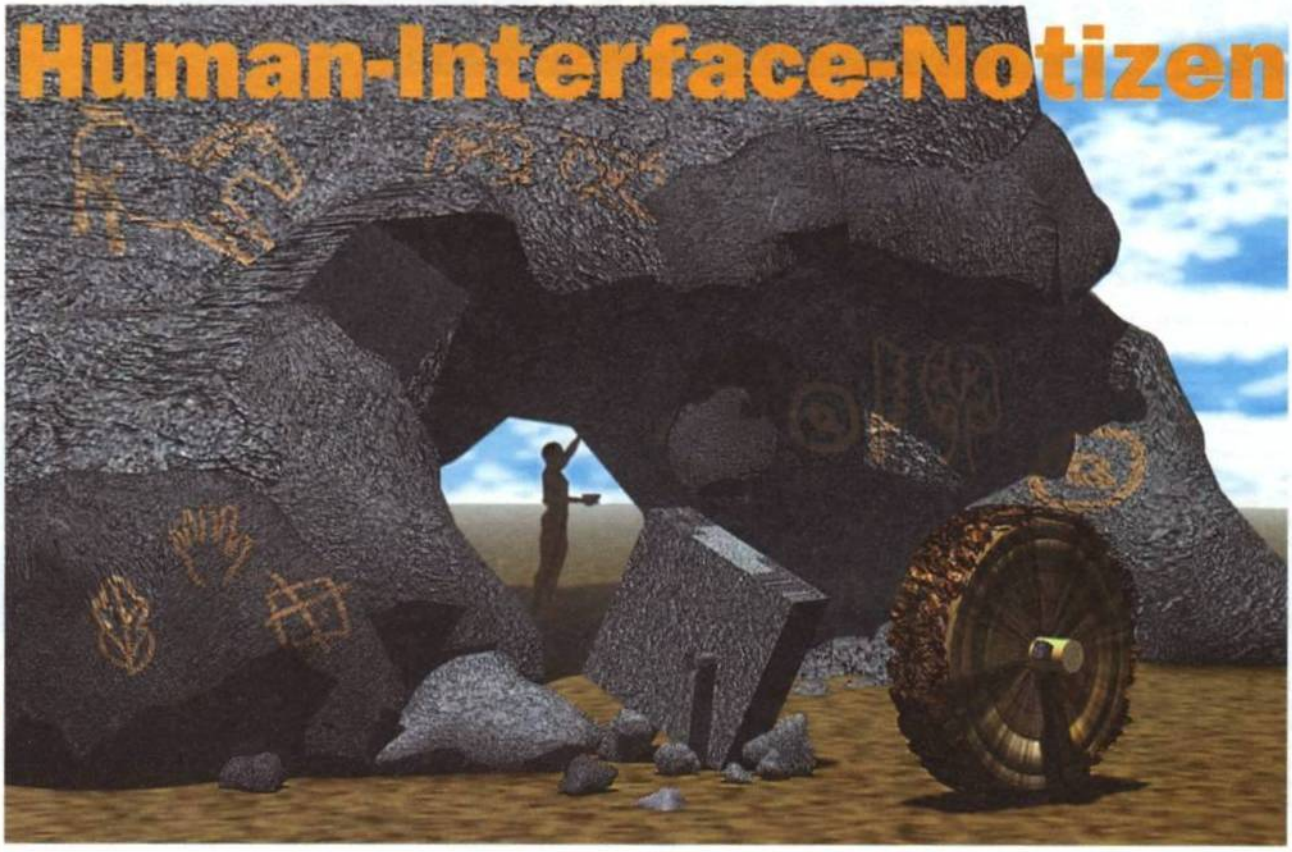
Human-Interface-Notizen



Benutzeroberflächen sind so alt wie Menschheit. Früh lernten die Menschen ihre Werkzeuge so zu verbessern, daß man so einem Ding die Handhabung ansehen konnte. Warum das Rad zweimal erfinden? In der dinglichen Welt kümmern sich ergonomische Richtlinien um die Menschenfreundlichkeit eines Gebrauchsgegenstandes. In der digitalen Welt sind es die Human-Interface-Guidelines, die uns die Computerbenutzung erleichtern.

