Xgraphics

hoffentlich eine Erleichterung ...

Martin Lüders

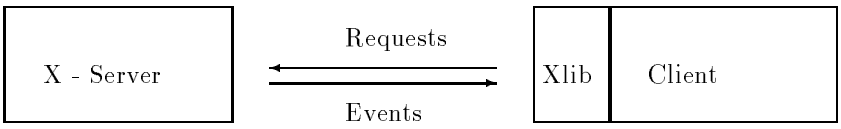
Inhaltsverzeichnis

1. Motivation

Ein wichtiger Bestandteil vom Computersimulationen ist neben den eigentlichen Rechnungen natürlich auch die graphische Ausgabe der Ergebnisse. Auf modernen Systemen (Workstations) stehen dazu vielfältige Werkzeuge zur Verfügung. Deren Handhabung, im speziellen die Programmierung der X-Windows Oberfläche, ist im allgemeinen allerdings bei weitem nicht trivial und erfordert einigen Aufwand der Einarbeitung. Bei der Programmierung von kleinen Simulationen und Demonstrationsprogrammen soll allerdings die eigentliche Thematik, z.B. die Physik des zu beschreibenden Phänomens im Mittelpunkt des Programms stehen, und nicht die Programmierung der graphischen Benutzeroberfläche. Diese sollte sich mit wenig Aufwand und ohne ein tieferes Studium der entsprechenden Handbücher leicht handhaben lassen. Zu diesem Zweck habe ich die Bibliothek *Xgraphics* konzipiert. Die Graphikprogrammierung soll damit so einfach bleiben wie z.B. auf einen PC, soll aber gleichzeitig Programme erzeugen, die sich wie ein "vernünftiges" X-Programm verhalten, d.h. wenn ein überdecktes Fenster wieder freigelegt wird, sollte der Inhalt des Fensters wieder erscheinen, und einige andere Eigenschaften, die der Benutzer als selbstverständlich hinnimmt. Trotzdem sollte ein einfaches Programm, wie z.B. "Hello-World", nicht mehrere Seiten lang sein, um diese Eigenschaften zu erfüllen, sondern es sollte sich in wenigen Zeilen programmieren lassen.

2. Das Konzept von X-Windows

X-Windows stellt die graphische Benutzeroberfläche von Unix-, bzw. Unix-artigen Systemen dar. X wurde am MIT entwickelt und wird seit 1987 von allen bekannten Workstation-Herstellern unterstützt. Die Grundausstattung von X ist *public domain*, und kann daher für Workstations als Grundausstattung vorrausgesetzt werden. Im Vergleich zu anderen Windows-systemen, für PC's beispielsweise, ist X ein sehr allgemeines System, das noch keine bestimmte Benutzeroberfläche mit bestimmten Verhalten und Aussehen definiert. Diese wird erst durch die Entscheidung für einen bestimmten Window-Manager festgelegt. X ist als ein Multi-Fenster-System für Netzwerke konzipiert. Das ermöglicht ein Programm auf einem anderen Computer laufen zu lassen (z.B. Großrechner) als der, auf dem die graphische Ausgabe erscheinen soll (z.B. Workstation mit Farbmonitor). Aus diesem Gesichtspunkt ergibt sich die *Server-Client* Struktur von X.



X11-Protokoll

X-Server: Der *X-Server* läuft auf dem Computer, auf dem die Graphik dargestellt werden soll. Er führt die Zeichenbefehle, die er als *Requests* erhält aus und sendet Informationen über den Zustand der Tastatur, der Maus etc. als *Events* an die Anwendung, den *Client*.

Client: Der *Client* ist das eigentliche Anwenderprogramm. Um Graphikausgaben zu erzeugen sendet der *Client* Zeichen-*Requests* an den Server. Um dies zu tun verwendet er Funktionen der *Xlib*.

Xlib: *Xlib* ist die Bibliothek, die die Grundbefehle für die Kommunikation zwischen dem *Client* und dem *X-Server* bereitstellt. Auf dieser aufbauend gibt es noch weitere Bibliotheken wie z.B. XToolkit, Athena etc. die erheblich komplexere Befehle zur Verfügung stellen, aber in ihrer Handhabung leider auch nicht gerade einfacher als *Xlib* werden.

Events: Mit *Events* teilt der *X-Server* dem *Client* mit, ob sich am Zustand der Bildschirms (z.B. ein verdecktes Fenster wurde freigelegt), der Tastatur oder der Maus etwas geändert hat. Der *Client* kann dann mit Hilfe von *Xlib*-Funktionen auf diese *Events* reagieren.

Requests: Sie werden vom *Client* an den *X-Server* geschickt, um diesen dazu zu veranlassen eine Zeichenaktion auszuführen.

X11-Protokoll: Darunter versteht man das Format der Daten, die in *Events* und *Requests* ausgetauscht werden.

