Allgemein

Hi und willkommen zu GLBasic, Deiner neuen Programmiersprache.

Hier lernst Du innerhalb kürzester Zeit wie Du mit GLBasic Deine Spielidee selbst verwirklichen kannst. Ein Wort vorweg: GLBasic ist kein Zaubermittel. Es hat seine Stärken und auch seine Grenzen. GLBasic lächelt über Aufgaben wie Sprite-Rotationen, AlphaBlending und Zoomen. (Wenn Du nichts davon kennst, egal. Wird alles später erklärt.).Um stabile Geschwindigkeiten auf allen Rechnen zu erhalten, kann GLBasic wenn nötig auf eine einstellbare Anzahl von Bildern pro Sekunde gebremst werden.

Kapitel 1: Programmiergrundlagen.

Ein Programm kann grundsätzlich fast gar nichts. Und genau das ist das Schwierige daran. Programme können nur:
-Variablen deklarieren
-Sprünge zu anderen Programmteilen machen
-Rechenoperationen mit Variablen durchführen
-Variablen an Funktionen übergeben

1.1 Grundlegendes zu GLBasic:

GLBasic ist eine Programmiersprache. Wenn auch eine sehr einfache. Programmiersprachen haben ihre eigene Syntax (Grammatik). Solange Du sie 100% einhältst, versteht GLBasic Dich auch zu 100%!

GLBasic beendet jeden Programmbefehl mit einem ';' oder einer neuen Zeile.

Alle Befehle von GLBasic müssen IMMER GROSS geschrieben werden. Der Editor tut das von selbst, wenn er einen Befehl erkennt .

Jedes Zeichen RECHTS von einem '//' oder dem Befehl 'REM' wird als Kommentar zum Quellcode (also dem Programm) aufgefasst und hat keine Wirkung auf das Programm selbst. Es macht es nur übersichtlicher.

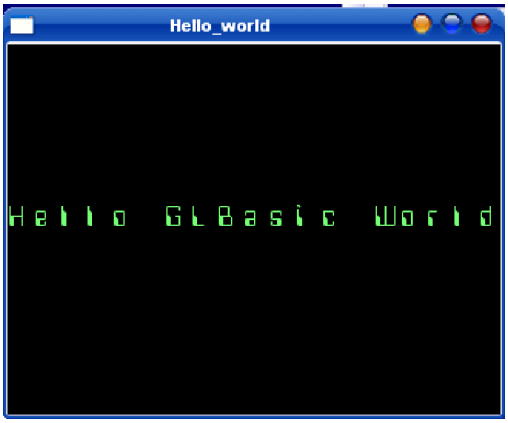
GLBasic benützt versteckte Bildschirmseiten für flackerfreien Grafikaufbau. Wenn Du irgend etwas mit GLBasic programmierst, siehst Du zunächst gar nichts! Erst wenn Du den Befehl 'SHOWSCREEN' aufrufst, wird Deine gerade bearbeitete Seite sichtbar, und Deine neue zu bearbeitende Seite wird gelöscht, bzw. mit dem geladenem Hintergrundbild versehen. Das ist jetzt etwas früh, trotzdem merken: Wenn Du nichts siehst, überlegen ob Du 'SHOWSCREEN' vergessen hast.

1.2 Unser erstes Programm:

Starte den Editor und wähle im Assistenten "Neues Projekt". Gib als Namen "HelloWorld" ein und klick auf OK. Nun tippe folgendes Programm in den Editor, speichere alles (Menü: Datei/Alles Speichern) und Kompiliere (erstelle) dann das Programm mit dem Sandeimer-Knopf oder Menü: Compiler/Compilieren.
Nun starte es mit dem Joypad Knopf, oder wähle Menüpunkt Compiler/Run

```
///////////////
// HELLO WORLD
///////////////
// Mein erstes Programm
PRINT "HELLO WORLD!" ,100,100
SHOWSCREEN
MOUSEWAIT
END
```

Du solltest nun einen schwarzen Bildschirm mit der Aufschrift 'HELLO WORLD!' sehen.



Wenn Du auf die Maus drückst, ist das Programm beendet. Damit hast Du den

ersten Schritt in Deine neue Welt getan.

Genug, fangen wir mit dem Leichtesten an: Variablen deklarieren.

Bei GLBasic gibt es 2 Variablen-Typen: Zahlen und Wörter.
Variablen stellen wir uns als Schubladen vor. Auf der Schublade steht der Name der Variable, also der Variablenname und in der Schublade finden wir dann den Inhalt der Variablen, den Variablenwert. Beginnen wir mit Zahlen und sehen dann, was wir damit so alles anstellen können.

1.3 ZAHLEN

```
LET a=5
```

Wir rufen den Befehl 'LET' auf, und sagen wir hätten gerne eine Variable mit dem Namen 'a' die den Wert 5 beinhalten soll.
Beende den Befehl mit ';' und haben unsere erste Zeile Programm geschrieben.

Damit das Ganze etwas schneller geht, gibt GLBasic die Möglichkeit, das 'LET' auch wegzulassen. Die Zeile sähe dann so aus:

```
a=5
```

und

```
LET a=5
LET a=a+1
```

sähe dann so aus:

```
a=5
a=a+1
```

Was tun wir hier? Nun, wir sagen die Variable mit dem Namen 'a' hat den Wert 'a+1'. Da 'a' aber von vorher schon den Wert 5 hat, weisen wir der Variable 'a' in Wirklichkeit den Wert '5+1' zu, also 6.
Wir können auch andere Rechenoperationen durchführen. GLBasic kennt folgende Operatoren:
+, -, *, /; (Plus=Addition, Minus=Subtraktion, Mal=Multiplikation, Geteilt=Division)

ACHTUNG:

```
LET a= 3+4*5;
```

Wäre laut Adam Riese 3+(4*5)=23 (Punkt vor Strich). Das war bei älteren Versionen von GLBasic nicht so. GLBasic hat zum Berechnen von Termen folgende Hierarchie:
(Klammern zuerst) * / + - AND OR < > =

Es gibt bei GLBasic ein paar Sonderfunktionen, die anders arbeiten als der Rest.
SIN(), COS(), TAN(), RND(), KEY()
Diese Funktionen geben einen Wert zurück. GLBasic arbeitet übrigens mit Gleitkommazahlen. Das heißt, bei Divisionen muss die Funktion INTEGER() verwendet werden, wenn man eine Ganzzahlenoperation durchführen möchte.

Der Befehl RND() gibt eine Zufallszahl zurück. (Random)

```
LET a=RND(50) // a= Zufallszahl von 0 bis einschl. 50
```

1.4 WÖRTER (Zeichenketten)

Wörter unterscheiden sich in ihren Variablennamen durch ein angefügtes '$' von Zahlen. Zahlen lassen sich einfach in Zeichenketten umwandeln. Mehr dazu später in einem Beispiel.

```
LET a$="HALLO"
```

Wie oben geben wir der Variablen mit dem Namen 'a$' den Wert "HALLO"; Wörter die statisch, also fest im Programm verankert sind (Konstanten), müssen immer durch " " abgetrennt werden, um sie von Variablen unterscheiden zu können.
Der Befehl 'LET' ermöglicht ein paar feine Funktionen mit Wörtern.

```
LET a$="MEINE"
LET b$="LIEBLINGSZAHL"
LET c= 7
// Jetzt kommts!!
LET c$=a$ + " " + b$ + " : " + c
PRINT c$,0,20
SHOWSCREEN
```

Ausgabe:
MEINE LIEBLINGSZAHL : 7
Wir können also mit 'LET' Zahlen in Wörter umwandeln (c$=c) und Wörter, sowie statische Texte miteinander verknüpfen (+). Was für einen Sinn das hat? Wenn unser Spiel Level 5 laden soll, muss es den Dateinamen in einem Wort an die Funktion übergeben. Wir könnten also, wenn die Datei heisst: 'Level5.dat' ein Wort bilden indem wir schreiben:

```
LET levelnummer=5 // Das passiert natürlich irgendwo im Spiel...
LET level$="Level" + levelnummer + ".dat"
```

Dazu aber später in einem Beispiel mehr. Das 'LET' ist übrigens auch hier nur zur Übersichtlichkeit und kann auch einfach weggelassen werden.

GLBasic bietet Dir noch ein paar besondere Funktionen an, um mit Wörtern umzugehen.

```
INPUT name$, x, y
```

Gibt an der Stelle x, y einen blinkenden Cursor aus, mit dem Du ein Wort eingeben kannst. Nach Abschluss mit der Return-Taste wird das eingegebene Wort in die Variable 'name$' eingesetzt. Das funktioniert auch mit Zahlen
(INPUT zahl, x,y;)

```
ziel$=MID$(quelle$, start, länge)
```

Dieser Befehl ermöglicht es, aus Wörtern 'Unterwörter' herauszuschneiden. Dabei wird aus der Variablen quelle$ ein Wort, beginnend mit dem Buchstaben Nr. 'start' und der Länge 'länge' in die Variable mid$ geschrieben. Der erste Buchstabe des Wortes quelle$ hat den Startindex '0'.

Beispiel:

```
name$="Mein Auto ist blau"
m$=MID$(name$, 5, 4)
PRINT m$ , 20, 20
SHOWSCREEN
```

Ausgabe:

Auto

Du kannst damit z.B. Namenseinträge in einer Highscoreliste auf 7 Buchstaben beschränken, indem Du einfach nur die ersten 7 Buchstaben herausschneidest.

```
INPUT eingabe$, 100, 100
name$ = MID$(eingabe$, 0, 7)
```

```
wort$ = FORMAT$(numbuchst, numnachkomma, zahl)
```

Damit kann man Zahlen "schöner" in Wörter umwandeln. Behalte das im Gedächtnis, falls Du es einmal brauchen solltest.

1.5 DATENFELDER

Du hast nun einen Überblick über Variablen bekommen. Wie würdest Du aber darin ein Spielfeld speichern? Richtig! Gar nicht. Dafür benutzt Du am besten Datenfelder.
Ein Datenfeld kann man sich vorstellen wie ein Schachbrett. Es hat 8 Felder nach rechts (x) und 8 Felder nach unten (y). Ein solches Feld definieren wir in GLBasic so:

```
DIM schachbrett[8][8]
```

Dadurch wird im Rechner der Speicherplatz vorbereitet, in dem unser Schachbrett gespeichert ist, und alle Felder werden auf 0 gesetzt.
Wollen wir jetzt auf das Feld 3 nach rechts und 4 nach unten den Wert 7 schreiben, so schreiben wir:

```
LET schachbrett[2][3]=7
```

Warum 2 und nicht 3?? Nun weil Datenfelder nun einmal bei 0 beginnen. 0,1,2,3,4,5,6 sind 7 Felder. Zugegeben, das mag etwas verwirren, ist aber nach kurzer Zeit vergessen und wird automatisch richtig gemacht.
Du kannst Datenfelder auch mit Variablen anspringen.

```
LET Schachbrett[2][6]=7
LET x=2
LET y=6
LET a=schachbrett[x][y]
PRINT "Schachbrett[2][6] hat den Wert:", 100, 80
PRINT a, 100, 100
```

Ausgabe:
Schachbrett[2][6] hat den Wert:
7

Weil Du ihn vorher in das Feld geschrieben hast.

ACHTUNG:

Datenfelder dürfen nicht mehr als 4 Dimensionen haben.

```
DIM a[1][2][3][4]; // Gerade noch OK
```

Datenfelder können auch aus Wörtern und mit unterschiedlichen Dimensionen geschaffen werden.
Beispiele:

```
DIM raum[10][10][10] // Ein Zahlenraum mit x, y und z
                     // je 10 Feldern (0-9)
DIM name$[5] // 5 Wörter
LET name$[0]="Hugo"
LET name$[1]="Tim"
DIM kinobesucher$[30][10]
LET kinobesucher$[sitz][reihe]="Tom"
```

Mit dem Befehl REDIM kann man einem Feld eine neue Größe zuweisen und die bereits gespeichterten Datenelemente beibehalten.

Um ein Feld zu kopieren schreibt man:

```
DIM a[5]
b[] = a[]
```

Das erstellt ein neues Feld b[5] und kopiert die Daten von a[] hinein.

Um aus einem Feld eine Reihe herauszunehmen, kann man DIMDEL verwenden. Das ist sehr angenehm, da man dadurch einfach dynamische Anzahlen von Elementen handhaben kann. Mehr Information dazu gibt es in der Befehlsreferenz.

1.6 SPRUNGMARKEN:

Problem: Das Programm läuft einmal durch und ist dann aus.
Lösung: Wir lassen das Programm wieder zum Anfang springen.
Beispiel:

```
// Ein GOTO Beispiel
anfang:
LET a=RND(600)
LET b=RND(400)
PRINT "TOLL",a ,b
SHOWSCREEN
MOUSEWAIT
GOTO anfang
```

Was tut das Programm? Es setzt eine Sprungmarke mit dem Namen 'anfang', sagt 'a' habe einen zufälligen Wert zwischen 0 und 600, b zwischen 0 und 400. Dann schreibt es an die Position a-Pixel nach rechts und b-Pixel nach unten das Wort "TOLL", zeigt das Ganze und wartet, bis die Maus gerückt wird.
Nun springt das Programm zu der Sprungmarke 'anfang' und wiederholt das Ganze. Um das Programm zu beenden, drückst Du auf die 'Esc' Taste. Das solltest Du Dir gut merken. Hängt Dein Programm, kannst Du es mit 'Esc' beenden.

Damit kann man schon ganz schön viel anfangen. Will man aber einen Teil eines Programms so schreiben, dass man ihn von verschiedenen Positionen aus anspringen kann, braucht man eine sog. Sub-Funktion. Dadurch lassen sich Programmteile mehrmals benutzen. Der Aufruf erfolgt jetzt über 'GOSUB', und das Programm fährt an der aktuellen Stelle wieder fort, sobald eine Sub-Funktion mit 'RETURN' oder 'ENDSUB' abgeschlossen wird.

SUBs definierst Du in GLBasic am einfachsten mit dem Menüpunkt "Projekt/Neue SUB"

Ein Beispiel:

```
// Ein GOSUB Beispiel
PRINT "ANFANG",0,0
GOSUB mitte
PRINT "ENDE",0 ,30
END

SUB mitte:
  PRINT "MITTENDRIN",0,20
```

```
  RETURN // Das springt zurück, falls nötig
ENDSUB     // Ende der Sub
```

Ausgabe:
ANFANG
MITTENDRIN
ENDE

Ein gutes Beispiel für eine Sub-Funktion ist zum Beispiel die Routine zum Laden von Spielfeldern. Sie kann so im Spiel als auch im Leveleditor aufgerufen werden. Oder die Routine, die den Spieler sterben lässt. Sie kann so von jeder möglichen Todesart aufgerufen werden...

ACHTUNG:

SUBs stehen immer am Ende des Hauptprogramms. Zwischen den Befehlen ENDSUB und SUB darf KEIN Befehl erfolgen ausser Kommentaren.
Du kannst nicht mit GOTO aus einer SUB ins Hauptprogramm springen, und nicht vom Hauptprogramm in eine SUB springen.

FOR-SCHLEIFEN:

Angenommen, wir wollen Datenfelder mit 1 befüllen. Wie lange würde das wohl zu Programmieren dauern, wenn das Feld 10 Zahlen groß ist?
Nicht lange:

```
FOR x=0 TO 9
  LET feld[x]=1
NEXT
```

Das Programm zwischen FOR und NEXT wird so lange ausgeführt, bis x>9 ist. Bei jedem Aufruf von NEXT wird x um 1 erhöht.
Kompliziert? Nochmal langsam:

```
FOR x=0 TO 9
```

Es sei die Variable x mit dem Wert 0 gefüllt. Das folgende Programm ist zu wiederholen, bis x größer als 9 ist.

```
LET feld[x]=1
```

Die Datenfeld 'feld' wird an der Stelle 'x' mit dem Wert 1 gefüllt.

```
NEXT
```

x wird um 1 erhöht. Das Programm zwischen FOR und NEXT wird wiederholt.

Nun füllen wir ein Feld der Größe 100x100 mit 1ern.

```
FOR x=0 TO 99
  FOR y=0 TO 99
    LET feld[x][y]=1
  NEXT
NEXT
```

Das geht auch rückwärts oder in anderen Sprüngen. Mit

```
STEP
```

kannst Du angeben, um wieviel Dein Zähler erhöht/vermindert werden soll.

```
FOR x=24 TO 0 STEP -5
  PRINT x, 0, (x*20)
NEXT
```

Die Befehle

```
BREAK
```

und

```
CONTINUE
```

ermöglichen ein Verlassen einer Schleife oder ein frühzeitiges Wiederholen der Schleife.

WHILE-SCHLEIFEN

Eine WHILE Schleife wird so lange ausgeführt, bis das Argument falsch wird. Wir wollen also beispielsweise eine Zufallszahl zwischen 3 und 10 haben. Nun lassen wir uns eine Zufallszahl zwischen 0 und 10 ausgeben, und wiederholen das Ganze bis die Zahl größer als 3 ist.

```
LET z=0      // z von vornherein < 3,
             // sonst wird WHILE Schleife
             // nicht ausgeführt!
WHILE z<3    // Solange z kleiner als 3,
  z=RND(10) // ist z eine Zufallszahl von 0-10
WEND         // Wiederholen.
```

Oder eine Zahl zwischen 0 und 10, aber nicht die 5!

```
LET z=5      // z gleich 5
             // sonst wird WHILE Schleife
             // nicht ausgeführt!
WHILE z=5    // Solange z gleich 5
  z=RND(10) // ist z eine Zufallszahl von 0-10
WEND         // Wiederholen.
```

Übrigens, Zufallszahlen zwischen 3 und 10 machst Du besser so:

```
z= RND(7)+3 // (0 bis 7) +3 = (3 bis 10)
```

Refs:

DIMDEL

Allgemein

Hi und willkommen zu GLBasic, Deiner neuen Programmiersprache.

Hier lernst Du innerhalb kürzester Zeit wie Du mit GLBasic Deine Spielidee selbst verwirklichen kannst. Ein Wort vorweg: GLBasic ist kein Zaubermittel. Es hat seine Stärken und auch seine Grenzen. GLBasic lächelt über Aufgaben wie Sprite-Rotationen, AlphaBlending und Zoomen. (Wenn Du nichts davon kennst, egal. Wird alles später erklärt.).Um stabile Geschwindigkeiten auf allen Rechnen zu erhalten, kann GLBasic wenn nötig auf eine einstellbare Anzahl von Bildern pro Sekunde gebremst werden.

Kapitel 1: Programmiergrundlagen.

Ein Programm kann grundsätzlich fast gar nichts. Und genau das ist das Schwierige daran. Programme können nur:
-Variablen deklarieren
-Sprünge zu anderen Programmteilen machen
-Rechenoperationen mit Variablen durchführen
-Variablen an Funktionen übergeben

1.1 Grundlegendes zu GLBasic:

GLBasic ist eine Programmiersprache. Wenn auch eine sehr einfache. Programmiersprachen haben ihre eigene Syntax (Grammatik). Solange Du sie 100% einhältst, versteht GLBasic Dich auch zu 100%!

GLBasic beendet jeden Programmbefehl mit einem ';' oder einer neuen Zeile.

Alle Befehle von GLBasic müssen IMMER GROSS geschrieben werden. Der Editor tut das von selbst, wenn er einen Befehl erkennt .

Jedes Zeichen RECHTS von einem '//' oder dem Befehl 'REM' wird als Kommentar zum Quellcode (also dem Programm) aufgefasst und hat keine Wirkung auf das Programm selbst. Es macht es nur übersichtlicher.

GLBasic benützt versteckte Bildschirmseiten für flackerfreien Grafikaufbau. Wenn Du irgend etwas mit GLBasic programmierst, siehst Du zunächst gar nichts! Erst wenn Du den Befehl 'SHOWSCREEN' aufrufst, wird Deine gerade bearbeitete Seite sichtbar, und Deine neue zu bearbeitende Seite wird gelöscht, bzw. mit dem geladenem Hintergrundbild versehen. Das ist jetzt etwas früh, trotzdem merken: Wenn Du nichts siehst, überlegen ob Du 'SHOWSCREEN' vergessen hast.

1.2 Unser erstes Programm:

Starte den Editor und wähle im Assistenten "Neues Projekt". Gib als Namen "HelloWorld" ein und klick auf OK. Nun tippe folgendes Programm in den Editor, speichere alles (Menü: Datei/Alles Speichern) und Kompiliere (erstelle) dann das Programm mit dem Sandeimer-Knopf oder Menü: Compiler/Compilieren.
Nun starte es mit dem Joypad Knopf, oder wähle Menüpunkt Compiler/Run

```
///////////////
// HELLO WORLD
///////////////
// Mein erstes Programm
PRINT "HELLO WORLD!" ,100,100
SHOWSCREEN
MOUSEWAIT
END
```

Du solltest nun einen schwarzen Bildschirm mit der Aufschrift 'HELLO WORLD!' sehen.

Wenn Du auf die Maus drückst, ist das Programm beendet. Damit hast Du den ersten Schritt in Deine neue Welt getan.

Genug, fangen wir mit dem Leichtesten an: Variablen deklarieren.

Bei GLBasic gibt es 2 Variablen-Typen: Zahlen und Wörter.
Variablen stellen wir uns als Schubladen vor. Auf der Schublade steht der Name der Variable, also der Variablenname und in der Schublade finden wir dann den Inhalt der Variablen, den Variablenwert. Beginnen wir mit Zahlen und sehen dann, was wir damit so alles anstellen können.

1.3 ZAHLEN

```
LET a=5
```

Wir rufen den Befehl 'LET' auf, und sagen wir hätten gerne eine Variable mit dem Namen 'a' die den Wert 5 beinhalten soll.
Beende den Befehl mit ';' und haben unsere erste Zeile Programm geschrieben.

Damit das Ganze etwas schneller geht, gibt GLBasic die Möglichkeit, das 'LET' auch wegzulassen. Die Zeile sähe dann so aus:

```
a=5
```

und

```
LET a=5
LET a=a+1
```

sähe dann so aus:

```
a=5
a=a+1
```

Was tun wir hier? Nun, wir sagen die Variable mit dem Namen 'a' hat den Wert 'a+1'. Da 'a' aber von vorher schon den Wert 5 hat, weisen wir der Variable 'a' in Wirklichkeit den Wert '5+1' zu, also 6.
Wir können auch andere Rechenoperationen durchführen. GLBasic kennt folgende Operatoren:
+, -, *, /; (Plus=Addition, Minus=Subtraktion, Mal=Multiplikation, Geteilt=Division)

ACHTUNG:

```
LET a= 3+4*5;
```

Wäre laut Adam Riese 3+(4*5)=23 (Punkt vor Strich). Das war bei älteren Versionen von GLBasic nicht so. GLBasic hat zum Berechnen von Termen folgende Hierarchie:
(Klammern zuerst) * / + - AND OR < > =

Es gibt bei GLBasic ein paar Sonderfunktionen, die anders arbeiten als der Rest.
SIN(), COS(), TAN(), RND(), KEY()
Diese Funktionen geben einen Wert zurück. GLBasic arbeitet übrigens mit Gleitkommazahlen. Das heißt, bei Divisionen muss die Funktion INTEGER() verwendet werden, wenn man eine Ganzzahlenoperation durchführen möchte.

Der Befehl RND() gibt eine Zufallszahl zurück. (Random)

```
LET a=RND(50) // a= Zufallszahl von 0 bis einschl. 50
```

1.4 WÖRTER (Zeichenketten)

Wörter unterscheiden sich in ihren Variablennamen durch ein angefügtes '$' von Zahlen. Zahlen lassen sich einfach in Zeichenketten umwandeln. Mehr dazu später in einem Beispiel.

```
LET a$="HALLO"
```

Wie oben geben wir der Variablen mit dem Namen 'a$' den Wert "HALLO"; Wörter die statisch, also fest im Programm verankert sind (Konstanten), müssen immer durch " " abgetrennt werden, um sie von Variablen unterscheiden zu können.
Der Befehl 'LET' ermöglicht ein paar feine Funktionen mit Wörtern.

```
LET a$="MEINE"
LET b$="LIEBLINGSZAHL"
LET c= 7
// Jetzt kommts!!
LET c$=a$ + " " + b$ + " : " + c
PRINT c$,0,20
SHOWSCREEN
```

Ausgabe:
MEINE LIEBLINGSZAHL : 7
Wir können also mit 'LET' Zahlen in Wörter umwandeln (c$=c) und Wörter, sowie statische Texte miteinander verknüpfen (+). Was für einen Sinn das hat? Wenn unser Spiel Level 5 laden soll, muss es den Dateinamen in einem Wort an die Funktion übergeben. Wir könnten also, wenn die Datei heisst: 'Level5.dat' ein Wort bilden indem wir schreiben:

```
LET levelnummer=5 // Das passiert natürlich irgendwo im Spiel...
LET level$="Level" + levelnummer + ".dat"
```

Dazu aber später in einem Beispiel mehr. Das 'LET' ist übrigens auch hier nur zur Übersichtlichkeit und kann auch einfach weggelassen werden.

GLBasic bietet Dir noch ein paar besondere Funktionen an, um mit Wörtern umzugehen.

```
INPUT name$, x, y
```

Gibt an der Stelle x, y einen blinkenden Cursor aus, mit dem Du ein Wort eingeben kannst. Nach Abschluss mit der Return-Taste wird das eingegebene Wort in die Variable 'name$' eingesetzt. Das funktioniert auch mit Zahlen
(INPUT zahl, x,y;)

```
ziel$=MID$(quelle$, start, länge)
```

Dieser Befehl ermöglicht es, aus Wörtern 'Unterwörter' herauszuschneiden. Dabei wird aus der Variablen quelle$ ein Wort, beginnend mit dem Buchstaben Nr. 'start' und der Länge 'länge' in die Variable mid$ geschrieben. Der erste Buchstabe des Wortes quelle$ hat den Startindex '0'.

Beispiel:

```
name$="Mein Auto ist blau"
m$=MID$(name$, 5, 4)
PRINT m$ , 20, 20
SHOWSCREEN
```

Ausgabe:

Auto

Du kannst damit z.B. Namenseinträge in einer Highscoreliste auf 7 Buchstaben beschränken, indem Du einfach nur die ersten 7 Buchstaben herausschneidest.

```
INPUT eingabe$, 100, 100
name$ = MID$(eingabe$, 0, 7)

wort$ = FORMAT$(numbuchst, numnachkomma, zahl)
```

Damit kann man Zahlen "schöner" in Wörter umwandeln. Behalte das im Gedächtnis, falls Du es einmal brauchen solltest.

1.5 DATENFELDER

Du hast nun einen Überblick über Variablen bekommen. Wie würdest Du aber darin ein Spielfeld speichern? Richtig! Gar nicht. Dafür benutzt Du am besten Datenfelder.
Ein Datenfeld kann man sich vorstellen wie ein Schachbrett. Es hat 8 Felder nach rechts (x) und 8 Felder nach unten (y). Ein solches Feld definieren wir in GLBasic so:

```
DIM schachbrett[8][8]
```

Dadurch wird im Rechner der Speicherplatz vorbereitet, in dem unser Schachbrett gespeichert ist, und alle Felder werden auf 0 gesetzt.
Wollen wir jetzt auf das Feld 3 nach rechts und 4 nach unten den Wert 7 schreiben, so schreiben wir:

```
LET schachbrett[2][3]=7
```

Warum 2 und nicht 3?? Nun weil Datenfelder nun einmal bei 0 beginnen. 0,1,2,3,4,5,6 sind 7 Felder. Zugegeben, das mag etwas verwirren, ist aber nach kurzer Zeit vergessen und wird automatisch richtig gemacht.
Du kannst Datenfelder auch mit Variablen anspringen.

```
LET Schachbrett[2][6]=7
LET x=2
LET y=6
LET a=schachbrett[x][y]
PRINT "Schachbrett[2][6] hat den Wert:", 100, 80
PRINT a, 100, 100
```

Ausgabe:
Schachbrett[2][6] hat den Wert:
7

Weil Du ihn vorher in das Feld geschrieben hast.

ACHTUNG:

Datenfelder dürfen nicht mehr als 4 Dimensionen haben.

```
DIM a[1][2][3][4]; // Gerade noch OK
```

Datenfelder können auch aus Wörtern und mit unterschiedlichen Dimensionen geschaffen werden.
Beispiele:

```
DIM raum[10][10][10] // Ein Zahlenraum mit x, y und z
                     // je 10 Feldern (0-9)
DIM name$[5] // 5 Wörter
LET name$[0]="Hugo"
LET name$[1]="Tim"
DIM kinobesucher$[30][10]
LET kinobesucher$[sitz][reihe]="Tom"
```

Mit dem Befehl REDIM kann man einem Feld eine neue Größe zuweisen und die bereits gespeichterten Datenelemente beibehalten.

Um ein Feld zu kopieren schreibt man:

```
DIM a[5]
b[] = a[]
```

Das erstellt ein neues Feld b[5] und kopiert die Daten von a[] hinein.

Um aus einem Feld eine Reihe herauszunehmen, kann man DIMDEL verwenden. Das ist sehr angenehm, da man dadurch einfach dynamische Anzahlen von Elementen handhaben kann. Mehr Information dazu gibt es in der Befehlsreferenz.

1.6 SPRUNGMARKEN:

Problem: Das Programm läuft einmal durch und ist dann aus.

Lösung: Wir lassen das Programm wieder zum Anfang springen.
Beispiel:

```
// Ein GOTO Beispiel
anfang:
LET a=RND(600)
LET b=RND(400)
PRINT "TOLL",a ,b
SHOWSCREEN
MOUSEWAIT
GOTO anfang
```

Was tut das Programm? Es setzt eine Sprungmarke mit dem Namen 'anfang', sagt 'a' habe einen zufälligen Wert zwischen 0 und 600, b zwischen 0 und 400. Dann schreibt es an die Position a-Pixel nach rechts und b-Pixel nach unten das Wort "TOLL", zeigt das Ganze und wartet, bis die Maus gerückt wird.
Nun springt das Programm zu der Sprungmarke 'anfang' und wiederholt das Ganze. Um das Programm zu beenden, drückst Du auf die 'Esc' Taste. Das solltest Du Dir gut merken. Hängt Dein Programm, kannst Du es mit 'Esc' beenden.

Damit kann man schon ganz schön viel anfangen. Will man aber einen Teil eines Programms so schreiben, dass man ihn von verschiedenen Positionen aus anspringen kann, braucht man eine sog. Sub-Funktion. Dadurch lassen sich Programmteile mehrmals benutzen. Der Aufruf erfolgt jetzt über 'GOSUB', und das Programm fährt an der aktuellen Stelle wieder fort, sobald eine Sub-Funktion mit 'RETURN' oder 'ENDSUB' abgeschlossen wird.

SUBs definierst Du in GLBasic am einfachsten mit dem Menüpunkt "Projekt/Neue SUB"

Ein Beispiel:

```
// Ein GOSUB Beispiel
PRINT "ANFANG",0,0
GOSUB mitte
PRINT "ENDE",0 ,30
END

SUB mitte:
  PRINT "MITTENDRIN",0,20
  RETURN // Das springt zurück, falls nötig
ENDSUB     // Ende der Sub
```

Ausgabe:
ANFANG
MITTENDRIN
ENDE

Ein gutes Beispiel für eine Sub-Funktion ist zum Beispiel die Routine zum Laden von Spielfeldern. Sie kann so im Spiel als auch im Leveleditor aufgerufen werden. Oder die Routine, die den Spieler sterben lässt. Sie kann so von jeder möglichen Todesart aufgerufen werden...

ACHTUNG:

SUBs stehen immer am Ende des Hauptprogramms. Zwischen den Befehlen ENDSUB und SUB darf KEIN Befehl erfolgen ausser Kommentaren.
Du kannst nicht mit GOTO aus einer SUB ins Hauptprogramm springen, und nicht vom Hauptprogramm in eine SUB springen.

FOR-SCHLEIFEN:

Angenommen, wir wollen Datenfelder mit 1 befüllen. Wie lange würde das wohl zu Programmieren dauern, wenn das Feld 10 Zahlen groß ist?
Nicht lange:

```
FOR x=0 TO 9
  LET feld[x]=1
NEXT
```

Das Programm zwischen FOR und NEXT wird so lange ausgeführt, bis x>9 ist. Bei jedem Aufruf von NEXT wird x um 1 erhöht.
Kompliziert? Nochmal langsam:

```
FOR x=0 TO 9
```

Es sei die Variable x mit dem Wert 0 gefüllt. Das folgende Programm ist zu wiederholen, bis x größer als 9 ist.

```
LET feld[x]=1
```

Die Datenfeld 'feld' wird an der Stelle 'x' mit dem Wert 1 gefüllt.

```
NEXT
```

x wird um 1 erhöht. Das Programm zwischen FOR und NEXT wird wiederholt.

Nun füllen wir ein Feld der Größe 100x100 mit 1ern.

```
FOR x=0 TO 99
  FOR y=0 TO 99
    LET feld[x][y]=1
  NEXT
NEXT
```

Das geht auch rückwärts oder in anderen Sprüngen. Mit

```
STEP
```

kannst Du angeben, um wieviel Dein Zähler erhöht/vermindert werden soll.

```
FOR x=24 TO 0 STEP -5
  PRINT x, 0, (x*20)
NEXT
```

Die Befehle

```
BREAK
```

und

```
CONTINUE
```

ermöglichen ein Verlassen einer Schleife oder ein frühzeitiges Wiederholen der Schleife.

