Ausgabe 4/2011 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

SNMP::Info
SNMP mal ganz einfach

Perl in Multithreaded Programmen einbetten
Perl-Code direkt aus C zu nutzen

$foo als ebook
... vom POD-Format zum Ebook

Nr 20

BOOKING.COM

online hotel reservations

NOW HIRING!

SysAdmins

MySQL DBAs

Perl Devs

Software Devs

Web Designers

Front End Devs ...

Interested? Booking.com/jobs

Booking.com B.V., part of Priceline.com (Nasdaq:PCLN), owns and operates Booking.com (TM), one of the world's leading online hotel reservations agencies by room nights sold, attracting over 30 million unique visitors each month via the Internet from both leisure and business markets worldwide.

We use Perl, puppet, Apache, MySQL, Memcache, Git, Linux …and many more!

Established in 1996, Booking.com B.V. guarantees the best prices for any type of property, ranging from small independent hotels to a five star luxury through Booking.com. The Booking.com website is available in 41 languages and offers 120,000+ hotels in 99 countries.

- Great location in the center of Amsterdam
- Competitive Salary + Relocation Package
- International, result driven, fun & dynamic work environment

VORWORT

YAPC::Europe 2012, Perl Foundation und die Umfrage

Viele von Ihnen werden es schon mitbekommen haben: Die europäische Perl-Konferenz (YAPC::Europe) wird 2012 in Frankfurt stattfinden. Damit ist die Veranstaltung genau 10 Jahre nach dem Gastauftritt in München wieder in Deutschland. Die Frankfurt Perlmongers wollen das Event zur größten Konferenz der YAPC::Europe-Geschichte machen. Dafür sind auch viele Helfer und Sponsoren nötig. Interessierte können sich bei questions@yapc2012.de melden.

Mit dieser Veranstaltung vor Augen, haben die Frankfurt Perlmongers einen Verein gegründet, der das Event durchführen wird und danach auch weiter für die Förderung von Perl eintritt. Überhaupt ist die "Nachwuchsgewinnung" ein Thema bei den YAPCs. Zusammen mit der YAPC::NA (Nord Amerika) werden Überlegungen angestellt, wie neue Leute für Perl begeistert werden können - mit Hilfe der Konferenz.

Die Organisatoren sind schon auf Facebook, Twitter und auf blogs.perl.org vertreten und suchen den Kontakt zu den potentiellen Konferenzteilnehmern. Wer also Ideen oder Fragen hat, dann einfach dort oder unter der oben genannten Mailadresse melden.

Die Perl Foundation hat einen Spendentopf für den Perl 5 Kern eingerichtet, mit dem Dave Mitchell, Nicholas Clark und weitere Personen für Arbeiten am Perl 5 Kern bezahlt werden sollen. Da wir in diesem Jahr mehr Werbeeinnahmen haben und auch die Zahl der Abonnenten - langsam aber stetig - steigt, waren wir in der Lage, 1.000 US$ zu spenden. Vielen Dank an die Leser für ihre Treue, denn ohne die Leser wäre das nicht möglich gewesen.

Je nach Entwicklung des Magazins werden wir das in den kommenden Jahren gerne wiederholen. Denn eine Weiterentwicklung des Perl 5 Kerns hat für alle Seiten Vorteile.

Wir haben ja Anfang des Jahres eine Umfrage zum Perl-Magazin gemacht - im Rahmen einer Hausarbeit. Die Ergebnisse stehen jetzt auf unserer Webseite unter http://downloads.perl-magazin.de zur Verfügung. Es gab dabei einige interessante Ergebnisse - z.B. dass der Sammeleffekt gar nicht auftritt, wobei //SEIBERT/MEDIA das Cover genau darauf ausgerichtet hat. Es gab auch einiges an Kritik: Neben dem Layout wurde mehrfach genannt, dass ein Lektorat notwendig ist. Wir sind jetzt auf der Suche nach einem freiberuflichen Lektor/in, die wir uns leisten können. Das ist ein Punkt, an dem man die Nachteile eines sehr kleinen Magazins erkennt.

