Bachelorarbeit im Rahmen des Studiengangs

Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

Programmierung eines GKS-Plugins in Python zur Erzeugung von statischen JavaScript/HTML5 Inhalten

---

Jülich, den 23. August 2013

David Knodt

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. Dipl. Math. 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 des Peter Grünberg Instituts/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. Zur Visualisierung der Messdaten wird häufig ein Grafisches Kernsystem verwendet, welches eine einheitliche Schnittstelle und viele Ausgabemöglichkeiten bietet (z. B. PDF, Qt, PNG).

Im Rahmen dieser Bachelorarbeit ist ein in Python geschriebener Ausgabetreiber zu entwickeln, der die Daten in einem statischen HTML-Dokument darstellt. Hierfür ist das Canvas-Element des HTML5-Standards zu verwenden. Das Python-Modul zur Interpretation der GKS-Daten ist so zu erstellen, dass es für weitere Python-Ausgabetreiber wiederverwendet werden kann. Die Kommunikation zwischen dem GKS und dem Treiber ist mit der Bibliothek ZeroMQ zu realiseren.

Inhaltsverzeichnis

1 Motivation **1**

2 GKS **3**

2.1 Grundlagen . . . 3
2.1.1 Koordinatentransformation . . . 4
2.1.2 Workstationkonzept . . . 4
2.1.3 Aufbau der Displayliste . . . 5
2.2 Verwendung von Python . . . 6
2.2.1 Aufbau der Module . . . 6
2.3 GKS-Datenparser . . . 7
2.4 *ctypes*-Wrapper . . . 8
2.4.1 Einfache Funktionen und Standarddatentypen . . . 9
2.4.2 Nachbilden der GKS-Strukturen . . . 10
2.4.3 Callback-Funktionen . . . 10

3 ZeroMQ **13**

3.1 Grundlagen . . . 13
3.2 Publish-Subscribe . . . 14
3.3 Push-Pull . . . 16

4 JavaScript und HTML5 **17**

4.1 Grundlagen . . . 17
4.1.1 HTML . . . 17
4.1.1.1 HTML5 . . . 18
4.1.2 JavaScript . . . 19
4.2 `<canvas>`-Element . . . 20
4.3 Zeichenroutinen . . . 22
4.3.1 Polyline . . . 22
4.3.2 Fillarea . . . 23
4.3.3 Polymarker . . . 24
4.3.4 Cellarray . . . 26
4.3.5 Text . . . 28
4.3.5.1 Textpositionierung . . . 28

4.3.5.2 Textrotation 29
4.3.5.3 Schriftartendefinition in HTML 31
4.3.5.4 Schriftdarstellung im GKS 31
4.3.6 Dynamische Skalierung der Gesamtgrafik 32
4.4 Browserunterstützung 33
4.4.1 Testen der verschiedenen Browser 34
4.4.1.1 Google Chrome 34
4.4.1.2 Mozilla Firefox 34
4.4.1.3 Apple Safari 34
4.4.1.4 Internet Explorer 35
4.4.2 Schriftdarstellung 35

5 Zusammenfassung **37**

6 Ausblick **39**

Literaturverzeichnis **40**

Abbildungsverzeichnis

2.1 GKS-Schichtenmodell mit zu entwickelndem Treiber 3
2.2 Koordinatenräume des GKS [1, S. 51] 4
2.3 Schematischer Aufbau der Displayliste 5
2.4 Module des Treibers und ihre Beziehungen 6
2.5 Aufrufkette der GKS-Routinen 8

3.1 Gewünschte 1:N Kommunikation zwischen GKS und Ausgabetreiber . 14
3.2 1:1 Kommunikation mit dem Push-Pull-Pattern 16

4.1 Zusammenspiel zwischen Canvas und Zeichenkontext 21
4.2 Linienmuster am Beispiel `[6, 1, 3, 2]` 23
4.3 Fillarea Beispiele (v. l. n. r.: `hollow`, `solid`, `pattern`, `hatch`) 23
4.4 Erstellung eines Fillarea-Musters 24
4.5 Beispiele für Markersymbole 25
4.6 Relative Koordinatenangaben 26
4.7 Cellarray Beispiel 26
4.8 Auslesen der RGB-Werte aus einem Integer 27
4.9 Verarbeitung der Bilddaten 28
4.10 Textpositionierung im GKS [21] 28
4.11 Richtung des Zeichenaufwärtsvektors 29
4.12 Bestimmung des Rotationswinkels 30
4.13 Ausgabe des existierenden PostScript-Treibers 33

1 Motivation

Im wissenschaftlichen Bereich werden zur Visualisierung häufig 2D-Grafiken verwendet. Sie stellen große Datenmengen übersichtlich dar und erleichtern so die Analyse. Eine Implementierung des Grafischen Kernsystems (GKS) generiert diese Grafiken und stellt verschiedene Ausgabemedien bereit. Diese Ausgabemedien werden mit sogenannten Ausgabetreibern realisiert, indem die vom GKS vorgesehenen Grafikprimitiven gerätespezifisch umgesetzt werden.

Die Verwendung mobiler Endgeräte hat zugenommen und es wird ein Ausgabeformat benötigt, das auch für diese geeignet ist. Hier hat sich in modernen Browsern HTML5 etabliert. Es gibt sowohl für mobile als auch für stationäre Geräte Browser mit HTML5-Unterstützung. Daher bietet sich die Entwicklung eines HTML5-Ausgabetreibers auf der Basis des neu eingeführte Canvas-Element an, um die 2D-Grafiken darzustellen. Das Zeichnen mit JavaScript bietet zusätzlich Spielraum für Interaktionsmöglichkeiten mit der Grafik.

Der Treiber soll in der Skriptsprache Python geschrieben werden und Module zum Verarbeiten des GKS-Datenstroms und zur Kommunikation mit dem GKS beinhalten. Diese Module können von weiteren Python-Treibern wiederverwendet werden. Für die Kommunikation bietet sich die Bibliothek ZeroMQ an, da sie eine einfache Schnittstelle zur Netzwerkkommunikation mit vielen Sprachanbindungen bereitstellt, sodass auch Treiber in anderen Sprachen von der performanten Kommunikation mit ZeroMQ Gebrauch machen können.

2 GKS

Im Folgenden wird das GKS vorgestellt, welches zur Erstellung der Grafiken genutzt wird. Es wird auf die Verwendung von Python und der Bibliothek *ctypes* eingegangen und die Realisierung des GKS-Datenparsers dargestellt.

2.1 Grundlagen

Das *Grafische Kernsystem* (GKS) bietet Grundfunktionen zur Erstellung bzw. Darstellung von zweidimensionalen Grafiken.

In einer internationalen Norm (ISO 7942 [8]) werden Schnittstelle und Funktionsumfang festgelegt. Eine Implementierung stellt elementare Grafikoperationen und Treiber für verschiedene Ausgabeformate (z. B. PNG, PDF, wxWidgets, ...) bereit. Als Ausgabemedium dienen, abhängig vom verwendeten Treiber, Dateien, Fenster oder Drucker. Die erstellten Grafiken sind plattformunabhängig und werden deshalb auf jedem System gleich dargestellt. Die Realisierung dieser geräte- und plattformunabhängige Schnittstelle erfolgt durch logische Gerätetreiber, welche die Grafikoperation gerätespezifisch umsetzen.

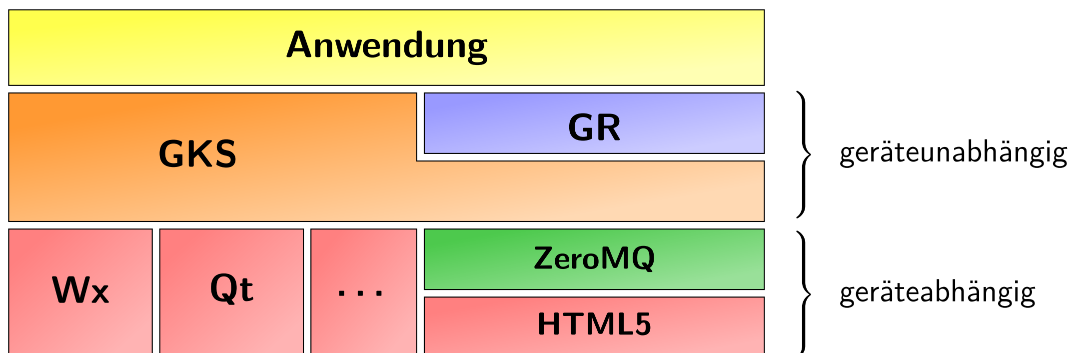


Abbildung 2.1: GKS-Schichtenmodell mit zu entwickelndem Treiber

Der HTML5-Treiber ist ein weiterer Ausgabetreiber des GKS, wie in Abbildung 2.1 in rot zu sehen. Er soll mit dem GKS über die Bibliothek ZeroMQ kommunizieren. Das GKS wiederum stellt dem Anwender eine Schnittstelle mit Zeichenroutinen zur Verfügung, die durch die Graphics Library (GR) weiter abstrahiert wird. Sie stellt

einen umfassenderen Satz an Grafikfunktionen bereit, die auf den elementaren Operationen des GKS aufbauen.

2.1.1 Koordinatentransformation

Das GKS bietet dem Anwender die Möglichkeit, ein eigenes Koordinatensystem zu definieren, das zum Zeichnen genutzt wird. Intern verwendet das GKS ein normiertes Koordinatensystem, das wiederum unabhängig von dem Koordinatensystem des Ausgabemediums ist. Daher sieht die GKS-Norm eine 2-Wege-Transformation vor, um aus den Anwendungskoordinaten die Gerätekoordinaten zu erhalten. Der darzustellende Bereich des *windows* wird auf das normierte Koordinatensystem abgebildet. Auf dieser Ebene gibt es wiederum einen Bereich, genannt *workstation window*, der auf die Gerätekoordinaten projiziert wird (*workstation viewport*). Die Schnittmenge des *viewports* und des *workstation windows* enthält die zu visualisierenden Elemente.

Außerdem kann im GKS zusätzlich *clipping* verwendet werden. Hierbei wird eine rechteckige Fläche des Zeichenbereichs definiert. Alle Elemente, die innerhalb dieser Fläche liegen, werden angezeigt. Alle anderen Elemente, die über den Rand der Fläche hinausragen, werden abgeschnitten.