WHILE-SCHLEIFEN

Eine WHILE Schleife wird so lange ausgeführt, bis das Argument falsch wird. Wir wollen also beispielsweise eine Zufallszahl zwischen 3 und 10 haben. Nun lassen wir uns eine Zufallszahl zwischen 0 und 10 ausgeben, und wiederholen das Ganze bis die Zahl größer als 3 ist.

```
LET z=0      // z von vornherein < 3,
             // sonst wird WHILE Schleife
             // nicht ausgeführt!
WHILE z<3    // Solange z kleiner als 3,
  z=RND(10)  // ist z eine Zufallszahl von 0-10
WEND         // Wiederholen.
```

Oder eine Zahl zwischen 0 und 10, aber nicht die 5!

```
LET z=5      // z gleich 5
             // sonst wird WHILE Schleife
             // nicht ausgeführt!
WHILE z=5    // Solange z gleich 5
  z=RND(10)  // ist z eine Zufallszahl von 0-10
WEND         // Wiederholen.
```

Übrigens, Zufallszahlen zwischen 3 und 10 machst Du besser so:

```
z= RND(7)+3 // (0 bis 7) +3 = (3 bis 10)
```

Refs:

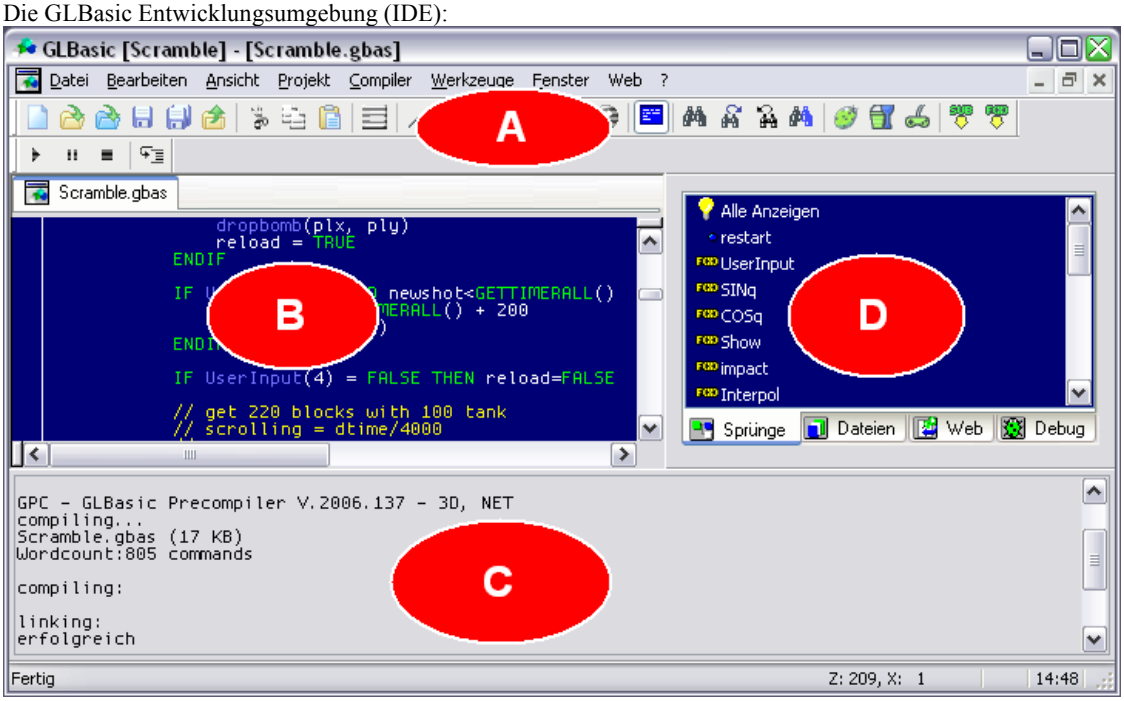
DIMDEL

Editor

Der Editor ist das Kernstück von GLBasic. Hier gibst Du Dein Programm ein und compilierst es (in Maschinensprache übersetzen). Der Editor gibt Dir Fehlermeldungen beim Compilieren aus, und springt mit dem Cursor in die Zeile an der der Fehler erkannt wurde.
Wenn Du mit dem GLBasic-Editor ein Programm schreibst, werden alle erkannten Befehle automatisch groß geschrieben und in

der Statusleiste unten wird die richtige Schreibweise (Syntax) eingeblendet. Ist ein Befehl nicht groß geschrieben, hast Du Dich vielleicht vertippt. Dann sieh' in der Hilfe nach.

Die GLBasic Entwicklungsumgebung (IDE):



Die Entwicklungsumgebung besteht aus 4 Hauptbereichen.

A) Menüs/Werkzeugleisten
B) Text Editor Fenster
C) Ausgabefenster
D) Hilfsfenster

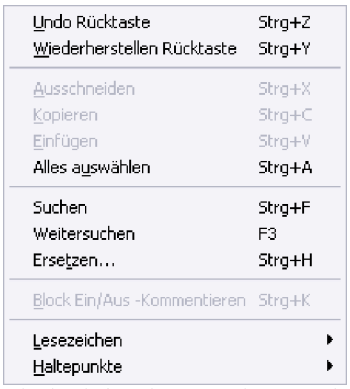
(A) Menüs:

Der "_" symbilsiert den Buchstaben, den man drücken muss umschnell diesen Befehl auszuführen. Dazu drücke man einfach die ALT-Taste und zusätzlich den Buchstaben. ALT+D öffnet z.B. das Dateimenü.

Datei:

Project - Öffnen/Speichern eines Projektes
Datei - Öffnen/Speichern einzelner Quellcodedateien
Alles speichern - Speichert alle Dateien und das Projekt
Beispiele... - öffnet den Beispielordner
Beenden - beendet den Editor

Bearbeiten:

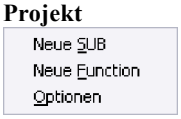


Block Ein/Auskommentieren - schreibt vor jede Zeile der Auswahl ein // oder entfernt es
Lesezeichen - für schnelle Navigation innerhalb einer Datei
Haltepunkte - Stop-Zeichen für den Debugger

Ansicht:

Verstecken / Anzeigen von verschiedenen Editorkomponenten.
Editor outfit - hier kann man Schriftart und Farben festlegen

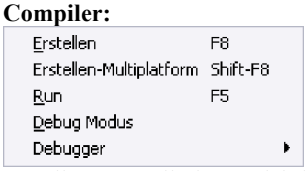
Projekt



Neue SUB - Hilft eine neue SUB einzufügen. Das ist bequem und schützt vor Fehlern
Neue Function - Hilft eine FUNCTION einzufügen
Optionen - Einstellungen wie Auflösung und platformspezifisches

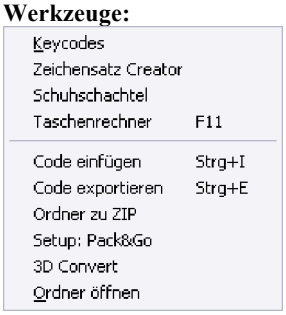
Bei den Projektoptionen gibt es die Möglichkeit verschiedene Zielplatformen anzuwählen. Die Einstellungen an der jeweiligen Plattform werden erst aktiv, wenn der Knopf "Übernehmen" gedrückt wurde.

Compiler:



Erstellen - Erstellt das ausführbare Spiel
Erstellen-Multiplatform - Erstellt für andere Platformen (Linux, ...)
Stop - Stoppt das Erstellen.
Run - Startet das erstellte Spiel
Debug Modus - Schalten den Debug-Modus ein/aus
Debugger - Steuerung des Debuggers, wenn deiser aktiv ist

Werkzeuge:



1) KeyCodes - Zeigt den KEY()-code einer gedrückten Taste an
2) Zeichensatz Creator - Erstellt aus TTF-Schriftarten BMP-Schriftarten
3) Schuhschachtel - Konvertiert einen Dateiordner in eine "Schuhschachtel" (ähnlich .zip Archiv) Siehe Befehl: SETSHOEBOX
4) Taschenrechner - ein eingebauter Rechner
5) Ordner zu ZIP - zippt einen Ordner (zum archivieren)
6) Setup: Pack&Go - erstellt ein Setup + Uninstaller
7) 3D Convert - Konvertiert 3D Modelle in GLBasic .ddd Format

8) Ordner öffnen - öffnet den Ornder, in dem die aktuelle Quelldatei ist mit dem Windows-Explorer

Web:

Internet Update - das einmal die Woche klicken um nach Updates zu suchen
Homepage - die GLBasic.com Homepage
Forum - das GLBasic Forum
Showroom - der GLBasic Showroom für eure Spiele

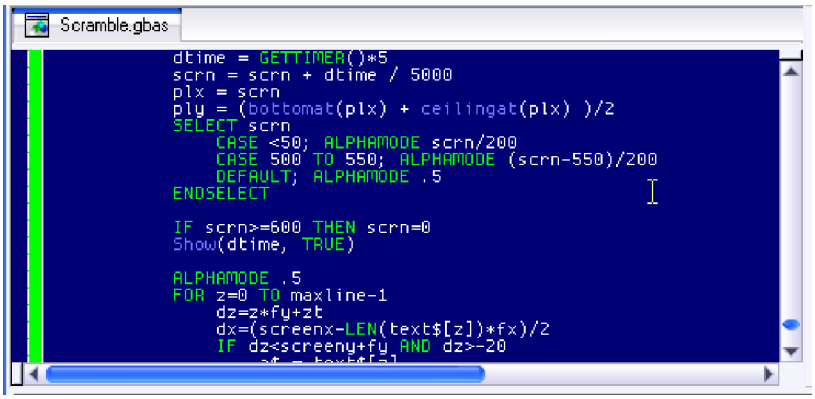
?:

Info... - über GLBasic Editor (Versionsnummer)
Hilfe - Zeigt diese Datei an
Logfile - Zeigt das aktuelle Logfile
Registrieren - hier gibt man den Lizenzcode ein

Werkzeugleiste

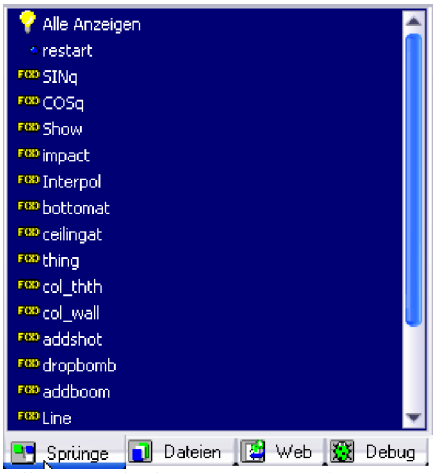
Hier sind die wichtigsten Menübefehle nochmals aufgeführt. Wenn man den MAuszeiger über einen Knopf hält, bekommt man Information dazu.

Editor-Fenster



Hier gibt man den Quellcode ein. Der Editor übernimmt automatisch die Groß-/Kleinschreibung bekannter Wörter.

Sprungmarken



Alle Sprungkarken (GOTO) sowie SUB und FUNCTION Anweisungen werden hier aufgelistet. Wenn die Zeile mit einem '@' beginnt, kann man diese nur einblenden indem man den Knopf "Alle Anzeigen" betätigt.

Ausgabefenster

```
linking:
erfolgreich
------------------------------------------------------------
*** Fertig ***
Zeit: 9.7 sek
Erstellen: 1 erfolgreich, 0 fehlgeschlagen
```

Das Ausgabefenster (unten am Bildschirm) gibt beim Compilieren Informationen über den Vorgang. Sollte ein Fehler auftreten, kann man die Zeile hier doppelklicken um im Editor an die Stelle zu springen.

Hilfe Erstellen

Mit dem Kontextmenü: "Werkzeuge/Hilfe erstellen" kann man aus der aktuellen .gbas Quellcodedatei eine Onlinehilfe erstellen lassen.
Dabei werden folgende Kommentare verwendet:
//! Ein Hinweistext (primäre Sprache)
//? Ein Hinweistext (alternative Sprache)
// \param pname - Beschreibung | Par - Beschr. alternativ
// \return Rückgabewert | Rückg. alternativ

Zusammenhängende \param und //! Hinweise werden den folgenden Function/Sub zugeordnet. Wird ein Hinweis durch eine Leerzeile oder einen normalen Kommentat (//) von einer Funktionsdefinition abgelöst, so entsteht ein eigenständiger Textblock. Einfach mal diesen Test eingeben und ausprobieren:

```
//! GLBasic's DoctorGBAS - Test
//! --------------------------
//! Some English text
//? Ein deutscher Text

//! Makes a + b
//? Macht a + b
// \param  a - Number 1 | a - Zahl 1
// \param  b - Number 2 | b - Zahl 2
// \return a plus b     | a plus b
@FUNCTION Plus: a, b
   RETURN a + b
ENDFUNCTION

//! Thanks a lot
//? Vielen Dank
```

Grafik(2D)

Allgemeines:

GLBasic benutzt 2 (Eigentlich 3) Bildschirmseiten. Sehen wir uns diese einmal genau an.

1. Primary Surface - Der Primärbildschirm.
Das ist, was der Spieler am Bildschirm sieht.

2. Back Buffer - Der hintere Puffer.
Das ist, worauf alle Grafikbefehle von GLBasic schreiben. Er ist für niemanden sichtbar. Erst wenn der Befehl: 'SHOWSCREEN' aufgerufen wird, ist der Back Buffer die neue Primary Surface. Später dazu im Detail. Vergisst Du einmal den Befehl 'SHOWSCREEN', erlebst Du Black-Screen-Entertainment, kurz Du siehst gar nichts.

3. Offscreen Surface - Der Hintergrundpuffer.
Dieser Bildschirm kann nur durch 3 Befehle angesprochen werden. 'LOADBMP', 'BLACKSCREEN' und 'USEASBMP'.
LOADBMP lädt ein BMP-Bild auf diesen unsichtbaren Puffer. USEASBMP kopiert den aktuellen BackBuffer mit allen gezeichneten Sprites auf das Offscreen Surface.
BLACKSCREEN löscht, bzw. macht das Offscreen Surface schwarz.

Wird der Befehl SHOWSCREEN aufgerufen, so tauschen Primary Surface und Back Buffer ihre Plätze. Da der Back Buffer nun unbrauchbar ist, da er den vor-vorherigen Bildschirminhalt aufweist, wird das Offscreen Surface auf ihn kopiert. Ist kein Bild geladen, wir der BackBuffer schwarz.

Ein Spiel ist daher meist so aufgebaut:

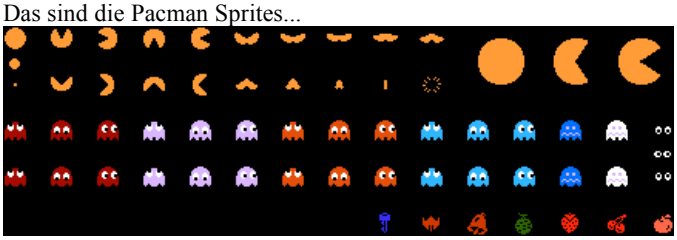
```
// Mein Spiel
start:
```

```
  LOADBMP "Hintergrund.bmp"
  // Weitere Grafiken laden als Sprites...
  // Level laden, Spielsetup (Leben=3, Zeit=100...)
maingame:
  // Tastaturabfrage
  // Spieler bewegen & Zeichnen
  // Gegner bewegen & Zeichnen
  SHOWSCREEN
GOTO maingame
```

Sprites

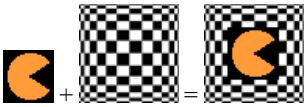
Was ist ein Sprite? Ein Sprite ist normalerweise ein kleines Grafikobjekt das sich bewegt. Pacman und die Geister sind z.B. Sprites, und das Labyrinth ist Hintergrund.

Das sind die Pacman Sprites...



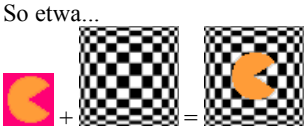
Wenn Du Ahnung von Computern hast, kennst Du vielleicht die Warheit: Der PC kann keine Sprites. Die gute Nachricht: GLBasic tut so, als gäbe es sie. Mit allem Komfort. Zunächst aber, was um alles in der Welt ist so ein 'Sprite'? Nein. Kein Erfrischungsgetränk. Sprites sind kleine Bildteilchen, die von Haus aus tolle Effekte beherrschen, die man mit normalen Grafikblöcken erst mühselig progammieren müsste.
ZSprites haben ein sog. CookieCut (dt.: Plätzchenform). Das heisst, wenn es vor einer Mauer steht, ist der Bereich knapp neben dem Sprite transparent. (Man sieht die Mauer). Ohne CookieCut (aus Grafikblöcken) hätte jede Figur ein schwarzes, viereckiges Rechteck um sich herum.

Etwa so...



Nun, woher weiß aber jetzt GLBasic, welche Stelle eines Bildes, das als Sprite geladen wird, transparent sein soll? Es gäbe tausend Möglichkeiten. GLBasic benutzt folgende:
ALLE PIXEL DER FARBE rot:255, grün:0, blau:128 (Hex: FF 00 80), also ein ätzendes Pink, werden als transparent behandelt.

So etwa...



ACHTUNG:

Alle Grafiken, die Du in GLBasic lädst, haben das Windows-Bitmap Format, unkomprimiert, mit 8 oder 24Bit Farbtiefe. Du kannst keine 16Farb-Grafiken laden. Diese müsstest Du zuerst mit MSPaint laden und unter 8Bit Farbtiefe speichern.
PNG Dateien können auch geladen werden. Debei wird entweder RGB(255,0,128) als Transparent oder der Alpha-Kanal falls vorhanden.

Dein erstes Sprite:

```
////////////////
// SPRITES
////////////////
// SPRITE aus BMP Datei laden und Nummer 0 zuweisen
LOADSPRITE "Bubble.bmp" ,0
// SPRITE Nummer 0 anzeigen an Stelle x=300, y=150;
SPRITE 0, 300, 150
SHOWSCREEN
END
```

Wie Du siehst, mit nur 3 Befehlen hat man ein sichtbares Ergebnis. Das Programm, dass ausgeführt wird, um diese paar Zeilen zu ermöglichen ist mehrere Bildschirmseiten lang. Und genau darin liegt Dein Vorteil gegenüber Programmierern von Hochsprachen (C++, Assembler, Pascal...)
Der Befehl LOADSPRITE erklärt sich selbst. Er lädt eine BMP Datei, die dann als SPRITE benutzt werden kann. Um ein SPRITE ansprechen zu können, geben wir ihm eine Nummer (Hier: 0)

Sieh' Dir ruhig mal die BMP Datei an. Du wirst feststellen, dass um die Blase herum ein ätzendes Pink ist -> Transparente Farbe.

Um das Ganze besser zu sehen, laden wir jetzt ein Bild. bevor wir das SPRITE anzeigen. Dazu fügen wir folgenede am Anfang des Programmes ein:

```
LOADBMP "back.bmp"
```

Der Befehl 'SPRITE' hat 3 Parameter. Der Erste ist die Nummer des Sprites. 2 und 3 sind die x und y Position der linken oberen Ecke des Sprites. Wie stark das Sprite mit anderen Grafiken vermischt wird, wird im Befehl ALPHAMODE angegenben. Richtig GLBasic unterstützt Alpha-Blending! Noch nie gehört? Kein Problem.

AlphaBlending ist, wenn man zwei Bilder, z.B. Hintergrund und Sprite, miteinander verschmilzt. Dabei unterstützt GLBasic 2 Möglichkeiten.

ADDITIVES ALPHABLENDING

ein Wert zwischen 0..00001 und 1.0 mischt beide Farbwerte, in dem es die Rot, Grün und Blau-Werte jedes einzelnen Pixels einfach zusammenzählt und aufpasst, dass diese nicht über 255 (FF) kommen. Klingt kompliziert, wird aber alles von GLBasic übernommen. Ein Wert über +1 ist gut für Effekte wie Feuer, Explosionen, Glasscheiben..

INTERPOLIERTES ALPHABLENDING

ein Wert zwischen -0.00001 und -1.0 verschmilzt beide Objekte, indem die Rot, Grün und Blau-Werte zusammengezählt und durch 2 geteilt werden. Effekt? Man kann durch dunkle Formen sehr leicht Schatten erzeigen.

Ein Tip: Nimm Dir ein Stündchen Zeit, zeichne ein paar Formen in verschiedenen Farben und probiere AlphaBlending auf einem bunten Bild aus. (Auch mit Schwarz-Weiß Bildern lassen sich durch AlphaBlending mit bunten Sprites tolle Effekte erzielen).

SPRITE SPEZIALEFFEKTE

Ein Sprite auf eine beliebige Größe zoomen kannst Du mit:

```
ZOOMSPRITE id, xstart, ystart, x%, y%
```

oder:

```
STRETCHSPRITE id, xstart, ystart, xbreite, ybreite
```

Du drehst ein Sprite um seinen Mittelpukt mit:

```
ROTOSPRITE id, x, y, winkel
```

Dabei sind x und y die Koordinaten des Sprites, wenn es nicht gedreht wird. Also ist der Befehl:

```
ROTOSPRITE 1,50,70, 0
```

identisch mit:

```
SPRITE 1,50,70
```

Beides kombiniert gibt:

```
ROTOZOOMSPRITE id, x, y, winkel, groesse
```

groesse gibt die Größe des Sprites bezogen auf die Originalgröße in Prozent an. 0.50 ist also halb so groß wie das Original, 2.0 das Doppelte.

GRAFIK SONDERFUNKTIONEN

Ein besonderes Schmankerl bietet GLBasic zum Überblenden von Bildern. Mit dieser Funktion kannst Du den Inhalt des BackBuffers in ein BMP-Bild überblenden. Ist das Bild schwarz, wird der BackBuffer-Inhalt sanft ausgeblendet (fading), also immer dunkler bis alles schwarz ist.

```
BLENDSCREEN bmpdatei$
```

Und es kommt noch besser: GLBasic kann alle verfügbaren Videoformate wiedergeben. (Also alles, was sich im Windows

MediaPlayer(2) abspielen lässt.)
Zum Beispiel .avi, .mpg... Dateien. Alles supereinfach mit:

```
PLAYMOVIE filename$
```

EINFACHE ZEICHNUGEN

Manchmal ist es praktischer eine Linie zu zeichnen statt ein Sprite zu gebrauchen. Daher hat GLBasic auch Zeichenbefehle für Linien, ausgefüllte Rechtecke und einzelne Punkte.

Siehe dazu:

```
SETPIXEL - DRAWLINE - FILLRECT
```

Refs:

LOADBMP, SETPIXEL, SPRITE

Input

TASTATUR:

GLBasic bietet 2 Möglichkeiten der Tastatureingabe. Wörter und Abfrage von Einzeltasten.

```
INPUT name$, x, y
INPUT zahl, x, y
```

Wörter hast Du bereits kennengelernt. Tastenabfragen sind auch ganz einfach.

```
IF KEY (code) = 1; PRINT "TASTE GEDRÜCKT", 20, 20
```

Die Codes der einzelnen Tasten erhält man bequem über das Programm "Keycodes", das sich bei den Tools befindet.

MAUS:

```
MOUSESTATE mx, my, ma, mb
```

Schreibt die aktuelle Mausposition (x,y) in die Variablen mx und my. Die linke Maustaste wir in ma geschrieben und die Rechte in mb. Willst Du die Geschwindigkeit wissen, solltest Du den Befehl

```
MOUSEAXIS
```

ansehen. Damit kann man einzelne Tasten, das Mausrad und die Geschwindigkeit erfassen.

Wenn man also wissen möchte, ob sich das Mausrad bewegt, schreibt man:

```
WHILE TRUE
v=MOUSEAXIS(2)
SELECT v
 CASE 0
  PRINT "Mausrad ist still", 20, 20
 CASE 1
  PRINT "Mausrad geht HOCH", 20, 20
 CASE -1
  PRINT "Mausrad geht RUNTER", 20, 20
ENDSELECT
WEND
```

JOYSTICK:

```
JOYSTATE jx, jy, buttona, buttonb
```

Liefert in jx und jy die Joystickrichtung. Siehe auch JOYSTATE Befehlsreferenz. Sonst wie MOUSESTATE. Für registrierte Benutzer gibt es auch die Möglichkeit mehrere Joysticks abzufragen. Siehe dazu ein Beispiel bei GETJOY....

DATEIEN (einfach):

```
GETFILE file$, line, linedata$
GETFILE file$, line, linedata
```

Liest aus der Datei 'file$' die Zeile Nummer 'line' (0=Erste Zeile!!) und schreibt sie in die Variable linedata$ (linedata). Ist die Datei nicht vorhanden, wird linedata$ das Wort "NO_FILE" enthalten. Ist in der Zeile kein Wort/Zahl ist linedata$ "NO_DATA". Linedata (als Zahl) ist dann 0. Am Besten du liest immer ein Wort aus, und wandelst das im sinnvollen Fall in eine Zahl um. So kannst Du auf Fehler überprüfen.

```
PUTFILE file$, line, linedata$
PUTFILE file$, line, linedata
```

Schreibt in die Datei 'file$' das Wort 'linedata$' (die Zahl linedata) in die Zeile 'line'. Ist die Datei nicht vorhanden, wird sie erstellt. Hat die Datei weniger Zeilen als 'line', werden die fehlenden Zeilen mit dem Wort "NO_DATA" versehen. Solltest Du einmal eine Zeile lesen, die das zurückgibt, weisst Du, dass Du Dich in der Zeilennummer geirrt hast.

DATEIEN (flexibel):

Man öffnet eine Datei mit OPENFILE, benützt dann READ... oder WRITE... Funktionen und schließt die Datei anschließend mit CLOSEFILE.
Mit dem READ/WRITE Funktionen kann man auch binäre Dateien verwenden, ist schneller als mit GETFILE/PUTFILE und hat keine Begrenzung der Dateigröße.

INI-Dateien

```
INIOPEN file$
value$ = INIGET$(section$, key$)
INIPUT(section$, key$, value$);
```

Vie Komfortabler als GETFILE/PUTFILE sind die seit Version 3.000 enthaltenen Funktionen um INI Dateien zu lesen oder zu schreiben. Damit ist man von der Größe her nicht begrenzt und kann viel schneller und komfortabler auf Daten zugreifen.

Eine INI Datei ist eine normale Textdatei, die sich in Sektionen von Schlüsseln und Werten trennt. Schlüssel sind wie Variablen. Man kann über deren Namen (Zahl oder Wort) auf Daten zugreifen.

Hier ein Beispiel für eine INI Datei...

```
[Sound]
MusicVolume=60
SoundVolume=100
SpeachVolume=80

[Video]
ScreenResX=800
ScreenResX=600
```

[Sound] und [Video] sind Sektionen. MusicVolume usw. sind Schlüssel. Um den Wert von Video/ScreenResX zu lesen schreibt man:

```
ScreenWidth = INIGET$("VIDEO", "ScreenResX")
```

Ein Ausführliches Beispiel dazu liefert die Befehlsreferenz zu INIOPEN.

Mit diesen Befehlen ist man an keine Dateigröße gebunden und kan viel flexibler und schneller auf Daten zugreifen.

Refs:

GETFILE, GETJOY...(), INIOPEN, JOYSTATE, KEY(), MOUSEAXIS, MOUSESTATE, OPENFILE()

Sound+Musik

SOUND

Dies wird ein recht kurzes Kapitel. Du behandelst Sounds wie Sprites. Es gibt 3 Befehle für das Abspielen von Sounds.

```
LOADSOUND name$, id, nPuffer
```

Dieser Befehl lädt eine Sound-Datei in den Speicher und weist ihr eine Nummer zu, um sie damit später abzuspielen. Es wird nur das .wav-PCM format unterstützt.
nPuffer gibt an, wie oft der Sound gleichzeitig abgespielt werden kann, ohne 'abgeschnitten' zu werden. (1-4)

```
PLAYSOUND (n, pan, vol)
```

Damit wird ein geladener Sound abgespielt. Der Befehl erwartet 2 Argumente:
1. Die Nummer des abzuspielenden Sounds.
2. Der Stereo-Wert in %. -1.0=links, 0=mittig, +1.0=rechts
3. Die Lautstärke (0.0=aus, 1.0=laut)

```
HUSH
```

Stopt alle Sounds sofort.

Du kannst mehrere Sounds gleichzeitig abspielen. Die Soundkarte mixt die Daten dann in Echtzeit.

MUSIK

Du kannst Hintergrundmusiken abspielen, in den Formaten: mp3, wav, midi...
Verwende dazu folgende Befehle:

Abspielen:

```
PLAYMUSIC datei$
```

Stoppen:

```
STOPMUSIC
```

Refs:

PLAYMUSIC, PLAYSOUND(), SOUNDPLAYING()

Netzwerk

Netzwerk Spiele

Diese Sektion ist für die Programmierung von Multiplayer Netzwerkspielen mit GLBasic gedacht.

Allgemeines

Netzwerkspiele sind ein echter Spass. Aber Vorsicht, sie zu programmieren ist nicht einfach. Die von GLBasic zur Verfügung gestellten Befehle sind zwar sehr einfach, sie zu benutzen bedeutet etwas Übung. Bevor Du ein Multiplayerspiel schreibst, überlege lieber, ob man das Ganze nicht auch zu zweit an einem Computer spielen kann. Nicht jeder hat ein Netzwerk zuhause und Multiplayer-Spiele an einem Computer sind wesentlich leichter zu programmieren.

Das folgende Beispiel ist eine Client/Server Applikation, aber GLBasic kann noch viel mehr. Ein Beispiel der Flexibilität ist z.B. dass man eine Bestenliste aus dem Internet lädt und verwaltet (dazu das Scramble Beispiel auf der GLBasic Webseite ansehen).

Initialisieren

Um ein Netzerkspiel zu starten, gibt es immer einen Host und mehrere Clients, die dem Spiel beiwohnen (joinen). Um ein Spiel zu 'hosten', benutze folgenden Befehl:

```
ok=NETHOSTGAME(prt, sn$, p1$, p2$)
```

Wobei 'prt' das Protokoll ist, das verwendet werden soll.
0=TCP/IP (Internet, oder lokales Netzwerk)
1=IPX (Lokales Netwerk)
sn$ ist der Name der Sitzung (z.B. "Mein_Spiel")
p1$ und p2$ sind Verbindungsparameter. Bei NETHOSTGAME müssen Sie leer sein ("").

Wenn das geklappt hat, ist ok=TRUE, sonst FALSE.

Um einer aktiven Sitzung als Teilnehmer (Client) beizutreten, verwende den Befehl:

```
ok=NETJOINGAME(prt, sn$, p1$, p2$)
```

Die Parameter sind wie oben. Jedoch braucht NETJOINGAME natürlich eine Information, wo denn nun der Host ist. Bei TCP/IP kann in p1$ die IP-Adresse des Hosts angegeben werden. Dabei ist leer oder "0.0.0.0" irgend ein Computer im lokalen Netzwerk. Alle weiteren Adressen können entweder so:"149.232.23.2" oder so "www.Dream-D-Sign.de" angegeben werden. Für IPX muss p1$ und p2$ leer sein.
Hat das geklappt, ist ok=TRUE, sonst FALSE.

Spieler bitte

Jetzt ist es an der Zeit, einen Spieler ins Netz zu bringen. Jedes Programm (jeder User) kann mehrere Spieler (Player) verwalten. Darum hat jeder Player einen Namen und eine eindeutige Player-ID. Man erstellt einen Spieler mit:

```
id$=NETCREATEPLAYER$( name$~
```

Dabei ist name$ ein alphanumerischer Name ("Max"). id$ enthält nun eine globale ID für diesen Player. Ein Spieler wird aus dem Spiel genommen, wenn das Programm beendet wird, oder mit dem Befehl
NETDESTROYPLAYER id$.

Der Postbote kommt

Ein erstellter Player kann nun Nachrichten verschicken und empfangen. Nachrichten die verschickt werden, erreichen alle Player in der Sitzung, ausser dem Player, der sie abgeschickt hat. Ein Player kann nur Nachrichten abrufen, die auch an ihn geschickt wurden. Es gibt dafür folgende Befehle:

```
msg$=NETGETMSG$( id$ )
ok = NETSENDMSG( id$, msg$~
```

Wobei id$ immer die ID des Players ist, der die Nachricht schickt / empfangen soll.
msg$ ist die gesendete / empfangene Nachricht. Sie ist leer ("") wenn keine Nachricht für diesen Player verfügbar ist.
ok=TRUE/FALSE je nach dem, ob die Nachricht geschickt werden konnte oder nicht.

Seid ihr alle da?

Um festzustellen, wer alles an einer Session angemeldet ist, was für eine ID$ und was für einen Namen die Spieler haben, sind in GLBasic ein paar nützliche Funktionen implementiert worden.

```
NumPlayers=NETNUMPLAYERS()
playerID$[i]=NETGETPLAYERID$(i)
playername$[i]=NETPLAYERNAME$(playerID$[i])
```

Der Befehl NETPLAYERNAME$ liefert "NO_DATA" wenn ein Fehler vorliegt.
Man benutzt die Befehle wie im folgenden Code:

```
PRINT "Angemedete Spieler:", 100, 50
   NumPlayers=NETNUMPLAYERS()
   FOR i=0 TO NumPlayers
     playerID$[i]=NETGETPLAYERID$(i)
     playername$[i]=NETPLAYERNAME$(playerID$[i])
     PRINT playername$[i] + " (ID:" + playerID$[i]+") " + i, 100, i*20+100
   NEXT
```

Refs:

NET...

Grafik(3D)

Allgemein

3D Grafik ist ein komplexes Thema. In GLBasic haben wir all unser Wissen in einfache zu verwendende Befehle gesteckt. Die Befehle unterscheiden sich in Einfachheit kaum von den 2D Befehlen, die hier als bekannt vorausgesetzt werden. Wer kein 2D Spiel geschrieben hat, sollte sich mit 3D nicht bemühen. Man lernt ja auch erst laufen und dann tanzen.

GLBasic benutzt ein Rechte-Hand-System, bei dem Y immer positiv nach oben gerichtet ist. Alle 3D Befehle beginnen mit X_..., so dass sie in der Hilfe schnell gefunden werden können. Die 3D Befehle sind nur in der Ausbaustufe GLBasic 3D möglich. Texturen werden immer in der Größe 2^n geladen und auch verwendet. Manche Modelle aus dem Internet haben andere Größen der Texturen mitgeliefert. Diese müssen dann gestreckt werden auf 64, 128, 256, 512 oder 1024 Pixel im Quadrat. Andernfalls

wird das Objekt hässliche Texturfehler aufweisen.