Außerdem wurde von einem Teilnehmer gewünscht, mehr Themen zu haben, die die Innereien von Perl betrachten. Da ich mich selbst dafür sehr interessiere, versuche ich, Nicholas Clark für eine Artikelserie über den Perl 5 Kern zu begeistern. Eine Entscheidung steht aber noch aus.

Für Themenwünsche sind wir immer offen, auch wenn es bis zur Umsetzung etwas dauern kann, weil häufig entsprechende Autoren gefunden werden müssen. So auch bei meinem Aufruf aus der letzten Ausgabe nach Perl-Projekten, die in $foo vorgestellt werden können. Hier habe ich von einem Leser Feedback bekommen und ich habe die vorgeschlagenen Projekte kontaktiert...

... Jetzt wünsche ich Ihnen aber viel Spaß bei der 20. Ausgabe des Perl-Magazins.

Renée Bäcker

Die Codebeispiele können mit dem Code
xk94xnh
von der Webseite www.foo-magazin.de heruntergeladen werden!

IMPRESSUM

Herausgeber:	Perl-Services.de Renée Bäcker Bergfeldstr. 23 D - 64560 Riedstadt
Redaktion:	Renée Bäcker
Anzeigen:	Renée Bäcker
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

ALLGEMEINES

6 Über die Autoren

8 SNMP::Info

13 Rezension - Programmiersprachen

PERL

16 Perl in Multithreaded Programmen einbetten

ANWENDUNGEN

22 $foo als ebook

29 Perl für Vortragende

MODULE

32 2D Spieleprogrammierung mit OpenGL - Teil 2

NEWS

44 TPF News

46 CPAN News

49 Termine

50 LINKS

The use of the camel image in association with the Perl language is a trademark of O'Reilly & Associates, Inc. Used with permission.

Alle weiterführenden Links werden auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue20

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshops, Grant Manager bei der TPF und bei der Perl-Marketing-Gruppe.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: "Programmiersprache Perl" beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

Thomas Fahle

Perl-Programmierer und Sysadmin seit 1996.

Websites:

- http://www.thomas-fahle.de
- http://Perl-Suchmaschine.de
- http://thomas-fahle.blogspot.com
- http://Perl-HowTo.de

Sebastian "tokkee" Harl

Sebastian "tokkee" Harl (*http://tokkee.org/*) beschäftigt sich seit mehreren Jahren mit Linux und Open Source Software. Er arbeitet aktiv an Projekten wie Debian, collectd und RRDtool mit und programmiert seit über 10 Jahren, vorwiegend mit C und Perl. Weiterhin beteiligt er sich regelmäßig als Dozent an verschiedenen Open-Source-Konferenzen (z.B. FrOSCon, Grazer Linuxtage) und ist aktives Mitglied der Linux User Schwabach.

Seit 2011 ist er als Trainer und Consultant in den Bereichen Betriebssystem Linux, Monitoring und Performance-Analyse bei der teamix GmbH in Nürnberg tätig. Dort hat er unter anderem die Perl-Schulung übernommen.

Tom Kirchner

Tom Kirchner, Berliner Jahrgang 1984, studierter Medieninformatiker, fand vor vielen Jahren zu Perl und erledigt seitdem fast alles in Perl. Er arbeitet zur Zeit als Softwareentwickler in Hamburg sowie als Entwickler bei Perl-Services.de. Daneben ist er bei den Perl Mongers Hamburg aktiv und hŠlt Vorträge. Sein besonderes Interesse liegt im Bereich Visual Analytics sowie allgemein Grafikprogrammierung und dort v.a. die Visualisierung von Daten mit OpenGL. Zu seinen Lieblingsmodulen auf CPAN zählen u.a. OpenGL, Imager, DBI, Parse:: RecDescent und Dancer. Zu erreichen ist er unter *http://www.tomkirchner.com*.

