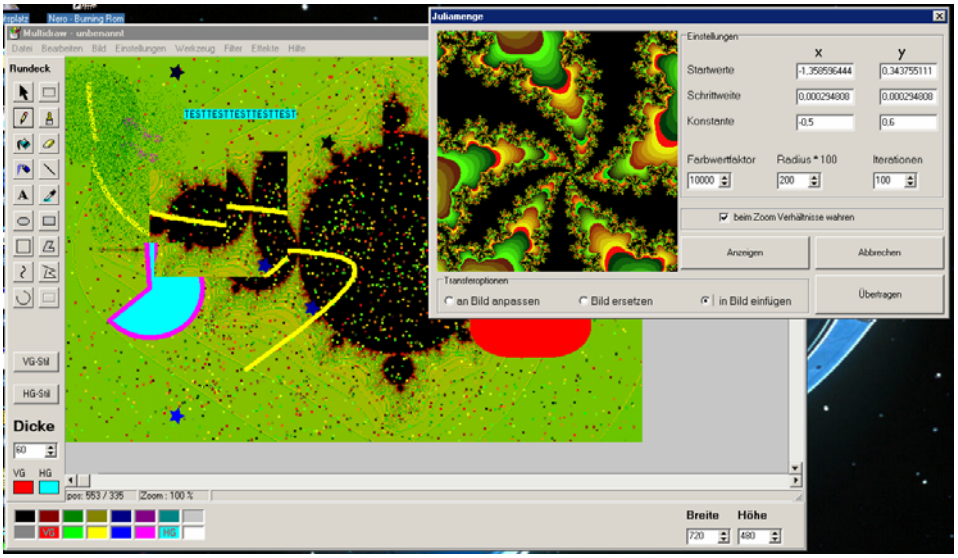
BG/BRG Wels Brucknerstraße

Fachbereichsarbeit

aus

Informatik

Grafikprogrammierung unter Delphi 3
Überblick und Beispiele



Eingereicht von: Christoph Saulder Klasse: 8N

Betreuungslehrer: Mag. Jürgen Rathmayr

Abgegeben am:

Inhalt

Inhalt 2
Prolog 4
1. Grafikprogrammierung allgemein 5
1.1 Grundlagen der Grafikprogrammierung 5
1.2 Begriffe der Grafikprogrammierung 5
1.2.1 Pixel 5
1.2.2 Vektor 6
1.2.3 Auflösung 6
1.2.4 Farbmodelle 6
1.2.5 2D Grafiken – 3D Grafiken 7
1.3 Geschichte: Immer neue Möglichkeiten 7
1.4 Anwendungen von Computergrafiken 9
1.5 Grafikspeicherung 10
1.5.1 Das Format .avi 10
1.5.2 Das Format .bmp 11
1.5.3 Das Format .cgm 11
1.5.4 Das Format .eps 11
1.5.5 Das Format .gif 11
1.5.6 Das Format .ico 12
1.5.7 Das Format .jpg 12
1.5.8 Das Format .mac 12
1.5.9 Das Format .mpg 12
1.5.10 Das Format .pcx 12
1.5.11 Das Format .pdf 13
1.5.12 Das Format .pict 13
1.5.13 Das Format .png 13
1.5.14 Das Format .tga 13
1.5.15 Das Format .tif 14
1.5.16 Das Format .wmf 14
2. Grafikprogrammierung in Delphi 3 15
2.1 Möglichkeiten in Delphi 3 15
2.2 Die Canvas 15

2.3 Grafikobjekte ... 17
2.3.1 TBitmap ... 17
2.3.2 TPaintbox ... 18
2.3.3 TTurtle ... 19
2.3.4 TImage ... 20
2.3.5 TAnimate ... 21
2.3.6 TPrinter ... 21
3. Ausgewählte Beispiele aus meinem Programm ... 23
3.1 Einfache Zeichenwerkzeuge ... 23
3.1.1 Der Pinsel ... 23
3.1.2 Ein Rechteck ... 25
3.1.3 Das Tortenstück ... 27
3.1.4 Farbradierer ... 30
3.2 Die RGB-Zerlegung ... 32
3.3 Funktionen und Fraktale ... 33
3.3.1 Mandelbrot-Menge ... 34
3.3.2 Rekursionen ... 36
3.3.3 Polynomfunktionen ... 39
3.3.4 trigonometrische Funktionen ... 40
3.4 Rotation und Spiegelung ... 41
3.4.1 Rechter Winkel ... 41
3.4.2 Genauer Winkel ... 42
3.4.3 Spiegelungen an den Achsen ... 45
3.4.4 Spiegelungen an den Meridianen ... 46
3.5 Formen – Objektorientiert ... 47
3.6 Anwendung von TMediaplayer ... 49
3.7 Laden und Speichern von Bildern ... 50
3.7.1 Speichern ... 50
3.7.2 Laden ... 51
3.7.3 JPG-Bilder ... 53
3.8 Verknüpfung von mehreren Formularen ... 54
3.9 Lokalisierung des Bildschirmschoners ... 56
Epilog ... 59
Quellenverzeichnis ... 60

Prolog

Ich habe mich entschlossen das Thema „Grafikprogrammierung in Delphi 3„ für meine Informatik-FBA zu wählen, weil ich mich schon seit längerem mit dieser Programmiersprache beschäftige. Seit zwei Jahren programmiere ich bereits Computerspiele und um sie „etwas gleich schauen zu lassen„ musste ich mich auch mit Grafikprogrammierung beschäftigen. Da ich beabsichtige mein Wissen in diesem Bereich auszuweiten und zu vertiefen, entschied ich mich ein dazu passendes Thema für meine FBA zu wählen. Ich habe meine Arbeit in drei Abschnitte gegliedert von denen jeder einen Aspekt der Thematik behandelt. Einerseits versuche ich im ersten Teil dieser Dokumentation mich allgemein mit Grafikprogrammierung zu beschäftigen, doch im zweiten Teil spezialisiere ich mich schon auf die Möglichkeiten, die dem Programmierer in Delphi 3 zur Verfügung stehen. Im letzten Abschnitt werde ich ganz konkret und beschreibe die Umsetzung von diversen Methoden anhand ausgewählter Beispiele, welche alle ein Teil meines Programms Multidraw sind.

1. Grafikprogrammierung allgemein

In diesem ersten Abschnitt behandle ich die grundlegenden Aspekte der Thematik meiner Arbeit. Ich versuche hier die im Hintergrund stehende Theorie zu verdeutlichen und einen Überblick über Grafikprogrammierung zu schaffen. Im Grunde genommen ist Grafikprogrammierung das Mittel mit dem man die abstrakten Vorgänge im Computer visuell darstellt und für den Anwender erkennbar macht.

1.1 Grundlagen der Grafikprogrammierung

Das gängige Betriebssystem Windows ist Grafik pur. Text wird nun sogar als eine besondere Form von Grafik aufgefasst, wodurch die in DOS übliche Unterscheidung zwischen Text- und Grafikmodus nicht mehr existiert. Man kann (vor allem in Delphi) drei grundsätzliche Varianten von Grafikprogrammierung unterscheiden: Anzeige von „vorgefertigten„ Grafikdateien in speziellen Container-Komponenten, Verwendung von vordefinierten Formen, Einsatz von Grafikmethoden auf der Grafikoberfläche einer Komponente. Da die ersten beide Möglichkeiten bereits zu Entwurfszeit ihre Anwendung finden, kann man erst bei der letzten von eigentlicher Grafikprogrammierung sprechen, da diese zur Laufzeit und ausschließlich per Quellcode vorgenommen wird. [1]

1.2 Begriffe der Grafikprogrammierung

Es gibt eine Unzahl von Begriffen, welche in der Grafikprogrammierung verwendet werden und in den Ohren eines Leihens wie eine Fremdsprache klingen. Einige der gängigste Terme der Grafikprogrammierung werde ich nun hier mitsamt ihrer Bedeutung für den Anwender erklären.

1.2.1 Pixel

Das Wort Pixel ist eine Abkürzung für picture element (=Bildelement) und bezeichnet einen Bildpunkt. „Das Computerbild stellt sich auf dem Bildschirm als eine Matrix von Punkten (Pixels) dar. Diese sind für den Computer die kleinsten Einheiten, aus denen er ein Bild zusammensetzt. Pixels haben mindestens zwei mögliche Zustände (zum Beispiel schwarz und weiß) . Bei extremen Grafikanwendungen arbeitet man mit Pixels mit ca. 16 Millionen Farben

[1] Vgl.: Doberenz Walter / Kowalski Thomas(1997): Bordland Delphi 3 für Einsteiger und Fortgeschrittene, München/Wien :Hanser; S. 223

und einer Matrix von ca. tausend mal tausend Pixels. Je mehr Pixel auf dem Bildschirm dargestellt werden können, um so feiner und schärfer ist das dargestellte Bild."[2]

1.2.2 Vektor

Ein Vektor zeigt den Weg von einem Punkt zu einem anderen. Es gibt Vektoren im zwei und auch im drei dimensionalen Raum. Sie bestehen aus zwei bzw. drei Werten, die gemeinsam eine Richtung und eine Länge ergeben. Diese Werte geben den Weg zum anderen Punkt aufgegliedert als Weg in x-Richtung, Weg in y-Richtung und Weg in z-Richtung(wenn drei Dimensionen vorhanden sind) an. Weiters sind Vektoren nicht ortsgebunden, sondern können an jeden belieben Ausgangspunkt versetzt werden. Sie sind für die Mathematik der Grafikprogrammierung essentiell und jeder Programmierer sollte in der Lage sein mit Kreuzprodukten, Projektsformeln, Vektoradditionen und Skalaprodukten umzugehen

1.2.3 Auflösung

Die Auflösung einer Grafik wird in der Einheit dpi angegeben, was eine Abkürzung für dots per Inch (Punkte pro Inch) ist. Sie wird also durch die Anzahl der Pixels pro Inch bestimmt. Eine sehr übliche Auflösung sind 72 dpi, da dies das vom Internet übertragbare Maximum darstellt. Höhere Auflösungen empfehlen sich nur für Qualitätsdrucke, hochauflösende und entsprechend große Bildschirme und wenn eine starke Vergrößerung(und somit eine hohe Detailgenauigkeit) zu erwarten ist.

1.2.4 Farbmodelle

Es gibt verschiedenste Modelle wie man ein Farbsystem aufbauen kann. Hierbei ist das üblichste das RGB –System. Man nimmt die Farben Rot, Grün und Blau und erzeugt durch Mischen die restlichen Farben. Aus den 256 Abstufungen der Intensität bei jeder Farbe ergibt sich eine Farbtiefe von 24 Bit(=16777216 verschiedne Farben). Parallel dazu existiert auch ein CMY –System, welches im Grunde genommen gleich funktioniert, aber die Komplementärfarben zu RGB, nämlich Cyan(Zyan), Magenta und Yellow(Gelb), verwendet. Um eine Farbtiefe von 32 Bit(=4294967296 verschiedene Farben) zu erreichen existiert das CMYK –System, welches dem CMY –System ähnelt. Es wird hier aber ein weiterer Wert(auch mit 256 Möglichkeiten) verwendet um die Helligkeit der Farbe zu bestimmen. Ein Modell, um eine Farbtiefe von ca. 25 Bit(=23592960verschiedene Farben) zu beschreiben, ist das selten benützte HSB –System. Diese basiert auf einem Farbkreis in dem H einen

[2] Kaltenbach, Reetz, Woerrlein(1990): Das große Computerlexikon, Frankfurt am Main: Fischer; S. 246

Winkel(0-355°) vom Mittelpunkt aus angibt und S die Entfernung(0-255) zum Mittelpunkt definiert. Der Wert B bestimmt die Helligkeit der Farbe. Für geringe Farbtiefen wie 8 Bit(=256 Farbe) existiert das Graustufenmodell in dem es nur einen Wert für die Helligkeit gibt. Als konsequent existieren nur verschiedene Grautöne. Man nennt dieses System auch fälschlicher Weise Schwarz-Weiß. Denn beim echten Schwarz-Weiß System existieren ausschließlich die Farben Schwarz und Weiß und somit wird nur ein Bit für die Farbtiefe beansprucht. Es gibt jedoch auch ein buntes 256 Farben-Modell, welches mit einer vordefinierten Farbpalette funktioniert. Auf gleiche Art und Weise arbeitet auch das 16 Farben Modell. Es gibt sicherlich noch mehr Farbmodelle, doch ich gebe mich hier mit einem kurzen Überblick über die wichtigsten zufrieden.

1.2.5 2D Grafiken – 3D Grafiken

Die meisten Grafiken besitzen lediglich zwei Dimension. Drei dimensionale Grafiken sind im täglichem Gebrauch selten anzutreffen, außer vielleicht bei Computerspielen. 2D Bilder kann man mit jedem Zeichenprogramm ohne Schwierigkeiten erstellen, doch mit 3D Bilder wird alles um eine Dimension komplizierter. Weiters müssen 3D Grafiken in speziellen Programmen erstellt werden. Wenn man mit drei Dimensionen arbeitet, muss man bedenken, dass der Computerbildschirm auch nur eine zweidimensionale Oberfläche ist und man daher das Bild auf eine 2D-Grafik projektieren muss. Dabei sollte man stets die Position des Betrachters und seine Blickrichtung beachten. Eine weitere Schwierigkeit beim Anzeigen von drei dimensionalen Bildern ist die Tatsache, dass man für gewöhnlich nicht durch einen Gegenstand hindurchblickt, sondern es kann vorkommen, dass eine Wand ein Objekt verdeckt, welches dahinter steht. Dies ist bei Beleuchtungseffekten besonders wichtig. Darüber hinaus muss man berücksichtigen, dass alle 2D-Muster abhängig von der Perspektive verzerrt werden. Heute hat man das Glück, dies nicht mehr alles selbst berechnen zu müssen. Es wird nun von komplexen Grafikengines übernommen, welche die gewünschten Effekte mittels Raytracing und Polygongrafik erzeugen. Dennoch ist die Eingabe einer dritte Koordinate nicht immer leicht.