Um die Leistungsfähigkeit der Workstations besser ausnutzen zu können arbeitet X "asynchron". das bedeutet, daß *Events* und *Requests* von den jeweiligen Programmen nicht sofort ausgewertet werden, sondern sie werden in einer Queue zwischengespeichert und erst dann verarbeitet, wenn das Programm "gerade mal Zeit hat". Dies ist besonders dann von Nutzen, wenn der *X-Server* mehrere Anwendungen versorgen muß.

Die Struktur von Anwenderprogrammen sollte sich natürlich daran orientieren, daß es immer wieder auf *Events* reagieren muß, um dem Benutzer eine Eingriffsmöglichkeit zu geben. Die Hauptroutine des Programms sollte daher immer auch die *Events* abfragen und darauf reagieren. Der Grundaufbau eines Programms, welches die *Xlib* verwendet, sollte also etwa wie folgt aussehen:

1. Initialisierung
   - Verbindung zum *Server* aufbauen
   - Fenster öffnen
   - Resourcen (z.B. Zeichenmodus, Schrift, ...) definieren
2. Hauptschleife
   - *Events* abfragen
   - *Events* bedienen
   - eigentlicher Programm Code
3. Ende
   - eigener Code
   - Fenster schließen
   - Resourcen freigeben
   - Verbindung zum *Server* schließen

3. Konzepte von Xgraphics

Die Bibliothek *Xgraphics* soll eine einfache Programmierung von X-Windows Anwendungen ermöglichen. Dazu werden viele Aufgaben intern bearbeitet. So z.B. das Festlegen der Zeicheneigenschaften wie Farbe, Liniendicke etc., das Verhalten der Fenster, wenn sie vom Benutzer vergrößert bzw. verkleinert werden. Der Programmierer ist also von diesen Aufgaben weitgehend befreit[1].

3.1 Zeichenbereiche: *Worlds*

Zu den Grundbausteinen von *Xgraphics* gehören die Zeichenbereiche oder *Worlds*. Sie stellen einen definierten Bereich eines Fensters dar, in den mittels effektiver Koordinaten gezeichnet werden kann. Dies ermöglicht die Zeichenroutinen unabhängig von der Geometrie der Fenster zu programmieren. Zu den Eigenschaften eines Zeichenbereiches gehört weiterhin auch ein definiertes Verhalten bei einer Größenänderung des Fensters. Grundsätzlich zu unterscheiden sind dabei die Zeichenbereiche, deren Größe mit dem Fenster skaliert, und die, die ihre Größe beibehalten. Letztere werden, wenn das Fenster wieder sichtbar wird, d.h. ein Expose-Event eintritt, automatisch wieder abgebildet. Für die anderen muß dies vom Programmierer selbst übernommen werden. Für diese schreibt man am besten eine `Redraw()` Funktion, die dann im Rahmen des Event-handling aufgerufen wird.

3.2 Die Maus: *Buttons*

Ein Bauelement zur Gestaltung der Benutzeroberfläche eines Programms sind die *Buttons*. Sie erlauben die Steuerung des Programmablaufes mit der Maus, erfordern aber nicht mehr Programmieraufwand als ein Tastatur-gesteuertes Programm auf einem PC. Die Maus-Events werden dabei intern in Tastaturevents übersetzt. Der Programmierer muß am Anfang nur festlegen, welcher Tastendruck bei Anklicken eines bestimmten Feldes simuliert werden soll.

3.3 Farben

Die Darstellung verschiedener Farben unter X ist ein schwieriges Kapitel. Die meisten X-Server können nur 256 Farben gleichzeitig bereitstellen. Laufen viele verschiedene Anwendungen auf einem Server, so kann es leicht vorkommen, daß diese Anzahl der Farben nicht ausreicht, um alle Programme zufrieden zu stellen. Läuft beispielsweise ein Programm zur Darstellung eines Farbphotos, so wird dieses i.allg. schon alle verfügbaren Farben verwenden. Wird dann ein weiteres Programm gestartet, welches z.B. Graustufen anzeigen will, beginnen schon die Schwierigkeiten. Das Programm, besser gesagt der Programmierer, hat dann die Wahl sich mit den verbleibenden Farben zufrieden zu geben, auf ähnliche Farben auszuweichen, oder rigoros die Farbtabelle zu überschreiben, was zur Folge hat, daß sich die Farben der anderen Programme recht unliebsam verändern, solange der Fokus auf diesem Fenster liegt.

[1] In bestimmten Fällen muß dennoch auf das Expose-Event reagiert werden. Mehr dazu später.