Thomas Fahle

SNMP::Info - SNMP mal ganz einfach

SNMP::Info

• ist eine objektorientierte Schnittstelle für Perl5, um Informationen per SNMP abzufragen oder Einstellungen eines Devices per SNMP zu setzen
• unterstützt sehr viele Netzwerkkomponenten
• bietet eine device orientierte Sicht mit einer gemeinsamen Schnittstelle für alle unterstützten Devices
• verhüllt die Details des nicht ganz so simplen Protokolls SNMP
• unterstützt SNMP v1, v2c und v3
• bietet SNMP Einsteigern einen guten Einstieg
• lässt sich einfach erweitern
• ist gut dokumentiert

SNMP

Simple Network Management Protocol (SNMP) ist ein Netzwerkprotokoll, das von Netzwerk-Management-Systemen (NMS) genutzt wird, um Netzwerkkomponenten (Devices) wie z.B. Router, Server, Switches von einer zentralen Station aus überwachen und steuern zu können.

Netdisco

SNMP::Info ist die Perl-Bibliothek hinter *Netdisco*, einem NMS für mittlere bis große Installationen. Eine Spezialität von Netdisco ist, wie der Name schon andeutet, Network Discovery. Netdisco kann via Layer 2 Discovery eine Topologie des Netzes erstellen und ist z.B. bei der Suche nach `rogue access point` recht nützlich.

SNMP::Info wird aktiv weiterentwickelt und kann ohne Netdisco in eigenen Applikationen genutzt werden.

Unterstützte Devices

SNMP::Info unterstützt zahlreiche Netzwerkkomponenten. Ein detaillierte Übersicht findet sich in der Datei *DeviceMatrix.txt*, die mit SNMP::Info ausgeliefert wird. Die Übersicht steht auch online auf der Website des Projekts bereit.

Installationsvoraussetzungen

SNMP::Info benötigt die `Net-SNMP` Bibliothek mit Perlunterstützung (SNMP::Session) und funktionierende MIB-Module für die zu unterstützenden Netzwerkkomponenten.

Die nicht ganz triviale Installation der benötigten MIB-Module, lässt sich durch die Installation der *netdisco-mibs*, einer gepflegten MIB-Sammlung, erheblich vereinfachen.

Schritt für Schritt

Konstruktor new()

Die Methode `new()` erstellt ein neues Objekt der Klasse SNMP::Info.

Pflichtparameter ist die IP-Adresse oder der DNS-Name des Devices. Wenn kein SNMP Community String angeben wird, wird `public` verwendet.

Die SNMP-Version kann über den Parameter `Version` konfiguriert werden (v2c ist Default).

Der Parameter `AutoSpecify` erlaubt SNMP::Info automatisch die passende Subklasse für das Device auszuwählen. Dazu gleich mehr.

```
#!/usr/bin/perl
use warnings;
use strict;

use SNMP::Info;

my $desthost  = '10.0.0.1';
my $community = 'public';

my $info = new SNMP::Info(

  # Auto Discover more specific Device Class
  AutoSpecify => 1,
  Debug       => 0,

  # The rest is passed to SNMP::Session
  DestHost    => $desthost,
  Community   => $community,
  Version     => 1,
) or die "Can't connect to $desthost!\n";

print "Connected to $desthost.\n";
```

Unter der Haube von SNMP::Info werkelt SNMP::Session, d.h. alle von SNMP::Session unterstützten Parameter können im Konstruktor verwendet werden.

Automatische Auswahl der richtigen Subklasse

SNMP::Info fragt dazu per SNMP auf dem Device `sysServices.0`, `sysDescr.0` und `sysObjectID.0` ab und wählt aus diesen Informationen die passende gerätespezifische Subklasse.

Die Methode `device_type()` liefert den Namen der ausgewählten Subklasse für das Device zurück.

```
print "Detected Device Type: ",
 $info->device_type(), "\n";
```

Ausgabe bei einem Cisco Catalyst 2960S:

```
Detected Device Type:
  SNMP::Info::Layer3::C6500
```

Ausgabe bei einem HP ProCurve J9021A Switch 2810-24G:

```
Detected Device Type:
   SNMP::Info::Layer2::HP
```

Innerhalb der gerätespezifischen Subklassen werden allgemeine Methoden passend für die Netzwerkkomponente überschrieben. SNMP::Info bietet so eine gemeinsame Schnittstelle für alle unterstützten Devices an.