1.3 Geschichte: Immer neue Möglichkeiten

Nach der Entwicklung der ersten Computer wurden die ersten digitalen Bilder bei den Bell Labs & Universities im Jahre 1950 gebraucht. Es mussten einige Jahre vergehen bis 1959 mit DAC-1 (Design Augmented by Computers) das erste Computer Zeichnungs-System entwickelt wurde. Das von General Motors und IBM geschaffene System wurde aber erst

1964 bei der JointComputerConference präsentiert. In den Sechzigern setzte dann entgültig eine Entwicklung in diesem Sektor ein, die bis heute unaufhaltbar scheint. Bereits 1961 entstand das erste Videospiel „Spacewar„, welches von Steve Russel am Mit für den DEC PDP-1 entwickelt wurde. Vier Jahre später wurde an der Universität von Utah das Computer Science Department von Dr. David Evans gegründet. Im Jahr darauf schaffte es Ivan Sutherland am mit das Head Mounted Display zu entwickeln, worauf er ans das Computer Science Department geht. 1968 gründet Sutherland gemeinsam mit Evans E & S und diese Firma veröffentlich im folgendem Jahr das erste CAD Drahtgitter Zeichnungs- System, welches LDS-1 (Line Drawing System) genannt wird. John Warnok entwickelt im selben Jahr den Warnok Recirsove Subdivison, was den ersten Algorithmus für Hidden Surface Elimination darstellt. Alan Kay definiert auch 1969 die Metapher eines grafischen Benutzerinterfaces, welches zum Design des Apple Macintosh führt. 1972 wird PONG von Nilan Bushnell, dem späteren Gründer von Atari, entwickelt. Ein Jahr später wird zum erstem Mal ein Objekt vollständig im Computer erzeugt. Dieses große Osterei, welches von Roland Resch generiert wurde, steht noch immer in Vegreville, Alberta. Auch in 1973 entwickelt Frank Crow das Anti-Aliasing Verfahren zur Kantenglättung. Seit 1974 halten Computergrafiken auch verstärkten Einzug in die Filmindustrie, dann in diesem schafft Peter Foldes mit „Hunger„ den ersten voll animierten Film. 1976 tauchten die ersten Computergrafiken auch in Kinofilmen auf. In „Future World„ kann mein eine von Edwin Catmull generierte Drahtgitterhand sehen. 1977 wird in „Star Wars der Todesstern auch mit Hilfe von Computern simuliert. Im Jahr 1979 entsteht dann das Frame Buffer Konzept zum Speichern und Anzeigen von Single-Raster Images durch Jim Kajiya. In gleichem Jahr entsteht der Disneyfilm „The Black Hole„ der eine Simulation eines schwarzen Lochs von John Hughes beinhaltet. Mit dem Beginn der Achtziger wird von den Bell Labs und der Conrnell University das Raytracing entwickelt. Dieses Verfahren zur dreidimensionalen Darstellung ist auch heute noch von größter Bedeutung und stellt somit einen Meilenstein in der Entwicklung von Computergrafiken dar. Die Frucht einer Zusammenarbeit zwischen Lucasfilm Games Group und Atari ist das legendäre Spiel „Rescue on Fractalus„. Die Genesis Sequenz im Kinofilm „Star Trek II: Der Zorn des Kahn„ stellt die erste vollständig computer-generierte Sequenz da. 1982 entsteht der Kultfilm „Tron„ welcher der erste Kinofilm mit massivem Einsatz von Computergrafiken ist. Der erste Versuch von digitalem Compositing war „Where the Wild Things are„ in dem Charakter Animationen von Disney und Computergrafikhintergründe, sowie Rendering, Painting und Compositing von MAG I gemacht wurden. 1983 tauchte mit „Cube Quest„ ein frühes 3D Videospiel und auch „Die

Rückkehr der Jediritter„ setzte neue Maßstäbe in der Anwendung von Computergrafiken. In den nächsten Jahren folgten zahlreiche weitere Filme, die sich durch den Einsatz von Computergrafiken einen Namen machten. 1985 entwickelte ein Team unter Richard Taylor „LOOKER„, eine digitale Repräsentation eines vollständigen menschlichen Körpers. Mit dem 1986 entstandenen AMIGA Demo „The Juggler„ hielt Rayreacing und 3D Software Einzug an Heimcomputern. Im selben Jahr wurde „Luxo Jr.„ als erste Kurzfilm mit Computergrafiken für einen Oscar nominiert, doch erst 1988 gewann der Kurzfilm „Tin Tov„ als erster Film dieses Typus einen Oscar in der Kategorie: „Bester animierter Kurzfilm„. Ebenfalls 1988 entwickelte PIXAR gemeinsam mit Disney CAPS (Computer Animation Paint System). Mit „Indiana Jones und der letzte Kreuzzeug„ kam der erste Film mit vollständig digitalen Compositing in die Kinos. Nun wurden die Computergrafiken in Filmen und am PC immer besser und ausgefeilter. Schon 1992 tauchte in „Death becomes her„ ein photorealistischer Austausch von Haut und Körpern auf. Auch 1993 schienen in „Jurassic Park„ die 3D Dinosaurier täuschend echt. In „Forrest Gump„ wurden mit Hilfe von Computern in altes Filmmaterial neue Szenen eingeflochten. 1995 war es endlich so weit mit „Toy Story„ erreichte der erste komplett im Computer entstandene Film die Kinosäle. Ab nun wurden immer mehr Menschen durch computeranimierte Objekte ersetzt. In „Dragonheart„ gelang eine bahnbrechende computergrafische Charakter Animation und auch 1996 in „Twister„ eine überzeugende Simulation von Tornados. In „Titanic„ und „AntZ„ wurden Computer im großem Umfang für Crowd Simulation verwendet. Auch im einundzwanzigsten Jahrhundert setzt sich die rasche Entwicklung von Computergrafiken fort. Die Kinofilme zum Buch „Der Herr der Ringe„ setzten wieder neue Maßstäbe für Animation oder auch die Entwicklung der Quake 3 Engine bringt ein neues grafisches Niveau in die Welt der Computerspiele ein. Es bleibt nur abzuwarten wohin diese Entwicklung noch führt. [3]

1.4 Anwendungen von Computergrafiken

In der heutigen Zeit, wo Computer aus unserem täglichen Leben nicht mehr wegzudenken sind, finden wir Computergrafiken und Animation fast überall. Die Verzierungen einer Einladung und auch ganze Kinofilmen werden heute mit unterschiedlichsten Grafikprogrammen auf Computern erstellt. Zum Beispiel für die Filmtrilogie „Der Herr der Ringe„ wurde eigens ein Programm entwickelt, welches eine große Anzahl an computeranimierten Soldaten steuert, die man früher durch Statisten hätte ersetzen müssen. Dieses Programm wurde „Massive„ genannte und ermöglichte den einzelnen Figuren sogar

[3] Vgl. Online im Internet: http://www.3danimation.de (26. 1. 2003)

eigenständig erscheinende Handlungen. Weiters sahen die Figuren echten „Menschen„ zum verwechseln ähnlich und dies sogar in den Bewegungsabläufen. Auch die Kunst greift auf den Computer als Medium zurück um immer neue und noch faszinierender Kreationen zu erschaffen. Mit Grafikprogrammen wie Adobe Photoshop können sogar Leihen wahre Kunstwerke zu Stande bringen. Es werden vollständig neue Bilder und auch ganze Animationen am Computer erzeugt. Weiters eignen sich Grafikprogramme auch zum Manipulieren von Photos, da man heute in der Lage es ganz gewöhnliches Bild mit nur wenigen Kunstgriffen in eine Photomontage zu verwandeln. Darüber hinaus wird in der Filmindustrie nicht selten historisches Videomaterial neu bearbeitet, verändert und ergänzt um es so mit zusätzlichen Akteuren versehen in einen Film einzubauen wie es unter anderem in „Forrest Gump„ geschah. Hier konnte der Hautdarsteller auf diese Weise J. F. Kennedy „die Hand zu schütteln„.

1.5 Grafikspeicherung

Bilder können genauso wie Texte auch in Dateien gespeichert werden und es existiert eine Vielzahl von Formaten, die für die Aufbewahrung von Bildinformation verwendet werden. Im Grunde unterscheiden sich die einzelnen Formate in Qualität, Speicherbedarf und Ladeaufwand (Dekomprimierung und Grafikaufbau). Es ist einmal grundsätzlich gültig, dass Vektorgrafiken eine bessere Qualität als Pixelgrafiken besitzen und darüber hinaus sind Formate ohne Komprimierung den komprimierten in dieser Hinsicht überlegen. Weiters steigt der Speicherbedarf meist mit zunehmender Qualität, während der Ladeaufwand sich bei unkomprimierten Pixelgrafiken in Grenzen hält, doch bei komprimierten Bildern oder Vektorgrafiken drastisch zunimmt.

1.5.1 Das Format .avi

Dieses Format beinhaltet eine Bilderfolge zu der parallel auch eine Audiosequenz abgespielt wird, obwohl es auch Avis gibt, welche ausschließlich aus Bildern bestehen. Diese Dateien können komprimiert oder unkomprimiert, was häufiger der Fall ist, sein. Je nach Auflösung und Anzahl der Bilder kann die Größe von Avis zwischen ganzen 600 MB(1,5 Stunden Video und Audio mit Detailgenauigkeit von 600x800 Pixels) und 35 KB(1 Sekunde ohne Sound, bestehend aus 13 Einzelbilder mit weniger als je 10000 Pixels) schwanken. Dieses Format besitzt eine mittelmäßige Qualität und sehr kurzer Ladezeiten.

1.5.2 Das Format .bmp

Die Endung bmp steht für **B**it**m**a**p**. Dieser Typ ist eine nicht komprimierte Pixelgrafik mit einer Farbtiefe von 24 Bit. Die meisten Bitmaps verwenden, dass RGB-System, doch es existieren einigen wenige auch, welche in CMY-System gespeichert werden und somit die komplementären Farben im anderen System besitzen. Der Speicherbedarf eines Bitmaps ist leicht zu berechnen: Höhe(in Pixel) mal Breite(in Pixel) mal 24(wegen der Farbtiefe) durch 8 liefert das Ergebnis in Byte zu dem man anschließen noch 4 Byte für die Eckpunkte und 50 Byte für den Header der Datei addieren muss. Dieser sehr einfach aufgebaute Dateityp ist sehr weit verbreitet und lässt sich schnell laden.

1.5.3 Das Format .cgm

Meist enthalten CGM - Dateien entweder nur Vektoren oder nur Pixels, in Ausnahmefällen kann beides vorkommen. Obwohl die Information bei diesem Format normaler Weise als ASCII Code gespeichert werden, sind auch Speicherungen im binären Format möglich um kleiner Dateien zu erhalten. Es kann allgemein verwendet werden, ist jedoch besonders gut für den Einsatz in Vektorgrafik-Anwendungen konzipiert.[4]

1.5.4 Das Format .eps

„Das Format Encapsulated PostScript (EPS) wird zur Übertragung von PostScript-Grafiken zwischen Programmen benützt und von den meisten Grafik- und Seitenlayoutprogrammen unterstützt. Normaler Weise bestehen EPS - Dateien aus einzelnen Bildern oder Tabellen, die auf eine Seite platziert werden, jedoch können sie auch eine komplette Seite darstellen.„[5]

1.5.5 Das Format .gif

Die Endung gif bedeutet **G**raphics **I**nterchange **F**ormat und bezeichnet somit ein bereits sehr altes Grafikformat, welches bereits in den Achtzigern auf Commodore 64 verwendet wurde. Gif – Bilder besitzen lediglich eine Farbtiefe von 8 Bit, wodurch nur 256 Farben möglich sind. Heute werden zwei verschiedene Versionen von Gifs verwende: GIF87a und GIF89a. Durch ihre geringe Anzahl verschiedenen Farben sind Bilder dieses Dateityps gut komprimiert, während erstaunlicher Weise die Qualität nicht darunter leidet. Weiters sind Gifs Pixelgrafiken. Es gibt auch animierte Gifs, welche aus einer sich wiederholenden Sequenz von Bildern bestehen. Die Anwendungsgebiete dieses Formates sind hauptsächlich

[4] Vgl. Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)
[5] Vgl. Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)

im Internet zu finden, wo man es gern als kleiner Logos oder als Icons für Knöpfe verwendet, und auch zu Speicherung von Cartoons wird es benützt.

1.5.6 Das Format .ico

Dieser Dateityp wird gewöhnlich für Symbole(von exe - Dateien oder Verknüpfungen) verwendet. Üblich sind zwei verschiedene Größen: 16 mal 16 Pixel und 32 mal 32 Pixel. Dieses Formates ist auf 16 Farben beschränkt, doch diese sind für sein dürftiges Anwendungsgebiet ausreichend. Auf Grund der geringen Größe und des einfachen Aufbaus einer ico – Datei wird diese sehr schnell geladen.

1.5.7 Das Format .jpg

Jpg eigentlich Jpeg bedeutet **J**oint **P**hotographic **E**xperts **G**roup und ist damit der Name eines der wichtigsten und bekanntesten Speicherformate für Grafiken. Dieses auf Pixels und Vektoren(wegen Komprimierung) basierende Format verfügt über eine Farbtiefe von 24 Bit. Weiters ist es stark komprimiert, leider geht diese Prozess sehr auf Kosten der Qualität und manche Programme haben nicht die Fähigkeit ein Jpeg zu verwenden. Auf Grund seiner geringen Größe wird dieser Dateityp gern im Internet für Bilder verwendet.

1.5.8 Das Format .mac

„Dieses von Apple Computer Inc. entwickelte Bitmap-Dateiformat wird auf Macintosh-Plattformen unterstützt. Das MAC - Dateiformat wird in erster Linie von Macintosh-Grafikanwendungen genutzt, um schwarz/weiße Grafiken und Clipart zu speichern.„[6]

1.5.9 Das Format .mpg

Das Format .mpg ist ein Video und Audio unterstützender Dateityp mit hoher Qualität. Als Konsequenz sind diese Dateien verhältnismäßig groß, denn auch eine nur 30 Sekunden lange Sequenz braucht bereits ca. 1,4 Megabyte an Speicher. Trotzdem ist dieser Dateityp weit verbreitet um Videos zu sichern.

1.5.10 Das Format .pcx

„Das PCX - Dateiformat ist ein Bitmap, welches den Vorteil hat von plattformübergreifenden Anwendungen genützt zu werden. Das PCX - Dateiformat unterstützt beim Importieren und

[6] Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)

Exportieren eine maximale Bildgröße(für quadratische Bilder) von guten 4 Megapixel. Es kann ferner die RLE – Komprimierung verwenden.„[7]

1.5.11 Das Format .pdf

Dieser Dateityp, dessen Endung **P**ortable **D**ocument **F**ile bedeutet, wird von Adobe-Software unterstützt. Er wird zum Speichern von Dokumenten, welchen aus Bild und Text bestehen verwendet. Die Größe dieser Dateien ist oftmals enorm und die Ladezeiten sind auch nicht als gering zu bezeichnen, doch die Qualität kann man als erstklassig bezeichnen. Im Internet wird dieses Format gern verwendet.

1.5.12 Das Format .pict

„Das PCT- Dateiformat wurde von Apple Computer Inc. für den Macintosh entwickelt. Es ist das programmspezifische Dateiformat von QuickDraw, das sowohl Bitmaps als auch Vektoren enthalten kann. Seine Verwendung ist in Macintosh-Anwendungen, in denen Grafiken genutzt werden, weit verbreitet.„[8]

1.5.13 Das Format .png

Die Endung dieses Dateityps steht für **P**ortable **N**etwork **G**raphic. Png -Dateien, auch Pings genannt, wurde erschaffen um die Vorherrschaft des Gifs zu brechen.. Pings verfügen über eine effiziente LZW- Kompression und eine einmalige Transparenzmaske, welche ähnlich einem Alpha-Kanal arbeitet. Diese ermöglicht das Bild auch halbdurchsichtig erscheinen zu lassen. Weiters unterstützen Pings eine Farbtiefe von bis zu 24 Bit und sie werden gerne im Internet verwendet.

1.5.14 Das Format .tga

Dieses Dateiformat basiert auf Pixels und besitzt eine Farbtiefe von 24 Bit. Es wird hauptsächlich zur Speicherung von digitalisierten Farbfotos verwendet, welche entweder mit einer Digitalkamera aufgenommen wurden oder einfach nur eingescannt wurden. Tga – Dateien sind dank ihrer Plattformenabhängigkeit weit verbreitet und werden von zahlreichen Programmen unterstützt.

[7] Vgl. Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)
[8] Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)

1.5.15 Das Format .tif

Die Dateierweiterung .tif steht eigentlich für Tiff und dies bedeutet „Tag Image File Format„. Obwohl es auch unkomprimierte Tiff gibt, so verwenden die meisten Dateien dieses Typs eine LZW - Komprimierung. Diese ist Verlustfrei und garantiert die hohe Qualität dieses Formates. Doch dieses Verfahren hat auch seine Nachteile, da diese Komprimierung nicht sonderlich stark ist, bleiben die Tiff – Dateien doch noch relativ groß und die LZW – Komprimierung, welche nach Wiederholungen im Datenblock sucht, ist verhältnismäßig kompliziert und als logische Konsequenz benötigen Tiffs länger zum Laden bzw. zum Speichern. Üblicher Weise benützen Tiffs eine Farbtiefe von 24 Bit, doch es werden auch Farbtiefen bis zu 48 Bit unterstützt. Tiffs werden gerne zum Sichern von Bildern mit hoher Qualität, wie zum Beispiel Geschäftsausdrücke oder von Profis bearbeiteten Grafiken, verwendet.

1.5.16 Das Format .wmf

„Das WMF-Format (Windows Meta File) wurde von Microsoft entwickelt und ist in der Lage Vektoren und Bitmaps zu verwalten. Es wird nur von Windowskompatiblen Grafikanwendungen bis zu einer Farbtiefe von 24 Bit unterstützt.„[9]

[9] Vgl. Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)

2. Grafikprogrammierung in Delphi 3

Borland Delphi 3 stellt eine umfangreiche Palette an Tools für die zweidimensionale Grafikprogrammierung zur Verfügung und erleichtert auf diese Art und Weise dem Programmierer seine Gedanken in Form von Software umzusetzen. Häufig gibt es für eine Aufgabe mehrere Befehle, welche sich durch diverse Vor- und Nachteile unterscheiden. Der Versuch ein Grafikprogramm mit Delphi umzusetzen ist für mich, wenn es auch manchmal mühsam war, höchst interessant gewesen.