Für *Xgraphics* habe ich mich für die zurückhaltende Methode entschieden. *Xgraphics* enthält eine Wunschliste von Farben und versucht möglichst viele dieser Farben zu erhalten. Ist ein Farbe dieser Liste bereits in der aktuellen Farbpalette des Servers, so wird sie übernommen und kann verwendet werden. Ist die spezielle Farbe nicht enthalten, so wird, solange noch freie Plätze in der Palette sind, die Farbe neu in die Palette aufgenommen. Sind keine Einträge mehr verfügbar, so sucht das Programm nach einer möglichst ähnlichen Farbe und verwendet diese anstelle der gesuchten. Ist auch keine ähnliche Farbe in der Palette, so kann diese Farbe nicht verwendet werden, und das Programm versucht die nächste Farbe aus der Wunschliste zu bekommen. Dadurch kann es leider vorkommen, daß gewisse Farben in nicht vorhersagbarer Weise nicht zur Verfügung stehen. Man sollte daher nicht versuchen bestimmte Farben explizit zu verwenden. Andererseits ist die Chance daß die Grundfarben Rot, Grün und Blau schon in der Palette sind sehr groß. Diese stehen daher am Anfang der Wunschliste, da sich durch den Ausfall einer Farbe die Nummerierung der Farben verschiebt.

Um Anwendungen zu schreiben, die auf verschiedenen X-Servern, also zum Beispiel auch auf einem schwarz-weißen X-Terminal, sinnvoll dargestellt wird, sollte man das folgende beachten: Will man verschiedene Objekte farblich unterscheiden, die aber alle auf jeden Fall dargestellt werden sollen, so darf die Farbe `0`, die immer der Hintergrundfarbe entspricht, nicht verwendet werden. Um dies bei einer durchlaufenden Numerierung der Farben zu verhindern, kann man folgende Konstruktion verwenden:

```
color = (index)%(NofColors-1)+1.
```

Sollen andererseits zwei Zustände auf jeden Fall voneinander unterschieden werden, so muss die eine der Farben im Falle eines Schwarz-Weiß Servers auf Weiß und die andere auf Schwarz abgebildet werden. Das erreicht man dadurch, daß man eine Farbe mit gerader Nummer und eine mit ungerader Nummer verwendet.

3.4 Aufbau eines Programms

Die Struktur eines *Xgraphics*-Programms muß natürlich der eines X-Programms entsprechen. Das bedeutet, daß zu Beginn die Initialisierungen stattfinden müssen. Dies geschieht mit der Routine `InitX()`, die alle weiteren Festlegungen intern bewältigt. Dann werden i.allg. die Fenster und Zeichenbereiche und schließlich die Funktion der Buttons definiert. In der Hauptschleife des Programms müssen wieder die Events abgefragt und bedient werden. Das beschränkt sich allerdings i.allg. auf das Expose-Event, falls Zeichenbereiche verwendet werden, deren Größe mit dem Fenster skaliert, und auf Tastatur-Events. Eventuell will man auch direkt auf Mausklicks reagieren. Ein typischer Aufbau eines Programms findet sich im Beipsielprogramm in Kapitel 5.

4. Xgraphics - Funktionen

4.1 Initialisierung und Verwaltung

4.1.1 X Initialisieren und Beenden

`void InitX()`

Die Verbindung zum X-Server wird hergestellt, X-Datenstrukturen wie der Graphics-Context und andere werden erzeugt und mit Defaultwerten belegt. Die Schriftart wird festgelegt und eine *Xgraphics*-interne Datenstruktur zur Verwaltung der Zeichenbereiche wird initialisiert.

`void ExitX()`

Alle noch verbliebenen Fenster werden geschlossen, von X und *Xgraphics* verwendete Speicherbereiche wieder freigegeben und schließlich wird die Verbindung zum X-Server geschlossen.

4.1.2 Vom Umgang mit Fenstern

`Window CreateWindow( int width, int height, char* name )`

Diese Funktion erzeugt ein Fenster der angegebenen Größe `width`× `height` und dem Namen `name`. Das Fenster erscheint allerdings noch nicht auf dem Bildschirm, sondern muß explizit mit `ShowWindow` sichtbar gemacht werden. Die Funktion gibt eine Datenstruktur zurück, die das Fenster beschreibt. Diese muß bei allen Zeichenaktionen angegeben werden.

`void DestroyWindow( Window win )`

Das Fenster `win` wird geschlossen. Dabei werden zunächst alle in diesem Fenster enthaltenen Zeichenbereiche entfernt und deren Speicherplatz wieder freigegeben.

`void ShowWindow( Window win )`

Das Fenster `win` wird am Bildschirm angezeigt.

`void HideWindow( Window win )`

Das Fenster `win` wird vom Bildschirm genommen.
Beachte: Die Datenstruktur `win` bleibt im Gegensatz zu `DestroyWindow` erhalten.

`void ClearWindow( Window win )`

Der Inhalt des Fensters `win` wird gelöscht.

