Das Magazin für PureBasic-Entwickler

Tux-Basic

PureBasic für den Pinguin

-Gute Aussichten: Erste Blicke auf PureBasic 4.0 für Linux

-Plattformübergreifendes Arbeiten: PureWinLin

.Net-Basic

Dietmar Schölzel im Interview: Was bringt PB.Net?

Gut-Basic

Entwickler des Monats: Thorsten Hoeppner (EasySetup)

Ordnungs-Basic

Übersicht und Ordnung im Code

Macro oder Procedure?

Frank Wiebeler zeigt die Vor- und Nachteile von Macros und Prozeduren

Parsen leichtgemacht

Effizient und einfach Text-dateien auswerten

www.pb-cm.net

Editorial

Hier spricht die Redaktion

In dieser ersten Ausgabe wollen wir an dieser Stelle die Entstehung des Magazins kurz umreißen.

Alles begann damit, dass Peter im Mai dieses Jahres auf die Idee kam, ein Magazin für PureBasic herauszugeben. Kurze Zeit darauf wurde dieses Vorhaben in der PureBasic-Lounge publik und Philipp stieß dazu, woraufhin ein reger PN-Kontakt mit Designstudien und ersten Artikel-Entwürfen folgte zwischen den beiden folgte. Etwas später, Anfang Juni, ging Chris mit einer ähnlichen Idee im PureBoard an die Öffentlichkeit. Aufgrund der überwältigenden und positiven Reaktionen und trotz anfänglichem Misstrauen beschlossen wir, beide Projekte unter dem Namen des "Ur-Magazins", *PB.CM*, zusammenzulegen.

In den folgenden Monaten erkannten wir erst, wie viel Arbeit wir uns aufgeladen hatten. Wir zogen von Entwickler zu Entwickler und bettelten um Artikel. Wir layouteten hin, layouteten her und wechselten dabei viermal die Software. Wir besprachen, diskutierten, stritten und fanden Kompromisse. Die Früchte dieser unserer Arbeit und der vieler freiwilligen Autoren liegen nun der Allgemeinheit - euch - zur Begutachtung vor.

Über konstruktives Feedback und Verbesserungsvorschläge freuen wir uns übrigens auch weiterhin!

Viel Spaß mit PB.CM - dem weltweit ersten PureBasic-Magazin.

Peter Kastberger | Christopher Higgs | Philipp Schmieder

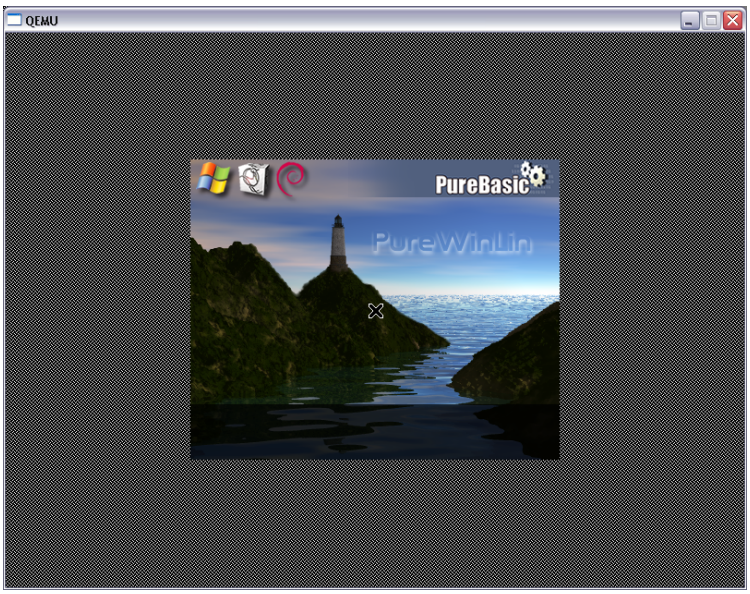
Inhalt

Eine Brücke schlagen

von Peter Kastberger

Liebäugeln mit dem Pinguin

Es passiert immer wieder. Sorglose Programmierer werden von dem Drang erfasst, sich auf ein Stelldichein mit dem bekannten Pinguin Tux einzulassen - seines Zeichens Maskottchen des freien Betriebssystems GNU/Linux [1] - um herauszufinden, ob der gerade geschriebene Code wirklich plattformunabhängig ist. Denn dies ist ja eines der Markenzeichen von PureBasic. Nur wie geht man diese Sache am besten an?



Der GNOME-Begrüßungsbildschirm von PureWinLin

Für all jene, die davor zurückschrecken, gleich eine ganze Partition dem ungestümen Pinguin zu opfern, oder ihre Zeit besser nutzen wollen, als zwischen mehreren Sytemen hin- und her zu booten, wird im folgenden Artikel beschrieben, wie man mit dem Emulator Qemu [2] und einem passenden Linux-Image seine PureBasic-Sourcecodes ganz einfach testen kann.

Auf der Suche nach den Quellen

Sowohl Quemu, als auch passende Images sind im weltweiten Netz eigentlich sehr leicht aufzutreiben. Da wir aber besondere Anforderungen an unser System stellen, muss es schon was Besonderes sein. Immerhin benötigen wir eine Vielzahl von Bibliotheken, damit wir wirklich alle von PureBasic gebotenen Funktionen nutzen können.Am einfachsten ist es, die Dateien aus dem einem Thread im englischen PureBasic-Forum [3] zu nutzen. Beach stellt dort alle notwendigen Dateien zur Verfügung.

Die wichtigsten Features:

- keine Konfiguration notwendig
- ein aktuelle Debian 3.1 Stable Linux System
- GNOME 2.8 als Oberfläche
- Firefox als Browser
- Vorkonfigurierte PureBasic-Umgebung mit der Beta-IDE von Freak!

Die Image-Datei ist unter [4] verfügbar; Qemu unter [5].

Sobald man die Dateien auf der Platte hat, kann es auch schon losgehen. Zuallererst müssen alle Dateien in einem Verzeichnis abgelegt werden.

Wichtig hierbei ist: Die Linux-Image Datei *debian.img*, die im Zip-Paket *PureWinLin.zip* enthalten ist, liegt hier nun entpackt im selben Verzeichnis wie *qemu.exe*. Die zweite wichtige Datei ist *qemu-win.bat*. Diese muss

unseren Anforderungen angepasst werden. Immerhin soll ja das Debian-Image gestartet werden. Folgender Inhalt der Datei lässt das Debian-Image starten:

```
@ECHO OFF
START qemu.exe -L . -m 128 -hda debian.img
-soundhw sb16 -localtime
CLS
EXIT
```

Wahlweise kann auch die ausführbare Datei *PureWinLin.exe* (im PureWinLin-Verzeichnis) benutzt werden, um Qemu samt Image zu starten. Hier hat man die Möglichkeit, zu bestimmen, wie viel RAM-Speicher man für die Anwendung aufbringen möchte und ob sie im Vollbildmodus gestartet werden soll.

Sind die Einstellungen alle vorgenommen, kann auch schon das erste Mal gestartet werden. Voller Erwartung

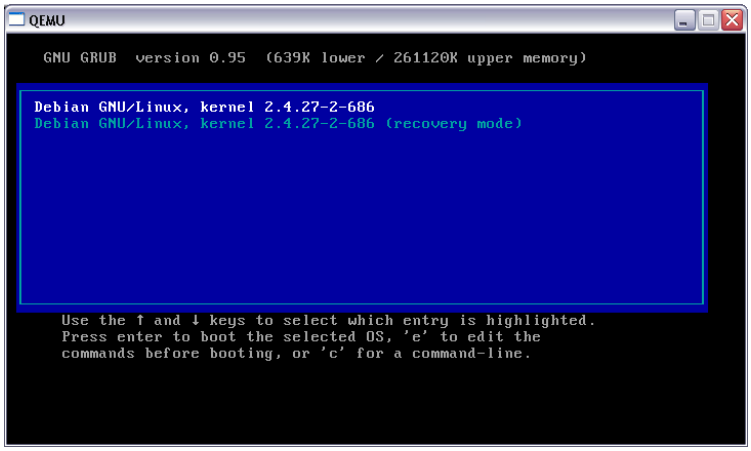


Abbildung 1: Der Bootloader GRUB

führt man also die Batchfile aus und trifft so zunächst auf GRUB, den Bootloader (Abbildung 1). Wer einen Multi-OS-PC hat, kennt das Prinzip wahrscheinlich. Mit ihm lässt sich über die Cursortasten das gewünschte System starten. In unserem Fall wollen wir das Debian-Image dazu bewegen, uns seine Dienste zu leisten, also wird dieses auch ausgewählt. Da GRUB ohnehin so vorkonfiguriert ist, würde das richtige System nach fünf Sekunden von selbst starten.

Der nur einige Sekunden andauernde Bootvorgang zeigt uns die Kernelmeldungen, bis der sogenannte X-Server gestartet wird. Dieser ist für die Grafik verantwortlich und macht sich durch eine graue Oberfläche mit X-förmigen Mauszeiger bemerkbar

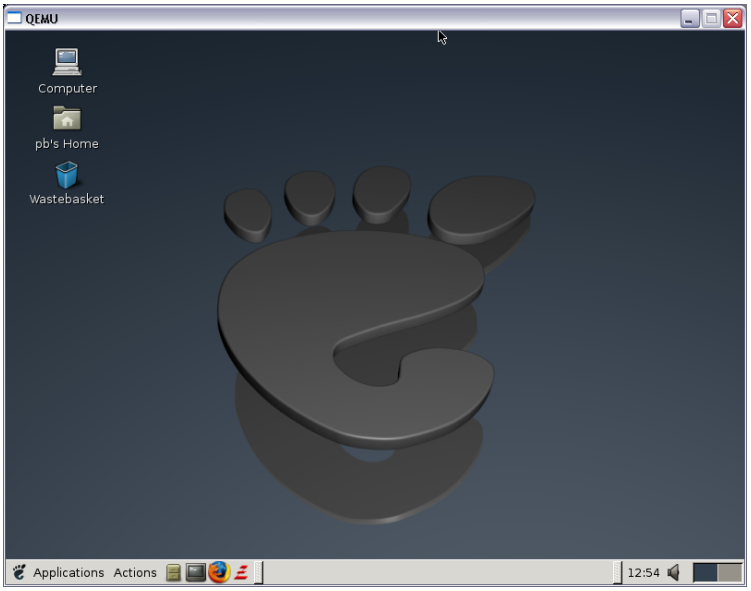


Abbildung 2: Der GNOME-Desktop mit PB-Symbol

Entgegen aller gängigen Sicherheitsvorkehrungen, ist PureWinLin so konfiguriert, dass der GNOME-Desktop (Abbildung 2) automatisch startet, ohne dass eine Benutzerauthentifizierung erfolgt. Mann kann dies einfach umstellen, aber für eine Emulation dürfte es aber wohl kaum von Nöten sein. Beim Laden des Desktops blickt uns noch ein netter Splashscreen entgegen, der schon sehr schön andeutet, worauf wir uns einlassen.