2.1 Möglichkeiten in Delphi 3

Graphische Darstellungen in Delphi 3 stützen sich zum Großteil auf das *TCanvas* – Objekt, doch da es schon von Grund auf über eine visuelle Benutzeroberfläche verfügt, gibt es auch noch ein paar andere Möglichkeiten eine Idee graphisch umzusetzen. Eine dieser Alternativen ist das Objekt *TShape*, welches immerhin 6 verschiedene Formen annehmen kann, eine andere wären die *ActivX* – Komponenten, welche sich zum Darstellen von Diagrammen hervorragend eigenen. Wenn man sich Mühe gibt, ist man wahrscheinlich auch in der Lage einige Effekte durch geschickte Anordnung von Objekten wie *Panels* oder *Buttons* zu erzielen, doch wie auch immer *TCanvas* bleibt, meiner Meinung nach, das vielseitigste Objekt in Delphi 3. Dreidimensionale Darstellungen in Delphi 3 sind leider um vieles Aufwändiger als zweidimensional, zumindest wenn man über *ActiveX* – Diagramme hinaus will. Da Delphi an sich nur sehr wenig anbietet, ist man gezwungen sich entweder selbst Methoden zu programmieren, welche aus Erfahrung eher langsam sind, oder man verwendet OpenGL oder DirectX. Delphiprogrammierer bevorzugen OpenGL, doch mit Delphi 3 lassen sich leider erst durch viel Aufwand damit brauchbare Effekte erzielen. Dies ist ein Grund warum ich mich hier nur auf die zweidimensionale Darstellung von Grafiken beschränke.

2.2 Die Canvas

Das englische Wort „Canvas„ bedeutet „Leinwand„ auf Deutsch und in Delphi bezeichnet dieser Begriff die Zeichenfläche, welche man auf unterschiedlichsten Objekten finden kann. Unter anderem verfügen *TBitmap*, *TPaintbox*, *TImage*, *TForm* und *TPrinter* über eine solche *Canvas*. „Statt einzelner Grafikfunktionen bietet eine *Canvas* eine Reihe von Methoden und Eigenschaften für die Grafikausgabe.„[10] Weiters besteht die *Canvas* aus Pixels, welche direkt oder auch im Rahmen eine Methode angesprochen und deren Farben entsprechend verändert

werden können. Darüber hinaus sollte man wissen, dass das Koordinatensystem auf einer *Canvas* nicht dem gängigen mathematischen entspricht, da bei diesem Delphiobjekt die Werte der x-Achse zwar nach rechts zunehmen, aber die Werte der y-Achse nehmen im Gegensatz zum mathematischen Koordinatensystem nach unten hin zu. Dieses Faktum erfordert, sollte man beabsichtigen einen Graphen einer Funktion darzustellen, eine manchmal komplizierte Umrechnung.

```
i:=(ex-ax)/anzpx;
rx:=(x*i+ax)

h:=(ey-ay)/anzpy;
v:=((ey+ay)/2)-ay)*2;
ry:=(((v)-y+ay)/h);
```

Die Variablen *rx* und *ry* geben den Punkt auf der Zeichenfläche an, wenn man die Variablen *x* und *y* im mathematischen Koordinatensystem darstellen möchte. Die Höhe der Zeichenfläche in Pixel gibt *anzpy* an während die Breite durch *anzpx* definiert ist. Der Wert von *ax* und *ex* sind die Grenzen des auf der Zeichenfläche dargestellten Intervalls in x-Richtung und *ay* und *ey* diejenigen in y-Richtung. Die restlichen Variablen *i*, *h* und *v* dienen nur als Hilfsvariablen um die Formeln zu vereinfachen. Diese Umrechung ist sehr allgemein gehalten und man kann sie vereinfachen, wenn es sich um ein konkreteres Beispiel handelt.
Einige der gängigen Methoden von *TCanvas* wären *arc*(Kreisbogen), *ellipse*(Ellipse), *pie*(Kreisstück), *polygon*(Vieleck) und *rectangle*(Rechteck), welche allesamt zur Darstellung von Formen nützlich sind. Es gibt auch noch weitere Methoden, die zur Grafikausgabe dienlich sind wie *lineto*(Linie ziehen), *draw*(Bitmap zeichnen), *textout*(Text ausgeben) und *copyrect*(Stück einer Zeichenfläche kopieren). Dies waren nur einige Beispiele für die Vielfalt, welche in *TCanvas* steckt, doch man ist in der Lage, wenn man eine starke Beeinträchtigung der Geschwindigkeit in Kauf nimmt, alle Methoden durch *canvas.pixels[x,y]* zu ersetzen. Mit ausreichendem mathematischen Wissen könnte man sämtliche Methoden dieses Objektes auf diese Weise ersetzen, da alle durch eine bestimmte Anordnung von Pixels definiert sind. Zwei weitere wichtige Eigenschaft von *TCanvas* sind *Pen* und *Brush*. *TPen* selbst verfügt über die Eigenschaften *Color*, *Mode*, *Style* und *Width*. Doch bevor ich auf diese näher eingehe, würde ich gerne noch das Anwendungsgebiet von

[10] Doberenz Walter / Kowalski Thomas(1997): Bordland Delphi 3 für Profis, München/Wien :Hanser, Seite75

TPen erwähnen. Ein *Pen*(Pinsel) zieht die Umrandung von Figuren und Linien in der gewählten Einstellung. Seine Farbe wird durch die Eigenschaft *Color* bestimmt und seine Dicke durch *Width*. *Pen.Style* gibt die Art der Linienziehung an, welche zum Beispiel durchgezogen, punktiert oder liniert sein kann. Hingegen bestimmt die Eigenschaft *Mode* die Art der Beeinflussung von Hintergrundfarbe und Stiftfarbe, also in wie weit sich die beiden Farben kombinieren. Im Gegensatz zu *TPen* füllt *TBrush* den Innenraum einer Figur aus und verfügt nur über die Methoden *Color* und *Style*. *Color* bestimmt wieder die Farbe, während *Style* das Füllmuster, welches beispielsweise gestreift, kariert oder vollständig sein kann, angibt.
Für die Textausgabe verfügt *TCanvas* über den Befehl *textout*, welcher in der Lage ist einen String an angegebene Koordinaten zu schreiben. Um dem Text eine Gestallt zu geben besitzt die *Canvas* als logische Konsequenz auch die Eigenschaft *Font*, wodurch Schriftart und Schriftfarbe festgelegt werden können. Dank dieser Eigenschaft ist man in der Lage über *TCanvas* Text und Zeichnung zu kombinieren.
Ich hoffe diese Beispiele haben gezeigt wie vielseitig und in Folge bedeutsam das *TCanvas* – Objekt für Delphi 3 ist. Ich glaube man kann behaupten, dass ohne diese Klasse das Darstellen von Grafiken in Delphi fast unmöglich wäre.

2.3 Grafikobjekte

In Delphi 3 werden dem Programmierer mehrere Objekte zur Arbeit mit Grafiken zu Verfügung gestellt, welche sich durch unterschiedliche Eigenschaften auszeichnen. Die meisten dieser Objekte stützen sich auf die Klasse *TCanvas* und sind dadurch miteinander kompatibel.

2.3.1 TBitmap

Ich nehme an, dass man ein *Bitmap* am leichtesten als eine Matrix von Bildpunkten(Pixels) beschreiben kann, wobei jeder Pixel einen Farbwert als Eigenschaft besitzt. In Delphi kann man *Bitmaps* selbst nicht sehen, es sei denn man zeichnet sie in ein anderes Objekt, welches in der Lage ist Grafiken wiederzugeben. Wenn man beabsichtigt mit diesem Objekt zu Arbeiten so besteht der Bedarf zuerst einmal eine Variable von Typ *TBitmap* zu deklarieren. Bevor man es jedoch verwenden kann, muss man es noch erzeugen mittels *Bild:=TBitmap.create*, wobei hier *Bild* der Name des *Bitmaps* ist. Wenn man es nicht mehr benötigt so kann man es wieder freigeben über *Bild.free*. Dies ist empfehlenswert, wenn man mit vielen *Bitmaps* arbeitet, da es in Delphi vorkommen kann, dass wenn man über 500 *Bitmaps* verwendet, „einige„ Fehler auftreten, welche das Programm zum Absturz bringen.

Dieses Objekt verfügt wie so viele auch über eine *Canvas*, wodurch ihm die damit verbundenen Eigenschaften zur Verfügung stehen. Normaler Weise verwendet man Bitmaps um entweder Bilder virtuell zu bearbeiten oder aber um Grafiken von der Festplatte zu laden oder dort abzuspeichern. Dafür verfügt das Objekt über Eigenschaften *loadfromfile*(laden) und *savetofile*(speichern). Eine weitere Fähigkeit des *TBitmap*-Objektes ist die Möglichkeit, dass man eine Farbe transparent, also durchsichtig, macht. Diese Eigenschaft kann man sich ganz leicht zu Nutze machen und damit ist man in der Lage eine beachtliche Anzahl von Effekten zu erzielen.

```
Bild.Transparent := True;
Bild.TransParentColor := clwhite;
```

In diesem Beispiel ist *Bild* ein Objekt vom Typ *TBitmap*. Die Einstellung der Farbe, welche anschließend unsichtbar sein soll, erfolgt über *TransParentColor* und ich habe in diesem Fall die Farbe weiß(*clwhite*) gewählt, doch man ist ohne weiteres dazu fähig jede beliebige Farbe zu nehmen.
Ich persönlich habe recht gute Erfahrungen mit *TBitmap* gemacht und halte es für eines der handlichsten Grafikobjekte in Delphi.

2.3.2 TPaintbox

Wie bereits erwähnt können Bitmaps sich nicht selbst sichtbar machen, da sie nur für das Arbeiten im Hintergrund gedacht sind. Doch wenn man einen Nutzen von seiner Arbeit mit Grafiken zeihen will, so sollte man sie irgendwo darstellen können und ein Weg dazu ist die Verwendung des Objektes *TPaintbox*. Da eine Paintbox über ein eigenes *Canvas* – Objekt verfügt ist man in der Lage auf ihr eine breite Vielfalt an Grafiken wiederzugeben. Über die Methode *canvas.draw* kann man auf einer *Paintbox* auch *Bitmaps* darstellen. Der große Vorteil der Paintbox ist ihre Schnelligkeit, doch die hat auch ihren Preis. Sobald sich ein Fenster vor das *TPaintbox* – Objekt drängt, gehen die in der *Canvas* befindlich Informationen, welche durch das Fenster abgedeckt werden, verloren und können nicht wiedergeholt werden. Durch einen Trick, welcher dem Objekt kaum was an seiner Geschwindigkeit nimmt, war ich in der Lage diesen Nachteil zu umgehen. Ich verwende einfach zusätzlich zur *Paintbox* ein *Bitmap* als Puffer. Alles was auf die *Paintbox* gezeichnet wird, wird eigentlich in das *Bitmap* gezeichnet, welches ich dann mittels *canvas.draw* auf das *TPaintbox* – Objekt übertrage. Diese Technik verhindert den Verlust von Bildinformation, da wenn sie auf der *Paintbox*

verloren geht noch immer im *Bitmap* gespeichert ist und erneut gezeichnet werden kann. Normaler Weise erzeugt man eine *Paintbox* nicht wie ein *Bitmap*, obwohl dieser Weg auch möglich wäre. Man zieht sie üblicher Weise schon in der Entwurfsansicht in Delphi auf dem gewünschten Formular auf. Man kann eine *Paintbox* auch unsichtbar machen in dem man *visible* auf *false* setzt. Dadurch ist man in der Lage die Grafik, welche auf dem Objekt dargestellt wird, verschwinden zu lassen. Meiner Meinung nach ist das Objekt *TPaintbox* durchaus nützlich und ich verwende es gern, trotz seiner Nachteile.

2.3.3 TTurtle

Das Objekt *TTurtle* und die dazugehörigen Units entstammen einem Delphi – Package, welches ein beherzter Delphiprogrammierer für andere Nutzer geschaffen hat. An sich dürfte das *Turtle*-Konzept einem Programmierer nicht fremd sein, da es der alten Logo-Programmierung entstammt. Anstelle mit einem Pinsel Lienen von Punkt A zu Punkt B zu zeihen, ist es bei *Turtle*-Grafiken üblich den Cursor von einem Punkt aus in einem gegebenen Winkel eine bestimmte Länge entlang zu bewegen und somit ein Bild zu konstruieren. Obwohl in der mir zur Verfügung stehenden *Turtle* auch das konventionelle Linienziehen möglich ist, verwende ich sie nur wenn ich ihre speziellen Eigenschaften unbedingt brauche.
Von dem Cursor der *Turtle* aus gesehen kann man sich in alle Richtungen bewegen, also 360°. Sollte sich der Cursor denn Winkel Null Grad als Richtung zugewiesen bekommen so würde er sich direkt nach rechts bewegen. Da 90° als Zielwinkel für den Cursor nach oben zeigen, verläuft das Winkelsystem gegen den Uhrzeigersinn. Die Länge der zu ziehenden Linie wird in Pixels angegeben. Nun werde ich versuchen einige Methoden des Objektes *TTurtle* zu erklären.
TurnTo : Mittels diesem Befehl kann man die Ausrichtung des Cursors der *Turtle* festlegen. Als Parameter wird ein Winkel in Altgrad übergeben.
Turn : Dieser Methoden kann man positive oder negative Werte als Parameter mitgeben. Dieser Wert wird zum Winkel der aktuellen Ausrichtung addiert, wodurch sich der Cursor in eine andere Richtung bewegen würde.
Draw : Mit dieser Anweisung zieht der Cursor eine Linie, deren Länge als Parameter dieser Methode gegeben ist, und deren Richtung durch die Ausrichtung des Cursors bestimmt ist. Den Anfangspunkt bildet die ursprünglich Cursorposition und der Cursor wird dann auf, dem durch die gegebenen Werte bestimmten, Endpunkt der Linie gesetzt. Sollte der Wert des Parameters negativ sein, so wird die Linie in die entgegengesetzte Richtung, welche durch die Ausrichtung des Cursors definiert ist, gezogen.

MoveTo : Diese Methode setzt den Cursor auf die in den Parametern bestimmte Position ohne eine Linie zu ziehen. Die Ausrichtung bleibt natürlich erhalten.
DrawTo : Dieser Befehl verhält sich wie *MoveTo* nur dass hier eine Linie von der ursprünglich zur neuen Cursorposition gezogen.
Ich hoffe die Erläuterungen zu diesen Befehlen haben einen Einblick in die Funktionsweise des *TTurtle*-Objekts gegeben. Dieses Objekt verfügt auch über eine *Canvas,* womit es alle damit verbunden Methoden und Eigenschaften benützen kann. Weiters möchte ich darauf hinweise, dass die Farbe der Linien, welche durch den Cursor gezogen werden können, durch die Eigenschaft *canvas.pen.color* bestimmt ist und somit verändert werden kann.
Sinnvolle Anwendungsgebiete dieses Objektes wären die Darstellung von Rekursion oder das Zeichnen von Gittergrafiken, welche mit den entsprechenden mathematischen Kenntnissen auch dreidimensional gestallten werden können. Ein bedeutender Nachteil von diesem *Turtle*-Objekt ist mir durch Experimentieren bekannt geworden. Sollten die Werte, welche als Parameter den Methoden übergeben werden zu nahe bei Null liegen, so bringt dies das gesamte Programm, welches die *Turtle* verwendet, um Absturz. Ich habe versucht dieses Problem irgendwie zu umgehen, doch wie mir scheint ist dies unmöglich.