4.1.3 Vom Umgang mit Zeichenbereichen

```
World CreateWorld (
        Window win,
        int px, int py,
        int pwidth, int pheight,
        float wx1, float wy1,
        float wx2, float wy2,
        int scalable,
        int gravity
        )
```

Die Funktion `CreateWorld` erzeugt einen Zeichenbereich im Fenster `win`. Die linke, obere Ecke kommt an den Koordinaten (`px,py`) relativ zum Fenster zu liegen und der Zeichenbereich hat die Größe `pwidth`×`pheight` in Pixeln. Die effektiven Koordinaten des oberen linken, bzw. unteren, rechten Punktes des Zeichenbereiches sind `(wx1, wy1)` bzw. `(wx2, wy2)`. Die Variable `scalable` bestimmt das Verhalten des Zeichenbereiches bie einer Größenänderung des zugehörigen Fensters. Für die Werte sind die folgenden Konstanten definiert, die wenn gesetzt werden folgende Bedeutung haben:

`0:`	Der Zeichenbereich ändert seine Größe nicht.
`SCALABLE:`	Der Zeichenbereich kann seine Größe ändern
`FREE_ASPECT:`	Das Höhe zu Breite Verhältnis darf sich ändern
`FIXED_WIDTH:`	Die Breite bleibt fixiert. Dabei ist `FREE_ASPECT` natürlich impliziert.
`FIXED_HEIGHT:`	Wie `FIXED_WIDTH`. Die Höhe bleibt fixiert.

Sie werden durch den logischen *Oder*-Operator | miteinander verbunden. Beispiele für den Wert von `scalable` sind:

- `0`
  Der Zeichenbereich ändert seine Größe nicht. Wenn das Fenster vergrößert wird, verschiebt er sich entsprechend dem `gravity`-Wert. Solche, nichtskalierbare Zeichenbereiche sind die einfachsten. Hier kann *Xgraphics* bei einem Expose-Event den Inhalt des Zeichenbereiches selber wieder herstellen. Es ist kein `Redraw()`-Befehl nötig.
- `SCALABLE` Die Größe des Zeichenbereiches paßt sich der des Fensters an. Das Höhe zu Breite Verhältnis bleibt dabei erhalten.
- `SCALABLE | FREE_ASPECT` Die Größe des Zeichenbereiches paßt sich der des Fensters an. Dabei bleiben die Abstände des Zeichenbereiches zu den Rändern des Fensters konstant.

Für die Konstante `gravity` stehen folgende mögliche Werte zur Verfügung: `NorthGravity, NorthWestGravity, WestGravity, ...` . Die Himmelsrichtung gibt dabei diejenige Ecke oder Kante des Fensters an, mit der sich der Zeichenbereich verschiebt, wenn die Größe des Fensters geändert wird. Dabei sollte natürlich darauf geachtet werden, daß Dinge, die sich am unteren Rand des Fensters befinden, sinnvollerweise mit dem unteren Rand bewegen, um nicht in Konflikt mit darüberliegenden, skalierenden Bereichen zu kommen.

```
void DestroyWorld(Wold world)
```

Beseitigt den Zeichenbereich `world` aus dem Speicher. Dieser Befehl ist notwendig, wenn man in einem Fenster einen Bereich der mit einem Zeichenbereich

belegt ist, zu einem späteren Zeitpunkt durch einen anderen Zeichenbereich beschreiben möchte. Löscht man den ersten nicht durch `DestroyWorld()`, so kann es geschehen, daß diese Fläche bei einem Expose-Event immernoch gelöscht wird, und zwar evtl. nachdem der Inhalt des neuen Zeichenbereiches wiederhergestellt wurde.

`void ClearWorld(World world)`
Die Fläche des Zeichenbereiches `world` wird gelöscht.

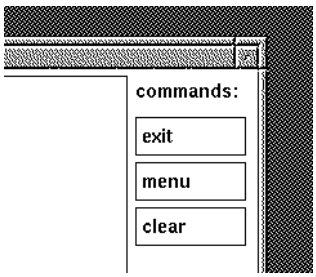
4.1.4 Buttons

```
void InitButtons (
        Window win,
        const char* buttonstring,
        int width
        )
```

`InitButtons` erzeugt am rechten Rand des Fenster `win` eine Reihe von Buttons, die mit der Maus angeklickt werden können und dann ein Tastatur-Event liefern. Die Zahl, Funktion und Beschriftung der Buttons wird durch den String `buttonstring` festgelegt. Dabei werden für jeden Button drei Angaben gemacht, die jeweils durch ein Komma voneinander getrennt sind.

- Unterscheidung zwischen Button / Textzeile:
  für einen Button muß ein `b` und für eine Text ein `t` gesetzt werden.
- Beschriftung des Buttons, bzw. Text.
- Tastensymbol, welches zurückgegeben werden soll.

Zum Beispiel erzeugt der String
`"t,commands,,b,exit,e,b,menu,m,b,clear,c"`
die folgenden Buttons:



In einem Programm mit mehreren Fenstern kann immer nur ein Satz von Buttons existieren. Dies entspricht der Philosophie, den Programmablauf an dem eines PC-Programms zu orientieren. Übergibt man die Kontrolle an ein Unterprogramm mit einem eigenen Fenster indem man in diesem neue Buttons initialisiert, so verschwinden die Buttons des Hauptfensters für diese Zeit. Wenn das Unterprogramm verlassen wird, müssen die Buttons der Hauptroutine wieder neu initialisiert werden.