Einfache SNMP Abfragen als Einstieg

Als Einstieg ein paar einfache SNMP-Queries, wie sie in vielen SNMP-Tutorien für Einsteiger verwendet werden.

Alle Methoden geben einen skalaren Wert zurück.

```
print "sysUpTime: ",  $info->uptime(), "\n";
print "sysContact: ", $info->contact(),"\n";
print "sysName: ",    $info->name(),   "\n";
print "sysLocation: ",
        $info->location(), "\n";
print "MAC Root Port: ", $info->mac(), "\n";

print "OS: ",     $info->os(),     "\n";
print "Vendor: ", $info->vendor(), "\n";
print "Model: ",  $info->model(),  "\n";
print "Software Version: ",
        $info->os_ver(), "\n";
```

Ausgabe bei einem Cisco Catalyst 2960S:

```
sysUpTime: ddddddddd
sysContact: XXXXXXX
sysName: XXXXXXXXXX
sysLocation: XXXXXX
MAC Root Port: xx:xx:xx:xx:xx:xx
OS: ios
Vendor: cisco
Model: 29xxStack
Software Version: 12.2(55)SE3
```

Ausgabe bei einem HP ProCurve J9021A Switch 2810-24G:

```
sysUpTime: ddddddddd
sysContact: XXXXXXX
sysName: XXXXXXXXXX
sysLocation: XXXXXX
MAC Root Port: xx:xx:xx:xx:xx:xx
OS: hp
Vendor: hp
Model: 2810-24G
Software Version: N.11.25
```

Hardwarestatus

Dank der gemeinsamen Schnittstelle für alle Devices, lässt sich der Status eingebauter Hardware, wie Netzteile, Ventilatoren oder RAM, sehr einfach abfragen. Alle aufgezeigten Methoden geben einen skalaren Wert zurück.

```
# Fan and Power supplies
print "Fan status: ",
  $info->fan(), "\n";
print "Power supply 1 status: ",
  $info->ps1_status(), "\n";
print "Power supply 2 status: ",
  $info->ps2_status(), "\n";

# Returns bytes of total memory
print "Speicher gesamt: ",
  $info->mem_total(), " Bytes\n";

# Returns bytes of free memory
print "Freier Speicher: ",
  $info->mem_free(), " Bytes\n";

# Returns bytes of used memory
print "Speicher genutzt: ",
  $info->mem_used(), " Bytes\n";
```

Ausgabe bei einem Cisco Catalyst 2960S:

```
Fan status: ok
Power supply 1 status: ok
Power supply 2 status: other
Speicher gesamt: 89697640 Bytes
Freier Speicher: 50696124 Bytes
Speicher genutzt: 39001516 Bytes
```

Ausgabe bei einem HP ProCurve J9021A Switch 2810-24G:

```
Fan status: good
Power supply 1 status: good
Power supply 2 status: notPresent
Speicher gesamt: 24034944 Bytes
Freier Speicher: 16851344 Bytes
Speicher genutzt: 7183600 Bytes
```

SNMP::Info wählt hier automatisch für jedes Device die passenden SNMP-Abfragen.

So wird beispielsweise zur Ermittlung des freien Speichers bei Cisco `ciscoMemoryPoolFree` abgefragt, bei HP hingegen `hpGlobalMemFreeBytes`.

Tabellarische Daten

Oft liegen Daten, die per SNMP ermittelt werden, als Tabellen vor.

Zur Abbildung von Tabellen nutzt SNMP::Info *parallele Hashes* mit identischen Schlüsseln als Datenstruktur. Dazu mehr in den nächsten Abschnitten.

Interface Daten

Um Interfacedaten, wie Beschreibung, Typ oder Geschwindigkeit, eines Devices zu ermitteln, benötigt man zunächst eine Liste der Interfaces, die man über die Methode `interfaces()` erhält.

```
my $interfaces = $info->interfaces();
```

`$interfaces` ist eine Referenz auf einen Hash, der als Schlüssel Interface-IDs (iid) enthält.