2.3.4 TImage

Das Objekt *TImange* ist von der Erzeugung und Benutzung her einer *Paintbox* sehr ähnlich, doch unterscheiden sich beide Objekte außerordentlich in den Eigenschaften und den Verfügung stehenden Möglichkeiten. Wie bereits erwähnt ist der entscheidende Nachteil einer *Paintbox*, dass Bildinformation verloren gehen kann. Ein Image behandelt das in seiner *Canvas* liegende Bild ein wenig anders und verhindert somit den Verlust von Information. Leider hat dieser Sicherheit auch ihren Preis, da das Objekt *TImage* viel langsamer mit Grafiken arbeitet und dies führt oft zu störendem Flimmern. Doch gegen dieses Manko stehen noch viele weitere Vorteile. Ein *Image* ist an der Lage Bilder selbstständig zu laden und zu speichern. In dieser Eigenschaft übertrifft es sogar das Objekt *TBitmap*, da eine *Image* auch in der Lage Bilder vom Dateityp .jpg zu laden, zu verwalten und zu speichern. Zu dem existieren noch weitere Vorteile dieses Objektes, wie zum Beispiel, dass das *Image* fähig ist, wenn man die Eigenschaft *AutoSize* auf *true* setzt, sich selbst der Größe der geladenen Bilder anzupassen. Darüber hinaus ist dies auch umgekehrt möglich, wenn man *stretch* auf *true* setzt, passen sich alle Bilder der Größe des *Images* an.
Es ist nur logisch, dass man sich die Vorteile dieses Objektes nützt, während man versucht seine Nachteile durch Tricks zu umgehen. Ich bin der Meinung man erreicht die besten

Ergebnisse, wenn man sich die positiven Eigenschaften aller Grafikobjekte sich zu Nutze macht. Daher bin ich der Ansicht man sollte ein *Image* nur verwenden, wenn man ein JPG-Bild ladet oder speichert, doch zur Bearbeitung sollte man es in ein *Bitmap* kopieren. Man kann ein Objekt vom Typ *TImage* auch für das dauerhafte Anzeigen eines Bildes verwenden, doch wenn sich das Bild oft ändert so empfiehlt sich ein *Paintbox*, welche sich durch seine Schnelligkeit auszeichnet.
Ich persönlich bin kein Freund des Images, aber ich sehe auch seine Vorteile und verwende es gern zur Unterstützung meiner Programme und lasse es dabei eher im Hintergrund arbeiten.

2.3.5 TAnimate

TAnimate wird zum Abspielen von Avi - Dateien ohne Sound verwendet. Man kann mit ihm gezielt einzelne Bilder ansteuern oder auch die ganze Sequenz abspielen. Ich möchte darauf hinweise, dass eine Avi – Datei aus mehreren Bildern und häufig auch aus einer Audiosequenz besteht. Durch die Eigenschaft *AutoSize* sollte sich die Komponente an die Größe der Bilder der Avi – Datei anpassen, doch leider verweigert sie es, wenn die eine derartige Datei nur indirekt(von einem *Mediaplayer* aus) in das *Animate* geladen wird. Wenn man nicht auf die akustischen Effekte eines Avis verzichten will, so empfiehlt sich ein Objekt vom Typ *TMediaplayer*. Wie schon erwähnt, funktioniert die Wiedergabe einer Avi - Datei in einem *Animate* nicht fehlerfrei, wenn sie durch einen *Mediaplayer* gesteuert wird. Dies brachte mich so weit, dass ich auf das Objekt *TAnimate* verzichtete. Doch hat es auch Vorteile, wenn man zum Beispiel beabsichtigt Bilder einer Avi - Datei einzeln zu betrachten. Eine weitere sinnvolle Anwendung dieses Grafikobjektes wäre das Anzeigen von *CommonAVIs*. „*CommonAVI* wird verwendet, wenn einer der Standard-Clips abgespielt werden soll, die aus der Windows-Benutzeroberfläche bekannt sind. Meist handelt es sich um Illustrationen üblicher Dateiverwaltungsoperationen. *CommonAVI* hat den Wert *aviNone*, wenn der abzuspielende Clip in anderer Form vorgegeben wurde„[11] Trotz mancher Schwächen ist das Objekt TAnimate gelegentlich zur Verbesserung des Eindrucks mancher Programme gut geeignet oder auch zur Betrachtung tonloser Avi - Dateiein.

2.3.6 TPrinter

Das Objekt *TPrinter* steuert den Drucker, doch es verfügt auch über eine *Canvas*, welche für das Drucken von Grafiken nötig ist, und daher hielt ich es für wichtig *TPrinter* hier zu

[11] Delphi-Hilfe, TAnimate.CommonAVI

erwähnen. Um einen Drucker verwenden zu können besteht Bedarf die Unit *Printers* einzubinden. Ein Bild in Delphi zu drucken ist ein relativ einfacher Vorgang.

```
Printer.create;
Printer.BeginDoc;
Printer.Canvas.Draw(0,0,bild);
Printer.EndDoc;
```

Zuerst muss man einmal einen Drucker erzeugen. Die Anweisungen *BeginDoc* und *EndDoc* kennzeichnen das den Anfang und das Ende des Druckauftrages. Über *canvas.draw* kann man ganz einfach eine Grafik den Drucker schicken, welcher er dann ausdruckt. Der Parameter *bild* ist hier ein Objekt vom Typ *TBitmap*. Der Drucker ist in der Lage alles auszudrucken, was sich auf seiner *Canvas* befindet, doch es ist ratsam die zu druckende Grafik zuerst auf einem *Bitmap* vorzubereiten und es dann mit *canvas.draw* dem Drucker zu übergeben.

3. Ausgewählte Beispiele aus meinem Programm

Um meine Arbeit über Grafikprogrammierung abzurunden, habe ich ein sehr umfangreiches Programm entwickelt, dessen Methoden nun als Beispiele dienen, um meine Ausführungen zu diesem Thema zu veranschaulichen. Es würde dem Rahmen sprengen, wenn ich alle Möglichkeiten (und ihre Funktionsweise) die mein Programm dem Benutzer eröffnet, eingehenden zu beschreiben. Daher habe ich beschlossen mich hier nur auf einige ausgewählte Beispiele zu beschränken, welche entweder mehrmals in der Arbeit wiederkehrten oder bei denen ich ungewöhnliche Probleme zu lösen hatte. Im folgendem Abschnitt werde ich nun diverse Eigenschaften meines Programms anhand von Quellcode und gut verständlichen Erläuterungen in ihrer Funktion und Auswirkung erklären.

3.1 Einfache Zeichenwerkzeuge

Jedes computerunterstützte Malprogramm besitzt eine Palette gängiger Hilfsmittel, die man immer wieder braucht um ein Bild zu editieren. Ich werde hier nun ein paar der üblichen Stifte und Formen vorstellen und natürlich auch die Techniken, die hinter ihrer Funktion stecken.

3.1.1 Der Pinsel

Um Freihand beliebige Objekte zu zeichnen, ist als Basiswerkzeug ein Pinsel von Nöten. Dieser Pinsel ist in der Lage jede gewünschte Dicke und Farbe anzunehmen, die sich der Anwender einfallen lässt. Das Verwendungsgebiet dieses Hilfsmittels ist sehr vielfältig. Vom wilden Malen über ungefähres Skizzieren bis hin zu feinsten Korrekturen am Bild liegt alles im Bereich des Möglichen. Der Pinsel in meinem Programm arbeitet mittels zwei Prozeduren. Die erste, welche ich *pins1* genannt habe, wird durch das Klicken der Maus ausgelöst und die zweite, welche den Namen *pins2* trägt, durch die Bewegung der Maus.

```
procedure Tform1.pins1(x,y:integer);
begin
backspeicher;
 puffer.canvas.pen.color:=vfarb;
 puffer.canvas.brush.color:=vfarb;
 puffer.Canvas.pen.width:=1;
puffer.canvas.Ellipse(x-trunc(dick/2)+varx,y-
trunc(dick/2)+vary,x+trunc(dick/2)+varx,y+trunc(dick / 2)+vary);
```

```
    puffer.Canvas.pen.width:=dick;
   puffer.canvas.moveto(x+varx,y+vary);
    end;
```

Die Parameter *x* und *y*, die der Prozedur mitgegeben sind, enthalten die Koordinaten des Mauszeigers und werden in der Prozedur *PaintBox1MouseDown* zugewiesen. *Backspeicher* ist eine weitere äußerst bedeutsame Prozedur, da sie ein Widerrufen des letzten Befehls ermöglicht und somit in der Lage ist nach einem Zeichenfehler, das alte Bild wieder herzustellen. Anschließend wird die Farbe des Stiftes festgelegt, wobei die Variable *vfarb*, die aktuelle vom Benutzer zuvor ausgewählte Vordergrundfarbe ist. Im nächsten Schritt wird ein Punkt, mit dem Durchmesser des Pinsels erzeugt. Dies ist nötig, weil die in *pins2* verwendete Delphi-Funktion *lineto* keine Linie zeichnet, wenn ihr Ausgangspunkt und Endpunkt identisch sind. Daher bestand die Notwendigkeit sofort diesen Punkt zu Zeichnen, da ich den Anwender die Option lassen will, mittels des Pinsels auch Bilder mit kleinen(oder auch größeren) Punkten zu versehen. Der nächste Schritt ist das Bestimmen der vom Benutzer beabsichtigen Dicke der Linie. Dazu verwende ich die Variable *dick*, welche mit Hilfe eines *Spinedits*, entsprechend seinen Wünschen, verändern kann. Die Prozedur schließt mir dem Setzen des Zeichencursors auf die Mausposition. Es sind nun, in dieser Prozedur, schon mehrmals die Variablen von Typ Integer *varx* und *vary* aufgetaucht. Sie sind von essentieller Bedeutung für das ganze Programm, da sie den Unterschied zwischen dem Koordinatensystem des Bildes und der wirklichen Zeichenfläche angeben. Sollte ein Bild größer sein als die vordefinierte Zeichenfläche so kann man sich darin via *Scrollbars* bewegen. Dadurch verschiebt sich aber das Koordinatensystem des Bildes relativ zu dem der Zeichenfläche. Um diese Abweichung beim Malen auszugleichen wird der Unterschied in die Variablen *varx* und *vary* gespeichert um ein fehlerfreies Zeichnen auch bei größeren Bildern gewährleisten zu können.

```
procedure Tform1.pins2(x,y:integer);
   begin
   if mal=true
   then
   puffer.canvas.lineto(x+varx,y+vary);
   end;
```

Die Prozedur *pins2* benötig wie auch *pins1* die Parameter *x* und *y* vom Typ Integer, nur dass sie diesmal *PaintBox1MouseMove* entstammen. Die Variable *mal* wurde in der Prozedur *PaintBox1MouseDown* auf *true* gesetzt und in der beim Auslassen der Maustaste aktiv werdenden Prozedur *PaintBox1MouseUp* wird ihr wieder der Wert *false* zugewiesen. Auf diese Weise zeichnet der Pinsel nur wenn die Maus bewegt wird und wenn zur gleichen Zeit die Maustaste gedrückt ist. Da alle Voreinstellungen bereits in *pins1* getroffen wurden reicht es hier aus dem Programm zu befehlen eine Linie von der alten Cursorposition zur jetzigen neuen zu ziehen. *Lineto* zieht eine Linie mit den in *Pen* definierten Eigenschaften von der Grafikcursorposition zu den als Parameter angegebenen Koordinaten. Danach setzt sie gleich den Grafikcursor auf die aktuelle Position. Auf diese durchaus simple Weise lässt sich ein hervorragend funktionierender Pinsel programmieren.

3.1.2 Ein Rechteck

Ich nehme das Reckeck als einfaches Paradeexemplar für eine beliebige Figur. In meinem Programm Multidraw kann man unter anderem auch Quadrate, Dreiecke, Rundecke, Kreise Ellipsen und Polygone aufziehen. Da jedoch das Prinzip bei allen anderen das gleiche ist, nehme ich mir das Rechteck als leicht erläuterbares Beispiel heraus und versuchen damit die Programmiertechniken, welche zum Darstellen von verschiedensten Figuren nötig sind, zu erklären. Man kann ein parallel zu den Achsen des Koordinatensystems positioniertes Rechteck eindeutig mittels zweier Punkte beschreiben. Es reicht also aus die Lokalisationen der linken oberen Ecke und der rechten unteren Ecke anzugeben. Auf Grundlage des Umstandes, dass sich alle Figuren ähnlich programmieren lassen, habe ich eine Prozedur *ansatz* entwickelt, welche bereits am Anfang des Aufziehprozesses alle erforderlichen Grundeinstellungen für jede kompatible Figur trifft.

```
procedure Tform1.ansatz(x,y:integer);
  begin
 backspeicher;
   lx:=x;
   ly:=y;
   puffer.canvas.pen.color:=vfarb;
   puffer.canvas.Pen.width:=dick;
   puffer.canvas.brush.color:=hfarb;
  end;
```

Die Prozedur *ansatz* wird in *PaintBox1MouseDown* aufgerufen und weist den *Pen* und den *Brush* die vom Benutzer gewählten Eigenschaften zu. Die hier vorkommenden Variablen *lx* und *ly* dienen als globaler Speicher für die Koordinaten des, mit dem Drücken der Maustaste ausgewählten, Eckpunkts. Noch einmal zurück zu den Stiften. *Dick* und *vfarb*, die den Pen zugewiesen werden, wurden bereits besprochen, doch nicht Variable *hfarb*. Sie kann ebenfalls vom Benutzer gewählt werden und bestimmt, im Gegensatz zu *vfarb*, welche die Farbe der Umrandung des Rechtecks festlegt, die Füllfarbe der Figur. Man kann auch, wenn man es beabsichtigt, die Füll- und Umrandungsart seinen Wünschen anpassen. Über die Knöpfe HG-Stil bzw. VG-Stil ist man in der Lage das Rechteck mit z.B. einem Karomuster zu füllen und die Begrenzungslinie zu punktieren. Die nächste Prozedur, welcher es bei der Erzeugung eines Rechteckes bedarf, erhielt von mir die Bezeichnung *recht1*.

```
procedure Tform1.recht1(x,y:integer);
    begin
     if mal=true
  then
  begin
  puffer.canvas.draw(0,0,bild);
  puffer.Canvas.rectangle(lx+varx,ly+vary,x+varx,y+vary);
  end;
    end;
```

Auch *recht1*, welche unter *PaintBox1MouseMove* gestartet wird, enthält als Parameter die Koordinaten des Mauszeigers und weiters dient die Variable *mal* wieder dem gleichen Zweck wie zuvor schon beim Pinsel. Die Prozedur *Backspeicher* hat noch einen weiteren Sinn nämlich beim Aufziehen einer Figur das alte Bild parat zu halten um die Zeichenfläche, bis man das Objekt richtig platziert hat, immer wieder von der „Vorschau„ zu säubern. Dies geschieht indem das ursprüngliche Bild immer wieder vor dem, in diesem Fall, Rechteck auf den *puffer* gezeichnet wird. Zu guter Letzt wird dann immer wieder beim Bewegen der Maus das Rechteck mit den in *ansatz* vordefinierten Eigenschaften gemalt. Abschließend muss das Rechteck noch fixiert werden, wenn man die Maustaste auslässt und dies geschieht in der folgenden Prozedur *recht2*.

```
procedure Tform1.recht2(x,y:integer);
   begin
```

```
 puffer.canvas.draw(0,0,bild);
puffer.Canvas.rectangle(lx+varx,ly+vary,x+varx,y+vary);
  end;
```

Unter *PaintBox1MouseUp* wird diese Prozedur aufgerufen und ihr die Parameter *x* und *y* zugewiesen. Die zuletzt gezeigte Vorschau wird übermalt und schließlich das endgültige Rechteck mit dem Eckpunkten, deren Koordinaten *lx* und *ly*, bzw. *x* und *y* sind, dargestellt. Abschließend möchte ich nochmals darauf hinweisen, dass *rectangle* nicht der einzige Befehl ist um ein Rechteck zu zeichnen. Eine andere Option wäre *fillrect*, welche jedoch nur den Innenraum dieses regelmäßigen Vierecks zeichnet und die Umrandung ignoriert. Der Befehl *framerect* bewirkt das Gegenteil, da es lediglich die Begrenzungslinien der Figur wiedergibt. Für meinen Befehl Rahmen, welche nur die Umrandung eines Rechtecks zeichnet, verwende ich nicht den Befehl *framerect*, da dieser einen, für ein benutzerfreundliches Programm, entscheidenden Nachteil besitzt: Die Koordinaten des ersten Eckpunktes müssen kleiner sein als die des zweiten. Aus diesem Grund würde der Befehl, hätte ich ihn benützt, den Anwender nur befähigt ein Rechteck von links oben nach rechts unten aufzuziehen. Dieses Manko habe ich übergangen in dem ich den Rechteckrahmen mit mehreren *lineto* gezeichnet habe. Offensichtlich fordert in manchen Bereichen Delphi noch die Kreativität des Programmierers.