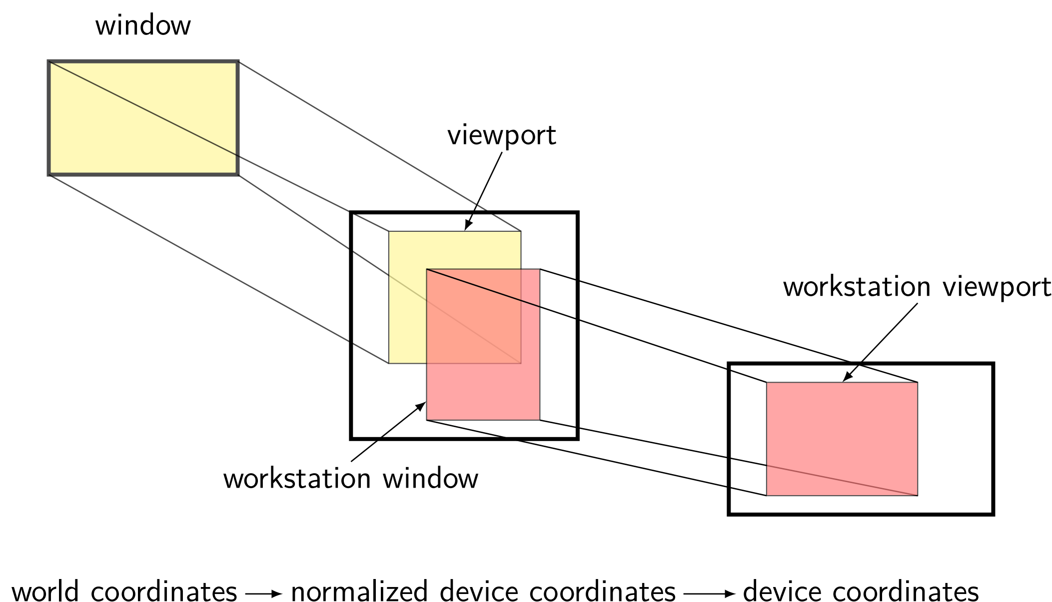


Abbildung 2.2: Koordinatenräume des GKS [1, S. 51]

2.1.2 Workstationkonzept

Der Begriff *workstation* bezeichnet die „logische Abstraktion eines graphischen Ein-/Ausgabegerätes“[1, S. 38]. Somit ist sie der Stellvertreter dieser Geräte und enthält

Informationen zu diesen. Diese werden in der sogenannten *workstation description table* gespeichert. Die *workstation state list* stellt den aktuellen Zustand der Workstation dar und enthält alle anwendungsspezifischen Daten, die vom Treiber gezielt verändert werden können. Zusätzlich zu den Daten beinhaltet die Workstation die grafische Realisierung. Sie ist die Umsetzung der Transformations- und Zeichenroutinen für das entsprechende Ausgabemedium. Jeder Workstation ist eine eindeutige Nummer (*workstation-type*) zugeordnet, über die sie angesprochen wird. Es ist auch möglich, Zeichenketten (z. B. `"png"` oder `"qt"`) stellvertretend für ein Ausgabemedium anzugeben. Diese werden intern auf den *workstation-type* abgebildet.

2.1.3 Aufbau der Displayliste

Die darzustellenden Daten werden dem Treiber in einer sogenannten *Displayliste* übermittelt. Sie beginnt mit vier Byte, in denen die Länge der gesamten Liste gespeichert ist. Darauf folgt eine Reihe von Funktionsblöcken, an deren Anfang die Länge des Blocks und die Funktions-ID der aufzurufenden Funktion stehen. Der Rest jedes Funktionsblocks ist mit deren Aufrufparametern belegt. Diese Länge variiert je nach Funktion.

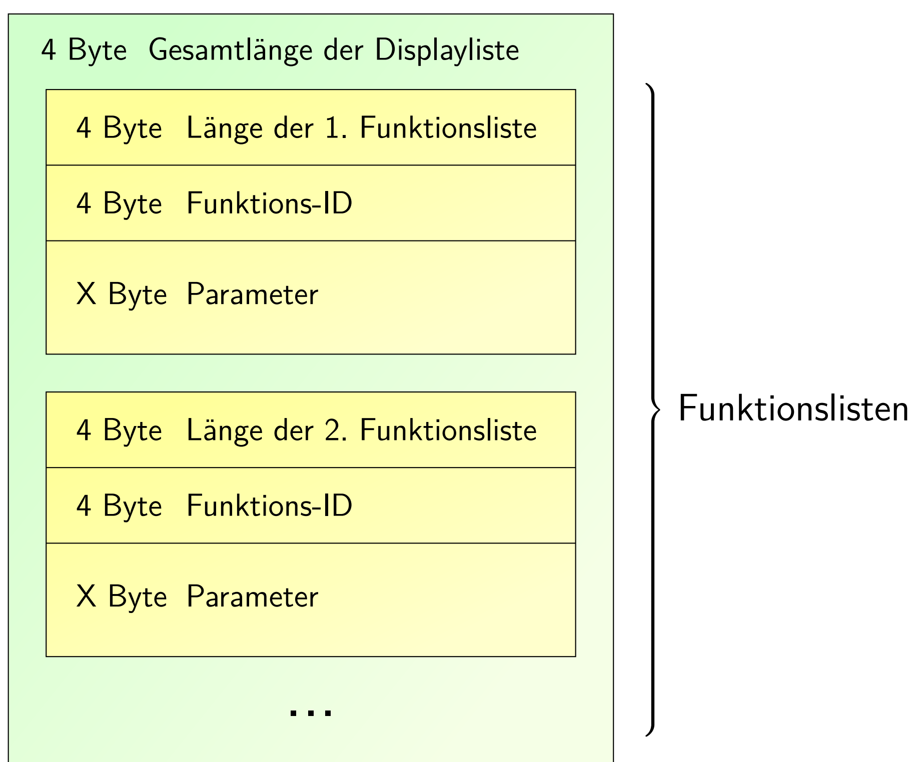


Abbildung 2.3: Schematischer Aufbau der Displayliste

2.2 Verwendung von Python

Es existieren bereits einige Ausgabetreiber, die jedoch alle in C, C++, Objective-C oder in nach C kompilierbaren Sprachen (z. B. *Vala*) geschrieben wurden. Im Rahmen dieser Bachelorarbeit soll ein HTML5[1]/JavaScript-Ausgabetreiber entwickelt werden, der die Grafik statisch in eine HTML-Datei einbettet und mithilfe von JavaScript visualisiert. Diese Vorgehensweise erfordert die Generierung von HTML- und JavaScript-Quellcode und somit auch Zeichenkettenverarbeitung. Die Skriptsprache Python bietet hierfür viele und performante Möglichkeiten. Zusätzlich beinhaltet die vielfältige Standardbibliothek auch ein Modul zum Laden und Aufrufen von C-Funktionen. So kann ein Python-Programm die Displayliste verarbeiten, ohne dass Änderungen am GKS vorgenommen werden müssen. Außerdem zeichnet sich Python durch eine schnelle Entwicklungszeit und gute Lesbarkeit aus. So werden Entwicklungstechniken wie Prototyping gefördert und Realisierungsprobleme werden frühzeitig erkennbar.

2.2.1 Aufbau der Module

Die Abbildung 2.4 beschreibt den modularen Aufbau des Treibers:

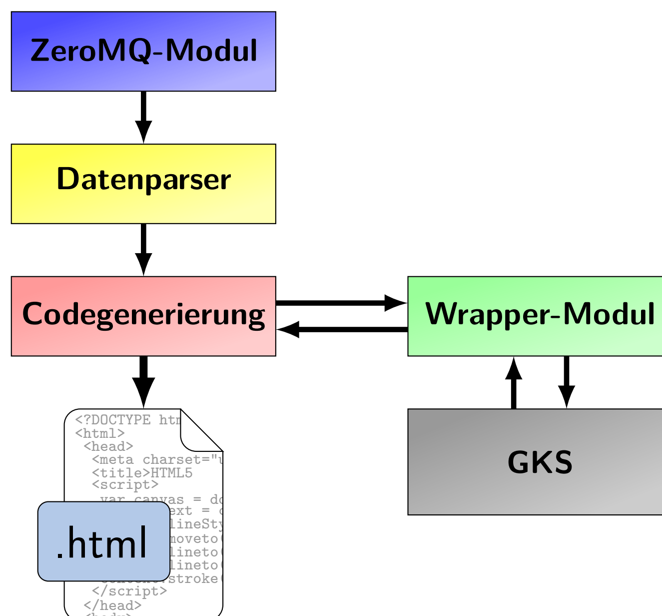


Abbildung 2.4: Module des Treibers und ihre Beziehungen

[1] *Hypertext Markup Language* – eine Auszeichnungssprache zur Strukturierung multimedialer Inhalten

Es folgt eine Auflistung der Funktionen der einzelnen Module:

ZeroMQ-Client	Daten empfangen und entpacken
Datenparser	Verarbeiten der Daten und Überführen in Python-Datenstrukturen
Codegenerierung	Erzeugung der HTML-Datei
Wrapper-Modul	Laden und Aufrufen der GKS-Funktionen

2.3 GKS-Datenparser

Das Datenparser-Modul hat die Aufgabe, die empfangene *Displayliste* zu verarbeiten. Dafür ist es nötig, die in Abbildung 2.3 dargestellte Struktur auszulesen. Das Modul arbeitet mit einem assoziativen Datentyp, der jeder Funktions-ID den entsprechenden Funktionsnamen und einen String mit den Parametertypen zuordnet. Die Parametertypen sind folgendermaßen kodiert:

Zeichen	Datentyp
`i`	Integer Wert
`I`	zweidimensionales Feld von Integer Werten
`f`	Float Wert
`F`	eindimensionales Feld von Float Werten
`s`	Zeichenkette

Sie wird elementweise nach dem Prinzip eines Zustandsautomaten aus dem Datenstrom gelesen. Für Integer- und Float-Werte werden die jeweils nächsten vier Byte eingelesen und direkt in den entsprechenden Datentyp umgewandelt. Handelt es sich jedoch um ein Feld, werden zusätzliche Informationen benötigt. Die Struktur der Parameterliste sieht vor, dass die Länge des Feldes als Integer vor dem Feld übergeben wird. Stehen mehrere Felder hintereinander, so haben diese alle die gleiche Länge.

Die zweidimensionalen Integer-Felder, die dazu genutzt werden Rastergrafiken zu kodieren, werden von zwei Integern eingeleitet, deren Produkt die Anzahl der enthaltenen Elemente darstellt. Aus diesen Werten lassen sich die Dimensionen des Bildes gewinnen. Zeichenketten werden immer mit einer festen Länge von 132 Zeichen übertragen und entsprechend ausgelesen. Längere Zeichenketten werden abgeschnitten und kürzere aufgefüllt. Folgende Tabelle zeigt einige Beispiele für Parameterlisten:

ID	Name	Format-String	Beispielparameter
52	`select_xform`	`i`	0
12	`polyline`	`iFF`	3, [1,2,3], [1,2,3]
16	`cellarray`	`ffffiiiI`	0.0, 0.5, 0.75, 0.5, 8, 10, 8, [0, 1, ..., 79]

2.4 *ctypes*-Wrapper

ctypes ist ein Python-Modul zum dynamischen Laden und Aufrufen von C-Funktionen aus Programmbibliotheken (*shared objects*, *dynamic link libraries*, ...). Daneben bietet es Möglichkeiten zur Erstellung, Verarbeitung und Manipulation von C-Datentypen. Im Rahmen des HTML5-Treibers wird das Modul verwendet, um C-Strukturen nachbilden und benutzen zu können. ctypes ermöglicht es auf diese Weise, Daten aus den in C geschriebenen Kernmodulen abzufragen.