Eine moderne Benutzeroberfläche arbeitet mit Metaphern. Begriffe wie Schreibtisch, Ordner und Papierkorb kennt der Benutzer und kann sie direkt zuordnen. Da der Mensch sich Zusammenhänge besser merken kann als einzelne Dinge, spart er sich hier das erneute Lernen. Natürlich sollte man nicht mit Gewalt versuchen, alles in ein Schreibtisch-Modell einzupassen. Schnell ist man dann an einem Punkt angelangt, wo man die Vorteile des Computers nicht mehr nutzt.

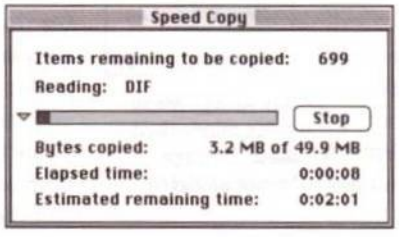
Der Benutzer sollte Objekte direkt verändern können. Wer würde einen Locher auf dem Schreibtisch ansehen und dann ein Buch hervorholen, in dem man einen Zauberspruch nachschlagen muß, um diesen Locher in eine Schublade zu legen? Für ein paar Dinge kann man sich den dazugehörigen Spruch noch merken, aber häufig muß man doch nachschlagen. Wäre es nicht viel einfacher, den Locher mit der Hand in die Schublade zu legen? Nun, das ist der Grund, warum es so wenige Zauberer gibt und so viele Bürokräfte ... Für den Computer bedeutet dies: Möglichst viele Dinge sollte man direkt am Objekt erledigen können (z.B. Verschieben, Kopieren, Löschen). Einige wenige – häufig benutzte – Dinge sollte man zusätzlich per Tastatur-Shortcut anbieten.

Dabei gilt fast immer folgendes: Der Benutzer wählt ein oder mehrere Objekte und macht dann etwas mit ihnen. Also: „Datei" - „Kopieren", „Ordner" - „Löschen". Die Objekte sind im Fenster oder auf dem Schreibtisch zu sehen. Der Benutzer kann sie auf verschiedene Weisen anwählen (einzelne, mehrere, alle) und dann entweder die Aktion direkt auslösen (z.B. Verschieben) oder über einen Menüpunkt wählen. Diese Regel sollte man möglichst immer einhalten. Auch sollte z.B. die Operation „Kopieren" mit allen Objekten funktionieren – der Benutzer wundert sich über Ausnahmen und versteht eventuell nicht, warum dies oder jenes nun nicht funktioniert. Wie kann man einem Benutzer klarmachen, daß von den vielen gleichzeitig ausgewählten Objekten einige nicht kopierbar sind? Häufig ist es für den Programmierer einfacher, diese Einschränkung zu beseitigen, als hier Sonderfälle einzuführen.

Der Computer verhält sich üblicherweise passiv: er wartet auf Anweisungen durch den Benutzer, und dieser erwartet, daß der Computer sich genauso verhält. Nichts ist ärgerlicher als ein „vorlauter" Computer, der meint, die Gedanken des Benutzers zu erraten und dabei doch ständig daneben liegt. Wenn man sein Programm schon mit KI versieht, dann sollte man sie entweder sehr dezent einsetzen (z.B. kann man sinnvolle Vorgaben in Dialogboxen machen) oder aber sie abschaltbar gestalten. Merke: Der Benutzer kontrolliert den Rechner und nicht umgekehrt! Bestenfalls darf der Computer den Benutzer davor warnen, etwas Gefährliches zu tun, aber die Kontrolle behält der Anwender.

Dazu gehört es auch, den Benutzer zu informieren, wenn der Computer etwas macht, z.B. das Kopieren von tausenden von Dateien grafisch zu visualisieren. Auch „verkürzt" eine

solche Visualisierung die Wartezeit: wenn etwas auf dem Bildschirm passiert und der Benutzer abschätzen kann, wie lange es dauert, dann kommt es ihm nicht so lange vor! Dabei darf man nicht übertreiben: sonst glaubt der Benutzer am Ende noch, daß diese Darstellung mehr Rechenzeit kostet als der eigentliche Vorgang. Eine bessere Lösung wäre es, wenn der Benutzer gar nicht warten müßte: sei es, weil ein besserer Algorithmus schneller ist; sei es, weil man sein Programm modeless halten kann. So kann man z.B. mit Hilfe des Thread-Managers Aktionen parallel ablaufen lassen.



Sogenannte „Progressbars" verkürzen, geschickt eingesetzt, subjektiv die Wartezeit.