Nachdem das System nun geladen ist, steht es uns zum Arbeiten zur Verfügung. Los gehts!

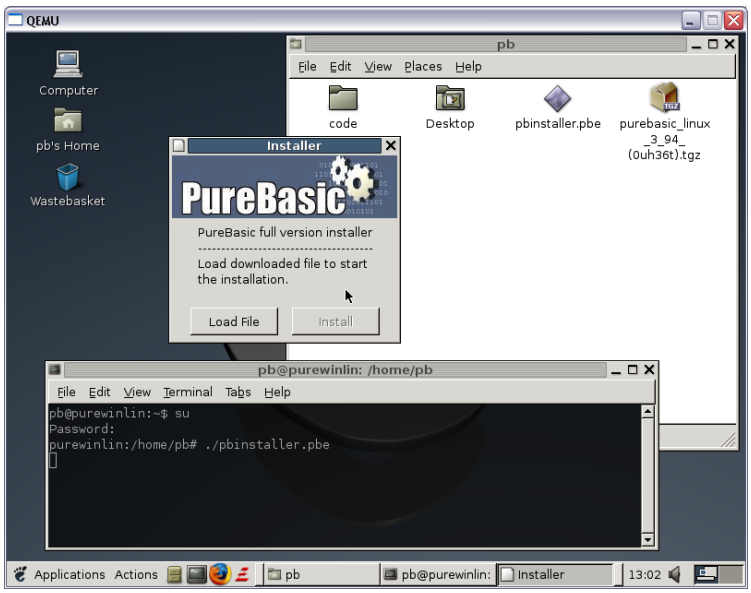


Abbildung 3: Das Installationsprogramm für die PB-Vollversion

Sogar einen PB-Button gibt es schon. Bei der Erstellung des Linux-Images wurde also ganze Arbeit geleistet. Installiert auf diesem System ist die Demo-Version von PureBasic. Um an die Vollversion zu gelangen muss diese von der PureBasic Homepage heruntergeladen werden. Dazu ist natürlich ein PB-Account notwendig. Die Internetverbindung der Windows-Testmaschine wurde von Qemu ohne Probleme übernommen. Es dürften also keine Einstellungen notwendig sein

Der Installer von Beach [6], welcher die vorhandene Demo-Installation automatisch patcht, sollte als Benutzer *root* (Administrator) ausgeführt werden. Somit wird sicher gestellt, dass man auch Schreibrechte in allen notwendigen Verzeichnissen hat.

Sobald der Installer erscheint (Abbildung 3), muss mittels *Load File* die heruntergeladene Archivdatei ausgewählt werden.

Nachdem man die Datei ausgewählt hat, werden vom Installer alle notwendigen Dateioperationen durchgeführt. Der sonst so umständliche Installationsprozess von PureBasic unter Linux entfällt dadurch. Die Installation ist abgeschlossen und es kann mit der Vollversion von PureBasic unter Linux munter drauf los programmiert werden.

Links:

[1] http://de.wikipedia.org/wiki/Linux

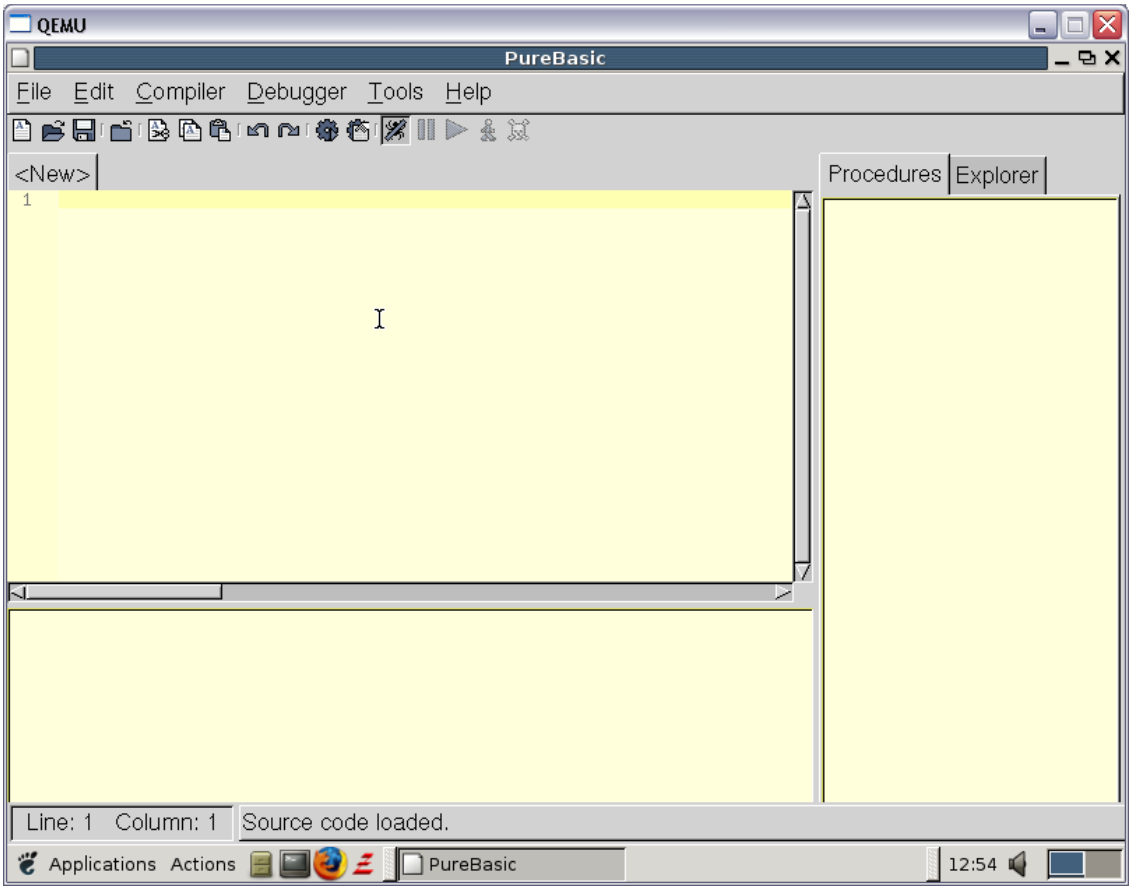
[2] http://fabrice.bellard.free.fr/qemu/

[3] http://www.purebasic.fr/english/viewtopic.php?t=15844

[4] http://www.penguinbyte.com/software/PureWinLin/download/?filename=PureWinLin.zip

[5] http://sbougribate.free.fr/Files/Emulation/Qemu/Qemu-0.8.2-gcc4.1.1-KarLKoX.rar

[6] http://PureBasic.myftp.org/?filename=files/3/projects/pbinstaller.pbe



Die PureBasic-IDE in Qemu

Nicht auf die Länge kommt es an, ...

von Frank Wiebeler

".... sondern wie man Ihn einsetzt." Wer kennt diesen Spruch nicht. Zu Urzeiten, als die Programmierer noch auf Dinosauriern zur Arbeit ritten, bekamen sie bei langen Quelltexten arge Probleme dabei, ihr Machwerk noch zu durchschauen und zu ändern. Wie aber jeder gute Informatiker weiß, können Programmierschwierigkeiten oftmals gelöst werden, indem man sich am Wissen der Reallife.exe bedient. So zermarterten sich die klügsten Köpfe ihrer Zeit die Gehirne und alsbald gab es zweierlei Lösungsansätze, um Quelltextteile zusammen zu fassen:

Zum einen gab es die **Procedure**: Eine Procedure liegt nach dem Compilieren immer noch als
Procedure vor. Sie liegt also nur einmal im Speicher. Ihr Nachteil ist allerdings, dass sie bei jedem Aufruf mindestens 3, zumeist jedoch mehr Recheneinheiten benötigt, um gestartet zu werden.
Zum zweiten gab es das **Macro**:
Ein Macro ist ein Textbaustein, der beim Compilieren direkt eingesetzt wird. Der Vorteil eines Macros ist die Tatsache, dass durch das direkte Einsetzen keine zusätzlichen Recheneinheiten benötigt werden. Der Nachteil ist allerdings, dass durch das direkte Einsetzen die ausführbare Datei größer wird.

Nach Jahren der Procedure-Nutzung ist mit dem Release von PureBasic 4.0 nun der Tag gekommen, an dem es heißt: "So, lieber Programmierer, nun musst du dich entscheiden.".
Mit dem vierten Buch Fred hält das Macro Einzug in die PureBasic-Gemeinde, ein großartiges Feature, das einen genaueren Blick wert ist. Nun ergibt sich aus dieser Errungenschaft aber auch eine wichtige Frage: **Wann nutze ich "Procedure" und wann "Macro"?**
Um diese Frage zu beantworten, sollte man zuerst genauer betrachten, wo sie sich überhaupt stellt, denn nicht jedes Macro kann auch als Procedure erstellt werden, genauso wie nicht jede Procedure durch ein Macro ersetzt werden kann.

Was geht?

Proceduren mit ProcedureReturn lassen sich oftmals nur schwer in ein Macro umwandeln, sofern sie keine Einzeiler sind.

Während sich

```
Procedure Median(wert1.f,wert2.f)
  ProcedureReturn wert1+(wert1-wert2)/2
EndProcedure
```

Locker durch

```
Macro Median(wert1,wert2)
  (wert1+(wert1-wert2)/2)
EndMacro
```

ersetzen lässt, ist es im Falle von

```
Procedure GibFarbe(bild,x,y)
  StartDrawing(ImageOutput(bild))
    clr=Point(x,y)
  StopDrawing()
  ProcedureReturn clr
EndProcedure
```

unmöglich, ein Macro zu benutzen, ohne den Rest des Quelltextes zu verändern.

```
Macro GibFarbe(bild,x,y)
  StartDrawing(ImageOutput(bild))
    clr=Point(x,y)
  StopDrawing()
EndMacro
```

Hier muss entweder ein zusätzlicher Parameter angegeben werden, nämlich die Variable, die den Farbwert erhalten soll, oder aber man muss jedesmal im Hauptquelltext *clr* benutzen, um den Farbwert zu übergeben.

Sollte man sich an dieser Stelle für ein Macro in oben gezeigter Form entscheiden, wäre es hilfreich, die Variable *clr* zum Beispiel als *MacroReturn_GibFarbe* zu benennen. Dies hat zum einen den Vorteil, dass beim Durchlesen des Quelltextes trotz geänderter Syntax die Herkunft des Wertes erkennbar wird. Zum anderen läuft man aber auch nicht so schnell Gefahr, Probleme zu bekommen, da man zwei Variablen gleich benannt hat.