Die vorliegende Implementierung des GKS stellt Emulationsroutinen für die gerätespezifische Umsetzung von Ausgabeprimitiven zur Verfügung. Sie emulieren bestimmte Grafikoperationen, wenn die Schnittstelle zu diesem Ausgabemedium eine einfache Umsetzung der gewünschten Operation nicht vorsieht. So können zum Beispiel bestimmte Schriften aus Linien und Füllmustern zusammengesetzt werden. Diese Routinen sind jedoch in C geschrieben und werden aus Python deshalb ebenfalls mithilfe von ctypes aufgerufen.

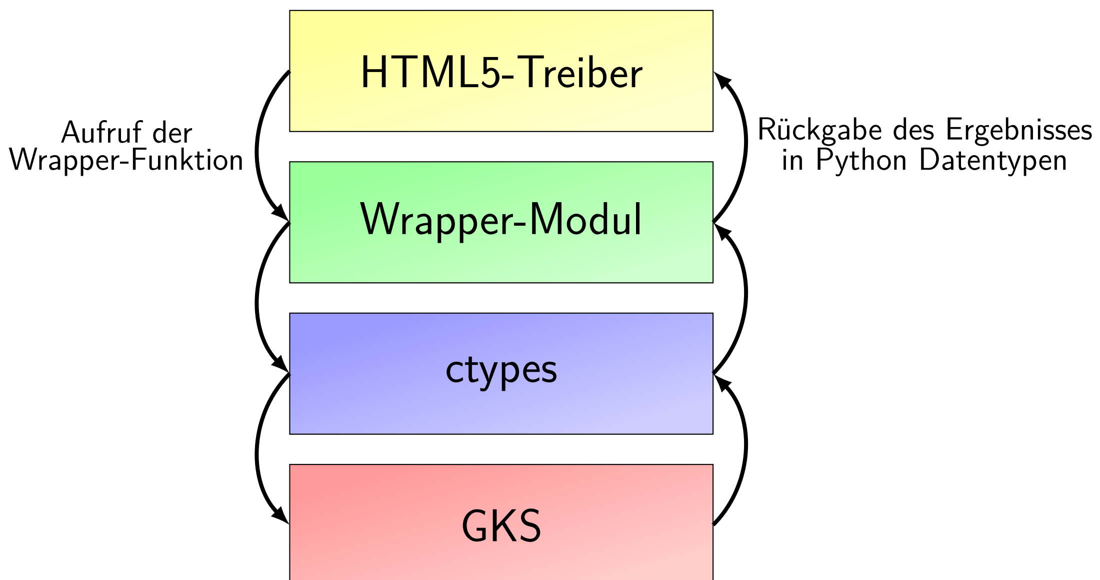


Abbildung 2.5: Aufrufkette der GKS-Routinen

Der Aufruf einer Wrapper Funktion durchläuft mehrere Ebenen (siehe Abbildung 2.5). Die oberste Ebene stellt der Treiber dar, der die GKS-Funktion nutzt. Darunter liegt das Wrapper-Modul zum Anpassen der Datentypen, welches dazu die von der Bibliothek ctypes bereitgestellten Datentypen verwendet. Nach der Anpassung wird schließlich die gewünschten C-Routine aus ctypes aufgerufen.

2.4.1 Einfache Funktionen und Standarddatentypen

Üblicherweise umfasst das Erstellen einer Python-Methode zum Aufruf einer C-Funktion folgende fünf Schritte:

1. Beschreibung der Argumentliste der C-Funktion mit C-kompatiblen Datentypen (*Handler)*
2. Definition der Schnittstelle der Python-Funktion (Name, Parameter)
3. Umwandeln/Erstellen der benötigten Parameter für die aufzurufende C-Funktion
4. Aufruf der Funktion
5. Verarbeiten der Rückgabewerte und eventuell Rückgabe der verarbeiteten Werte als Python-Datentyp

Im Folgenden ein Beispiel zur Abfrage des RGB-Farbwertes anhand eines *color indices*:

```
1  _gks.gks_inq_rgb.argtypes = (c_int,
2                               POINTER(c_float),
3                               POINTER(c_float),
4                               POINTER(c_float) )
5
6  def inq_rgb(color_index):
7      global _gks
8      color = [c_float(0) for i in range(3)]
9      _gks.gks_inq_rgb(c_int(color_index), byref(color[0]), byref(
           color[1]), byref(color[2]))
10     return [c.value for c in color]
```

Die erste Zeile legt die Argumenttypen der C-Funktion fest. Die darauf folgende Zeile legt fest, dass die Variable `_gks` aus dem globalen Namensraum verwendet werden soll. Sie repräsentiert die dynamisch geladene C-GKS-Bibliothek und somit auch die aufzurufende Funktion. Danach wird eine Liste von `c_float`-Objekten erstellt und die Funktion mit dem `color_index` und Pointern auf die soeben erstellten Floats aufgerufen. Nach dem Aufruf werden den Elementen in der Liste die gewünschten Werte

zugewiesen. Schließlich wird aus diesen noch eine Liste von Python-Floats erstellt und zurückgegeben.

Auf diese Weise lassen sich C-Funktionen mit Standarddatentypen als Parameter aufrufen und die Ergebnisse verarbeiten. Müssen jedoch selbstdefinierte Strukturen oder Callback-Routinen[2] verwendet werden, benötigt man andere Konstrukte der Bibliothek ctypes.

2.4.2 Nachbilden der GKS-Strukturen

Jede C-Struktur, die vom Treiber genutzt wird, muss im Wrapper-Modul neu definiert werden. Dazu muss eine neue Klasse erstellt werden, die von der Klasse `Structure` abgeleitet wird. Die Variable `_fields_` wird mit den Namen und Datentypen der gewünschten Strukturelemente belegt:

```
1 class state_list(Structure):
2     _fields_ = [
3         ('lindex', c_int),
4         ('ltype', c_int),
5         ('lwidth', c_float),
6         ...
7         ]
```

Nun können Instanzen der Klasse verwendet werden, als hätten sie die Variablen der C-Struktur als Attribut:

```
1 state_list = gks.state_list()
2 state_list.lindex = 5
3 ...
```

Das Python-Objekt enthält einen zu der C-Struktur binär identischen Speicherbereich. Deshalb ist es beispielsweise mit der von ctypes bereitgestellten Routine `memmove` möglich, komplette C-Strukturen in das Wrapper-Objekt zu kopieren. Dabei werden die Variablen automatisch richtig zugewiesen.

2.4.3 Callback-Funktionen

Beim Aufruf von Funktionen mit Callback definiert man eine entsprechende Python-Routine, die von der Funktion aufgerufen werden soll. Diese erwartet Python-Parameter und ist aus C-Funktionen heraus nicht aufrufbar. Daher muss ein C-Funktionstyp

[2] Callback-Routinen werden anderen Funktionen als Funktionspointer übergeben und von diesen aufgerufen. Hierfür ist es wichtig, dass die aufgerufene Funktion die Schnittstelle der übergebenen Funktion kennt.

(CFUNCTYPE) definiert werden, der die Schnittstelle für den Aufruf und die Rückgabe darstellt. Der erste Parameter beim Erstellen des CFUNCTYPE ist der Rückgabewert. Die restlichen Parameter entsprechen den Argumenten der Python-Funktion. Dieser Typ wird genutzt, um aus der Callback-Routine in Python eine C-Funktion mit der festgelegten Schnittstelle zu erzeugen. Schließlich wird noch eine Wrapper-Funktion für die Routine mit Callback verwendet. Hierbei werden die vorher definierten Funktionstypen als Parametertypen angegeben.

```
1  fill_functype = CFUNCTYPE(c_void_p,
2                            c_int,
3                            POINTER(c_float),
4                            POINTER(c_float),
5                            c_int )
6  line_functype = CFUNCTYPE(c_void_p,
7                            c_int,
8                            POINTER(c_float),
9                            POINTER(c_float),
10                           c_int,
11                           c_int )
12
13 fill_callback = fill_functype(fill_routine)
14 line_callback = line_functype(line_routine)
15
16 _gks.gks_emul_text.argtypes = (c_float, c_float, c_int, c_char_p,
       line_functype, fill_functype)
17 def emul_text(x_pos, y_pos, text):
18     _gks.gks_emul_text(c_float(x_pos), c_float(y_pos), c_int(len(
           text)), c_char_p(text), line_callback, fill_callback)
```

3 ZeroMQ

Die Bibliothek ZeroMQ (auch ØMQ) dient der Nachrichtenübermittlung und stellt hierfür einige Möglichkeiten zur Verfügung. Diese sollen zunächst erläutert werden, um danach die Anwendung auf das GKS zu veranschaulichen.

3.1 Grundlagen

ZeroMQ ist eine *messaging library*, die eine vereinfachte, Schnittstelle zur Socketkommunikation bietet und Bindings für fünfzehn verschiedene Programmiersprachen (z. B. Python, C, C++) hat. Sie zeichnet sich durch folgende Eigenschaften aus [12]:

- hohe Performanz durch *message batching*[1]
- Einfachheit
- hohe Skalierbarkeit

Die hohe Skalierbarkeit wird dadurch erreicht, dass ZeroMQ eigene Sockets zur Verfügung stellt, die sich im Gegensatz zu gewöhnlichen Sockets mit mehreren Endpunkten verbinden können. Dabei wird automatisch eine Art Lastbalancierung der Nachrichten durchgeführt. Außerdem können so auch Daten von mehreren Datenquellen über einen Socket eingelesen werden.

Die Möglichkeit Daten über einen Socket an mehrere Empfänger zu verteilen, kann vom GKS sinnvoll genutzt werden, um die Displaylisten gleichzeitig an verschiedene Empfänger zu senden. Dabei ist es einerseits möglich die Daten parallel mit verschiedenen Ausgabemedien auszugeben. Andererseits können so die Daten an beliebige andere Endgeräte gesendet werden, die über das Netzwerk erreichbar sind, um somit die Grafiken auch über Standorte hinweg zu verteilen. Diese Datenverteilung wird in Abbildung 3.1 dargestellt.

[1] Zusammenfassen mehrerer Nachrichten, um die erforderlichen Systemaufrufe zu minimieren.