Grundlegend ist GLBasic 3D folgendermaßen aufgebaut:

Umschalten in 3D Modus:

```
X_MAKE3D znahe, zfern
```

Dabei ist znahe und zfern der Bereich, in dessen Abstand zur Kamera sichtbare Objekte gezeichnet werden sollen. Die Werte 1 und 1000 liefern gute Ergebnisse. Größere Abstände (1, 32000) liefern u.U. hässliche Ergebnisse in der Tiefenprüfung beim Zeichnen.

Kamera

```
X_CAMERA x, y, z, px, py, pz
```

Setzt die Kamera an die Position x, y, z und richtet sie so aus, dass sie auf den Punkt px, py, pz zeigt. Oben ist dabei immer entlang der positiven Y-Achse.

Licht

```
X_AMBIENT_LT num, col
```

Schaltet ein ambientes (von allen Seiten gleichmäßig stark scheinendes) Licht ein. col gibt die Farbe des Lichtes an. (Siehe Befehl: RGB).
num ist dabei die Numer der Lichtquelle. Es können maximal 8 Lichtquellen (0-7) benutzt werden, da die Hardware nicht mehr unterstützt. Sind mehr Lichtquellen in einer Szene, empfiehlt es sich, die 8 nahestehendsten zu verwenden. In professionellen Spielen sind selten mehr als 2 Lichtquellen aktiv, da Lichter enorm viel Rechenzeit beanspruchen.

```
X_SPOT_LT num, col, x, y, z, dirx, diry, dirz, cutoff
```

Erstellt ein Spot-Licht an der Position x, y, z, das in die Richtung dirx, diry, dirz scheint. Der Öffnungswinkel der Blende wird bei cutoff angegeben. Das Licht wirft (auch wenn auf den ersten Blich etwas unlogisch) keinen Schatten! Es wird lediglich die Helligkeit des Lichtes anhand Poisition und Richtung zum Objekt beeinflusst.

Transformation
Kamera und Licht sind bereit. Jetzt fehlt noch das Wichtigste: Objekte.

```
X_MOVEMENT x, y, z
X_SCALING fx, fy, fz
X_ROTATION phi1, dx, dy, dz
X_ROTATION phi2, dx, dy, dz
```

X_MOVEMENT schiebt die aktuelle Zeichenposition auf die Koordinaten x, y, z. Der lokale Nullpunkt eines Objektes wird damit um diesen Betrag beim Zeichnen verschoben.
X_SCALING skaliert das Objekt entlang der lokalen Achsen.
X_ROTATION fügt dem Objekt eine Rotation um den Richtungvektor dx, dy, dz mit dem Winkel phi zu. Es können mehrere Rotationen hintereinander gemacht werden. X_MOVEMENT oder X_SCALING setzen alle aktuellen Rotationen wieder zurück. Sie müssen daher zuvor aufgerufen werden.

Objekte
Der einfachste Weg zu einem Objekt ist die Datei. Man erstellt mit einem 3D-Programm ein Objekt und exportiert es als Quake-MD2 Modell, oder als 3D Studio .3ds Objekt. Nun lädt man es in das 3DConvert-Tool und erhält ein komprimiertes .ddd Objekt, das man so laden kann:

```
X_LOADOBJ datei$, num
```

Es besteht auch die Möglichkeit, Objekte zur Laufzeit zu generieren. Dazu verwendet man folgende Befehle:

```
X_OBJSTART num
X_OBJADDVERTEX x1, y1, z1, tx1, ty1, col
X_OBJADDVERTEX x2, y2, z2, tx2, ty2, col
X_OBJADDVERTEX x3, y3, z3, tx3, ty3, col
X_ENDOBJ
```

num gibt dabei die Nummer des Objektes an. Unter dieser Nummer kann ein Objekt später gezeichnet werden. Bestehende Nummern können überschrieben werden.

X_OBJADDVERTEX fügt dem Objekt einen Vektor hinzu. tx, ty geben dabei die Textur-Koodinaten an. 0=links/oben, 1=rechts/unten.
Die ersten 3 Vektoren bilden ein Dreieck. Jeder weitere Vektor bildet mit den beiden zuletzt eingegebenen Vektoren ein Dreieck. Soll dies nicht geschehen, kann man den 'Stift absetzen' und wieder von neuem 3 Vektoren eingeben die dann ein Dreieck aufspannen und dann weitere Punkte hinzufügen mit dem Befehl:

```
X_OBJNEWGROUP
```

Die Lichtnormalenvektoren, die nötig sind damit der Grafikprozessor weiß, wo vorne und hinten bzw. wie hell ein Licht ein Dreieck beleuchtet, werden automatisch berechnet und an die Oberfläche des Objekts angeglichen. Diese Technik wäre sonst sehr aufwändig und spart bei GLBasic sehr viel Programmierarbeit.
Wenn man nur die Vorder- bzw. Rückseite eines Objekts zeichnen möchte, kann man den Befehl:

```
X_CULLMODE cmode#
```

verwenden.

Nun muss das geladene oder erstelte Objekt noch gezeichnet werden. Das geschieht mit:

```
X_DRAWOBJ num, stufe
```

Wobei stufe die Animationsstufe ist. Bei Benutzerdefinierten Objekten ist sie immer '0' - also die erste Stufe.
Animationen spielt man wieder mit:

```
X_DRAWANIM num, stufe, von, bis, interpol, volle_anim
```

von und bis geben die Stufen an, zwischen denen interpoliert werden soll. 'interpol' gibt den Fortschritt an (0 bis 1) und 'volle_anim' gibt an, ob die Zwischenstufen auch berücksichtigt werden sollen.

Um dem Objekt eine Textur zuzuweisen, die vorher mit LOADSPRITE geladen wurde, benutzt man den Befehl:

```
X_SETTEXTURE num, num_bump
```

Wenn num_bump ungleich -1, ist die Textur eine Bump-Mapping Normalen-Map und musste vorher mit LOADBUMPTEXTURE geladen werden.

Beenden
Im 3D Modus können keine 2D Befehle ausgeführt werden. Darum muss vorher in den 2D Modus umgeschaltet werden:

```
X_MAKE2D
```

Spezialeffekte
Die Spezieleffekte "Cel Shading", "Bump Mapping" und "Stencil Shadows" können über den Befehl X_SPOT_LT angesprochen werden.
Mit X_SPHEREMAPPING kann man Metallische Reflektionen programmieren.

Refs:

X_SPOT_LT

Types

Von Variablen in Types

Types sind eine Methode um Daten zu strukturieren. Warum sollte man Types verwenden, um Daten zu strukturieren? Types können die Lesbarkeit des Codes drastisch erhöhen, und werden damit leichter zu Debuggen. Gut gemachte Types können den Code erheblich verkürzen, da man mit Blöcken von Variablen arbeitet, statt einzelne Variablen zu verwenden.

Um einen Typ zu definieren scheibt man:

```
TYPE name_des_tys
  // Typ Mitglieder
```

```
ENDTYPE
```

Der TYPE ist quasi eine Blaupause, wie die Variable später auszusehen hat.

Jetzt, da man einen Typ hat, kann man ihm Mitglieder (member) hinzufügen. Das sind Variablen, auf die mit einer Instanz eines Typen zugegriffen werden kann.
Man kann entweder Zahlen oder Nummern-Variablen als Member verwenden und diesen Voreingestellte Werte zuweisen. Wenn man keine Werte zuweist haben Nummern den Wert 0, und Wörter den Wert "".

```
TYPE BUCH
   // Zahl, hat den Wert 0
    AnzahlDerSeiten

   // Zahl mit voreingestelltem Wert
   CoverColor = RGB(128,64,0)

   // Wort mit Wert = ""
   Titel$

   // Wort mit zugewiesenem Wert
   Autor$ = "John Doe"
ENDTYPE
```

Man kann auch Felder in einen Typ packen. Diese Felder sind anfangs nicht initialisiert (DIM name[0]). Man kann jedoch eine Standardgröße vorgenben, so dass beim Erstellen einer Instanz des Typen (siehe später) das Feld bereits dimensioniert ist. Selbstverständlich kann man trotzdem ein REDIM auf das Feld anwenden.

```
TYPE KINO
   // ein Feld, kein DIM
   verkaufte_sitze[]

   // ein Feld, dimensioniert mit DIM[32][32]
   freie_sitze[32][32]

ENDTYPE
```

Um eine Instanz eines Typen zu erstellen schreibt man einfach:

```
GLOBAL Bibel AS BUCH
LOCAL MobyDick AS BUCH
```

und greift auf die Mitglieder der Bibel (vom Typ BUCH) so zu:

```
Bibel.AnzahlDerSeiten=4500
Bibel.Titel$ = "The Holy Bibel"
Bibel.Autor$ = "King James"
```

Man kann auch Typen (per Referenz) an Funktionen übergeben. "per Referenz" heisst, dass ein ändern des Wertes innerhalb der Funktion auch den Wert ausserhalb ändert. Wie das auch mit Feldern der Fall ist.

```
// <...> BUCH Typ Definition
LOCAL Bibel AS BUCH, Another AS BUCH
MakeBibel(Bibel)
Another = Bibel // copy 'Bibel' to 'Another'

FUNCTION MakeBibel: any AS BUCH
   any.AnzahlDerSeiten=4500
   any.Titel$ = "The Holy Bibel"
   any.Autor$ = "King James"
ENDFUNTCTION
```

Um die Sache noch zu versüßen, kann man Typen auch verschachteln:

```
TYPE ARBEITER:
   Name$
   Annual_Wage
ENDTYPE

TYPE BUECHEREI:
   BUCHs[] AS BUCH
   Employees[12] AS ARBEITER
ENDTYPE
```

Und als Rückgabewert einer Funktion verwenden:

```
LOCAL Bibel AS BUCH
```

```
Bibel = MakeBUCH("The Holy Bibel", 4500)

FUNCTION MakeBUCH AS BUCH: name$, nPages
   LOCAL temp AS BUCH
   temp.Titel$ = name$
   temp.AnzahlDerSeiten = nPages
   RETURN temp
ENDFUNCTION
```

Vergisst man von solch einer Funktion das RETURN, wird eine Instanz des Rückgabetyps (hier BUCH) zurückgegeben, dessen Werte mit den Standardwerten initialisiert sind.

Hat man ein Feld von TYPEs, kann man mit DIMDEL oder DELETE und DIMPUSH Elemente herauslöschen und neue unten hinzufügen.

Möchte man durch alle Elemente in einem TYPE Feld durchlaufen, kann man FOREACH verwenden.

Refs:

DELETE, DIMDEL, DIMPUSH, FOREACH

Das PONG Spiel

PONG

Was ist PONG

Jetzt, da die Grundlagen klar sind, muss das Gelernte gefestigt und geübt werden. Das Spiel heißt: "PONG". Es ist nicht nur eines Deiner ersten Spiele mit GLBasic, sondern auch das erste Computerspiel der Welt. (1972 von Ralph H. Baer)
Wenn Du PONG nicht kennst, sieh hier nach:
www.pong-story.com

Neues Projet

Nach dem Start von GLBasic den Kopf "Neues Projekt" auswählen, und einen Namen + Pfad angeben. z.B.
..\GLBasic\Projects\Pong

Variablen

Nachdem das Projekt angelegt ist, können wir schon mit der Programmierarbeit anfangen.
Wir brauchen Variablen für die Schlägerpositionen x und y, sowie die bisher erzielten Punkte. Für all diese Variaben legen wir je ein Feld an, damit später 3,4 oder mehr Spieler einfach eingebaut werden könnten. Auch kann man so schnell den Code für Schlägerbewegung und Punktestand in einer Schleife wiederverwenden. Später mehr dazu...

Unser Code sieht jetzt so aus:

```
// Pong

DIM bat_y[2]
DIM bat_x[2]
DIM score[2]
```

Nun müssen wir die Spieler initialisieren. Dazu fügen wir eine SUB hinzu mit dem Menüpunkt: "Projekt/Neue SUB". Name: "Init"

In die SUB schreiben wir nun Quellcode, damit der Hintergrund des Spielfelds gezeichnet wird und verwenden den BackBuffer als aktuelles Hintergrundbild (man kann stattdessen natürlich auch einfach ein Bild laden).

Danach setzen wir die Position der Schläger auf die Randkoordinaten.

```
// ------------------------------------------------------------ //
// -=#  INIT  #=-
// ------------------------------------------------------------ //
SUB Init:
   // Spielfeld zeichnen, als Hintergrund nutzen
   BLACKSCREEN
   FILLRECT 0,   0, 639, 15,  RGB(255, 255, 255)
   FILLRECT 0, 464, 639, 479, RGB(255, 255, 255)
   FILLRECT 312, 0, 327, 479, RGB(255, 255, 255)
   USEASBMP

   // Schläger 0 setzten
   bat_y[0]=240; bat_y[1]=240
```

```
   // Schläger 1 setzten
   bat_x[0]=16; bat_x[1]=600
ENDSUB // INIT
```

Ball zurücksetzen

Zu Begin jedes Spiels muss der Ball wieder in die Mitte platziert werden, und seine Geschwindigkeit wieder auf 1 gesetzt werden. Dazu fügen wir eine neue SUB ein, wie vorher beschrieben, und nennen Sie: ResetBall

```
// ------------------------------------------------------------ //
// -=#  RESETBALL  #=-
// ------------------------------------------------------------ //
SUB ResetBall:
   ball_x=320
   ball_y=240

   IF ball_sx<0 // Wenn Ball nach links, dann jetzt nach rechts und vice versa
      ball_sx=1
   ELSE
      ball_sx=-1
   ENDIF

   ball_sy=1
ENDSUB // RESETBALL
```

Diese Funktion rufen wir auch in Init auf. Dazu in die SUB Init

```
GOSUB ResetBall
```

einfügen.

Anzeigen

Jetzt haben wir schon einiges programmiert, aber noch immer nichts gesehen. Es wird Zeit, dass wir das Spielfeld und den Ball anzeigen. Also, neue SUB mit Namen ShowAll:

```
// ------------------------------------------------------------ //
// -=#  SHOWALL  #=-
// ------------------------------------------------------------ //
SUB ShowAll:
   // Die Schläger anzeigen
   FOR num=0 TO 1
      FILLRECT bat_x[num], bat_y[num], bat_x[num]+16, bat_y[num]+64, RGB(255, 255, 255)
      PRINT score[num], num*320 + 32, 16
   NEXT
   // Der Ball
   FILLRECT ball_x, ball_y, ball_x+16, ball_y+16, RGB(255, 255, 255)
   SHOWSCREEN
ENDSUB // SHOWALL
```

Hauptschleife

Jetzt brauchen wir eine Hauptschleife, die immer wieder die SUBs aufruft. Dazu die folgenden Zeilen über die erste SUB schreiben. Vor der ersten SUB oder FUNCTION steht das Hauptprogramm. Dieses wird beim Programmstart ausgeführt.

```
// Voreinstellungen
GOSUB Init

// Hauptschleife
main:
→GOSUB ShowAll
GOTO main
```

Achtung! Der Code würde das Programm dazu bringen, dass es nicht mehr reagiert, wenn wir nicht in ShowAll den Befehl

```
SHOWSCREEN
```

aufrufen würden.

Der erste Start

Drücke jetzt den Knopf zum Kompilieren drücken und starte dann das Programm.
(F8, dann F5)

Bewegung!

Jetzt muss nur noch Bewegung in das Spiel. Dazu legen wir eine SUB mit dem Namen MoveAll an und fügen den Aufruf vor

```
GOSUB ShowAll
```

mit

```
GOSUB MoveAll
```

ein.

```
// ------------------------------------------------------------ //
// -=#  MOVEALL  #=-
// ------------------------------------------------------------ //
SUB MoveAll:
→// Schläger
→FOR num=0 TO 1
→→// Spieler 1: Tasten: A, Y
→→IF KEY(30) THEN bat_y[0]=bat_y[0]-2
→→IF KEY(44) THEN bat_y[0]=bat_y[0]+2
→→// Tasten: auf, ab
→→IF KEY(200) THEN bat_y[1]=bat_y[1]-2
→→IF KEY(208) THEN bat_y[1]=bat_y[1]+2
→
→→// Schläger am oberen/unteren Rand?→→→IF bat_y[num]<0   THEN bat_y[num]=0
→→IF bat_y[num]>416 THEN bat_y[num]=416
→NEXT
```

Die Codes für den KEY()-Befehl findest Du unter dem Menüpunkt "Werkzeuge/KeyCodes".

Jetzt kannst Du das Programm wieder kompilieren und starten. Mit den Tasten "A, Y" und "Auf, Ab" kann man jetzt beide Schläger steuern.

Nun bewegen wir den Ball um die aktuelle x/y Geschwindigkeit des Balls:

```
// Ball
ball_x=ball_x+ball_sx
ball_y=ball_y+ball_sy
```

Kollision mit Rand

Wenn der Ball oben oder unten anstößt, drehen wir einfach seine y-Geschwindigkeit um:

```
// Ball unten am Rand
IF ball_y>464
→ball_y=464
→ball_sy=-ball_sy
ENDIF

// Ball oben am Rand
IF ball_y<0
→ball_y=0
→ball_sy=-ball_sy
ENDIF
```

Wenn der Ball links/rechts an den Rand stößt, bekommt ein Spieler einen Punkt und der Ball wird wieder in die Mitte gesetzt.

```
// Ball linker Rand -> Punkt für Spieler 1
IF ball_x<0
→score[1]=score[1]+1
→GOSUB ResetBall
ENDIF

// Ball rechter Rand -> Punkt für Spieler 0
IF ball_x>624
→score[0]=score[0]+1
→GOSUB ResetBall
ENDIF
```

Kollision mit Schläger

Wir durchlaufen eine Schleife für alle Spieler:

```
FOR num=0 TO 1
```

Zunächst fragen wir ab, ob sich der Ball in Richtung des Schlägers bewegt. Wenn der Ball nämlich schon umgedreht wurde, kann

er evtl. trotzdem noch mit dem Schläger kollidieren. Dadurch würde er sonst wieder auf die eigene Seite bewegt werden.

```
IF (ball_sx<0 AND num = 0) OR (ball_sx>0 AND num=1)
```

Ist dies der Fall, prüfen wir, ob sich die beiden Rechtecke Ball + Schläger überlappen:
Wenn der Ball an einen Schläger kommt, wird seine X-Geschwindigkeit umgedreht. Die Geschwindigkeiten in X und Y werden beide erhöht. Jedoch nicht im gleichen Maße, weil sich sonst der Flugwinkel des Balls nicht ändern würde.

```
col = BOXCOLL(bat_x[num], bat_y[num], 16, 64,   ball_x, ball_y, 16, 16)
→IF col=TRUE
→// Ballgeschwindigkeit in X umdrehen
→ball_sx= -ball_sx
→// Schneller machen / Speed up
→ball_sx = ball_sx * 1.2
→ball_sy = ball_sy * 1.05
```

Jetzt noch unsere IF Bedingungen und die Schleife abschließen:

```
→→ENDIF
→ENDIF
NEXT
```

Voila. Fertig ist das Spiel.
Überprüfe alle Programmteile sorgfältig und verstehe jede einzelne Zeile des Codes, bevor Du versuchst ein eigenes Spiel zu schreiben. Sieh' die Referenz der Befehle in dieser Hilfe nach und versuche das Spiel auf 4 Spieler zu erweitern (oben und unten noch ein Schläger).
Erstelle einen Computer-Spieler, so dass Du auch allein spielen kannst. Dabei musst Du nur prüfen, ob die Y-Koordinate des Balls höher oder tiefer als der Mittelpunkt des Schlägers liegt, und dann den Schläger evtl. auf oder ab bewegen.
Oder versuche, dass je nach Aufschlagsposition des Balls der Flugwinkel geändert wird.
Das Programm lässt viel Platz für eigene Ideen.

Viel Spass beim Basteln.

Gesamter Quellcode:

```
// Pong

DIM bat_y[2]
DIM bat_x[2]
DIM score[2]

// Voreinstellungen / Setup values
GOSUB Init

// Hauptschleife / Main Loop
main:
→GOSUB MoveAll
→GOSUB ShowAll
GOTO main



// ------------------------------------------------------------ //
// -=#  INIT  #=-
// ------------------------------------------------------------ //
SUB Init:
→GOSUB ResetBall

→// Spielfeld zeichnen, als Hintergrund nutzen
→// Draw playfield, use as background bitmap
→BLACKSCREEN
→FILLRECT 0,   0, 639, 15,  RGB(255, 255, 255)
→FILLRECT 0, 464, 639, 479, RGB(255, 255, 255)
→FILLRECT 312, 0, 327, 479, RGB(255, 255, 255)
→USEASBMP

→// Schläger 0 setzten / Reset Bat 0
→bat_y[0]=240; bat_y[1]=240
→// Schläger 1 setzten / Reset Bat 1
→bat_x[0]=16; bat_x[1]=600
→
→// Klassischer Zeichensatz / Classic Font
→LOADFONT "FontBW.bmp", 0
ENDSUB // INIT

// ------------------------------------------------------------ //
// -=#  RESETBALL  #=-
// ------------------------------------------------------------ //
```

```
SUB ResetBall:
→// Diese Variablen sind als LOCAL definiert:
→// These variables are defined LOCAL:
→// void
→ball_x=320
→ball_y=240

→IF ball_sx<0
→→ball_sx=1
→ELSE
→→ball_sx=-1
→ENDIF

→ball_sy=1
ENDSUB // RESETBALL


// ------------------------------------------------------------ //
// -=#  SHOWALL  #=-
// ------------------------------------------------------------ //
SUB ShowAll:
→// Die Schläger anzeigen / Show the bats
→FOR num=0 TO 1
→→FILLRECT bat_x[num], bat_y[num], bat_x[num]+16, bat_y[num]+64, RGB(255, 255, 255)
→→PRINT score[num], num*320 + 32, 16
→NEXT
→// Der Ball / The ball
→FILLRECT ball_x, ball_y, ball_x+16, ball_y+16, RGB(255, 255, 255)
→SHOWSCREEN
ENDSUB // SHOWALL


// ------------------------------------------------------------ //
// -=#  MOVEALL  #=-
// ------------------------------------------------------------ //
SUB MoveAll:
// Paddles
→FOR num=0 TO 1
→→// Tasten / Keys:  A, Y
  IF KEY(30) THEN bat_y[0]=bat_y[0]-2
  IF KEY(44) THEN bat_y[0]=bat_y[0]+2
  // Tasten / Keys:  /\, \/
  IF KEY(200) THEN bat_y[1]=bat_y[1]-2
  IF KEY(208) THEN bat_y[1]=bat_y[1]+2

  // Schläger am Rand? / Bat at border?
  IF bat_y[num]<0   THEN bat_y[num]=0
  IF bat_y[num]>416 THEN bat_y[num]=416
 NEXT

// Ball
 ball_x=ball_x+ball_sx
 ball_y=ball_y+ball_sy

 // Ball unten am Rand / Ball at lower border
 IF ball_y>464
  ball_y=464
  ball_sy=-ball_sy
 ENDIF

 // Ball oben am Rand / Ball at upper border
 IF ball_y<0
  ball_y=0
  ball_sy=-ball_sy
 ENDIF

 // Ball Linker Rand / Ball at left border
 IF ball_x<0
  score[1]=score[1]+1
  GOSUB ResetBall
 ENDIF

 // Ball rechter Rand / Ball at right border
 IF ball_x>624
  score[0]=score[0]+1
  GOSUB ResetBall
 ENDIF

 // Pong
 FOR num=0 TO 1 // Für jeden Spieler / For each player
  // Bewegt sich der Ball auf den Schläger 'num' zu?
  // Does ball move toward bat 'num'
  IF (ball_sx<0 AND num = 0) OR (ball_sx>0 AND num=1)
   col=BOXCOLL(bat_x[num], bat_y[num], 16, 64,   ball_x, ball_y, 16, 16)
```

```
   IF col=TRUE
    // Ballgeschwindigkeit in X umdrehen / Flip ball's X-direction
    ball_sx= -ball_sx
    // Schneller machen / Speed up
    ball_sx = ball_sx * 1.2
    ball_sy = ball_sy * 1.05
   ENDIF
  ENDIF
 NEXT

ENDSUB // MOVEALL
```

Das erste Spiel

One More

Über das Spiel

Das ursprüngliche Spiel (denke ich), war ein Teil eines Spiels, das LOGO von Starbyte für den Amiga500 genannt wurde. Ziel ist, alle farbigen Punkte von einem Spielfeld zu entfernen. Wann immer der Spieler einen Block anklickt, ändert er dessen Status und den der vier angrenzenden Blöcke.

Also, klickt man auf den Mittelpunkt dieses Spielfelds:
###
###
###
sähe es danach so aus:
#+#
+++
#+#

Dieses Spiel ist recht einfach zu programmieren und läßt eine Menge Raum für eigene Ideen. Es ist auch nett, da wir Computer erzeugte Spielstufen haben können. Wir lassen einfach den Computer zufällig Punkte anklicken. Das Spielen ist ziemlich schwierig, sobald der Computer mehr als etwa 10 Punkte angeklickt hat...

Neues Projet

Nach dem Start von GLBasic den Kopf "Neues Projekt" auswählen, und einen Namen + Pfad angeben. z.B. ..\GLBasic\Projects\OneMore

Das Hauptprogramm

Nun, der Grundcode sieht etwa so aus (Pseudo-code):

```
main:
   WENN level_komplett DANN neues_level_erstellen
   WENN maus_geklickt DANN steine_aendern
   spielfeld_anzeigen
GEHE_ZU main
```

Zuerst benötigen wir ein Spielfeld (playfield), um zu speichern ob ein Block aktiviert oder deaktiviert (leer) ist. Wir verwenden den Wert 0 für leer, da der DIM Befehl immer alle Werte mit 0 belegt und folglich veranlasst, dass die erste WENN Aussage ausgewertet wird - ein neues Level wird für uns erstellt.

```
// ------------------------------- //
// Project: OneMore

DIM playfield[10][10]
→level = 0
```

Jetzt, haben wir ein Feld, das "playfield" genannt wird von der Größe 10x10 Zahlen. Wir können die Zellen über die Indizes 0-9 ansprechen.Siehe DIM Befehl.

```
// Hauptprogramm
  main:
    MOUSESTATE mx, my, b1, b2
    PRINT "<=", mx, my-8
    SHOWSCREEN
  GOTO main
  END
```

Kompilieren und laufen lassen. Schaut nett aus, aber macht nicht viel. Wir haben einen Zeiger, den wir mit der Maus verschieben

können. Am besten alle verwendeten Befehle in deren Referenz nachlesen. Die PRINT Zeile ist unser Mauszeiger. Man kann später ein nettes SPRITE dafür laden und verwenden. Das my-8 verschiebt die Anzeige um eine halbe Zeichenhöhe. Das SHOWSCREEN nicht vergessen!

Spielfeld anzeigen

Jetzt, möchten wir die playfield-Daten anzeigen, also fügen wir eine neue SUB hinzu. Drücke dazu den SUB-Knopf und gebe den Namen "ShowPlayfield" ein.

```
// ------------------------------------------------------------ //
// -=#  SHOWPLAYFIELD  #=-
// ------------------------------------------------------------ //
SUB ShowPlayfield:
→LOCAL x, y, color[]

DIM color[2]
→color[0] = RGB( 50,  50, 255) // Not set
→color[1] = RGB( 50, 255,  50) // Set

→FILLRECT 0,0, 320, 320, RGB(255,255,255)
→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→FILLRECT x*32+1, y*32+1, x*32+29, y*32+29, color[playfield[x][y]]
→→NEXT
→NEXT
→PRINT "Klicks: "+clicks, 360, 120
→PRINT "Level:  "+level, 360, 160
ENDSUB // SHOWPLAYFIELD
```

Hier erstellen wir ein lokales Feld, das color (=Farbe) genannt wird und weisen dem ersten Index [0] Blau zu und dem zweiten Index [1] Grün. Zunächst zeichnen wir ein weißes Viereck, das über den gesamten Berich des Spielfelds gezogen wird. Dann kommt eine doppelte FOR Schleife. Eine für x und dann wiederum für jedes x noch ein y für jede Zeile. Für jede Zelle füllen wir ein Rechteck mit der Zellenfarbe. Wir benutzen color[playfield[x][y]] für den RGB-Wert. So erhält 0 das Blau, 1 bekommt Grün. Weiter informieren wir den Benutzer über das gegenwärtige Level und die Anzahl der Mausklicks bis jetzt für dieses Level.
Füge diese Zeile im Hauptprogamm unterhalb der "MOUSESTATE-" Zeile hinzu:

```
GOSUB ShowPlayfield
```

Kompiliere + starte das Spiel. Sieht schon besser aus, aber macht noch nicht viel?

Interaktion

Zunächst, fügen wir eine Funktion mit dem FUNKTIONS-Knopf hinzu und tragen den Namen "Change" ein. Die Argumente sind: x, y
Dieses ist der Punkt, an dem wir die playfield-Felder ändern möchten.

```
// ------------------------------------------------------------ //
// -=#  CHANGE  #=-
// ------------------------------------------------------------ //
FUNCTION Change: x, y
→// Diese Variablen sind LOCAL definiert:
→// x, y
→IF x>=0 AND x<10 AND y>=0 AND y<10
→→set = playfield[x][y] // Alten Zustand holen
→→
→→IF set = TRUE
→→→set=FALSE
→→ELSE
→→→set=TRUE
→→ENDIF
→→// order einfach:
→→// set = 1-set
→→playfield[x][y]=set // Neuen Zustand setzen
→ENDIF
ENDFUNCTION
```

In dieser Funktion setzen wir das playfield[x][y] auf 1, wenn es 0 war und umgekehrt. Aber wir überprüfen auch, ob der Punkt x, y die 0-9, 0-9 Grenzen überschreitet. Wenn das so ist, ändern wir nichts, sonst kommt eine Fehlermeldung vom laufenden Programm: DIM ausserhalb zulässigem Bereich.

Füge jetzt eine Funktion mit dem FUNKTIONS-Knopf hinzu. Gebe den Namen "Click" und die Argumentliste: x, y an.

Dieses ist der Punkt, der angeklickt werden soll.

```
// ------------------------------------------------------------ //
// -=#  CLICK  #=-
```

```
// ------------------------------------------------------------ //
FUNCTION Click: x, y
→// Diese Variablen sind als LOCAL definiert:
→// x, y
→clicks=clicks+1
→Change(x-1, y)
→Change(x+1, y)
→Change(x, y)
→Change(x, y-1)
→Change(x, y+1)
ENDFUNCTION
```

Hier ändern (Change) wir den Wert des angeklickten Punktes und seiner Nachbarn. Weiter erhöhen wir die Variable für die getätigten Mausklicks um 1. Ändere das Hauptprogramm, so dass es wie folgt aussieht:

```
DIM playfield[10][10]

→level = 0

→// Hauptprogramm
→start:
→→MOUSESTATE mx, my, b1, b2
→→IF b1
→→→IF mousefree THEN Click(mx/32, my/32)
→→→mousefree=FALSE
→→ELSE
→→→mousefree=TRUE
→→ENDIF
→
→→GOSUB ShowPlayfield
→→PRINT "<=", mx, my-8
→→SHOWSCREEN
→GOTO start
END
```

Die Variable mousefree gibt uns an, ob der Spieler die Maustaste seit dem letzen Click() immer noch gedrückt hält. Wenn ja, wird die Schleife solange durchlaufen, bis er die Maustaste wieder losgelassen hat. Starte jetzt das Programm. Es sollte ganz nett aussehen. Mach Dich mit der Funktionsweise des Spiels vertraut. Klicke zufällig einige Male und versuche, das Brett wieder zu löschen. Macht das Spaß?