Das ewige Kreisen um sich selbst

Das ewige Kreisen um sich selbst alias Rekursion; hierbei handelt es sich um den altbekannten Selbstaufruf, bekannt aus FloodFill-Algorithmus und Sortieralgorithmen. Eine Procedure mit direkter oder indirekter Rekursion (direkt: Procedure1 ruft Procedure1 auf; indirekt: Procedure1 ruft Procedure2 auf, Procedure2 ruft Procedure1 auf) lässt sich keinesfalls durch ein Macro ersetzen.

Als Beispiel hierfür einmal dieser kurze Quelltext:

```
Macro Mehr()
  a=a+1
  Mehr()
EndMacro
Mehr()
```

Ein Vergleichbarer Quelltext ohne Macro würde so erstellt:

```
If CreateFile(1,"Datei.pb")
  Repeat
    WriteStringN(1,"a=a+1")
  ForEver
  CloseFile(1)
EndIf
```

Wer diesen Code bis zum Ende ausführen kann, verdient meinen Respekt ;)

Darum merke: "Ruft's sich selber auf, schreib nicht Macro drauf!"

Protektionismus unerwünscht

In den meisten Proceduren gibt es Static oder Protected Variablen, Macros kennen allerdings keine eigenen Variablenarten. Ein all zu großes Problem stellen diese Variablen nicht dar, wenn man weiß, wie man sie einsetzen muss. Zuerst ist wiedereinmal auf die Benennung der Variablen zu achten; damit es nicht zur Namensüberschneidung und Doppelbenennung mit dem normalen Quelltext kommt, wäre *Macro_MacroName_Variablenname* eine einfache, wenn auch sehr lange Variante, den Variablen einzigartige Namen zuzuordnen. Bei **Protected-**Variablen sollte am Anfang jedes Macros der Wert der Variable zurück auf 0 gesetzt werden, um das Verhalten von Protected-Variablen zu simulieren. Statt Static muss die Static-Variable globalisiert werden, damit sie beim Macroaufruf innerhalb einer Procedure den gleichen Wert erhält wie beim Aufruf außerhalb. Sofern die Variable keinen Anfangswert erhalten muss, kann die Zeile "Global Variable" auch in das Macro geschrieben werden. Dies hat weder Einfluss auf die Dateigröße der Exe, noch auf die Richtigkeit der Werte.

Einen Sprung in der Schüssel

Sprungmarken können manchmal ganz nützlich sein, in Macros sind sie aber nicht möglich.

```
Global a.l
Macro Baum()
  a=a+1
  Goto posi
  a=a+5
  posi:
  Debug a
EndMacro

Baum()
Baum()
```

Der Grund hierfür liegt darin, dass, nachdem das Macro beim Compilieren eingesetzt wurde die Sprungmarke doppelt existiert. Die einzige Möglichkeit, hier ein Macro zu benutzen liegt in der Nutzung eines Parameters zur Namenserweiterung der Sprungmarke.

```
Global a.l
Macro Baum(var)
  a=a+1
  Goto posi#var
  a=a+5
  posi#var:
  Debug a
EndMacro

Baum(1)
Baum(2)
```

Darum merke: "Ist ein Sprungpunkt markiert, wird kein Macro riskiert!"

Hat nun eine Procedure all diese Einschränkungen überwunden und es stellt sich heraus, das sie würdig ist, Meister Macro Platz zu machen, stellt sich die Frage:

Was lohnt?

Diese Frage ist natürlich immer philosophischer Natur, allerdings gibt es zumindest einige Haltepunkte, nach denen man entscheiden kann. Auf Seiten des Users stehen sich Dateigröße sowie Speicherverbrauch und Geschwindigkeit gegenüber, dem Programmierer stellt sich zudem die Frage nach Übersichtlichkeit und Aufwand.

Eine Procedure, die weder eigene Variablen noch einen Rückgabewert enthält, stellt beim Umbau in ein Macro den geringsten Anspruch an den Programmierer. Somit bleiben noch die Parameter Dateigröße, Speicherverbrauch und Geschwindigkeit, wobei aber Dateigröße und Speicherverbrauch meist äquivalent sind. Die Umwandlung einer Procedure in ein Macro lohnt sich besonders beim häufigen Aufruf innerhalb von Schleifen oder Rekursionen(wenn ein Macro innerhalb einer Rekursiven Procedure aufgerufen wird). Schlussendlich muss aber nun noch eingeschränkt werden, in welchen Dimensionen gedacht wird, wenn man von einer Optimierung spricht. Wie bereits erwähnt, verbraucht jeder Procedureaufruf mindestens 3 Recheneinheiten. Ein Rechner mit einem GHz Rechenleistung kann aber pro Sekunde 333.333.333 solcher Aufrufe tätigen. Andersherum kann man aber auch ein Stückchen Quelltext 100-fach hintereinandersetzen, ohne dass die ausführbare Datei so sehr aufgebläht wird, wie wenn man mittels IncludeBinary eine Bitmap-Datei im Format 100*100 eingefügt hätte. Zudem bieten Packer wie UPX gute Chancen, diesem Bläh-Effekt entgegen zu wirken.

Ordnungswahn?

Bei der Entscheidung, ein Macro oder eine Procedure zu wählen sollte stets ein Gedanke an die Zukunft verschwendet werden. Es stellt sich die Frage, ob man morgen noch versteht, was man heute programmiert. Bei einer Procedure mit Parametern sieht man beim Eintippen des Aufrufs direkt die Datentypen, die verlangt sind,

wohingegen bei Macros die Angabe von Datentypen nicht möglich ist. Hierdurch kann es bei ungenauer Parameterbenennung zu Verständnisfehlern kommen, die dann einige Zeit in Anspruch nehmen, bevor man sie findet. Die Flickschußterei des "Quelltext#Parameter#Quelltext" sollte unter umständen bei ProcedureAufrufen oder Konstanten benutzt werden. Bei Variablen wird dies schnell unübersichtlich. Beispiele wie...

```
Global Spieler1x.l
Global Spieler1y.l
Global Spieler2x.l
Global Spieler2y.l
Macro Move(p_p)
  p_p#x= p_p#x+4
  p_p#y=p_p#y+5
EndMacro
Move( Spieler1)
Move(Spieler2)
```

...bieten gute Möglichkeiten, stattdessen mit Structures zu arbeiten. Diese sind nicht nur übersichtlicher (Spieler1x zu Spieler1\x), sondern auch schneller erweiterbar.

```
Structure Punkt
  x.l
  y.l
EndSTructure
Global  Spieler1.punkt
Global Spieler2.punkt
Procedure Move(*p_p.punkt)
  *p_p\x=*p_p\x+5
  *p_p\y=*p_p\y+5
Endprocedure
Move(Spieler1)
Move(Spieler2)
```

Man bedenke den Aufwand für Beispiel1 bzw. Beispiel 2, wenn man noch einen 3.,4.5.... Spieler oder eine zusätzliche Z-Koordinate einbauen wollte.

Das Ende aller Fragen?

Nachdem nun alle Anwendungsmöglichkeiten (zumindest alle, die dem werten Herrn Autor eingefallen sind ;)) beleuchtet wurden, wird man sich an einigen Stellen wohl immer noch nicht sicher sein, was man nun nutzen soll. In diesem Fall gilt die Frage, ob sich das Nachdenken überhaupt lohnt. Ein Aufruf an einer Stelle? Egal, was man nutzt, es wird niemanden stören. Eine Milliarde Aufrufe, aber sehr langer Quelltextteil und Aufrufe sind auf viele Verschiedene Quelltextteile verstreut? Da lohnt sich das Fragen. Am besten wägt man zunächst die Pros und Contras für das eine und das andere ab und fragt dann im Forum nach. Aus so einem Thread ergeben sich teils sehr interessante Diskussionen. Ansonsten sollte es doch kein all zu großer Aufwand sein, es einmal aus zu testen und zu sehen, was man für besser empfindet. Optimierung hängt halt immer von der Sichtweise ab.

Und wirst du voller Fehler sein,..

Und wirst du voller Fehler sein,.. dann schlag ich mir den Schädel ein. Genau so wirkt sich ein Fehler in einem Macro nämlich aus. Zwar wird guter Wille gezeigt, Macrofehler vernünftig an zu zeigen, aber wie so oft ist der Wunsch noch etwas zu fern der Realität. Und so kann es dann schonmal passieren, dass man so locker vor sich hin buggt im Vertrauen auf die korrekte Fehleranzeige des Compilers bzw. Debuggers (je nachdem, ob Syntaxfehler oder Fehler im Verlauf) und dann wird man durch die lustige Meldung "SyntaxError" belästigt, die aber nur die Zeile des Macroaufrufs markiert, nicht aber den wirklichen Fehlerpunkt. Und schon kann das lustige Rätselraten losgehen. "In welcher meiner XXX-Zeilen mag er sich verstecken, der Belzebub, oder ist es doch wieder ein Fehler in einem Macro, das innerhalb dieses Macros aufgerufen wird?". Noch besser ist dies allerdings bei alternativen Editoren wie JaPBe, wo es sich nicht hin und

wieder ereignet, sondern leider Standard ist. Die Fehlersuche kann Ewigkeiten dauern und so missversteht mancher PBler die Meldung "SyntaxError" als Abwandlung des beliebten "Knock your head on Keyboard to return". Wo wir dann wieder bei der Überschrift wären....

Ein kleiner Tipp am Rande: Wenn ihr in einem Macro oben genanntes Problem habt, versucht das Macro in eine Procedure um zu wandeln. Dann wird euch die Zeile direkt angezeigt.)

Parsen leicht gemacht

von Marc-Sven Rudolf

Parsing ist oft eine Angelegenheit die Geschwindigkeit verlangt da sie ja nur eine vorbereitende Maßnahme ist und daher noch weitere Maßnahmen auf sie folgen.

Ich möchte zeigen, wie man das Maximum an Geschwindigkeit aus einem Parser herausholen kann.

Basis der Hochperformance ist eine Verarbeitung des zu parsenden Strings direkt über Einzelbytes.
Hierzu wird der Parsing-Prozedur einen Pointer auf einen String übergeben während die Prozedur selbst den String als mit *BYTE* strukturierten Speicherblock ansieht.
Im Beispiel sähe das so aus:

```
Procedure Parse (*CodeData.Byte)

EndProcedure

Text.s="Ich bin ein Test"
Parse (@Text)
```

Wir haben dadurch die Möglichkeit den String Byte für Byte zu untersuchen und nach Entsprechungspunkten zu parsen. Um aber überhaupt etwas parsen zu können bedarf es von Vornherein der Kenntnis über die Syntaxregeln nach denen Der Text geparst werden soll.
Nehmen wir einmal, wir möchten in einen String alle Passagen innerhalb Hochkomata die sich ausserdem noch innerhalb von Rundklammern befinden gegen eine neue Zeichenkette tauschen.
Das ginge nach der Byte-Methode kinderleicht wie im Beispiel zu sehen ist:

```
Procedure.s Parse (*CodeData.Byte,Change.s)
  Protected Clamp.l, tickmark.l,tmp.s

  While *CodeData\b<>0
    Select $FF & *CodeData\b
    Case $28; Klammer auf
      If Clamp.l=#False
        Clamp.l=#True
      EndIf
    Case $29; Klammer zu
      If Clamp.l=#True
        Clamp.l=#False
      EndIf
    Case $27; Hochkomma
      If Clamp.l=#True
        If tickmark.l=#False
          tickmark.l=#True
          tmp.s+Change.s
        Else
          tickmark.l=#False
        EndIf
        EndIf
    Default
      If tickmark.l=#False
        tmp.s+Chr(*CodeData\b)
      EndIf
    EndSelect
    *CodeData+1
  Wend
ProcedureReturn tmp.s
EndProcedure


Text.s="Hallo ('test'), wie geht es dir ? "
Text.s+"Du bist doch ('test'), oder ?"
Debug Parse (@Text,"Peter")
```

Wie in diesem Beispiel deutlich zu sehen war, wurden nach Entdeckung von Entsprechungspunkten Schalter geöffnet und geschlossen.