4.2 Event-Handling

`int GetEvent( XEvent* event, int waitflag )`
`GetEvent` liest das nächste Event von der Event-Queue des Programms und schreibt die Informationen in die Eventvariable, deren Zeiger durch `event` übergeben wird. Im Normalfall ist es nicht (bis auf `event.type`) notwendig die innere Struktur der Eventvariablen zu kennen. Da diese aber direkt verwendet wird,

kann der X-kundige Programmierer natürlich auch auf weitere Informationen eines Events zurückgreifen. Die Variable `waitflag` legt den Betriebsmodus von `GetEvent` fest. Mögliche Werte sind:

- `0`: Falls kein Event vorliegt gibt die Routine den Wert `0` zurück, erzeugt ein Event mit der ID 0 (um Fehler zu verhindern, wenn dennoch versucht wird dieses Event zu bearbeiten) und übergibt die Kontrolle wieder an die übergeordnete Routine.
- `1`: Falls kein Event vorliegt wartet `GetEvent` solange, bis wieder ein Event eintritt. Dieser Modus ist zu empfehlen, wenn das Programm keine weiteren Rechnungen mehr zu reledigen hat, sondern nur noch auf eine Eingabe vom Benutzer wartet. In diesem Modus, verbraucht das Programm dabei keine Rechenleistung von der CPU. (Läßt man das Programm statt dessen in eine Warteschleife laufen, saugt es relativ viel Rechenzeit.)

Hat man das Event ausgelesen, sollte i.allg. auch darauf reagiert werden. Dazu verwendet man am besten eine `switch` Konstruktion. Die hier eingehende Größe ist der Typ des Events. Er ist bei allen Events gekennzeichnet durch `event.type` (oder `event->type`, falls `event` ein Zeiger auf eine Eventvariable ist). Die wichtigsten Events, die auftreten können und auf die reagiert werden sollte sind:

`Expose:` Das Fenster ist aus irgendeinem Grunde wieder sichtbar geworden. Der Inhalt des Fensters muß neu gezeichnet werden. Für Zeichenbereiche, die nicht skalierbar sind, d.h. `scalable = 0`, wird dies von `GetEvent` intern bewerkstelligt. Lediglich für skalierbare Zeichenbereiche muß das Programm selber eine Routine bereitstellen, die diesen wiederherstellt. Diese Routine ist dann im falle eines Expose-Events auszurufen.

`KeyPress:` Es wurde eine Taste gedrückt, oder einer der Buttons mit der Maus angecklickt. Der Character-Code der Taste kann mit `ExtractChar` erhalten werden.

`ButtonPress:` Es wurde eine Maustaste betätigt. Informationen über das Ereignis, wie die Koordinaten und die Nummer der Maustaste können mit `WGetMousePos` gewonnen werden.

`char ExtractChar(XEvent event)`

`ExtractChar` liefert bei einem `KeyPress` Event den ASCII Code der gedrückten Taste.

`int WGetMousePos(World world, XEvent event, float* x, float *y)`

`WGetMousePos` liefert bei einem `ButtonPress` Event die Koordinaten des Events, relativ zu dem Zeichenbereich `world`. Der Wert der Funktion ist `0`, wenn die Maus nicht innerhalb des gegebenen Zeichenbereiches war, ansonsten wird die Nummer der gedrückten Maustaste zurückgegeben.

`int GetNumber(Window win, int x, int y, float *value)`

Mit `GetNumber` kann man eine Zahl von der Tastatur einlesen. An der Position `x,y` erscheint dabei die eingetippte Zahl. `GetNumber` erlaubt die Verwendung der *Backspace*-Taste zum Editieren der Zahl. Die Zahlen dürfen vorzeichenbehaftet sein und ein Dezimalkomma enthalten. Das Ergebniss wird in die durch den Pointer `*value` übergebene Variable geschrieben.

`int WGetNumber(World world, float x, float y, float *value)`

Wie `GetNumber`, allerdings beziehen sich die Koordinaten auf den Zeichenbereich `world`.

4.3 Zeichenbefehle

4.3.1 Farben und Zeichenfunktionen

Die Farbe eines zu zeichnenden Objektes und die Funktion, mit der es dargestellt werden soll, sind wie folgt in der Variable `int c` kombiniert. Die vier niedrigsten Bits kodieren die Farbe. Sie stellen den Eintrag in der internen Liste `mycolors`, die während der Initialisierung mit `InitX()` angelegt wird (Siehe auch Kapitel Farben). Die Bits 4-7 kodieren die Funktion, mit der dargestellt werden soll. Da die normale Funktion `GXcopy` in X leider nicht dem Wert `0` entspricht, gibt es zusätzlich noch ein Flag in Bit 8, welches festlegt, ob eine von `GXcopy` abweichende Funktion verwendet werden soll. Am besten betrachtet man mal ein paar Beispiele:

- Normale Darstellung:
  `c = 2`. Ein Punkt, beispielsweise, wird mit der Farbe `mycolors[2]` dargestellt, d.h. das Pixel wird mit dieser Farbe überschrieben.
- Darstellung mit XOr:
  `c = 4096 + 256*GXxor + 2`. Ein Punkt wird durch die Funktion Xor mit dem vorhergehenden Wert des entsprechenden Pixel verknüpft. Zur Vereinfachung ist für die Darstellung mit Xor die Konstante `Xor = 4096 + 256*GXxor` definiert. Wenn der Punkt ursprünglich die Farbe Weiß hatte, dann erscheint der Punkt nach dem Befehl mit der Farbe `mycolors[2]`. Hatte der Punkt vorher eine andere Farbe, dann läßt sich das Ergebnis i.a. nicht vorhersagen.

4.3.2 Zeichnen in Fenster

`void ClearArea (Window win, int x, int y, int width, int height)`
löscht die angegebene Fläche.

`void DrawPoint (Window win, int x, int y, int c)`
zeichnet einen Punkt der Farbe `c` in das Fenster `win`.

`void DrawPoints (Window win,`
`XPoint *points, int NofPoints, int c)`
zeichnet `NofPoints` viele Punkte. Sie werden übergeben in der Struktur `XPoint *points`, einem Array von `struct { int x,y } XPoint`.

`void DrawLine (Window win,`
`int x1, int y1, int x2, int y2, int c)`
zeichnet eine Linie von `(x1,y1)` nach `(x2,y2)`.

`void DrawLines (Window win,`
`XPoint *points, int NofPoints, int c)`
zeichnet ein offenes Polygon zwischen den durch `points` definierten Punkten.

`void DrawCircle (Window win, int x, int y, int r, int c)`
zeichnet einen Kreis mit dem radius `r` und dem Mittelpunkt `(x,y)`.

`void FillCircle (Window win, int x, int y, int r, int c)`
zeichnet einen ausgefüllten Kreis.

`void DrawString (Window win, int x, int y, const char *text,`
`int c)`
zeichnet an der Stelle `(x,y)` den Text `text`.

```
void DrawRectangle (Window win,
        int x1, int y1, int x2, int y2, int c)
```
zeichnet ein Rechteck mit der linken oberen Ecke (`x1,y1`) und der rechten unteren Ecke (`x2,y2`).

```
void FillRectangle (Window win,
        int x1, int y1, int x2, int y2, int c)
```
zeichnet ein ausgefülltes Rechteck.

```
void DrawPoly (Window win, XPoint *points, int NofPoints, int c)
```
zeichnet ein geschlossenes Polygon. Der durch `points` gegebene Polygonzug muß nicht notwendigerweise geschlossen sein.

```
void FillPoly (Window win,
        XPoint *points, int NofPoints, int c, int cfill)
```
zeichnet mit der Farbe `c` ein geschlossenes Polygon und füllt es mit der Farbe `cfill` aus.

4.3.3 Zeichnen in Zeichenbereiche

Die Zeichenfunktionen für Zeichenbereiche haben den gleichen Syntax wie die Zeichenfunktionen für Fenster. Allerdings wird statt des Fensters der Zeichenbereich angegeben. Die koordinaten sind nun Fließkommazahlen.

```
void WDrawPoint (World world, float x, float y, int c)

void WDrawPoints (World world, XPoint *points,
        int NofPoints, int c)

void WDrawLine (World world, float x1, float y1, float x2, float y2,
        int c)

void WDrawLines (World world, WPoint *points, int NofPoints,
        int c)

void WDrawCircle (World world, float x, float y, float r, int c)

void WFillCircle (World world, float x, float y, float r, int c)

void WDrawString (World world, float x, float y, const char *text,
        int c)

void WDrawRectangle (World world, float x1, float y1,
        float x2, float y2, int c)

void WFillRectangle (World world, float x1, float y1,
        float x2, float y2, int c)

void WDrawPoly (World world, WPoints *points, int NofPoints, int c)
```

```
void WFillPoly (World world, WPoints *points, int NofPoints, int c,
          inf cfill)
```

5. Demo-Programm

In diesem Kapitel soll an Hand von Beispielprogrammen die Funktionsweise von Xgraphics verdeutlicht werden und außerdem einige Richtlinien zur Erzeugung "guter" Programme gegeben werden. Das beinhaltet Vorschläge zur Anordnung von Zeichenbereichen in einem Fenster, zur Verwendung von Farben und zur Behandlung von Warteschleifen.

