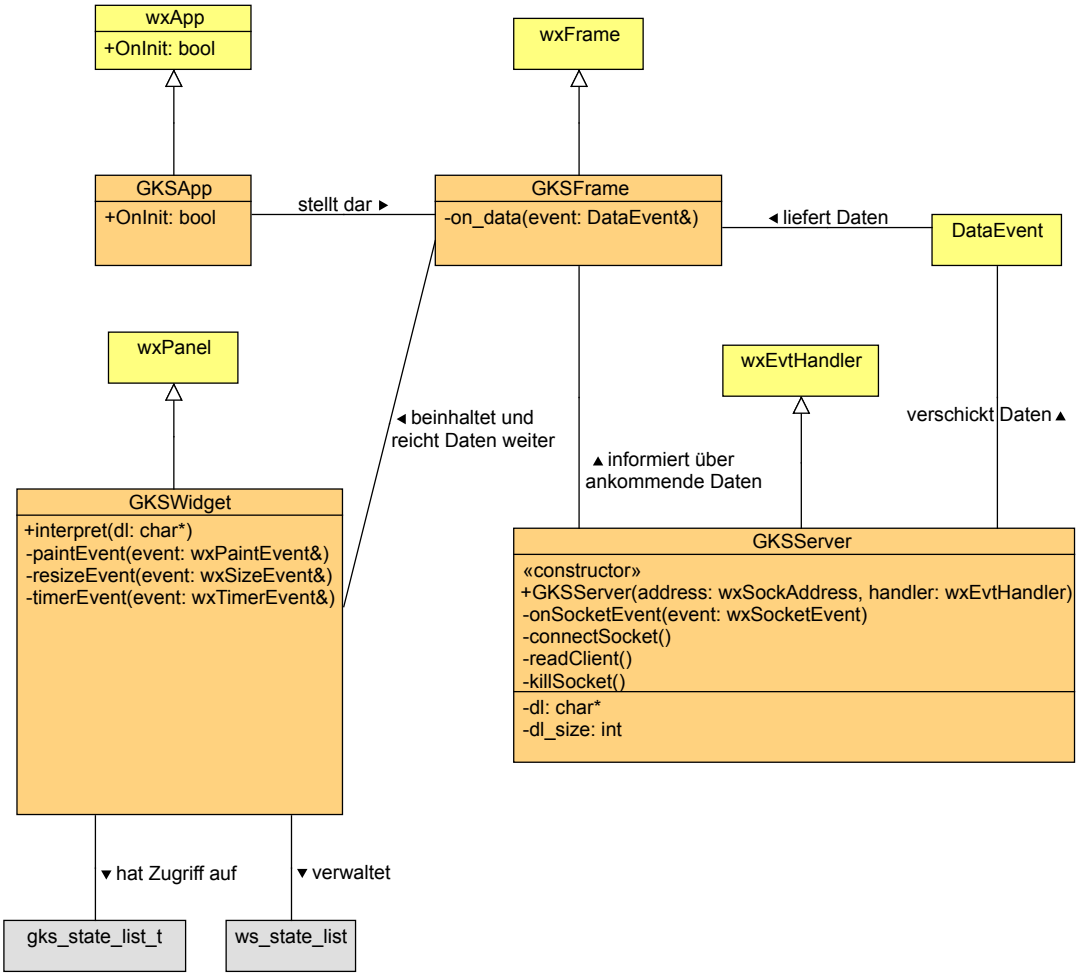


Abbildung 10: Interner Aufbau von GKSwx

liefert bereits eine Vorlage der *Application*-Klasse, in der durch Spezialisierung der Methode `OnInit` nur noch der Code zur Instantiierung des Hauptfensters integriert werden muss.

Per

```
IMPLEMENT_APP(GKSApp)
```

kann wxWidgets dann mitgeteilt werden, dass aus der Klasse `GKSApp` das globale Anwendungsobjekt erstellt werden soll. `IMPLEMENT_APP` ist ein codeerzeugendes Makro, welches auch die passende `main`-Funktion liefert, um die gesamte Anwendung zu starten, sodass keine eigene `main`-Funktion codiert werden muss.

GKSFrame Die `GKSFrame`-Klasse liefert das Hauptfenster für die Stand-Alone-Fassung des Programms. Es dient als Container für das `GKSWidget` und in-

formiert das *Widget*, sobald der `GKSServer` neue Daten liefert. Hierzu wird auf dem `GKSWidget`-Objekt die öffentliche Methode

```
void interpret(const char *display_list)
```

aufgerufen, welche die GKS-Displayliste in Form eines `char`-Arrays erwartet.

GKSServer Der `GKSServer` kapselt die gesamte Netzwerkfunktionalität der Anwendung. Da die Stand-Alone-Anwendung ausschließlich eine Visualisierungsfläche in Form des `GKSWidgets` anzeigen soll, ist es in diesem Szenario nicht notwendig, dass die grafische Oberfläche möglichst kurze Antwortzeiten liefert. Viel wichtiger ist hingegen, dass nach dem Einlesen der Displayliste möglichst schnell die entsprechende grafische Darstellung der übertragenen Direktiven geliefert wird. Zur Realisierung dieser sequentiellen Reihenfolge ist daher der Einsatz einer parallelen thread-basierten Netzwerkkommunikation nicht gerechtfertigt. Stattdessen wird die bereits vorgestellte ereignisbasierte Kommunikation verwendet, die vollständig ohne den Einsatz weiterer Threads arbeitet. Da die Benutzeroberfläche für die (im Forschungszentrums-Netzwerk kurze) Zeit des Datenempfangs nicht reaktiv sein muss, kann die Einlese-Operation des *Sockets* auf `WAITALL` und `BLOCK` gestellt werden, sodass für die Zeit des Einlesens zwar keine GUI-Ereignisse verarbeitet werden, es jedoch möglich ist, alle ankommenden Daten mit einer Operation zu empfangen und zeitgleich die CPU-Auslastung möglichst niedrig zu halten. Die eingelesenen Daten werden anschließend an das `GKSFrame` in Form eines benutzerdefinierten Ereignisses (`DataEvent`) gesendet. Wie dies genau funktioniert, wird noch an späterer Stelle genau erläutert.

GKSWidget Das `GKSWidget` ist die Kernkomponente zur Visualisierung der GKS-Direktiven. Es erhält die GKS-Displayliste in Form eines `char`-Arrays, interpretiert die enthaltenen Sublisten dieses Puffers nacheinander und führt sofort für jede Subliste die notwendigen Grafikbefehle zur Darstellung der enthaltenen Direktive aus. Es wurde so konzipiert, dass möglichst wenige direkte Abhängigkeiten zu den anderen vorgestellten Klassen bestehen, um das Widget später einfach aus dem Klassenverbund herauslösen und als eigenständige Komponente für den Plug-In-Einsatz verwenden zu können. Eine direkte Abhängigkeit zu der Serverklasse wäre beispielsweise nicht sinnvoll, da das eigenständige *Widget* später ohne Netzwerkkommunikation arbeiten wird.

Der Status des *Widgets* wird in Form eines *structs* (`ws_state_list`) verwaltet, dessen Aufbau der in dem Grundlagenabschnitt 2.1.2 (S. 3) genannten *Workstation Description Table* entspricht. Der interne Status des GKS wird zusätzlich mit einer Variable des *structs* `gks_state_list_t` festgehalten.

3.4 Ereignisdeklaration

Wie bereits angesprochen wird die Datenkommunikation vom `GKSServer` zum `GKSFrame` mit Hilfe eines selbst definierten Ereignisses vorgenommen, welches allgemein das Versenden eines beliebigen Datenpuffers ermöglichen soll. Wie die Implementierung dieses *Events* realisiert wird, wird in folgendem Abschnitt beschrieben [1, S. 34ff.][4][5].
Alle von wxWidgets verwendeten Ereignisse sind von der gemeinsamen Basisklasse `wxEvent` abgeleitet, um den polymorphen Charakter der Objektorientierung in C++ nutzen zu können. Alle selbst implementierten Ereignisse sind somit auch von der Klasse `wxEvent` abzuleiten:

```
1  class DataEvent : public wxEvent {
2
3  public:
4    DataEvent(wxObject* obj = (wxObject*) NULL, void* data =
       (void*) NULL, int data_len = 0) {
5      CreateEvent(obj, data, data_len);
6    }
7
8    DataEvent(const DataEvent &event) {
9      CreateEvent(event.GetEventObject(), event.data_,
         event.data_len_);
10   }
11
12   wxEvent* Clone() const {
13     return new DataEvent(*this);
14   }
15
16 private:
17   void CreateEvent(wxObject* win = (wxObject*) NULL, void*
       data = (void*) NULL, int data_len = 0) {
18     data_len_ = data_len;
19     data_ = new char[data_len+1];
20
21     SetEventType(EVT_DATA);
22     SetEventObject(obj);
23
24     memcpy(data_, data, data_len_+1);
25   }
26
27   DECLARE_DYNAMIC_CLASS(DataEvent)    // in der Header-Datei
28   .
29   .
30   .
31 };
32
33 IMPLEMENT_DYNAMIC_CLASS(DataEvent, wxEvent)    // in der cpp
```

Der standardmäßig zu verwendene Konstruktor (Zeile 4) erwartet den Absender des Ereignisses von einem beliebigen Typ der wxWidgets-Klassenbibliothek und einen beliebigen Datenpuffer, dessen Länge ebenfalls spezifiziert werden muss, da diese Information in dem `data`-Zeiger nicht gespeichert werden kann.
Da die vorhandenen Konstruktoren beide auf dieselbe Weise das Objekt initialisieren, ist der eigentliche Konstruktorcode in die private Methode `CreateEvent` (17) ausgelagert worden, in der neben der Speicherung des Datenpuffers die Ereignisquelle (22) und der Ereignistyp (21) festgelegt werden. `EVT_DATA` musste zuvor natürlich auch deklariert und in das Eventsystem von wxWidgets integriert werden:

```
BEGIN_DECLARE_EVENT_TYPES()             // h-Datei
  DECLARE_EVENT_TYPE(EVT_DATA, 1)       // h-Datei
END_DECLARE_EVENT_TYPES()               // h-Datei

DEFINE_EVENT_TYPE(EVT_DATA)             // cpp-Datei
```