Gekürzte Data::Dumper Ausgabe

```
$VAR1 = {
    '10127' => '10127',
    '10131' => '10131',
 };
```

Im nächsten Schritt werden die gewünschten Interfacedaten geholt, z.B. die Geschwindigkeit via `i_speed()`.

```
my $interfaces_speed = $info->i_speed();
```

`$interfaces_speed` ist ebenfalls eine Referenz auf einen Hash, dessen Schlüssel, wie oben, Interface-IDs enthalten; die jeweiligen Geschwindigkeiten werden als Werte gespeichert.

Gekürzte Data::Dumper Ausgabe

```
$VAR1 = {
    '10127' => '1.0 Gbps',
    '10131' => '100 Mbps',
 };
```

Dieser Vorgang wird für weitere Interfacedaten beliebig oft wiederholt.

So liefert beispielsweise `i_mac()` die MAC-Adresse (sofern vorhanden) und `i_description()` eine Beschreibung des Interfaces.

Zwecks Ausgabe der Werte, läuft man in einer Schleife über die Schlüssel (iid) der Intefaceliste und holt mit eben diesem Schlüssel in den parallelen Hashes die Werte.

```
my $interfaces = $info->interfaces();
my $interfaces_description =
  $info->i_description();
my $interfaces_mac   = $info->i_mac();
my $interfaces_speed = $info->i_speed();

foreach my $iid ( sort { $a <=> $b }
  keys %$interfaces )
{
  my $description =
    $interfaces_description->{$iid}
    || 'No interface description.';
  my $mac = $interfaces_mac->{$iid}
    || 'Keine MAC-Adresse gefunden.';
  my $speed = $interfaces_speed->{$iid}
    || 'Unbekannte Geschwindigkeit.';

  printf( "%20s %10s\n",
    $description, $speed );
}
```

Beispiel Ausgabe:

```
Vlan1                     1.0 Gbps
GigabitEthernet1/0/47      10 Mbps
GigabitEthernet1/0/48     100 Mbps
GigabitEthernet1/0/49     1.0 Gbps
```

SNMP::Info zeigt die Geschwindigkeiten in einem menschenlesbarem Format an und wählt im Hintergrund die passende SNMP-Abfrage für das jeweilige Gerät, z.B. `ifSpeed` oder `ifHighSpeed`.

Interface Statistiken

Statistische Interfacedaten, wie empfangene und gesendete Daten oder Übertragungsfehler, lassen sich mit den Methoden `i_octet_in()`, `i_octets_out()`, `i_errors_in()` und `i_errors_out()` einfach ermitteln.

```
my $interfaces = $info->interfaces();
my $interface_descriptions =
  $info->i_description();

my $i_octet_in   = $info->i_octet_in();
my $i_octets_out = $info->i_octets_out();
my $i_errors_in  = $info->i_errors_in();
my $i_errors_out = $info->i_errors_out();

foreach my $iid ( sort { $a <=> $b }
  keys %$interfaces )
{
  my $description =
    $interface_descriptions->{$iid}
    || 'No description';
  my $octets_in = $i_octet_in->{$iid}
    || 0;
  my $octets_out = $i_octets_out->{$iid}
    || 0;
  my $errors_in = $i_errors_in->{$iid}
    || 0;
  my $errors_out = $i_errors_out->{$iid}
    || 0;

  printf(
    "%20s: %10s %10s %10s %10s\n",
    $description, $octets_in, $octets_out,
    $errors_in, $errors_out
  );
}
```

Beispiel Ausgabe:

```
GigabitEthernet1/0/48: 2017710952  0 0  0
GigabitEthernet1/0/49: 3239869357  0 0  0
```

Network Discovery (CDP und LLDP)

Das nachfolgende Beispiel, das ich der SNMP::Info Dokumention entnommen habe, ermittelt die IP-Adressen und physischen Portnamen auf den direkt angeschlossen Devices (Neighborhood) mittels Cisco Discovery Protocol (CDP) oder Link Layer Discovery Protocol (LLDP).