Ein brauchbares Feedback ist aber immer noch viel seltener zu finden als eine brauchbare Nutzung des Thread-Managers. Die Meldung „Fehler -108" sagt deutlich weniger aus als „Der Speicher ist voll" - trotzdem sieht man sie recht häufig. Noch besser wäre: „Das Kopieren konnte nicht beendet werden, weil der freie Speicher nicht reicht". Und noch besser wäre es, wenn ein Lösungsvorschlag genannt würde - z.B. „Teilen Sie dem Programm mehr Speicher zu und versuchen Sie es nocheinmal". Und optimal wäre es, wenn wir - als Programm - den Fehler selbst behöben.

Zur modernen Oberfläche gehört natürlich auch das berühmte WYSIWYG (What you see is what you get). Der Benutzer erwartet z.B., daß seine formatierte Seite beim Ausdruck nicht das Aussehen verändert.

Internationales Design

Zu einem guten Programm gehört auch, daß es nicht nur in Deutschland brauchbar funktioniert. Selbst für Shareware-Programmierer lohnt sich z.B. der Aufwand einer englischen Version. Allerdings gibt es dabei einiges zu beachten:

Alle Texte sollten in ‚STR#'- bzw. ‚STR ,- Resourcen stehen. Dies ermöglicht es, das Programm ohne Änderung des Programmcodes zu übersetzen. Mit Hilfe des Programms „AppleGlot" von Apple kann man sich diesen Vorgang zudem sehr erleichtern, gerade wenn es darum geht, neuere Programmversionen zu erzeugen, in denen sich nur Teile geändert haben. Wir sollten übrigens daran denken, daß wir nicht einzelne Wörter auslagern, um zur Laufzeit daraus Sätze zu bilden, denn der Satzbau ist in unterschiedlichen Sprachen ebenfalls völlig unterschiedlich! Zu einem ordentlichen Programm gehört natürlich auch, daß es keine Probleme mit 8-Bit-Zeichen hat! Hiermit haben besonders C-Programmierer teilweise Probleme, die ihre Texte als „char*" beschreiben. Der Datentyp „char" ist nämlich vorzeichenbehaftet, und ein Vergleich „c < 32" z.B., um Steuerzeichen zu erkennen, filtert ebenfalls alle Sonderzeichen >127 heraus, die nämlich negative Zahlen darstellen! Im MacOS wird deswegen mit StringPtr geabeitet, der einen Zeiger auf „unsigned char" darstellt. Wer den Aufwand in seinem Programm noch etwas erhöhen und die Flexibilität vergrößern will, kann auch die Informationen, welcher Zeichensatz in welcher Größe für den Text genommen wird, in eine extra Resource auslagern – die Klassen-Library MacApp nutzt dafür ‚TxSt'-Resourcen – ein Beispielprogramm zum Setzen des Zeichensatzes aus einer Resource folgt am Ende des Artikels. Wir müssen in unserem Programm eh zur Laufzeit die Höhe des Texte abfragen, um den Zeilenabstand zu bestimmen, dann kommen wir auch automatisch mit anderen Einstellungen zurecht.

Mit der Auslagerung der Texte in die Resource-Fork ist es aber nicht getan, denn unser Programm muß flexibel genug sein, um z.B. mit unterschiedlichen Textlängen zurechtzukommen. Wir müssen also Teile der Formatierung zur Laufzeit durchführen. Richtig aufwendig wird es dabei, wenn wir nicht nur Versionen für romanische Sprachen erzeugen wollen: Hebräischer Text wird von rechts nach links geschrieben - wodurch wir unsere Formatierung umdrehen müssen; japanischer bzw. chinesischer Text zeichnet sich dadurch aus, daß ein Zeichen auf dem Bildschirm aus zwei Bytes im Speicher besteht (die Zeichensätze umfassen jeweils viele tausend Zeichen). Das erinnert uns daran, daß der Systemzeichensatz übrigens nicht zwangsweise Chicago und der Applikationszeichensatz nicht zwangsweise Geneva ist! Näheres zur Ausgabe und Editierung von Texten findet sich im Inside Macintosh: Text.

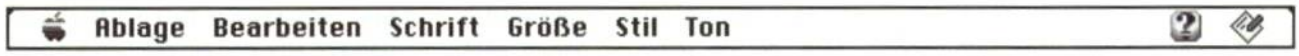
Für Bildschirmdialoge sollten wird bevorzugt den Dialog-Manager des MacOS nutzen, denn dieser ermöglicht nicht nur das einfache Übersetzen von Texten innerhalb von Dialogboxen, er ermöglicht auch die Korrektur des Layouts einer Dialogbox, wenn ein Button zu klein für den übersetzten Text ist.