So weiß der Parser zum Beispiel beim Schalten von *Clamp.l* auf *#True* „Aha, eine Klammer wurde geöffnet."

Oder beim Schalten von *tickmark.l* auf *#True*, dass ein Hochkomma geöffnet und beim Schalten auf *#False* ein Hochkomma geschlossen wurde.

Der wesentliche Teil in unserem Beispielcode ist jedoch der strukturierte Pointer **CodeData.Byte*.

Nach jedem Schleifendurchlauf wird dem Pointer ein Byte hinzuaddiert.

Da ein Pointer eine Adresse ist wird das Element **CodeData\b* nur eine Strasse weiter ausgelesen und enthält somit immer jeweils das nächste Byte nach der Addition.

Oft haben Parser aber auch die Aufgabe einen Text vorzubereiten um mit ihm anschließend Weiteres anzustellen.

So zum Beispiel ein Interpreter.

Dieser besteht in der Regel aus einem Preparser, der den Code in einen leichter verarbeitbaren Zustand bring, dann auch noch ein Parser der den Interpreter damit unterstützt, die zu interpretierenden Elemente vor zu filtern.

Da es allerdings den Rahmen dieses Artikels sprengen würde eine Interpretersystem zu schreiben, beschränken wir uns treu dem Titel auf das Parsen in einem Interpretersystem.

Als Beispielaufgabe nutzen wir eine Baumstruktur wie sie bei HTML und XML verwendet wird.

Diese Art des Parsens unterscheidet sich nicht sehr vom vorigen Beispiel.

Lediglich das Speichern der geparsten Strings verhält sich anders.

Denn diese werden nur temporär gebuffert und am Ende einer geparsten Anweisung, in unserem Fall ein Tag, in eine Job-Liste gespeichert.

Ein Scriptcode ,wie wir ihn nun parsen werden benötigt eindeutige und etwas kompliziertere Syntaxregeln.

Führen wir uns anhand eines Scriptbeispiels vor Augen welche Syntaxregeln aufgestellt werden müssen.

```
<tag parameter="">
Dies ist ein Test
</tag>
```

Die Regeln sind hier deutlich erkennbar.

In der folgenden Liste stellen wie die Regeln dar.

Zeichen	Schalter-Aktionen
<	TAG und NAME werden geöffnet wenn QUOTE unwahr ist.
>	TAG und ARG werden geschlossen wenn TAG geöffnet und QUOTE unwahr ist.
" oder '	Wenn TAG wahr und QUOTE unwahr ist, wird QUOTE wahr. Wenn Tag und QUOTE wahr sind, werden QUOTE und ARG unwahr.
SPACE	Wenn TAG wahr und QUOTE unwahr ist wird ARG wahr und NAME unwahr.
/	Wenn TAG und NAME wahr sind, wird ein interner Parameter-Schalter auf CLOSED gesetzt.

Eine Erklärung unserer Schalter:

- **TAG**

  Weiß ob ein Tag geöffnet ist.

- **NAME**

  Weiß ob zurzeit ein Tag-Name eingelesen wird.

- **QUOTE**

  Weiß ob zurzeit die Werte eines Arguments eingelesen werden.

- **ARG**

  Weiß ob zurzeit ein Argument eingelesen wird.

Damit es kein Durcheinander gibt werden wir die Schalter im Beispiel exakt wie in der Liste aufgeführt, aber mit einem voranführenden KEY_ bezeichnen.

Als Buffer für unsere Tags werden wir eine strukturierte Linkedlist nutzen.

Die Structure wird in unserem Beispiel folgende Elemente enthalten:

- Eine Stringvariable für den Name des Tags.
- Ein Array welches die Argumentnamen
- enthält.
- Ein Array welches die Werte der Argumente enthält.
- Eine Stringvariable für den Inhalt des Tagsblocks

Beispiel:

```
<echo wrapped="yes">
Ich bin der Inhalt des Block.
</tag>
```

Der Inhalt besteht ausschließlich aus ungetaggtem Inhalt.

Wird ein Tag über einen Backslash geschlossen, so wird dieser Schliesser als Tag angelegt und dessen erstes Argument heisst automatisch *close="this"*.

Dieses wird vom Parser automatisch angelegt.

So wird der Interpreter angewiesen den gleichnamigen, geöffneten Tag zu schließen.

Somit wird also ein Tagblock erzeugt.

Wir könnten nun auch noch einen Blockpfad anlegen der immer den aktuellen Pfad des jeweiligen Tags innerhalb eines Blocks enthält. Doch da wir ohnehin hier keinen Interpreter aufführen können und uns rein auf das Parsen konzentrieren wollen,

sollte das bestehende genügen.

Der Artikel zeigte das Parsen und Verarbeiten einer sehr einfachen Syntax. Kompliziertere Syntax-Muster werden nach einem anderen Prinzip verarbeitet. Hierzu muss der Code in Tokens zerlegt und anschließend im Parser mit syntaktischen Möglichkeiten verglichen werden. Doch hierzu mehr im nächsten Artikel von mir zu diesem Thema.

Der komplette Parser ist auf unter [1] zu finden.

Links:

[1] http://dl.pb-cm.net/2006-1/parser.pb

Ordnung im Code

von Marc-Sven Rudolf

Es gibt nichts Schlimmeres während der Arbeit an einem Projekt als der Verlust der Übersicht im Code.

Schon viele Projekte sind an fehlender Ordnung und Organisation gescheitert.

Man stelle sich ein wachsendes Lager vor, welches immer reicher an Inhalt wird.

Wenn der Lagermeister schlampig wird, findet er bald nichts mehr und kann nicht mehr arbeiten.

Genauso ist das mit dem Programmieren.

Mit jeder Zeile die geschrieben wird wächst der Code und bei fehlender Ordnung schwindet die Übersicht was unweigerlich dazu führt dass das Projekt eingestellt werden muss.

Wir werden in diesem Artikel sehr ausführlich über eine angenehme und leicht übersichtliche Ordnung in Codes funktionaler Programmierung sprechen.

Die Grundlage der programmiererischen Ordnung ist das Verschachteln der Codes.

Diese Verschachtelung wird benutzt um eindeutige Blöcke des Codes visuell heraus zu arbeiten.

Eine Verschachtelung wird bei heutigen Editoren über die Tabulatortaste bewirkt.

Ein Tab ist standardmässig 2 Leertasten lang.

Hier gibt es das Einrücken und Ausrücken.

Eingerückt wird mit [Tab] und ausgerückt mit [Shift]+[Tab].

Im ersten Beispiel sehen wir einen Code ohne Einrückung und im Zweiten einen Code mit Einrückung.

Beispiel 1:

```
Procedure Hallo_Welt ()
Titel.s="Hallo Welt"
Text.s="Programmierst du mit PureBasic ?"
Eingabe.s=Input ()
If Eingabe.s="ja"
PrintN ("Prima")
ProcedureReturn #True
Else
PrintN (":-(")
ProcedureReturn #False
EndIf
EndProcedure
```

Beispiel 2:

```
Procedure Hallo_Welt ()
  Titel.s="Hallo Welt"
  Text.s="Programmierst du mit PureBasic ?"
  Eingabe.s=Input ()
  If Eingabe.s="ja"
    PrintN ("Prima")
    ProcedureReturn #True
  Else
    PrintN (":-(")
    ProcedureReturn #False
  EndIf
EndProcedure
```

Wie im eingerückten Beispiel zu sehen ist wurden zusammenhängende Bereiche separat zu einem Block eingerückt.

So weiss man zum Beispiel schnell, dass

```
    PrintN ("Prima")
    ProcedureReturn #True
```

zu

```
If Eingabe.s="ja"
```

gehört.

Auch wichtig ist eine Benennung von Variablen, Prozeduren und Konstanten nach Thema bzw. Kategorie. Ohne ordentliche Nomenklatur ist ein komplizierter Code kaum übersichtlich.

So würde man zum Beispiel einer ID-Konstanten das einem ButtonGadget gehört auf dem 'OK' steht und im Fenster 'Datei-Fenster' erscheint einen eindeutigen Namen wie *#DateiFenster_Button_OK* zuweisen.

Und eine Variable welche zum Beispiel das Alter eines Benutzer enthält und programmweit verarbeitet werden kann, würde man zum Beispiel *vGlobal_Benutzer_Alter.l* nennen.

Das Ganze würde dann bei einer, nur lokal in bestimmten Procedure verarbeitbaren Variable so aussehen *vShared_Benutzer_Alter.l* .

Hierbei handelt es sich lediglich um Vorschläge die ein gewisses Prinzip bei der Vergabe von Namen vermitteln sollten.

Auch hier variiert die Handhabung unter den Programmierern stark.

Auch bei den Sub-Routinen ist eine eindeutige Benennung nur von Vorteil.

Wir nehmen zum Beispiel an, eine Procedure zu haben mit der eine Listbox erstellt, eine Procedure mit der diese Listbox eingefärbt und eine Procedure mit der die Listbox gefüllt werden kann.

Vorteilhafterweise sollte man deren Namen hauptfunktionsbezogen vergeben.

Das Stichwort heißt hier Meister und Sklave.

Die Hauptfunktion ist hier die Listbox.

Oft kommt es vor dass man eine Funktion hat die ein Objekt erzeugt und mehrere die mit diesem Objekt arbeiten.

In diesem Falle ist die Erzeuger-Funktion die Master-Funktion und ihre Versorger sind die Slave-Funktionen.

Je nach dem ist es anzuraten, sollten die Funktionen ihre Namen bekommen.

```
ListBox_Create
(ID.l,x.l,y.l,width.l,height.l)
ListBox_Color
(ID.l,BackColor.l,FrontColor.l)
ListBox_AddItem (ID.l,Position.l,Text.s)
```

Wie hier zu sehen ist wurden die Funktionen anführend zur objektiven Hauptfunktion und nachführend zur Methode benannt.

Solche Bezeichnungen lassen sich sehr gut merken und erleichtern die Arbeit daher sehr.