3.1.3 Das Tortenstück

Es gibt viele Objekte, die man mit wenig Information, eindeutig darstellen kann, doch einige erfordern mehr als zwei Punkte. Als Folge dieses Faktums reicht es nicht mehr aus die Figur einfach aufzuziehen, sonder man muss mittels Mausklicke, die für die Erzeugung der Form nötigen Koordinaten setzen. Ein Beispiel für einen solchen Prozess ist das Tortenstück, welche mathematisch korrekt Kreissegment heißen müsste. Es kann im Grunde als eine Ellipse beschrieben werden, von der zwei Geraden, beide vom Mittelpunkt der Figur ausgehend, ein Stück herausschneiden. Um im Zeichenprogramm, den Benutzer ein Tortenstück nach seinem Belieben erschaffen zu lassen, bedarf es dreier Prozeduren.

```
procedure Tform1.tort1(x,y:integer);
begin
case bclick of
0:begin
backspeicher;
bclick:=1;
```

```
lx:=x;
ly:=y;
    puffer.canvas.pen.color:=vfarb;
    puffer.canvas.Pen.width:=dick;
    puffer.canvas.brush.color:=hfarb;
end;
2:begin
bx:=x;
by:=y;
bclick:=3;
end;
3:begin
puffer.canvas.draw(0,0,bild);
puffer.canvas.pie(lx+varx,ly+vary,llx+varx,lly+vary,bx+varx,by+vary,x+varx,y+vary);
bclick:=0;
end;

end;
end;
```

Das Erste, was man in der Prozedur *tort1* erblickt ist nach der Deklaration die *case of* – Abfrage. Die daran vorkommende Variable *bclick* wurde bereits beim Auswählen des Zeicheninstrumentes auf Null gesetzt. Der Bedarf von *bclick* besteht daran, die Anzahl der *PaintBox1MouseDown*(*tort1* wird natürlich in dieser Prozedur aufgerufen) und ähnlicher Ereignisse zu zählen, die für das Zeichnen eines Tortenstückes gebraucht werden, zu zählen. Im Fall, dass der Wert von *bclick* Null ist, werden erstmals die Grundeinstellungen getroffen und die Variable auf Eins weitergeschalten. Auch wenn ich diese Prozedur noch nicht fertig erklärt habe, ist es unausweichlich, dass ich bereits zur nächsten übergehe, da dies der Reihenfolge entspricht in der die Prozeduren im Programm aufgerufen werden. Ich werde jedoch später wieder auf *tort1* zu sprechen kommen.

```
procedure Tform1.tort2(x,y:integer);
begin
case bclick of
1:begin
puffer.canvas.draw(0,0,bild);
puffer.canvas.ellipse(lx+varx,ly+vary,x+varx,y+vary);
```

```
end;
2:begin
puffer.canvas.draw(0,0,bild);
puffer.canvas.ellipse(lx+varx,ly+vary,llx+varx,lly+vary);
puffer.canvas.moveto(round((lx+llx)/2)+varx,round((ly+lly)/2)+vary);
puffer.canvas.lineto(x+varx,y+vary);
end;
3:begin
puffer.canvas.draw(0,0,bild);
puffer.canvas.pie(lx+varx,ly+vary,llx+varx,lly+vary,bx+varx,by+vary,x+varx,y+vary);
end;
end;
end;
```

Auch in *tort2* steht die *bclick* in Verwendung und erfüllt die gleiche Funktion wie in *tort1*, da beide Prozeduren eng zusammen arbeiten. Zuvor wurde *bclick* auf Eins gesetzt und nun, da *tort2 PaintBox1MouseMove* untersteht, wird sie wieder benützt. Im Punkt 1 der *Case of* – Abfrage wird die Ellipse, aus welcher später das Tortenstück herausgeschnitten wird, vorgezeigt. In der nächsten Prozedur, welche auf *PaintBox1MouseUp* gestartet wird, setzt dieses Beispiel fort.

```
procedure Tform1.tort3(x,y:integer);
begin
if bclick=1
then
begin
puffer.canvas.draw(0,0,bild);
puffer.canvas.ellipse(lx+varx,ly+vary,x+varx,y+vary);
llx:=x;
lly:=y;
bclick:=2;
end;
end;
```

tort3 ist nur in Gebrauch um die Ellipse vorläufig hinzuzeichnen und *bclick* den Wert 2 zu zuweisen. Anschließend kann mit dem Erstellen der beiden Geraden begonnen werden und ich kehre wieder zu *tort2* zurück. Wenn *bclick* auf 2 gesetzt ist so wird dort zusätzlich zur

Ellipse noch eine Gerade, deren Endpunkte der Mittelpunkt der Ellipse und die aktuelle Position des Mauszeigers sind, angezeigt. Beim Drücken der Maustaste speichert das Programm in der Prozedur *tort1* die Koordinaten des Mauszeigers, welche mit dem Endpunkt der Geraden identisch sind, in die Variablen *bx* und *by*. Weiters wird der Wert von *bclick* von 2 auf 3 geändert um nun mit der Erfassung des letzen Koordinatenpunktes beginnen zu können. Doch als nächstes bewegt sich die Maus noch einmal und wir gelangen über *PaintBox1MouseMove* zu *tort2*, mit der man endlich in der Lage sein wird, die erste Vorschau des Tortenstückes, welches durch die Prozedur *pie*, die vier bekannte Koordinaten als Parameter besitzt, angezeigt wird, erblicken können. Die Koordinaten *lx*, *ly*, *llx* und *lly* bestimmen die Eckpunkte der Ellipse, während *bx* und *by*, wie bereits erwähnt den Endpunkt der ersten Geraden beschrieben. Die Variablen *x* und *y* enthalten in der Vorschau nun, die sich ändernde, Position des Mauszeigers und sind damit die Koordinaten des Endpunktes der zweiten Geraden, deren Bedarf darin besteht das Tortenstück eindeutig abzugrenzen. Dieser Punkt wird durch das Aufrufen der Prozedur *tort1* mittel Mausklick fixiert und nun wird das vom Benutzer entworfene Tortenstück endlich in die Zeichenfläche auf Dauer eingefügt. Abschließend wird in der zuletzt genannten Prozedur das viel verwendete *bclick* wieder auf Null gesetzt, um den Anwender die Möglichkeit zu geben, ein weiteres Tortenstück aufzuziehen.

3.1.4 Farbradierer

Ein klassischer Radierer entfernt den Blei- oder Farbstift vom Papier, damit es wieder die ursprüngliche, meist weiße, Farbe wiederbekommt. In vielen Zeichenprogrammen, auch in meinem Programm Multidraw, wird ein Radierer verwendet, der das Gezeichnete mit der Hintergrundfarbe übermalt. Doch ich habe mir die Freiheit genommen und noch einen weiteren etwas spezialisierteren Radierer einzubauen, den sogenannten Farbradierer. Dieses Werkzeug besitzt die Fähigkeit nur eine bestimmte, von Anwender ausgewählte, Vordergrundfarbe durch die Hintergrundfarbe zu ersetzen. Ich möchte sie an dieser Stelle darauf hinweisen, dass dieser Radierer nur vor kleinere Arbeiten gedacht, denn wenn man eine Farbe auf der Zeichenfläche vollständig durch eine andere zu ersetzen beabsichtige, so empfehle ich im Menü Filter den Menüpunkte „Farbe austauschen„ zu verwenden. Aber es gibt ohne Zweifel zahlreiche Gelegenheiten, wo sich auch der Farbradierer als nützlich erweist. Dieses Hilfsmittel funktioniert auf Basis zweier Prozeduren.

```
procedure tform1.farbr1(x,y:integer);
```

```
begin
 backspeicher;
 for i:=(x-round(dick/2)+varx) to (x+round(dick/2)+varx) do
 for e:=(y-round(dick/2)+vary) to (y+round(dick/2)+vary) do
 begin
 if puffer.canvas.pixels[i,e]=vfarb
 then
 puffer.canvas.pixels[i,e]:=hfarb;
 end;
end;
```

Über *PaintBox1MouseDown* wird *farbr1*, welche wiederum als Parameter die Mauskoordinaten beisitzt, aufgerufen. Zuerst wird das aktuelle Bild für das mögliche Widerrufen mittels *backspeicher* gesichert. Da dieses Werkzeug pixelgenau arbeiten muss, verwende ich zwei *for* – Schleifen, welche alle Bildpunkte in einem der gewählten Dicke breiten Quadrat, abdecken. Anschließend wird jeder Pixel dahingehend überprüft, ob seine Farbe mit der gewählten Vordergrundfarbe *vfarb* identisch ist. Sollte dies zutreffend sein, dann wird die Farbe dieses Pixels durch die Hintergrundfarbe *hfarb* ersetzt.

```
Procedure Tform1.farbr2(x,y:integer);
 begin
  if mal=true
  then
  begin
  for i:=(x-round(dick/2)+varx) to (x+round(dick/2)+varx) do
  for e:=(y-round(dick/2)+vary) to (y+round(dick/2)+vary) do
  begin
  if puffer.canvas.pixels[i,e]=vfarb
  then
  puffer.canvas.pixels[i,e]:=hfarb;
  end;
      end;
 end;
```

Die Prozedur *farbr2*, die durch die Bewegung der Maus gestartet wird, entspricht im wesentlichen *farbr1*. Der einzige Unterschied ist, dass an der Stelle von *backspeicher*, die

Variable *mal* in einer *if* – Abfrage prüft, ob zur Zeit auch die Maustaste gedrückt ist. Die Funktionsweise dieses Radierers ist simple, doch es lassen sich damit gute Effekte erzielen.

3.2 Die RGB-Zerlegung

Einer der bedeutungsvollsten Vorgänge in meinem Programm ist zweifelsohne die RGB-Zerlegung. In Delphi werden alle Farben als Integerzahlen verarbeitet. Die Werte dieser Zahlen sollte man in das Hexadezimalsystem übertragen um sie dort leichter analysieren zu können. Jedenfalls wenn man beabsichtigt den Mittelwert zwischen 2 Bildschirmfarben auszurechnen so darf man nicht einfach beide Integerwerte addieren und anschließend durch zwei dividieren. Es würde ein falsches Ergebnis liefern, da diese Zahlen die Farbwerte von Rot Grün und Blau enthalten. Um zu einem korrekten Ergebnis zu gelangen, besteht die Notwendigkeit den Integerfarbwert zuerst einmal in das Hexadezimalsystem umzuwandeln und dann die ersten zwei Stellen für Blau, die zweiten zwei für Grün und das letzten Paar für Rot zu nehmen. Mit diesen drei Werten kann man dann in konventioneller Weise rechnen und wenn man plant die neu berechnete Farbe wieder anzeigen zu lassen, so müssen sie die Werte wieder zurück in eine Integerzahl umrechnen. Bevor ich mit weiteren Erläuterungen fortfahre möchte ich erstmals die entsprechende von mir entwickelte Prozedur vorstellen.

```
procedure tform1.aufschlussel(f,z:integer);
begin
blau[z]:=f div (256*256);
w:=f mod (256*256);
grun[z]:=w div 256;
rot[z]:=w mod 256;
end;
```

Auch wenn der Vorgang kompliziert klingt, so ist seine Umsetzung doch relativ simpel. Der Prozedur *aufschlussel* werden zwei Integerzahlen als Parameter übergeben, wobei *f* die Farbe ist, die aufgeschlüsselt werden soll und *z* hat die Funktion das richtige *Array* – Feld anzusprechen, da es öfters nötig ist mehrere Zahlen nebeneinander umzuwandeln. Die Variable *blau*, *grun* und *rot* dienen zu Speicherung des Endergebnisses der Umformung, während w lediglich ein Zwischenspeicher ist. Wenn man dann die Farbe entsprechend seinen Wünschen und Vorstellungen editiert hat, sollte man auch wieder in der Lage sein sie zurück ins Integerformat zu transformieren.

```
f:= (blau[1] shl 16)+ (grun[1] shl 8 )+ (rot[1]) ;
```

Dies ist ein Beispiel wie man eine Rückumwandlung zu Stande bringen kann. Die Variablen hier haben den gleichen Zweck wie in der vorhergehenden Prozedur.
Die RGB-Zerlegung kann man überaus vielseitig anwenden. Ich habe sie für den Weichzeichner und in Folge auch für die Unschärfemaskierung verwendet. Große Dienste hat mir diese Prozedur auch bei zahlreichen weiteren Filtern und Effekten geleistet. Als ein Beispiel für ihre Anwendung will ich nun die Erschaffung eines fließenden Überganges von einer Farbe zu einer anderen anführen. Dies ist Teil einer Prozedur, sollte man in Fenster die richtigen Einstellungen auswählen, die im Menü Effekte unter dem Menüpunkt Muster gestartet werden kann.

```
aufschlussel(form15.shape1.brush.color,1);
aufschlussel(form15.shape2.brush.color,2);

rschritt:=(rot[2]-rot[1])/puffer.height;
gschritt:=(grun[2]-grun[1])/puffer.height;
bschritt:=(blau[2]-blau[1])/puffer.height;

for i:=0 to puffer.width-1 do
for e:=0 to puffer.height-1 do
puffer.canvas.pixels[i,e]:=(round(blau[1]+bschritt*e) shl 16 +
round(grun[1]+gschritt*e) shl 8  + round(rot[1]+rschritt*e) );
```

Wie bereits erwähnt ist dies nur eine von vielen Anwendungsmöglichkeiten dieser so hilfreichen Prozedur. Es steckt meiner Meinung nach eine beeindruckende Menge von mathematischen Überlegungen hinter einer am Ende doch so einfachen Prozedur.

3.3 Funktionen und Fraktale

Es mag zwar etwas ungewöhnlich klingen, aber durch Mathematik lassen sich beeindruckende grafische Effekte erzielen. Im Grunde genommen handelt es sich bei Fraktale um eine relativ einfach Vorschrift(Gleichung(en)), welche immer wieder angewandt werden. Auf diese Art und Weise bringt man eine ungewöhnliche Vielfalt an ästhetisch ansprechenden Mustern zusammen. Ich mir auch die graphische Wirkung von Graphen verschiedenster Funktionen zu Nutze gemacht, um die Bandbreite der zur Verfügung stehenden

künstlerischen Hilfsmitteln zu erweitern und auch, die hinter meinen Programm stehende Mathematik, ein wenig mehr zu Tage treten lassen.

3.3.1 Mandelbrot-Menge

Das Bild des sogenannten Apfel-Männchens ist vielen Menschen ein Begriff, doch nur wenigen wissen um seine Bedeutung und was geschieht, wenn man die Randbereiche vergrößert. Es handelt sich bei der Mandelbrot-Menge einfach um die grafische Darstellung einer rekursiven Gleichung auf der Gauss'schen Zahlenebene. Jedes Pixel entspricht einer imaginären Zahl, welche durch eine Rekursion läuft, bis ihr Wert einen vordefinierten Bereich verlässt.

```
procedure TForm5.mbm;
begin
dd:=dstartx/2;
d:=dstarty/2;
a:=astart;
b:=bstart;

   for i:=0 to 720 do begin
     b:=bstart;
     for e:=0 to 480 do begin
       x:=xs;
       y:=ys;
       z:=0;
       while (z<ill) and (sqrt(x*x+y*y)<(rad*rad/10000)) do begin
         inc(z);
         xx:=x*x-y*y+a;
         y:=2*x*y+b;
         x:=xx;
       end;

        pic3.canvas.Pixels[i,e]:=(ill-z)*f;

       b:=b+d;
     end;
     a:=a+dd;

   end;
```

```
    pic.canvas.stretchdraw(rect(0,0,360,240),pic3);
    Paintbox1.canvas.draw(0,0,pic);

end;
```

Die Prozedur *mbm* wird aufgerufen um die Mandelbrotmenge zu berechnen und auch umgehend anzuzeigen. Die Variablen *d* und *dd* erhalten von *dstartx* und *dstarty* die horizontalen und vertikalen Schrittweiten, also die Zoomfaktoren. Weiters werden *a* und *b*, die vom Benutzer gewählten Startwerte, welche den Koordinaten der linken oberen Ecke auf der Gauss'schen Zahlenebene entsprechen, zugewiesen. Nun werden mittels zweier *for* – Schleifen die Werte aller Pixel, welche logischer Weise mit imaginären Zahlen auf der Ebene gleichzusetzen sind, in der rekursiven Gleichung berechnet. Die Variablen *x* und *y* erhalten im Normalfall den Wert Null vor jeder Berechnung, doch es steht dem Anwender frei eine kleine Abweichung(steht in *xs* und *ys*) hinzu zugeben. Dies lässt das Apfelmännchen „angeknabbert„ erscheinen und ermöglicht eine größere Vielfalt an Darstellungen. Die Anzahl der Iterationen (Durchläufe) wird in der Integervariablen *z* mitgezählt. Die *while* – Schleife enthält zwei Abbruchbedingungen: Einerseits darf der Wert der immer wieder neu berechneten Zahl aus der Menge C nie einen vorbestimmten Betrag überschreiten, andererseits würde die Schleife bei manchen Zahlen schier endlos laufen und daher besteht der Bedarf eines Abbruches nach einer gewissen Anzahl an Durchläufen. Die in der Mandelbrotmenge verwendete Gleichung, in gängiger Schreibweise, lautet:

$$Z_{n+1}=Z_n^2 + C$$

Doch um sie programmiertechnisch umzusetzen, musste ich sie in zwei zerlegen, eine für den Realteil und eine für den Imaginärteil der Zahl. Die Prozedur schließt mit dem Zeichnen des Bitmaps auf die Zeichenfläche. Doch nun möchte ich endlich zu den grafischen Merkmalen der Mandelbrotmenge überleiten, den diese zeigen eine besondere Mischung von Chaos und Ordnung auf. Wenn man nahe des Überganges zum schwarz gefärbten Zentrum in das Apfelmännchen hineinzoomt, so wird dem Betrachter auffallen, dass sich immer wieder unterschiedliche Formen und Muster wiederholen. Dieses Phänomen wird Selbstähnlichkeit genannt und weiters ist es faszinierend zu wissen, dass sich die Farben, welche die Anzahl der Iterationen bis zum Überschreiten des Radius angeben, stark unterscheiden, wenn der Unterschied von der Ausgangszahl auch nur bei der fünften Stelle hinter dem Komma liegt. Mich persönlich hat die Vielfalt und die ästhetische Anmut, der durch eine noch so kleine