Zu den Script-Systemen zählt übrigens auch die Möglichkeit, daß ein Benutzer zwei Tastaturen am Mac angeschlossen, hat z.B. eine internationale und eine nationale mit Sonderzeichen. Für das Programm ist dabei jeweils die Tastatur aktuell, auf der der Benutzer zuletzt getippt hat. Aber auch bei nur einer vorhandenen Tastatur müssen wir daran denken, daß wir möglichst nie mit Scancodes arbeiten sollten, denn die Taste mit dem Scancode $01 kann auf zwei Tastaturen völlig unterschiedliche ASCII-Codes erzeugen! Wir sollten daher möglichst immer den ASCII-Code abfragen. Übrigens: auch die Cursor-Tasten etc. erzeugen ASCII-Codes.

Aber nicht nur an fremde Sprachen sollten wir denken, auch an Benutzer mit mehr oder weniger großen Behinderungen, die den Recher nutzen!

Als erstes wäre da die Gruppe der Leute mit Sehschwächen. Hier sollte es die Möglichkeit geben, in unserem Programm mit unterschiedlichen Zoom-Stufen zu arbeiten. Das ist nicht nur nützlich für Leute, die kleine Zeichen nicht gut lesen können, sondern auch für eine Seitenübersicht (indem man die Zoom-Stufe z.B. auf 25% stellt). Eine sehr verbreitete Sehschwäche ist auch die Farbenblindheit. Gerade die Rot/Grün-Blindheit sollte uns zu denken geben, wenn wir eine Signalleuchte entweder in Rot oder Grün aufleuchten lassen wollen, um dem User ein Feedback zu geben ... Wir sollten dem

Human-Interface-Notizen



Die Menüleiste von SimpleText geht mit gutem Beispiel voran: Die Reihenfolge der Menütitel muß beachtet werden.

Benutzer also zusätzliche Unterscheidungsmöglichkeiten anbieten.

Ebenfalls recht verbreitet sind Hörschwächen. Es gilt das gleiche wie bei den Sehschwächen: wir sollten also nicht zwangsweise einen *SysBeep()* als einzigen Userfeedback geben. Das Problem haben übrigens auch Benutzer, die die Lautstärke zurückgedreht haben, damit sie beim Spielen nicht die anderen Angestellten im Büro nerven ... Weitere Behinderungen sind Sprachstörungen - hier wird der Computer häufig zur Erzeugung von Sprache eingesetzt (Stichwort: PlainTalk und Speech Manager) – oder Personen, die aufgrund von Behinderungen keine Tastatur nutzen können. Wir brauchen nicht zwangsweise unser gesamtes Programm auf reinen Mausbetrieb o.ä. umzustellen, aber wir sollten zumindest keine festen Annahmen über die Art der Tastatur machen, wenn es nicht unbedingt nötig ist. Hier ist auch die Möglichkeit zu erwähnen, mittels des sehr praktischen Kontrollfeldes „PopChar" Sonderzeichen einzugeben.

Trotz Reglementierung recht vielseitig einsetzbar: das Apfel-Menü

Menüs

Kommen wir nun zur eigentlichen Benutzeroberfläche. Als erstes der Aufbau der Menüs. Jedes Menü in einem Mac-Programm hat Teile, die stets gleich sind. So sind das Apfel-, das Ablage- und das Bearbeiten-Menü sowie das Hilfe- und das Applikation-Menü in allen Programmen vorhanden. Hat der Benutzer mehrere Script-Systeme installiert, so existiert noch ein Script-Menü neben dem Hilfe-Menü. Die weiteren Menütitel sind jedoch vom jeweiligen Programm abhängig. Die Menüleiste sollte sich während des Programmlaufes nicht ständig verändern - man verwirrt nur den Benutzer, wobei es in komplexen Programmen natürlich trotzdem sinnvoll sein kann - Beispiel: ClarisWorks, welches unterschiedliche Menüs für die einzelnen Programmteile hat (Datenbank, Textverarbeitung, usw.). Achtung: die Menüleiste sollte auch auf einem 9"-Monitor noch auf den Bildschirm passen!