Ein Wirrwarr von nicht aussagenden Namen wird dadurch vermieden und man schafft sich automatisch einen arbeitssicheren Standard.

Das Gleiche wie für die Variablen gilt auch für Structures.

Empfehlenswert ist es eine Structure besonders abgrenzend zu kennzeichnen.

Ein anführender Unterstrich wäre hier meine Empfehlung.

```
Structure _Test
  Name.s
  Alter.s
EndStructure
```

Der Grund dafür liegt nahe.

Struktierte Variablen, Listen und Arrays sind dadurch viel leichter erkennbar.

Einige Programmierer nutzen auch ein nachfolgendes *_Struct* als Kennzeichnung einer Structure.

```
Structure Test_Struct
  Name.s
  Alter.s
EndStructure
```

Auch bei der Vergabe von Bezeichnungen für Listen und Arrays sollten besonders abgrenzende Kennzeichnungen verwendet werden, da gerade Listen im Gebrauch oft aussehen wie gewöhnliche Procedures oder PB-Funktionen.

Dies könnte man durch ein anführendes oder nachführendes _Lst erreichen

```
NewList  Test_Lst()
```

Auch ein deutlich sichtbarer Übersichtlichkeitsfaktor mit besonders gekennzeichneten Structures.

```
NewList  Test_Lst._Test()
```

oder

```
NewList  Test_Lst.Test_Struct()
```

Damit eine Zeitung übersichtlich ist ordnet sie die Redaktion nach Wichtigkeit der Inhalte, bildet kategorische Blöcke und Absätze.

Genauso wie eine Zeitung profitiert ein Code von bedachter Strukturierung.

Hierbei ist stets darauf zu achten wie die Abhängigkeiten in der Reihenfolge der Inhalte sind.

So sollten zum Beispiel globale Variablen und Enumerierungen stets am Kopf des Codes stehen.

Auf jeden Fall aber sollte der Code in einzelne Includes untergliedert sein.

Ganz die objektorientierte Programmierung zum Beispiel genommen, ist es ratsam die einzelnen Master- und deren Slave-Funktionen einer Funktions-Kategorie in separate Includes mit jeweils eigenem Initierungskopf zu verstauen.

Wie schon einmal erwähnt handelt es sich bei diesem Artikel nur rein beispielhafte Vorschläge.

Es gibt in PureBasic unmengen von Programmierstilen bzw Ordnungssystemen.

Letztendlich kommt es nur darauf an, dass der Programmierer Effizient arbeiten kann ohne sich mit seinem Programmierstil selbst im Wege zu stehen.

Tunnelblick

von Thorsten Will

Die im weiteren Verlauf des Artikels genannten PureBasic-Codes sind unter [1] zu finden.

Hallo und herzlich willkommen in der Wunderwelt der so genannten tabellenbasierten Demo-Effekte.

Was sind tabellenbasierte Demo-Effekte und warum werden hierfür Tabellen benutzt? Viele ältere Demo-Effekte wie Tunnel, Sphere, BumpMap usw. sind oft lediglich 3D-Imitationen und basieren auf vorberechneten 2D-Daten, welche für die spätere Verwendung in Tabellen (Arrays) gespeichert werden. Da die benötigen Daten vorberechnet werden, brauchen wir diese später nicht im Hauptprogramm und in Echtzeit immer wieder neu zu berechnen. Hierdurch ersparen wir uns einige Berechnungen und können die Performance des einzelnen Effekts oft um einiges steigern.

Heute werden wir uns Schritt für Schritt die Grundlagen zur Programmierung eines Tunnel-Effektes anschauen und Schritt für Schritt einen solchen selber Programmieren.

Was brauchen wir überhaupt um unseren ersten Tunnel-Effekt zu realisieren und wie funktioniert dieser Effekt? Der Tunnel-Effekt an sich besteht aus drei wichtige Elementen. Wir hätten da zum einen die Textur, welche wir später auf unseren Tunnel mappen (kleistern). Als Textur werden wir uns eine eigene 256x256 große, so genannte „XOR-Textur“ berechnen. Selbstverständlich könnten wir auch einfach per Zufall Punkte, Linien oder Kreise auf unserer Textur zeichnen - oder sogar eine 256x256 große "tileable" Textur als BMP benutzen.

Nun werden wir unsere erste XOR-Textur berechnen, in dem wir ein 256x256 großes Image erstellen und dieses nach Fertigstellung auf unserer Festplatte als „TunnelFx_Textur.bmp“ speichern. Als Ergebnis erhalten wir unsere fertige Textur im Bitmap-Format.

```
Listing: Part1_Textur.pb
```

Des weiteren benötigen wir zwei große Texturen (Distance und Angle), die unserer darzustellenden Bildschirmgröße 640x480 entsprechen. Wenn wir uns den Blick in einen runden Tunnel vorstellen, so könnte man dieses mit einer Röhre vergleichen, die aus vielen Ringen besteht und die Ringe mit zunehmender Distance immer kleiner werden. Wir werden nun eine entsprechende „Distance-Textur“ erstellen und diese ebenfalls als Bitmap speichern (Abbildung 1).

```
Listing: Part2_Distance.pb
```

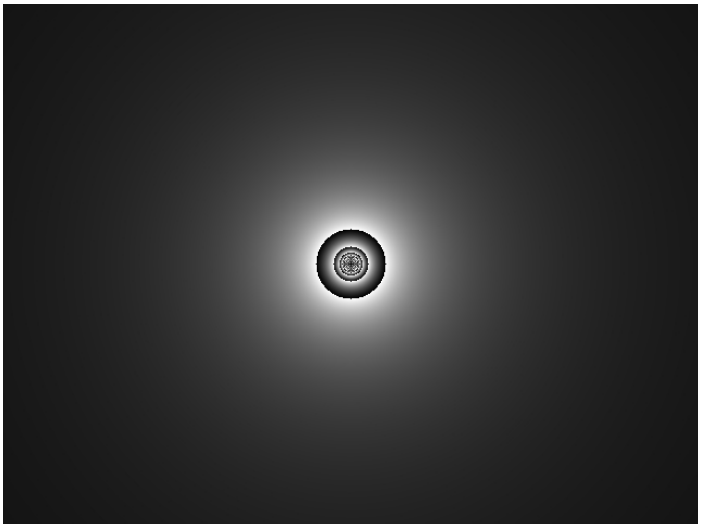


Abbildung 1: Die Distance-Textur

Nach dem wir nun die Distance-Textur erstellt haben, benötigen wir noch die „Angle-Textur“ (Winkel-Textur). Diese Textur wird benötigt um unsere XOR-Textur im Tunnel richtig anzupassen und unseren Tunnel später u.a. rotieren zu lassen. Die Distance-Textur wird u.a. benötigt, um in den Tunnel zu fliegen.

Listing: Part3_Angle.pb

Abbildung 2: Die Winkel-Textur

Bisher haben wir die drei wichtigsten Elemente für einen Tunnel-Effekt und ihre jeweiligen Routinen kennen gelernt, dessen Ergebnisse wir bisher immer auf eine Bitmap darstellten und speicherten. Nun werden wir die berechneten Daten direkt die Tabellen (Arrays) *aTextur*, *aDistance* und *aAngle* speichern und die einzelnen Routinen der ersten drei Listings zusammenlegen und somit ein wenig optimieren. In Teil 1 erstellen wir unsere 256x256 große XOR-Textur, bei der wir die x- und y-Schleife von 0 bis 255 durchlaufen lassen. Da wir keine weitere Routine verwenden, in der wir ebenfalls von 0 bis 255 zählen, lassen wir diese Routine unverändert.

Allerdings verwenden wir für die Berechnung der Distance-Table und Angle-Table in beiden Fällen Schleifen, deren Werte in beiden Routinen identisch sind. Daher können wir diese beiden Routinen ohne Probleme zusammenfassen und uns Zeit für die Vorberechnung sparen und kleineren Programmcode erzeugen.

Listing: Part4_MergingStuff.pb

Wir werden das bisher gelernte nun anwenden, einen Screen öffnen und das erste Resultat unseres Tunnels auf dem Bildschirm darstellen.

Um den Programmcode einfacher zu lesen, werden wir einfach den *Plot()*-Befehl verwenden und nicht auf DirectScreenAccess aufbauen.

Mit unserer *aDistance*- und *aAngle*-Tabelle können wir nun ohne Probleme einen Tunnel darstellen. Damit aber auch unsere 256x256 große XOR-Textur (*aTextur*) im Tunnel richtig gemapped (abgebildet) wird, werden wir diese beiden Tabellen im Innerloop einfach auslesen und wir erhalten die entsprechende X- und Y-Koordinaten der XOR-Textur. Das Ergebnis speichern wir im Array *aBuffer*, welchen wir zuvor noch zu unserem Code hinzufügen müssen. Wir könnten nun in *aBuffer* weitere Manipulationen vornehmen oder das Ergebnis wie in der nächsten Zeile zu sehen ist, mit *Plot()* auf den Bildschirm darstellen lassen.

```
aBuffer(x,y) = aTextur (aDistance(x,y),    =>
aAngle(x,y))
Plot(x, y, aBuffer(x,y) )
```

Der komplette Sourcecode würde nun wie folgt aussehen und wir würde beim ausführen des Sources ein Standbild des Tunnel-Effektes erhalten.

Listing: Part5_FirstTunnel.pb

Um unseren Tunnel nun zu animieren, werden wir diesen drehen lassen und dabei in den Tunnel hinein fliegen. Hierzu benötigen wir die Variablen *dSpeedX.d* und

SpeedY.d, um die Geschwindigkeit für die X- und Y-Bewegung des Tunnels zu definieren. Des weiteren benötigen wir im Innerloop die Variablen *lShiftX.l* und *lShiftY.l*, um die entsprechenden Bewegungen in den Tabellen zu berechnen und später auszulesen. In unserem Beispiel werden wir für die Animation die Variable *dAnimation.d* nehmen, welche wir pro Loop um 0.005 erhöhen.