Level

Fügen wir eine SUB "NewLevel" hinzu:

```
// ------------------------------------------------------------ //
// -=#  NEWLEVEL  #=-
// ------------------------------------------------------------ //
SUB NewLevel:
→LOCAL x, y, i

→// Spielfeld löschen - um sicher zu gehen
→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→playfield[x][y]=FALSE
→→NEXT
→NEXT

→level=level+1
→// Zufällig 'rumklicken
→FOR i=0 TO level
→→Click(RND(9), RND(9))
→NEXT
→clicks=0
→GOSUB ShowPlayfield
→PRINT "Level: "+level + " - bereit?", 0, 360
→SHOWSCREEN
→MOUSEWAIT
ENDSUB // NEWLEVEL
```

In dieser SUB, löschen wir zuerst das playfield, um wirklich sicher zu gehen, dass es leer ist. Zunächst erhöhen wir die "level" Variable. Dann lassen wir den Computer ein paar mal zufällig Click()en und setzen die Anzahl der clicks auf 0, da der Spieler ja noch nicht geklickt hat. Jetzt kann der Computer schon Level erzeugen, aber noch nicht feststellen, ob ein Level gelöst wurde. Dazu fügen wir eine FUNCTION ein mit dem Namen: "IsLevelComplete":

```
// ------------------------------------------------------------ //
// -=#  ISLEVELCOMPLETE  #=-
// ------------------------------------------------------------ //
FUNCTION IsLevelComplete:
→LOCAL x, y
```

```
→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→IF playfield[x][y]=TRUE THEN RETURN FALSE
→→NEXT
→NEXT
→RETURN TRUE
ENDFUNCTION
```

Hier prüfen wir einfach in einer doppelten Schleife, ob eine Zelle gesetzt (TRUE = 1) ist. Wenn ja, geben wir FALSE zurück. Das Level ist also noch nicht leer. Wenn die Schleife durchläuft ohne durch ein RETURN herauszuspringen, muss davon ausgegangen werden, dass das Feld leer ist. Also geben wir TRUE zurück.
Um das komplette Spiel zu testen, füge bitte folgenden Code im Hauptprogramm unter "main:" ein:

```
→→IF IsLevelComplete()
→→→GOSUB NewLevel
→→→mousefree=FALSE
→→ENDIF
```

Starte das Spiel. Wie fühlt man sich, wenn man ein völlig qualifiziertes Computerspiel geschrieben hat?
Wenn Du

```
BLENDSCREEN "images"+level".bmp"
```

der SUB NewLevel hinzufügst, ist das Spiel sogar besser als die ursprüngliche Amiga-Version! Ist das nicht ein großartiges Gefühl? Du bist jetzt der Herr dieses Computers. Sage ihm, was zu tun ist, und er folgt Deinem Befehl. Du kannst Welten mit einigen einfachen Befehlen erschaffen. Hier ist der komplette Quellcode, falls Du etwas verpasst haben solltest:

```
// ------------------------------- //
// Project: OneMore

DIM playfield[10][10]

→level = 0

→// Hauptprogramm
→start:
→→IF IsLevelComplete()
→→→GOSUB NewLevel
→→→mousefree=FALSE
→→ENDIF

→→MOUSESTATE mx, my, b1, b2
→→IF b1
→→→IF mousefree THEN Click(mx/32, my/32)
→→→mousefree=FALSE
→→ELSE
→→→mousefree=TRUE
→→ENDIF
→
→→GOSUB ShowPlayfield
→→PRINT "<=", mx, my-8
→→SHOWSCREEN
→GOTO start
END

// ------------------------------------------------------------ //
// -=#  SHOWPLAYFIELD  #=-
// ------------------------------------------------------------ //
SUB ShowPlayfield:
→LOCAL x, y, color[]

DIM color[2]
→color[0] = RGB( 50,  50, 255) // Not set
→color[1] = RGB( 50, 255,  50) // Set

→FILLRECT 0,0, 320, 320, RGB(255,255,255)
→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→FILLRECT x*32+1, y*32+1, x*32+29, y*32+29, color[playfield[x][y]]
→→NEXT
→NEXT
→PRINT "Clicks: "+clicks, 360, 120
→PRINT "Level:  "+level, 360, 160
ENDSUB // SHOWPLAYFIELD


// ------------------------------------------------------------ //
// -=#  CLICK  #=-
```

```
// ------------------------------------------------------------ //
FUNCTION Click: x, y
→// These values are defined LOCAL:
→// x, y
→clicks=clicks+1
→Change(x-1, y)
→Change(x+1, y)
→Change(x, y)
→Change(x, y-1)
→Change(x, y+1)
ENDFUNCTION


// ------------------------------------------------------------ //
// -=#  CHANGE  #=-
// ------------------------------------------------------------ //
FUNCTION Change: x, y
→// These values are defined LOCAL:
→// x, y
→IF x>=0 AND x<10 AND y>=0 AND y<10
→→set = playfield[x][y] // alten Zustand merken
→→
→→IF set = TRUE
→→→set=FALSE
→→ELSE
→→→set=TRUE
→→ENDIF
→→// oder einfach nur:
→→// set = 1-set
→→playfield[x][y]=set // Neuen Zustand setzen
→ENDIF
ENDFUNCTION


// ------------------------------------------------------------ //
// -=#  NEWLEVEL  #=-
// ------------------------------------------------------------ //
SUB NewLevel:
→LOCAL x, y, i

→// Spielfeld lösche - nur um sicher zu gehen
→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→playfield[x][y]=FALSE
→→NEXT
→NEXT

→level=level+1
→
→// Zufällig 'rumklicken
→FOR i=0 TO level
→→Click(RND(9), RND(9))
→NEXT
→clicks=0
→GOSUB ShowPlayfield
→PRINT "Level: "+level + " - bereit?", 0, 360
→SHOWSCREEN
→MOUSEWAIT
ENDSUB // NEWLEVEL


// ------------------------------------------------------------ //
// -=#  ISLEVELCOMPLETE  #=-
// ------------------------------------------------------------ //
FUNCTION IsLevelComplete:
→LOCAL x, y

→FOR x=0 TO 9
→→FOR y=0 TO 9
→→→IF playfield[x][y]=TRUE THEN RETURN FALSE
→→NEXT
→NEXT
→RETURN TRUE
ENDFUNCTION
```

Hausaufgaben

AUFGABEN

So. Das ist schon eine ganze Menge auf einmal gewesen. Die folgenden Tests geben Dir einen guten Start in Dein

Programmiererleben. Du solltest Dir jeweils einen ganzen Tag Zeit nehmen und mit den Aufgaben herumspielen.

Allgemein

Du hast den Befehl 'PRINT' kennengelernt. Hier nochmal seine Syntax:

```
PRINT [ZAHL/WORT], X-Pixel nach rechts, Y-Pixel nach Unten;
```

Ein kleiner Test, der Dich bereit für das nächste Kapitel machen soll:
Stell Dir vor, Du bist ein Lehrer und hast eine Klasse mit 4 Schülern (2 Reihen, 2 Spalten). Weil Dir die Schüler so leid tun, möchtest Du einen Computer die Aufgabe überlassen, den Namen des Schülers auszuwählen, der ausgefragt wird.
Schreibe ein Programm, bei dem Du für die Schulbänke in einer Sub-Funktion die Namen der Schüler eingeben musst, und wähle zufällig eine Reihe und eine Spalte aus, von der Du dann den Schülernamen, die Reihe und die Spalte auf den Bilschirm schreibst.

Klassenaufteilung:

+	Spalte0	Spalte1
Reihe0	Tom	Markus
Reihe1	Sabine	Manuela

Ausgabe (zum Beispiel)

Sabine
sitzt in:
Reihe 1
Spalte 0

Tips:
-Wenn Du den Zeichensatz nicht änderst, ist jeder Buchstabe 16x16 Pixel groß.
-Eine Zufallszahl gibt Dir die Funktion RND(maximum). (Sieh' ruhig im Tutorial nach).

2D-Grafik

Schreibe ein Programm, das 2 Sprites lädt, und mit ihnen alle Sprite-Funktionen der Reihe nach durchgeht. Also, zuerst die Sprites mit 'SPRITE' von links nach rechts, dann mit ROTOSPRITE einmal drehen, dann mit ROTOZOOMSPRITE einmal drehen und auf Bildschirmgröße aufblasen und schließlich einmal über den ganzen Bildschirm mit ZOOMSPRITE zoomen.
Nun soll das Programm ein Bild laden, und das gleiche Programm noch einmal durchgehen, dabei aber den Alphawert von Sprite1 auf 100 und den Alphawert von Sprite2 auf -100 setzten. Du findest ein Projekt, in dem die Grafikdateien und ein paar Hilfestellungen gegeben sind, im Projektordner.

Fortgeschritten

Fortgeschrittene Techniken:

Diese Sektion ist für Programmierer und fortgeschrittene Benutzer. Man kann eine Menge fotgeschrittene Programmiertechniken in GLBasic anwenden.

Lokale und globale Variablen

Man definiert lokale Variablen in SUBs und der Hauptschleife, wenn man die SUBs unabhängig vom Hauptprogramm erstellen möchte, um diese später in anderen Programmen nutzen zu können. Der Befehl LOCAL definiert Variablen als lokal:

```
LOCAL num, wort$, num_feld[], wort_feld$[]
```

Um auf eine globale Variable mit gleichem Namen einer lokalen Variable zuzugreifen, benutzt man den Befehl GLOBAL . Am besten schildert das ein Beispiel aus der Befehlsreferenz der beiden Befehle.

Dynamische Felder

Ja, GLBasic bietet die Möglichkeit, Felder mit DIM zu re-dimensionieren. Will man den Inhalt behalten, muss man REDIM verwenden.

```
DIM a[5]
...
DIM a[4][12] // Ja!!
```

Speicher freigeben

Speicherfreigabe wird von GLBasic nicht direkt unterstützt. Aber man kann belegten Speicherplatz sehr einfach wieder freigeben, indem man eine Resource lädt, die nicht vorhanden ist.

```
LOADSPRITE "", 0
```

Das gibt den von der Grafik '0' belegten Speicherplatz wieder frei.

```
LOADSOUND "", 0, 1
```

Das git den Speicher des Sounds '0' wieder frei

```
DIM a[0]
```

Das gibt den Speicher des Feldes wieder frei

Hexadezimalzahlen

Es können Zahlen nicht nur im Dezimalsystem sondern auch hexadezimal eingegeben werden. Das macht Sinn, wenn mit den Binär-Operatoren AND und OR gearbeitet wird.

```
ad=127; ax=0xff
```

Inline C++

Die ganz perversen können mit den Befehlen INLINE/ENDINLINE C++ Quellcode einbinden.

Funktionen verstecken

Wenn der Jump-Browser zu voll wird, kann man vor unwichtige Funktionen ein '@' schreiben. Dann wird die Sprungmarke nicht mehr aufgeführt. Der Jump-Browser lässt diese trotzdem auf Wunsch wieder darstellen.
Beispiel:

```
@hier:
@FUNCTION test: a, b
@SUB MeineSub:
```

Refs:

BYREF, DIM, INLINE, REDIM

Andere Betriebssysteme

Plattformen

GLBasic unterstützt verschiedene Ziel-Plattformen. Damit kann man z.B. das selbe Spiel unter Win32 (Windows 98, NT, 2000, XP, ...) und WinCE (PocketPC2002, 2003, Windows Mobile, ...) erstellen.
Dazu gibt es in dem Menüpunkt "Projekt/Optionen" die Klappe "Plattform". Hier einfach das gewünschte Bettriebssystem auswählen, die erforderlichen Einstellungen vornehmen und "Übernehmen" klicken. Wenn "Kompilieren für diese Plattform" angehakt ist, wird beim nächsten Erstellen auch eine ausführbare Datei für diese Plattform erstellt. Die Dateiendung ist dabei anders als bei Win32, so hat WinCE z.b. Spielname_PPC.exe.
Weitere Platformen sind in Vorbereitung. Die zusätzlichen Plattformen benötigen die Ausbaustufe "NET" oder GLBasic SDK premium.

ABS()

n# = ABS(m#)

Gibt den absoluten (psoitiven) Wert von m# zurück.

ABS()

n# = ABS(m#)

Gibt den absoluten (psoitiven) Wert von m# zurück.

ACOS()

.

Refs:

SIN()

ALLOWESCAPE

ALLOWESCAPE a#

Erlaubt (a#=TRUE) oder verbietet (a#=FALSE), dass der User mit der ESC Taste abbrechen kann.
Das entfernt auch das X-Icon bei PocketPC Spielen.

Refs:

END, KEY()

ALPHAMODE

ALPHAMODE alpha#

Ändert den Alpha-modus für alle kommenden Grafikausgaben. Der alpha# Wert reicht von -1 bis +1.

ALPHA WERTE:
alpha# > 0 - Helles Alpha blending. (Feuer, Explosionen)
alpha# = 0 - Kein Alpha blending.
alpha# < 0 - Interpoliertes Alpha blending (Schatten, andere coole Effekte)

Sample:

```
// ALPHAMODE
LOADBMP "Back.bmp"
PRINT "Kein Alpha", 100, 100
ALPHAMODE .5
PRINT "Alpha +0.5", 100, 140
ALPHAMODE 1
PRINT "Alpha +1", 100, 160
ALPHAMODE -1
PRINT "Alpha -1", 100, 180
SHOWSCREEN
MOUSEWAIT
```

Refs:

DRAWLINE, INPUT, POLYVECTOR, PRINT, SPRITE, SETPIXEL

AND

a# = b# AND c#
a# = b# OR c#

Binäre Operatoren. Häufig für IF Anweisung benutzt.

AND:
Bitmuster b#: 0010010
Bitmuster c#: 0010100
Bitmuster a#: 0010000
Jedes Bit, das in b# UND c# vorhanden ist wird 1, alle anderen 0.

OR:
Bitmuster b#: 0010010
Bitmuster c#: 0010100
Bitmuster a#: 0010110
Jedes Bit, das in b# ODER c# vorhanden ist wird 1, alle anderen 0.

Sample:

```
a=0
b=5
IF a>0 OR b>0 THEN PRINT "a>0 OR b>0", 100, 100

a=125000
lowbyte=a AND 0x00FF
hibyte= a AND 0xFF00
SHOWSCREEN
MOUSEWAIT
```

Refs:

IF

ASC()

num# = ASC(wort$)

Gibt den ASCII code des ersten Zeichens von wort$ zurück.

```
PRINT ASC("a"), 100, 100
SHOWSCREEN
MOUSEWAIT

// Dec Hx Char Dec Hx Char  Dec Hx Char  Dec Hx Char
// ----------- -----------  -----------  -----------
//   0  0 NUL  32 20 SPACE  64 40 @       96 60 `
//   1  1 SOH  33 21 !      65 41 A       97 61 a
//   2  2 STX  34 22 ''     66 42 B       98 62 b
//   3  3 ETX  35 23 #      67 43 C       99 63 c
//   4  4 EOT  36 24 $      68 44 D      100 64 d
//   5  5 ENQ  37 25 %      69 45 E      101 65 e
//   6  6 ACK  38 26 &      70 46 F      102 66 f
//   7  7 BEL  39 27 '      71 47 G      103 67 g
//   8  8 BS   40 28 (      72 48 H      104 68 h
//   9  9 TAB  41 29 )      73 49 I      105 69 i
//  10  A LF   42 2A *      74 4A J      106 6A j
//  11  B VT   43 2B +      75 4B K      107 6B k
//  12  C FF   44 2C ,      76 4C L      108 6C l
//  13  D CR   45 2D -      77 4D M      109 6D m
//  14  E SO   46 2E .      78 4E N      110 6E n
//  15  F SI   47 2F /      79 4F O      111 6F o
//  16 10 DLE  48 30 0      80 50 P      112 70 p
//  17 11 DC1  49 31 1      81 51 Q      113 71 q
//  18 12 DC2  50 32 2      82 52 R      114 72 r
//  19 13 DC3  51 33 3      83 53 S      115 73 s
//  20 14 DC4  52 34 4      84 54 T      116 74 t
//  21 15 NAK  53 35 5      85 55 U      117 75 u
//  22 16 SYN  54 36 6      86 56 V      118 76 v
//  23 17 ETB  55 37 7      87 57 W      119 77 w
//  24 18 CAN  56 38 8      88 58 X      120 78 x
//  25 19 EM   57 39 9      89 59 Y      121 79 y
//  26 1A SUB  58 3A :      90 5A Z      122 7A z
//  27 1B ESC  59 3B ;      91 5B [      123 7B {
//  28 1C FS   60 3C <      92 5C \      124 7C |
//  29 1D GS   61 3D =      93 5D ]      125 7D }
//  30 1E RS   62 3E >      94 5E ^      126 7E ~
//  31 1F US   63 3F ?      95 5F _      127 7F DEL
```

Refs:

CHR$(), MID$()

ASIN()

.

Refs:

SIN()

ATAN()

.

Refs:

SIN()

AUTOPAUSE

AUTOPAUSE modus#

Schaltet den Auto-Pausemodus ein (TRUE) oder aus (FALSE). Autopause bewirkt, dass beim verlieren des Fensterfokuses das Spiel in den Pausemodus umstellt, was man bei Netzwerkservern nicht will.

bAND()

num# = bAND(num1#, num2)
num# = bAND(num1#, num2)
num# = bOR(num1#, num2)
num# = bXOR(num1#, num2)
num# = bNOT(num1#)

Führt boolsche Operationen an Ganzzahlwerten aus. Die Größe der Zahlen ist dabei 32 Bit.

Beispiel:

```
FOR a=0 TO 1
   FOR b=0 TO 1
      x = (a+b*2)*160
      PRINT "a="+a+" b="+b, x, 30

      PRINT "bAND="+bAND(a, b), x,  60
      PRINT "bOR ="+bOR (a, b), x,  80
      PRINT "bXOR="+bXOR(a, b), x, 100
      IF b=0 THEN PRINT "bNOT="+bNOT(a),    x, 120
   NEXT
NEXT
SHOWSCREEN
MOUSEWAIT
```

Refs:

bNOT(), bOR(), bXOR()

BLACKSCREEN

BLACKSCREEN

Der Bildschirm wird nun nach jedem SHOWSCREEN mit schwarzer Farbe gelöscht. Dieser Befehl revidiert die Befehle LOADBMP und USEASBMP.

Sample:

```
LOADBMP "Bild.bmp"
```

```
BLACKSCREEN // Wieder Schwarz
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

BLENDSCREEN, LOADBMP, USEASBMP

BLENDSCREEN

BLENDSCREEN bmp$

Überblendet den aktuellen Back-Buffer (verdeckter Grafikbildschirm) in das Bild bmp$.

Sample:

```
PRINT "Back-Buffer vorher", 100, 100
BLENDSCREEN "Bild.bmp"
```

Refs:

BLACKSCREEN, LOADBMP, USEASBMP

bNOT()

.

Refs:

bAND()

bOR()

.

Refs:

bAND()

BOUNDS()

n# = BOUNDS(feld#$[], ndim#)

Gibt die mit DIM angegebenen Feldgrößen für die gewünschte Dimension wieder.

```
// BOUNDS()

DIM a[5][7]

   PRINT BOUNDS(a[], 0), 0, 100 //  = 5
   PRINT BOUNDS(a[], 1), 0, 120 // = 7
   SHOWSCREEN
   MOUSEWAIT
```

BOXCOLL

col# = BOXCOLL(xa#, ya#, ba#, ha#, xb#, yb#, bb#, hb#)

Überprüft, ob sich 2 Rechtecke überlappen. Damit kannst Du Kollisionsabfragen einfach durchführen.
col# enthält bei Kollision den Wert TRUE, sonst FALSE.
xa#, xb#, ya#, yb# sind die oberen linken Ecken der Rechtecke a und b
ba#, bb# sind die Breiten
ha#, hb# sind deren Höhen

Sample:

```
start:
MOUSESTATE mx, my, ba, bb
PRINT "X", 100, 100
PRINT "H", mx, my
IF BOXCOLL( mx, my, 16, 16, 100, 100, 16, 16)
    PRINT "!!!", 100, 150
ENDIF
SHOWSCREEN
GOTO start
```

Refs:

SPRCOLL

BREAK

BREAK

Unterbricht die aktuelle FOR oder WHILE Schleife.

Refs:

CONTINUE, FOR, WHILE

bXOR()

.

Refs:

bAND()

BYREF

FUNCTION f: BYREF arg#$

Mit dem Modifizierer "BYREF" kann man eine Variable als Referenz übergeben. Damit kann man innerhalb der Function den Wert der Übergabevariable ändern. Wörter werden auf diese Weise schneller übergeben, da nur ein Zeiger statt dem gesamten Inhalt kopiert werden muss.
Felder werden immer als Referenz übergeben. Hier ist ein BYREF obsolet.

Dieses Schlüsselwort ist für Profis gedacht.

```
LOCAL b$
b$ = "XY"
foo(b$)

// print "test"
PRINT b$, 0,0
SHOWSCREEN
MOUSEWAIT

FUNCTION foo: BYREF a$
   a$ = "test"
ENDFUNCTION
```

Refs:

GLOBAL, LOCAL

CALLBACK

CALLBACK FUNCTION foo: args#$

Der CALLBACK Modifizierer erlaubt es Funcktionen überschreibbar zu machen. Damit kann man eine standart Behandlung von

Dingen einfügen, jedoch einer Projektdatei zusätzlich erlauben diesen Standartweg zu überschreiben.

Wenn die gleiche Funktion ohne den CALLBACK Modifizierer existiert wird diese anstatt der als CALLBACK markierten Funktion aufgerufen.

```
SuperPrint("Hello World")


CALLBACK FUNCTION SuperPrint: a$
PRINT a$, 0,0
ENDFUNCTION

// Das auskommenieren, wenn nur CALLBACK
// aufgerufen werden soll
FUNCTION SuperPrint: a$
LOCAL tx,ty
GETFONTSIZE tx, ty
   tx = tx * LEN(a$)
   FILLRECT 0,0,tx,ty, RGB(0,0,255)
   PRINT a$, 0,0
ENDFUNCTION
```

Refs:

FUNCTION

CASE

.

Refs:

SELECT

CHR$()

wort$ = CHR$(num#)

Erstellt das ASCII Zeichen num#.

```
PRINT CHR$(0x61), 100, 100 // a
SHOWSCREEN
MOUSEWAIT

// Dec Hx Char Dec Hx Char  Dec Hx Char  Dec Hx Char
// ---------- ----------  ----------  ----------
//   0  0 NUL  32 20 SPACE  64 40 @      96 60 `
//   1  1 SOH  33 21 !      65 41 A      97 61 a
//   2  2 STX  34 22 ''    66 42 B      98 62 b
//   3  3 ETX  35 23 #      67 43 C      99 63 c
//   4  4 EOT  36 24 $      68 44 D     100 64 d
//   5  5 ENQ  37 25 %      69 45 E     101 65 e
//   6  6 ACK  38 26 &      70 46 F     102 66 f
//   7  7 BEL  39 27 '      71 47 G     103 67 g
//   8  8 BS   40 28 (      72 48 H     104 68 h
//   9  9 TAB  41 29 )      73 49 I     105 69 i
//  10  A LF   42 2A *      74 4A J     106 6A j
//  11  B VT   43 2B +      75 4B K     107 6B k
//  12  C FF   44 2C ,      76 4C L     108 6C l
//  13  D CR   45 2D -      77 4D M     109 6D m
//  14  E SO   46 2E .      78 4E N     110 6E n
//  15  F SI   47 2F /      79 4F O     111 6F o
//  16 10 DLE  48 30 0      80 50 P     112 70 p
//  17 11 DC1  49 31 1      81 51 Q     113 71 q
//  18 12 DC2  50 32 2      82 52 R     114 72 r
//  19 13 DC3  51 33 3      83 53 S     115 73 s
//  20 14 DC4  52 34 4      84 54 T     116 74 t
//  21 15 NAK  53 35 5      85 55 U     117 75 u
//  22 16 SYN  54 36 6      86 56 V     118 76 v
//  23 17 ETB  55 37 7      87 57 W     119 77 w
//  24 18 CAN  56 38 8      88 58 X     120 78 x
//  25 19 EM   57 39 9      89 59 Y     121 79 y
//  26 1A SUB  58 3A :      90 5A Z     122 7A z
//  27 1B ESC  59 3B ;      91 5B [     123 7B {
//  28 1C FS   60 3C <      92 5C \     124 7C |
//  29 1D GS   61 3D =      93 5D ]     125 7D }
//  30 1E RS   62 3E >      94 5E ^     126 7E ~
```

```
//  31 1F US   63 3F ?      95 5F _     127 7F DEL
```

Refs:

ASC(), MID$()

CLOSEFILE()

Siehe OPENFILE

Refs:

OPENFILE()

CONTINUE

CONTINUE

Setzt die aktuelle FOR oder WHILE Schleife fort, als ob NEXT oder WEND hier stünde.

Refs:

BREAK, FOR, WHILE

COPYFILE

COPYFILE quelle$, ziel$

Kopiert eine Datei von einem lokalen Datenträger auf einen anderen. Dateien aus Shoebox'en können nicht kopiert werden.

```
// Backup
f$ = FILEREQUEST$(TRUE, "*.*")
COPYFILE f$, f$+".bak"
END
```

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETFILE, KILLFILE, PUTFILE

COS()

.

Refs:

SIN()

DEBUG

DEBUG text#$

Schreibt einen Text auf die Ausgabekonsole des GLBasic Editors, falls das Programm im Debug Modus läuft. Bei Release (nicht-Debug) Versionen wird der Befehl gar nicht erst in's Programm kompiliert, so dass keine Geschwindigkeitseinbußen zu verzeichnen sind.

Um eine neue Zeile anzufangen muss man das Zeichen "\n" schreiben.

```
c$ = GETCURRENTDIR$()
DEBUG "Dir="+c$+"\n"
```

Refs:

PRINT

DEC

DEC x##, num#$
DEC x$$, num#$

Verringert den Wert der Variablen x## oder x$$ um 'num#$'. Das geht selbstverständlich auch mit Feld-Variablen.

```
a = 100
DIM b[5]
b[3] = 200
DEC a, 5
DEC b[3], 5
PRINT a, 100, 100
PRINT b[3], 100, 120
SHOWSCREEN
MOUSEWAIT
```

Refs:

INC

DELETE

DELETE item

Dieser Befehl funktioniert nur innerhalb von FOREACH/NEXT und wirft den Eintrag, auf den "item" zeigt aus dem Feld heraus. Direkt dannach wird die Schleife von oben mit dem nächsten Element neu durchlaufen.
Es ist also gleichbedeutend mit:

```
DIMDEL feld[], index
CONTINUE
```

Beispiel:

```
// Ein Feld von Zahlen
DIMDATA a[], 3,4,5,  12,13,  6

// Die Zahlen durchlaufen
FOREACH num IN a[]
   // Wirf Zahlen größer 10 raus, und continue
   IF num>10 THEN DELETE num
   a$=a$ + num + ", "
NEXT
PRINT a$, 0,0

SHOWSCREEN
MOUSEWAIT
```

Refs:

DIMDEL, FOREACH

DIM

DIM feld#$ [d1#] | [d2#] [d3#] ... [d8#]

Erstellt ein Speicherfeld für Zahlen / Wörter. Die einzelnen Felder werden über feld#$[index#] angesprochen. Das erste Element hat den Index 0.

Sample:

```
DIM feld[8][8]
FOR x=0 TO 7
    FOR y=0 TO 7
        feld [x][y] = x*100 + y
    NEXT
NEXT

PRINT feld [4][5], 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

DIMDATA, DIMDEL, DIMPUSH, REDIM

DIMDATA

DIMDATA feld#$[], daten1#$, daten2#$, ...

Ermöglicht ein Feld mit Werten zu erstellen. Der Befehl

```
DIMDATA feld[], 34, 45
```

ist identisch mit:

```
DIM feld[2]
feld[1]=34
feld[2]=45
```

Refs:

DIM, DIMDEL, REDIM

DIMDEL

DIMDEL feld#$[], index#$

Löscht aus dem 'feld' den Eintrag mit dem Index 'index#$' heraus. Alle darunterliegenden Einträge werden nach oben geschoben. Es ist dabei egal wie viele Dimensionen das Feld hat.

Zur Verdeutlichung. Dieses Feld:

```
DIM a[3][2]
a[0][0] =  0
a[0][1] =  1
a[1][0] = 10
a[1][1] = 11
a[2][0] = 20
a[2][1] = 21
DIMDEL a[], 1
```

sieht dannach so aus:

```
a[0][0] =  0
a[0][1] =  1
a[1][0] = 20
a[1][1] = 21
```

Ein Beispiel aus der Praxis. Hier werden per Knopfdruck Sterne erstellt und mit der rechten Maustaste gelöscht.

```
// StarMaker
// ------------------------
LOCAL mx, my, b1, b2, best
DIM star[0][2]
mousedown = FALSE


// Hauptschleife
WHILE TRUE
   MOUSESTATE mx, my, b1, b2
   // Mauspfeil
   PRINT "<=", mx, my
   // Klick und Loslassen
   IF b1=FALSE AND b2=FALSE THEN mousedown=FALSE
   IF mousedown
      b1=FALSE
      b2=FALSE
   ENDIF
   IF b1 OR b2 THEN mousedown=TRUE

   // Liks = neu, Rechts= entfernen
   IF b1 THEN AddStar(mx,my)
   IF b2 THEN DelStar(mx,my)

   // nächsten Stern suchen
   best = Nearest(mx, my)
```

```
   // Alle Sterne zeigen
   FOR i=0 TO BOUNDS(star[],0)-1
      // nächsten hervorheben
      IF best = i
         PRINT "*", star[i][0], star[i][1]
      ELSE
         PRINT "+", star[i][0], star[i][1]
      ENDIF
   NEXT
   SHOWSCREEN
WEND

// -----------------------------------
// DELSTAR
// Stern entfernen
// -----------------------------------
FUNCTION DelStar: x,y
LOCAL i
   i = Nearest(x,y)
   // Magic!
   IF i>=0 THEN DIMDEL star[], i
ENDFUNCTION

// -----------------------------------
// ADDSTAR
// Stern hinzufügen
// -----------------------------------
FUNCTION AddStar: x,y
LOCAL m
   m = BOUNDS(star[], 0)
   REDIM star[m+1][2]
   star[m][0] = x
   star[m][1] = y
ENDFUNCTION

// -----------------------------------
// NEAREST
// Findet den nachsten Stern zu x,y
// -----------------------------------
FUNCTION Nearest: x,y
LOCAL best, bestdist, dist, i, dx, dy
   best = -1
   FOR i=0 TO BOUNDS(star[],0)-1
      dx = x-star[i][0]
      dy = y-star[i][1]
      dist = SQR(dx*dx + dy*dy)
      IF i=0 OR dist<bestdist
         bestdist = dist
         best = i
      ENDIF
   NEXT
   RETURN best
ENDFUNCTION
```

Refs:

DELETE, DIM, REDIM

DIMPUSH

DIMPUSH feld[], eintrag

Dieser Befehl fügt einem eindimensionalen Feld ein neues Element unten hinzu. Er funktioniert auch mit Feldern von TYPEs.

Er macht nichts anderes als:

```
m=BOUNDS(feld[], 0)
REDIM feld[m+1]
feld[m] = eintrag
```

Refs:

DIM, DIMDATA, DIMDEL, REDIM

DOESFILEEXIST()

ok#=DOESFILEEXIST(file$)

Liefert zurück, ob eine Datei file$ existiert.

Sample:

```
ok=DOESFILEEXIST("test.txt")
IF ok
 PRINT "Test.txt existiert", 100, 100
ELSE
 PRINT "Test.txt nicht vorhanden", 100, 100
ENDIF
```

Refs:

COPYFILE, FILEREQUEST$(), GETCURRENTDIR$, GETFILE, PUTFILE

DRAWLINE

DRAWLINE sx#, sy#, ex#, ey#, farbe#

Zeichnet eine Linie in der angegebenen Farbe vom Punkt (sx#, sy#) aus bis (ex#, ey#).

Sample:

```
DRAWLINE 0,0, 100, 200, RGB(255, 0, 0) // Rot
SHOWSCREEN
MOUSEWAIT
```

Refs:

FILLRECT, SETPIXEL

ELSE

.

Refs:

IF

END

END

Beendet das Programm.

Sample:

```
END  // Alles weitere wird nicht ausgefürht
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

GOTO, RETURN

ENDFUNCTION

.

Refs:

FUNCTION

ENDIF

.

Refs:

IF

ENDINLINE

.

Refs:

INLINE

ENDOFFILE()

Siehe OPENFILE

Refs:

OPENFILE()

ENDPOLY

.

Refs:

POLYVECTOR

ENDSELECT

.

Refs:

SELECT

FILEREQUEST$()

file$=FILEREQUEST$(open#, filter$)

Blendet das Spiel-Fenster aus und öffnet einen Dateibrowser, mit dem man eine Datei auswählen kann. Diese wird dann in file$ gesichert.

open#: TRUE - Öffnen Dialog / FALSE - Speichern Dialog
filter$: Die Einträge im Filterdialog, getrennt durch '|' (oder-Zeichen)
Bsp:
"Texte *.txt|*.txt"
"Texte *.txt|*.txt|Alle *.*|*.*"

Sample:

```
file$=FILEREQUEST$(TRUE, "Text|*.txt|All|*.*")
PRINT file$, 0, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

DOESFILEEXIST(), GETCURRENTDIR$, GETFILE, PUTFILE

FILLRECT

FILLRECT sx#, sy#, ex#, ey#, farbe#

Zeichnet ein ausgefülltes Rechteck mit den Koordinaten (sx#, sy#) links oben und (ex#, ey#) rechts unten.
farbe# ist dabei ein RGB-Wert.

Sample:

```
FILLRECT 0, 100, 639, 120, RGB(0, 255, 255) //Türkis
SHOWSCREEN
MOUSEWAIT
```

Refs:

DRAWLINE, SETPIXEL

FINDPATH()

gefunden# = FINDPATH (karte#[], loesung#[], heuristik#, startx#, starty#, endex#, endey#)

Sucht im Feld karte#[] nach einem Pfad von startx#, starty# nach endex#, endey#. Dabei wird ein A* Algorithmus verwendet, der Wegkosten berücksichtigen kann. D.h. umso höher ein Wert in dem Feld karte[] ist, desto höher sind die Kosten für dieses Wegstück. Ein Wert von kleiner gleich 0 ist unpassierbar.
Mit dem Heuristikregler kann man festlegen, ob die kürzeste Route (=0.0) oder die Kostengünstigste Route (=1.0) gefunden werden soll. Zwischenwerte beschleunigen die Rechenzeit und verringern aber auch die Qualität der Lösung.
Der Rückgabewert ist die Länge des Pfads.
Wenn ein Pfad gefunden wurde, so enthält loesung#[] ein 2-dimensionales Feld, das im ersten index die Nummer des Schritts angibt und im 2. Index die x (=Wert 0) und y(=Wert 1) position des Schritts angibt.

Das Beispiel verdeutlicht die Anwendung:

```
// -------------------------------- //
// Project: PathFinder
// Start: Tuesday, September 23, 2003
// IDE Version: 1.30923

// Diesen Wert ändern für mehr/weniger Gelände
MAXX = 500


MAXY = MAXX/2

scalex = 640 / MAXX
scaley = 320 / MAXY

DIM map[MAXX][MAXY]
DIM solution[0]

// Irgend eine Karte machen
FOR x=0 TO MAXX-1; FOR y=0 TO MAXY-1; map[x][y]=100; NEXT; NEXT
FOR i=0 TO MAXX*MAXY*4; map[RND(MAXX-1)][RND(MAXY-1)] = RND(25); NEXT
FOR i=0 TO MAXY*.5; map[i+1][i] = 0; map[i+MAXY/2][MAXY-i-1]=0; NEXT

 // Go Pfadfinder, go!
 gut = FINDPATH(map[], solution[], .3,  0,0,  MAXX-1, MAXY-1)
 // Karte zeichnen
 FOR x=0 TO MAXX-1; FOR y=0 TO MAXY-1; FILLRECT x*scalex, y*scaley, x*scalex+scalex-1, y*scaley+scaley-
1, RGB(255*(1-map[x][y]), 255*map[x][y], 0); NEXT; NEXT

 IF gut
  PRINT "Loesung gefunden!", 0, 400
  x=0; y=0
  FOR i=0 TO BOUNDS(solution[], 0)-1
   xl = x
   yl = y
   x = solution[i][0]
   y = solution[i][1]
   DRAWLINE xl*scalex+scalex/2, yl*scaley+scaley/2, x*scalex+scalex/2, y*scaley+scalex/2, RGB(0, 0, 255)
  NEXT
 ELSE
  PRINT "Keine Loesung gefunden!?", 0, 400
 ENDIF
```

```
 SHOWSCREEN
 MOUSEWAIT
END
```

FOR

FOR zähler# = start# TO ende# |STEP schritt#|
...
NEXT

Durchläuft eine Programmschleife. Der Wert zähler# wird dabei beim ersten Durchlauf auf den Wert start# gesetzt. Nach jedem Durchlauf (beim Erreichen des NEXT) wird der Wert von zähler# um schritt# erhöht, bzw. bei negativer schritt# erniedrigt.
Ist STEP nicht angegeben, ist schritt# = 1.
Die Schleife wird so lange durchlaufen, bis zähler# nicht mehr innerhalb start# und erde# ist.