Damit auch eine *Callback*-Routine angegeben werden kann, die im Falle des Eintreffens eines `DataEvents` ausgeführt wird, muss zunächst ein korrespondierender Funktionstyp der *Callback*-Routine definiert werden,

```
typedef void (wxEvtHandler::*DataEventFunction)(DataEvent&);
```

sodass Funktionen dieses Typs im Eventsystem von wxWidgets registriert werden können. Klassischerweise wird dies in wxWidgets über statische Makro-basierte Ereignistabellen realisiert. Um diese Art der Funktionsangabe zu ermöglichen, muss ein neuer Tabelleneintrag definiert werden:

```
#define EVT_DATA(func)                                          \
  DECLARE_EVENT_TABLE_ENTRY\
    ( EVT_DATA,                                                 \
      -1,                                                       \
      -1,                                                       \
      (wxObjectEventFunction) (DataEventFunction) & func,       \
      (wxObject *) NULL ),
```

In aktuellen wxWidgets-Versionen kann jedoch auch die `Connect`-Methode der *EventHandler* genutzt werden, um *Callback*-Routinen anzumelden. Um solche Registrierungen zuzulassen, ist zusätzlicher Code notwendig, da `Connect` den Funktionszeiger auf eine *Callback*-Routine als Typ einer *EventHandlerFunction* erwartet:

```
#define DataEventHandler(func) \
  (wxObjectEventFunction)      \
  (wxEventFunction)            \
  (wxCommandEventFunction)     \
   wxStaticCastEvent(DataEventFunction, &func)
```

wxWidgets nutzt intern ein eigenes Typüberprüfungssystem, um zur Laufzeit effizient dynamische Objekttypen bestimmen zu können. Um die neue Ereignisklasse mit diesem System nutzen zu können, muss sie zunächst als dynamische Klasse deklariert (27) und anschließend implementiert (33) werden.

Schließlich fordert die wxWidgets-Bibliothek die Implementierung einer `Clone`-Methode (12), die intern bei der Ereignisverarbeitung an Stelle des Copy-Konstruktors verwendet wird. Da die Bedeutung der `Clone`-Methode bei diesem benutzerdefinierten Ereignis jedoch identisch mit der des Copy-Kontruktors (8) ist, wird dieser kurzerhand in der `Clone`-Methode wiederverwendet.

3.5 Konzept des Netzwerkmoduls

Wie bereits beschrieben arbeitet der `GKSServer` rein ereignisbasiert. Dieser Unterabschnitt erläutert, wie die Netzwerkfunktionalität implementiert und die Kommunikation zwischen `GKSServer` und `GKSFrame` hergestellt wurde.

```
1  #define SERVER_ID 1000
2  #define SOCKET_ID 1001
3
4  GKSServer::GKSServer(const wxSockAddress& address,
       wxEvtHandler& handler)
5    :
6    event_handler(handler),
7    socket_server(new wxSocketServer(address)) {
8
9    socket_server->SetEventHandler(*this, SERVER_ID);
10   socket_server->SetNotify(wxSOCKET_CONNECTION_FLAG);
11   socket_server->Notify(true);
12
13   Connect(SERVER_ID, wxEVT_SOCKET,
         wxSocketEventHandler(GKSServer::onSocketEvent));
14   Connect(SOCKET_ID, wxEVT_SOCKET,
         wxSocketEventHandler(GKSServer::onSocketEvent));
15 }
```

Bei der Erstellung des `GKSServers` wird ein `wxSocketServer` mit der angegebenen IPv4-`address` instanziiert, der neue TCP-Verbindungen annehmen kann. Wird eine neue Verbindunganfrage des GKS an den `SocketServer` gesendet, so kann ein beliebiger `EventHandler` über dieses Ereignis informiert werden. Daher muss zunächst der zu informierende `EventHandler` spezifiziert (9) und die zu übermittelnden Ereignisse festgelegt werden (10). Um diese Ereignisse intern verarbeiten zu können, ist der `GKSServer` als Unterklasse des `wxEvtHandlers` realisiert, sodass das `GKSServer`-Objekt selbst (`*this`) als *Handler* verwendet werden kann. Mit Hilfe von

```
socket_server->Notify(true);
```

wird der gesamte Mechanismus schließlich gestartet.
Um die empfangenen Ereignisse auch tatsächlich verarbeiten zu können, muss nun nur noch eine geeignete Callback-Routine registriert werden, die sowohl für den `Socket-Server` (13) als auch für die resultierenden Verbindungen (14) zuständig ist.

Der beim Konstruktor noch zusätzlich übergebene `handler` (in dem Gesamtsystem das `GKSFrame`) (6) wird an späterer Stelle noch als Zieladresse der abzuschickenden `DataEvents` benötigt.

```
1  void GKSServer::onSocketEvent(wxSocketEvent &event) {
2
3      switch(event.GetSocketEvent()) {
4      case wxSOCKET_CONNECTION:
5        connectSocket();
6        break;
7      case wxSOCKET_INPUT:
8          readClient();
9          break;
10     case wxSOCKET_LOST:
11         killSocket();
12         break;
13     default:
14         break;
15     }
16 }
```

Innerhalb der Callback-Routine werden dann die verschiedenen Ereignistypen unterschieden. Unmittelbar nach Erstellung des `GKSServers` ist nur ein `SOCKET_CONNECTION`-Ereignis möglich, welches mittels `connectSocket` behandelt wird:

```
1  void GKSServer::connectSocket() {
2
3    if(s != 0)
4        killSocket();
5
6    s = socket_server->Accept();
7    s->SetFlags(wxSOCKET_WAITALL | wxSOCKET_BLOCK);
8    s->SetEventHandler(*this, SOCKET_ID);
9    s->SetNotify(wxSOCKET_INPUT_FLAG | wxSOCKET_LOST_FLAG);
10   s->Notify(true);
11
12 }
```

Sollte bereits eine Verbindung bestehen, so wird diese zunächst geschlossen (3 und 4), da immer nur eine Verbindung zur selben Zeit behandelt werden soll. Würden mehrere Verbindungen zugelassen, so könnten die ankommenden GKS-Direktiven vermischt und die Ausgabe würde unbrauchbar werden.
Nach der Verbindungsannahme (6) wird der resultierende *Socket* so eingestellt, dass der spezifizierte Lesepuffer vollständig mit ankommenden Daten gefüllt wird (`WAITALL`) ohne weiterhin weitere GUI-Ereignisse zu verarbeiten (`BLOCK`), sodass die Systemlast niedrig gehalten wird (7). Wie im Konstruktor des `GKSServers` wird anschließend der Server als *Eventhandler* festgelegt, der sowohl über ankommende Daten, als auch über geschlossene Verbindungen informiert werden soll (8 bis 10).

Kommen über die angenommene Verbindung nun Daten in Form einer Displayliste an (es wird implizit angenommen, dass ausschließlich GKS-Displaylisten an den Server gesendet werden), so wird über die allgemeine *Callback*-Routine `onSocketEvent` die Methode `readClient` aufgerufen, welche die Daten des Clienten der Gegenseite in Empfang nimmt.

```
1  void GKSServer::readClient() {
2
3      int char_count;
4
5      s->Read(reinterpret_cast<void *>(&char_count),
          sizeof(char_count));
6      if(s->Error()) sendError(1);
7
8      if(char_count > 0) {
9          if(display_list != 0) {
10             delete[] display_list;
11         }
12         distplay_list = new char[char_count+sizeof(int)];
13         s->Read(reinterpret_cast<void *>(display_list),
              char_count);
14         if(s->Error()) sendError(2);
15         *reinterpret_cast<int*>(display_list+char_count) = 0;
16
17         DataEvent event(this, reinterpret_cast<void
              *>(display_list), char_count);
18         event_handler.ProcessEvent(event);
19     }
20 }
```

Innerhalb dieser Funktion wird zunächst die 4 Byte lange Größe der GKS-Displayliste ausgelesen (5), sodass anschließend ein ausreichend großer Puffer für die gesamte Displayliste angelegt (9 bis 12) und diese mit einem einzigen blockierenden Befehl vollständig eingelesen werden kann (13). Weil die spätere Interpretationsroutine einen nullterminierten Puffer zur Verarbeitung erwartet, muss zwingend sichergestellt werden, dass der Puffer tatsächlich mit Nullbytes beendet wird, um im späteren Programmablauf Speicherüberläufe zu vermeiden (15). Abschließend wird der Puffer an ein `DataEvent` übergeben (17) und in die *Eventqueue* des `GKSFrames` gelegt (18). Innerhalb des `GKSFrames` wird darauf mit der Übergabe der Displayliste an das `GKSWidget` reagiert:

```
gks_widget->interpret(data);
```

Sollten während dem Empfangsvorgang Fehler auftreten, so können diese über die `Error`-Methode erfragt werden (6 und 14). Mit Hilfe der selbst implementierten Funktion `sendError` kann im Fehlerfall dann ein weiteres benutzerdefiniertes Ereignis vom Typ `ErrorEvent` an den `event_handler` gerichtet werden. Das `GKSFrame` reagiert in diesem Ausnahmefall dann mit der Beendigung der gesamten Anwendung.