Das Apfel-Menü enthält neben dem Aufruf der üblichen About-Box und dem Inhalt des Apple-Menü-Ordners (innerhalb des Systemordners) keine weiteren Menüpunkte. Unter System 6 war hier häufig noch ein Hilfe-Menüpunkt zu finden, doch seit System 7 sollte man eigene Hilfe-Menüpunkte an das Hilfe-Menü anhängen.

Das Ablage-Menü sollte mindestens diese Einträge immer anbieten

Das Ablage-Menü – auf anderen Rechner auch Datei-Menü genannt – hat eine relativ strenge Aufteilung:

Der Menüpunkt „Neu" legt dabei ein neues Dokument an, „Öffnen" öffnet ein eben solches. Mit „Schließen" kann man – wer hätte es gedacht? – ein Dokument schließen. Wurde es allerdings verändert, erhält der Benutzer eine Anfrage, ob er das Dokument vorher speichern möchte oder nicht. Die gleiche Abfrage erscheint ebenfalls, wenn der Benutzer das Programm beendet. Mit „Speichern" wird das aktuelle Dokument gesichert, mit „Speichern unter" kann es unter einem anderen Namen gesichert werden. „Papierformat" erlaubt das Einstellen des Druckers, „Drucken" druckt das aktuelle Dokument. Mit QuickDraw GX wurde zudem „eine Kopie Drucken" eingeführt, welches die Drucken-Dialogbox umgeht und sofort eine Kopie ausdruckt. Der letzte Menüpunkt schließlich ist „Beenden". Das „Ablage" Menü ist somit schon recht lang, und man sollte es nur im Notfall noch weiter verlängern. Die Tastatur-Shortcuts sind übrigens vorgegeben und sollten unter keinen Umständen verändert werden!

Wie man sieht, geht Apple davon aus, daß ein Programm stets beliebig viele offene Dokumente haben kann, die jeweils einem Fenster entsprechen. Dies ist natürlich für den Benutzer auch die übersichtlichste Art. Allerdings gibt es auch Programme, die nur ein offenes Dokument erlauben - aus welchen Gründen auch immer. Diese disablen einfach den „Neu"- und „Öffnen"-Menüpunkt, wenn das Dokument offen ist. Deutlich komplizierter wird der Fall, wenn ein Dokument aus vielen Fenstern bestehen kann! Was bedeutet in diesem Fall „Schließen"? Das oberste Fenster schließen? Das Dokument schließen? In einem solchen Fall wird man üblicherweise ein eigenes „Fenster"-Menü einführen, wo der Benutzer neben der Sortierung seiner vielen Fenster auch bestimmte Fenster öffnen und schließen kann. Bleibt die Frage was Command-W bedeutet - normalerweise ist es ja einfach „Schließen". Ich persönlich tendiere dazu mit Command-W ein Fenster zu schließen und beim letzten offenen Fenster schließlich das ganze Dokument.

Auch die Einträge des Bearbeiten-Menüs sind fest vorgegeben.

Beim Bearbeiten-Menü sind nur die oberen Menüpunkte vorgegeben: Undo/Redo, dann eine Linie und Ausschneiden, Kopieren, Einfügen, Löschen. Üblicherweise folgt noch ein „Alles auswählen". Für diese 6 Menüpunkte sind ebenfalls Tastatur-

Human-Interface-Notizen

Shortcuts vorgeben, die keinesfalls anders genutzt werden sollten! Auch für den Fall, daß man Cut/Copy/Paste nicht direkt selbst braucht, sind diese Menüpunkte doch wichtig, denn alle Dialogboxen mit Eingabefeldern nutzen dieses Menü.

Noch etwas ganz allgemeines zu Menüs: Bevor man seine Menüs entwirft, sollte man sich andere ähnliche Programme ansehen. Nirgendwo lernt man so gut, wie bei der Konkurrenz! Und wenn sie auch nur als schlechtes Beispiel dient ...

Weiteres

Im nächsten Artikel werde ich dann mit Informationen zum Design von Dialogboxen fortfahren. Unbedingt ans Herz legen möchte ich das Buch „Human Interface Guidelines". Es ist ein Pflichtwerk für jeden Macintosh-Programmierer!

Die „Making It Macintosh",CD-ROM ist eine hervorragende Einführung in die Art, wie man sein Programm „Mac-Like" gestaltet. Nebenbei ist sie eine gute Demonstration, wie man eine Präsentation ansprechend gestaltet.

Leider ist die CD nicht ganz billig, und nach dem Durchwandern wird man sie kaum wieder ansehen, weil man die Informationen (hoffentlich) verinnerlicht hat, was einerseits schade ist, andererseits zeigt, wie gut die CD gemacht ist.