Sample:

```
FOR i=0 TO 10 STEP 2
    PRINT i, 100, 32*i
NEXT

SHOWSCREEN
MOUSEWAIT
```

Refs:

BREAK, CONTINUE, FOREACH, GOTO, WHILE

FORCEFEEDBACK

FORCEFEEDBACK nJoy#, dauer#, x_motor#, y_motor#

Aktiviert eine Kraft auf den ForceFeedback - Joystick nJoy#. Die dauer# wird in millisekunden angegeben. Ein neuer Aufruf von FORCEFEEDBACK löscht die aktuellen Einstellungen. motor# ist ein Wert von -1 bis 1, der angibt wie stark der Motor drücken soll. Hat das Gerät "nur" Rumble-Funktion sollte man beide Achsen auf "1" setzen, oder abwechselnd auf -1/+1 setzen - dann funktioniert's auch mit anderen Joysticks. Wobei ich dann keine Garantie übernehme, dass diese evtl. dadurch geschrottet werden...

```
// ------------------------------- //
// Project: ForceFeedback - Test
// IDE Version: 2.50107

LIMITFPS 50 // 20 ms / frame
id = 0
WHILE TRUE
   jx = GETJOYX(id)
   jy = GETJOYY(id)
   PRINT "+", 160, 160
   PRINT "X", jx*160+160, jy*160+160
   FORCEFEEDBACK id, 21, jx, jy
   SHOWSCREEN
WEND
```

Refs:

GETDIGI...(), GETJOY...(), JOYSTATE

FOREACH

FOREACH ref IN feld[]
...
NEXT

Mit dem FOREACH Befehl kann man schnell und einfach ein Feld durchlaufen und daraus Werte lesen und zurückschreiben.
Dabei wird "ref" je Schleifendurchlauf ein Zeiger (BYREF) auf das nächste Element im Feld zugewiesen.
Man muss "ref" nicht als LOCAL definieren. Das passiert automatisch. Jedoch ist "ref" nach dem zugehörigem "NEXT" nicht mehr gültig.

Mit anderen Befehlen ausgedrückt bedeutet FOREACH in etwa:

```
FOR i = 0 to BOUNDS(feld[],0)-1
```

```
   ref = feld[i]
   ...
   feld[i] = ref
NEXT
```

Beispiel:

```
// ------------------------------- //
// Project: ForEach

// Make an array of numbers
// Ein Feld von Zahlen
DIMDATA a[], 12,14,15,17

// Enumerate the numbers
// Die Zahlen durchlaufen
FOREACH num IN a[]
   a$=a$ + num + ", "
NEXT
PRINT a$, 0,0

// a simple TYPE
// ein einfacher TYPE
TYPE SHOP
   milk$
ENDTYPE

// An array-instance of it
// Eine Feld-Instanz davon
LOCAL kmart[] AS SHOP

// Fill with values
// Mit Werten füllen
DIM kmart[3]
   kmart[0].milk$ = "fat free"
   kmart[1].milk$ = "fatty"
   kmart[2].milk$ = "butter"

   a$=""
   // Enumerate all shops
   // Alle shops durchlaufen
   FOREACH item IN kmart[]
      a$ = a$ + item.milk$ + ", "
   NEXT
   PRINT a$, 0, 40


   SHOWSCREEN
   MOUSEWAIT
```

Refs:

DELETE, FOR

FORMAT$()

fmt$ = FORMAT$(numBuchstaben#, numNachKomma#, zahl#)

Wandelt eine Zahl in ein Wort um. Das Wort ist dabei mindestens numBuchstaben# lang (es werden evtl. Leerzeichen vorangestellt) und hat genau numNachKomma# Stellen nach dem Dezimalpunkt.

```
// ------------------------------- //
// Project: FORMAT$()
// Start: Tuesday, May 25, 2004
// IDE Version: 1.40525

FOR i=0 TO 20
   num = RND(100000)/RND(10000)
   PRINT num, 0, i*20
   PRINT FORMAT$(6, 2, num), 300, i*20
NEXT
SHOWSCREEN
MOUSEWAIT
```

Refs:

ASC(), CHR$(), PRINT

FUNCTION

FUNCTION name#$: arg1#$, arg2#$...
...
ENDFUNCTION

Mit FUNCTION erstellt man Funktionsdefinitonen. name#$ ist der Name der Funktion. argx#$ sind die Argumente, die die Funktion als Übergabeparameter benötigt. Diese Variablen sind im Bereich der Funktionsdefinition LOKAL.
Der Rückgabewert der Funktion wird duch den Parameter des Befehls: RETURN angegeben.

Ein Funktionsaufruf geschieht durch den Namen der Funktion mit einer Klammer:
a=name(3,4)
Eine Funktion muss nicht immer einen Wert zurückgeben. Das erreicht man, indem man nur RETURN ohne Wert verwendet. Der Rückgabewert wird dann '0' sein.

Sample:

```
// Function

  a$=Right$("123456",  2); // 56
  PRINT a$, 100, 80

LOCAL a
  a=7; b=12; GLOBAL a=15
  c=Max(a, b) // 7, 12
  Center("c="+c, 100) // 12
  Center ("a="+a, 140) // 7
  Center("b="+b, 180) // 12
  Center("GLOBAL a="+GLOBAL a, 220) // 0 (function: Max)
  SHOWSCREEN
MOUSEWAIT

// ------------------------------------------------------------ //
// -=#  MAX - größere Zahl(a, b) / bigger number(a, b) #=-
// ------------------------------------------------------------ //
FUNCTION Max: a, b
  // Diese Variablen sind als LOCAL definiert:
  // a, b
LOCAL ret
  IF a>b
    ret=a
  ELSE
    ret=b
  ENDIF
  // Das hat keinen Einfluss auf die globalen Variablen
  a=0
  // Das schonl
  GLOBAL a=0
  RETURN ret
ENDFUNCTION // MAX

// ------------------------------------------------------------ //
// -=#  CENTER  #=-
// ------------------------------------------------------------ //
FUNCTION Center: text$, y
  // Diese Variablen sind als LOCAL definiert:
  // text$, y
  PRINT text$, (640-LEN(text$)*16)/2, y
ENDFUNCTION // CENTER


// Eine Funktion mit Wort als Rückgabewert
// ------------------------------------------------------------ //
// -=#  RIGHT$  #=-
// ------------------------------------------------------------ //
FUNCTION Right$: word$, size
  // Diese Variablen sind als LOCAL definiert:
  // These values are defined LOCAL:
  // word$, size
  RETURN MID$(word$, LEN(word$)-size, size)
ENDFUNCTION // RIGHT$
```

Refs:

GOSUB

GETCOMMANDLINE$()

dat$ = GETCOMMANDLINE$()

Gibt die Parameter der Kommandozeile zurück.

```
// ------------------------------- //
// Project: GETCOMMANDLINE$()
// Start: Friday, March 28, 2003
// Compiler Version: 1.30328


 line$ = GETCOMMANDLINE$()
 PRINT line$, 0, 0

 // Argumente zerstückeln / Split arguments
DIM args$[100]
 lastgood=0
 isgood=FALSE
 line$=" "+line$+" "
 FOR p=0 TO LEN(line$)
  c$ = MID$(line$, p, 1)
  IF isgood
   IF c$=" "
    args$[narg]=MID$(line$, lastgood, p-lastgood)
    narg=narg+1
    isgood=FALSE
   ENDIF
  ELSE
   IF c$<>" "
    lastgood=p
    isgood=TRUE
   ENDIF
  ENDIF
 NEXT

 // Argumente Zeigen / show arguments
 FOR i=0 TO narg-1
  PRINT "'" + args$[i] +"'", 0, 50+16*i
 NEXT
 SHOWSCREEN
 MOUSEWAIT
```

GETCURRENTDIR$

dir$=GETCURRENTDIR$()

Liefert das aktuelle Verzeichnis in dem sich die .exe Datei befindet.

Sample:

```
dir$=GETCURRENTDIR$()
PRINT dir$, 100, 100
SHOWSCREEN; MOUSEWAIT
```

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETFILE, GETFILELIST(), PUTFILE

GETDIGI...()

.

Refs:

GETJOY...(), JOYSTATE

GETFILE

GETFILE file$, lin#, dat#$ | PUTFILE file$, lin#, dat#$

Liest (GET) oder schreibt (PUT) den Wert der Variablen dat#$ aus/in die Datei file$ an die Zeile Nr. lin#. Die erste Zeile hat den Index 0. Es sind maximal 256 Zeilen möglich.

Der Befehl ist veraltet. Es wird OPENFILE empfohlen.

Sample:

```
a$="NAME"
PUTFILE "Test.txt", 0, s$
a$=""
GETFILE "Test.txt", 0, a$
PRINT a$, 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETCURRENTDIR$, GETFILELIST(), OPENFILE()

GETFILELIST()

num = GETFILELIST(find$, files$[])

Erstellt eine Liste mit den Datei- und Verzeichnisnamen im aktuellen Verzeichnis.
find$ ist dabei eine Zeichenkette mit Platzhaltern * und ?, wobei * eine beliebige Anzahl von Zeichen repräsentiert, und ? nur genau ein beliebiges Zeichen darstellt.

?aus.bmp findet z.B.
Maus.bmp
Haus.bmp
Laus.bmp
Aus.bmp

num ist dabei nVerzeichnisse * 0x1000 + nDateien. Also im oberern und im unteren WORD gespeichert. Das Beispiel verdeutlicht, wie man beide Zahlen errechnen kann.

Sample:

```
// -------------------------------- //
// Project: Files and Folders
// Start: Tuesday, November 25, 2003
// IDE Version: 1.31125

 LOADFONT "minifont.bmp", 0
 GETFONTSIZE font_x, font_y

 ok = SETCURRENTDIR("..") // One up
 cur$ = GETCURRENTDIR$()
 num = GETFILELIST("*.*", files$[])
 num_dir  = INTEGER(num / 0x10000) // Hi-Word
 num_file = MOD(num, 0x10000)      // Lo-Word
 PRINT cur$, 0, 0
 PRINT "ok? " + ok+ " num: " + num + " -> nDir: "+num_dir+" nFile: "+num_file, 0, font_y
 FOR i=0 TO BOUNDS(files$[], 0)-1 // BOUNDS(files$[], 0)-1 = num = num_dir+num_file
  PRINT files$[i], 0, (i+3)*font_y
 NEXT

 SHOWSCREEN
 MOUSEWAIT
```

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETCOMMANDLINE$(), GETCURRENTDIR$, SETCURRENTDIR(), SHELLCMD()

GETFILESIZE()

bytes# = GETFILESIZE(file$)

Liefert die Größe der Datei file$ in Bytes.
1024B = 1KB
1024*1024B = 1MB

GETFONTSIZE

GETFONTSIZE sx##, sy##

Ein Befehl, mit dem die Buchstaben-Größe des aktuell verwendeten Fonts ermittelt werden kann.

Sample:

```
FILLRECT 300, 100, 500, 400, RGB(20,20,120)
BOXPRINT ("Dies ist ein langer Text, der umgebrochen wird", 300, 100, 200)
SHOWSCREEN
MOUSEWAIT


// ------------------------------------------------------------ //
// -=#  BOXPRINT  #=-
// ------------------------------------------------------------ //
FUNCTION BOXPRINT: str$, x, y, wx
 // Diese Variablen sind als LOCAL definiert:
  // str$, x, y, wx, void

 LOCAL tx, ty // Width of one character
 LOCAL cx, cy, word$, c$, cpos

  GETFONTSIZE tx, ty

 cy=y
 str$=str$+" "
 cx=x
 WHILE cpos < LEN(str$)
  MIDSTR str$, c$, cpos, 1

  word$=word$+c$

  IF c$=" "
   IF cx-x+(LEN(word$)-1)*tx > wx
    cx=x; cy=cy+ty
   ENDIF
   PRINT word$, cx, cy
   cx=cx+ LEN(word$)*tx
   word$=""
  ENDIF
  cpos=cpos+1
 WEND
ENDFUNCTION void // BOXPRINT
```

Refs:

GETSCREENSIZE, GETSPRITESIZE, LOADFONT, PRINT, SETFONT

GETJOY...()

n# = GETNUMJOYSTICKS( )
n$ = GETJOYNAME$(n#)
x# = GETJOYX(n#)
y# = GETJOYY(n#)
z# = GETJOYZ(n#) // Throttle
rx# = GETJOYRX(n#)
ry# = GETJOYRY(n#)
rz# = GETJOYRZ(n#)
b# = GETJOYBUTTON(n#, #m)
dx# = GETDIGIX(n#) // Hat switch
dy# = GETDIGIY(n#)

Es lassen sich mit diesen Befehlen bis zu 10 Joysticks (n#) mit je 32 Knöpfen (m#) abfragen. Diese Befehle funktionieren nur mit der Vollversion von GLBasic.

Sample:

```
// ------------------------------------------------------------ //
// JOYSTICK DEMO II
// ------------------------------------------------------------ //
```

```
LOADFONT "minifont.bmp"

WHILE TRUE
 FOR i=0 TO GETNUMJOYSTICKS()-1 // Anzahl der Joysticks
  PRINT GETJOYNAME$(i), 0, i*48 // Name des Geräts
  a$="X:"+GETJOYX(i) + " Y:"+GETJOYY(i)+ " Z:"+ GETJOYZ(i) // XYZ-Position
  a$=a$+"Rx:"+GETJOYRX(i) + " Ry:"+GETJOYRY(i)+ " Rz:"+GETJOYRZ(i) // XYZ-Rotation
  FOR n=0 TO 31 // Bis zu 32 Knöpfe
   a$=a$+" B"+n+":"+GETJOYBUTTON (i, n)
  NEXT
  PRINT a$, 0, i*48+16

  // DIGI-Joystick
  b$="DX:"+ GETDIGIX(i)+" DY:"+GETDIGIY(i)
  PRINT b$, 0, i*48+32
 NEXT
 SHOWSCREEN
WEND
```

Refs:

GETDIGI...(), JOYSTATE, KEY(), MOUSESTATE, SETMOUSE, MOUSEWAIT, KEYWAIT

GETPIXEL()

col# = GETPIXEL(x#, y#)

Liefert einen Farbwert für den Pixel an der Stelle x#, y#.
Achtung: Die Farbe kann bis zu 13% vom Wert, den man evtl. vorher gesetzt hat abweichen. Besonders bei 16 Bit Farbtiefe sind Abweichungen vorprogrammiert!

Achtung: Dieser Befehl muss Daten aus dem AGP Speicher der Grafikkarte laden und ist darum langsamer als erlaubt.

```
// -------------------------------- //
// Project: GetPixel
// IDE Version: 2.50316

LOADSPRITE "bubble.bmp", 0
WHILE TRUE
   MOUSESTATE mx, my, b1, b2
   SPRITE 0, 0, 0
   FOR x=0 TO 31
      FOR y=0 TO 31
         FILLRECT x*8+160, y*8, x*8+167, y*8+7, GETPIXEL(x+mx,y+my)
      NEXT
   NEXT
   ALPHAMODE .1
   FILLRECT mx, my, mx+31, my+31, RGB(255,255,255)
   ALPHAMODE 0
   SHOWSCREEN
WEND
```

Refs:

RGB(), SETPIXEL

GETSCREENSIZE

GETSCREENSIZE sx##, sy##

Ermittelt die unter Projektoptionen eingestellte Auflösung des Bildschirms. Damit kann man z.B. für verschiedene Plattformen die Spielskalierung anpassen.

```
// GETSCREENSIZE

GETSCREENSIZE sx, sy
PRINT "Screen: "+sx + " : " + sy, 0,0
SHOWSCREEN
MOUSEWAIT
```

Refs:

GETFONTSIZE, GETSPRITESIZE

GETSPRITESIZE

GETSPRITESIZE id#, sx##, sy##

Liefert die Breite/Höhe des Sprites id# in den Koordinaten sx# und sy#.

```
GRABSPRITE 13, 15, 19
GETSPRITESIZE 13, sx, sy
PRINT "Sprite: "+sx + " : " + sy, 0,0
SHOWSCREEN
MOUSEWAIT
```

Refs:

GETFONTSIZE, GETSCREENSIZE

GETTIMER()

n#=GETTIMER()

Gibt die Zeit seit letztem Aufruf von SHOWSCREEN in 1/1000 sec zurück.

Sample:

```
// FPS Zähler
WHILE TRUE
 dtime = GETTIMER()
 fps = ((1000/dtime)+fps)/2
 delay=delay+dtime
 IF delay>500 // 1/2 sec
  delay=0
  showfps=fps
 ENDIF
 PRINT "FPS: "+showfps+" dtime:"+dtime, 0,0
 SHOWSCREEN
WEND
```

Refs:

GETTIMERALL(), LIMITFPS

GETTIMERALL()

zeit# = GETTIMERALL

Gibt die Zeit seit dem Programmstart in 1/1000 sek zurück.

```
// ------------------------------- //
// Project: GETTIMERALL - Demo
// Start: Friday, March 12, 2004
// IDE Version: 1.40304

 LIMITFPS -1
 WHILE TRUE
  TextScroller("Hello World")
  SHOWSCREEN
 WEND

// ------------------------------------------------------------ //
// -=#  TEXTSCROLLER  #=-
// ------------------------------------------------------------ //
FUNCTION TextScroller: text$
 // Diese Variablen sind als LOCAL definiert:
 // These variables are defines as LOCAL:
 // text$

LOCAL fx, fy, xoffset, screenx
 screenx = 640 // Screen width / Bildschirmbreite
 GETFONTSIZE fx, fy
 xoffset = -GETTIMERALL()/10 // X-position of text / X-Stelle des Textes

 // Modulo the offset, so that text comes from the right after it completely
 // dissapeared left.
 // Modulo operation auf Offset anwenden, so dass der text von rechts
```

```
 // kommt, nachdem er komplett links verschwunden ist.
 xoffset= MOD(xoffset, screenx + LEN(text$)*fx) + screenx
 PRINT text$, xoffset, 100 - ABS(COS(GETTIMERALL()/5)) * 100
ENDFUNCTION
```

Refs:

GETTIMER(), LIMITFPS

GLOBAL

.

Refs:

BYREF, LOCAL, STATIC

GOSUB

GOSUB mysub

SUB mysub:
RETURN
ENDSUB

Mit GOSUB springt man in ein Unterprogramm.
Definition einer SUB mit SUB marke: (Doppelpunkt nicht vergessen). Wieder zurück aus einer SUB mit ENDSUB.
SUBs müssen hinter dem Ende des Hauptprogrammes stehen. Zwischen ENDSUB und SUB darf kein Befehl stehen.
Mit RETURN kommt man aus einer SUB wieder zurück hinter den Punkt, an dem GOSUB aufgerufen wurde. Ruft man RETURN im Hauptprogramm auf, so wird das Programm beendet.
In einer SUB darf natürlich ein weiteres GOSUB verwendet werden (Schachtelung).
Unter dem Menüpunkt Projekt/neue SUB kann sehr einfach und übersichtlich eine neue SUB erstellt werden.

Sample:

```
GOSUB draw
SHOWSCREEN
MOUSEWAIT
END

SUB draw:
   PRINT "Eine SUB" COLOR="white">, 100, 100
    RETURN // Zurückspringen zum GOSUB
    PRINT "Das sieht man nie!", 100, 150
ENDSUB
```

Refs:

END, FUNCTION, ENDFUNCTION, GOTO

GOTO

marke:
GOTO marke

Springt zur Sprungmarke 'marke'.
Kann nicht von SUBs nach außerhalb, in andere SUBs und vom Hauptprogramm in eine SUB angewandt werden.
Sprungmarken werden mit ':' (Doppelpunkt) gekennzeichnet.

Sample:

```
PRINT "Das erscheint", 100, 100
GOTO shortcut
PRINT "Such mich", 100, 150 // Das wird nie erscheinen
shortcut:
SHOWSCREN
MOUSEWAIT
```

Refs:

END, FUNCTION, GOSUB

GRABSPRITE

GRABSPRITE nr#, x#, y#, breite#, hoehe#

Erstellt aus dem Hintergrund-Puffer ein Sprite. Dieser Befehl ist sehr nützlich für z.B. Spiegeleffekte oder zum dynamischen Erstellen von Texturen von 3D Objekten.

Sample:

```
FOR i=0 TO 256
 DRAWLINE 0,i,256,i,RGB(0, i, 255-i)
NEXT

GRABSPRITE 0, 0,0,7, 256

BLACKSCREEN
SPRITE 0, 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

LOADBMP, LOADSPRITE, SAVESPRITE, SPRITE, X_OBJ..., X_SETTEXTURE

HIBERNATE

HIBERNATE

Schaltet das Programm in einen Schlaf-modus, bis der Benutzer irgend etwas tut (Maus bewegen, Taste drücken, Joystick anfassen, ...) Dadurch lassen sich die CPU-Resourcen drastisch reduzieren wenn man z.B. einen Leveleditor schreibt. Es ermöglicht Laptops und PocketPC Geräten einen längere Akkulebensdauer.

```
// -------------------------------- //
// Project: Hibernate
// Start: Tuesday, May 04, 2004
// IDE Version: 1.40430
WHILE TRUE
→a$=""
→FOR i=0 TO 255
→→IF KEY(i) THEN a$=a$+i+", "
→NEXT
→MOUSESTATE mx, my, b1, b2
→JOYSTATE jx,jy,jb1,jb2
→PRINT a$, 0,0
→PRINT "Mx:"+mx+" My:"+my+" b1:"+b1+" b2:"+b2, 0,40
→PRINT "Jx:"+jx+" Jy:"+jy+" b1:"+jb1+" b2:"+jb2, 0,80
→SHOWSCREEN
→HIBERNATE
WEND
```

Refs:

SHOWSCREEN

HUSH

HUSH

Stoppt alle Sounds (nicht Musik) sofort.

Sample:

```
LOADSOUND "Super.wav", 0
PRINT "STOP", 100, 100
PLAYSOUND 0, 0
SHOWSCREEN
MOUSEWAIT
HUSH
PRINT "OK", 100, 100
SHOWSCREEN
```

```
MOUSEWAIT
```

Refs:

LOADSOUND, PLAYMUSIC, STOPMUSIC, PLAYSOUND()

IF

IF a#$ = < <= > >= <>

b#$
...
| ELSE
| ...
ENDIF

oder:

IF a#$ = < <= > >= <> b#$ THEN ...

Führt den Code zwischen IF und ENDIF nur aus, wenn der Vergleich von a#$ und b#$ WAHR ist. Ansonsten wird der Code zwischen ELSE und ENDIF ausgeführt wenn ELSE vorhanden ist.
Mit dem Befehl THEN kann nur ein einzelner Befehl ausgeführt werden.

Sample:

```
a=5; b=3

IF a < b
    PRINT "a < b", 100, 100
ELSE
    PRINT "a < b ist FALSE" COLOR="white">, 100, 100
ENDIF

IF a <> b THEN PRINT "a ist nicht gleich b", 100, 150

SHOWSCREEN
MOUSEWAIT
```

Refs:

AND, OR, FOR, GOTO, WHILE

INC

INC x##, num#$
INC x$$, num#$

Erhöht den Wert der Variablen x## oder x$$ um 'num#$'. Das geht selbstverständlich auch mit Feld-Variablen.

```
a = 100
DIM b[5]
b[3] = 200
INC a, 5
INC b[3], 5
PRINT a, 100, 100
PRINT b[3], 100, 120
SHOWSCREEN
MOUSEWAIT
```

INIGET$

wert$ = INIGET$(sektion$, schluessel$)

Holt den Wert von "schluessel$" der Sektion "sektion$". Wenn entweder sektion$ oder schluessel$ nicht vorhanden sind, wird "NO_DATA" zurückgegeben.

Bei sektion$ und schluessel$ spielt Groß-/Kleinschreibung keine Rolle.

Für weitere Informationen bitte bei INIOPEN nachsehen.

Refs:

INIOPEN

INIOPEN

INIOPEN dateiname$

Öffnet eine .ini Datei. Wenn vorher eine .ini Datei geöffnet und verändert wurde (INIPUT), wird diese zuvor gespeichert.

Eine .ini Datei hat folgende Syntax:

```
; Kommentar
[Sektion1]
Schluessel1 = Wert Zeichenkette
Schluessel2 = Wert2

; Kommentar
[Sektion2]
...
```

Beispiel:

```
// zur Sicherheit löschen
KILLFILE "test.ini"

INIOPEN "test.ini"

color$ = "Colors"
shape$ = "Shapes"

INIPUT color$, "red", 1
INIPUT color$, "green", 2
INIPUT shape$, "Circle", 1
INIPUT shape$, "Poly", "fine"

// schreibt "test.ini" auf die Platte
INIOPEN ""

// Jetzt die Datei neu einlesen
INIOPEN "test.ini"
PRINT "Red  = " + INIGET$(color$, "rEd"), 0, 0
PRINT "Poly = " + INIGET$(shape$, "polY"), 0, 20

PRINT "nach dem loeschen: ", 0, 50

// Color Sektion entfernen
INIPUT color$, "", ""
// Poly Schlüssel entfernen
INIPUT shape$, "pOLy", ""

// nochmals anzeigen
PRINT "Red  = " + INIGET$(color$, "Red"), 0, 70
PRINT "Poly = " + INIGET$(shape$, "PoLy"), 0, 90

SHOWSCREEN
MOUSEWAIT
```

Refs:

GETFILE, INIGET$, INIPUT, PUTFILE

INIPUT

INIPUT sektion$, schluessel$, wert$

Schreibt einen Wert in eine geöffnete .ini Datei. Wenn wert$="", so wird der schluessel$ herausgenommen. Zum Löschen ganzer Sektionen muss man schluessel$="" übergeben.

Mehr Information unter INIOPEN.

Refs:

INIOPEN

INKEY$

a$=INKEY$( )

Liefert nicht den Tasten-Code, sondern den eingegebenen Buchstaben zurück.
Langsamer als KEY( ), aber nützlich für eigene INPUT Funktionen.

Sample:

```
// ------------------------------------------------------------ //
// INPUT via INKEY$ Mit Sternenhintergrund
// ------------------------------------------------------------ //

// Sternfeld vorbereiten
DIM spd[640]
DIM sy[640]
 FOR x=0 TO 639
  spd[x]=RND(3)+1
  sy[x]=-RND(480)
 NEXT

 name$=""
 WHILE TRUE
  IF KEY(14)
   MIDSTR name$, name$, 0, LEN(name$)-1 // Backspace
  ELSE
   in$=INKEY$()
   IF in$<>"" THEN name$=name$+in$
  ENDIF
  PRINT name$, 0, 100
  GOSUB ShowStarfield
  SHOWSCREEN
 WEND
END


// ------------------------------------------------------------ //
// -=#  SHOWSTARFIELD  #=-
// ------------------------------------------------------------ //
SUB ShowStarfield:
LOCAL x
 FOR x=0 TO 639
  c=spd[x]
  y=sy[x]
  y=y+c
  IF y>480 THEN y=0
  sy[x]=y
  c=c*64 -1
  SETPIXEL x, y, RGB(c, c, c)
 NEXT
ENDSUB // SHOWSTARFIELD
```

Refs:

INPUT, KEY(), KEYWAIT

INLINE

INLINE
C++ commandos
ENDINLINE

Erlaubt C++ code mitten im GLBasic Programm.
Achtung: Das ist nur für Benutzer gedacht, die exakt wissen was sie machen möchten! Ab hier ist Schluss mit "einfachste Programmiersprache der Welt"!

- Die Zahlen haben den Datentyp "DGInt". Das ist unter PC ein 64bit double, beim PocketPC ein 32bit float.
- Strings sind vom Typ "DGStr". Zeichneketten sind immer 8bit ascii. Unicode-Konvertierung erfolg mittels GLBasic Befehlen.
- Ein '$' ist in C++ "_Str". Also a$ wird a_Str.
- Einen const char* von einem DGStr bekommt man mit my_Str.c_str();

- Einen char* mit mindestens 'x' bytes Länge bekommt man mit: my_Str.GetStrData(x);
- Alle GLBasic Funktionen haben eine Parameterliste: z.B. MOUSEWAIT( );
- Ein Zahlenfeld hat den Typ "DGIntArray".
- Ein Stringfeld ist vom Typ "DGArray". DGArray ist eine Template Klasse für DIM/REDIM und benötigt einen ( ) und einen = operator.
- Auf Felder greift man statt a[x][y] mit a(x, y) zu.
- es ist _nichts_ definiert! Bitte nötige Funktionen deklarieren.
- INLINE startet im namespace __GLBASIC__. Beim Ändern bitte wiederherstellen!
- GLBASIC_HWND() liefert das HWND des GLBasic Hauptfensters.
- get_sprite_texture(id) liefert die Texture-ID für glBindTexture.

DLL - Funktionen ansprechen

Die wohl am häufigsten genutze Funktionalität is das laden von Funktionen aus DLLs zur Verwendung mit GLBasic. Dazu wurden einige angenehme Funktionen hinzugefügt:

Von Hand laden:

```
// Lädt die Funktion "foo" aus my.dll in den Zeiger: foo
void (__stdcall* foo)(int, char); // Deklaration
void LoadDllFunction()
{
   DLLCALL("my.dll", "foo", (void**) &foo);
}
```

Automatisch laden lassen:

```
// ausserhalb von Funktionen:
// DECLARE(name, "dll", (parameter), rückgabetyp)
DECLARE(foo, "my.dll", (int, char), void);

// DECLARE_ALIAS(name, "dll", "echter_name", (parameter), rückgabewert)
DECLARE_ALIAS(foo, "my.dll", "foo@4", (int, char), void);
```

Die Parameter müssen in Runden Klammern stehen. Es kann sein, dass der Funktionsname in der DLL von dem Prototypen abweicht. Dafür kann man DECLARE_ALIAS verwenden.

Wenn die Funktion eine __cdecl Funktion ist (also kein __stdcall), hängt man an DECLARE ein _C an:

```
DECLARE_C(foo, "my.dll", (int, char), void);
DECLARE_C_ALIAS(foo, "my.dll", "foo@4", (int, char), void);
```

Die DLL und die Funktion wird beim Programmstart geladen und beim Beenden wieder frei gegeben. Man sollte untersuchen, ob die Funktion geladen werden konnte:

```
if(foo)
{
   // alles im grünen Bereich
}

// INLINE

LOCAL C
C = 41
INLINE
   C++;
ENDINLINE

PRINT "C="+C, 100, 100
```

INPUT

INPUT in#$, x#, y#

Wartet mit blinkendem Cursor auf eine Benutzereingabe. Als Hintergrund wird der aktuelle Back-Buffer verwendet.

Sample:

```
PRINT "Name:", 100, 100
INPUT name$, 100, 150
BLACKSCREEN
PRINT "Name: " + name$, 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

INKEY$, KEY(), KEYWAIT

INSTR()

pos = INSTR(wort$, such$)

Liefert die erste Position, an der such$ in wort$ gefunden wurde (erstes Zeichen = index 0). Wird such$ nicht in wort$ gefunden, wird -1 zurückgegeben.

```
pos = INSTR("Dream Design", "sign")
PRINT pos, 0,0
SHOWSCREEN
MOUSEWAIT
```

Refs:

MID$(), REPLACE$(), SPLITSTR()

INTEGER()

a#=INTEGER(b#)

Schneidet die Nachkommastellen einer Zahl ab.

```
// INTEGER() Demo

FOR i = 0 TO 1 STEP .1
 PRINT i + " - "INTEGER(i), 0, i*200
NEXT
SHOWSCREEN
MOUSEWAIT
```

ISFULLSCREEN()

vb#$ = ISFULLSCREEN()

Liefert, ob das Spiel im Vollbildmodus läuft(TRUE) oder im Fenster (FALSE).

Refs:

SETSCREEN

JOYSTATE

JOYSTATE jx##, jy##, jba##, jbb##

Setzt die Werte jx# und jy# auf die aktuelle Stellung der Joystickachsen. jba# und jbb# sind die Knöpfe A und B. (1 = Gedrückt, 0=Nicht gedrückt)
-1 = Links / Auf
0 = Keine Bewegung
+1 = Rechts / Ab

Sample:

```
// Ende mit ESC
WHILE TRUE // Endlos-Schleife
    a$="JX:" + jx + " JY:" + jy + " KnopfA:" + ba + " KnopfB" +jb
    PRINT a$, 10, 100
    SHOWSCREEN
WEND
```

Refs:

GETJOY...(), GETDIGI...(), KEY(), MOUSESTATE

KEY()

a=KEY(code#)

Gibt 1 zurück, wenn Taste mit dem Code 'code#' gedrückt ist, sonst 0.
Um Codes für alle Tasten zu erhalten, bitte unter den Tools suchen nach dem Programm 'Keycode.exe'

Folgende Tasten werden für PocketPC/Smartphones verwendet:

```
// D-Pad (any device)
   Key PC-Key
--------------
<- 203 Cursors
-> 205
^  200
V  208
o   28 Return

// Application_Buttons (PocketPC)

Key    iPAQ3600 PC-Key
----------------------
199    calender home
207    mail     end
210    Q        insert
211    ->       del
157,29 audiorec ctrl


Num-Pad (Smartphone)

SMPH   | KEY-Code
-------+--------------------
       | PC-NUM     PC-Main
1 2 3  | 71 72 73   2   3  4 (Nummern)
4 5 6  | 75 76 77   5   6  7
7 8 9  | 79 89 81   8   9 10
* 0 #  | 55 81 83   27 11 43 (+ 0 #)

SMPH   Key PC-Key
-----------------
dial   211 del
hangup 207 end
home   199 home
back     1 esc


Application Buttons - SmartPhone

SMPH Key PC-Key
---------------
(@t)  67 F9
prev  68 F10
play  87 F11
next  88 F12

Application Buttons - GP2X

GP2X     Key(s)           PC-Key
-----------------------------------------
DPad     203,205,200,208,57  Cursor Keys/Space
Volume   201,209             PageDn/PageUp
Front    42, 54              LShift/RShift
Start    28                  Enter
Select   15                  Tab
A,B,X,Y  44,45,29,58         A,D,S,W
         30,32,31,17         Y,X,LCtr,CapsLock
Vol +&-  01                  ESC
```

Sample:

```
// Beenden mit ESC
WHILE TRUE // Endlos-Schleife
    IF KEY(57) = 1; PRINT "Leertaste gedrückt", 100, 100; ENDIF
    SHOWSCREEN
WEND
```

Refs:

ALLOWESCAPE, INKEY$, INPUT, JOYSTATE, KEYWAIT, MOUSESTATE, MOUSEWAIT

KEYWAIT

KEYWAIT

Wartet darauf, dass eine Taste gerückt wird.