Bei Verbindungsabbrüchen veranlasst `onSocketEvent` einen Aufruf der Methode `killSocket`, welche die Verbindung auf Seiten des Servers korrekt beendigt.

3.6 Dateninterpretation

Nach der Datenübergabe vom `GKSFrame` an das `GKSWidget` wird innerhalb des Widgets die private Methode

```
void GKSWidget::interp(char *data) { ... }
```

aufgerufen, welche die eigentliche Interpretation des übergegebenen Datenpuffers übernimmt. Mit Hilfe des im Quelltext definierten Makros `RESOLVE` werden dann Schritt für Schritt die einzelne Bytefolgen aus dem Puffer extrahiert und mit bestimmten Datentypen interpretiert, welche von der Byteposition und den extrahierten Befehlscodes abhängen. So wird zu Beginn zunächst die Länge und der Befehlscode der ersten Subliste bestimmt:

```
RESOLVE(len,  int, sizeof(int));
RESOLVE(code, int, sizeof(int));
```

Abhängig vom eingelesenen `code` werden dann auf dieselbe Weise die Befehlsparameter extrahiert und anschließend die passende Routine zur Darstellung der decodierten Direktive aufgerufen. Für jede Grafikdirektive existiert eine eigene Funktion, die die Übersetzung der Direktive mit den dazugehörigen Parametern auf geeignete Aufrufe der wxWidgets-Grafikroutinen übernimmt.
Dieser Vorgang wird solange wiederholt, bis alle Sublisten des Puffers verarbeitet wurden. Dies ist dann der Fall, wenn zu Beginn der Schleife die Längenangabe 0 ausgelesen wird. Genau aus diesem Grund war es im Netzwerkabschnitt so wichtig, die Abgeschlossenheit der eingelesenen Daten mit Nullbytes zu garantieren, damit bei der Interpretation keine unkontrollierten Endlosschleifen entstehen können.

3.7 Besonderheiten bei der grafischen Umsetzung der GKS-Direktiven

Die programmiertechnische Umsetzung jeder GKS-Direktive im Detail zu beschreiben, würde weit über den Rahmen dieser Seminararbeit hinausgehen. Daher werden im Folgenden Besonderheiten und Probleme aufgeführt, die während der Implementierung des Anzeigetreibers aufgetreten sind.

3.7.1 Aktualisierung der Anzeige

Wie bereits in den Grundlagen erläutert, muss der Code für die Darstellung der GKS-Befehle in die Auffrischung der Anwendung integriert werden. Es ist also nicht möglich, die Grafikbefehle direkt nach Erhalt der Daten auszuführen und dabei die Synchronisierung mit der wxWidgets-eigenen Anzeigeaktualisierung zu umgehen, da bei der danachfolgenden Anzeigeerneuerung des Systems die zuvor erzeugte grafische Anzeige der GKS-Befehle verloren gehen würde.

Um eine Integration der Grafikbefehle in die Anzeigeauffrischung zu erreichen, muss daher eine *Callback*-Routine registriert werden, die die Visualisierung des Displayliste übernimmt:

```
Connect(wxEVT_PAINT,
   wxPaintEventHandler(GKSWidget::paintEvent));
```

Im Abschnitt 2.2.3.2 (S. 15) wurde auch schon erklärt, dass zusätzlich das automatische Löschen des Hintergrundes deaktiviert werden muss, um die Aktualisierung des Widgets mit einer Operation durchführen zu können:

```
SetBackgroundStyle(wxBG_STYLE_CUSTOM);
Connect(wxEVT_ERASE_BACKGROUND,
   wxEraseEventHandler(GKSWidget::eraseBackgroundEvent));

void GKSWidget::eraseBackgroundEvent(wxEraseEvent& event) {

}
```

Die `paintEvent`-Methode hat dann folgende Gestalt:

```
1  void GKSWidget::paintEvent(wxPaintEvent& event) {
2      wxPaintDC paint_dc(this);
3
4      if (display_list) {
5          destroy_clipping_region(*gcdc, antialias);
6          gcdc->SetBackground(*wxWHITE_BRUSH);
7          gcdc->Clear();
8          interp(dl);
9          paint_dc.DrawBitmap(*bmp, wxPoint(0, 0));
10     }
11 }
```

Nach der obligatorischen Erstellung eines `paint_dc` für das `GKSWidget` (2) erfolgt die Überprüfung, ob überhaupt Daten vorliegen, die visualisiert werden können (4). Ist dies der Fall, so können die eigentlichen Grafikbefehle ausgeführt werden. Statt direkt den `paint_dc` zu verwenden, wird ein an anderer Stelle definiertes Objekt der Klasse `wxGCDC` eingesetzt, der auf einen `memory_dc` aufbaut, der schließlich die Daten eines `wxBitmap`-Objektes (`bmp`) manipuliert. So können *Antialiasing* (Abschnitt 2.2.3.3 auf S. 15) und *Double Buffering* (Abschnitt 2.2.3.2 auf S. 15) realisiert werden, indem zunächst die gesamte Bitmapfläche weiß übermalt (5 bis 7) und anschließend unter Verwendung der `antialias`-Einstellung (5) mit den grafischen Umsetzungen der Displayliste ausgefüllt wird (8). Abschließend wird das gesamte Ergebnis mit einer Operation auf das *Widget* selber übertragen (9).
Um diese Aktualisierung der Anwendung nun auch zeitnah nach Erhalt der Daten vorzunehmen, kann die von der `wxWindow`-Klasse bereitgestellt Methode

```
Refresh();
```

verwendet werden, die ein *PaintEvent* generiert und somit zum Aufruf der `paintEvent`-Methode führt. Genauere Informationen zur Anzeigeaktualisierung folgen noch im Abschnitt 3.7.8 (S. 35).

3.7.2 Zeichnen von Punkten

Erhält das Widget die Anweisung einen einzelnen Punkt zu zeichnen, so soll zur Umsetzung dieser Direktive die kleinstmögliche Bildschirmeinheit (also ein Pixel) verwendet werden. Die von wxWidgets angebotene Routine

```
void wxDC::DrawPoint(const wxPoint &pt)
```

setzt jedoch bei einigen *wxWidget ports* immer mehr als einen Pixel, selbst, wenn die Linienbreite auf einen Pixel begrenzt wird.
Um dieses Problem zu umgehen, kann alternativ die Methode

```
void wxDC::DrawLine(const wxPoint &pt1, const wxPoint &pt2)
```

genutzt werden, indem man als Start- und Endpunkt dieselbe Angabe verwendet. Auf diese Weise wird bei einer gesetzten Linienbreite von einem Pixel auch tatsächlich nur ein einzelner Pixel auf dem Bildschirm gezeichnet.

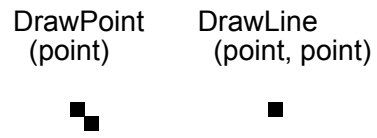


Abbildung 11: Vergleich von `DrawPoint` und `DrawLine` mit wxMac

3.7.3 Verknüpfungsarten der Einzellinien von Polylines

Sollen Linienzüge mit wxWidgets gezeichnet werden, so bietet das Grafiksystem verschiedene Möglichkeiten, wie die einzelnen Linien des Verbundes an den End- und Startpunkten verknüpft werden können (sog. *line joins*) [1, S. 143][2]. Diese können beim aktuellen „Stift“[15] per

```
void wxPen::SetJoin(wxPenJoin join_style)
```

gesetzt werden. Zu den möglichen `join_styles` zählen:

JOIN_BEVEL Verbindungsecken werden abgeflacht dargestellt.

JOIN_MITER Ecken laufen spitz zu.

JOIN_ROUND Die Verbindungspunkte des Linienzuges werden abgerundet. Dies ist der Standard von wxWidgets und wird auch im Anzeigetreiber verwendet.

[15] Die wxWidgets-Bibliothek kapselt Grafikattribute in Form einer `wxPen`- und einer `wxBrush`-Klasse. `wxPen` enthält die Eigenschaften, die zur Darstellung von Linien und Konturen eingesetzt werden, während `wxBrush` Informationen darüber enthält, wie geschlossene Flächen gefüllt werden sollen.

Für nicht geschlossene Linienzüge kann extra spezifiziert werden, wie der Start- und Endpunkt dargestellt wird (sog. *cap style*). Die zur Verfügung stehenden Stile sind:

CAP_BUTT Die Linienenden sind quadratisch geformt und starten bzw. enden genau an den spezifizierten Punkten.

CAP_PROJECTING Wie `CAP_BUTT` mit dem Unterschied, dass die Linienabschlüsse um die Hälfte der Linienbreite über die angegeben Endpunkte hinausgehen.

CAP_ROUND Am Anfang und Ende ist der Linienzug abgerundet. Der Abschluss geht wie bei `CAP_PROJECTING` um die Hälfte der Linienbreite über die angegebenen Punkte hinaus. Wie bei den *line joins* ist dies der wxWidgets-Standard und wird auch im Anzeigetreiber verwendet.