MFR

Quellen:

Macintosh Human Interface Guidelines
Addison-Wesley,
ISBN 0-201-62216-5 ($29.95)

Making It Macintosh (CD-ROM)
Addison-Wesley,
ISBN 0-201-62626-8 ($39.95)


```
// Beispielprogramm zur Demonstration
// der Human Interface Guidelines
// (c) 1996 MAXON Computer
// Autor: Markus Fritze

// FontUtilities.h

typedef struct {
    short       txFont; // Font-ID
    short       txMode; // Transfer-Modus
    short       txSize; // Text-Größe
    Style       txFace; // Text-Stil
    RGBColor    fGround;// Vordergrundfarbe
    RGBColor    bGround;// Hintergrundfarbe
    long        fgColor;// Farben o. ColorQuickdraw
    long        bkColor;
} FontStruct;

extern FontStruct       fDefault;

void        SaveText(FontStruct *f);
void        RestoreText(const FontStruct *f);
short       SetFont(short resid);


// FontUtilities.c

// gQDVersion hat einen Wert > 0, wenn ColorQuickDraw
// vorhanden ist. Die Version kann man mit
// abfragen Gestalt(gestaltQuickdrawVersion, &val).
// Der Rückgabewert ist dann:
// gQDVersion = (val > 8) & 0xFF

/***
 *  aktuelle Fonteinstellungen retten bzw setzen
 ***/
// ein Default-Font: System-Font, OR-Copymode,
// Default-Größe, kein spezieller Stil
FontStruct  fDefault = { 0,srcOr,0,0,
                         { 0x0000, 0x0000, 0x0000},
                         { 0xFFFF, 0xFFFF, 0xFFFF},
                         blackColor, whiteColor };

void        SaveText(FontStruct *f)
{
    f->txFont = qd.thePort->txFont;
    f->txMode = qd.thePort->txMode;
    f->txSize = qd.thePort->txSize;
    f->txFace = qd.thePort->txFace;
    f->fgColor = qd.thePort->fgColor;
    f->bkColor = qd.thePort->bkColor;
    if(gQDVersion > 0) {
        GetForeColor(&f->fGround);
        GetBackColor(&f->bGround);
    }
}

void        RestoreText(const FontStruct *f)
{
    TextFont(f->txFont);
    TextMode(f->txMode);
    TextSize(f->txSize);
    TextFace(f->txFace);
    ForeColor(f->fgColor);
    BackColor(f->bkColor);
    if(gQDVersion > 0) {
        RGBForeColor(&f->fGround);
        RGBBackColor(&f->bGround);
    }
}

/***
 *  Font gemäß einer TxSt-Resource einstellen
 *  Zeilenabstand bei diesem Zeichensatz zurückgeben
 ***/
short           SetFont(short resid)
{
typedef struct {    // TxSt-Resource-Format
    Style       stil;
    short       size;
    RGBColor    color;
    Str255      fontname;
} TxStStruct;

short           fontNum;
Str255          systemFontName;
TxStStruct      **txs;
TxStStruct      *txp;

    // Resource suchen
    txs = (TxStStruct**)GetResource('TxSt', resid);
    if(!txs || !*txs || ResError()) {
    error:
        DebugStr("\pLesefehler bei TxSt-Resource!");
        return 0;
    }

    HLock((Handle)txs);
    txp = *txs;
    // Nummer zu einem Fontnamen ermitteln
    GetFNum(txp->fontname, &fontNum);
    // Font 0 = System-Font bzw. Font nicht gefunden?
    if(fontNum == 0) {
        // Namen vom System-Font wählen
        GetFontName(systemFont, systemFontName);
        // nicht der System-Font?
        if(!EqualString(txp->fontname, systemFontName,
        false, false))
            goto error;     // => raus
    }
    // Font setzen
    TextFont(fontNum);

    // Fontstil setzen
    TextFace(txp->stil);

    // Fontgröße setzen
    TextSize(txp->size);

    // RGB-Color setzen
    if(gQDVersion > 0)
        RGBForeColor(&txp->color);

    HUnlock((Handle)txs);
    ReleaseResource((Handle)txs);

    // Zeilenabstand bei diesem Zeichensatz zurückgeben
    FontInfo        fInfo;
    GetFontInfo(&fInfo);
    return fInfo.ascent + fInfo.descent + fInfo.leading;
}
```