```
PRINT "Test", 100, 100
SHOWSCREEN
KEYWAIT
PRINT "OK", 100, 100
KEYWAIT
END
```

Refs:

INKEY$, INPUT, KEY(), MOUSEWAIT

KILLFILE

KILLFILE dateiname$

Löscht eine Datei von der Festplatte.

```
// Datei anlegen
PUTFILE "test.txt", 0, "test"
// und löschen
KILLFILE "test.txt"

// bischen dummes Beispiel ;)
```

Refs:

COPYFILE, DOESFILEEXIST(), FILEREQUEST$(), GETCURRENTDIR$, GETFILELIST(), SHELLCMD()

LCASE$()

.

Refs:

UCASE$()

LEN

len# = LEN(wort$)

Gibt die Länge eines Wortes (Anzahl der Buchstaben) zurück.

Sample:

```
a$="Super"
b=LEN(a$)
PRINT b, 100, 100 // Schreibt '5'
SHOWSCREEN
MOUSEWAIT
```

Refs:

INSTR(), MID$(), SPLITSTR()

LET

LET a#$ = b#$

Weist dem Wert a#$ den Wert von b#$ zu. Worte werden in Zahlen und umgekehrt umgewandelt. Dieser Befehl kann auch weggelassen werden.

Sample:

```
LET a=5   // Oder einfach a=5
LET b=a+3 // Oder einfach b=a+3
PRINT a, 100, 100
PRINT b, 100, 150
SHOWSCREEN
MOUSEWAIT
```

Refs:

IF, PRINT

LIMITFPS

LIMITFPS n#

Legt die maximale Bildwiederholfrequenz des Spiels fest (in Bildern pro Sekunde).
-1 = Maximale Geschwindigkeit

Sample:

```
LIMITFPS 12
FOR i=0 TO 640
 PRINT "*", i, 100
  SHOWSCREEN
NEXT
```

Refs:

SHOWSCREEN

LOADBMP

LOADBMP bmp$

Lädt die Datei bmp$ und benutzt diese beim nächsten SHOWSCREEN als Hintergrund für den Grafikbildschirm.

Sample:

```
LOADBMP "Bild.bmp"
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

BLACKSCREEN, BLENDSCREEN, SAVEBMP

LOADBUMPTEXTURE

LOADBUMPTEXTURE datei$, num#

Lädt eine Graustufengrafik als Höhen-Map (weiß ist hoch, schwarz=tief) und konvertiert sie in eine Normalen-Vektoren Map. Diese ist nötig, wenn man Dot3 Bump Mapping verwenden möchte. Die geladene Normalen-Textur kann bei X_SETTEXTURE3D als 2. Parameter angegeben werden.

Sample:

```
// Bump Mapping Demo
// -----------------

// Bild-Daten
LOADSPRITE      "image.bmp",0
```

```
// Wie LOADSPRITE, jedoch wird aus einer Heightmap eine Normalen-Map erstellt
LOADBUMPTEXTURE "bump.bmp", 1

// Ein einfaches Quadrat
X_OBJSTART 1
 X_OBJADDVERTEX   -20, -20, 0,  0,1, RGB(255,255,255)
 X_OBJADDVERTEX   -20,  20, 0,  0,0, RGB(255,255,255)
 X_OBJADDVERTEX    20, -20, 0,  1,1, RGB(255,255,255)
 X_OBJADDVERTEX    20,  20, 0,  1,0, RGB(255,255,255)
X_OBJEND

// Hauptschleife
WHILE TRUE
 phi=phi+GETTIMER()/20; IF phi>=360 THEN phi=phi-360
 X_MAKE3D 1, 500, 45
 X_CAMERA 0, 10, 100, 0,0,0
 // Licht Nr. -1 is Bump-Licht Postition
 X_SPOT_LT -1, 0, 0,0,50, 0,0,0,90
 X_SETTEXTURE 0, 1 // 0=Textur, 1=Bump-normal-map
 X_ROTATION phi, 0, 1, 0
 X_DRAWOBJ 1, 0
 SHOWSCREEN
WEND
```

Refs:

LOADSPRITE, X_MAKE3D, X_OBJ..., X_SETTEXTURE, X_SPOT_LT

LOADFONT

LOADFONT bmp$, num#
SETFONT num#

Lädt die Datei bmp$ als aktuellen Zeichensatz für die Befehle PRINT und INPUT in einen Schriftpuffer. Ein bmp-Font muss die Abmessungen 16*x und 8*y Pixel haben. Das Tool GLBasicFont erstellt aus TrueTypeFonts GLBasic Zeichensätze.
Mit SETFONT kann der aktuelle Zeichensatz geändert werden, wenn vorher einer geladen wurde. Standardmäßig wird "Smalfont.bmp" auf den Puffer 0 geladen.

Sample:

```
PRINT "Hallo Welt", 100, 100
LOADFONT "Comp_fnt.bmp", 1
SETFONT 1
PRINT "Computer", 100, 150
SHOWSCREEN
MOUSEWAIT
```

Refs:

PRINT, SETFONT

LOADSOUND

LOADSOUND datei$, num, buffer

Lädt die .wav Datei datei$ (PCM-Wav Format) zur Verwendung unter der Nummer num
buffer gibt an, wie oft der Klang abgespielt wird, bis ein Abhaken des zuerst gespielten Klanges erfolgt. Es sind maximal 4 Buffer möglich. Achtung: jeder weitere Buffer kostet zusätzlichen Speicherplatz.

Sample:

```
LOADSOUND "Super.wav", 0, 1
PRINT "SUPER", 100, 100
PLAYSOUND 0, -100
SHOWSCREEN
MOUSEWAIT
```

Refs:

HUSH, PLAYMOVIE, PLAYMUSIC, PLAYSOUND()

LOADSPRITE

LOADSPRITE bmp$, num#

Lädt eine Grafikdatei in den Speicher, um sie später mit den SPRITE-Befehlen darzustellen. Jedem geladenen Sprite wird eine ID (Nummer) zugewiesen, unter der es später identifizierbar ist. Wird eine ID (num#) doppelt verwendet, wird das zuvor geladene Sprite im Speicher überschrieben.

Sample:

```
LOADSPRITE "Sprite.bmp", 0
PRINT "Hallo Welt", 100, 100
SPRITE 0, 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

LOADBMP, LOADFONT, SPRITE, ROTOSPRITE, ROTOZOOMSPRITE, ZOOMSPRITE, STRETCHSPRITE

LOCAL

LOCAL var#$, var#$[]
GLOBAL var#$, var#$[]
LOCAL var#=wert#, var#=wert(), ...

a#$ = LOCAL var#$
LOCAL a#$ = GLOBAL a#$
...

LOCAL definiert lokale Variablen in einer SUB / dem Hauptprogramm. Auf diese Variablen kann nur innerhalb dieser SUB zugegriffen werden. Nach dem Verlassen der SUB ist der Speicherplatz dieser Variablen wieder frei.

GLOBAL greift auf globale, namensgleiche, lokale Variablen in einer SUB zu.

Ohne Verwendung von LOCAL/GLOBAL vor einem Funktionsargument wird immer zuerst versucht die LOCALe zu verwenden. Ist diese nicht mit LOCAL definiert worden, wird die GLOBALe verwendet.

Definiert man eine Variable explizit als GLOBAL:

```
GLOBAL var#$
```

so wird die Verwendung dieser Variable dem Compiler mitgeteilt. Er gibt dann keine Warnung mehr aus: "Variable might be unassigned", wenn die Variable z.B. bei MOUSESTATE oder GETSCREENSIZE verwendet wird.

Sample:

```
LOCAL text$ // Local in "Hauptprogramm"

text$="LOCAL_MAIN"    // LOCAL text$
GLOBAL text$="GLOBAL" // GLOBAL text$

GOSUB show
PRINT text$, 100, 150
SHOWSCREEN
MOUSEWAIT
END

// ------------------------------------------------------------ //
// -=#  SHOW  #=-
// ------------------------------------------------------------ //
SUB show:
LOCAL text$="LOCAL_SUB"
// Erstellen und definieren von LOCALen Feldern:
LOCAL array[]; DIM array[5]

PRINT text$, 100, 50
// ist gleich mit:
// PRINT LOCAL text$, 100, 50
PRINT GLOBAL text$, 100, 100
```

```
ENDSUB // SHOW
```

Refs:

BYREF, GLOBAL, STATIC

LOOPMOVIE

LOOPMOVIE datei$

Wie PLAYMOVIE, jedoch wird die Animation ständig wiederholt. Mit der Leertaste kann abgebrochen werden.

Sample:

```
// LOOPMOVIE
file$="Test.avi"

PRINT "Video: "+file$, 0, 120
PRINT "Maustaste / Hit mouse" ,140, 160
SHOWSCREEN
MOUSEWAIT
LOOPMOVIE file$

PRINT "Film abgespielt" ,100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

PLAYMOVIE

MAX()

num#$ = MAX(a#$, b#$)

Liefert den größeren Wert zwischen a#$ und b#$.

```
// MIN / MAX
   a = RND(100)
   b = RND(100)
   PRINT "a: "+a+" b: "+b, 0, 100
   PRINT "max: " + MAX(a,b), 0,120
   PRINT "min: " + MIN(a,b), 0, 140
   SHOWSCREEN
   MOUSEWAIT
```

Refs:

MIN()

MID$()

ziel$=MID$(quelle$, start#, laenge#)

Kopiert aus quelle$ ein Unterwort in ziel$. Dabei ist start# der Index des ersten zu kopierenden Buchstaben. Der erste Buchstabe eines Wortes hat den Index 0.

Sample:

```
a$="GLBasic"
b$=MID$(a$, 3, 2) // "as"
PRINT a$, 100, 70
PRINT b$, 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

ASC(), CHR$(), INSTR(), LEN, REPLACE$(), SPLITSTR()

MIN()

num#$ = MIN(a#$, b#$)

Liefert den kleineren Wert zwischen a#$ und b#$.

```
// MIN / MAX
   a = RND(100)
   b = RND(100)
   PRINT "a: "+a+" b: "+b, 0, 100
   PRINT "max: " + MAX(a,b), 0,120
   PRINT "min: " + MIN(a,b), 0, 140
   SHOWSCREEN
   MOUSEWAIT
```

Refs:

MAX()

MOD()

a# = MOD(b#, c#)

Gibt den Rest einer Division zurück.

Beispiel:

```
// MOD
PRINT MOD(5,3), 100, 100 // 5/3=1, Rest=2
SHOWSCREEN
MOUSEWAIT
```

MOUSEAXIS

v=MOUSEAXIS(ax#)

Gibt Auskunft über die Achsen/Tasten der Maus.
ax#:
0=X - Geschwindigkeit
1=Y - Geschwindigkeit
2=Rad (1 auf, -1 ab)
3=linke Maustaste
4=rechte Maustaste
5=mittlere Maustaste

```
// ------------------------------- //
// Project: MOUSEAXIS
// Start: Tuesday, November 16, 2004
// IDE Version: 2.41111

WHILE TRUE
→PRINT "X:" + MOUSEAXIS(0), 0,  0
→PRINT "Y:" + MOUSEAXIS(1), 0, 20
→PRINT "Z:" + MOUSEAXIS(2), 0, 40
→PRINT "A:" + MOUSEAXIS(3), 0, 60
→PRINT "B:" + MOUSEAXIS(4), 0, 80
→PRINT "C:" + MOUSEAXIS(5), 0,100
→SHOWSCREEN
WEND
```

Refs:

MOUSESTATE, MOUSEWAIT

MOUSESTATE

MOUSESTATE mx##, my##, mba##, mbb##
SETMOUSE mx#, my#

Liefert die Mauskoordinaten und -knöpfe zurück. Die Werte mx# und my# enthalten nach Aufruf die Koordinaten des (versteckten) Mauszeigers. mba# und mbb# sind die Werte der Knöpfe (1=Gedrückt, 0=Losgelassen).
Mit SETMOUSE lassen sich die Mauskoordinaten auf einen bestimmten Wert setzen.

Sample:

```
WHILE TRUE // Endlos Schleife
  LimitMouse(100, 100, 400, 300)
  PRINT "<=", mx, my
  IF b1 THEN END
  SHOWSCREEN
WEND

// ------------------------------------------------------------ //
// -=#  LIMITMOUSE  #=-
// MOUSESTATE mx, my, b1, b2 (GLOBAL)
// Mit Limitierung des Bereichs
// ------------------------------------------------------------ //
FUNCTION LimitMouse: minx, miny, maxx, maxy
  // Diese Variablen sind als LOCAL definiert:
  // minx, miny, maxx, maxy, void
  MOUSESTATE mx, my, b1, b2
  IF mx<minx THEN mx=minx
  IF mx>maxx THEN mx=maxx
  IF my<miny THEN my=miny
  IF my>maxy THEN my=maxy
  SETMOUSE mx, my
ENDFUNCTION void // LIMITMOUSE
```

Refs:

GETDIGI...(), GETJOY...(), JOYSTATE, KEY(), KEYWAIT, MOUSEAXIS, MOUSEWAIT, SETMOUSE

MOUSEWAIT

MOUSEWAIT | KEYWAIT

Wartet darauf, dass der Spieler eine Maustaste / Keyboard-Taste drückt.

Sample:

```
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT // Warten auf Maustaste
END
```

Refs:

MOUSESTATE

NET...

NET...

Die Netzwerkbefehle bitte im Tutorial nachlesen

Refs:

06 Netzwerk, NETWEBGET()

NETGETIP$()

ip$ = NETGETIP$()

Gibt eine Ansammlung möglicher IP Adressen für diesen Computer aus. Die IPs sind mit "|" getrennt.

```
PRINT "IP:", 0, 0
ips$ = NETGETIP$()
```

```
num = SPLITSTR(ips$, id$[], "|")
FOR i=0 TO num-1
   PRINT id$[i], 0, i*20+20
NEXT
```

Refs:

NET...

NETWEBGET()

ok = NETWEBGET(server$, url$, port#, file$)

Der Befehl holt eine Datei vom Webserver "server$" mit dem Pfadnamen "url$" unter Benutzung von port# und speichert die Datei unter file$ ab.

Direkte Verbindung:

```
NETWEBGET("www.server.com", "x/y/z.html", 80, local$)
```

Über einem Proxy-Server:

```
NETWEBGET( proxyserver$, "http://www.server.com/x/y/z.html", proxyport, local$)
```

Wenn man keinen Proxa hat, bitte das "http://" weglassen.

Du bekommst eine kostenlose Highscore-Liste auf der GLBasic.com Webseite. Hier ist, wie man sie benutzt:

```
name$ = "dein name"
score = 300
INPUT name$, 0,0

ShowHighscore: name$, score

// ------------------------------------------------------------ //
//   --- SHOWHIGHSCORE ---
// ------------------------------------------------------------ //
FUNCTION ShowHighscore: name$, score
LOCAL hi$[], hi[]
   // Firewall braucht evtl einen Mauszeiger
   SYSTEMPOINTER TRUE
   ReadHighScores(hi$[], hi[], name$, score)
   SYSTEMPOINTER FALSE

   FOR i=0 TO BOUNDS(hi[], 0)-1
      PRINT MID$(hi$[i]+".................", 0, 12) + FORMAT$(6,0,hi[i]),0,i*20
   NEXT
   SHOWSCREEN
   MOUSEWAIT
ENDFUNCTION // WIDESCROLLER


FUNCTION ReadHighScores: names$[], scores[], username$, thescore
LOCAL ct, i, str$, proxy$, port

   // Wenn Du eine persönliche Highscore-liste
   // willst, schreib an www.glbasic.com
   // damit eine erstellt wird.
   // Es gibt 2 Optinen bei der Liste:
   // game="whatever" & name="kung Fu" & secret = "the word"
   // game="whatever" & reset=1        & secret = "the word"

   str$ = "highscores/listgame.php"
   str$ = str$ + "?game=quack foo"
   str$ = str$ + "&name="+username$
   str$ = str$ + "&score="+thescore
   str$ = str$ + "&secret=1234"

   PRINT ".: WebGet scores :.", 0, 0
   SHOWSCREEN

   INIOPEN "config.ini"
   proxy$ = INIGET$("server", "proxy")
   port   = INIGET$("server", "port")
   KILLFILE "scores.ini"
   IF port=0 OR proxy$="NO_DATA"
      NETWEBGET("www.glbasic.com", str$, 80, "scores.ini")
   ELSE
      NETWEBGET(proxy$, "http://www.glbasic.com/" + str$, port, "scores.ini")
```

```
   ENDIF

   INIOPEN "scores.ini"
   ct = INIGET$("scores", "count")
   DIM names$[ct]
   DIM scores[ct]

   FOR i=1 TO ct
      names$[i-1] = INIGET$("scores", "n"+i)
      scores[i-1] = INIGET$("scores", "s"+i)
   NEXT
ENDFUNCTION
```

Refs:

NET..., NETGETIP$()

NEXT

.

Refs:

FOR

OPENFILE()

ok = OPENFILE(channel#, file$, in#)

Öffnet die Datei file$ zum schreiben (in#=FALSE) oder lesen (in#=TRUE) und weist den Kanal channel# zu. Der Rückgabewert ist TRUE oder FALSE, je nach dem, ob die Datei geöffnet werden konnte.

Lesen von Dateien

READBYTE channel#, val##
READWORD channel#, val##
READLONG channel#, val##
READIEEE channel#, val##
READSTR channel#, val$#, nc#
READLINE channel#, val$#

Schreiben von Dateien

WRITEBYTE channel#, val#
WRITEWORD channel#, val#
WRITELONG channel#, val#
WRITEIEEE channel#, val#
WRITESTR channel#, val$
WRITELINE channel#, val$

BYTE = 1 Byte, (0..255)
WORD = 2 Byte, (0..65535)
LONG = 4 Byte, (0..4294967295)
IEEE = 8 Byte, IEEE 64 Bit Gleitkommezahl

Ganzzahlen werden ohne Vorzeichen geladen. Ein Wert -127 wird bei BYTE dann 255.
Mit IEEE kann man Gelitkommazahlen speichern. Damit kann man die GLBasic Zahlen am besten speichern.

Der Wert nc# bei READSTR gibt an, wieviele Zeichen (BYTE) gelesen werden sollen.

Bei READLINE wird bis zum "\n" gelesen, und ein mögliches "\r" wieder getrimmt. Somit ist es möglich auch UNIX Dateien zu lesen. Beim Schreiben mit WRITELINE wird immer ein "\r\n" angehängt.
Möchte man UNIX Dateien schreiben, verwendet man:
WRITESTR 1, text$ + "\n"

ok# = ENDOFFILE(channel#)

Gibt an, ob keine weiteren Daten mehr zum Lesen bereitstehen.

CLOSEFILE(channel#)

Schießt eine Datei wieder. Dateien müssen geschlossen werden, damit sie von anderen Anwendungen oder GLBasic wieder gelesen werden können.

```
// Files
test$ = "test.bin"
OPENFILE(1, test$, FALSE)

WRITEBYTE 1, 42
WRITEBYTE 1, -42
WRITEWORD 1, 16767
WRITEWORD 1, -16767
WRITEIEEE 1, 1.234E-4

xx$= "Hello World\nYes, Hello\n"
WRITESTR  1, xx$
WRITELONG 1, 16767
WRITELONG 1, -16767
CLOSEFILE 1

LOCAL b1, b2, w1, w2, l1, l2, ieee
LOCAL x2$, _x1$, _x2$
OPENFILE(1, test$, TRUE)
READBYTE 1, b1
READBYTE 1, b2
READWORD 1, w1
READWORD 1, w2
READIEEE 1, ieee

// READSTR 1, x2$, LEN(xx$)
READLINE 1, _x1$
READLINE 1, _x2$
READLONG 1, l1
READLONG 1, l2
CLOSEFILE 1

DGInt i=0;
PRINT "b1="+b1, 0,i); INC(i, 10);
PRINT "b2="+b2, 0,i); INC(i, 10);
PRINT "w1="+w1, 0,i); INC(i, 10);
PRINT "w2="+w2, 0,i); INC(i, 10);
PRINT "ie="+ieee, 0,i); INC(i, 10);
PRINT "st="+_x1 + "-" + _x2, 0,i); INC(i, 10);
PRINT "l1="+l1, 0,i); INC(i, 10);
PRINT "l2="+l2, 0,i); INC(i, 10);

SHOWSCREEN
MOUSEWAIT
```

Refs:

INIOPEN

OR

.

Refs:

AND

PLATFORMINFO$()

info$ = PLATFORMINFO$(was$)

Gibt ein Wort mit platformspezifischen Infomrationen aus. Dabei entscheidet was$ über den Inhalt von info$.

was$ = "" (leer): die Platform wird ausgegeben:
"LINUX" = Linux Betriebssystem
"WIN32" = Windows Desktop (9x, NT, XP, 2003, ...)
"WINCE" = PocketPC

was$ = "ID" gibt eine eindeutige Computer/Gerätekennung zurück, die aus den Ziffern 0-9 und 'A'-'F' besteht.

was$ = "BATTERY" gibt den Akkustand in Prozent zurück. (Nur PocketPC)

was$ = "TIME" gibt das Datum und die Uhrzeit zurück. Mit SPLITSTR kann man die einzelnen Teile trennen.

```
info$ = PLATFORMINFO$("") // Was für ein Gerät ist das
PRINT "Platform = " + info$, 0, 20

info$ = PLATFORMINFO$("ID") // Einzigartige Geräte-ID
PRINT "ID = " + info$, 0,40

// 2006-7-28 18:59:45
info$ = PLATFORMINFO$("TIME") // Zeit + Datum
PRINT "Time = " + info$, 0, 80

SHOWSCREEN
MOUSEWAIT
```

Refs:

GETFILELIST(), GETFONTSIZE, GETSCREENSIZE, GETTIMER(), GETTIMERALL(), SPLITSTR()

PLAYMOVIE

PLAYMOVIE animation$

Unterbricht das Programm und gibt die Video-Datei animation$ wieder. Es werden alle gängigen Formate unterstützt. (AVI, MPG, MP2...)
GLBasic muss eine externe Komponente dem Projekt hinzufügen. Du musst diese Datei dann auch mit Deinem Spiel mitliefern.

Sample:

```
PLAYMOVIE "animation.avi"
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

LOOPMOVIE

PLAYMUSIC

PLAYMUSIC datei$
STOPMUSIC

Spielt im Hintergrund eine Musik ab. Mit STOPMUSIC wird sie gestoppt. Es werden alle Musikformate unterstützt, deren Codecs installiert sind. (midi, wav, mp3...)

Sample:

```
PLAYMUSIC "Music.mp3"
PRINT "Musik!", 100, 100
SHOWSCREEN
MOUSEWAIT
STOPMUSIC // am Ende des Programms auch selbständig
```

Refs:

HUSH, STOPMUSIC

PLAYSOUND()

kanal# = PLAYSOUND(num#, pan#, volume#)

Spielt den Sound ab, der unter der Nummer num# geladen wurde.

pan# gibt den Stereo-Wert an:
-1.0 links
0.0 mittig
+1.0 rechts

volume# gibt die Lautstärke an:
0.0 leise
1.0 laut

kanal# ist der Kanal, auf dem der Sound läuft. Siehe SOUNDPLAYING().

Sample:

```
LOADSOUND "Super.wav", 0, 1

PRINT "SUPER", 0, 100
PLAYSOUND (0, -1, 1)
SHOWSCREEN
MOUSEWAIT
HUSH

PRINT "SUPER", 500, 100
PLAYSOUND (0, 1, 1)
SHOWSCREEN
MOUSEWAIT
```

Refs:

LOADSOUND, PLAYMUSIC, SOUNDPLAYING()

POLYVECTOR

STARTPOLY spriteID#
POLYVECTOR x#, y#, tx#, ty#, col#
...
ENDPOLY

Dieser Befehl ist der mächtigste Befehl der Familie SPRITE. Mit ihm können beliebige Polygone gezeichnet werden. Auch die Farbwerte können für jede Ecke des Polygons verändert werden. Es müssen mindestens 3 POLYVECTOR-Sätze zwischen STARTPOLY und ENDPOLY stehen, da sonst keine Fläche zu sehen ist. Zwischen STARTPOLY und ENDPOLY dürfen keine weiteren SPRITE/PRINT-Befehle als POLYVECTOR stehen. Missachten führt zu ungewünschten Resultaten bis hin zu Systemabstürzen!

o spriteID# ist ein ID eines mit LOADSPRITE geladenen Sprite-Bitmaps.
-1 bedeutet, keine Textur (=Bitmap) verwenden.
o x# und y# sind die Bildschirmkoordinaten des Vektors.
Bitte beachten, dass Polygone gegen den Uhrzeigersinn eingegeben werden müssen!
o tx# und ty# sind die Koordinaten aud der Textur,
also die Pixelwerte des geladenen Sprite-Bitmaps für dieses Polygoneck.
o col# ist ein RGB( ) Farbwert, der die Ecke zusätzlich zur Textur färbt.
RGB(255, 255, 255) (Weiss) ist unverfälschte Darstellung.

Sample:

```
// POLYSPRITES

LOADSPRITE "Block.bmp", 0 // 64x64 Bitmap
LOADBMP "Test.bmp"
 STARTPOLY 0 // Bitmap = Nr.0
  POLYVECTOR  0,   0,  0,  0, RGB(255, 255, 255)
  POLYVECTOR   0, 300,  0, 64, RGB (255, 255, 255)
  POLYVECTOR 300, 300, 64, 64, RGB(255, 255, 255)
  POLYVECTOR 250,  50, 64,  0, RGB(  0, 255,   0)
 ENDPOLY
  SHOWSCREEN
  MOUSEWAIT
```

Refs:

SPRITE, ROTOSPRITE, ROTOZOOMSPRITE, ZOOMSPRITE, STRETCHSPRITE

POW()

a#=POW(b#, c#)

Berechnet den Wert von b# hoch c#.

Sample:

```
// POW
PRINT POW(5, 3), 100, 100 // 5*5*5 = 125
SHOWSCREEN
MOUSEWAIT
```

Refs:

SQR()

PRINT

PRINT ausgabe#$, x#, y#

Druckt ein Wort / Zahl auf dem Grafikbildschirm.

Sample:

```
PRINT "Hallo Welt", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

DEBUG, INPUT

PUTFILE

.

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETCURRENTDIR$, GETFILE, OPENFILE()

READ...

Siehe OPENFILE

Refs:

OPENFILE()

REDIM

REDIM feld#$[a1] | [a2][a3]...

Weist einem Feld eine neue Größe zu (kleiner oder größer ist egal) und erhält dabei alle Einträge, die sowohl in der alten als auch in der neuen Größe vertreten waren.

```
// ---------------------------------- //
// Project: REDIM
DIM a[5][4]
→Fill(a[])
→view(a[], 0)

REDIM a[6][3]
→view(a[], 256)

SHOWSCREEN
MOUSEWAIT
```

```
// ------------------------------------------------------------ //
// -=#  FILL  #=-
// ------------------------------------------------------------ //
FUNCTION Fill: f[]
→FOR x=0 TO BOUNDS(f[],0)-1
→→FOR y = 0 TO BOUNDS(f[],1)-1
→→→f[x][y] = x+10*(y+1)
→→NEXT
→NEXT
ENDFUNCTION // FILL

// ------------------------------------------------------------ //
// -=#  VIEW  #=-
// ------------------------------------------------------------ //
FUNCTION view: f[], dy
→FOR x=0 TO BOUNDS(f[],0)-1
→→FOR y = 0 TO BOUNDS(f[],1)-1
→→→PRINT f[x][y], x*80, y*20 + dy
→→NEXT
→NEXT
ENDFUNCTION // VIEW
```

Refs:

DIM

REPLACE$()

neu$ = REPLACE$(worin$, suche$, ersetzen$)

Ersetzt die Zeichenfolge suche$ im Text worin$ durch ersetzen$ und gibt sie an neu$ zurück. Mehrfache Vorkommen werden auf einmal ersetzt.

```
// Replace

PRINT "Turn a Frog into a Prince", 0, 0

a$ = "Frog"
PRINT a$, 0, 20

a$ = REPLACE$(a$, "F", "P")
PRINT a$, 0, 40

a$ = REPLACE$(a$, "og", ince")
PRINT a$, 0, 60

SHOWSCREEN
MOUSEWAIT
```

Refs:

INSTR(), LCASE$(), LEN, MID$(), UCASE$()

RETURN

.

Refs:

GOSUB

RGB()

farbe=RGB(rot#, gruen#, blau#)

Liefert den Farbwert mit den angegebenen Rot-, Grün und Blauanteilen. Die Werte für rot#, gruen# und blau# müssen im Bereich von 0 bis 255 liegen.

Sample:

```
FILLRECT 0, 0, 200, 200, RGB(255, 255, 0) // Gelb
SHOWSCREEN
```

```
MOUSEWAIT
//   R   G   B
// 255   0   0 - Rot
// 255  80   0 - Orange
// 255 255   0 - Gelb
//   0 255   0 - Grün
//   0 255 255 - Türkis
//  80  80 255 - Himmel-Blau
//   0   0 255 - Blau
//  80   0 255 - Lila
//  80  40  10 - Braun
//  80  80  80 - Grau
```

Refs:

DRAWLINE, FILLRECT, GETPIXEL(), SETPIXEL

RND()

a#=RND(max#)

Liefert eine Zufallszahl zwichen [0; max#]. Also kann 0 als auch max# zurückgegeben werden.

Sample:

```
PRINT "Eine Zahl zwischen 0 und 100: " + RND(100), 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

PRINT

ROTOSPRITE

.

Refs:

SPRITE

ROTOZOOMSPRITE

.

Refs:

SPRITE

SAVEBMP

SAVEBMP bmp$

Speichert den aktuellen Back-Buffer unter dem Dateinamen bmp$ ab. Dieser Befehl ist für Screenshots gedacht.

Sample:

```
PRINT "Hallo Welt", 100, 100
SAVEBMP "Test.bmp"
```

Refs:

LOADBMP, SAVESPRITE

SAVESPRITE

SAVESPRITE num#, datei$

Speicher das Sprite mit der ID num# unter dem Dateinamen datei$ im BMP Format ab.

```
// SaveSprite
FOR i = 0 TO 1000
→SETPIXEL RND(100), RND(100), RND(1677215)
NEXT
GRABSPRITE 0, 0,0, 100,100
SAVESPRITE 0, "ugly.bmp"

// MSPaint?
SHELLCMD("explorer.exe ugly.bmp", FALSE, TRUE, i)
```

Refs:

GRABSPRITE, SAVEBMP

SELECT

SELECT n
CASE 3; ...
CASE >5; ...
CASE 2 TO 3; ...
DEFAULT; ...
ENDSELECT

Mit SELECT kann man schnell Fallunterscheidungen treffen. Dabei werden die Blöcke zwischen den CASE-Wörtern nur ausgeführt, wenn für das SELECT Argument (n) der Fall zutrifft. Treffen mehrere Fälle zu, wird nur der erste, passende ausgewertet.

-CASE 3
'n' muss gleich 3 sein
-CASE >=5
'n' muss größergleich 5 sein
-CASE 2 TO 5
'n' muss zwischen 2 und 5 sein
-DEFAULT
'n' erfüllte keine der vorhergehenden Bedingungen

Siehe auch: IF_THEN_ELSE

Sample:

```
// SELECT CASE DEFAULT ENDSELECT

FOR n=0 TO 10

PRINT "n="+n, 100, 100

  SELECT n
  CASE 7
   PRINT "CASE 7", 0, 0
  CASE >3
   PRINT "CASE >3", 0, 20
  CASE >9
   PRINT "CASE >9", 0, 40
  CASE 3 TO 8
   PRINT "CASE 3 TO 8", 0, 60
  DEFAULT
   PRINT "DEFAULT", 0, 80
 ENDSELECT

 SHOWSCREEN
 MOUSEWAIT
NEXT
```

Refs:

IF

SETCURRENTDIR()

ok = SETCURRENTDIR(dir$)

Setzt das aktuelle Arbeitsverzeichnis auf dir$.

Sample:

```
// ------------------------------- //
// Project: Files and Folders
// Start: Tuesday, November 25, 2003
// IDE Version: 1.31125

 LOADFONT "minifont.bmp", 0
 GETFONTSIZE font_x, font_y

 ok = SETCURRENTDIR("..") // One up
 cur$ = GETCURRENTDIR$()
 num = GETFILELIST("*.*", files$[])
 num_dir  = INTEGER(num / 0x10000) // Hi-Word
 num_file = MOD(num, 0x10000)      // Lo-Word
 PRINT cur$, 0, 0
 PRINT "ok? " + ok+ " num: " + num + " -> nDir: "+num_dir+" nFile: "+num_file, 0, font_y
 FOR i=0 TO BOUNDS(files$[], 0)-1 // BOUNDS(files$[], 0)-1 = num = num_dir+num_file
  PRINT files$[i], 0, (i+3)*font_y
 NEXT

 SHOWSCREEN
 MOUSEWAIT
```

Refs:

DOESFILEEXIST(), FILEREQUEST$(), GETCOMMANDLINE$(), GETCURRENTDIR$, GETFILELIST(), SHELLCMD()

SETFONT

.