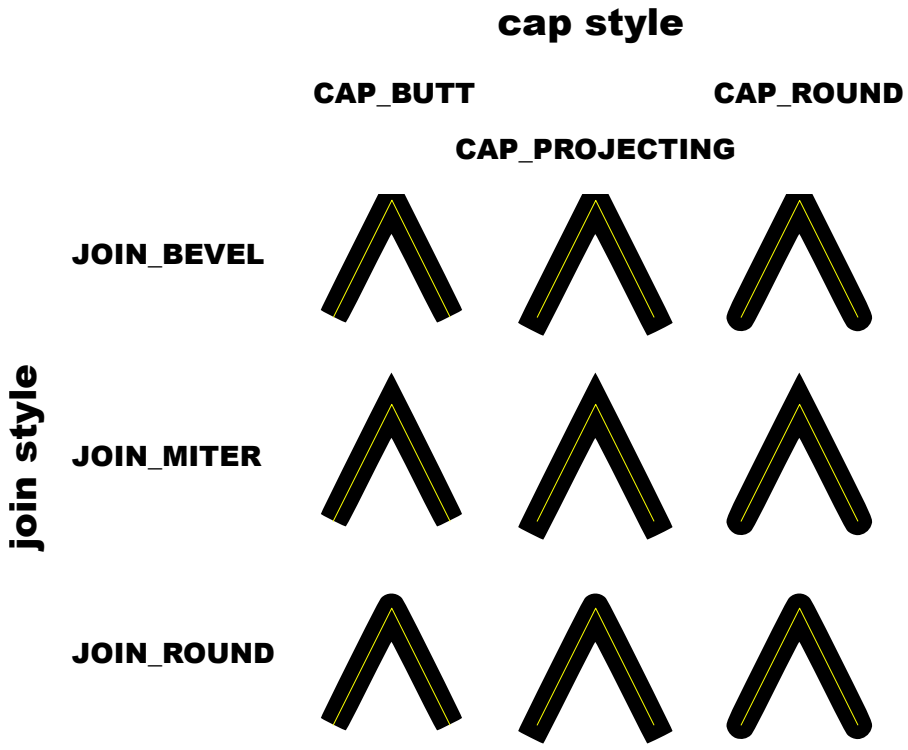


Abbildung 12: join und cap styles von wxWidgets

3.7.4 Linienmuster

Das GKS schreibt genau vor, welche Linienmuster von den Anzeigetreibern unterstützt werden müssen. wxWidgets bietet schon einige vordefinierte Linienmuster, jedoch stimmen diese nicht mit den Musterungen des GKS überein. Die GUI-Bibliothek lässt jedoch

die Angabe eines benutzerdefinierten Musters zu. Hierzu wird der Stil des verwendeten Stiftes zunächst auf benutzerdefiniert umgestellt:

```
pen.SetStyle(wxPENSTYLE_USER_DASH);
```

Anschließend kann ein Array vom Typ `wxDash[]`[16] übergeben werden, dessen Feldeinträge abwechselnd bestimmen, wie viele Längeneinheiten eines Striches gezeichnet bzw. nicht gezeichnet werden sollen. Die GKS-Bibliotheksfunktionen bieten bereits eine Routine, die genau eine solche Liste generiert und zurückliefert:

```
gks_get_dash_list(line_type, line_width, dash_list);
```

Die Funktion erwartet als Parameter das gewünschte Linienmuster (`line_type`) und die Linienbreite, da die Länge der Striche und Lücken proportional an die Breite der Linie angepasst wird. Das Ergebnis der Operation wird schließlich als Feld von Integern auf der Variablen `dash_list` abgelegt, wobei der erste Eintrag des Arrays die Gesamtanzahl der noch folgenden Feldeinträge enthält. Das Ergebnis kann nach einer Konvertierung zu einem `wxDash`-Array direkt an den Zeichenstift mit der Methode

```
pen.SetDashes(dash_list[0], wxdash_list);
```

übergeben werden [2].
Im Zusammenhang mit wxWidgets ist zu beachten, dass die Bibliothek intern auch eine Skalierung der Linienstriche anhand der Stiftbreite vornimmt. Somit muss der Parameter `line_width` der Funktion `gks_get_dash_list` immer auf `1` gesetzt werden, um korrekte Ergebnisse zu erzielen.
Vereinzelte GKS-Linienmuster geben vor, dass der Leerraum zwischen zwei Strichen eine Einheit betragen soll, was bei einer Liniendicke von einem Pixel ebenfalls einem Pixel Leerraum entspricht. Problematischerweise erstellt wxWidgets ungeachtet der Längenangabe immer mindestens drei Pixel Zwischenraum (s. Abb. 13), sodass diese Muster momentan noch nicht korrekt visualisiert werden, wenn die Linienbreite weniger als drei Pixel beträgt.

Abbildung 13: Vergleich der Ausgabe von Linienmustern des wxWidgets- (links) und des X11-Treibers (rechts), eine Einheit Abstand gefordet

3.7.5 Farbtiefe der Cell Arrays

Die Daten eines *Cell Arrays* werden innerhalb des GKSwx-Anzeigetreibers auf einem `wxBitmap`-Objekt abgelegt, um eine Übertragung des Bildinhaltes auf den *Device Con-*

[16] `wxDash` ist ein vorzeichenbehafteter Ganzzahltyp, welcher ein Byte belegt und daher intern dem Typen `signed char` entspricht.

text der Paintroutine mittels der Methode `DrawBitmap` vornehmen zu können. Zu beachten ist hierbei, dass *Cell Arrays* in einer 8-Bit oder 24-Bit Codierung[17] vorliegen können. Bitmaps der wxWidgets-Bibliothek sind jedoch plattformabhängig und unterstützen daher nicht auf jedem System eine 8-Bit-Codierung. Die Codierung mit 24 Bit ist allerdings auf den für diese Arbeit relevanten *wxPorts* wxMSW, wxGTK und wxMac verfügbar [2]. Da die 8-Bit Codierung verlustfrei in eine 24-Bit Codierung übernommen werden kann, werden daher einfach beide Cell Array-Typen auf 24-Bit-Bitmaps übertragen, sodass zwar evtl. Arbeitsspeicher verschwendet wird, aber die Daten auf allen notwendigen Systemen angezeigt werden können.
In der *Workstation Description Table* ist eine Abbildung der 8-Bit Codes auf RGB-Farbwerte[18] enthalten, die genau der 24-Bit Codierung von Bitmaps entspricht. Diese kann somit zur Übersetzung der 8-Bit Cell Arrays auf 24-Bit Bitmaps benutzt werden.

3.7.6 Text

Der GKS-Standard definiert verschiedene Wege der Textdarstellung (sog. Precisions). Zu den gebräuchlichen Varianten zählen [7, S. 97]:

Text Precision Stroke Der darzustellende Text wird mit GKS-Routinen vor der Darstellung in Polylines zerlegt, die dann vom Anzeigetreiber visualisiert werden.

Text Precision String Der Text wird als Zeichenkette an den Treiber übermittelt. An Hand zuvor eingestellter Textparameter und einer zusätzlich übertragenen Textposition kann dann die Textdarstellung vom Anzeigetreiber vorgenommen werden.

Da *Text Precision Stroke* durch Polylines emuliert wird und somit keine größeren Komplikationen bei der Implementierung auftreten, beschränkt sich dieser Unterabschnitt auf die Besonderheiten der Darstellung mit *Text Precision String*. wxWidgets liefert für diesen Modus zwar die Methode

```
void wxDC::DrawText(const wxString& text, wxCoord x, wxCoord y)
```

allerdings sind bei ihrer Verwendung und der Setzung der zugehörigen Paramter einige Punkte zu beachten, wie im Folgenden ausgeführt wird.

3.7.6.1 Schriftenwahl in wxWidgets

Eine der höchsten Prioritäten setzt das wxWidgets-Toolkit bei der Plattformunabhängigkeit bei gleichzeitiger Nutzung so vieler systemeigener Bibliotheken wie möglich, um

[17] Die Codierung gibt an, wie viele Bit zur Speicherung der Farbinformation eines Pixels genutzt werden. Daraus ergibt sich, dass bei einer 8-Bit Codierung maximal 256 Farben verwendet werden können, wohingegen mit 24 Bit 16.777.216 Farben möglich sind.

[18] RGB ist ein Farbraum, der Farben getrennt nach den Farbintensitäten an roter, grüner und blauer Farbe betrachtet. Diese Farbanteile werden typischerweise mit einer Ganzzahl zwischen 0 und 255 beschrieben, wobei 0 keine Intensität und 255 eine sehr hohe Intensität bedeutet.

eine gute Performance und das native *Look and Feel* von wxWidgets-Anwendungen zu realisieren. Wie bereits im vorherigen Abschnitt gesehen (Nutzung plattformabhäniger Bitmaps) lassen sich diese beiden Grundsätze jedoch nicht immer problemlos miteinander vereinen. Aus diesem Grund werden Funktionalitäten, die systemabhängige Komponenten enthalten, in der wxWidgets-Bibliothek wenn möglich immer in einen plattformabhängigen und einen -unabhängigen Teil aufgespalten, um den nicht plattformübergreifenden Teil der Bibliothek möglichst klein zu halten[19] [1, 12f.]. Dieser Grundsatz gilt auch für die Schriftenwahl in wxWidgets. Die Bibliothek stellt hierzu eine Klasse `wxFont` bereit, deren Konstruktor jedoch lediglich Angaben über die gewünschte Schriftenfamilie und Attribute, wie Schriftgröße und -stil (z. B. Fettschreibung) entgegennimmt. Eine direkte Angabe einer konkreten Schriftart ist nicht möglich, da nicht garantiert werden kann, dass eine solche Schriftart auf allen Systemen auch wirklich zur Verfügung steht. Stattdessen ermittelt wxWidgets automatisch auf dem jeweiligen System, auf dem die Anwendung eingesetzt wird, zu den Schriftfamilien passende Schriftarten und setzt diese ein. Auf diese Weise werden Schriften zwar nicht auf allen Systemen gleich aber zumindest ähnlich dargestellt [2].
Das im Forschungszentrum eingesetzte GLIGKS gibt jedoch genau vor, welche Schriftarten darstellbar sein müssen, sodass die reine Nutzung der `wxFont`-Klasse nicht ausreichend ist. Genau für diesen Zweck existiert allerdings die Klasse `wxNativeFontInfo`, welche das plattformabhängige Gegenstück zu `wxFont` repräsentiert. `wxNativeFontInfo` kapselt eine beliebige Schriftart des Systems, die mit der Methode

```
bool wxNativeFontInfo::FromUserString(const wxString&
   font_string)
```

als Zeichenkette spezifiziert werden kann[20]. Anschließend kann mit dem Konstruktor

```
wxFont::wxFont(const wxNativeFontInfo& nativeInfo)
```

ein Schriftenobjekt erzeugt werden, welches die gewünschte Schriftart enthält und einem *Device Context* per

```
wxDC::SetFont(const wxFont& font)
```

übergeben werden kann, sodass die Schrift bei der nächsten Zeichenoperation eingesetzt wird [2].