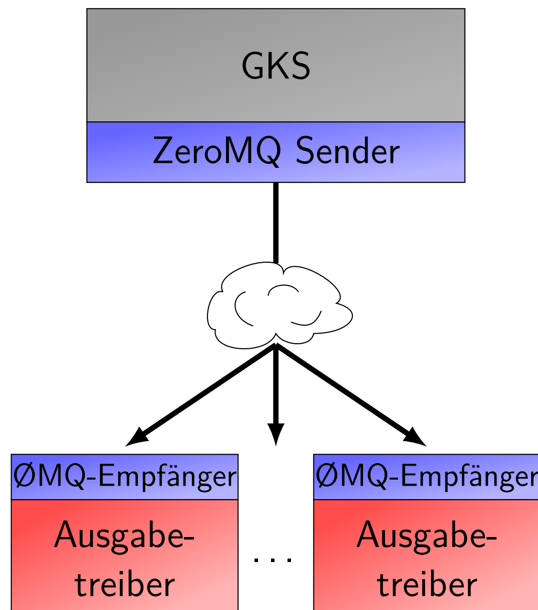


Abbildung 3.1: Gewünschte 1:N Kommunikation zwischen GKS und Ausgabetreiber

3.2 Publish-Subscribe

ZeroMQ stellt für verschiedene Anwendungszwecke Kommunikationsmuster (*messaging patterns*) zur Verfügung. Diese sind auf das entsprechende Anwendungsgebiet optimiert und stellen Socketpaare für die Kommunikation bereit. Publish-Subscribe gehört zu diesen Kommunikationsmustern und wird durch die Sockets `zmq.PUB` als *Publisher* und `zmq.SUB` als *Subscriber* repräsentiert.

```
1  import zmq
2
3  #  Socket to talk to server
4  context = zmq.Context()
5  socket = context.socket(zmq.SUB)
6
7  socket.connect ("tcp://localhost:5556")
8  subscription = ''
9  socket.setsockopt(zmq.SUBSCRIBE, subscription)
10
11 string = socket.recv()
12 print string
```

Die Subscriber verbinden sich mit dem Publisher und empfangen alle Nachrichten, die der Publisher versendet. Dafür verbinden sie sich mittels `connect` und legen eine

subscription fest, die als Filter für eingehende Nachrichten verwendet wird. Danach kann die Nachricht mit `recv` empfangen werden. Das obige Beispiel verwendet keinen Filter und empfängt die nächste Nachricht des Publishers und gibt diese aus.

```
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5556")

while True:
    temperature = random.randrange(1,215) - 80
    relhumidity = random.randrange(1,50) + 10

    socket.send("%d␣%d" % (temperature, relhumidity))
```

Dieser Publisher sendet beispielsweise unbegrenzt zufällige „Wetterdaten“ an die Subscriber, die mit dem Port 5556 verbunden sind. Die Funktion `bind` stellt einen Verbindungsendpunkt zur Verfügung und bindet ankommende Verbindungsanfragen an den aufrufenden Socket.

Dieser Ansatz wurde implementiert und getestet. Dabei zeigte sich, dass die ersten Nachrichten des Publishers nicht ankommen und ein Synchronisationsmechanismus geschrieben werden muss, der dafür sorgt, dass Daten erst versendet werden, sobald die Subscriber bereit sind diese zu empfangen. Dafür wurde im Voraus eine feste Anzahl von Subscribern festgelegt, die vorhanden sein muss, damit mit dem Senden der Daten begonnen wird. Zur Synchronisierung wird ein Socket des Typs Request-Reply verwendet, über den die Synchronisierungsnachricht geschickt wird. Auf diese Weise kamen die Nachrichten an, jedoch nur etwa 80%. Da das GKS eine zuverlässige Kommunikation benötigt, wurde ein anderer Ansatz gewählt.

3.3 Push-Pull

Das Push-Pull-Pattern wird ebenfalls zur Datenverteilung verwendet. Hierbei werden die Nachrichten jedoch nicht kopiert und an alle versendet, sondern gleichmäßig auf die Empfänger aufgeteilt. So erhält jeder Pull-Socket einen Nachrichtenblock. Nach ersten Tests stellte sich heraus, dass diese Verteilung der Daten jedoch nur innerhalb eines Prozesses vorgenommen werden kann und hauptsächlich in der Parallelprogrammierung genutzt wird, um den Workern Arbeit zuzuweisen (*Intraprozesskommunikation*). Somit ließ sich nur eine zuverlässige Punkt-zu-Punkt-Verbindung mit den Sockets `zmq.PUSH` und `zmq.PULL` herstellen, da es bislang nicht möglich ist explizit über mehrere Sockets Verbindungen zu jedem Empfänger herzustellen. Deshalb sieht die Kommunikation zwischen dem GKS und ZeroMQ wie folgt aus:

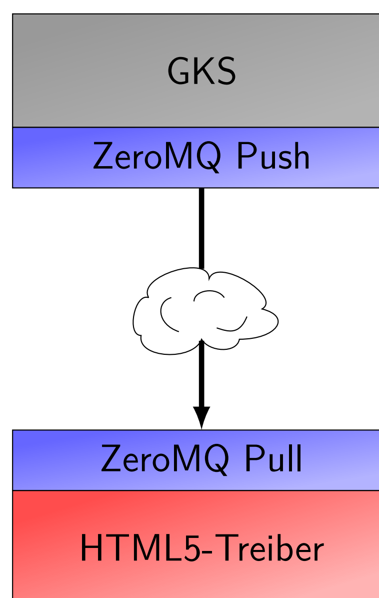


Abbildung 3.2: 1:1 Kommunikation mit dem Push-Pull-Pattern

Nach dem Verbindungsaufbau schickt das GKS die Daten an den Pull-Socket, der diese dann direkt an den HTML5-Treiber weiterleitet.

4 JavaScript und HTML5

Das folgende Kapitel beschäftigt sich mit den theoretischen Grundlagen und der Realisierung der eigentlichen Grafikfunktionen.

4.1 Grundlagen

Zunächst werden kurz die Auszeichnungssprache HTML5 und die Skriptsprache JavaScript vorgestellt. Sie dienen der Darstellung des Canvas und der Umsetzung der Zeichenroutinen.

4.1.1 HTML

HTML steht für *Hypertext Markup Language* und bezeichnet eine textbasierte Auszeichnungssprache zur Strukturierung von Webseiten. Es bietet die Möglichkeit, Webseiten in semantische Bereiche einzuteilen und verschiedene Inhalte einzubinden. Zu diesen Inhalten gehören:

1. Text
2. Hyperlinks (Verweise auf andere Webseiten)
3. Bilder

Das Einfügen und Strukturieren geschieht mit Auszeichnungselementen (*Tags*), denen bestimmte Eigenschaften und Inhalte zugewiesen werden können. Sie bestehen aus einem öffnenden und schließenden Tag, die den Inhalt des Elementes begrenzen. Ein Ausnahme bildet beispielsweise der inhaltslose Zeilenumbruch `<br />`, der mit dem Öffnen direkt geschlossen wird. Tags können auch geschachtelt werden, was zu einer hierarchischen Gliederung führt. Hierbei ist zu beachten, dass der Inhalt eines Unterelementes abgeschlossen sein muss bevor das Oberelement geschlossen werden kann. Daher lässt sich das HTML-Dokument in eine Baumstruktur überführen.

Die Darstellung von HTML-Seiten wird von Browsern übernommen. Sie interpretieren den Code aus der *.html*-Quelltextdatei und stellen ihn dar. Layout und Formatierung eines solchen HTML-Dokumentes können mit *CSS*[1] angepasst werden.

[1] *Cascading Style Sheets* ist eine Formatierungssprache zur optischen Gestaltung von strukturierten Dokumenten [9].

HTML-Webseiten sind in erster Linie statisch, lassen sich jedoch mit Erweiterungen auch dynamisch gestalten. Zu diesem Zweck wird JavaScript verwendet. So können auch Interaktionen mit dem Benutzer realisiert und umgesetzt werden. Das `<canvas>`-Element aus dem aktuellen HTML5-Standard wird als Zeichenfläche für die grafischen Primitiven verwendet. Deshalb wird nun näher auf die neuen Funktionen von HTML5 und das Canvas eingegangen.

```
1 <!DOCTYPE html>
2 <html>
3   <body>
4     <h1>My First Heading</h1>
5     <p>My first paragraph.</p>
6   </body>
7 </html>
```

Listing 4.1: HTML Beispieldatei[14]

4.1.1.1 HTML5

HTML5 ist der neuste Entwurf von HTML. Er soll 2014 offiziell verabschiedet werden und befindet sich im Entwicklungsstadium *Canditate Recommendation*[2]. Die Hauptziele von HTML5 sind [16]:

- weniger Bedarf an externen Plugins (z. B. Flash)
- mehr Auszeichnungselemente, um die Verwendung von Skriptsprachen zu reduzieren
- Geräteunabhängigkeit

Diese Zielsetzung resultiert in einem deutlich größeren Funktionsumfang verglichen mit den Vorgängerversionen. Zu den neuen Funktionen gehören:

- vereinfachte Einbettung von multimedialen Inhalten mittels `<video>` und `<audio>`
- CSS3 Unterstützung zur verbesserten Layouterstellung
- Erstellen von 2D-Grafiken mithilfe
  - des neuen `<canvas>`-Elements (Rastergrafik)
  - von eingebettetem SVG[3] (Vektorgrafik)

[2] Die Entwicklungsstadien einer W3C-Empfehlung sind: *Working Draft* → *Candidate Recommendation* → *Proposed Recommendation* → *Recommendation*

[3] *Scalable Vektor Graphics* ist eine Spezifikation zur Beschreibung von zweidimensionalen Vektorgrafiken

- neue, spezifischere Strukturelemente
- Bereitstellung von lokalem Datenspeicher

HTML5 ist noch kein offizieller Standard und es gibt noch keinen Browser, der die komplette Funktionalität implementiert hat. Die bekanntesten Browser (Safari, Chrome, Firefox, Internet Explorer, ...) werden jedoch in Richtung HTML5 weiterentwickelt. Die momentane Unterstützung der HTML5-Funktionen der verschiedenen Browser variiert recht stark [18]:

Browser	**Bewertung**
Chrome 28	463
Opera 15	423
Firefox 22	410
Safari 6	378
Internet Explorer 10	320

Tabelle 4.1: Bewertung der HTML5-Unterstützung der Browser auf einer Skala von 1 bis 500

Das neu eingeführte `<canvas>`-Element wird als Zeichenfläche verwendet. Das Zeichnen auf dieser wird durch Programmierung mit JavaScript erreicht, welches im Folgenden vorgestellt wird.

4.1.2 JavaScript

JavaScript ist eine objektorientierte Skriptsprache, die hauptsächlich dazu verwendet wird, dynamische Webseiten zu realisieren. Der im Webbrowser eingebettete JavaScript-Interpreter ermöglicht das Modifizieren des HTML-Dokumentes. Hierbei kann auf bestimmte Browsereigenschaften bzw. Benutzereingaben reagiert werden. Diese Reaktion erfolgt auf Clientseite und erfordert somit keine Ressourcen auf dem Server (*Clientseitiges JavaScript*). Ein weiteres Merkmal von JavaScript ist die dynamische Typisierung. Durch sie muss der Typ von Variablen nicht angegeben werden und kann sich zur Laufzeit ändern.