Refs:

INPUT, LOADFONT, PRINT

SETMOUSE

SETMOUSE x#, y#

Setzt die Mauskoordinaten auf x#, y#. Der Befehl ist sinnvoll um die Mausbewegung zu limitieren.

```
// SETMOUSE

   WHILE TRUE
      LimitMouse(100, 100, 400, 300)
      PRINT "<=", mx, my
      IF b1 THEN END
      SHOWSCREEN
   WEND

// ------------------------------------------------------------- //
// -=#  LIMITMOUSE  #=-
// MOUSESTATE mx, my, b1, b2 (GLOBAL)
// Mit Limitierung des Bereichs
// With limiting of the area
// ------------------------------------------------------------- //
FUNCTION LimitMouse: minx, miny, maxx, maxy
   // Diese Variablen sind als LOCAL definiert:
   // These variables are defined LOCAL:
   // minx, miny, maxx, maxy
   MOUSESTATE mx, my, b1, b2
   IF mx<minx THEN mx=minx
   IF mx>maxx THEN mx=maxx
   IF my<miny THEN my=miny
   IF my>maxy THEN my=maxy
   SETMOUSE mx, my
ENDFUNCTION // LIMITMOUSE
```

Refs:

MOUSESTATE

SETPIXEL

SETPIXEL x#, y#, farbe#

Setzt einen farbigen Pixel (Bildpunkt) an die Stelle (x#, y#) mit der Farbe farbe#.

Sample:

```
// Kreis zeichnen
FOR i=0 TO 359
    SETPIXEL 100*SIN(i)/1000+320, 100*COS(i)/1000+240, RGB(255, 255, 255)
NEXT
SHOWSCREEN
MOUSEWAIT
```

Refs:

DRAWLINE, FILLRECT, GETPIXEL(), RGB()

SETSCREEN

SETSCREEN beite#, hoehe#, vollbild#

Ändert die Größe des Bildschirms. Wenn breite# oder hoehe# = 0 sind, wird der aktuelle Wert beibehalten.
Vollbild muss man angeben. Man kann den aktuellen Status mit ISFULLSCREEN() auslesen.

```
SETSCREEN 640, 480, 1
PRINT "Vollbild", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

ISFULLSCREEN(), LIMITFPS

SETSHOEBOX

SETSHOEBOX daten$, medien$

Gibt eine Schuhschachteldatei an, aus der Dateien geladen werden, falls sie nicht unkomprimiert gefunden werden. Schuhschachteln sind Dateien aus einem Ordner, die in ein Archiv komprimiert wurden. Dazu gibt es bei den Werkzeugen ein Programm, das aus den Dateien eines Ordners Schuhschachteln macht.

daten$ gibt eine Schuhschachtel an für die Befehle:
LOADSPRITE, LOADSOUND, LOADFONT, LOADBMP, BLENDSCREEN, GETFILE, X_LOADOBJ
Die medien$ Schuhschachtel wird verwendet von PLAYMUSIC, PLAYMOVIE, LOOPMOVIE

```
// Erstelle eine Schuhschachtel mit dem Werkzeug aus dem Menü:
// Werkzeuge/Schuhschachtel
// - Ordner auswählen (..\GLBasic\Projects\Samples\Media\ShoeBox)
// - Schuhschachtel wird erstellt: ..\Media\ShoeBox.sbx

//         Daten Dateien, Dateien für PLAYMUSIC und PLAYMOVIE
SETSHOEBOX "ShoeBox.sbx", ""

// Versuche die Datei von der Festplatte zu laden.
// Wenn sie nicht existiert, wird sie aus der
// Schuhschachtel geladen. Die Pfadangeben sind egal.

LOADBMP "blah/blah/ShoeBox_file.bmp"

SHOWSCREEN
MOUSEWAIT
```

Refs:

LOADBMP, LOADFONT, LOADSOUND, LOADSPRITE, LOOPMOVIE, PLAYMOVIE, PLAYMUSIC, PLAYSOUND(), X_LOADOBJ

SHELLCMD()

ok# = SHELLCMD(cmd$, wait#, show#, rv##)

Führt ein anderes Programm aus. Will man DOS-Befehle verwenden, so muss man das Programm CMD bzw. COMMAND (unter Win9x/ME) verwenden und dann den DOS-Befehl eingeben. z.B. "CMD /C dir > files.txt"

wait# gibt an, ob gewertet werden soll, bis das andere Program beendet wurde.
show# gibt an, ob das Fenster des anderen Programms gezeigt werden soll, oder ob es versteckt gestartet wird.
ok# liefert TRUE oder FALSE, je nach dem, ob das andere Programm gestartet werden konnte.
Der Rückgabewert des anderen Programms wird in rv## übertragen.

```
// ------------------------------- //
// Project: SHELLCMD

// Warnung: Kompliziertes Programm voraus!

DIM files$[1]

 ListFiles(files$[])

 FOR i=0 TO BOUNDS(files$[], 0)-1
  PRINT "-" + i + ":" + files$[i], 0, i*16+32
 NEXT
 SHOWSCREEN
MOUSEWAIT



// ------------------------------------------------------------ //
// -=#  LISTFILES  #=-
// ------------------------------------------------------------ //
FUNCTION ListFiles: names$[]
 // Diese Werte werden per Referenz übergeben:
 // names$[],
 LOCAL tmp$[]
 win9x=FALSE
 ok = SHELLCMD("CMD /C dir /B /ON /OG > dir.txt", TRUE, FALSE, rv)
 IF ok =FALSE
  ok = SHELLCMD("COMMAND /C dir /B /ON /OG > dir.txt", TRUE, FALSE, rv)
  win9x=TRUE
 ENDIF
 IF ok=FALSE
  PRINT "Strange error - no cmd supplied!?", 0, 0
 ELSE
  IF win9x
   PRINT "Windows 9x based", 0, 0
  ELSE
   PRINT "Windows NT based", 0, 0
  ENDIF
 ENDIF

 FOR i=0 TO 255
  GETFILE "dir.txt", i, data$
  IF data$ = "NO_DATA" OR data$ = "NO_FILE" THEN RETURN i
  // realloc space for old files + new file
  DIM tmp$[i+1]
  // copy existing files into temp array
  FOR n=0 TO i-1
   tmp$[n] = names$[n]
  NEXT
  // realloc space for original array
  DIM names$[i+1]
  // copy temp files to array
  FOR n=0 TO i-1
   names$[n] = tmp$[n]
  NEXT
  // add new file
  names$[i] = data$
 NEXT
 RETURN i
ENDFUNCTION
```

Refs:

SHELLEND

SHELLEND

SHELLEND cmd$

Startet das Programm cmd$ und beendet das aktuelle Programm.

Refs:

SHELLCMD()

SHOWSCREEN

SHOWSCREEN

Zeigt den Grafikbildschirm an. Der ehemalige Grafikbildschirm (Back-Buffer) wird angezeigt, der neue Back-Buffer wird mit dem geladenen Bild (oder schwarzer Farbe) gelöscht für neue Grafiken.

Sample:

```
// Schreiben auf Back-Buffer
PRINT "Hallo Welt", 100, 100
// Anzeigen des Back-Buffers
SHOWSCREEN
MOUSEWAIT
```

Refs:

BLACKSCREEN, BLENDSCREEN, HIBERNATE, LOADBMP, USEASBMP

SIN()

a=SIN(winkel#)
a=COS(winkel#)
a=TAN(winkel#)
winkel=ASIN(v#)
winkel=ACOS(v#)
winkel=ATAN(dy#, dx#)

Gibt den Sinus (Cosinus / Tangens) von winkel# im DEG-Format (0-360°) aus.

Sample:

```
// Kreisflug eines X
FOR alpha=0 TO 360
    PRINT "X", 150*SIN(alpha)/100 + 150, 150*COS(alpha)/100 +150
    SHOWSCREEN
NEXT
```

SOUNDPLAYING()

status#$ = SOUNDPLAYING(kanal#$)

Gibt zurück, ob ein Sound noch läuft. Der kanal#$ ist der Rückgabewert von PLAYSOUND()

```
LOADSOUND "super.wav", 0, 1

ch = PLAYSOUND(0, 1,0)
WHILE SOUNDPLAYING(ch)
   PRINT "Playing...", 100,100
    SHOWSCREEN
WEND
PRINT "Stop!", 100, 100
SHOWSCREEN
MOUSEWAIT
```

Refs:

PLAYSOUND()

SPLITSTR()

anzahl# = SPLITSTR(text$, feld#$[], split$)

Splittet text$ in seperate Blöcke auf, wobei jedes Zeichen in split$ als Trennzeichen verwendet wird und liefert die Anzahl der Blöcke zurück.

```
num = SPLITSTR("  Dream.Design - Entertainment", splits$[], ". -")
PRINT num, 0, 0
FOR i=0 TO num-1
   PRINT ">"+splits$[i]+"<", 0, 20*(i+1)
NEXT
SHOWSCREEN
MOUSEWAIT
```

Ausgabe:
3
>Dream<
>Design<
>Entertainment<

Refs:

INSTR(), MID$()

SPRCOLL

SPRCOLL id1#, x1#, y1#, id2#, x2#, y2#

Dieser Befehl bietet eine Kollisionsabfrage auf Pixelebene. Der Befehl führt intern ein BOXCOLL aus, so dass man das nicht mehr zu machen braucht, um Geschwindigkeit zu sparen.

Sample:

```
// SPRCOLL DEMO

LOADBMP "test.bmp"
LOADSPRITE "Bubble.bmp", 0    // LOADSPRITE $name, #num
LOADSPRITE "Block.bmp",  1

WHILE TRUE
 MOUSESTATE mx, my, b1 ,b2
 SPRITE 0, mx, my, 0
 SPRITE 1, 100, 150, 0

 IF SPRCOLL (0, mx, my, 1, 100, 150)
  PRINT "Wahoooo", 100, 100
 ENDIF
 SHOWSCREEN
WEND
```

Refs:

BOXCOLL

SPRITE

SPRITE num#, x#, y#
STRETCHSPRITE nr#, x#, y#, breite#, hoehe#
ROTOSPRITE num#, x#, y#, phi#
ROTOZOOMSPRITE num#, x#, y#, phi#, rel#
ZOOMSPRITE num#, x#, y#, relx#, rely#

Ein SPRITE wird als Grafik auf den Backbuffer gezeichnet. Dabei kann man die num# mehrmals verwenden und muss die Grafik nur einmal laden. Der Name SPRITE sollte eigentlich BLIT oder IMAGE heissen, ist aber aus Kompatibilitätsgründen so erhalten

geblieben.

SPRITE

Zeichnet das Sprite mit der Nummer num# an die Position x# und y#.

STRETCHSPRITE

Ähnlich wie ZOOMSPRITE, jedoch mit genauer Angabemöglichkeit der Eckkoordinaten des Sprites.

ROTOSPRITE

Wie SPRITE jedoch mit Rotation um den Winkel phi# (DEG) gegen den Uhrzeigersinn.

ROTOZOOMSPRITE

Wie SPRITE mit relativer Größenänderung um Faktor rel# und Drehung um phi# gegen den Uhrzeigersinn. Der Mittelpunkt bleibt dabei der Gleiche wie bei SPRITE

ZOOMSPRITE

Wie SPRITE, jedoch mit Skalierung um Faktor relx# horizontzal und rely# vertikal. Der Mittelpunkt bleibt dabei der Gleiche wie bei SPRITE.

Sample:

```
LOADSPRITE "sprite.bmp", 0
SPRITE 0, 100, 100
SHOWSCREEN
MOUSEWAIT

LOADSPRITE "Bubble.bmp", 0

STRETCHSPRITE 0, 100, 100, 128, 128
SHOWSCREEN
MOUSEWAIT

LOADSPRITE "Sprite.bmp", 0
ROTOSPRITE 0, 100, 100, 45
SHOWSCREEN
MOUSEWAIT

LOADSPRITE "Sprite.bmp", 0
ROTOZOOMSPRITE 0, 100, 100, 45, 50
SHOWSCREEN
MOUSEWAIT

LOADSPRITE "Sprite.bmp", 0
ZOOMSPRITE 0, 100, 100, 50, 200 // Gaaaanz laaaang
SHOWSCREEN
MOUSEWAIT
```

SQR()

n# = SQR(zahl#)

Berechnet die Quadratwurzel einer Zahl.

Beispiel:

```
// SQR
PRINT SQR(9), 100, 100 // = 3
SHOWSCREEN
MOUSEWAIT
```

STARTPOLY

.

Refs:

POLYVECTOR

STATIC

STATIC var#$

Wie LOCAL, nur dass die Variable ihren Wert beim erneutem Aufruf der Funktion nicht verliert.

```
// STATIC / LOCAL
FOR i=0 TO 5
   foo()
NEXT
SHOWSCREEN
MOUSEWAIT

FUNCTION foo:
LOCAL  l = 5
STATIC s = 5
   INC l, 1
   INC s, 1
   PRINT "l="+l+" s="+s, 0, s*20
ENDFUNCTION
```

Refs:

GLOBAL, LOCAL

STEP

.

Refs:

FOR

STOPMUSIC

.

Refs:

HUSH, PLAYMUSIC

STRETCHSPRITE

.

Refs:

SPRITE

SUB

SUB name:
...
ENDSUB

...
GOSUB name

SUB ist eine FUNCTION ohne Parameter und Rückgabewert.

```
// GOSUB subcode; <-> RETURN;
```

```
a$="-"
GOSUB Setup
PRINT a$, 0, 0
SHOWSCREEN
MOUSEWAIT
// Ende des Hauptprogramms



// Ab hier nur noch SUBs
// Ab hier kein Code ausserhalb von SUB und ENDSUB !!
SUB Setup:
  a$="Setup_Complete"
  RETURN          // Zurück zum GOSUB
  a$="Das erscheint nie"
ENDSUB
```

Refs:

FUNCTION, GOSUB

SYSTEMPOINTER

SYSTEMPOINTER modus#

Zeigt den aktuellen System Mauspfeil an (modus#=TRUE) oder schaltet ihn ab (modus#=FALSE). Damit kann man GUIs programmieren, oder PocketPC Stifteingabe am PC durch den Mauspfeil simulieren.

Refs:

MOUSEAXIS, MOUSESTATE, MOUSEWAIT, SETMOUSE

TAN()

.

Refs:

SIN()

THEN

.

Refs:

IF

TO

.

Refs:

FOR

TYPE

TYPE name
members
ENDTYPE

Mit dem TYPE Befehl lassen sich Datenstrukturen zusammenfassen. Eine detailierte Beschreibung findet man in den Tutorials.
Um eine Variable vom Typ 'name' zu erstellen schreibt man:

```
LOCAL variable AS name
GLOBAL variable AS name
```

Beispiel:

```
TYPE CAR
   firma$
   kosten
ENDTYPE

LOCAL Audi AS CAR

Audi.firma$ = "Audi AG, Ingolstadt"
Audi.kosten = 40000
```

Refs:

08 Types

UCASE$()

up$ = UCASE$(text$)

Macht aus text$ Großbuchstaben.

```
// UCASE$ - LCASE$
PRINT UCASE$("123 - miniMIN"), 0, 20
PRINT LCASE$("123 - MAXImax" COLOR="white">), 0, 40
SHOWSCREEN
MOUSEWAIT
```

Refs:

ASC(), CHR$(), FORMAT$(), INKEY$, INPUT, INSTR(), MID$()

USEASBMP

USEASBMP

Benutzt den aktuellen Back-Buffer (verdeckter Grafikbildschirm) als Hintergrundbild wie LOADBMP.

Dieser Befehl kann auf manchen Grafikkarten langsam werden.

Sample:

```
// Sternenhintergrund
FOR i=0 TO 1000
    PRINT "*", RND(800), RND(600)
NEXT
USEASBMP // Speichern als Hintergrund

FOR i=0 TO 600
    PRINT "-=v=-", 100, i
    SHOWSCREEN
NEXT
MOUSEWAIT
```

Refs:

BLACKSCREEN, BLENDSCREEN, LOADBMP

VIEWPORT

VIEWPORT x#, y#, breite#, hoehe#

Setzt den Zeichenbereich auf ein anderes Rechteck. Damit kann man z.b. Splitscreen Spiele schreiben, oder den Rückspiegel eines Autos rendern. Die Pixelkoordinaten fangen mit 0,0 im Viewport bei x#, y# an. 3D Grafiken werden kleiner skaliert, als ob der Viewport das Hauptfenster wären.
Mit breite#<=0 oder Höhe#<=0 wird der Ausschnitt wieder zurückgesetzt auf das gesamte Fenster.

```
// ------------------------------- //
// Project: Multiple Viewports
// Start: Wednesday, August 11, 2004
// IDE Version: 2.40811

WHILE TRUE
 PRINT "Test", 0, 0

 MOUSESTATE mx, my, b1, b2
 GOSUB Scene // 3D Scene

 VIEWPORT mx, my, 150, 100
 // Über den Bereich hinaus Zeichnen
 FILLRECT -100, -100, 500, 700, RGB(0, 0, 64)
 PRINT "Viewport.....", 0, 0

 GOSUB Scene
 VIEWPORT 0,0,0,0 // Zurücksetzen
 PRINT "Wieder normal", 0, 20
 SHOWSCREEN
WEND

// ------------------------------------------------------------ //
// -=#  SCENE  #=-
// ------------------------------------------------------------ //
SUB Scene:
 X_MAKE3D 1, 100, 45
 X_CAMERA 0,0,10, 0,0,0
 X_OBJSTART 1
  X_OBJADDVERTEX  0, 0, 0, 0,0, RGB(255,255,255)
  X_OBJADDVERTEX  1, 1, 0, 1,1, RGB(255,255,255)
  X_OBJADDVERTEX  0, 1, 0, 0,1, RGB(255,255,255)
 X_OBJEND
 X_ROTATION GETTIMERALL()/30, 0, 1, 0
 X_DRAWOBJ 1, 0
 X_DRAWAXES 0,0,0
ENDSUB // SCENE
```

WEND

.

Refs:

WHILE

WHILE

WHILE a#$ = < <= > >= <> b#$
WEND

Führt eine Schleife aus, solange der Vergleich von a#$ und b#$ WAHR ist.

Sample:

```
i=0

WHILE i < 5
    i=i+1
    PRINT i, i*32, 100
WEND

SHOWSCREEN
MOUSEWAIT
```

Refs:

BREAK, CONTINUE, FOR, GOTO

WRITE...

Siehe OPENFILE

Refs:

OPENFILE()

X_AMBIENT_LT

X_AMBIENT_LT num#, col#

Schaltet ein ambientes Licht ein, also ein Licht, das aus allen Richtungen scheint.
num# ist die Nummer des Lichtes. Erlaubte Werte sind 0-7.
col# gibt die Farbe des Lichts an. Siehe RGB().

Refs:

X_SPOT_LT

X_AUTONORMALS

X_AUTONORMALS mode#

Bestimmt ob, und wie die Licht-Normalenvectoren für die nächsten User-Objekte berechnet werden.
mode#:
0 : Keine Normalenberechnung - schnell, wenn man kein Licht braucht
1 : Normalenvektoren einzeln für jedes Dreieck
2 : Flächennormalen

Modus 1 gibt die Möglichkeit Würfel oder andere Objekte mit "harten Kanten" zu erstellen.
Bei Modus 2 wird angenommen, dass alle Dreiecke, die an den gleichen Punkt anschließen auch zu diesem gehören. Die Normalenvektoren werden an einem Flächenknick weich gemittelt.

```
// -------------------------------- //
// Project: AutoNormals
// Start: Thursday, September 30, 2004
// IDE Version: 2.40924

X_AUTONORMALS 1
Cube(0)
X_AUTONORMALS 2
Cube(1)

WHILE TRUE
 X_MAKE3D 1, 100, 45
 X_CAMERA 0,2,7, 0,.5,0
 X_SPOT_LT 0, RGB(255,255,255), 0,2,7, 0,0,-1, 45
 phi=phi+GETTIMER()/50
 X_MOVEMENT -2, 0, 0
 X_ROTATION phi, 0,1,0
 X_DRAWOBJ 0, 0
 X_MOVEMENT 2, 0, 0
 X_ROTATION phi, 0,1,0
 X_DRAWOBJ 1, 0
 SHOWSCREEN
WEND



// ------------------------------------------------------------ //
// -=#  CREATECUBE  #=-
// ------------------------------------------------------------ //
FUNCTION Cube: num
LOCAL col
 col=RGB(0xff, 0xff, 0xff) // white
 X_OBJSTART num
  // Front Face
  X_OBJADDVERTEX  .5, -.5,  .5, 1, 0, col
  X_OBJADDVERTEX -.5, -.5,  .5, 0, 0, col
  X_OBJADDVERTEX  .5,  .5,  .5, 1, 1, col
  X_OBJADDVERTEX -.5,  .5,  .5, 0, 1, col
  X_OBJNEWGROUP
  // Back Face
  X_OBJADDVERTEX -.5,  .5, -.5, 1, 1, col
  X_OBJADDVERTEX -.5, -.5, -.5, 1, 0, col
  X_OBJADDVERTEX  .5,  .5, -.5, 0, 1, col
      X_OBJADDVERTEX  .5, -.5, -.5, 0, 0, col
      X_OBJNEWGROUP
      // Top Face
```

```
      X_OBJADDVERTEX -.5,  .5,  .5, 0, 0, col
      X_OBJADDVERTEX -.5,  .5, -.5, 0, 1, col
      X_OBJADDVERTEX  .5,  .5,  .5, 1, 0, col
      X_OBJADDVERTEX  .5,  .5, -.5, 1, 1, col
      X_OBJNEWGROUP
      // Bottom Face
      X_OBJADDVERTEX  .5, -.5, -.5, 0, 1, col
      X_OBJADDVERTEX -.5, -.5, -.5, 1, 1, col
      X_OBJADDVERTEX  .5, -.5,  .5, 0, 0, col
      X_OBJADDVERTEX -.5, -.5,  .5, 1, 0, col
      X_OBJNEWGROUP
      // Right face
      X_OBJADDVERTEX  .5,  .5, -.5, 1, 1, col
      X_OBJADDVERTEX  .5, -.5, -.5, 1, 0, col
      X_OBJADDVERTEX  .5,  .5,  .5, 0, 1, col
      X_OBJADDVERTEX  .5, -.5,  .5, 0, 0, col
      X_OBJNEWGROUP
      // Left Face
      X_OBJADDVERTEX -.5, -.5,  .5, 1, 0, col
      X_OBJADDVERTEX -.5, -.5, -.5, 0, 0, col
      X_OBJADDVERTEX -.5,  .5,  .5, 1, 1, col
      X_OBJADDVERTEX -.5,  .5, -.5, 0, 1, col
      X_OBJNEWGROUP
   X_OBJEND

ENDFUNCTION // sz
```

Refs:

X_DRAWANIM, X_DRAWOBJ, X_OBJ..., X_SPOT_LT

X_CAMERA

X_CAMERA x#, y#, z#, px#, py#, pz#

Setzt die Kamera auf die Position x#, y#, z# und richtet sie aus, damit sie auf den Punkt px#, py#, pz# zeigt. "Oben" ist dabei immer entlang der positiven Y-Achse.

Refs:

X_CAMERAUP, X_FOG, X_MAKE3D, X_WORLD2SCREEN

X_CAMERAUP

X_CAMERAUP x#, y#, z#

Setzt die "oben" Richtung der Kamera. Vor X_CAMERA aufrufen. Standard nach jedem X_MAKE3D ist 0,1,0.

Refs:

X_CAMERA, X_MAKE3D

X_CLEAR_Z

X_CLEAR_Z

Löscht die Tiefenpuffer-Bits. Damit kann man z.B. eine Skybox zeichnen, und nachher den Tiefenpuffer löschen, so dass die Skybox von nun an immer überzeichnet wird.

Refs:

BLACKSCREEN, X_MAKE3D

X_COLLISION()

col#=X_COLLISION(stufe#, num#, rad#, x#, y#, z#)

Führt eine Kugel gegen 3D Ojekt-Kollision durch. stufe# ist die Animationsstufe des Objekts num#. x#, y#, z# sind der Mittelpunkt der Kugel mit dem Radius rad#.

Im Debug Modus wird die Kugel als Drahtgittermodel gezeichnet.

Refs:

X_COLLISIONAABB(), X_COLLISIONRAY()

X_COLLISIONAABB()

col=X_COLLISIONAABB(num#, frame#, xmitte#, ymitte#, zmitte#, breite_xz#, hoehe_y#)

Führt eine Kollisionsberechnung zwischen dem Quader (Axis Aligned Bounding Box) mit dem Mittelpunkt xmitte#, ymitte#, zmitte# und den Abmessungen breite_xy# und hoehe_y# mit dem Objekt num# und dessen Animationsframe frame#.

Im Debug Modus wird der Quader als Drahtgittermodel gezeichnet.

Refs:

X_COLLISION(), X_COLLISIONRAY()

X_COLLISIONRAY()

skal# = X_COLLISIONRAY num#, frame#, x#, y#, z#, dirx#, diry#, dirz#

Überprüft, od das 3D Objekt num# mit dem Strahl x#,y#,z# in Richtung dir# schneidet.
Wenn ja, ist skal# ein Wert, der den Abstand von Punkt x#,y#,z# in Richtung dir#.
Wenn nein, ist skal# = 0
skal# < 0 wird nur gesucht, wenn keine Kollision in positiver Richtung gefunden wurde. Wenn 2 Kollisionen möglich sind (-2 und +3), dann wird immer die Positive (+3) ausgegeben.

Im Debug Modus werden Linien gezeichnet:
- grün = Kollision, scal# > 0
- gelb = Kollision, scal# < 0 (rückwärts)
- rot = Keine Kollision

Refs:

X_COLLISION(), X_COLLISIONAABB()

X_CULLMODE

X_CULLMODE mode#

Hiermit kann man entscheiden, ob Vorder, Rück- oder beide Seiten eines Objekts gezeichnet werden sollen.
mode#=0 - All Seiten zeichnen
mode#=1 - Vorderseiten zeichnen
mode#=-1 - Rückseiten zeichnen

Es macht Sinn, nur bestimmte Seiten zu zeichnen, da das den Grafikprozessor entlastet. Voreinstellung bei jedem X_MAKE3D is 0.

Sample:

```
// Simple 3D
X_OBJSTART 1
   X_OBJADDVERTEX   0,  0, 0,  0,0, RGB(255,255,255)
   X_OBJADDVERTEX -20, 40, 0,  0,0, RGB(255,  0,  0)
   X_OBJADDVERTEX  20, 40, 0,  0,0, RGB(  0,  0,255)
X_OBJEND

WHILE TRUE
   phi=phi+1
   IF phi>=360 THEN phi=0

   X_MAKE3D 1, 1500
   X_CAMERA 0,0,400, 0,0,0

   FOR cmode=-1 TO 1
      X_CULLMODE cmode
      X_MOVEMENT cmode * 50, 0, 0
      X_ROTATION phi, 0, 1, 0
```

```
      X_DRAWOBJ 1, 0
   NEXT
   SHOWSCREEN
WEND
```

Refs:

X_CAMERA, X_DRAWOBJ, X_MAKE3D, X_MOVEMENT, X_OBJ..., X_ROTATION

X_DOT

X_DOT x#,y#,z#, groesse#, farbe#

Zeichnet einen 3D Punkt an x#, y#, z# mit groesse# Pixeln Durchmesser.

Refs:

RGB(), X_LINE

X_DRAWANIM

X_DRAWANIM num#, von#, bis#, interpol#, volle_anim#

Zeichnet eine interpolierte Animationsstufe des Objektes num#. Die Stufen von# und bis# geben an, zwischen welchen Animations-Schlüsselbildern interpoliert werden soll. Das erste Bild hat den Index '0'. interpol# gibt in einen Wert von 0 bis 1 an, wie weit die Interpolation fortgeschritten sein soll. volle_anim# gibt an, ob über alle Zwischenbilder interpoliert werden soll, oder nur zwischen den beiden angegebenen.
Wenn volle_anim# TRUE ist, und von#=3 und bis#=6, werden alle Zwischenstufen 3,4,5,6 benutzt für die Interpolation. Ist volle_anim# = FALSE, wird nur zwischen 3 und 6 interpoliert.

Sample:

```
// Animations Demo

ddd_file$ = "moon" // "trooper"
scale = 7

X_LOADOBJ ddd_file$+".ddd", 1
LOADSPRITE ddd_file$+".bmp", 1

WHILE TRUE
//   X_MAKE2D
//   FILLRECT 0,0,640,480,RGB(80,80,255)

   X_MAKE3D
   X_CAMERA 0, 20, 900, 0, 0, 0
   X_AMBIENT_LT 0, RGB(128,128,128)


   X_DRAWAXES 0,0,0

   phi=phi+(0.5)
   permil=permil+.001
   IF permil>=1 THEN permil=0

   X_SCALING scale, scale, scale
   X_ROTATION 270, 1,0,0
   X_ROTATION phi, 0.5,1,0

   X_SETTEXTURE 1, -1

   X_DRAWANIM 1, 0, 128, permil, TRUE
//   X_DRAWOBJ 1, 0 // ID of Object, Number of Keyframe to Render

   X_MAKE2D
   PRINT "PRINT/SPRITE ? MAKE2D !", 10, 20
   SHOWSCREEN
WEND
```

Refs:

X_DRAWOBJ, X_LOADOBJ, X_OBJ...

X_DRAWAXES

X_DRAWAXES x#, y#, z#

Zeichnet ein Koordinatensystem an die angegebene Position. Das ist sehr nützlich beim Testen von Code. Die Achsen haben folgende Farben:
X: rot
Y: grün
Z: blau

X_DRAWOBJ

X_DRAWOBJ num#, stufe#

Zeichnet eine Animationsstufe eines geladenen 3D Ojektes. num# gibt die Ojekt-ID an, stufe# den Wert der Animationsstufe. Die erste Stufe hat den Wert '0'.

Sample:

```
// DRAWOBJ Demo

ddd_file$ = "moon"
scale = 7

X_LOADOBJ ddd_file$+".ddd", 1
LOADSPRITE ddd_file$+".bmp", 1

WHILE TRUE
   X_MAKE3D
   X_CAMERA 0, 20, 900, 0, 0, 0
   X_AMBIENT_LT 0, RGB(128,128,128)

   X_DRAWAXES 0,0,0

   phi=phi+(0.5)
   X_SCALING scale, scale, scale
   X_ROTATION 270, 1,0,0
   X_ROTATION phi, 0.5,1,0

   X_SETTEXTURE 1, -1
   X_DRAWOBJ 1, 0

   X_MAKE2D
   PRINT "PRINT/SPRITE ? MAKE2D !", 10, 20
   SHOWSCREEN
WEND
```

Refs:

X_DRAWANIM, X_OBJ..., X_SETTEXTURE

X_FOG

X_FOG farbe#, exp_modus#, par1#, par2#

Aktiviert Nebel. Wenn farbe# = -1, schaltet das den Nebel aus. Der Schalter exp_modus# unterscheidet zwischen linearem Nebel (exp_modus#=FALSE) oder exponentiellem Nebel (exp_modes#=TRUE).
Die Parameter par1# und par2# unterscheiden sich je nach exp_modus#:

exp_modus# = TRUE:
par1 = Dichte des Nebels - Wert von 0 bis 1
par2 = Ignoriert

exp_modus# = FALSE:
par1 = zNah - Beginn des Nebels (Abstand von der Kamera)
par2 = zFern - Ende des Nebels
Wenn p1=0 und p2=0, werden die zNah und zFern Werte der aktuellen Kamera verwendet.

Refs:

X_CAMERA

X_GETFACE

X_GETFACE obj#, frame#, index#, face[]

Mit diesem Befehl bekommt man die Information über ein Dreieck im 3D Model obj# bei dessen Animationsphase frame#.
index# gibt dabei den Index des Dreiecks an und muss im Bereich 0 und X_NUMFACES(obj#)-1 liegen.
Die Informationen werden in das Feld face[] geschrieben. Das Feld wird intern dimensioniert auf [3][9]. Die erste Feldinformation gibt den Kontenindex an, die Zweite bestimmt den Wert:

```
face[i][0] = x
face[i][1] = y
face[i][2] = z
face[i][3] = tx
face[i][4] = ty
face[i][5] = farbe
face[i][6] = nx
face[i][7] = ny
face[i][8] = nz
```

x,y,z = Koordinates des Punktes
tx,ty = Textur-Koordinaten des Punktes
nx,ny,nz = Normalenvektorenrichtung des Punktes

Damit lässt sich ein 3D Objekt komplett analysieren.