3.7.6.2 Textpositionierung

Zu Texten, die mit einem auf dem GKS-Standard basierende Visualisierungssystem dargestellt werden sollen, gehört neben dem Textinhalt ebenfalls eine Textposition, welche sich auf einen bestimmten Punkt des späteren Textes bezieht (beispielsweise auf das Zentrum das Textes oder den linken Rand der Basislinie, s. a. Abb. 14) [7, S. 103]. Bei der Verwendung der wxWidgets-Bibliothek bezieht sich die Textposition jedoch immer

[19] Zu `wxBitmap` existiert als Gegenstück die Klasse `wxImage`, die auf allen System gleiche Ergebnisse liefert, aber kein Zeichnen auf *Device Contexts* ermöglicht.
[20] Sollte die Schriftart nicht existieren, so wird eine Standardschrift gewählt und `False` zurückgeliefert, um das Fehlschlagen der Operation anzuzeigen.

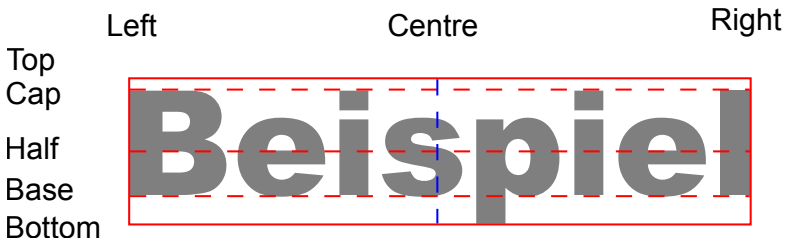


Abbildung 14: Textausrichtungspunkte des GKS

auf die linke obere Ecke eines Rechtecks, welches den anzuzeigenden Text umhüllt (s. Abb. 15) [1, S. 150ff.]. Dieser Punkt stimmt jedoch mit keinem der vom GKS angebotenen Bezugspunkte überein, sodass in jedem Fall eine Umrechnung des Bezugpunktes vorgenommen werden muss, um eine korrekte Textpositionierung zu erhalten.
Der für diese Seminararbeit als Vorlage dienende Qt4-Treiber beinhaltet Informationen über die verschiedenen Fontmetriken (insbesondere der Positionen der Oberlinie der Großbuchstaben) der einzelnen GKS-Schriften und eine Routine, die bereits eine Umrechnung des Bezugspunktes auf den linken Rand der Basislinie vornimmt. Für den wxWidgets-Treiber muss somit noch eine Umrechnung vom linken Rand des Basislinie auf die linke obere Ecke der Textbox vorgenommen werden. Hierzu sind zunächst die wxWidgets eigenen Fontmetriken zu bestimmen:

```
gcdc->GetTextExtent(string, &width, &height, &descent);
```

Mit Hilfe dieser Attribute und der allgemeinen Schrifteneigenschaften kann dann ein Verschiebungsvektor $\begin{pmatrix} xrel \\ -yrel \end{pmatrix}$ berechnet werden, der vom GKS-Bezugspunkt aus gesehen auf die wxWidgets-Textposition zeigt (wenn Rotationen erstmal außer Acht gelassen werden):

```
xrel = width     * xfac[alignment_x];
yrel = capheight * yfac[alignment_y] + (height - descent);
```

`xfac` und `yfac` stellen hierbei von der gewünschten Ausrichtung abhängige Faktoren dar, die die Textbreite bzw. Oberlinie der Großbuchstaben des aktuellen Fonts passend skalieren, sodass der resultierende Vektor auf die linke Seite der Basislinie des Textes verweist. Durch die Addition des Terms $(height - descent)$ in y-Richtung wird dann zusätzlich eine Umrechnung von der Basislinie auf die Oberlinie vorgenommen. Eine zusätzliche Anpassung in x-Richtung ist nicht notwendig.

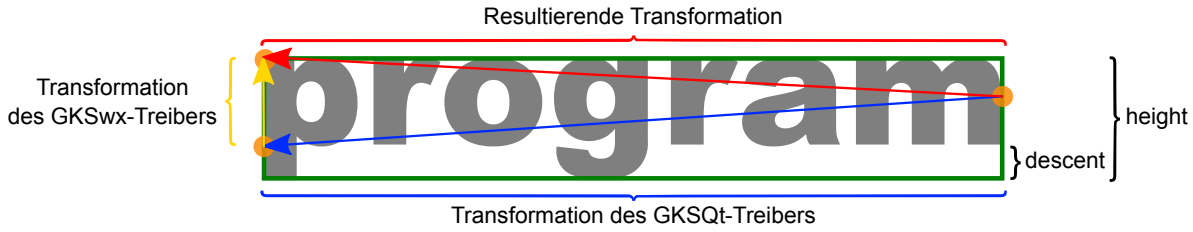


Abbildung 15: Umrechnung des Bezugspunktes (ohne Beachtung einer evtl. Rotation)

3.7.6.3 Textrotation

Das GKS liefert zusätzlich einen *Up*-Vektor $\begin{pmatrix} ux \\ uy \end{pmatrix}$, der senkrecht auf der Basislinie des Textes steht und so eine Textrotation spezifiziert [7, S. 100f.]. Mit Hilfe der Funktion arctan2[21] kann dann der Winkel zwischen *Up*-Vektor und y-Achse im Bogenmaß bestimmt werden:

$$alpha = \text{arctan2}(uy, ux) - \frac{\pi}{2}$$

Der Verschiebungsvektor zur Umrechnung des Bezugspunktes muss natürlich auch rotiert werden, um den Text noch korrekt zu positionieren. Dies kann mit einer gewöhnlichen Rotationsmatrix durchgeführt werden (s. Abb. 16 auf S. 33):

$$\begin{pmatrix} xrel' \\ yrel' \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} xrel \\ yrel \end{pmatrix}$$

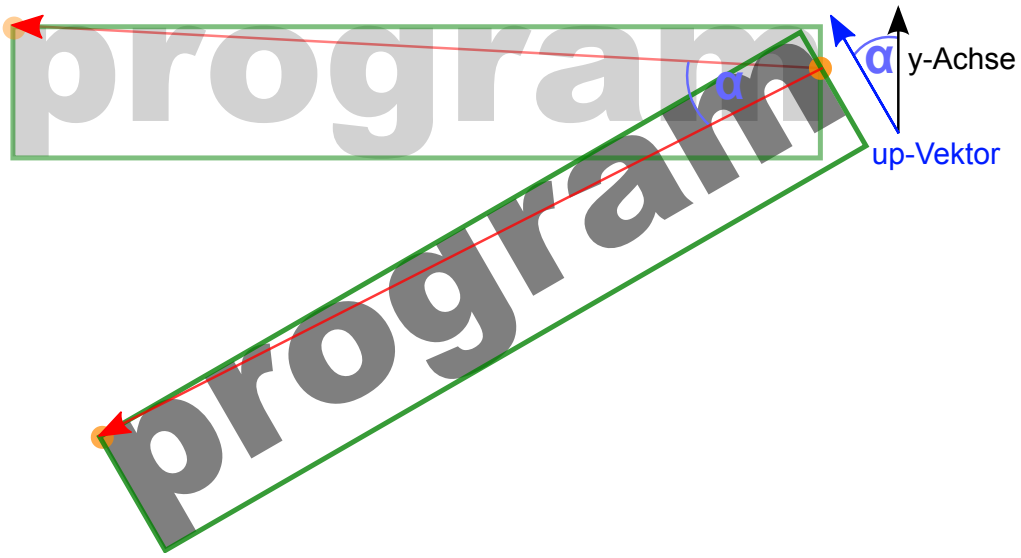


Abbildung 16: Zusätzliche Rotation des Verschiebungsvektors

Nach einer Addition von $\begin{pmatrix} xrel' \\ -yrel' \end{pmatrix}$ auf den GKS-Bezugspunkt kann man das Resultat als Textposition im wxWidget-Kontext verwenden[22]:

```
#define FEPS 1.0E-06

angle = rad_to_deg(alpha);
```

[21] arctan2 ist in vielen Programmiersprachen definiert und liefert den Winkel zwischen einem Vektor $\begin{pmatrix} ux \\ uy \end{pmatrix}$ und der x-Achse im Bogenmaß. Durch Subtraktion von $\frac{\pi}{2}$ erhält man den Winkel zur y-Achse.

[22] $-yrel'$ ist notwendig, da das hier verwendete wxWidgets-Koordinatensystem an der y-Achse gespiegelt ist.

```
if(fabs(angle) > FEPS) {
    gcdc->DrawRotatedText(string, xstart, ystart, angle);
} else {
    gcdc->DrawText(string, xstart, ystart);
}
```

Sollte der Winkel (unter Berücksichtigung der begrenzten Genauigkeit von Gleitkommazahlen) 0° betragen, so wird die gewöhnliche Textroutine aufgerufen, ansonsten die für gedrehte Texte.