Anmerkung: *Bitte für eigene Projekte mit Deltazeiten arbeiten, damit der Effekt auf jeden Rechner gleich schnell läuft!).*

```
dAnimation.d = dAnimation.d + 0.005
If dAnimation.d >= 1.0
  dAnimation = 0.0
EndIf
lShiftX.l = Int(lTexturSize * dSpeedX.d * => 
dAnimation.d)
lShiftY.l = Int(lTexturSize * dSpeedY.d * => 
dAnimation.d)
```

Nun haben wir alle wichtigen Berechnungen vorgenommen, damit der Tunnel animiert werden kann. Um nicht wieder wie zuvor nur ein Standbild zu erhalten, müssen wir bei der Darstellung des Tunnels folgende Zeilen hinzufügen beziehungsweise abändern:

```
lCoordinateX.l = (aDistance(x,y)+lShiftX) => 
% lTexturSize

lCoordinateY.l = (aAngle(x,y) + lShiftY)  => 
% lTexturSize

aBuffer(x,y) = aTextur (lCoordinateX.l ,  => 
lCoordinateY.l)

Plot(x, y, RGB(0, 0, aBuffer(x,y) ))
```

Der komplette Sourcecode für einen komplett animierten Tunnel, würde nun wie folgt aussehen:

```
Listing:  Part6_AnimateTunnel.pb
```

An dieser Stelle sind wir nun bei unseren animierten Tunnel angelangt. Leider wirkt das ganze auf eine Art doch noch ein etwas langweilig. Um unseren aktuellen animierten Tunnel noch ein wenig aufzumotzen und interessanter wirken zu lassen, werden wir ein paar kleine Änderungen vornehmen. Dabei werden wir den eigentlichen Tunnel tanzen (bzw. bewegen) lassen. Es entsteht der Eindruck, als würde man die Kamera im Tunnel bewegen.

Da unserer Tunnel nur auf Tables und nicht auf „echtem“ 3D basiert, werden sich nun sicher einige Fragen, wie man den Effekt mit der Kamera umsetzen will. Wir werden hierfür einen einfachen und zugleich genialen Trick verwenden. Unser Bildschirm und die *aDistance*- und *aAngle*-Tabellen sind alle für 640x480 ausgelegt. Wir werden nun die Breite und Höhe der Tables *aDistance* und *aAngle* verdoppeln, um später immer nur den gewünschten Ausschnitt auf den Bildschirm darzustellen.

```
Dim  aDistance(lScreenWidth*2,            => 
lScreenHeight*2)
Dim aAngle(lScreenWidth*2,lScreenHeight*2)
```

Bei den Berechnungen der *aDistance* und *aAngle*-Tables, haben wir bisher immer *lScreenWidth / 2* (geteilt durch 2) gerechnet, da *lScreenWidth* die zweifache Größe von *aDistance* und *aAngle* hatte. Weil wir aber nun die Größe von *aDistance* und *aAngle* verdoppelt haben, brauchen wir bei der Berechnung dieser Tabellen nicht mehr geteilt durch 2 zu rechnen.

Um unseren Tunnel nun schön tanzen zu lassen, verwenden wir einfach Sin() und fügen die Variablen *lLookX* und *lLookY* hinzu. Dafür werden folgende Zeilen geändert:

```
lLookX = lScreenWidth/2 + Int(lScreenWidth=>
/2 * Sin(dAnimation * 4.0 ))

lLookY = lScreenHeight/2 + Int(           =>
lScreenHeight /2 * Sin(dAnimation * 6.0 ))
```

Bei der Darstellung des Tunnels müssen wir nun selbstverständlich die Werte *lLookX* und *lLookY* berücksichtigen. Dafür ändern wir den Code wie folgt ab:

```
lCoordinateX.l = (aDistance(x+lLookX,     =>
y+lLookY) + lShiftX) % lTexturSize

lCoordinateY.l = (aAngle   (x+lLookX,     =>
y+lLookY) + lShiftY) % lTexturSize
```

Wenn jetzt keine Fehler unterlaufen sind, sollten wir nun einen schönen texturierten und animierten Tunnel haben, dessen Kameraposition sich ständig zu verändern scheint. Hier nun nochmal der komplette Sourcecode:

```
Listing:  Part7_MovingTunnel.pb
```

Links:

[1] http://dl.pb-cm.net/2006-1/tunnel.tar.bz2

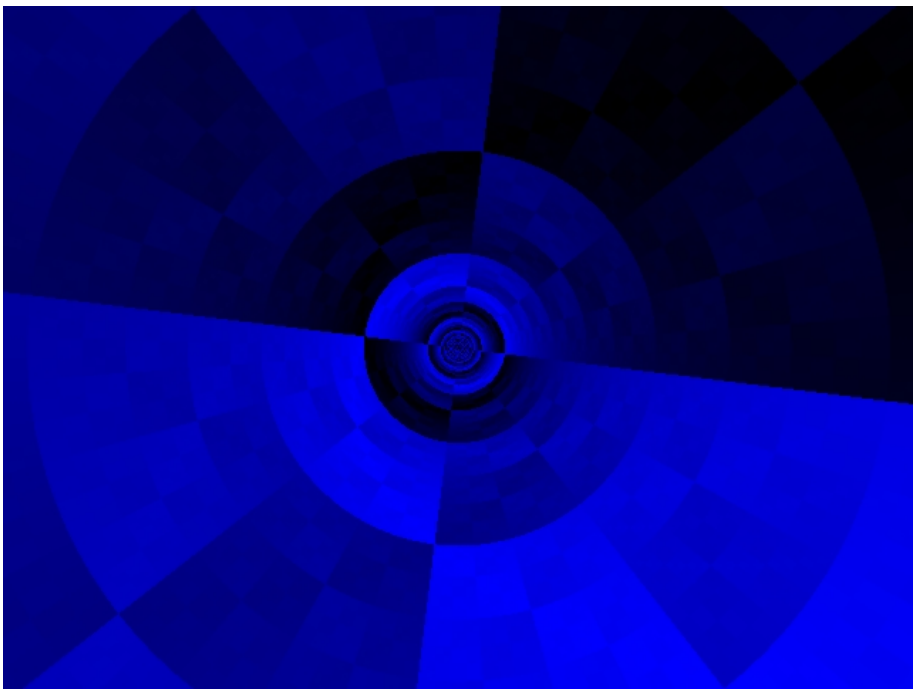


Abbildung 3: Der fertige Tunnel

Entwickler des Monats

von Thorsten Hoeppner

An dieser Stelle werden wir in jeder Ausgabe den Entwickler des Monats küren, der aufgefordert wird, einen Artikel über sich und sein prämiertes Programm zu schreiben.

Außerdem darf er mit der PB.CM-Auszeichnung für sein Programm werben.

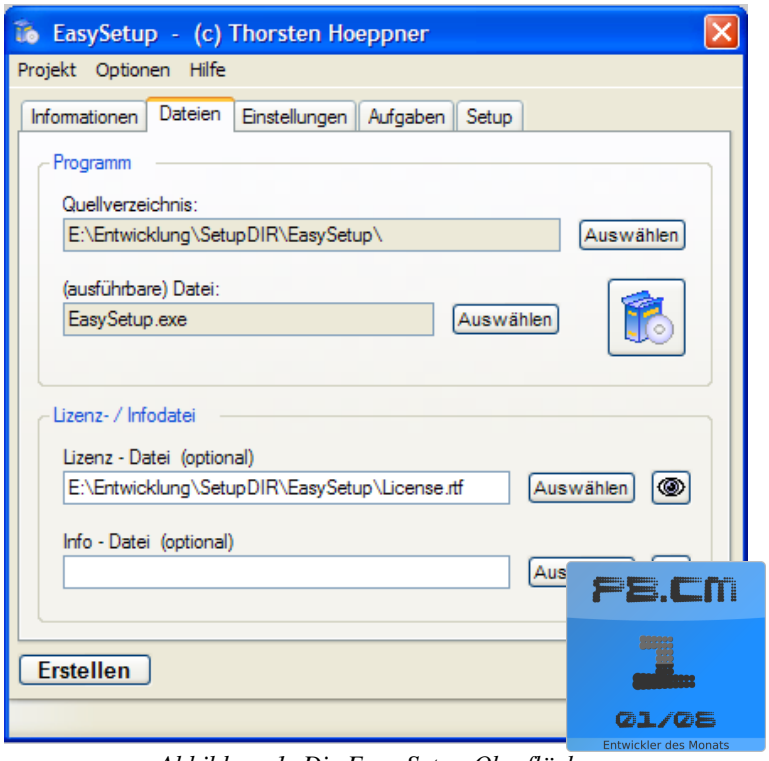


Abbildung 1: Die Easy-Setup-Oberfläche

EasySetup und sein Autor

Ich wurde von PB.CM gebeten etwas über *EasySetup* und auch über mich selber zu schreiben. Nachdem ich „leichtsinnigerweise“ zugesagt habe, sitze ich vor meinem Computer und versuche einen Artikel aus dem Ärmel zu schütteln. Also beginnen wir mal mit Punkt Nr. 1 „Wer bin ich?“ (Keine Angst es folgt jetzt kein philosophischer Exkurs Über „Sein oder Nichtsein“)

Wer verbirgt sich hinter „Thorsten1867“?

Nachdem die Profile in den PB-Foren diesbezüglich sehr mit Informationen geizen, hier die Auflösung. Meine richtiger Name ist Thorsten Hoeppner und ich komme aus Kaufbeuren, das im Ostallgäu (also so ziemlich am Ende der Welt) zu finden ist. Nachdem man neben dem Programmieren (entgegen anders lautenden Gerüchten) auch noch etwas anderes tun muss, habe ich natürlich auch eine Beschäftigung, die den größten Teil meiner Zeit beansprucht. Ich gehe „noch“ zur Schule. Wie der eine oder andere Scharfsinnige unter euch schon anhand meines Hauptprojektes *KvGS* bemerkt hat, ist diese Aussage zwar richtig, aber etwas irreführend. Ich bin kein Schüler mehr, sondern Schule ist mein Beruf. Nein, um eurer nächsten Frage gleich vorzugreifen, ich unterrichte keine Informatik oder Ähnliches. Versuche meinen Schülern die Grundlagen der Programmierung beizubringen, würden nur mit großen Augen und einem ungläubigen Staunen enden, denn Drittklässern fehlt noch irgendwie der Draht dazu. Die Verbindung von meinem Beruf und PureBasic liegt ganz woanders, nämlich in meinem Projekt KvGS - „Klassenverwaltung für die Grundschule“.

So, nachdem das „Wer?“ nun hoffentlich zu Genüge geklärt ist und die Schüler unter den PureBasic-Nutzern sich wieder von dem Schock bezüglich meines Berufes

wieder erholt haben, gleich zum nächsten Punkt.

Programmiererfahrungen und PureBasic

Meine ersten Programmierversuche fanden auf einem der sogenannten „Homecomputer“ (Sinclair Spectrum) statt. Nachdem die Homecomputer damals anstatt eines Betriebssystemes nur über einen Basic-Interpreter verfügten, blieb einem eigentlich nichts anderes übrig, als selber zu programmieren. Später kam, mit dem Aufkommen der ersten Personal Computer AT 8086 (512KB RAM!!!), in der Schule noch etwas TurboPascal dazu. Mit dem Aufkommen von MS Windows 3.1 und IBM OS/2 ist das Programmieren immer weiter in den Hintergrund geraten, da es einfach nicht mehr nötig war. Später habe ich dann als Lehrer mit Mediator einige Multimediaprogramme entwickelt, was man aber nicht wirklich „Programmieren“ nennen konnte. Aber dann etwa im Jahr 2003 habe ich mit der Entwicklung einer Homepage für unsere Schule [1] begonnen. Dazu habe ich mich gezwungenermaßen erst in die Sprache HTML und dann später in PHP und MySQL eingearbeitet. Da wurde mir wohl klar, was ich all die Jahre vermisst hatte, das Programmieren. Wie ich dann genau auf PureBasic gekommen bin, weiß ich nicht mehr genau. Vermutlich war es ein Artikel in der c't oder im Internet. Naja, „Basic“ ließ wohl Jugenderinnerungen wieder hochkommen und die Tatsache, dass es sich bei PureBasic nicht um einen Interpreter, sondern um einen Compiler handelte und letztendlich auch der günstige Preis haben dann den Ausschlag gegeben. So bin ich dann also Mitte 2004 dem erlauchten Kreis der PureBasic-Nutzer beigetreten. Nach einigen ersten Versuchen und Spielereien begann ich dann Anfang 2005 mit meinem ersten richtigen Projekt, KvGS.