Zunächst soll ein "Minimalprogramm" gezeigt werden, welches nur ein Fenster öffnet, darin eine Lissajous-Figur zeichnet und das Fenster nach Drücken einer Maustaste wieder schließt.

```
/*********************************************************
 *            Xgraphics demo program                     *
 *********************************************************/

#include <math.h>
#include "Xgraphics.h"
```

Zur Verwaltung der Fenster benötigte Variablen:

```
Window mywindow;
World  myworld;
XEvent myevent;
```

Hauptprogramm:

```
main(int argc, char **argv) {

  int i,done = 0;        /* control variable */
```

Zunächst muß die Verbindung zum X-Server hergestellt werden:

```
  InitX();             /* connect to X-server */
```

Festlegung der Fenster und Zeichenbereiche:

```
/* set up windows and worlds */

  mywindow = CreateWindow(
                          400,400,    /* width, height       */
                          "demo1"     /* name of window      */
                          );

  myworld = CreateWorld(
                        mywindow,      /* window to draw in  */
                        5,5,390,390,   /* x,y,width,height   */
                        -1,-1,1,1,     /* wx1, wy1, wx2, wy2 */
                        0,             /* scaling property   */
                        0              /* "gravity"          */
                        );
```

Darstellen des Fensters und Zeichnen einer Lissajous-Figur:

```
  ShowWindow(mywindow);

  for(i=1;i<620;++i) WDrawCircle(myworld,cos(i),sin(2*i),0.1,1);
```

Event-Behandlung: Warten auf einen Mausklick.

```
  while(!done) {
    GetEvent(&myevent,1);
    if (myevent.type == ButtonPress) done =1;
  }
```

Sauberes Beenden des Programms:

```
  ExitX();                                    /* clean up memory */
  return 0;
}
```

Das folgende Programm demonstriert die Verwendung von Zeichenbereichen mit verschiedenen Skalierungseigenschaften, die Verwendung von Buttons bei einer Verzweigung in ein Unterprogramm mit eigenen Buttons, das Auslesen der Mauskoordinaten und die Behandlung des Expose-Events für skalierende Zeichenbereiche.

```
/*********************************************************
 *            Xgraphics demo program                     *
 *********************************************************/

#include <math.h>
#include "Xgraphics.h"

Window mywindow, subwindow;
World  world1, world2, world3, world4, textworld;
XEvent myevent;

void redraw();
void submenu();
```

Hauptprogramm:

```
main(int argc, char **argv) {

  int done = 0;        /* control variable */
  float x,y;           /* coordinates      */

  InitX();             /* connect to X-server */

/* set up windows and worlds */

  mywindow = CreateWindow(
                          600,400,    /* width, height  */
                          "demo1"     /* name of window */
                          );

  subwindow = CreateWindow(
                           300,200,
                           "submenu"
                           );
```

Dieses Programms enthält fünf Zeichenbereiche. Drei von diesen sind skalierbar. Damit diese sich beim Vergrößern/Verkleinern des Fensters nicht gegenseitig durchdringen muß angegeben werden an welcher Seite eines solchen ein weiterer skalierbarer Zeichenbereich angrenzt. Dies geschieht mit den `FLOAT_xxx` *- Schaltern.*

```
world1 = CreateWorld(
                     mywindow,      /* window to draw in  */
                     5,5,245,122,   /* x,y,width,height   */
                     0,0,1,1,       /* wx1, wy1, wx2, wy2 */
                     FLOAT_SOUTH +
                     SCALABLE,      /* scaling property   */
                     0              /* "gravity"          */
                     );

world2 = CreateWorld(mywindow,
                     5,128,122,122,
                     0,0,1,1,
                     FLOAT_NORTH +
                     FLOAT_EAST +
                     SCALABLE,
                     0
                     );

world3 = CreateWorld(mywindow,
                     128,128,122,122,
                     0,0,10,10,
                     FREE_ASPECT +
                     FLOAT_NORTH +
                     FLOAT_WEST +
                     SCALABLE,
                     0
                     );
```

Bei den nicht-skalierbaren Welten muß angegeben werden, wie sie sich bei Vergrößern/Verkleinern des Fensters relativ zu den Rändern des Fensters bewegen. Die Ecke oder der Rand des Fensters, mit dem sich der Zeichenbereich bewegen soll wird durch die Richtungsangabe in der `gravity`*-Konstante festgelegt.*

```
world4 = CreateWorld(mywindow,
                     255,5,245,245,
                     0,0,10,10,
                     0,
                     NorthEastGravity);

textworld = CreateWorld(mywindow,
                     5,300,490,95,
                     0,0,489,94,
                     0,
                     SouthWestGravity);
```

Definition der Buttons:

```
InitButtons(mywindow,
            "t,commands:,,b,exit,e,b,menu,m,b,clear,c",
            100);
```

Nun muß das Fenster noch sichtbar gemacht werden und dann kann gezeichnet werden.

```
  ShowWindow(mywindow);

/* startup things */

  WDrawString(textworld,10,15,
              "Press any Mousebutton in the right field",1);
  WDrawString(textworld,10,30,
              "and resize the Window.",1);

  WDrawRectangle(world4,0,0,10,10,1);
  redraw();
```