3.7.7 Antialiasing

Die Klasse `wxGCDC` liefert als Aufsatz für andere `wxDC`-Objekte die Möglichkeit eine automatische Kantenglättung durchzuführen.
Mit der Shellvariablen `GKSWX_ENABLE_ANTIALIASING=1` kann der GKSwx-Anzeigetreiber veranlasst werden, Antialiasing auf dem verwendeten `wxGCDC` zu aktivieren bzw. zu deaktivieren:

```
char *shell_value = getenv("GKSWX_ENABLE_ANTIALIASING");
int antialias = shell_value && strcmp(shell_value, "1") == 0;

bmp = new wxBitmap(width, height);
memory_dc = new wxMemoryDC(*bmp);
gcdc = new wxGCDC(*memory_dc);

if(antialias)
    // Antialiasing aktivieren
    gcdc->GetGraphicsContext()->
       SetAntialiasMode(wxANTIALIAS_DEFAULT);
else
    // Antialiasing deaktivieren
    gcdc->GetGraphicsContext()->
       SetAntialiasMode(wxANTIALIAS_NONE);
```

Ein Fehler in der aktuellen wxWidgets Entwicklungsversion 2.9.3 sorgt jedoch dafür, dass nach jedem Reset der *Clipping Region*[23] wieder die Standardeinstellung der Kantenglättung eingesetzt wird (Kantenglättung aktiv). Dieses Problem kann mit folgenden Code umgangen werden:

```
1 void GKSWidget::destroy_clipping_region(wxGCDC &dc, bool
     antialias) {
2     dc.DestroyClippingRegion();
3
4     dc.GetGraphicsContext()->
         SetAntialiasMode(wxANTIALIAS_DEFAULT);
5     if(!antialias)
```

[23] Die *Clipping Region* gibt den Bereich auf einem *DC* an, auf dem Zeichenoperationen auch tatsächlich ausgeführt werden [2].

```
6           // Antialiasing deaktivieren
7           gcdc->GetGraphicsContext()->
               SetAntialiasMode(wxANTIALIAS_NONE);
8   }
```

Nach dem Reset der *ClippingRegion* (2) wird die Kantenglättung zunächst nochmal auf den Standardwert gesetzt (aktiv, 4). Dies ist notwendig, da sonst die folgenden Befehle keine Wirkung haben. Sollte keine Kantenglättung gewünscht sein, so wird diese dann nochmal explizit ausgeschaltet (6 und 7).
Statt die Methode `DestroyClippingRegion` auf einem `wxDC` aufzurufen, kann dann die hier vorgestellte Funktion genutzt werden, um den eigentlich gewünschten Effekt zu erreichen.

3.7.8 Timer-gesteuerter Refresh

Während den Testläufen des Anzeigetreibers ist aufgefallen, dass keine korrekte Anzeigeaktualisierung mehr vorgenommen wird, wenn sehr viele Displaylisten direkt hintereinander an den Treiber verschickt wurden. Dies findet sich in einer Überflutung der wxWidgets-*Eventqueue*[24] begründet, da ursprünglich nach jeder Verarbeitung einer neu angekommenden Displayliste explizit ein *PaintEvent* mit der Methode `Refresh` generiert wurde, um eine Aktualisierung des Fensterinhalts und somit der angezeigten Daten vorzunehmen (s. Abschnitt 3.7.1 auf S. 25). Bei der Erzeugung zu vieler *PaintEvents* werden diese allerdings ab einem bestimmten Zeitpunkt an verworfen, sodass die letzten übertragenen Datenlisten nicht mehr sofort angezeigt werden. Diese werden erst sichtbar, wenn vom System nachträglich ein *PaintEvent* erzeugt wird (z. B. nach dem Maximieren des Fensters, s. a. Abschnitt 2.2.3.1 auf S. 14).
Es war daher also notwendig die Menge an generierten Ereignissen zu reduzieren, indem nicht mehr jede neu ankommende Displayliste ein *PaintEvent* auslöst. Statt *PaintEvents* fest an die Daten zu koppeln, ist es alternativ auch möglich in regelmäßigen kurzen Abständen zu prüfen, ob neue Daten vorliegen und dann eine Aktualisierung durchzuführen. Auf diese Weise werden die Daten von der Anzeigenerneuerung entkoppelt und die durch diese Technik entstehenden Anzeigeverzögerungen sind kaum bemerkbar, wenn das Prüfintervall klein genug gewählt wird[25].
Für Aufgaben, die in regelmäßigen Abständen wiederholt werden sollen, bietet wxWidgets die Klasse `wxTimer`. Diese generiert innerhalb eines festlegbaren Intervalls wiederholt *TimerEvents*, auf die mit einer geeigneten *Callback*-Routine reagiert werden kann. Es muss also zunächst ein *Timer* erzeugt, eine Routine registriert und der *Timer* gestartet werden:

```
#define TIMER_INTERVAL 100

wxTimer paintTimer(this);
```

[24] Alle erzeugten Ereignisse werden in einer Warteschlange gesammelt, sodass sie nacheinander abgearbeitet werden können.
[25] Im fertigen Treiber beträgt die Zeit zwischen zwei Prüfungen 100ms.

```
Connect(wxEVT_TIMER,
   wxTimerEventHandler(GKSWidget::timerEvent));
paintTimer.Start(TIMER_INTERVAL, wxTIMER_CONTINUOUS);
```

`this` spezifiziert, dass die generierten Ereignisse in die *Eventqueue* des GKSWidget eingereiht werden sollen.
Nun wird wiederholt folgende Methode aufgerufen:

```
1  void GKSWidget::timerEvent(wxTimerEvent& event) {
2      if(do_redraw) {
3          Refresh();
4          do_redraw = false;
5      }
6  }
```

Statt bei der Ankunft neuer Daten direkt die `Refresh`-Methode aufzurufen, wird stattdessen nur der Schalter `do_redraw` auf `True` gesetzt. Beim nächsten Aufruf dieser Methode wird dann festgestellt, dass neue Daten vorliegen (`2`), sodass anschließend ein *PaintEvent* generiert wird (`3`). Der Schalter muss anschließend wieder auf `False` gestellt werden, um beim nächsten Aufruf der Methode keine unnötige erneute Aktualisierung vorzunehmen.
An diesem Punkt könnte der Eindruck entstehen, dass das Einlesen neuer Daten (und somit das Setzen das `do_redraw`-Flags) und das Überprüfen des Flags innerhalb der *Timer*-Routine parallel ablaufen würden und somit eine Synchronisierung der Lese- und Schreiboperationen auf `do_redraw` notwendig wäre. Da die GUI jedoch weiterhin unter Verwendung einer *Eventqueue* die Netzwerk- und *Timer*-Ereignisse sequentiell und *singlethreaded* verarbeitet, gibt es keine gleichzeitigen Zugriffe und es müssen keine Synchronisierungen durchgeführt werden.

3.7.9 Nutzung von C++-Methoden als C-Callback-Routinen

Einige GKS-Bibliotheksfunktionen erwarten als Paramter einen Zeiger auf eine C-Funktion, die als *Callback*-Routine verwendet wird. Alle Funktionen des *GKSWidgets* sind jedoch als C++-Methoden realisiert, die nicht direkt als *Callback*-Funktion verwendet werden können. Zwar bietet C++ die Möglichkeit ungebundene Methodenzeiger[26] zu definieren, aber dennoch können diese auch nicht als *Callback*-Funktion eingesetzt werden, da sie nur mit Angabe eines konkreten Objektes benutzt werden können (die in C nicht existieren). Benötigt wird also ein gebundener Methodenzeiger[27], der bereits die Information über ein konkretes Objekt enthält.
Da C++ solche gebundenen Methodenzeiger von sich aus nicht unterstützt, müssen sie nachgebildet werden. Die Überlegungen zur Emulation eines solchen Zeigers ist nun folgende:

[26] Ungebundene Methodenzeiger verweisen auf die Methode eines Objekts ohne die Information über ein konkretes Objekt zu enthalten. Sie können daher nur mit Angabe eines Objekts verwendet werden.
[27] Gebundene Methodenzeiger kapseln zusätzlich ein Objekt, auf dem die Methode aufgerufen werden soll. Sie können daher wie gewöhnliche Funktionszeiger benutzt werden.

- Die GKS-Routinen erwarten einen gewöhnlichen C-Funktionszeiger. Also muss eine Funktion eingeführt werden, auf die später der zu übergebene Zeiger verweist.
- Diese Funktion muss die Information über das zu verwendene Objekt und den zu verwendenen Methodenzeiger beinhalten und die Methode auf diesem Objekt aufrufen. Die Funktion dient also nur als Adapter [15].
- Die Adapterfunktion soll für verschiedene Methodenzeiger aufrufbar sein. Daher muss sie mit Hilfe eines Templates abstrahiert werden.
- Zusätzlich soll der Adapter für verschiedene Objekte verwendet werden können. Dies kann über eine Klassenvariable des *GKSWidgets* realisiert werden, die speichert, welches das aktuell zu verwendene Objekt ist. Diese Information muss also vor jedem Aufruf des Adapters gesetzt werden. Da dieses Vorgehen aber fehleranfällig ist (es könnte vergessen werden das aktuelle Objekt anzugeben), wird zusätzlich ein Makro definiert, welches diese beiden Anweisungen kapselt.