Abbildung 2: Ein mit EasySetup erstelltes Installationsprogramm

Nachdem mich die teuren kommerziellen Klassenverwaltungs- und Zeugnisprogramme nicht wirklich zufrieden stellen konnte, dachte ich mir: „Das ist doch die Gelegenheit mal auszuprobieren, was PureBasic leisten kann.“ Gesagt, getan! Inzwischen ist aus den paar Codezeilen zum Privatgebrauch, ein Programm mit Über 20.000 Zeilen geworden, dass sich nicht mehr vor den handelsüblichen Programmen zu verstecken braucht und erfolgreich an unserer Schule eingesetzt wird. Damit begann dann auch die Geschichte von *EasySetup*, das eigentlich nichts anderes ist, als eine Nebenprodukt von KvGS. Aber dazu mehr im nächsten Abschnitt.

Wie EasySetup entstanden ist

Nachdem ich mein eigentliches Projekt KvGS auch anderen Grundschullehrern zur Verfügung stellen wollte, musste ich mich auf die Suche nach einem Setup-Programm machen. Das Setup-Programm, das ich dann anfänglich verwendete, hatte drei große Nachteile, es war nur in Englisch, der Installer war oft größer als das eigentliche Programm und unglaublich komplex. Eine Lösung der Probleme kam dann mit dem *SpeedSetup*-Projekt von Trutia Alexandru in Sicht. Das Projekt entwickelte sich sehr vielversprechend, bis dann kurz vor

der Vollendung der zweiten Version, der Source-Code verloren ging und der Autor frustriert das Projekt aufgab. So stand ich dann wieder ohne benutzerfreundlichen Installer für mein Programm dar. So beschloss ich gezwungenermaßen mich in einem eigenen Setup-Programm zu versuchen. Nachdem ich einige Versuche zur Programmierung mehrsprachiger Programme unternommen und eine erste noch etwas sehr kompakte GUI entworfen hatte, konnte ich loslegen und bald einen ersten Entwurf zum Download ins Forum stellen. Dank der vielen Rückmeldungen und Verbesserungsvorschläge der Forum-Benutzer und einer neuen, Übersichtlicheren Benutzeroberfläche entwickelte sich EasySetup immer weiter. Das Ergebnis dürfte der eine oder andere inzwischen kennen und vielleicht sogar für seine Programme verwenden.

Was ist EasySetup und was kann es?

Wie schon oben erwähnt, ist EasySetup ein Tool zur Erstellung von Setup-Programmen für die eigenen Anwendungen. Dabei liegt der Hauptaugenmerk auf der Bedienerfreundlichkeit, Übersichtlichkeit und natürlich auf der Größe des Installer & Uninstallers. Es gibt also keine komplizierte Skript-Sprache oder unübersichtliche Benutzeroberfläche. Der Normalbenutzer kann also schnell und einfach sein Setup-Programm erstellen ohne durch komplizierte Einstellmöglichkeiten irritiert oder verwirrt zu werden. Fortgeschrittene Anwender finden bei den zusätzlichen benutzerdefinierten Aufgaben einige interessante Sonderfunktionen, die sie mit einem kleinen Häkchen aktivieren können. Zu einer Funktionsübersicht siehe den Kasten „Features“.

Was kostet EasySetup?

EasySetup ist „Donationware“, das heißt, es ist grundsätzlich einmal kostenlos. Wenn einem das Programm gefällt und man es regelmäßig einsetzt, steht es jedem frei mir auf der Homepage [2] einen kleinen Obolus (per PayPal) zukommen zu lassen.

Die Zukunft von EasySetup

Die Zukunft von EasySetup wird wie bisher von den Bedürfnissen und Ideen der Benutzer bestimmt werden. In erster Linie hoffe ich im Moment EasySetup, mit Hilfe freiwilliger Übersetzer, einige neue Sprachdateien hinzufügen zu können, damit sich der Aufwand der Mehrsprachunterstützung auch gelohnt hat und EasySetup möglichst vielen Nutzern in verschiedenen Ländern zur Verfügung steht.

Links:

[1] http://www.ass-neugablonz.de

[2] http://www.easysetup.de.vu

Features:

- Übersichtliche und bedienerfreundliche GUI
- kleiner Installer & Uninstaller (<85K-byte)
- einfache Einbindung eigener Sprachen mit Übersetzunstool
- Anpassung des Installer an Programm durch eigenes Bitmap (Splashscreen)
- Vorschau- und Editierfunktion für Lizenz-/Infodatei zum genauen Anpassen ans Installerfenster
- einfaches Festlegen von Startmenü-Einträgen
- Projektaufruf über Kommandozeilenparameter möglich
- Pfad-Variablen für Programmverzeichnis verwendbar (z.B. %ProgramFiles%, %AppData%, %CommonFiles%)
- Benutzerdefinierte Verknüpfungen (Desktop, Startmenü, Autostart, URL)
- Installation einzelner Dateien in Systemordner möglich
- Erstellen eigener Registry – Einträge
- Möglichkeit ein externes Programm beim Start des Installers aufzurufen und nach Abschluss automatisch zu beenden

PB.Net – die Zukunft von PureBasic?

von Philipp Schmieder

Bereits vor einigen Monaten zeigte die Community großes Interesse an der Meldung [1] über einen in Entwicklung befindlichen PureBasic-Compiler für .Net. Wir hatten das Glück, dem Entwickler von PB.Net, Dietmar „Deeem2031“ Schölzel , einige Fragen zu seinem Projekt zu stellen.

Wie bist du auf die Idee gekommen, PB.Net zu entwickeln?

Angefangen hat alles mit einem Gespräch mit Danilo (der ja den meisten PBlern bekannt ist). In dem Gespräch hab ich dann unter anderem, ihn gefragt, was er jetzt mit seinem Programmierwissen macht. Darauf hat er mir geantwortet, dass er jetzt mit dem .Net-Framework arbeitet und es auch nicht bereut. Er war sogar so nett, mir ein paar Code-Beispiele zu zeigen, wie einfach Programmierung mit C# funktioniert. Zuerst war ich skeptisch, schließlich hatte ich noch nie mit .Net zu tun gehabt und konnte mir auch nicht wirklich vorstellen das Firmen (von Microsoft abgesehen) auch mit dem .Net-Framework arbeiten, fand es aber trotzdem interessant. Ich habe mir dann ein E-Book für IL-Assembler besorgt, was der Assembler für .Net ist, und war schnell begeistert, wie schnell und einfach man selbst auf der Assembler-Ebene Programme für .Net erstellen kann. Ich wusste dann allerdings nichts mit meinem neu erworbenen Wissen anzufangen, aber irgend ein größeres Programm wollte ich schon mit oder für .Net erstellen. Also habe ich mich kurzer Hand für einen .Net Compiler für PB entschieden.

Welche Vorteile hat PB.Net gegenüber „PB.Fred“?

Ersteinmal ist es objektorientiert, was ja einige bei Frédérics PB vermissen. Dass es auf .Net basiert, sehen vielleicht einige Microsoft-Hasser nicht als Vorteil an, aber meiner Meinung nach ist der Compiler dadurch wesentlich zeitgemäßer. Alle Vorzüge, die .Net hat, wird der Compiler nämlich damit auch nutzen können. Durch die Laufzeit-Optimierung beispielsweise kann Zeit eingespart werden, die mit "normalen" Compilern compilierte Programme nicht ausnutzen können.

Wird man wirklich plattformübergreifend entwickeln?

Da ich bis jetzt immer mal überprüft habe, ob der Compiler auch mit der aktuellen Mono-Version klar kommt und das bisher immer kaum ein Problem war, wird plattformübergreifende Programmierung mit Mono wahrscheinlich auch in der fertigen Version verfügbar sein.

Welche .Net-Version wird eingesetzt werden?

Bis jetzt noch v1.1, da aber selbst v2.0 jetzt schon etwas älter ist werde ich bestimmt spätestens mit dem Release von v3.0 auch die neuen Features nutzen.

Wird man sowohl objektorientiert als auch prozedural programmieren können?

Ja, die Kompatibilität zu „PB.Fred“ und die direkte Unterstützung von .Net im PB Code macht dies zwingend notwendig.

Wird ein normaler PB-4.0-Code ohne Änderungen mit PB.Net funktionieren?

Ich will es hoffen. Das war eines der wenigen Ziele, die ich mir von Anfang an gesetzt habe. Während den Arbeiten an dem Projekt bin ich aber schon ein paar mal an Stellen gestoßen, die mit .Net einfach nicht möglich bzw. zu kompliziert zu umgehen sind. So kann man z.B. keine Pointer von Strings bekommen und diese dann direkt im Speicher ändern. Im Großen und Ganzen werden aber die Codes sehr ähnlich aussehen; schließlich heißt es ja auch PureBasic.

Hast du dich preislich schon festgelegt?

Nein, zwar gehe ich davon aus das es ein wenig billiger als Freds PB sein wird, aber selbst da bin ich mir noch nicht sicher, da es nicht nur von mir abhängt wie teuer der Compiler letztendlich wird.

Wann dürfen wir das erste Release erwarten? When it's... Ihr wisst schon. Ich hab wirklich noch keine Ahnung.

Links:

[1] http://purebasic.fr/german/viewtopic.php?t=7978

Auf zu neuen Horizonten

von Philipp Schmieder

Mit großer Freude stürzte sich die Community bereits auf die Beta-Versionen, der schließlich am 10 Mai 2006 als „Final“ erschienenen Version 4.0 von PureBasic.

Leider wurde ein nicht kleiner Teil der Entwicklergemeinde alsbald auf eine schwere Geduldsprobe gestellt; die Linux-Portierung sei noch im Anfangsstadium, hieß es.

Die Monate vergingen, bis unsere beiden Redakteure Peter und Philipp als bekennende Nutzer des freien Betriebssystems eine Einladung zum Alpha-Test der Linux-Version unserer Lieblingsprogrammiersprache zu werfen.