In `redraw()` *werden die skalierbaren Zeichenbereiche dargestellt. Die Hauptroutine des Programms ist in dden meisten Fällen die Bedienung der Events. Von hier aus werden, je nach eintretenden Event, die verschiedenen Routinen des Programms aufgerufen.*

```
/* main loop */

  while(!done) {

    if( GetEvent(&myevent,1) ) {              /* event handling  */
```

In der `GetEvent()` *Funktion ist hier das* `wait_flag` *auf* `1` *gesetzt. Das Programm wartet also auf Events, ohne Rechenzeit zu verbrauchen.*

```
      switch(myevent.type) {
```

Behandlung der `KeyPress`*-Events, ausgelöst durch eine Taste oder durch einen Mausclick in eines der durch* `InitButtons()` *definierten Felder.*

```
      case KeyPress:                          /* read keyboard   */
        switch ( ExtractChar(myevent) ) {     /* or button       */
        case 'e':
          done = 1;
          break;
        case 'c':
          ClearWorld(world4);
          WDrawRectangle(world4,0,0,10,10,1);
          break;
        case 'm':
          submenu();
          break;
        default:
          break;
        }
        break;
```

Mausaktionen: Auslesen der Mauskoordinaten.

```
      case ButtonPress:                       /* mouse click     */
        WGetMousePos(world4,myevent,&x,&y);   /* get coordinates */
        switch (myevent.xbutton.button) {
        case 1:
          WDrawCircle(world4, x, y, 0.2, 1);
```

```
          break;
        case 2:
          WDrawCircle(world4, x, y, 0.4,4096+256*GXxor+ 1);
          break;
        case 3:
          WFillCircle(world4, x, y, 0.2, 1);
          break;
        default:
          break;
        }
        break;
```

Das Fenster wird neu gezeichnet. Die nicht-skalierbaren Zeichenbereiche werden automatisch wiederhergestellt. Um die anderen muß man sich selber kümmern. Am besten definiert man eine Funktion `redraw()`*, die dies übernimmt, und im Falle eines* `Expose`*-Events nur aufgerufen werden muß.*

```
      case Expose:                          /* window must get */
        redraw();                           /* redrawn         */
        break;

      default:
        break;
      }
    }
  }

/* that´s it */
```

Am Ende des Programms sollte anständig aufgeräumt werden. `ExitX()` *sorgt dafür, daß alle noch offenen Fenster geschlossen werden und gibt von Xgraphics verwendeten Speicher wieder frei.*

```
  ExitX();                              /* clean up memory */
  return 0;

}
```

In `redraw()` *werden die skalierbaren Welten gezeichnet. Außerdem werden hier zwei Möglichkeiten zur Auswahl von Farben demonstriert.*

```
void redraw() {
/*
   Draw the scalable world.
   This function must be called at an Expose-Event.
*/

  int i,j;
  WPoint points[100];

  for(i=0; i<100; i++) {
    points[i].x = i/100.;
    points[i].y = ( sin(i * 2. * 0.031415) + 1) / 2.;
  }
  WDrawRectangle(world2,0,0,1,1,1);
  WDrawLines(world2,points,100,1);
```

```
  WDrawRectangle(world1,0,0,1,1,1);
  for (i=1;i<20;++i)
    WDrawLine(world1,0,(float)i/20.,1,1.-(float)i/20.,
              i%(NofColors-1)+1);
```

Diese Farben sollen immer verschieden von der Hintergrundfarbe sein.

```
  WDrawRectangle(world3,0,0,10,10,1);
  for( i=1;i<=10; ++i )
    for( j=1; j<=10; ++j )
      WFillRectangle(world3,i-1,j-1,i,j,2+(i+j)%2);
```

Diese Farben sollen sich immer voneinander unterscheiden.

```
}
```

Beispiel eines Unterprogramms mit eingenem Fenster und eigenem Menu. Es ist sinnvoll das Fenster und die hierin verwendeten Zeichenbereiche global zu definieren. Dadurch erspart man, daß bei jedem Aufruf der Routine die Resourcen neu belegt werden müssen.

```
void submenu() {

  int done=0;

  ShowWindow(subwindow);

  InitButtons(subwindow,
              "t,submenu:,,b,main,m",
              100);

  while(!done) {

    if( GetEvent(&myevent,1) ) {              /* event handling */
      switch(myevent.type) {

      case KeyPress:
        switch ( ExtractChar(myevent) ) {
        case 'm':
          done = 1;
          break;
        default:
          break;
        }
        break;

      case Expose:
        redraw();
        break;

      default:
        break;
      }
    }
  }
```

Bevor man in das Hauptprogramm zurückkehrt muß man die Buttons des Hauptprogramms wiederherstellen. Es ist sinnvoll, dies noch in dem Unterprogramm zu tun, so daß das Hauptprogramm unabhängig davon ist, ob ein Unterprogramm eigene Buttons verwendet.

```
  InitButtons(mywindow,                  /* reset to main-buttons */
              "t,commands:,,b,exit,e,b,menu,m,b,clear,c",
              100);

  HideWindow(subwindow);
}
```