Für die konkrete Implementierung wird also ein Funktionstemplate erstellt:

```
1  template <void (GKSWidget::*fp) (float, float)>
2  void call_adapter(float x, float y) {
3      (GKSWidget::current_global_object->*fp)(x, y);
4  }
```

Dieses hier vorgestellte Template stellt exemplarisch einen Adapter für eine Funktion bereit, die zwei `float`-Parameter erwartet und keinen Rückgabewert liefert (2). Als Template-Parameter wird analog ein entsprechender Zeiger auf eine `GKSWidget`-Methode erwartet (1). Mit dieser Angabe und dem vorher zu setzenden `GKSWidget`-Objekt `current_global_object`, kann dann die gewünschte Methode innerhalb des Adapters aufgerufen werden.
Um das Setzen des aktuellen `GKSWidget`-Objektes zu vereinfachen, wurde zusätzlich folgendes Makro definiert:

```
#define get_call_adapter(method, type) (current_global_object
   = this, static_cast<type>(call_adapter<&method>))
```

Das Makro erhält als Paramter den Namen der umzuwandelnden Methode und den Typen des zurückzuliefernden Funktionszeigers. Da Makros vor der Kompilierung in den Quellcode eingesetzt werden, kann mit Hilfe der Anweisung

```
current_global_object = this
```

die Variable `current_global_object` auf das im Aufrufkontext aktuell verwendete Objekt gesetzt werden. Mit Hilfe des Adressoperators `&` wird aus dem Methodenname der benötigte Zeiger generiert, um den korrekten Typ zur Templateinstanziierung angeben zu können. Die zusätzliche Verwendung des `static_casts` auf den gewünschten Funktionstyp ist notwendig, da der Compiler nicht den Rückgabetyp des Kommaoperators alleine aus dem Templateaufruf bestimmen kann. Dies ist aber zwingend erforderlich, um das Makro direkt an Stellen verwenden zu können, an denen ein Funktionszeiger vom Typ `type` benötigt wird.

3.8 Ablaufdiagramm des GKSwx-Treibers

Das folgende Aktivitätsdiagramm fasst die zuvor beschrieben Abläufe innerhalb des GKSwx-Treibers zusammen:

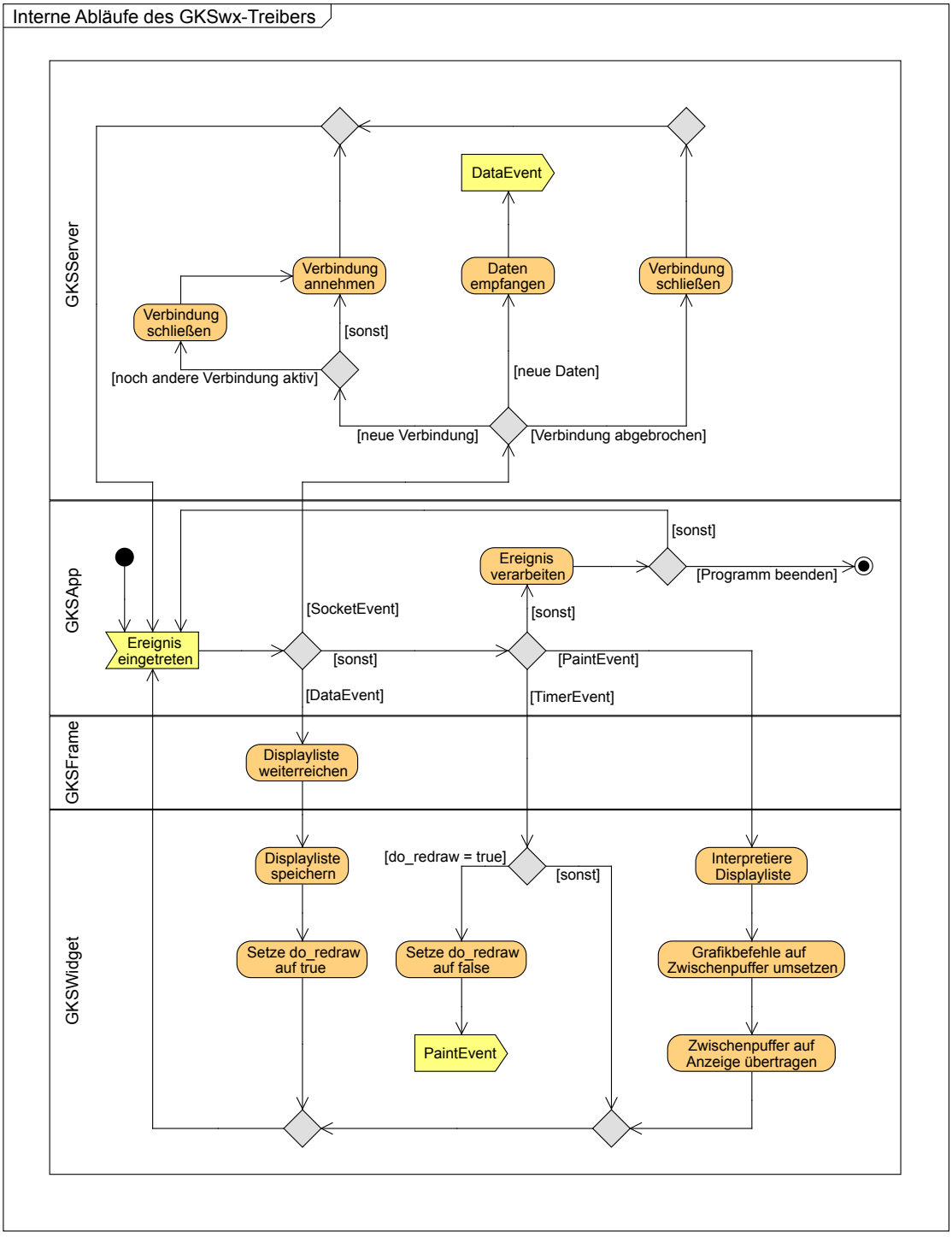


Abbildung 17: Darstellung der internen Arbeitsschritte des GKSwx-Treibers

4 Ergebnisse

Während der Implementierung des Anzeigetreibers sind verschiedene Schwierigkeiten aufgetreten (s. Realisierung Kap. 3 auf S. 17), von denen die meisten jedoch behoben werden konnten, sodass nun ein funktionierender Anzeigetreiber auf Basis des wxWidgets-Toolkits in der aktuellen Entwicklungsversion 2.9.3 für Mac OS X- und Linuxsysteme zur Verfügung steht. Aus Zeitgründen sind im Rahmen dieser Seminararbeit noch keine Tests auf Windowssystemen durchgeführt worden.
Die durchgeführten Tests haben gezeigt, dass auf Mac OS X-Systemen nur geringe Abweichungen zwischen dem neuen GKSwx- und dem Referenztreiber[28] bestehen. Alleinig die Linienmuster unterscheiden sich an einigen Stellen aufgrund von Einschränkungen der wxWidgets-Bibliothek (s. Linienmuster in Abschnitt 3.7.4 auf S. 28). Zusätzlich zu den bisherigen Treibern für grafische Oberflächen bietet der wxWidgets-Treiber eine optional zuschaltbare Kantenglättung (s. Abb. 19 auf S. 40 und Abb. 21 auf S. 41).
Auf GTK+-basierten Systemen haben sich allerdings noch weitere Schwächen im Bezug auf die GKS-Füllmuster gezeigt. Bisher fehlt im GTK-*Port* die Implementierung von Füllmustern unter Verwendung der `wxGCDC`-Klasse (s. Abb. 20 auf S. 41). Realisiert man den Treiber alleinig mit *MemoryDCs*, so können auch die Füllmuster umgesetzt werden, jedoch ist dann keine automatische Kantenglättung mehr möglich.

[28] Als Referenztreiber diente die X11-Implementierung.

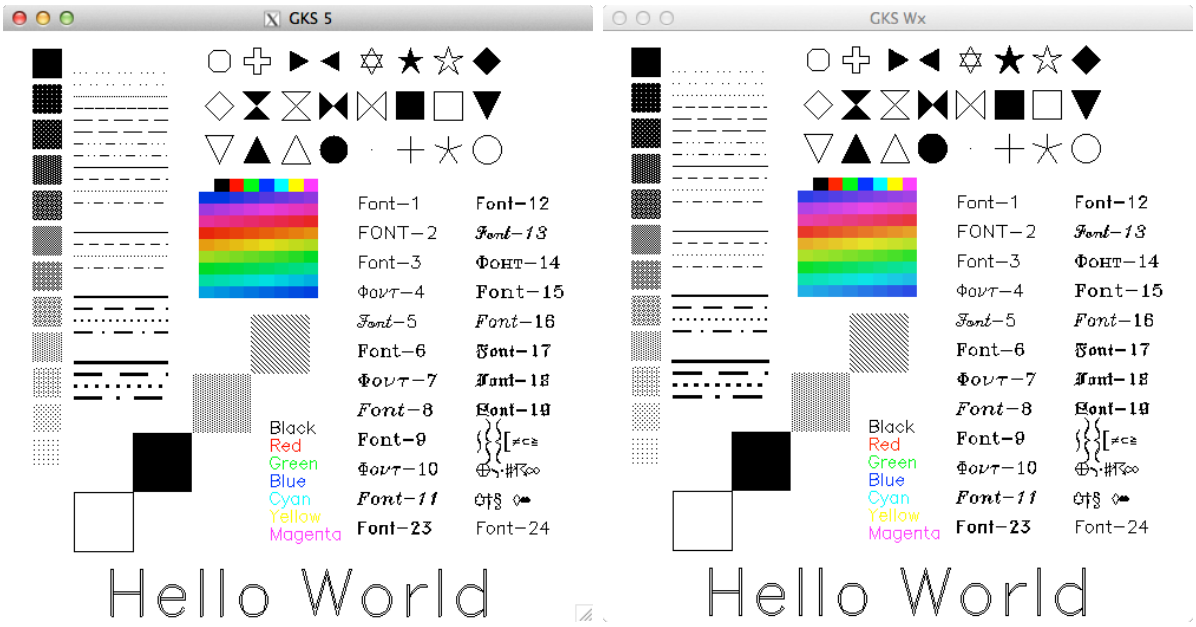


Abbildung 18: Mac OS X: Vergleich des Referenztreibers (links mit GKSwx ohne Antialiasing (rechts)