SNMP::Info verhüllt, wie gewohnt, die internen Details hinter einer allgemeinen Schnittstelle und schaltet intern je nach Device auf das entsprechende Protokoll um.

Als Mapping Tabelle zwischen den Interface-IDs der verschiedenen Devices dient `c_if()`. Der zurückgegebene Hash enthält die remote Interface-IDs als Keys und die lokalen Interface-IDs als Werte.

```
my $interfaces = $info->interfaces();

# Get CDP Neighbor info

# Returns the mapping
# to the SNMP Interface Table
my $c_if   = $info->c_if();

# returns remote IPV4 address
my $c_ip   = $info->c_ip();

# Returns remote port ID
my $c_port = $info->c_port();

# Print out data per port
foreach my $iid ( keys %$interfaces ) {

  # Print out physical port name,
  # not snmp iid
  my $port = $interfaces->{$iid};

  # The CDP Table has table entries
  # different than the interface
  # tables.
  # So we use c_if to get the map
  # from cdp table to interface
  # table.

  my %c_map = reverse %$c_if;
  my $c_key = $c_map{$iid};
  unless ( defined $c_key ) {
    next;
  }
  my $neighbor_ip   = $c_ip->{$c_key};
  my $neighbor_port = $c_port->{$c_key};

  if ( defined $neighbor_ip ) {
    print "Local port '$port'
          connected to '$neighbor_ip'
          remote port '$neighbor_port'\n";
  }
}
```

Beispiel Ausgabe:

```
Local port 'GigabitEthernet1/0/48'
       connected to '172.xx.xxx.x'
       remote port 'GigabitEthernet1/0/48'
Local port 'GigabitEthernet1/0/49'
       connected to '172.xx.xx.xx'
       remote port '9/3'
```

SNMP::Info kann noch viel mehr

SNMP::Info kann deutlich mehr als hier gezeigt. Eine vollständige Übersicht würde den Rahmen dieses Artikels deutlich sprengen.

Ich empfehle einen Blick ins umfangreiche, gut gepflegte Handbuch für mehr Informationen.

Hier könnte
Ihre Werbung stehen!

Interesse?

Email: werbung@foo-magazin.de
Internet: http://www.foo-magazin.de (hier finden Sie die aktuellen Mediadaten)

Herbert Breunung

Rezension - Programmiersprachen

Sieben Sprachen, Sieben Wochen:
Verstehen sie die modernen Sprachkonzepte
Bruce A. Tate
1. Auflage Juni 2011
ISBN 978-3-89721-322-7
O'Reilly (deutsch)
360 Seiten, broschiert
€ 34,90

Visionäre der Programmierung:
Die Sprachen und ihre Schöpfer
Federico Biancuzzi, Shane Warden
1. Auflage September 2009
ISBN 978-3-89721-934-2
504 Seiten, broschiert
€ 34,90

Um diese Rubrik zu vervollständigen, werden ab dieser Folge auch ausgewählte Neuerscheinungen erwähnt, die meist nichts mit Perl, wohl aber mit Programmierung, Administration oder Software im weitesten Sinne zu tun haben. Das wären für dieses Mal einige der bekannten Perl-Klassiker von O'Reilly. Learning Perl (in der letzten Ausgabe vorgestellt) erschien im Juni in seiner sechsten Edition, die auf 5.14 basiert (vorerst nur auf Englisch). Einen Monat später zog die Taschenreferenz von Johan Vromans nach und reicht nun bis 5.14.1. Auch die Bibel "Programming Perl" wird aufgefrischt, was voraussichtlich ein großes Thema der nächsten Episode wird. Eine Neuauflage des ebenfalls beim vorigen Mal vorgestellten "Perl Best Practices" könnte sich 2012 ereignen. Damian plant derzeit Inhalte, hat aber noch keine Zusage vom Verlag.