Variation entstehenden Gebilde beeindruckt. Doch nicht nur durch Zoomen kann man in ein zauberhaftes Spiel von Farben und Formen eintauchen. Es besteht die Möglichkeit, durch verändern(verkleinern) des Radius, das Apfelmännchen zu „verstümmeln„ bis es sich im Extremfall zu einem verschwindenden Kreis entwickelt. Eine weitere Option gibt Ihnen die Chance ein verzerrtes und „angeknabbertes„ Apfelmännchen zu bestaunen. Der Anwender ist vom Programm her befähigt eine kleine Abweichung in die Gleichung einzubringen, welche diesen Effekt erzielt, doch sollte man die Werte nie größer als eins werden lassen, da sich ansonsten die Figur verflüchtigt. Ich möchte nun diesen Abschnitt mit einem Zitat von Benoit Mandelbrot abschließen:

Wolken sind keine Kugeln, Berge keine Kegel, Küstenlinien keine Kreise. Die Rinde ist nicht glatt – und auch der Blitz bahnt sich seinen Weg nicht gerade... Die Existenz solcher Formen fordert uns zum Studium dessen heraus, was Euklid als formlos beiseite lässt, führt uns zur Morphologie des Amorphen. Bisher sind die Mathematiker jedoch dieser Herausforderung ausgewichen. Durch die Entwicklung von Theorien, die keine Beziehung mehr zu sichtbaren Dingen aufweisen, haben sie sich von der Natur entfernt. Als Antwort darauf werden wir eine neue Geometrie der Natur entwickeln und ihren Nutzen auf verschiedenen Gebieten nachweisen. Diese neue Geometrie beschreibt viele der unregelmäßigen und zersplitterten Formen um uns herum - und zwar mit einer Familie von Figuren, die wir Fraktale nennen werden[12].

3.3.2 Rekursionen

Rekursionen haben einen etwas anderen Charakter als Fraktale des Typs Mandelbrot, da sie die Zeichenfläche nicht vollständig ausfüllen, sondern nur ein verzweigtes Netzwerk aus Gitterlinien darstellen. Auf Grund ihrer Beschaffenheit empfiehlt sich ein Objekt vom Typ *TTurtle* um eine derartige Figur zu erzeugen. Normaler Weise handelt es sich um Vorschriften, die sich immer wieder ineinander wiederholen und den Winkel und die Länge der Turtelbewegung entsprechend abändern. Ich werde die simpelste Version einer Rekursion als erstes Beispiel anführen und dann erst auf die bekannteren Typen überleiten.

```
procedure TForm20.r1(l,a:extended);
begin
if l>ml
```

[12] Benoit Mandelbrot 1975 zitiert nach online im Internet: http://www.mathe-online.at/materialien/matroid/files/fraktale/fraktale.html (31. 7. 2002)

```
then
begin
l:=(l/ld);
a:=a+aa;
turtle1.Turn(a);
turtle1.draw(l);

r1(l,a);
end
else
begin
l:=sl;
a:=sa;
inc(i);
if i<maxi
then
r1(l,a);
end;

end;
```

Diese Prozedur, *r1* genannt, enthält als Parameter die beiden Extendervariablen *l* und *a*, welche, die vom Benutzer gewählte (später die vom Programm bearbeitete), Länge und den Winkel angeben. In der kommenden *If* – Abfrage wird durch Vergleich der Länge *l* mit der Variablen *ml*(Minimallänge) eine sinnlose Verkürzung verhindert und der folgende Prozess eingeleitet. Im nächsten Schritt wird die *Turtle*, entsprechend den Anweisungen, neu ausgerichtet und eine Linie in der vorgesehenen Länge gezogen. Es folgt ein Wiederaufruf der Prozedur *r1*, mit nun abgeänderten Parametern. Doch irgendwann kommen wir schließlich zur Möglichkeit einer Unterschreitung von *ml* und die Integervariable *i* wird erhöht um nach einer ausreichenden Anzahl von Durchläufen einen Abbruch(Ende des Aufrufens der Prozedur) einleiten zu können. Sollte es noch nicht zu einem Abbruch kommen so werden die Länge *l* und der Winkel *a* auf ihre Startwerte zurückgesetzt und *r1* ein weiteres Mal aufgerufen. Diese Rekursion zeigt gut auf, dass es sich um nichts weiters als eine Ändern von Variablen in einer sich ständig selbst aufrufenden Prozedur, handelt, nur mit dem Zusatz, dass dieser Vorgang mittels *TTurtle*, auch sichtbar gemacht wird.

```
procedure Tform20.baum(l,w:extended);
```

```
begin
  if (l>spinedit7.value/dx) then
   begin
     turtle1.draw(l/2);
     turtle1.turn(w);
     baum(l/2,w);
     turtle1.draw(l/2);
     turtle1.move(-l/2);
     turtle1.Turn(-w);
     baum(l/2,w);
     turtle1.draw(l/2);
     turtle1.move(-l/2);
     turtle1.turn(-w);
     baum(l/2,w);
     turtle1.draw(l/2);
     turtle1.move(-l/2);
     turtle1.turn(w);
     turtle1.move(-l/2);
   end;

end;
```

Es gibt auch Rekursion, welche die fragile Struktur eines Baumes darstellen können, doch auch diese funktionieren nach dem gleichen Prinzip wie die vorhergehende. Der Hauptunterschied in dieser sich wiederholenden Zeichenvorschrift zur ersten liegt daran, dass diese sich einerseits wesentlich öfters selbst aufruft und andererseits, dass nur eine *If* - Abfrage existiert, deren Zweck es ist eine Endlosschleife zu verhindern. Bei der Baumrekursion ist der vom Anwender ausgesuchte Winkel entscheiden für ihr Erscheinungsbild. Ein spitzer Winkel lässt den Baum eher steil gewachsen aussehen, während eine stumpfer Winkel die Grafik in ein fast deltaförmiges Gebilde verwandelt.

```
Procedure TForm20.koch(laenge:extended);
Begin
  If laenge>2/(dx*25) then
   begin
     koch(laenge/2);
     turtle1.turn(60);
     koch(laenge/2);
```

```
      turtle1.turn(-120);
      koch(laenge/2);
      turtle1.turn(60);
      koch(laenge/2);
      End
    Else turtle1.draw(laenge);
end;
```

Die Kochkurve ist mit Abstand eine der bekanntesten Rekursionen. Die Prozedur *koch* benötig lediglich eine vorgegebene Länge *laenge* als Parameter vom Typ *extended* um sich darstellen zu können. Da im Falle der Kochkurve die Winkel vordefiniert sind, läuft innerhalb der Prozedur lediglich eine Längeverkürzung und ein sich selbst Aufrufen ab. Hier möchte ich noch anmerken, dass in allen meinen Prozeduren ein Faktor *dx* vorkommt, welcher nur für das korrekte Zoomen zuständig ist. Auf Grund ihrer Beschaffenheit ist das verzerrte Zoomen bei Rekursionen äußerst schwierig zu programmieren, da sich der Winkel entsprechend des Faktors angepasst werden müsste und daher habe ich mich auf eine einfacher Methode zur Vergrößerung beschränkt.

3.3.3 Polynomfunktionen

Zu den, meiner Meinung nach, einfachsten mathematisch Funktion gehören die Polynomfunktionen, da ihr Definitionsbereich ganz R umfasst und da man sie leicht differenzieren und integrieren kann. Dies ist auch der Grund, dass ich ausschließlich bei diesen Funktionen, diese Rechenarten zur Verfügung gestellt habe, weil man hier, im Gegensatz zu den anderen, ohne Quotientenregel oder ähnlich aufwändige Verfahren auskommt. Ich halte es für angebracht hier zu erwähnen, dass man sehr leicht eine Symmetrie in Polynomfunktion erzeugen kann. Alle Funktionen diesen Typs sind zur y-Achse symmetrisch, wenn sie nur gerade Potenzen besitzen und zum Nullpunkt, wenn man darin nur ungerade Potenzen vorfindet.

```
procedure Tform11.rechne;
begin
y:=0;

for e:=0 to 7 do
begin
```

```
yy:=f[e];

if e>0
then
for a:=1 to e do
yy:=yy*x;

y:=y+yy;

end;

end;
```

In der Prozedur *rechne* wird aus dem gegeben x-Wert der y-Wert berechnet. Die Variable *yy* dient lediglich als Zwischenspeicher um *y* fehlerfrei zu berechnen. Das *Array f* gibt die Faktoren vor dem jeweiligen $x^{irgendwas}$ an und es steht dem Benutzer frei diese Werte nach seinem Belieben zu editieren. Trotz ihrer Einfachheit können Polynomfunktion eine beachtliche Vielfalt an Graphen beschreiben und eigenen sich gut um Begrenzungslinien für Figuren im Hauptzeichenprogramm zu erstellen.

3.3.4 trigonometrische Funktionen

Trigonometrische Funktionen sind mit anderen Worten Winkelfunktion wie Sinus, Cosinus und Tangens, auf welche ich mich auch in meinem Programm beschränkt habe. Es wird dem Anwender erlaubt sich für Sinus, Cosinus und Tangens je eine kleine Polynomfunktion auszusuchen und weiteres kann mein ein vielfaches der Winkelfunktion beziehungsweise eine Potenz der jeweiligen berechnen lassen. Durch die Addition der Ergebnisse lässt sich eine umfangreiche Bandbreite an teils sehr chaotisch wirkenden Graphen darstellen, welche man anschließend in seine Zeichnung einfügen kann. Zu $\sin^2$ und ähnlichen Funktion möchte ich noch anmerken, dass ich dass Programm aus programmiertechnischen Gründen nur die Möglichkeit gegeben habe ganzzahlige Hochzahlen zu verarbeiten. Alle anderen Eingaben werden auf Werte der Zahlenmenge $\mathbb{Z}$ gerundet. Meiner Meinung nach ist die Ergänzung eines Zeichenprogramms mit Funktionsgraphen und Fraktale sicherlich eine Bereicherung, welche ganz neue Perspektiven für den Anwender eröffnet.

3.4 Rotation und Spiegelung

Eine sehr gängige Funktion Bilder zu drehen und zu spiegeln und damit das Erscheinungsbild ändern ohne die Symmetrie zu gefährden. Man kann derartige Methoden dazu verwenden um Grafiken in mehren Perspektiven zu präsentieren.

3.4.1 Rechter Winkel

Die einfachste Drehung erfolgt um 90°, da man hierbei auf die Winkelfunktionen Cosinus und Sinus verzichten kann und mit einem einfachen Austauschen der Pixel sein Ziel erreicht. Dies kommt zu Stande, da die Werte der beiden trigonometrischen Funktionen bei Vielfachen von Π/2 in Radiant(bzw. 90° Altgrad) entweder 0, 1 oder –1 annehmen. Ich habe mein Zeichenprogramm dazu befähigt Drehungen um 90° Grad in beide Richtungen und auch 180° durchführen zu können.

```
procedure TForm1.N90nachrechts1Click(Sender: TObject);
begin

backspeicher;
zc:=true;

puffer.width:=spinedit2.value;        //Höhe breite umkehren
puffer.height:=spinedit1.value;

for i:=0 to puffer.width do
for e:=0 to puffer.height do
begin
puffer.canvas.pixels[i,e]:=bild.canvas.pixels[e,puffer.width-i];
end;

spinedit2.value:=puffer.Height;
spinedit1.value:=puffer.width;

if spinedit1.value-paintbox1.width>=0
then
scrollbar1.max:=spinedit1.value-paintbox1.width
else
scrollbar1.max:=0;
```

```
scrollbar1.position:=0;
varx:=0;

if spinedit2.value-paintbox1.height>=0
then
scrollbar2.max:=spinedit2.value-paintbox1.height
else
scrollbar2.max:=0;

scrollbar2.position:=0;
vary:=0;

paintbox1.Canvas.Brush.color:=clsilver;
paintbox1.Canvas.pen.color:=clsilver;
paintbox1.canvas.fillrect(rect(0,0,paintbox1.width,paintbox1.height));

Paintbox1.canvas.draw(-varx,-vary,puffer);

end;
```

Die Prozedur *N90nachrechts1Click* wird durch das Klicken auf dem Menüpunkt „90° Grad nach rechts„ ausgelöst und beginnt mit der üblichen Prozedur *backspeicher*. Anschließend müssen die Höhe und die Breite des Bildes vertauscht werden, da es ansonsten bei „nicht-quadratischen„ Grafiken einen Teil abschneiden würde. Das eigentliche Kernstück der Prozedur folgt nun in Form zweier *for* – Schleifen, welche ein einfaches Austauschen der Pixels des *puffer* – Bitmaps mit denen des im Hintergrund gespeicherten *bild* – Bitmaps. Nach diesem Vorgang schließt die Prozedur mit dem Anzeigen des gedrehten Bildes und der dazu nötigen Vorbereitungen(Anzeigen anpassen und altes Bild löschen).

3.4.2 Genauer Winkel

Manchmal ist auch eine Drehung um Winkel, welche nicht 90° oder 180° Grad entsprechen von Nöten, daher habe ich mein Programm mit einer Einstellung ergänzt, welche eine Drehung von Bilder um einen beliebigen Winkel ermöglich und auf Wunsch kann dieser Vorgang sogar automatisiert werden.

```
procedure Tform1.rotier(winkel:extended);
var dx,dy:integer;
```

```
    dia,be:extended;
    begin
backspeicher;

mmx:=bild.width div 2;
mmy:=bild.height div 2;

dia:=sqrt(puffer.width*puffer.width/4+puffer.height*puffer.height/4);
be:=arccos(puffer.width/(2*dia));

if ((winkel>=0) and (winkel<=90)) or ((winkel>=180) and (winkel<=270))
then
begin
puffer.width:=abs(round(dia*cos((winkel/180*PI)-be))*2);
puffer.height:=abs(round(dia*sin((winkel/180*PI)+be))*2);
end
else
begin
puffer.height:=abs(round(dia*sin((winkel/180*PI)-be))*2);
puffer.width:=abs(round(dia*cos((winkel/180*PI)+be))*2);
end;

mx:=puffer.Width div 2;
my:=puffer.height div 2;

for i:=0 to puffer.width-1 do
for e:=0 to puffer.height-1 do
begin
dx:=i-mx;
dy:=e-my;
r:=sqrt(dx*dx+dy*dy);
alpha:=arctan2(dy,dx);
alpha:=alpha-(winkel/180*PI);
dx:=round(r*cos(alpha));
dy:=round(r*sin(alpha));
lx:=mmx+dx;
ly:=mmy+dy;
puffer.canvas.pixels[i,e]:=bild.canvas.pixels[lx,ly];
```

```
end;

spinedit1.value:=puffer.Width;
spinedit2.value:=puffer.height;

if spinedit1.value-paintbox1.width>=0
then
scrollbar1.max:=spinedit1.value-paintbox1.width
else
scrollbar1.max:=0;

scrollbar1.position:=0;
varx:=0;

if spinedit2.value-paintbox1.height>=0
then
scrollbar2.max:=spinedit2.value-paintbox1.height
else
scrollbar2.max:=0;

scrollbar2.position:=0;
vary:=0;

    paintbox1.Canvas.Brush.color:=clsilver;
paintbox1.Canvas.pen.color:=clsilver;
paintbox1.canvas.fillrect(rect(0,0,paintbox1.width,paintbox1.height));

      Paintbox1.canvas.draw(-varx,-vary,puffer);
   end;
```

Der Umfang der Prozedur *rotier* zeigt sofort um wie viel komplizierte eine Drehung um einen x-beliebigen Winkel sein kann. Die Integervariablen *dx* und *dy* geben den Abstand des zu drehenden Pixels vom Mittelpunkt an, während *dia* den Durchmesser des ursprünglichen Bildes aufnimmt und *be* eine für die Berechnungen notwendige Dreiecksseite dient. Der nächste Teil der Prozedur berechnet die neue Höhe und die neue Breite der rotierten Grafik, welche sich mitunter stark verändern. Dies hängt mit der Wahrung der Größenverhältnisse des Bildes zusammen, weil ich auf diese Art verhindere, dass ein Stückchen abgeschnitten wird. Weiters schafft diese Eigenschaft eine leere Fläche, welche am größten bei 45° Grad ist,

und sich zwischen den neuen Pixel des gedrehten Bildes und dem Rand des Bitmaps erstreckt. Eine *If* – Abfrage überprüft auch welche Streckungsformel, der als Parameter an die Prozedur gegeben Winkel, *winkel* benötigt. Nachdem die Seitenlängen für das neue Bild angepasst wurden, kann die eigentliche Drehung, welche wiederum für jedes Pixel einzeln berechnet werden muss, beginnen. Hierzu wünsche ich nur noch anzumerken, dass ich von jeden neuen Pixel aus zu rechnen beginne und dann die Farbe des Pixels, welcher nach einer Drehung um den gegeben Winkel an diese Stelle käme, dem Pixel auf dem Bitmap zuweise. Auf diese Art und Weise verhindere ich, dass Löcher, welche durch Rundungsfehler entstehen, auftauchen und erhöhen somit die Qualität des rotierten Bildes. Der letzte Teil der Prozedur *rotier* passt lediglich die Zeichenfläche an das neue Bild an und fügt es auch am zuvor genannten Ort ein.