Das nachfolgende Beispiel veranschaulicht die Einbettung von JavaScript in HTML:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <script>
5     function displayDate()
6     {
```

```
7       document.getElementById("demo").innerHTML=Date();
8     }
9   </script>
10 </head>
11 <body>
12   <h1>My First Java\-Script</h1>
13   <p id="demo">This is a paragraph.</p>
14   <button type="button" onclick="displayDate()">Display</button>
15 </body>
16 </html>
```

Listing 4.2: HTML-JavaScript Interaktionsbeispiel[15]

Der JavaScript-Code innerhalb einer HTML-Seite wird mit dem `<script>`-Tag eingeleitet. In diesem Beispiel wird eine Funktion `displayDate` definiert, die das Element „demo“ abfragt und den Inhalt des Elements auf das aktuelle Datum setzt. Zur dynamischen Abfrage des Elements wird die Funktion `getElementById` genutzt. Mit dieser Funktion kann jedes Element anhand einer vorher definierten Id abgefragt werden. Dieses Beispiel beinhaltet – neben den Tags `<h1>` und `<p>` für eine große Überschrift (*headline*) und einen Absatz (*paragraph*) – einen `<button>`-Tag. Da Buttons weniger zur grafischen Strukturierung als zur Interaktionsmöglichkeit eingesetzt werden, wird mit dem Attribut `onclick` eine Callback-Funktion definiert, die bei Knopfdruck aufgerufen wird. So können Routinen festgelegt werden, die eine Benutzerinteraktion erlauben.

Wie bereits in Kapitel 4.1.1.1 erwähnt, wird JavaScript zur Realisierung der Zeichenroutinen verwendet. Dazu wird die JavaScript-Zeichenschnittstelle des Canvas genutzt. Auf diese Weise können die Grafikelemente dargestellt und benötigte, browserspezifische Anpassungen durchgeführt werden.

4.2 <canvas>-Element

Im Folgenden wird die Funktionalität des `<canvas>`-Elementes beschrieben. Der Begriff Canvas bezeichnet einen Bereich mit fest definierten Ausmaßen, der als Container für die JavaScript-Zeichenbefehle dient. Die Erzeugung der Grafiken und das Zeichnen werden von einem `CanvasRenderingContext2D` durchgeführt. Dieser wurde in einem Zug mit dem Canvas entwickelt, stellt aber nicht die einzige Anwendungsmöglichkeit dar, da es mit WebGL möglich ist 3D-Grafiken auf das Canvas zu zeichnen.

Der Kontext verhält sich wie ein Stift. Er wird zum Zeichnen auf dem Canvas verwendet und enthält die Attribute der potentiell zu zeichnenden, jedoch keine Informationen über bereits dargestellte Elemente. Er wird dazu genutzt, das Canvas zu bearbeiten. Das Canvas speichert die Änderungen und stellt sie dar. Dieser Zusammenhang ist in Abbildung 4.1 veranschaulicht.

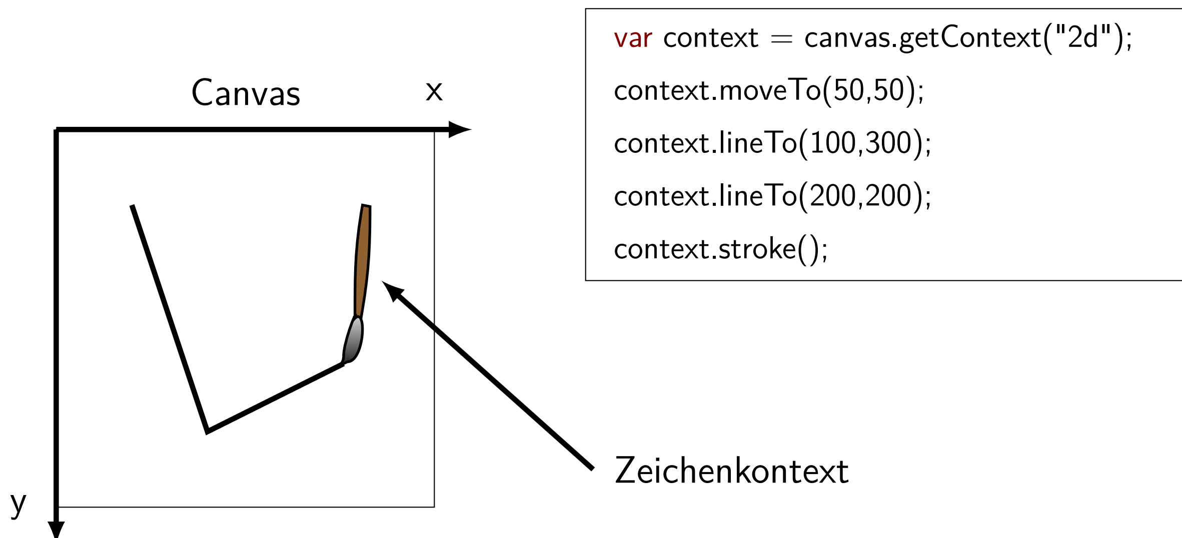


Abbildung 4.1: Zusammenspiel zwischen Canvas und Zeichenkontext

Zum Funktionsumfang eines solchen Kontextes gehören:

- Linienzüge
- Kreisbögen und Bézierkurven (quadratische und kubische)
- Füllflächen
- Farbverläufe und Füllmuster
- Rastergrafiken, die zusätzlich bearbeitet werden können
- Transparenz
- Text

Beim Aufruf der Zeichenroutinen ist zu beachten, dass sie dadurch nur definiert werden. Der Aufruf von `lineTo` sorgt folglich noch nicht dafür, dass eine Linie gezeichnet wird. Das Zeichnen der Linie wird durch `stroke` erreicht. Ebenso werden definierte Polygone bzw. geschlossene Pfade mit `fill` ausgefüllt und dargestellt.

Die oben erwähnten Elemente können auf dem Canvas beliebig positioniert werden. Dazu ist es möglich, Skalierungen, Verschiebungen und Rotationen anzugeben, die auf die Zeichenelemente angewendet werden sollen. Weiterhin ist zu beachten, dass das Ergebnis des Zeichenvorgangs eine Rastergrafik mit den Dimensionen der Zeichenfläche ist. Folglich ist die Qualität an die Größe des Canvas gebunden und kann mit steigender Canvasgröße verbessert werden.

4.3 Zeichenroutinen

In den Kapiteln 2 und 3 wurde beschrieben, wie die ankommenden Daten verarbeitet und für den Zeichenvorgang transformiert werden. Die Hauptaufgabe des Treibers besteht jedoch aus dem tatsächlichen Zeichnen der GKS-Primitiven. Deshalb sollen diese nun vorgestellt und die Umsetzung erläutert werden. Die zu realisierenden Grafikfunktionen sind:

Polylines Folge von Linien

Fill Area Füllen von Flächen mit einer Schraffur, einer Farbe oder einem Muster

Cell Array Darstellen von Farbmatrizen (Rastergrafiken)

Polymarker Symbole (z. B.: Kreise, Sterne, ...)

Text Schrift

4.3.1 Polyline

Eine Polyline ist eine Aneinanderreihung von mehreren Geraden. Beschrieben wird dieser Linienzug durch die Punkte, die mit Geraden verbunden werden. Die Anzahl der Punkte beträgt mindestens zwei und kann theoretisch beliebig groß werden. Die zu zeichnenden Geraden haben bestimmte Eigenschaften:

- Linienbreite
- Linienfarbe
- Linienmuster

Diese Eigenschaften lassen sich über die Attribute des Kontextes regulieren, da dieser alle nötigen Eigenschaften für das Zeichnen enthält. Die Linienbreite lässt sich über die `lineWidth` einstellen. Der zugewiesene Wert muss eine Gleitkommazahl sein.

Die Farbe der Linie wird in dem Attribut `strokeStyle` gespeichert und kann auf drei Arten angegeben werden:

1. hexadezimal im `#RRGGBB`-Format
2. dezimal in der Form `rgb(RRR,GGG,BBB)`
3. als Farbname (z. B. „`aqua`")

Das Linienmuster gibt die Strichelung der Linie an. Es wird mit einer Liste aus ganzzahligen Werten festgelegt. Dabei bezeichnet jede ganze Zahl die Anzahl der schwarzen bzw. weißen Pixel. Das Muster `[6, 1, 3, 2]` ergibt die Strichelung aus Abbildung 4.2. Die Art der Strichelung kann vom Anwender jedoch nicht frei gewählt werden. Er hat die Auswahl aus einer vorgegebenen Menge von Mustern, die vom GKS bereitgestellt wird.

Sind schließlich alle Zeicheneigenschaften gesetzt, ändert man mit `moveTo` die Position auf den ersten Punkt des Linienzugs und definiert mit `lineTo` die einzelnen Linien. Nach dem Aufruf von `stroke` werden diese auf die Zeichenfläche gezeichnet.

Abbildung 4.2: Linienmuster am Beispiel `[6, 1, 3, 2]`

4.3.2 Fillarea

Das Zeichnen von Füllflächen übernimmt die `fillarea`-Funktion. Sie erwartet ebenso wie die `polyline`-Funktion eine Menge von Punkten, die die Eckpunkte des zu füllenden Polygons bilden. Grundlegend unterscheidet das GKS beim Zeichnen von Füllflächen zwischen vier Stilen (Abbildung 4.3):

1. umrahmte Fläche, die nicht gefüllt wird (`hollow`)
2. gefüllte Fläche (`solid`)
3. gemusterte Fläche (`pattern`)
4. schraffierte Fläche (`hatch`)

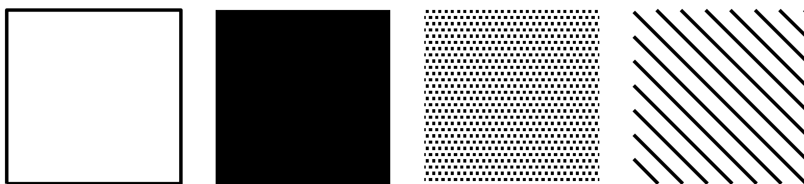


Abbildung 4.3: Fillarea Beispiele (v. l. n. r.: `hollow`, `solid`, `pattern`, `hatch`)

Soll das Polygon einen Rahmen haben, wird die Methode `stroke` aufgerufen und der Pfad zum Festlegen des Polygons wird gezeichnet. Die gefüllte Fläche wird, wie bereits in Abschnitt 4.2 erwähnt, über den Aufruf von `fill` erreicht. Standardmäßig werden Flächen ohne Muster gefüllt. Es kann jedoch ein Muster festgelegt werden.

In HTML5 werden Muster normalerweise über Bilddateien (PNG, JPEG, ...) angegeben. Die vom GKS bereitgestellten Muster bestehen aus einem 8×8 Pixel großen Binärbild[4] (Abbildung 4.4), das wiederholt auf die ganze Füllfläche angewendet wird. Das Muster wird mittels Wrapperfunktion vom GKS abgefragt und in Form einer Liste von 8 Integern zurückgegeben. Jeder Wert repräsentiert eine Zeile des Musters. In der binären Darstellung des Integers entspricht jede Stelle einem Pixel. Ist das Bit an der Stelle des Integers 0, wird der Pixel in der aktuell gesetzen Füllfarbe gezeichnet. Die Binärdarstellung wird dabei vom niedrigstwertigen Bit an durchlaufen.

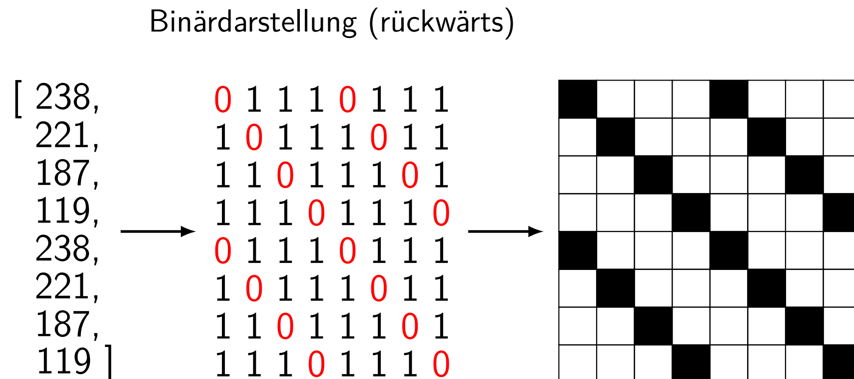


Abbildung 4.4: Erstellung eines Fillarea-Musters

Dieses Muster wird an JavaScript weitergereicht, um daraus ein sogenanntes *Pattern* zu erstellen. Dafür wird zunächst ein temporäres Canvas erstellt und die Größe auf die des Musters gesetzt. Daraufhin werden die entsprechenden Pixel mittels `rect` gefüllt und schließlich ein Pattern (`createPattern`) aus diesem Canvas erzeugt, das als `fillStyle` gesetzt werden kann.

4.3.3 Polymarker

Polymarker (auch Markersymbole genannt) sind Symbole zur Markierung eines bestimmten Punktes [6]. Sie werden intern aus anderen grafischen Primitiven zusammengesetzt. Es gibt eine fest definierte Menge von Markersymbolen, die vom GKS zur Verfügung gestellt wird (siehe Abbildung 4.5).

Der Anwender kann die Größe, die Farbe und den Markertyp festlegen. Das Zeichnen erfolgt mithilfe einer sogenannten Markerliste, welche für jeden Markentypen eine Liste von Daten bzw. *Operations-Ids* enthält, die für den Zeichenvorgang benötigt werden. Die Operations-Ids repräsentieren eine grafische Primitive, aus denen

[4] Binärbilder (Bitmaps) sind Rastergrafiken, die für jeden Pixel nur die Farbe Schwarz oder Weiß kennen.

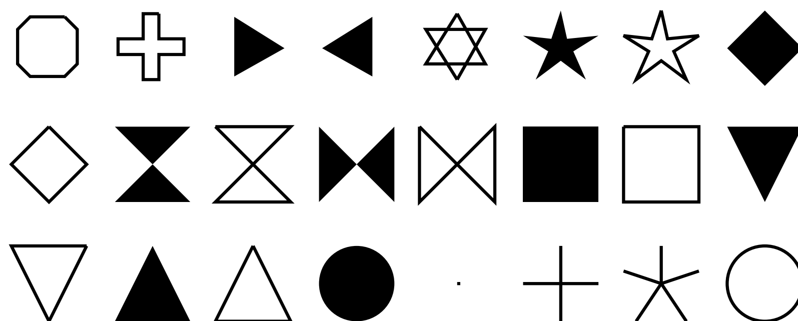


Abbildung 4.5: Beispiele für Markersymbole

der Polymarker zusammengesetzt wird. Dabei werden die folgenden vier grafischen Primitive verwendet:

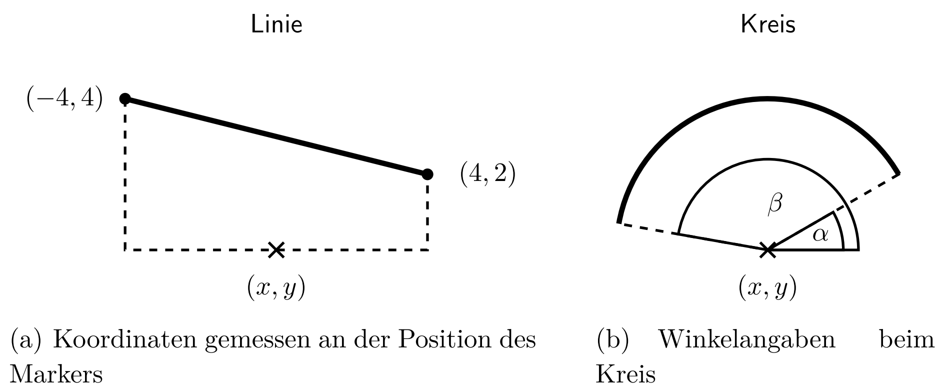
Punkte, Linien, Polygone, Kreisbögen

Polygone und Kreise bekommen zusätzlich zwei weitere Nummern, um die bekannten Stile `solid`, `hollow` und `filled` zu realisieren. Die Verarbeitung der Liste geschieht nach dem Prinzip eines Zustandsautomaten. Zu Beginn steht die Operations-Id (bzw. Zustand) gefolgt von den für diese Operation benötigten s Zusatzinformationen. Daher wird im Folgenden der Aufbau der Teillisten dargestellt.

Operation	**benötigte Argumente**
Punkt	-
Linie	x_1, y_1, x_2, y_2
Polygon	n (Anzahl der Eckpunkte), x_1, y_1, ..., x_n, y_n
Kreisbogen	α (Startwinkel), β (Endwinkel)

Diese Informationen stehen in obiger Reihenfolge nach dem Operator in der Markerliste. Alle Angaben sind relativ zu dem Punkt zu betrachten, an dem der Marker gezeichnet werden soll (siehe Abbildung 4.6(a)). Der aktuelle Aufbau der Markerliste lässt nur Punkte in der Mitte des Markers zu (keine zusätzlichen Informationen). Ebenso haben Kreisbögen denselben Punkt als Mittelpunkt und einen Radius, der sich aus der gewünschten Markergröße berechnet. Deshalb müssen nur der Start- und Endwinkel des Bogens angegeben werden (siehe Abbildung 4.6(b)). Kreisbögen werden in JavaScript mit der Funktion `arc` gezeichnet. Sie erwartet den Mittelpunkt, den Radius und die beiden Winkel, um den Kreisausschnitt festzulegen.

Nach dem Zeichnen der Primitive wird mit dem nächsten Element in der Markerliste fortgefahren. Handelt es sich um eine weitere Operations-Id, wird mit einem neuen Element begonnen. Anderenfalls ist der Polymarker fertiggestellt.



(a) Koordinaten gemessen an der Position des Markers

(b) Winkelangaben beim Kreis

Abbildung 4.6: Relative Koordinatenangaben

4.3.4 Cellarray

Zur Darstellung einer eingebetteten Rastergrafik (auch Farbmatrix oder Cellarray genannt, Abbildung 4.7) wird ein Image-Objekt und die Funktion `drawImage` benutzt.

Abbildung 4.7: Cellarray Beispiel

Die Parameter x_{min}, x_{max}, y_{min} und y_{max} legen den Bereich fest, in den die Grafik eingefügt wird. Für jeden der vier Werte können beliebige Koordinaten auf der Zeichenfläche angegeben werden. dx und dy geben die Anzahl der Zellen der zu zeichnenden Farbmatrix in x- bzw. y-Richtung an. Da bei der Angabe der x- und y-Werte für die Positionierung keine weiteren Einschränkungen vorliegen, muss zwischen zwei Sonderfällen unterschieden werden:

1. Seitenverhältnis des Zielbereichs entspricht nicht dem Ursprungsseitenverhältnis $\Rightarrow$ Grafik wird gestreckt bzw. gestaucht

2. Mindestens eine Maximalkoordinate ($x_{\max}$ bzw. $y_{\max}$) ist kleiner als die dazugehörige Minimalkoordinate
   ⇒ Grafik wird gespiegelt

Die Bilddaten werden als Integer-Liste im Parameter `colia` übergeben. Dabei ist zu berücksichtigen, dass diese Werte unterschiedliche Bedeutungen haben können. Der letzte Parameter `true_color` legt fest, ob die Integer-Werte als Index in einer Farbtabelle oder direkt als Farbe zu interpretieren sind. In letzterem Fall wird jeder Integer in 4 Byte zerlegt und jedes Byte getrennt als Wert für die jeweilige Farbkomponente angesehen (siehe Abbildung 4.8). Hierbei wird eine mögliche vierte Komponente für die Blickdichtigkeit (*Alpha*-Wert) nicht berücksichtigt, da diese von der Implementierung des GKS bisher nicht unterstützt wird.

Dezimal:			3310335
Binär:	00110010	10000010	11111111
	B: 50	G: 130	R: 255

Abbildung 4.8: Auslesen der RGB-Werte aus einem Integer

Die ausgelesenen Bilddaten müssen geeignet in der HTML-Datei gespeichert werden. Deshalb werden die Daten in das verlustfreie Grafikformat PNG überführt. Die so kodierten Daten werden in einer Zeichenkette gespeichert. Da die Zeichenkette auch Zeichen beinhalten kann, die je nach betriebssystemabhängigen Zeichensatz andere Zeichencodes haben, wird sie zusätzlich mit dem `Base64`-Verfahren kodiert. Dieses Verfahren verwendet zur Kodierung die ASCII-Zeichen „A–Z“, „a–z“, „0–9“, „+“und „/“. Die Zeichen sind Codepage[5]-unabhängig und werden auf jedem System gleich interpretiert. Um die kodierten Bilddaten in den Quellcode einzubetten und für JavaScript nutzbar zu machen, wird eine Data-URL[6] verwendet:

Allgemeiner Aufbau einer Data-URL:

`data:[<Datentyp[7]>][;base64],<Daten>`

Base64 kodierte PNG-Grafik:

```
data:image/png;base64,<Bilddaten>
```

Auf diese Weise wird Auswertung und Entschlüsselung der Daten von HTML übernommen.

[5] Codepage oder Zeichensatztabelle – Zuordnungstabelle von Zahl zu Zeichen

[6] eine URI (Uniform Resource Identifier), die es ermöglicht, Daten so einzubetten, als lägen sie in externen Dateien [10].

[7] Der Datentyp wird als MIME-Typ angegeben, um die Daten zu klassifizieren. Er besteht aus dem Medientyp (z. B. *image*) und dem Subtyp (z. B. *png*)[4].

RGBA → PNG → base64 → URI

Abbildung 4.9: Verarbeitung der Bilddaten

Die Abbildung 4.9 stellt die Verarbeitungsreihenfolge der Bilddaten dar.

Der HTML5-Standard zu eingebetteten Inhalten [20] sieht vor, dass Grafiken asynchron in die Webseite geladen werden. Folglich können die restlichen Elemente der Webseite verwendet werden, obwohl die Grafik noch aufgebaut wird. Diese Asynchronität wird über einen Callback-Mechanismus erreicht. Man definiert für das Image-Objekt eine Funktion `onload`, die beim Laden der Grafik aufgerufen wird. Das führt dazu, dass die Grafiken der Seite als letztes gezeichnet und Elemente, die über dieser Grafik liegen sollen, verdeckt werden. Deshalb wird die Verarbeitung der grafischen Elemente, die nach dem Cellarray dargestellt werden, in die `onload`-Funktion hinter den Aufruf von `drawImage` verschoben. Bei der Verwendung von mehr als einem Cellarray werden die `onload`-Funktionen ineinander geschachtelt.

4.3.5 Text

Die folgenden Unterkapitel beschäftigen sich mit der Positionierung und Darstellung von Text im GKS.

4.3.5.1 Textpositionierung

Text kann im GKS vertikal und horizontal ausgerichtet werden. Dafür sind bestimmte Ausrichtungspunkte in beide Richtungen gegeben (siehe Abbildung 4.10).

Abbildung 4.10: Textpositionierung im GKS [21]

Der Zeichenkontext, der zum Zeichnen auf dem Canvas verwendet wird, besitzt die Attribute `textAlign` und `textBaseline`, die mit folgenden Werten belegt werden können. Zunächst die möglichen Werte für `textAlign`:

`"left"`	linksbündig
`"center"`	zentriert
`"right"`	rechtsbündig

Die möglichen Werte für `textBaseline` mit zugehöriger GKS-Entsprechung:

`CanvasRenderingContext2D`	GKS
`"alphabetic"`	Base
`"top"`	Top
`"hanging"`	Cap
`"middle"`	Half
`"bottom"`	Bottom

Die vom GKS bereitgestellten Ausrichtungsmöglichkeiten werden alle vom `CanvasRenderingContext2D` abgedeckt. Daher wird für die Realisierung die Ausrichtung abgefragt und die Attribute `textBaseline` und `textAlign` werden mit den entsprechenden Werten belegt.

4.3.5.2 Textrotation

Neben dem Ausrichten von Text ist es möglich, den Text um den Ausrichtungspunkt rotiert darzustellen. Hierfür werden die Transformationsfunktionen `rotate` und `translate` genutzt. Das Koordinatensystem wird mit `translate` an den Ausrichtungspunkt verschoben und eine Rotation um α durchgeführt. Dieser Winkel wird nicht explizit angegeben, sondern aus dem Zeichenaufwärtsvektor (*up*-Vektor, siehe Abbildung 4.11), der senkrecht auf der Schreibrichtung steht, berechnet.

Abbildung 4.11: Richtung des Zeichenaufwärtsvektors

Hierbei ist zu beachten, dass die Funktion `rotate` standardmäßig im Uhrzeigersinn rotiert und deshalb das Vorzeichen vor der Rotation geändert wird. Die Berechnung der Winkel α und β wird mit den Formel aus Abbildung 4.12 durchgeführt. Dabei wird die `atan2`-Funktion verwendet. Sie berücksichtigt den Quadranten des Vektors und gibt den positiven Winkel zur x-Achse zurück. Der Rotationswinkel α wird der Transformationsfunktion `rotate` übergeben und die Rotation wird durchgeführt.

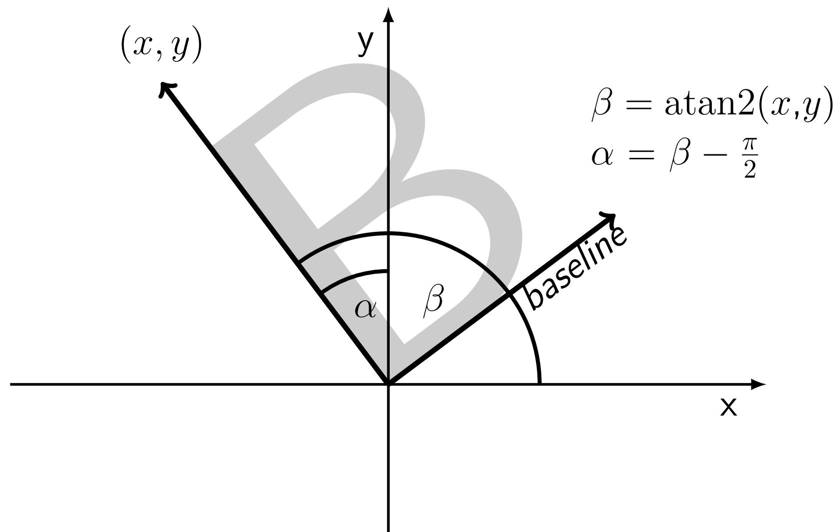


Abbildung 4.12: Bestimmung des Rotationswinkels

Diese Rotation wird für folgenden Elemente übernommen. Deshalb wird vor dem Aufruf von `rotate` der aktuelle Zustand gespeichert und nachher wieder hergestellt. Hierfür werden die Funktionen `save` und `restore` verwendet. Die Funktion `save` legt den aktuellen Zustand auf einen Stack und `restore` entfernt diesen wieder und stellt den vorherigen Zustand her.

Im Folgenden eine Beispiel-Ausgabe für die Rotation eines Textes um 90 Grad ($\widehat{=} -\frac{\pi}{2} \approx -1.5707963267$):