```
LOCAL face[]
   X_LOADOBJ "wumbo.ddd", 0

   X_MAKE3D 1,500,45
   X_CAMERA 10,2,-1, 0,2,0
   X_DRAWOBJ 0, 2
   count = X_NUMFACES(0)
   FOR i=0 TO count-1
      X_GETFACE 0, 2, i, face[]
      // face[node 0..2][x,y,z,tx,ty,col]
      X_LINE face[0][0], face[0][1], face[0][2], face[1][0], face[1][1], face[1][2], 4, RGB(0,0,255)
      X_LINE face[2][0], face[2][1], face[2][2], face[1][0], face[1][1], face[1][2], 4, RGB(0,0,255)
      X_LINE face[0][0], face[0][1], face[0][2], face[2][0], face[2][1], face[2][2], 4, RGB(0,0,255)

      X_DOT face[0][0], face[0][1], face[0][2], 8, RGB(0,255,0)
      X_DOT face[1][0], face[1][1], face[1][2], 8, RGB(0,255,0)
      X_DOT face[2][0], face[2][1], face[2][2], 8, RGB(0,255,0)
   NEXT

   SHOWSCREEN
   MOUSEWAIT
```

Refs:

X_NUMFACES(), X_OBJ...

X_LINE

X_LINE xa#,ya#,za#, xb#,yb#,zb#, breite#, farbe#

Zeichnet eine 3D Linie von (xa#, ya#, za#) nach (xb#, yb#, zb#) mit der Strichstärke von breite# Pixeln.

Refs:

RGB(), X_DOT, X_DRAWAXES

X_LOADOBJ

X_LOADOBJ datei$, num#

Lädt ein 3D-Objekt in den Speicher, um es unter der Referenznummer: num# zu benutzen. Das Objekt kann Animationen enthalten. Es können nur Dateien im .ddd Format geladen werden. Um MD2- und 3DS-Dateien in eine komprimierte .ddd Datei zu konvertierten, benutze das 3DConvert Programm bei den Tools.

Man kann auch eine .dda Datei laden, die aus ascii Text besteht. Das 3DConvert programm kann daraus eine .ddd machen, die dann schneller lädt.

Weiter gibt es .ddw Dateien. Das sind nur .ini Dateien, die eine "Welt" beschreiben.

Das Dateiformat (.ddw) sieht so aus:

```
;DDW1.0
[objects]
count=3 ; Anzahl der Objekte
1=Haus.ddd
ta1=1 ; Material.bmp
tb1=-1; Keine 2. Textur (z.B. Bump Mapping)

2=Garten.ddd
ta2=2 ; Natur.bmp
tb2=-1

3=Bäume.ddd
ta3=2
tb1=-1

[textures]
count=2
1=Material.bmp
2=Natur.bmp
```

Das Dateiformat (.dda) sieht so aus:

```
DDDA 1.0 # DDA file, version=1.0
# das müssen die ersten Bytes in der Datei sein
# ein # is ein Kommentar für den
# Rest der Zeile.


6 # tex count - Anzahl der Texturkoordinaten
1.000 0.004 # uv 0
0.996 0.992 # uv 1
0.000 0.992 # uv 2
1.008 0.000 # uv 3
0.000 0.988 # uv 4
0.176 0.281 # uv 5

2 # Anzahl der Flächen

# Flächenbeschreibung
142 142 142 # rgb für die Fläche

# Jetzt die Vektoren
# n p1 p2 px ... t1 t2 tx ...
# n = Anzahl der Knoten in dieser Fläche
# px = Index des Positionsvektors (siehe unten)
# tx = Index der Texturkoordinate (siehe oben)
3  0 0 3 1 2 2 # nxyz nuv ...

# nächste Fläche
142 142 142 # farbe
3  0 3 2 4 1 5 # nxyz nuv ...

2 # Anzahl der Animationsframes

# Beschreibung des Animationsframe
4 # Anzahl der Knoten dieses Frames
-0.988 3.000 0.000 # x y z 0
-2.017 1.966 -0.724 # x y z 1
-2.988 1.000 0.000 # x y z 2
-0.988 1.000 0.000 # x y z 3

# nächster Frame
4
 0 1 0 # x y z 0
-1 1 0 # x y z 1
-1 0 0 # x y z 2
 0 0 0 # x y z 3
```

Refs:

X_OBJ..., X_SAVEOBJ

X_LOADSHADER()

ok# = X_LOADSHADER(index#, vertex$, fragment$)

Dieser Befehl läd einen GLSL Shader in das Register index#. Später kann der Shader, wenn er erfolgreich geladen wurde (ok# = TRUE) mit

```
X_SETSHADER index#
```

verwendet werden.

```
// ------------------------------- //
// Project: Shaders
// Start: Monday, September 25, 2006
// IDE Version: 3.262

LOCAL shaders$[], num, current$
   IF X_LOADSHADER(13, "toon.vert", "toon.frag") = FALSE
      PRINT "Shader failed to load: ", 100,100
      SHOWSCREEN
      MOUSEWAIT
      END
   ENDIF

   CreateTorus(0, 5, 10, 12, 9, 2, 2, RGB(0xff, 0xff, 0xff))
   // CreateSphere(0, 10, 9, RGB(0xff, 0xff, 0xff) )
   WHILE TRUE
      FILLRECT 0,0,640,480,RGB(255,128,0)
      X_MAKE3D 1,500, 45
      X_CAMERA 0,0,-100, 0,0,0

      X_SPOT_LT 0, RGB(255,255,255),100,100,-1000, -.1,-.1,1, 360

      X_ROTATION 45, 1,1,0
      X_ROTATION GETTIMERALL()/100, 0,1,.2
      X_SETSHADER 13
      X_DRAWOBJ 0, 0
      SHOWSCREEN
   WEND
```

Refs:

X_PUTSHADER, X_SETSHADER, X_SPOT_LT

X_MAKE2D

X_MAKE2D

Schaltet zum 2D Modus um, damit Befehle wie PRINT und SPRITE wieder funktionieren.

Sample:
Siehe 3D Tutorials.

Refs:

X_MAKE3D

X_MAKE3D

X_MAKE3D znahe#, zfern#, brennweite#

Schaltet in den 3D Modus um. Nur Objekte oder Teile von Objekten innerhalb einer Entfernung von znahe# bis zfern# relativ zur Kamera werden gezeichnet. Eine geringe Differenz gibt bessere Resultate der Tiefenprüfung beim Zeichnen von Polygonen.
Die Brennweite ist der Öffnungswinkel der Kamera. 45 ist ein oft gewählter Wert.

Ein negativer Wert für brennweite# erstellt eine Orthogonale Ansicht. Die Größe gibt dabei einen Skalierfaktor an.

Sample:

```
X_LOADOBJ "pyramid.ddd", 5
WHILE TRUE
   MOUSESTATE mx, my, b1 ,b2
   phi=mx*360 / 640

   X_MAKE3D 1,1000,45 // Viewport 3D

   X_CAMERA 0, 150, -300,      0 ,0 ,0
   X_AMBIENT_LT 0, RGB(255,255,0)
```

```
   X_MOVEMENT mx-230, 0, 0
   X_SCALING 3, 3, 3
   X_ROTATION phi*3, 0, 1, 0
   X_DRAWOBJ 5, 0

   SHOWSCREEN
WEND
```

Refs:

X_CAMERA, X_DRAWOBJ, X_MAKE2D, X_OBJ...

X_MIPMAPPING

X_MIPMAPPING status#

Schaltet das Mip-Mapping für 3D Objekte ein oder aus. Mipmapping verwendet andere Texturen für weiter entfernte Objekte und verhindert somit grobpixelige Texturen von entfernten Objekten.

Refs:

X_SETTEXTURE

X_MOVEMENT

X_MOVEMENT x#, y#, z#

Setzt die Zeichenposition des nächsten 3D-Objekts auf die Position x#, y#, z#. Dieser Befehl muss vor X_ROTATION aufgerufen werden, da er alle vorherigen Rotationen entfernt.

Refs:

X_MULTMATRIX, X_ROTATION, X_SCALING

X_MULTMATRIX

X_MULTMATRIX mat[]

Dieser Befehl ist das Pendant zu OpenGLs "glMultMatrix". Damit kann man die aktuelle matrix multiplizieren. Das Feld mat[] muss DIM [16] sein und diesen Aufbau aufweisen:

```
| a0  a4  a8 a12 |
| a1  a5  a9 a13 |
| a2  a6 a10 a14 |
| a3  a7 a11 a15 |
```

Refs:

X_MOVEMENT, X_ROTATION, X_SCALING

X_NUMFACES()

anzahl = X_NUMFACES(obj#)

Damit lässt sich die Anzahl der Dreiecke in einem 3D Model bestimmen.

Refs:

X_GETFACE

X_OBJ...

X_OBJSTART num#
X_OBJADDVERTEX x#, y#, z#, tx#, ty#, col#
X_OBJNEWGROUP

X_ENDOBJ

Erstellt ein 3D-Objekt zur Laufzeit.
num# gibt die Objektnummer an. Bestehende Objekte werden überschrieben.
x#, y#, z# sind die Koordinatenpunkte des Dreiecks.
tx#, ty# setzen die Texturkoordinaten für diesen Punkt. 0=links/oben 1.0=rechts/unten.

Folgende Punkte werden miteinander verbunden:
n, n-1, n-2
Pro Gruppe werden n-2 Dreiecke gezeichnet.

```
1---2---5
\ / \ /|
3---4 |
\|
6
```

Sample:

```
// Pyramid demo
// ------------

// Erstelle eine Pyramide
 X_OBJSTART 5
  X_OBJADDVERTEX -5, -10, -5, 0,0,RGB(0,0,255)
  X_OBJADDVERTEX  5, -10, -5, 0,0,RGB(0,0,255)
  X_OBJADDVERTEX  0,   0,  0, 0,0,RGB(255,255,255) // Peak
  X_OBJADDVERTEX  5, -10,  5, 0,0,RGB(0,0,255)
  X_OBJADDVERTEX -5, -10,  5, 0,0,RGB(0,0,255)
// ..new Group
X_OBJNEWGROUP
// ...or Bottom
// X_OBJADDVERTEX  5, -10, -5, 0,0,RGB(0,0,255)
// X_OBJADDVERTEX -5, -10, -5, 0,0,RGB(0,0,255)

  X_OBJADDVERTEX -5, -10, -5, 0,0,RGB(0,0,255)
  X_OBJADDVERTEX  0,  0,   0, 0,0,RGB(255,255,255)
  X_OBJADDVERTEX -5, -10,  5, 0,0,RGB(0,0,255)
 X_OBJEND

//  X_SAVEOBJ "Pyramid.ddd", 5
//  X_LOADOBJ "Pyramid.ddd", 5
 WHILE TRUE
  MOUSESTATE mx, my, b1 ,b2
  phi=mx*360 / 640

  X_MAKE3D
  X_CAMERA 0, 150, -300,   0 ,0 ,0
  X_DRAWAXES 0, 0, 0

  X_AMBIENT_LT 0, RGB(255,255,0)

  X_MOVEMENT mx-230, 0, 0
  X_SCALING 3, 3, 3
  X_ROTATION phi, 0, 1, 0
  X_DRAWOBJ 5, 0

  SHOWSCREEN
 WEND
END
```

Refs:

X_AUTONORMALS, X_COLLISION(), X_DRAWANIM, X_DRAWOBJ, X_LOADOBJ, X_SAVEOBJ, X_SETTEXTURE

X_POPMATRIX

X_POPMATRIX

Das Pendant zu X_PUSHMATRIX

Refs:

X_PUSHMATRIX

X_PRINT

X_PRINT text#$, x#, y#, z#, skalierung#

Scheibt einen Text in eine 3D Szene und verdeckt ihn dabei wenn nötig. Mit skalierung# kann man die Textgröße verändern. Ein Wert von 0 schreibt den Text immer in seiner Originalgröße (nach hinten nicht kleiner werdend).
Der Texteinfügepunkt ist die untere Mitte des Textes (wie bei X_SPRITE).

Refs:

X_SPRITE

X_PUSHMATRIX

X_PUSHMATRIX

Von nun an sind alle X_MOVEMENT/X_SCALING/X_ROTATION **relativ** zu der aktuellen Matrix. Somit kann man sein Objekt platzieren, dann X_PUSHMATRIX, weitere Anbauteile mit X_MOVEMENT platzieren, zeichnen (X_DRAWOBJ), mit X_POPMATRIX beenden und zeichnet damit die Objekte als eine Einheit.

```
// -------------------------------- //
// Project: PushPopMatrix
// Start: Thursday, November 03, 2005
// IDE Version: 2.50920


donut  = 0
sphere = 1


// Donut machen
CreateTorus(donut, 12, 24, 12, 9, 2, 1)
// Kugel machen
CreateSphere(sphere, 24, 12, RGB(255,255,255))

LOCAL mx, my, b1, b2

WHILE TRUE
   // 1000 ist weit genug hier
   X_MAKE3D 1, 1000, 45

   // ein Winkel für alles
   phi = GETTIMERALL() / 150

   X_CAMERA 300,300,300, 0,0,0
   X_SPOT_LT  0, RGB(0,255,255), 0,0,0, -1,-1,-1, 180

   // Ringelreihen
   FOR i=0 TO 5
      psi = phi + i/6*360
      px = SIN(psi)*100
      py = COS(psi)*50
      pz = (SIN(psi)+COS(psi)*2)*120
      // bewegen und bisserl drehen
      X_MOVEMENT px, py, pz
      X_ROTATION psi*40, 3,4,5

      // Jetzt relaitv zum Torus die Kugeln
      X_PUSHMATRIX
         FOR j=0 TO 5
            X_MOVEMENT SIN(j/6*360)*50, 0, COS(j/6*360)*50
            X_SCALING .3, .3, .3
            X_DRAWOBJ sphere, 0
         NEXT
      X_POPMATRIX
      X_DRAWOBJ donut,0
   NEXT
   SHOWSCREEN
WEND



// ------------------------------------------------------------ //
// -=#  SPHERE  #=-
// ------------------------------------------------------------ //
FUNCTION CreateSphere: num, r, n, col
LOCAL i,j, theta1, theta2, theta3, pi
   pi = ACOS(0)*2
   IF r < 0 THEN r = -r
   IF n < 4 THEN n = 4

   X_AUTONORMALS 2 // smooth edges
```

```
   X_OBJSTART num
   FOR j=0 TO INTEGER(n/2) -1
      theta1 = j * 2*pi / n - pi/2;
      theta2 = (j + 1) * 2*pi / n - pi/2;
      FOR i=0 TO n
         theta3 = i * 2*pi / n;
         X_OBJADDVERTEX   r*COS(theta2) * COS(theta3), r*SIN(theta2), _
                     r*COS(theta2) * SIN(theta3), i/n, 2*(j+1)/n, col
         X_OBJADDVERTEX   r*COS(theta1) * COS(theta3), r*SIN(theta1), _
                     r*COS(theta1) * SIN(theta3), i/n, 2*j/n, col
      NEXT
      X_OBJNEWGROUP
   NEXT
   X_OBJEND
ENDFUNCTION // n



// ------------------------------------------------------------ //
// -=#  CREATETORUS  #=-
//
// By Samuel R. Buss
// http://math.ucsd.edu/~sbuss/MathCG
// ------------------------------------------------------------ //
FUNCTION CreateTorus: num, MinorRadius, MajorRadius, NumWraps, NumPerWrap, TextureWrapVert, TextureWrapHoriz
   // Diese Variablen sind als LOCAL definiert:
   // x, y,
   // Draw the torus
LOCAL i, di, j, wrapFrac, wrapFracTex, phi, thetaFrac, thetaFracTex, theta
LOCAL x, y, z, r

   X_AUTONORMALS 2
   X_OBJSTART num
   FOR di=0 TO NumWraps-1
      FOR j=0 TO NumPerWrap
         FOR i=di+1 TO di STEP -1
            wrapFrac = MOD(j, NumPerWrap)/NumPerWrap
            wrapFracTex = j/NumPerWrap
            phi = 360*wrapFrac
            thetaFrac = (MOD(i, NumWraps)+wrapFracTex)/NumWraps
            thetaFracTex = (i+wrapFracTex)/NumWraps
            theta = 360*thetaFrac
            r = MajorRadius + MinorRadius*COS(phi)
            x = SIN(theta)*r
            z = COS(theta)*r
            y = MinorRadius*SIN(phi)
            X_OBJADDVERTEX x,y,z,  thetaFracTex*TextureWrapVert, wrapFracTex*TextureWrapHoriz, _
                        RGB(255, 255, 255)
         NEXT
      NEXT
      X_OBJNEWGROUP
   NEXT
   X_OBJEND
ENDFUNCTION
```

Refs:

X_MOVEMENT, X_POPMATRIX, X_ROTATION, X_SCALING

X_PUTSHADER

X_PUTSHADER name$, wert#

Mit diesem Befehl kann man in den aktuell gesetzten Shader (X_SETSHADER) eine "uniform" variable ändern.
Damit kann man z.B. den Wert von GETTIMERALL() übergeben.

```
// test.frag
uniform float t;
varying float u, v;

void main()
{
   u += sin(t);
   v += t;
   gl_FragColor = vec4(sin(40*u),sin(40*v),0.0,1.0);
}

// test.vert
uniform float t;
const float pi = 3.1415926535;
```

```
varying float u, v;

void main()
{
   u = (gl_Vertex.x + 5.0) / 10.0 * 2 * pi;
   v = (gl_Vertex.y + 5.0) / 10.0 * 2 * pi;
   float r = sin(4*u+t)/4+1;
   float R = cos(6*cos(u)-t)/4+3;

   float a = R+r*cos(v);
   vec3 world = vec3(a*cos(u), a*sin(u), r*sin(v));

   gl_Position = gl_ModelViewProjectionMatrix * vec4(world,1.0);
}

// ------------------------------- //
// Project: Shaders
// Start: Monday, September 25, 2006
// IDE Version: 3.262

LOCAL shaders$[], num, current$
   IF X_LOADSHADER(13, "toon.vert", "toon.frag") = FALSE
      PRINT "Shader failed to load: ", 100,100
      SHOWSCREEN
      MOUSEWAIT
      END
   ENDIF

   CreateTorus(0, 5, 10, 12, 9, 2, 2, RGB(0xff, 0xff, 0xff))
   // CreateSphere(0, 10, 9, RGB(0xff, 0xff, 0xff) )
   WHILE TRUE
      FILLRECT 0,0,640,480,RGB(255,128,0)
      X_MAKE3D 1,500, 45
      X_CAMERA 0,0,-100, 0,0,0

      X_SPOT_LT 0, RGB(255,255,255),100,100,-1000, -.1,-.1,1, 360

      X_ROTATION 45, 1,1,0
      X_ROTATION GETTIMERALL()/100, 0,1,.2
      X_SETSHADER 13
      X_PUTSHADER "t", GETTIMERALL()
      X_DRAWOBJ 0, 0
      SHOWSCREEN
   WEND
```

Refs:

X_LOADSHADER(), X_SETSHADER

X_ROTATION

X_ROTATION phi#, x#, y#, z#

Rotiert das zunächst zu zeichnende Objekt um die Achse x#, y#, z# um den Winkel phi#. Es können mehrere Rotationen nacheinander auf ein einzelnes Objekt angewandt werden. X_MOVEMENT oder X_SCALING setzen alle Rotationen zurück.

Refs:

X_MOVEMENT, X_MULTMATRIX, X_SCALING

X_SAVEOBJ

X_SAVEOBJ datei$, num#

Speichert das erstellte oder geladene 3D-Objekt mit der Referenznummer num# in den angegebenen Dateinamen. Für ein Beispiel bitte bei OBJ... nachsehen.

Refs:

X_LOADOBJ, X_OBJ...

X_SCALING

X_SCALING fx#, fy#, fz#

Setzt die Skalierung des nächsten 3D-Objekts auf die Faktoren fx#, fy#, fz#. Dieser Befehl muss vor X_ROTATION aufgerufen werden, da er alle vorherigen Rotationen entfernt. Skalierte, inbesondere nicht orthotrope Skalierungen, können Probleme bei X_COLLISION hervorrufen.

Refs:

X_MOVEMENT, X_ROTATION, X_SCALING

X_SCREEN2WORLD

X_SCREEN2WORLD sx#, sy#, sz#, wx##, wy##, wz##

Konvertiert die Bildschirmkoordinaten sx#, sy#, sz# in 3D Weltkoordinaten.

Refs:

X_WORLD2SCREEN

X_SETSHADER

X_SETSHADER index#

Refs:

X_LOADSHADER(), X_PUTSHADER

X_SETTEXTURE

X_SETTEXTURE n#, nbump#

Benutzt ein SPRITE, das vorher mit LOADSPRITE geladen wurde, als aktuelle Textur für 3D-Objekte.
-1 gibt an, dass keine Textur verwendet werden soll.
nbump# gibt an, welche Normal-Map für Dot3-Bump Mapping verwendet werden soll. Die Normalen-Map sollte anstatt mit LOADSPRITE mit LOADBUMPTEXTURE generiert werden. Ein Wert von -1 gibt an, dass BumpMapping deaktiviert werden soll.

Sample:
Siehe Sample: PaperPlanes2 in den 3D Tutorials

Refs:

LOADBUMPTEXTURE

X_SETTEXTUREOFFSET

X_SETTEXTUREOFFSET dx#, dy#

Verschiebt die Texturkoordinaten aller Polygone um dx#, dy#. Damit kann man damit Wasser fließen lassen, oder Wolken bewegen, ohne das 3D Objekt zu ändern.

Refs:

X_SETTEXTURE

X_SPHEREMAPPING

X_SPHEREMAPPING status#$

Schaltet die 3D Ausgabe in den "Shperemapping" Modus um. Damit lassen sich metallische Reflektionen erzeugen. Es muss als aktive Textur eine Kugelprojektion der Umgebung geladen werden. Dazu benutzt man am besten Photoshop oder ein ähnliches Programm. (The GIMP ist Freeware)


Original


Sphere-Map davon

```
// ---------------------------------- //
// Project: SphereMapping - metallic reflection

   X_LOADOBJ "donut.ddd", 1
   // Bild-Daten
   LOADSPRITE    "back.bmp", 1
   LOADSPRITE "reflect.bmp", 0

   // Hauptschleife
   WHILE TRUE
      GETSCREENSIZE screenx, screeny
      STRETCHSPRITE 1, 0,0, screenx, screeny
      phi=phi+GETTIMER()/30
      X_MAKE3D 1, 250, 45
      X_CAMERA 0, 10, 50, 0,0,0

      X_SETTEXTURE 0, -1 // 0=Tex
      X_ROTATION phi, 0, 1, 0.1

      X_SPHEREMAPPING TRUE
      X_DRAWOBJ 1, 0
      X_SPHEREMAPPING FALSE
      SHOWSCREEN
   WEND
```

Refs:

X_FOG, X_MIPMAPPING, X_SETTEXTUREOFFSET, X_SPOT_LT

X_SPOT_LT

X_SPOT_LT num#, col#, x#, y#, z#, dirx#, diry#, dirz#, cutoff#

Schaltet ein Spotlicht ein.

num# ist die Nummer des Lichts (0-7 erlaubt)
col# ist die Farbe. Siehe RGB()
x#, y#, z# sind die Koordinaten der Lichtposition
dirx#, diry#, dirz# spezifizieren einen Vektor, der die Richtung des Lichts angibt.
cutoff# ist der Winkel der Abblende des Lichts. Bei 180 wird ein paralleles Licht erzeugt.
Wird ein Winkel von 0 eingegeben, ist das Licht abgeschalten.

Spezialitäten

Mit X_SPOT_LT erzeugt man Lichtquellen für Spezialeffekte. Dabei ist num# entscheidend für den Effekt.

num# = -1: Bump Mapping

Die 2. Textur wird als "Beulen" Textur verwendet. Dabei sind dunkle Pixel auf der Bump-Textur eine Senke und helle eine Erhebung.
Es werden nur die Koordinaten x#, y#, z# ausgewertet. Man müssen vorher mit LOADBUMPTEXTURE und LOADSPRITE 2 Texturen geladen worden sein und mit X_SETTEXTURE angegeben. Für ein Beispiel siehe LOADBUMPTEXTURE

num# = -2 : Toon Shading

Auch Cel-Shading order Cartoon Rendering genannt. Hier wird eine Lichtberechnung durchgeführt, die nicht linear, sondern in 2 Helligkeitsstufen geschieht. Dadurch wird ein Effekt erzielt, der einem Comic sehr ähnlich ist. Wenn man mit X_SETTEXTURE eine Texut angibt, so wird diese darübergelegt. Somit kann man z.B. Gesichter oder andere Details auf die Figur bringen. Es wird nur die Position des Lichts berücksichtigt: x#, y#, z#.

num# = -3: Schatten

Echtzeit Schatten mit Stencil Buffer. Funktioniert so:
- die ganze Szene zeichnen
- X_SPOT_LT -3, 0, x,y,z, 0,0,0, 360
dieses Licht ist die Lichtquelle und Schattenrichtung
- X_DRAWOBJ für alle schattenwerfende Objekte
- X_MAKE2D/SHOWSCREEN/X_MAKE3D
Das zeichnet die Schatten letztendlich.

```
// -------------------------------- //
// Project: ToonShading
// Start: Wednesday, October 07, 2004
// IDE Version: 2.41007

 CreateTorus(1, 5, 10, 24, 24, 2, 2)
 // Bild-Daten / Image data
// LOADSPRITE      "image.bmp", 0
 FILLRECT 0,0,64,64,RGB(0xff,0xff,0xff)
 FILLRECT 0,28,64,36,RGB(0,0,0xff)
 PRINT "GLBASIC", 0,24
 GRABSPRITE 0, 0,0, 64,64

 // Hauptschleife / main loop
 WHILE TRUE
    FILLRECT 0,0,640,480,RGB(0,0,200)
    phi=phi+GETTIMER()/30
    X_MAKE3D 1, 250, 45
    X_CAMERA 0, 10, 100, 0,0,0
    // Licht Nr. -2 is Cel-Shading Postition / Light no -1 is cel shading position
    X_SPOT_LT -2, 0, 0,0,100, 0,0,0,90
    X_SETTEXTURE 0, -1 // 0=Tex
    X_ROTATION 90, 1,0,0
    X_ROTATION phi, 0, 1, 0.1
    X_DRAWOBJ 1, 0
    SHOWSCREEN
 WEND

// ------------------------------------------------------------- //
// -=#  CREATETORUS  #=-
// Donut Objekt machen / Make a donut object
//
// By Samuel R. Buss
// http://math.ucsd.edu/~sbuss/MathCG
// ------------------------------------------------------------- //
FUNCTION CreateTorus: num, MinorRadius, MajorRadius, NumWraps, NumPerWrap, TextureWrapVert, TextureWrapHoriz
LOCAL i, di, j, wrapFrac, wrapFracTex, phi, thetaFrac, thetaFracTex, theta
LOCAL x, y, z, r

 X_AUTONORMALS 2 // smooth
 X_OBJSTART num
 FOR di=0 TO NumWraps-1
  FOR j=0 TO NumPerWrap
   FOR i=di+1 TO di STEP -1
    wrapFrac = MOD(j, NumPerWrap)/NumPerWrap
    wrapFracTex = j/NumPerWrap
    phi = 360*wrapFrac
    thetaFrac = (MOD(i, NumWraps + wrapFracTex)/NumWraps
    thetaFracTex = (i+wrapFracTex)/NumWraps
    theta = 360*thetaFrac
    r = MajorRadius + MinorRadius*COS(phi)
    x = SIN(theta)*r
    z = COS(theta)*r
    y = MinorRadius*SIN(phi)
    X_OBJADDVERTEX x,y,z,  thetaFracTex*TextureWrapVert, 1-wrapFracTex*TextureWrapHoriz, _
                   RGB(0xff,0xff,0xff)
```

```
   NEXT
  NEXT
  X_OBJNEWGROUP
 NEXT
 X_OBJEND
ENDFUNCTION // y
```

Refs:

LOADBUMPTEXTURE, X_AMBIENT_LT, X_AUTONORMALS, X_SPHEREMAPPING

X_SPRITE

X_SPRITE nr#, x#, y#, z#, skalierung#

Zeichnet ein 2D-Sprite an eine 3D-Position. Der Referenzpunkt ist dabei die untere Mitte der Grafik. Dieses Verfahren wird auch "Billboarding" genannt.

Sample:

```
// -------------------------------- //
// Project: Billboards

LOADBMP "back.bmp"
LOADSPRITE "tree.bmp", 0
LOADSPRITE "gras.bmp", 1

WHILE TRUE
   X_MAKE3D 1,1500,45
   MOUSESTATE mx, my, b1, b2
   X_CAMERA SIN(mx)*200, 44, COS(mx)*200, 0, 75, 0

   // Billboard Bäume
   FOR phi=0 TO 360 STEP 30
      FOR ny=0 TO 360 STEP 45
         // ein 3D Sprite, das immer zur Kamera hin zeigt
         X_SPRITE 0, SIN(phi)*500+COS(ny)*300, 0, COS(phi)*300+SIN(ny)*500, .5
      NEXT
   NEXT

   // Gras...
   X_OBJSTART 0
      f=RGB(255,255,255)
      X_OBJADDVERTEX -2000,0,-2000,  0 ,0,f
      X_OBJADDVERTEX  2000,0,-2000, 50, 0,f
      X_OBJADDVERTEX -2000,0, 2000,  0,50,f
      X_OBJADDVERTEX  2000,0, 2000, 50,50,f
   X_OBJEND
   X_SETTEXTURE 1, -1; X_DRAWOBJ 0,0

   X_MAKE2D // LOADBMP benötigt das vor einem SHOWSCREEN
   SHOWSCREEN
WEND
```

Refs:

SPRITE, X_MAKE3D

X_WORLD2SCREEN

X_WORLD2SCREEN wx#, wy#, wz, sx##, sy##, sz##

Berechnet aus den 3D-Koordinaten wx, wy, wz die Bildschirmkoordinaten in sx##, sy##, sz##. Achtung! Um die 2D-Bildschirmkoordinaten zu verwenden, muss man ein X_MAKE2D aufrufen.

Refs:

X_MAKE2D, X_MAKE3D, X_SCREEN2WORLD, X_SPRITE

ZOOMSPRITE

.

Refs:

SPRITE

Programmfluss

.

Refs:

ALLOWESCAPE, AUTOPAUSE, BREAK, BYREF, CALLBACK, CASE, CONTINUE, DELETE, DIM, DIMDATA, DIMDEL, DIMPUSH, ELSE, END, ENDFUNCTION, ENDIF, ENDSELECT, FOR, FOREACH, FUNCTION, GLOBAL, GOSUB, GOTO, HIBERNATE, IF, INLINE, LOCAL, NEXT, REDIM, RETURN, SELECT, SHELLCMD(), STATIC, STEP, SUB, THEN, TO, TYPE, WEND, WHILE

Programmfluss

.

Refs:

ALLOWESCAPE, AUTOPAUSE, BREAK, BYREF, CALLBACK, CASE, CONTINUE, DELETE, DIM, DIMDATA, DIMDEL, DIMPUSH, ELSE, END, ENDFUNCTION, ENDIF, ENDSELECT, FOR, FOREACH, FUNCTION, GLOBAL, GOSUB, GOTO, HIBERNATE, IF, INLINE, LOCAL, NEXT, REDIM, RETURN, SELECT, SHELLCMD(), STATIC, STEP, SUB, THEN, TO, TYPE, WEND, WHILE

Mathematik und Logik

.

Refs:

ABS(), ACOS(), AND, ASIN(), ATAN(), COS(), DEC, FINDPATH(), INC, LEN, LET, MAX(), MIN(), MOD(), OR, POW(), RND(), SIN(), SQR(), TAN(), X_SCREEN2WORLD, X_WORLD2SCREEN, bAND(), bNOT(), bOR(), bXOR()

Eingabe-Ausgabe

.

Refs:

COPYFILE, DEBUG, DOESFILEEXIST(), FILEREQUEST$(), FORCEFEEDBACK, GETCOMMANDLINE$(), GETCURRENTDIR$, GETDIGI...(), GETFILE, GETFILELIST(), GETFONTSIZE, GETJOY...(), GETSCREENSIZE, GETTIMER(), GETTIMERALL(), INIGET$, INIOPEN, INIPUT, INKEY$, INPUT, JOYSTATE, KEY(), KEYWAIT, KILLFILE, LOADBMP, LOADBUMPTEXTURE, LOADFONT, LOADSOUND, LOADSPRITE, MOUSEAXIS, MOUSESTATE, MOUSEWAIT, NET..., NETGETIP$(), NETWEBGET(), PLATFORMINFO$(), PRINT, PUTFILE, SAVEBMP, SAVESPRITE, SETCURRENTDIR(), SETFONT, SETMOUSE, SHELLCMD(), SHELLEND, SYSTEMPOINTER

Sound und Multimedia

.

Refs:

HUSH, LOADSOUND, LOOPMOVIE, PLAYMOVIE, PLAYMUSIC, PLAYSOUND(), SOUNDPLAYING(), STOPMUSIC

Wortmanipulation

.

Refs:

ASC(), CHR$(), FORMAT$(), INSTR(), LCASE$(), MID$(), REPLACE$(), SPLITSTR(), UCASE$()

Grafik (2D)

.

Refs:

ALPHAMODE, BLACKSCREEN, BLENDSCREEN, DRAWLINE, ENDPOLY, FILLRECT, GETFONTSIZE, GETPIXEL(), GETSCREENSIZE, GETSPRITESIZE, GRABSPRITE, ISFULLSCREEN(), LOADBMP, LOADFONT, LOADSPRITE, LOOPMOVIE, PLAYMOVIE, POLYVECTOR, PRINT, RGB(), ROTOSPRITE, ROTOZOOMSPRITE, SAVEBMP, SAVESPRITE, SETFONT, SETPIXEL, SETSCREEN, SHOWSCREEN, SPRITE, STARTPOLY, STRETCHSPRITE, USEASBMP, VIEWPORT, ZOOMSPRITE

Grafik (3D)

.

Refs:

LOADBUMPTEXTURE, VIEWPORT, X_AMBIENT_LT, X_AUTONORMALS, X_CAMERA, X_CAMERAUP, X_CLEAR_Z, X_COLLISION(), X_COLLISIONAABB(), X_COLLISIONRAY(), X_CULLMODE, X_DOT, X_DRAWANIM, X_DRAWAXES, X_DRAWOBJ, X_FOG, X_GETFACE, X_LINE, X_LOADOBJ, X_LOADSHADER(), X_MAKE2D, X_MAKE3D, X_MIPMAPPING, X_MOVEMENT, X_MULTMATRIX, X_NUMFACES(), X_OBJ..., X_POPMATRIX, X_PRINT, X_PUSHMATRIX, X_PUTSHADER, X_ROTATION, X_SAVEOBJ, X_SCALING, X_SCREEN2WORLD, X_SETSHADER, X_SETTEXTURE, X_SETTEXTUREOFFSET, X_SPHEREMAPPING, X_SPOT_LT, X_SPRITE, X_WORLD2SCREEN