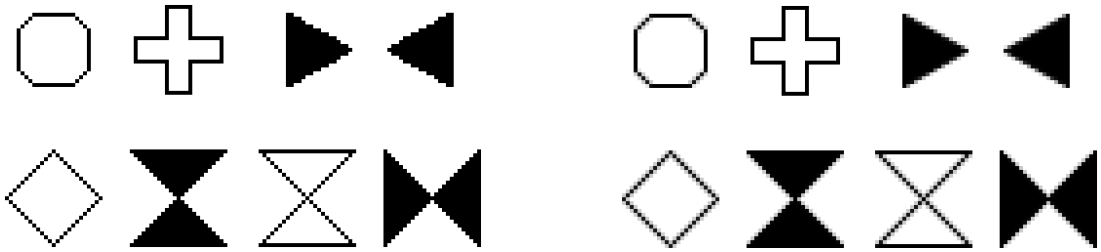


Abbildung 19: Mac OS X und GKSwx: Vergleich der Darstellung bei deaktivierter (links) und aktivierter Kantenglättung (rechts)

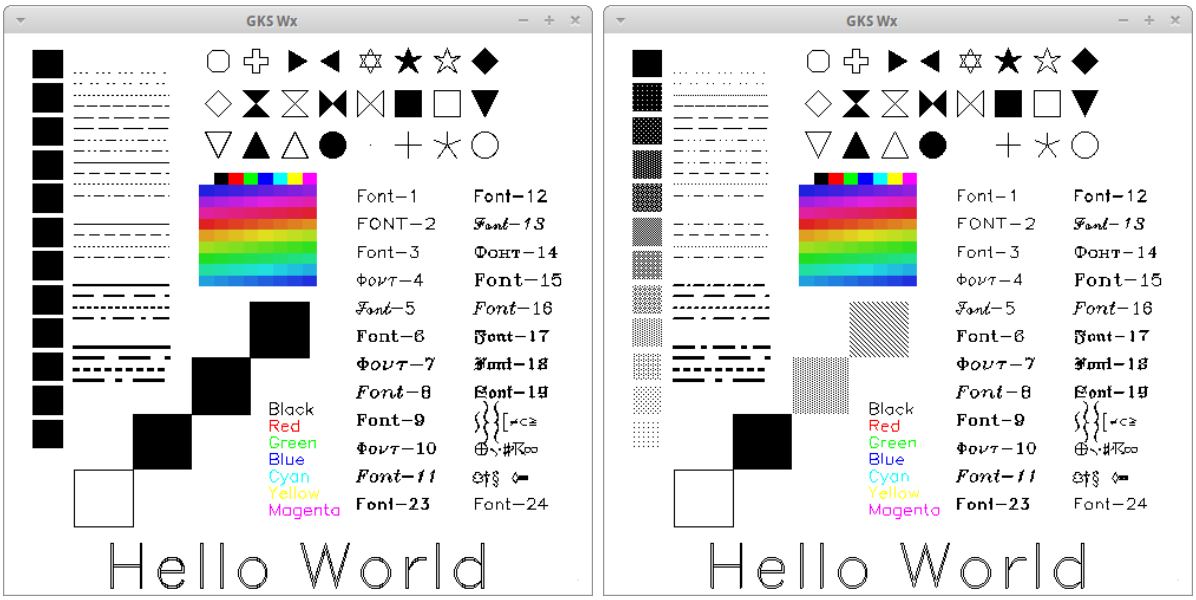


Abbildung 20: Linux mit GTK+: Implementierung mit (links) und ohne *GCDC* (rechts) (beides ohne Antialiasing)

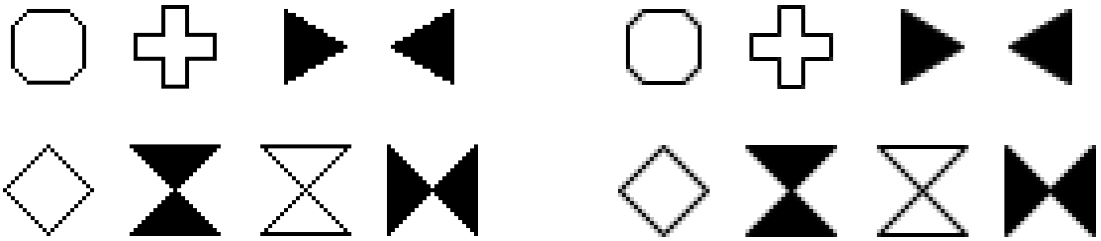


Abbildung 21: Linux mit GTK+ und GKSwx: Vergleich der Darstellung bei deaktivierter (links) und aktivierter Kantenglättung (rechts) (beide mit GCDC)

5 Ausblick

Die Ausgabe des Anzeigetreibers deckt sich noch nicht zu 100 Prozent mit der Referenzimplementierung. Es ist daher noch zu prüfen, inwieweit durch Feinabstimmungen die grafische Ausgabe des Treibers optimiert werden kann. Teile, die durch wxWidgets-Routinen nicht abgedeckt werden können (z. B. die Linienmuster, s. Abschnitt 3.7.4 auf S. 28), müssen evtl. zu Teilen durch Emulationsroutinen ergänzt werden.
Auf Linuxsystem muss weiterhin noch festgestellt werden, inwieweit eine manuelle Kantenglättung praktikabel ist, um ähnliche Ergebnisse wie auf Mac OS X zu erreichen.
Weiterhin ist zu überprüfen, inwieweit der Treiber auf Windows-Plattformen einsetzbar ist und ob noch weitere Anpassungen gemacht werden müssen.
Besonderes Augenmerk wurde während der Implementierungsphase darauf gelegt, den Code unkompliziert zur Entwicklung eines GKS-Plugins wiederverwenden zu können. Ein solches Zusatzprogramm ist im Folgenden zu entwickeln, sodass das im Rahmen dieser Arbeit implementierte *GKSWidget* auch tatsächlich als gewöhnliche GUI-Komponente in beliebigen Anwendungen einsetzbar ist.
Die bisher vorliegenden GKS-Treiber, die direkt in eine grafische Oberfläche eingebettet werden können, sind in C oder C++ implementiert. Anwendungen im Peter Grünberg Institut/Jülich Centre for Neutron Science werden allerdings vermehrt auf Basis objektorientierter Interpreter (Python) implementiert, da diese u. a. eine schnelle Anwendungsentwicklung ermöglichen. Im Folgenden ist daher zu untersuchen, inwieweit es möglich ist eine Metaebene zu entwerfen, die die Funktionalitäten der verschiedenen GUI-Toolkits und der zugehörigen GKS-Treiber abstrahiert, sodass vereinfacht Anwendungen entwickelt werden können, die auf GKS-Funktionalitäten zurückgreifen.

Literatur

[1] Julian Smart / Kevin Hock / Stefan Csomor:
Cross-Platform GUI Programming with wxWidgets.
Prentice Hall 2005.
http://www.informit.com/content/images/0131473816/downloads/0131473816_book.pdf
(zuletzt geprüft am: 18.12.2011).

[2] Dokumentation der wxWidgets-Entwicklungsversion 2.9.3
http://docs.wxwidgets.org/2.9.3/
(zuletzt geprüft am: 20.12.2011).

[3] wxWidgets-Lizenz
http://wxwidgets.org/about/newlicen.htm
(zuletzt geprüft am: 17.12.2011).

[4] Eintrag zu *Custom Events* im wxWiki
http://wiki.wxwidgets.org/Custom_Events
(zuletzt geprüft am: 18.12.2011).

[5] *Custom Events Tutorial* im wxWiki
http://wiki.wxwidgets.org/Custom_Events_Tutorial
(zuletzt geprüft am: 18.12.2011).

[6] Bug Ticketsystem der wxWidgets-Bibliothek
http://trac.wxwidgets.org/ticket/11981
(zuletzt geprüft am: 16.12.2011).

[7] Jörg Bechlars / Rainer Buhtz:
GKS in der Praxis.
Springer Verlag 1985.

[8] Josef Heinen:
Graphics Language Interpreter Reference Manual.
Forschungszentrum Jülich GmbH 1995.
http://iffwww.iff.kfa-juelich.de/gli/manual.pdf
(zuletzt geprüft am: 16.12.2011).

[9] Marlene Busch / Gerd Groten:
Benutzerhandbuch GR-Software. Forschungszentrum Jülich GmbH 1998.
http://www2.fz-juelich.de/jsc/files/docs/bhb/bhb-0096.pdf
(zuletzt geprüft am: 17.12.2011).

[10] Dr. H. Schumacher / M. Busch / J. Heinen:
2D- und 3D-Visualisierung mit den Graphiksystemen GR, GLI, IDL und AVS - Ein Überblick.

Forschungszentrum Jülich GmbH 1996.
http://iffwww.iff.kfa-juelich.de/gli/kurs.pdf
(zuletzt geprüft am: 16.12.2011).

[11] Marcel Dück:
Entwicklung eines GKS-Gerätetreibers als Java-basierte Client/Server Webanwendung.
Forschungszentrum Jülich GmbH 2010.

[12] Eintrag zu *Double Buffering* in der MSDN
http://msdn.microsoft.com/en-us/library/b367a457.aspx
(zuletzt geprüft am: 18.12.2011).

[13] Eintrag zu *Antialiasing* in der MSDN
http://msdn.microsoft.com/de-de/site/9t6sa8s9
(zuletzt geprüft am: 18.12.2011).

[14] EVA-Prinzip
http://www.edv-lehrgang.de/edv-grundlagen/
(zuletzt geprüft am: 21.12.2011).

[15] Parashift C++ FAQ
http://www.parashift.com/c++-faq-lite/pointers-to-members.html#faq-33.2
(zuletzt geprüft am: 17.12.2011).