Gleich nach der ersten Installation zeigte sich eine der vielen Neuerungen:
Die auf GTK2 basierende IDE (Abb. 1) funktioniert nun nicht mehr nur auf einigen wenigen Spezialdistributionen sondern auf jedem von uns getesteten Linux-Derivat (z.B. SUSE Linux, Mandriva oder Zenwalk) ohne Probleme, was Workarround-Lösungen mit Kate oder gar vim nun endlich der Vergangenheit angehören lässt.
Sämtliche von der Windows-Version gewohnten Komfortfunktionen wie Autovervollständigung, Einrückungen oder Syntaxhervorhebungen sind für die Alpha-Tester bereits Realität.
Ebenfalls erwähnenswert ist das eigens entwickelte dynamische Framework, welches sicherstellt, dass die IDE mit jedem Design jedes beliebigen Fenstermanagers zurechtkommt.

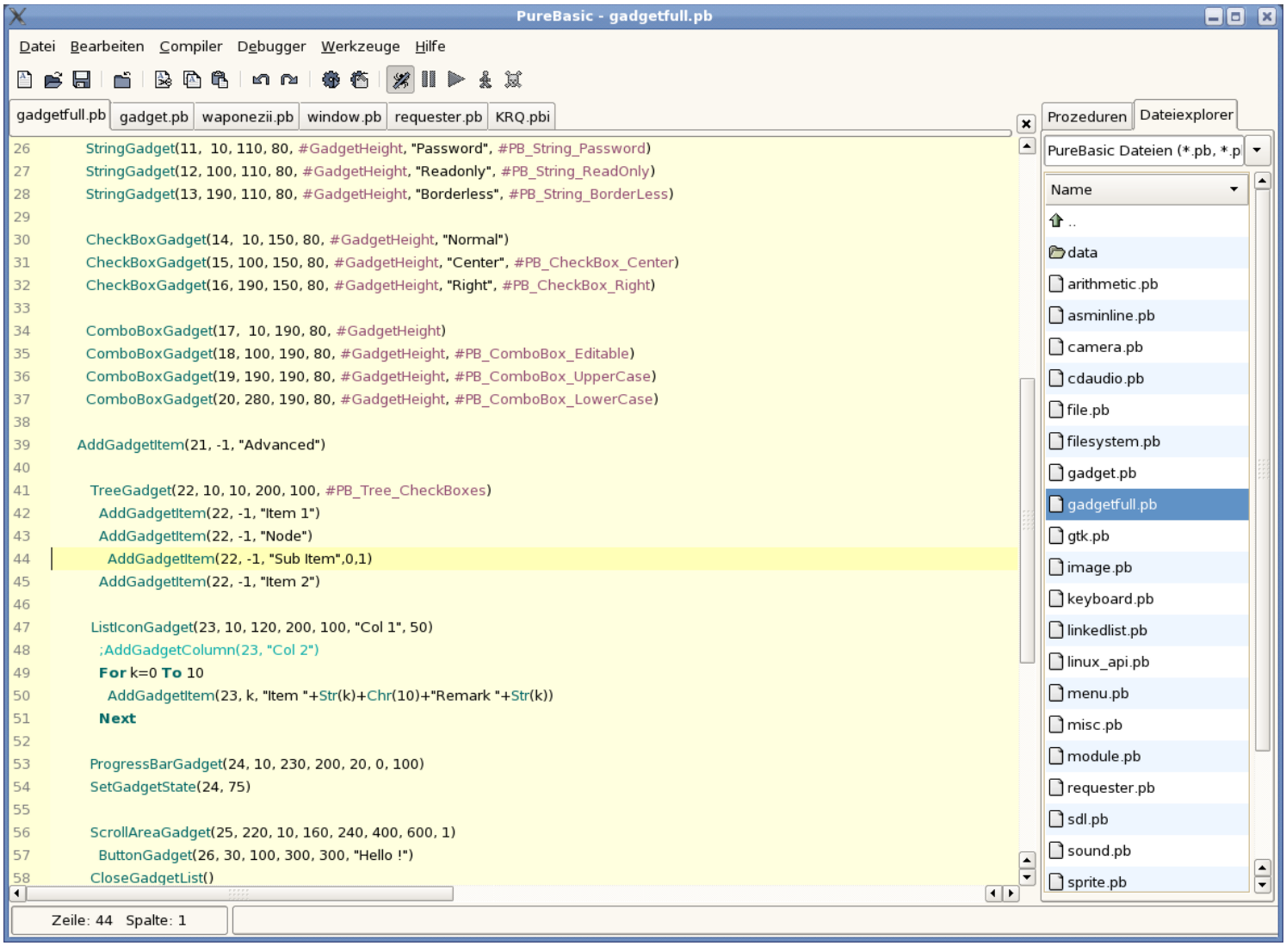


Abbildung 1: Die neue PureBasic-IDE

Auch der Debugger (Abb. 2) lässt mittlerweile keine Wünsche mehr offen. Hier wurde ebenfalls die komplette Bandbreite an Features übernommen.

Selbstverständlich beschränken sich die Änderungen nicht nur auf

die Oberfläche. Insbesondere unter der Haube hat sich vieles getan. Waren viele Bibliotheken in früheren Versionen von Linux-PB schlichtweg nicht vorhanden, so befindet sich diese Alpha-Version bereits weitgehend auf dem Stand ihres Windows-Pendants. Als Beispiel für neu integrierte Librarys wäre die lange ersehnte Systray-Icon-Lib, die mit allen gängigen Desktopumgebungen gemäß des FreeDesktop.org-Standards [1] der zusammenarbeitet, zu nennen.

Sogar mit UTF-8 bzw. UTF-16 kodierte Strings stellen kein Problem mehr dar (Abb. 3). Sie funktionieren ebenso problemlos wie der threadsafe-Modus des Compilers, der endlich die bequeme Ausführung mehrerer paralleler Prozesse ohne ständige IMAs (*Invalid Memory Access*) ermöglicht.

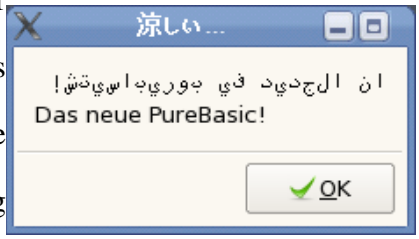


Abbildung 3: Unicode mit PureBasic

Bleibt noch zu hoffen, dass sich PureBasic dank dieser hervorragenden Linux-Version, die Portierungen eigener Programme auf das alternative Betriebssystem zum Kinderspiel macht, im breiten Entwicklermarkt Fuß fassen kann und so zur einer etablierten „Standard“-Sprache wird. Das Potenzial ist vorhanden.

Links:

[1] http://freedesktop.org

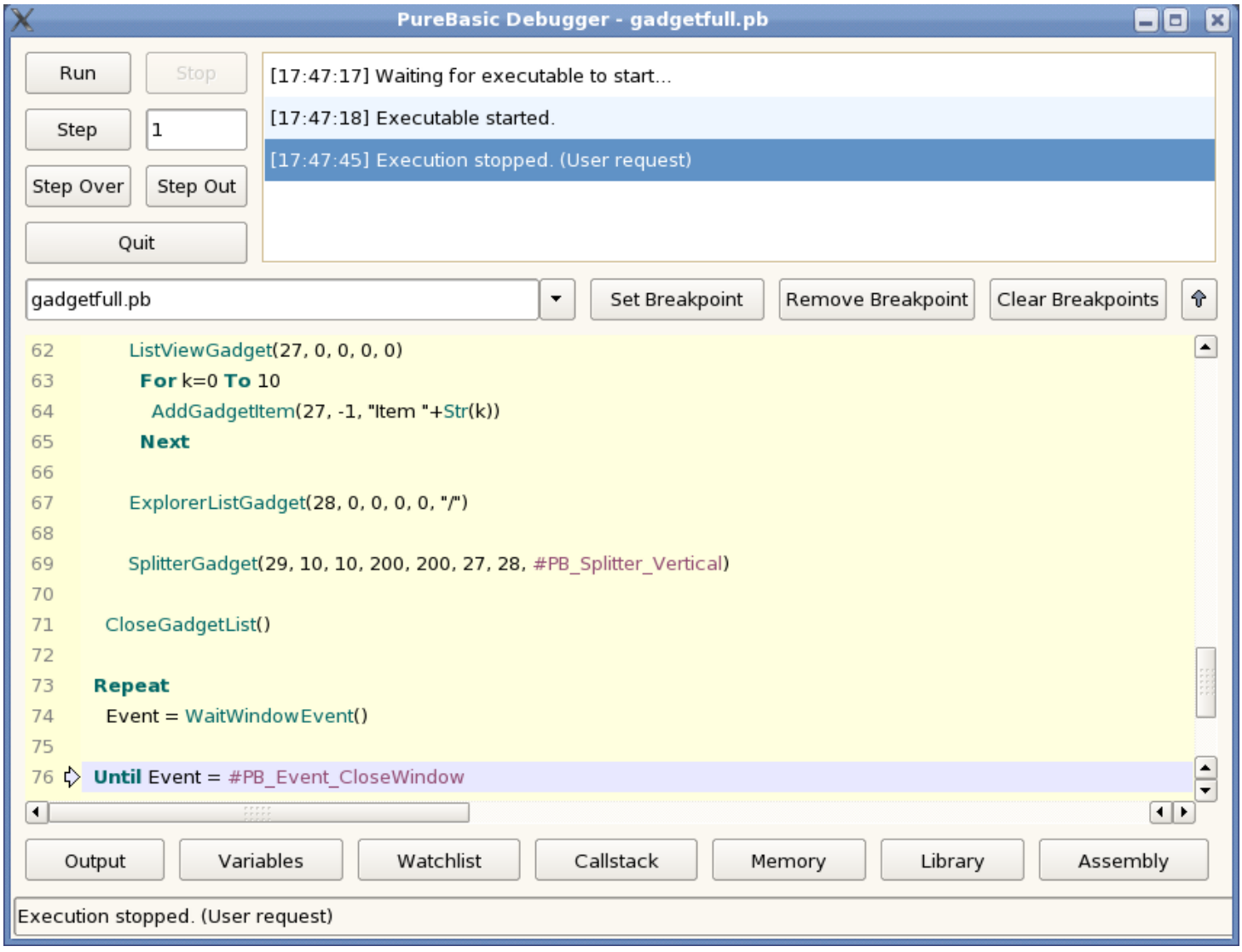


Abbildung 2: Der neue Debugger

Impressum

Redaktion:

Peter Kastberger

Christopher Higgs

Philipp Schmieder

Kontakt:

redaktion@pb-cm.net

Freie Autoren in dieser Ausgabe:

Frank Wiebeler (S. 7)

Marc-Sven Rudolf (S. 12, S. 15)

Thorsten Will (S. 18)

Bildnachweis:

Cover-Bild (S. 1): (CC) 2006 http://efekt.net/

Alle übrigen Bilder sind das Werk der Autoren bzw. der Redaktion

Verwendete Software:

OpenOffice.org 2.0

Xara Xtreme