Der Galileo-Verlag druckte ein dickes Buch (1118 Seiten + 2DVD) von Marcus Fischer zu Ubuntu 11.04, unter dem Namen "Ubuntu GNU/Linux", das es wahlweise als kostenloses "openbook" unter *http://openbook.galileocomputing.de/ubuntu/* gibt, aber auch für rund 40 Euro gekauft werden kann. Nur 25, pardon 24,90 kostet "Ubuntu Desktop und Server - Ubuntu 11.04: Installation, Anwendung, Administration" von Klaus H. Schmidt, vom Hanser-Verlag. Es hat lediglich halb so viel Seiten und umfasst noch 2 CD's. Zum Thema gehörig aber distributionsübergreifend gab Addison-Wesley die "Linux Kommando-Referenz" von Michael Kofler heraus. Kurzweiliges Schmökern erlaubt "Erfolgreiches Design" von Frederick P. Brooks, das heise Online besprach *http://www.heise.de/developer/artikel/Erfolgreiches-Design-1283360.html*. Kostenlose (und vergriffene) Lektüre (darunter einige Linux-Titel) bietet auch O'Reilly unter *http://www.oreilly.de/german/freebooks*.

Die heutigen zwei Titel drehen sich ebenfalls nicht um Perl, denn es gibt ja eine breite Fülle an Programmiersprachen. Trotz des ähnlichen Themas sind die Werke sehr gegensätzlich. Das Eine schaut voraus, das Andere zurück, das eine führt Monologe, das zweite Dialoge. Eines möchte eher zum Programmieren anregen, das Andere zum Nachdenken. Gemeinsam ist ihnen lediglich der Wunsch verschiedene Sprachen besser zu verstehen.

Sieben Sprachen, Sieben Wochen

Das klingt wie eine Ankündigung für einen Gewaltakt, ist aber in Wirklichkeit ein sehr gut und bequem lesbares Buch. Freilich sollte man in mindestens einer Sprache sicher sein und grundlegende Programmierkonzepte durchdrungen haben, um sich hierbei erfreuen zu können. Aber wenn Bruce A. Tate schreibt, er fordere seine Leser stark, dann meint er damit, dass er versucht deren Horizont zu erweitern. Als erfahrener Autor und Vortragender macht er dabei fast alles Menschenmögliche um das zu erleichtern. Die Anspielungen an bekannte Filme von *Ferris macht blau* über *Star Wars* bis *Mary Poppins* sind oft eine gute Beschreibung, geben den Kapiteln eine Handlung und vermitteln ein Gefühl für die vorgestellten Konzepte.

Um diese Konzepte geht es auch eigentlich in diesem Buch, weniger um Detailwissen. Dieses dient nur um im Augenblick des Ausprobierens ein Gefühl für die Sprache zu bekommen um später entscheiden zu können, welche Sprache man "richtig" lernen mag. Hier geht es aber nicht um *Ruby*, *Io*, *Prolog*, *Scala*, *Erlang*, *Closure* oder *Haskell* sondern alleinig um Mechanismen wie *Mixins*, *Duck Typing*, *Monaden*, *Funktionen Höherer Ordnung*, *Logische Programmierung* und so weiter. Konzepte denen sich viele Javaprogrammierer zunehmend stellen müssen und auch wollen, wie es zum Beispiel die wachsende Beliebtheit von Ruby oder Scala zeigt. Manches davon gibt es in Perl schon lange (besonders wofür Ruby gelobt wird), vieles jedoch erst rudimentär oder weniger elegant und ist oft in Modulen verstaut oder erst in Perl 6 anzutreffen. Deshalb tut es gut zu lernen, ohne jedoch einem Massenwahn (Hype) zu verfallen. Der eigentliche Verlag (O'Reilly vertreibt nur die deutsche Übersetzung) *The Pragmatic Bookshelf* bewegt sich in der Ruby-Mac-Agile-Blase, der gegenüber der gestandene Perlprogrammierer zu recht misstrauisch ist. Freimütig gesteht Bruce Tate, sich die meisten Sprachen für dieses Buch auf die schnelle angeeignet zu haben. Die nötige Erfahrung und den Überblick liefern die hochkarätigen Koautoren. Manche Erklärungen sind nicht notwendig, wenn man bereits eine dynamische Sprache wie Perl gut kennt. Die wesentlichen Kernsätze werden oft genug wiederholt, so dass es sich wirklich wie ein Roman durchlesen lässt. Der einzige echte Nachteil dieser deutschen Übersetzung liegt in den Beispielen, die wie üblich englisch blieben. Dieses Buch lebt von den vielen Zeilen, die simuliert ausgeführt (Ergebnisse sind wie in der Befehlszeile ausgedruckt) und intensiv besprochen werden. Sie enthalten englische Variablen- und Funktionsnamen welche zum Verständnis nicht unerheblich sind und auch Anspielungen auf die Filmcharaktere enthalten, mit denen die jeweilige Sprache verglichen wird. Das kann ausschließlich deutschsprachigen Lesern entgehen.