3.4.3 Spiegelungen an den Achsen

Neben dem Rotieren ist das Spiegeln eine weitere gängige Methode um ein Bild zu verändern. Am einfachsten ist es, wenn man eine Grafik entweder entlang einer horizontalen oder einer vertikalen Achse spiegelt. Im Falle einer Spiegelung an einer waagrechten Achse nimmt man üblicher Weise diejenige die an der halben Höhe des Bildes liegt und anderenfalls gebraucht man die halbe Breite.

```
procedure TForm1.horizontal1Click(Sender: TObject);
begin
backspeicher;
for i:=0 to bild.width do
for e:=0 to bild.height do
puffer.canvas.Pixels[i,puffer.height-e-1]:=bild.canvas.pixels[i,e];
paintbox1.canvas.draw(-varx,-vary,puffer);
end;
```

Die Prozedur *horizontal1Click* wird durch einen Klick auf dem entsprechenden Menüpunkt ausgelöst. Wie gewöhnlich fängt die Prozedur, da sie das Aussehen des Bildes verändert, mit *backspeicher* an. Es folgen zwei *for* – Schleifen, welche jeden Pixel auf der Grafik abdecken. Durch einfaches vertauschen von Bildpunkten gelingt die Spiegelung an der horizontalen Achse und man kann das Bild auf die Zeichenfläche malen.

```
procedure TForm1.vertikal1Click(Sender: TObject);
begin
```

```
backspeicher;
for i:=0 to bild.width do
for e:=0 to bild.height do
puffer.canvas.Pixels[puffer.width-i-1,e]:=bild.canvas.pixels[i,e];
paintbox1.canvas.draw(-varx,-vary,puffer);
end;
```

Dies ist die Prozedur *vertikal1Click*, welche das Bild an der vertikalen Achse spiegelt. Man kann klar erkennen, dass der einzige Unterschied der beiden Prozeduren in der Art der Vertauschung der Pixel liegt, aber ansonsten sind sie völlig identisch.

3.4.4 Spiegelungen an den Meridianen

Eine etwas unüblicher Art zu Spiegeln ist es, wenn man die erste oder die zweite Meridiane dazu verwendet. Diese Geraden sind als Linien definiert, welche durch den Ursprung eines Koordinatensystems(hier: Mittelpunkt) in einen Winkel von 45° laufen. Sie können mathematisch als folgende Gleichungen angeschrieben werden. $y = x$ (erste Meridiane) bzw. $y = -x$ (zweite Meridiane). Der Vorgang hier ist eine weniger komplizierte da sich die Höhe und die Breite des Bildes vertauschen.

```
procedure TForm1.N1Meriadiane1Click(Sender: TObject);
begin
backspeicher;
puffer.width:=bild.height;
puffer.height:=bild.width;
spinedit1.value:=puffer.width;
spinedit2.value:=puffer.Height;

if spinedit2.value-paintbox1.height>=0
then
scrollbar2.max:=spinedit2.value-paintbox1.height
else
scrollbar2.max:=0;
if spinedit1.value-paintbox1.width>=0
then
scrollbar1.max:=spinedit1.value-paintbox1.width
else
scrollbar1.max:=0;
```

```
for i:=0 to bild.width do
for e:=0 to bild.height do
puffer.canvas.Pixels[e,i]:=bild.canvas.pixels[puffer.height-i,puffer.width-e];

paintbox1.Canvas.Brush.color:=clsilver;
paintbox1.Canvas.pen.color:=clsilver;
paintbox1.canvas.fillrect(rect(0,0,paintbox1.width,paintbox1.height));
paintbox1.canvas.draw(-varx,-vary,puffer);
end;
```

Die Prozedur *N1Meriadiane1Click* spiegelt das Bild an der ersten Meridiane und beginnt mit dem bereits traditionellen *backspeicher*. Anschließend werden Höhe und Breite des neuen Bildes vertauscht und diese Änderung auch auf die Anzeigen übernommen. Im nächsten Schritt folgen zwei *for* – Schleifen, welche ähnlich dem Spiegeln an den Achsen, alle Pixel der Grafik vertauschen. Abschließend wird das gespiegelte Bild auf der übermalten Zeichenfläche angezeigt. Das Übermalen ist nötig, da bei einem Austausch von Höhe und Breite ein Stück vom alten Bild nicht durch das neue übermalt werden würden und störend überbleiben würde.

3.5 Formen – Objektorientiert

Eine sehr elegante programmiertechnische Lösung für eine Aufgabe ist es ein neues Objekt zu schaffen, doch steht der Aufwand leider nur selten in Rechnung zum Nutzen. Trotzdem hat es einige bedeutungsvolle Vorteile. Einerseits kann man ein Objekt auch in einer anderen Unit verwenden, wenn sich auf die entsprechende zurückbezieht, andererseits hat es den Vorteil, dass sollte man während des Programmierens bemerken, dass mehr Objekte als geplant nötig sind, so muss man lediglich ein oder mehrere weitere deklarieren, ohne alles umzuändern. Noch einmal kurz erklärt: ein Objekt(oder auch Klasse genannt) kann Eigenschaften und Methoden besitzen. Nehmen wir das Objekt „Hund„ als Beispiel, so kann es unterschiedliche Eigenschaften haben, wie in etwa die Fellfarbe, die Schulterhöhe, die Augenfarbe und noch viele mehr. Eine Methode eines Hundes wäre Sitzen oder das Stöckchen holen. Nun kann man dieses Beispiel auch auf die programmiertechnische Ebene übertragen. Die Eigenschaften eines Objekts wären Variablen mit einem bestimmten Wert, während hingegen die Methoden, die Funktionen und Prozeduren des Objekts darstellen. Um diesen abstrakten

Gedankengang zu verdeutlichen nehmen wir ein *TFigur* an, welches eine geometrische Form sein soll, die sich auf der Zeichenfläche bewegt.

```
  type TFigur=class
x,y,vx,vy,r:extended;
st:integer;
Procedure Init(stNeu:integer; xNeu,yNeu,vxNeu,vyNeu,rNeu:Extended);
Procedure Bewegung(scrd:Tscrdata);
Procedure Anzeigen(leinwand:Tcanvas);
end;
```

Dieses Objekt(in Delphi *class* (=Klasse) genannt) verfügt über die Eigenschaften(Variablen) *x, y, vx, vy* und *r* vom Typ Extended und *st* vom Typ Integer. Darüber hinaus besitzt es drei Prozeduren deren Namen *Init*, *Bewegung* und *Anzeigen* sind.

```
Procedure TFigur.Init(stNeu:integer; xNeu,yNeu,vxNeu,vyNeu,rNeu:Extended);
begin
r:=rNeu;
x:=xNeu;
y:=yNeu;
vx:=vxNeu;
vy:=vyNeu;
st:=stNeu;
end;
```

Dies ist die Prozedur *Init,* in welcher die Eigenschaften einen Wert zugewiesen bekommen. Das Programm übergebt als Parameter, die Werte, welche die Prozedur anschließend den Variablen zuweist. Das Aussehen der Figur wird durch *st* bestimmt, während *x* und *y* die Koordinaten, *r* die Größe und *vx* und *vy* die Geschwindigkeit der grafischen Darstellung des Objektes angeben. In der Prozedur *Bewegung* wird die Art und Weise der Bewegung der Figur berechnet. Entweder es handelt sich um eine „Herumfliegen„ der Figur oder um ein zufälliges Springen von Ort zu Ort. Die Entscheidung wie sich die Figur nun tatsächlich bewegt, hängt von dem Parameter *scrd*, welche vom selbst erstellten Typ *TScrdata* ist. Die nächste Prozedur habe ich *Anzeigen* genannt, da sie die Figur auf die als Parameter übergebene Zeichnfläche *leinwand* zeichnet. Welche geometrische Form, das Programm nun anzeigt, hängt von der Eigenschaft *st,* welche in der Prozedur *Init* einen Wert zugewiesen

bekam. Ich habe das Objekt so allgemein wie möglich gehalten und es wäre daher möglich es auch in einer anderen Unit einzubinden. Noch etwas zur Verwendung dieses Objektes: Man muss es als Variable deklarieren und vor der Verwendung ist unbedingt erforderlich es auch zu erzeugen.

```
figur[i]:=Tfigur.create;
figur[i].init(scrdata.fart,random(screen.width),random(screen.height),round((random(10
00)-500)/100),round((random(1000)-500)/100),random(50));
```

Die Variable *figur* ist ein Array vom Typ *Tfigur*, welches hier mittels des Befehls *create* erzeugt wird. Es ist empfehlenswert anschließend gleich die Prozedur *Init* folgen zu lassen, damit alle Eigenschaften des Objekts definiert sind.

3.6 Anwendung von TMediaplayer

Die Komponente *TMediaplayer* besitzt die Fähigkeit Audiodatei zu laden und abzuspielen. Diese Fähigkeit macht sich mein Programm PicWave zu Nutze, welches in der Lage ist Bilder, Audiodateien und Videos zu öffnen und zu bearbeiten. Das Dateiformat .avi kann auch mit Hilfe des *TMediaplayers* geladen werden. Man kann eine Avi – Datei entweder auf der Oberfläche eines *TAnimates* anzeigen oder es auch in einem eigenen Fenster. Letztere Methode habe ich schließlich vorgezogen, da sich, entgegen den Behauptungen in der Delphi-Hilfe, die Größe des *TAnimates* nicht der Avi-Datei anpasst. Andere Dateiformate die sich zum Abspielen in *TMediaplayer* eignen sind .wav und .mid.

```
if  (ExtractFileExt(dateiname))='.wav'
then
begin
With MediaPlayer1 do begin
    Filename := dateiname;
    DeviceType := dtWaveaudio;
    Open;

end;
l:=MediaPlayer1.length;
Trackbar1.position:=0;
trackbar1.min:=0;
trackbar1.max:=l;
p:=1;
```

```
d:=1000;
mpanzeige;

end
```

In der Ladeprozedur des Programms Picwave kommt dieser Ausschnitt vor, welcher durch die *If* – Abfrage nur aktiv wird, sollte es sich bei der zu öffnenden Datei um eine wav - Audioaufnahme handeln. *Mediaplayer1* ist vom Typ *TMediaplayer* und bekommt gleich am Anfang den Pfad der zu ladenden Datei, welche im String *dateiname* gespeichert ist, zugewiesen. *DeviceType* gibt die Art der Mediendatei an, welche in diesem Fall eine Wav-Datei ist. Mit der Einstellung *dtAutoSelect* würde sich das Programm den entsprechenden Typ anhand der Dateiendung suchen. Anschließend wird die Datei geöffnet und deren Länge in die Variable *l* gespeichert, welche dann zum Einstellen der Anzeigespur *Trackbar1* dient. Die Komponente *TMediaplayer* wird in Delphi auch visualisiert und man kann die dortigen Steuerknöpfe zum Abspielen der Datei verwenden. Die Ladevorgänge bei Avis oder mid – Dateien sind im Grunde genommen die gleichen und da man das Abspielen so und so über die Komponente selbst steuert sind keine weiteren Erläuterungen nötig.

3.7 Laden und Speichern von Bildern

In einem Programm zur Grafikbearbeitung ist das Abspeichern und Laden von Bilder das Um und Auf, da ansonsten das Programm seine eigentliche Funktion nicht erfüllen könnte. Im Grunde genommen handelt es sich hierbei immer um den selben Vorgang, welcher aber durch unterschiedlich Ereignisse ausgelöst werden kann. Wenn man überlegt, wie oft man eine Datei ladet bzw. speichert, wird man die essentielle Bedeutung der Prozedur sofort erkennen.

3.7.1 Speichern

Nachdem die Beareitung eines Bildes mit Hilfe meines Programms abgeschlossen ist, wird man beabsichtigen es auf der Festplatte zu sichern. Delphi stellt hierfür dem Programmierer ein sehr gutes Hilfsmittel zu Verfügung: *TSavePictureDialog*. Dieses Speichermenü ist im wesentlichen mit dem üblichen *TSaveDialog* identisch, nur dass es über ein zusätzliches Betrachtungsfenster für Bilder verfügt. Durch klicken auf dem entsprechenden Menüpunkt oder mittels des Shortcuts Strg + S wird dieser Dialog aufgerufen.

```
procedure TForm1.Speichernunter1Click(Sender: TObject);
begin
```

```
zoomnormal;
backspeicher;
SavePictureDialog1.DefaultExt := GraphicExtension(TBitmap);
SavePictureDialog1.Filter := GraphicFilter(TBitmap);
if SavePictureDialog1.Execute
then
begin
Bild.SaveToFile(SavePictureDialog1.FileName);
dateiname:=SavePictureDialog1.FileName;
form1.caption:='Multidraw - '+dateiname;
end;
end;
```

Dies ist die Prozedur, welche in Multidraw für „Speichern unter„ verantwortlich ist. Anfangs werden hier in *Speichernunter1Click* die Voreinstellungen für das Speichern getroffen. Das Bild wird über *backspeicher* gesichert und *zoomnormal* stellt die Vergrößerung richtig. In *TSavePictureDialog* ist es erforderlich gewisse Filtereinstellungen zu treffen um ein fehlerfreies Sichern der Datei zu ermöglichen. Anschließend wird der Dialog mittels *SavePictureDialog1.Execute* aufgerufen, welches Teil einer *If* – Abfrage ist. Sollte ein Dateiname vom Benutzer ausgewählt worden sein so wird, so wird dieser zum Speichern des Bildes verwendet und auch dem String *dateiname* übergeben, welcher ihn dann im Titel anzeigt. Unabhängig wann oder wo ein Bild gespeichert wird, so verwendet man den Namen der jeweiligen Grafik plus dem Befehl *SavetoFile* mit einem String als Parameter.

```
fract.savetofile(pfad+'\'+inttostr(round(beta*100))+'.bmp');
```

Dies ein winziger Ausschnitt aus der Prozedur *adreh*, in der zahlreiche Bilder innerhalb einer langen Schleife, immer wider nach einer Drehung um einen vorgegebenen Winkel abgespeichert werden. Diesmal wählt zwar nicht der Benutzer den Dateinamen aus, sondern das Programm selbst, aber trotzdem bleibt der eigentliche Speichervorgang der selbe.