```
c.textBaseline="alphabetic";
c.textAlign="left";
c.fillStyle="rgb(0,0,0)";

c.save();
c.translate(224,324);
c.rotate(-1.5707963267);
c.fillText("Beispiel", 0, 0);
c.restore();
```

Die ersten Zeilen legen die Ausrichtung fest und setzen die Textfarbe auf Schwarz. Daraufhin wird der aktuelle Zustand gesichert und die Verschiebung zum Ausrichtungspunkt $(224, 324)$, sowie die Rotation um 180 Grad werden vorgenommen. Der Text wird an die Stelle $(0, 0)$ gezeichnet und der vorherige Zustand wiederhergestellt.

4.3.5.3 Schriftartendefinition in HTML

Die letzten beiden Unterkapitel beschäftigten sich mit der Positionierung und Ausrichtung von Text. Im Folgenden wird näher auf die Darstellung des Textes eingegangen, beginnend mit der Auswahl der Schriftart in HTML5.

Der Zeichenkontext des Canvas enthält ein Attribut `font` vom Typ `DOMString`[8], das die aktuell gesetzten Schriftinformationen enthält. Sie werden dort in einem für CSS üblichen Format abgelegt. Zu diesen Informationen gehören [19]:

Font family Schriftfamilie (auch generische Familien möglich)

Font weight Glyphendicke

Font style ermöglicht kursive Schriften

Font size Schriftgröße

Vereinfacht betrachtet wird der zusammengesetzte String in folgender Form angegeben:

```
[ <font-style> || <font-weight> ]? <font-size> <font-family>
```

4.3.5.4 Schriftdarstellung im GKS

Bei der Darstellung von Text wird in der vorliegenden Implementierung des GKS zwischen zwei Qualitätsstufen (*Precisions*) unterschieden:

1. `Text Precision String`
2. `Text Precision Stroke`

Die `Text Precision String` wird verwendet, wenn der Text mit den Mitteln des Treibers dargestellt werden soll. In diesem Fall wird zunächst die Schriftart gesetzt und anschließend die `fillText`-Funktion mit der Position und der Zeichenkette aufgerufen. Das Setzen der Schriftart erfolgt mit dem `DOMString` aus Kapitel 4.3.5.3. Dafür

[8]Das Document Object Model ist eine Schnittstelle, um dynamisch die Eigenschaften von HTML- oder XML-Dokumenten abfragen und ändern zu können.

werden der Reihe nach die gesetzten Eigenschaften abgefragt und an den String angehängt.

Bei der `Text Precision Stroke` wird jede Schriftart durch Polylines angenähert. Diese Emulation wird vom GKS über den Aufruf der Funktion `gks_emul_text` übernommen. Sie erwartet die Linien- und Füllroutine, die mit dem Callback-Mechanismus aus Kapitel 2.4.3 übergeben werden.

4.3.6 Dynamische Skalierung der Gesamtgrafik

Wie schon in Kapitel 4.1.1.1 erwähnt, erlaubt das Canvas die Ausgabe in Form einer Rastergrafik. Folglich kann eine Qualitätverbesserung nur über die Vergrößerung des Canvas erreicht werden. Die *Rendering-Engine*[9] hat dann eine größere Zeichenfläche und generiert somit genauere Grafiken. Die Vergrößerung kann mithilfe von JavaScript auch dynamisch gestaltet werden, indem auf folgende Tastatureingaben des Benutzers reagiert wird:

Taste	Funktion
"+"	Vergrößern des Canvas
"-"	Verkleinern des Canvas
"0"	Zurücksetzen (Faktor = 1)

Die Tasten werden anhand ihrer Keycodes[10] zugeordnet und entsprechend verarbeitet. Der Zeichenkontext stellt neben den Transformationsfunktionen zur Verschiebung und Rotation auch eine Routine `scale` zur Skalierung bereit. Sie erwartet jeweils einen Faktor für die x- bzw. y-Richtung und multipliziert alle folgenden Positions- und Längenangaben mit diesem. Auf diese Weise können alle Elemente dynamisch größer gezeichnet werden. Die Skalierung wird – wie bei den Transformationen aus 4.3.5.2 – nicht kumulativ durchgeführt. Stattdessen wird bei jedem Tastendruck der Skalierungsfaktor angepasst und die Funktion `scale` aufgerufen. Außerdem wird bei jeder Skalierung die Canvasgröße mit dem Faktor multipliziert, da die Elemente nach der Skalierung eventuell nicht mehr auf die Zeichenfläche passen.

Das Neuzeichnen geschieht mithilfe von `clearRect` zum Löschen der alten Grafik und der Funktion `draw` zum Zeichnen der grafischen Elemente.

[9] Software, die aus den Elementen die Rastergrafik generiert
[10] Tastatur-Scancode zur Identifikation der gedrückten Taste [17]

4.4 Browserunterstützung

Die mit dem HTML5-Treiber erstellte Datei kann auf den unterschiedlichsten Plattformen und in potentiell jedem möglichen Browser geöffnet werden. Der HTML5-Standard ist noch in der Entwicklung. Deshalb ist es wichtig, die generierte Datei auf ihre Darstellungsunterschiede in verschiedenen Browsern zu untersuchen. Zu den getesteten Browser, gehören:

- Apple Safari Version 6 und 7 (Mac OS X 10.8)
- Google Chrome Version 28 (Mac OS X 10.8)
- Mozilla Firefox Version 23 (Mac OS X 10.8)
- Microsoft Internet Explorer Version 10 und 11 (Windows 7)

Zum Test des Treibers wird folgendes Beispielbild verwendet:

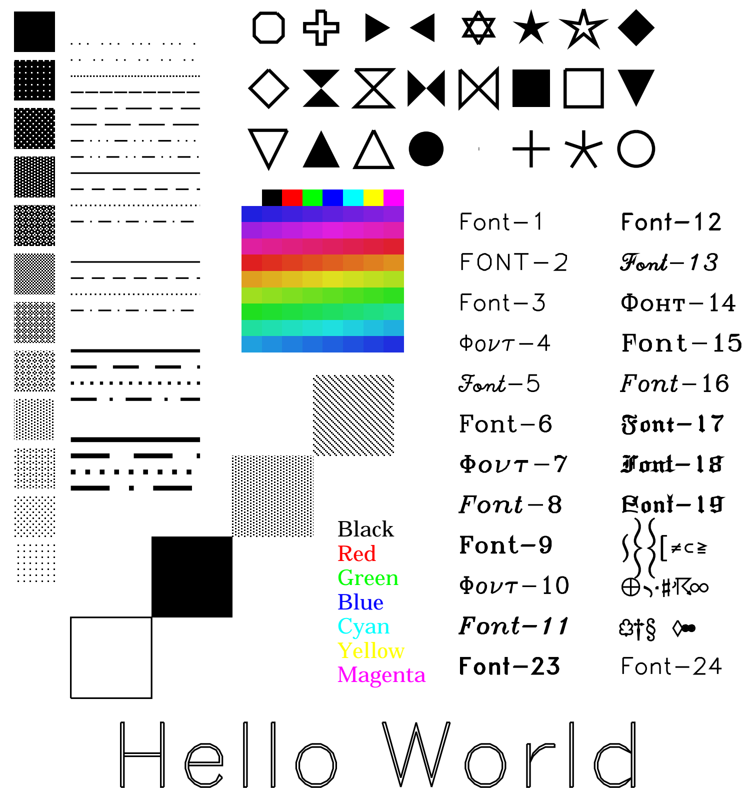


Abbildung 4.13: Ausgabe des existierenden PostScript-Treibers

Dieses Bild verwendet alle GKS-Grafikprimitiven und ruft diese auch mit unterschiedlichen Attributen auf. So werden zum Beispiel Fillareas mit verschiedenen Füllmustern gezeichnet und Polylines mit unterschiedlicher Strichelung. Die Darstellung

in den Browsern sollte auch im Detail möglichst wenig von der Referenzgrafik abweichen.

4.4.1 Testen der verschiedenen Browser

Zunächst wird eine HTML-Datei mit obiger Beispielgrafik generiert, um diese in den bereits genannten Browsern zu öffnen.

4.4.1.1 Google Chrome

Der Treiber wurde in der Entwicklungsphase regelmäßig unter Google Chrome getestet, da dieser von den bekannten Browsern die umfangreichste Implementierung des HTML5 Standards bereitstellt [18]. Er verfügt über alle vom HTML5-Standard vorgesehenen und benötigten Funktionen zum Erstellen der Grafik und stellt diese folglich richtig dar.

4.4.1.2 Mozilla Firefox

Mozilla Firefox zeigt alle Elemente an. Die Polylines sind nicht gestrichelt, weil die Routine `setLineDash` nicht vorhanden ist. Stattdessen bietet der Zeichenkontext die Möglichkeit, das Attribut `mozDash` mit der Liste der Strichelungslängen zu belegen. Auf diese Weise wird die gesamte Grafik richtig dargestellt.

4.4.1.3 Apple Safari

Apples Browser Safari in der Version 6.0.5. verhält sich beim ersten Test wie Firefox. Er beinhaltet ebenfalls keine Implementierung der Funktion `setLineDash` und stellt bislang keine Alternative bereit. Ein Test in der noch nicht veröffentlichten Version 7 ergab jedoch, dass sie mit der nächsten Version hinzugefügt wird. Bis zu dieser Version wird folgende Methode verwendet, um Fehler beim Laden der Seite zu vermeiden. Sie funktioniert ebenso bei anderen Browsern ohne Implementierung von `setLineDash`:

```
1 if (!ctx.setLineDash) {
2     ctx.setLineDash = function () {}
3 }
```

Enthält der aktuelle Zeichenkontext die Routine `setLineDash` nicht, so wird eine leere Implementierung der Routine hinzugefügt. Auf diese Weise wird keine Strichelung durchgeführt, aber ein Abbruch der Zeichenroutine verhindert.

4.4.1.4 Internet Explorer

Der Internet Explorer verhält sich genauso wie Apples Safari der Version 6. In der Version 10 existiert die `setLineDash`-Methode nicht. Mit Version 11 wurde diese jedoch eingeführt und das Bild wird richtig angezeigt. Das Vergrößern der Grafik über die Tastatur funktioniert nicht, da der Keycode im Internet Explorer nicht über das Attribut `event.keyCode`, sondern über `event.which` abgefragt wird. Wird die Abfrage um dieses Attribut erweitert, funktioniert auch die Skalierung:

```
...
if (event.keyCode == 43 || event.which == 43) { // + pressed
...
```

4.4.2 Schriftdarstellung

Die Darstellung von Schriften ist von Betriebssystem und Browser abhängig. Daher ist es sehr aufwendig, die Darstellung der Schriften umfassend zu realisieren und zu testen. Ein paar der verwendeten Schriftarten funktionieren deshalb nicht in jedem Browser bzw. auf jedem System. Der Schriftartenparser setzt in diesen Fällen eine Standardschriftart und der Text wird trotzdem dargestellt.

5 Zusammenfassung

Am Ende dieser Arbeit steht ein in Python geschriebener GKS-Treiber für HTML5 zur Verfügung. Die Daten werden statisch in einem Canvas angezeigt und lassen sich browserunabhängig visualisieren. Die Ausnahme bildet hier das Zeichnen von gestrichelten Linien. Die aktuellen Versionen des Safaris und Internet Explorers unterstützen diese noch nicht. Da bereits die nächsten Versionen dieser Browser die benötigte Routine bereitstellen und die Emulation von gestrichelten Linien aufwendig wäre, wurde auf eine eigene Implementierung der Strichelung verzichtet.

Mit dem Datenparser-Automaten und dem ctypes-Wrapper kann die empfangene GKS-Displayliste in Python verarbeitet werden. Zusätzlich ist es möglich, die vom GKS bereitgestellten Funktionen und Strukturen zu verwenden. Diese Module vereinfachen die Entwicklung weiterer Python-Treiber, da sie für alle Treiber gleich sind und wiederverwendet werden können.

Die Kommunikation mittels ZeroMQ bereitete einige Probleme, da sich die gewünschte 1:N-Verbindung mit Publish-Subscribe nicht realisieren ließ. Die 1:1-Verbindung nach dem PUSH-PULL-Muster stellt kein Problem dar und sendet die Daten immer vollständig und ohne Fehler an den Ausgabetreiber.

6 Ausblick

Die Darstellung der Schriftarten ist noch nicht vollständig getestet und Bedarf noch weiterer Anpassungen, da noch nicht alle vom GKS unterstützten Schriften genutzt werden können. Hierfür ist es sinnvoll, die Liste der Fonts auf sogenannte *Websafe Fonts* abzubilden, die es dem Schriftartenparser ermöglichen, die benötigte Schrift anzunähern, wenn sie nicht vorhanden ist.

Desweiteren ist es möglich neben der Vergrößerung der Grafik weitere Interaktionsmöglichkeiten mit der Grafik zu implementieren. Beispielweise ließen sich nach dem selben Muster Rotationen oder Verschiebungen der Grafik realisieren.

Aktuell generiert der Ausgabetreiber statische HTML-Dateien eines Bildes. Deshalb müssen die Bilder von Sequenzen nacheinander, einzeln im Browser geöffnet werden. In Zukunft wäre es möglich dem Treiber mitzuteilen, dass die folgenden Displaylisten eine Sequenz darstellen und dieser die Daten in eine HTML-Datei schreibt, die nacheinander, zeitlich versetzt im Browser angezeigt werden können.

Literaturverzeichnis

[1] Bechlars, Jörg ; Buhtz, Rainer: *GKS in der Praxis*. Springer-Verlag, 1986. – ISBN 978–3–540–16139–2

[2] Dück, Marcel: *Entwicklung eines GKS-Gerätetreibers als Java-basierte Client/-Server Webanwendung*. August 2010

[3] Flanagan, David: *JavaScript Das umfassende Referenzwerk*. O'Reilly, 2007. – ISBN 978–3–89721–491–0

[4] Freed, N.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. `http://tools.ietf.org/html/rfc2046`, Abruf: 13.08.2013

[5] Heimbach, Ingo: *Entwicklung einer logischen GKS Gerätetreibers für die wxWidgets-Klassenbibliothek*. Dezember 2011

[6] IBM Corporation: *Polymarker*. `http://publib.boulder.ibm.com/infocenter/zos/v1r12/index.jsp?topic=%2Fcom.ibm.zos.r12.admk100%2Fadmk1a0029.htm`, Abruf: 06.08.2013

[7] iMatix Corporation: *ØMQ - The Guide*. `http://zguide.zeromq.org/page:all`, Abruf: 21.08.2013

[8] ISO/IEC: *Information technology - Computer graphics and image processing - Graphical Kernel System (GKS)*. `http://webstore.iec.ch/preview/info_isoiec7942-2%7Bed1.0%7Den.pdf`, Abruf: 06.08.2013

[9] Lie, H. ; Bos, B. ; Lilley, C.: *The text/css Media Type*. `http://www.ietf.org/rfc/rfc2318.txt`, Abruf: 08.08.2013

[10] Masinter, L.: *The data URL scheme*. `http://tools.ietf.org/html/rfc2397`, Abruf: 06.08.2013

[11] Münz, Stefan ; Gull, Clemens: *HTML5 Handbuch*. Franzis Verlag, 2010. – ISBN 978–3–645–60079–8

[12] Piël, Nicholas: *ZeroMQ an introduction*

[13] Python Software Foundation.: *ctypes - A foreign function library for Python*. http://docs.python.org/2/library/ctypes.html, Abruf: 30.07.2013

[14] Refsnes Data: *HTML Examples*. http://www.w3schools.com/html/html_examples.asp, Abruf: 06.08.2013

[15] Refsnes Data: *HTML Examples*. http://www.w3schools.com/js/, Abruf: 06.08.2013

[16] Refsnes Data: *HTML5 Introduction*. http://www.w3schools.com/html/html5_intro.asp, Abruf: 30.07.2013

[17] SELFHTML e.V.: *keyCode*. http://de.selfhtml.org/javascript/objekte/event.htm#key_code, Abruf: 12.08.2013

[18] Sights: *The HTML5 Test – how well does your browser support HTML5?* http://html5test.com/results/desktop.html, Abruf: 13.08.2013

[19] W3C: *Basic Font Properties*. http://www.w3.org/TR/css3-fonts/#basic-font-props, Abruf: 13.08.2013

[20] W3C: *W3C Candidate Recommendation 17 December 2012*. http://www.w3.org/TR/2012/CR-html5-20121217/embedded-content-0.html#the-img-element, Abruf: 05.08.2013

[21] Winkler, Jörg: *Rasterisierung von Vektor-Schriften auf der Basis der FreeType Software*. Dezember 2012