3.7.2 Laden

Es hat wenig Sinn Hunderte von Grafiken zu speichern, wenn man nicht vor hat sie irgendwann einmal zu laden. Bei diesem Vorgang wählt man mit Hilfe eines *TOpenPictureDialog* ein bereits gesichertes Bild aus und öffnet es. Üblicher Weise gelangt man zu dem vorhin genannten Dialog über den Menüpunkt „Öffnen„ bzw. „Laden„ oder den

Shortcut Strg+O. Als Beispiel, für solche Prozedur, möchte ich ganz einfach, diejenige die in Multidraw diesen Vorgang steuert, verwenden.

```
procedure TForm1.ffnen1Click(Sender: TObject);
begin
if OpenPictureDialog1.Execute
then
begin
paintbox1.canvas.pen.color:=clsilver;
paintbox1.canvas.brush.color:=clsilver;
paintbox1.canvas.FillRect(rect(0,0,paintbox1.width,paintbox1.height));
            dateiname:=OpenPictureDialog1.FileName;
bild.LoadFromFile(dateiname);

form1.caption:='Multidraw - '+dateiname;
puffer.height:=bild.height;
puffer.width:=bild.width;
puffer.canvas.draw(0,0,bild);
spinedit1.value:=bild.width;
spinedit2.value:=bild.height;

if spinedit2.value-paintbox1.height>=0
then
scrollbar2.max:=spinedit2.value-paintbox1.height
else
scrollbar2.max:=0;

if spinedit1.value-paintbox1.width>=0
then
scrollbar1.max:=spinedit1.value-paintbox1.width
else
scrollbar1.max:=0;

end;
Paintbox1.canvas.draw(-varx,-vary,puffer);
end;
```

Diese Prozedur trägt den Namen *ffnen1Click* und wird durch das Klicken auf dem Menüpunkt „Öffnen„ im Menu Datei ausgelöst. Sie beginnt mit der Überprüfung ob im *OpenPictureDialog1* ein Dateiname ausgewählt und bestätigt wurde. Im nächsten Schritt wird die Zeichenfläche *Paintbox1* gelöscht und das Bild geladen. Anschließend werden alle Anzeigen und Einstellungen im Zeichenprogramm an die Grafik angepasst. Zum Schluss wird noch das Bild auf die Zeichenfläche gemalt.

3.7.3 JPG-Bilder

Die meisten von Delphi unterstützen Grafikformate können direkt in ein Objekt vom Typ *TBitmap* geladen werden, doch dies gilt nicht für JPG-Bilder, da sie nur von *TImage* unterstütz werden, wenn eine entsprechende Unit *jpeg* eingebunden ist. Als logische Konsequenz wird dadurch das Laden und Bearbeiten von Bildern des Dateiformats JPG erschwert, soweit man beabsichtigt, die Vorteile(vor allem Schnelligkeit) der *TBitmap* und *TPaintbox* – Objekte sich zu Nutze zu machen. Doch um soviel aufwendiger ist dieser Vorgang nun doch wieder nicht, da ein simpler Trick ausreicht um ein JPG-Bild in ein Bitmap zu laden.

```
if (ExtractFileExt(dateiname))='.jpg'
then
begin
image1.picture.LoadFromFile(dateiname);
paintbox1.visible:=true;
bild.width:=image1.picture.width;
bild.height:=image1.picture.height;
puf.width:=image1.picture.width;
puf.height:=image1.picture.height;
bild.canvas.draw(0,0,image1.picture.graphic);
puf.canvas.draw(0,0,image1.picture.graphic);

Paintbox1.width:=puf.width;
paintbox1.height:=puf.height;

paintbox1.Canvas.draw(0,0,bild);
p:=2;
end
```

Dies ist ein Ausschnitt aus der Ladeprozedur im Programm Picwave. Sollte es sich beim zu öffnenden Dateityp um ein JPG-Bild handeln so wird diese *If* – Abfrage ausgelöst und anfangs wird die Grafik in das Objekt *image1*, welches von Typ *TImage* ist, geladen. Anschließend werden die beiden Bitmaps *bild* und *puf* an die Größe des geladenen Bildes angepasst. Als Nächstes wird die Grafikoberfläche des Jpegs mittels *canvas.draw* auf die Oberfläche des Bitmaps kopiert, von wo aus man es dann, mit akzeptabler Geschwindigkeit, bearbeiten kann. Dieser Trick umgeht ganz einfach die Inkompatibilität der beiden Formate über die Zeichenfläche der beiden Objekte.

3.8 Verknüpfung von mehreren Formularen

Mein Programm verwendet über 40 selbsterstellte und noch zahlreiche von Delphi zur Verfügung gestellte Units. Jede von mir kreierte Unit unterstütz ein Formular, welches der Anwender für seine Arbeit aufrufen kann. Da jedes Formular irgendwie von einem anderen aus zu mindest gestartet werden muss, ist eine Verknüpfung zwischen ihnen erforderlich; nur das Hauptformular wird beim Öffnen des Programms von selbst aktiviert. Für normal reicht es, wenn man die zum Formular gehörende Unit (ich nenne sie hier „Alpha„) in der *uses* – Deklaration der Unit (in diesem Beispiel trägt sie den Namen „Beta„), von der aus die andere gesteuert wird aufführt. Leider kann es auch erforderlich sein, dass Alpha auf Daten aus Prozeduren von Beta zurückgreifen muss. Der erste logische Schritt wäre Beta in der *uses* – Deklaration von Alpha aufzuführen, doch dies funktioniert nicht, da Alpha bereits Beta „unterstellt„ ist. In diesem Fall darf man Beta nicht im *Interface* – Teil in die *uses* – Deklaration schreiben, sonder man muss sie als eine Art von Hilfsunit im *implementation* – Teil aufführen und dort eine *uses* – Deklaration mit Beta hinzufügen.

```
Unit Beta
Interface
Uses [diverse Units] , Alpha;
[Typen und Variablen]
implementation
[verschienden Prozeduren]
end.
```

```
Unit Alpha
Interface
Uses [diverse Units];
```

```
[Typen und Variablen]
implementation
uses Beta;
[verschiedene Prozeduren]
end.
```

Nach dem Muster dieses Beispiels kann man Formlure unter Einhaltung einer bestimmten Hierarchie untereinander verbinden. Es gibt aber auch noch einen weiteren Weg Formulare zu verbinden. Beim vorherigen Beispiel wurden die Formulare mittels Delphi über das Menü Datei und den Menüpunkt „Neues Formular„ erstellt, doch man ist auch in der Lage Formulare nicht nur einfach vorzufertigen(was leichter und einfacher geht), nein, man kann sie sogar während der Laufzeit aus einem bereits ausgeführten Formular heraus erstellen. Es sollte dazu zuerst einmal eine Variable vom Typ *Tform* deklariert worden sein, welche dann in einer Prozedur ein Formular erzeugt.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
pst:=Tform.create(Form1);
pst.parent:=Form1;
pst.formstyle:=fsStayOnTop;
pst.top:=400;
pst.left:=250;
pst.width:=150;
pst.height:=100;
pst.visible:=true;
pst.borderstyle:=bsdialog;
pst.caption:='VG-Stil';

pstc:=Tcombobox.create(pst);
pstc.parent:=pst;
pstc.top:=10;
pstc.left:=10;
pstc.width:=125;
pstc.height:=20;
pstc.text:='Bitte auswählen';
pstc.items.Add('durchgezogen');
pstc.items.Add('strichliert');
pstc.items.Add('punktiert');
```

```
pstc.items.Add('Strich - Punkt');
pstc.items.Add('Strich - Punkt - Punkt');
pstc.items.Add('unsichtbar');
pstc.items.Add('spezial');

pstb:=Tbutton.create(pst);
pstb.parent:=pst;
pstb.top:=40;
pstb.left:=10;
pstb.width:=130;
pstb.height:=30;
pstb.caption:='Annehmen und schließen';
pstb.onclick:=pbc;

Paintbox1.canvas.draw(-varx,-vary,puffer);
end;
```

In der Prozedur Button1Click sind die Variablen *pst* vom Typ *TForm*, *pstc* vom Typ *TCombobox* und *pstb* vom Typ *TButton* enthalten und es wird beabsichtig ein Formular, welches eine Combobox und einen Button enthält, zu erstellen. Gleich am Anfang wird das Formular *pst* kreiert mittel *Tform.create(form1)*, wobei *Form1* als Parameter, das dazugehörige Hauptformular angibt. Anschließend werden die Eigenschaften von *pst* bestimmt und seine Objekte nach dem gleichen Muster erzeugt. Dies wären nun alle mir bekannten Möglichkeiten um zwei Formulare miteinander zu verknüpfen gewesen.

3.9 Lokalisierung des Bildschirmschoners

Während meiner Arbeiten musste ich mich auch öfters Problemen von größerer Schwierigkeit entgegenstellen und eines davon war die Lokalisierung des ausgewählten Bildschirmschoners. Zuerst einmal ist es nötig die Umstände an sich zu verstehen, bevor ich die Lösung anführen kann. Es liegt leider jenseits meiner Fähigkeiten zur Laufzeit eine exe – Datei nach den Wünschen des Anwenders zu erstellen, doch möchte ich bei meinem Bildschirmschonereditor dem Benutzer eine so breit wie möglich gefächerte Auswahl an Optionen lassen. Daher habe ich mich auf eine Skriptdatei verlegt, welche in einem Ordner, welcher den gleichen Namen wie die dazugehörige scr – Datei trägt, mit den dazugehörigen Grafiken und Sounddateien liegt. Doch wenn der Bildschirmschoner(die scr – Datei) ausgeführt wird, so sollte sie den Namen ihres Ordners kennen. Hier stoßen wir auf das Dilemma: Wie weiß das Programm,

wie es heißt, denn es ist lediglich eine Kopie einer vorgefertigten allgemein gültigen Version, welche nur eine eigene Identität durch die Skriptdatei, welche das Programm aber ohne den Namen des Ordners nicht finden kann, erhält. Ein Paradoxon, da man um die nötige Information zu erhalten, bereits die Information haben muss. Ich habe mir anfangs überlegt mir den Namen der laufenden Datei irgendwie zu Nutze zu machen, doch ohne Erfolg. In meiner folgende Verzweiflung, da zahlreiche weitere Ideen ins Nichts führten, wollte ich sogar die Anzahl der selbsterstellten Bildschirmschoner pro System auf einen beschränken, welcher dann eine fixen Namen, der mit dem des Ordners identisch ist, hat. Doch schließlich fand ich doch noch eine Lösung. In Windows ist es üblich sich seine gewünschten Bildschirmschoner über ein Menü auszuwählen. Als Konsequenz dachte ich mir, dass diese Auswahl irgendwo gespeichert sein müsste. Also änderte ich den Bildschirmschoner des Computers und überprüfte, welche Dateien in der letzten Minute aktualisiert wurden. Doch zu meinem Bedauern machte ich den ersten Versuch auf einen Rechner mit dem Betriebssystem Windows 2000 und die nötig Datei wurde vom System selbst zur Laufzeit benötig und konnte daher nicht gelesen werden. Trotzdem wiederholte ich das Experiment mit Erfolg auf einem Windows Me Rechner. Ich fand in der Datei SYSTEM.INI eine Zeile mit der nötigen Information.

```
SCRNSAVE.EXE=C:\WINDOWS\SYSTEM\STSAVER.SCR
```

So lautete die Lösung dieses schwierigen Problems. Nun konnte ich in meinem vorgefertigten Bildschirmschoner die nötigen Änderungen vornehmen.

```
getdir(0,hier);
iLength := 255;
setLength(sWinDir, iLength);
iLength := GetWindowsDirectory(PChar(sWinDir), iLength);
setLength(sWinDir, iLength);
hier:=sWinDir+'/SYSTEM.INI';

assignfile(sysin,hier);
reset(sysin);
repeat
readln(sysin,zeile);
vergl:=zeile;
setlength(vergl,12);
```

```
until (vergl='SCRNSAVE.EXE') or EOF(sysin)=true;
closefile(sysin);

vergl:='';
setLength(vergl, Length(zeile)-13);
for i:=14 to Length(zeile) do
vergl[i-13]:=zeile[i];
hier:=ExtractFileName(vergl);
setlength(hier,length(hier)-4);

pname:=sWindir+'/SYSTEM/'+hier;}
```

Als erstes muss das Programm einmal die Datei SYSTEM.INI finden, welche ihren fixen Platz im Windowsverzeichnis hat. Als nächstes hatte ich die Zeile in der die nötig Information geschrieben steht zu lokalisieren. Dies löste ich mit Hilfe einer *repeat until* – Schleife welche nach dem passenden Anfangswort sucht. Wenn ich sie dann schließlich gefunden haben, so zerlege ich den String bis ich nur mehr dem Namen der scr – Datei habe. Dieser Name ist mit dem des dazugehörigen Ordners identisch. Durch Hinzufügen der üblichen Lage von Bildschirmschoner(Windowsverzeichnis – Unterverzeichnis „System„) war ich endlich in der Lage die Skriptdatei zu laden und den vom Benutzer ausgewählten, selbst erstellten Bildschirmschoner auszuführen. Der einzige, wenn auch entscheidende, Nachteil dieser Lösung ist, dass man die durch mein Programm FantsicScreen kreierten Bildschirmschoner nur unter dem Betriebssystem Windows Me verwenden kann.

Epilog

Ich fand die Arbeit an dieser Thematik als ausgesprochen interessant und lernte dabei viele neue Möglichkeiten und Gedankengänge kennen. Während des Programmierens oder des Schreibens der Dokumentation hatte noch zahlreiche geniale Eingebungen für weitere Features. Ich bedaure zu tiefst, dass ich bei weitem nicht alles umsetzen kann, was mir noch vorschwebt. Dafür reicht die Zeit beim besten Willen nicht aus und ich vermute ich könnte mein Programm noch ein Jahr lang ausbauen und verbessern. Ich musste sogar eine Ideen, welche in beabsichtigt habe in mein Programm einzufügen, weil es sonst bei weitem zu viel werden würde und ich damit auch noch unzählige weitere Stunden beschäftigt wäre. Daher habe ich Multidraw nicht so weit bringen können um mit einem professionellen Programm(Corel Draw, Adobe Photoshop) zu konkurrieren, doch ich betrachte die Arbeit trotzdem als gelungen. Die Dokumentation wurde umfangreicher als erwartet, obwohl ich zu manchen Themen(besonders im allgemeinen Teil) nur sehr spärliche Literatur fand. Ich schätze aber trotzdem, dass ich die wesentlichen Aspekte ausreichend behandelt habe. Alles in allem brachte mich die Arbeit an meiner FBA auf neue Denkansätze und Ideen und bereicherte mein Wissen und meine Erfahrung im Gebiet der Grafikprogrammierung.

Quellenverzeichnis

- Doberenz Walter / Kowalski Thomas(1997): Bordland Delphi 3 für Einsteiger und Fortgeschrittene, München/Wien :Hanser
- Doberenz Walter / Kowalski Thomas(1997): Bordland Delphi 3 für Profis, München/Wien :Hanser
- Warken Elmar(1997): Delphi 3- Entwicklung leistungsfähiger Anwendungen und Komponenten, Bonn : Addison-Wesley-Longman
- Kaltenbach, Reetz, Woerrlein(1990): Das große Computerlexikon, Frankfurt am Main: Fischer
- Delphi-Hilfe (direkt in der Entwicklungsumgebung)
- Online im Internet: http://www.ralphaltmann.de/bibliothek/fotografik/scharfmacher.html (17. 10. 2002)
- Online im Internet : http://www.mathe-online.at/materialien/matroid/files/fraktale/fraktale.html (31. 7. 2002)
- Online im Internet: http://www.ev-stift-gymn.guetersloh.de/uforum/physik-lk-12-1997-1998/fraktale (31. 7. 2002)
- Online im Internet: http://sebfisch.de/fraktale/feigenbaum.html (28. 8. 2002)
- Online im Internet: http://www.bandlows.de/uni/uni_idx.htm (31. 7. 2002)
- Online im Internet: http://www.pc-seminare.de/images/download/dateiformate.pdf (4. 1. 2003)
- Online im Internet: http://www.home.fh-karlsruhe.de/~keut0001/micrografx/seite0.html (4.1. 2003)
- Online im Internet: http://www.libpng.org/pub/png/ (10. 1. 2003)
- Online im Internet: http://www.khm.de/~computerkurs/ws9900 (18. 1. 2003)
- Online im Internet: http://www.3danimation.de (26. 1. 2003)
- Online im Internet: http://www.scantips.com (1. 2. 2003)