Für Menschen, die neben dem Beruf oder Schule die Ideen hinter den (vor allem funktionalen) Programmiersprachen grundsätzlich begreifen wollen, ist die Mischung ideal. Ohne viele Wälzer oder Papers durcharbeiten zu müssen, bekommt man die wichtigen Informationshappen frei Haus und kann sich das Verständnis in Kürze anlesen und praktisch selbst im Dialog mit dem Interpreter oder Compiler (geführt) erarbeiten. So ausgestattet lassen sich Sprachen wesentlich schneller lernen und auch Artikel über Programmierung besser verstehen. Und selbst wenn man keine dieser Sprachen jemals nutzen wird, wurde man (um viele Konzepte bereichert) ein vielseitigerer und effektiverer Programmierer.

Visionäre der Programmierung

Das zweite Druckwerk ist das *Galore* der Programmiersprachenväter - ein reiner Interview-Band. Er misst sich aber nicht an Trends, denn die vorgestellten 17 Sprachen haben alle den sprichwörtlichen Test der Zeit bestanden. *APL*, *Awk*, *Basic*, *C#*, *C++*, *Forth*, *Java*, *Lua*, *ML Objective-C*, *Perl*, *Postscript*, *Python*, *SQL*, *UML* sind längst Teil der Programmierkultur geworden. Einzig *Haskell* ist beiderseits vertreten.

Hier wird tiefer gegraben, aber auch mehr Verständnis vorausgesetzt. Und es gibt kein übergeordnetes Lernziel. Die Gespräche fließen natürlich, bewegen sich aber jedes Mal um drei ausgewiesene Themen, wovon das erste immer die Entstehung der entsprechenden Sprache ist. Ansonsten wurde über Gott und die Welt geplaudert was hier Softwareoptimierung, Konsequenzen von Kompatibilität und Evolution der Syntax, Debugging, Dokumentation, DSL's, akademische Ausbildung, Chipdesign und dergleichen mehr beinhaltet. Natürlich wurden auch die aktuellen Probleme der Sprachen wie Nebenläufigkeit und wachsende Komplexität der Systeme stark diskutiert. Der Text enthält sehr wenig Code und es kommen (selten) auch Antworten vor, die über zwei Seiten gehen. Wenn eine Sprache mehrere Ideengeber hatte, wurden meist auch alle befragt.

Die Gespräche wurden sie von einem Duo geführt, einem Journalisten und jemand mit mehr Programmiererfahrung. Hinter dessen Namen (Shane Warden) verbirgt sich kein geringerer als der in Perl-Kreisen bekannte cromatic, der jetzige Gestalter von *http://perl.com* und Autor von *Modern Perl*. Da er engstens mit Perl 5 und 6 vertraut ist, entstand ein für $foo-Interessierte besonders lesenswerte Befragung von Larry Wall. Fast nirgends ist die Philosophie und Geschichte von Perl so lässig, klar und umfangreich (nicht umfassend) dargelegt wie in diesem Gedankenaustausch. Wer immer schon einmal genau verstehen wollte, worum es bei Perl 6 wirklich geht und was Larry dazu antreibt, der lese dazu die Seiten 381 bis 400.

Die gut vorbereiteten Fragesteller versuchten immer wieder, ihre Gegenüber zum Nachdenken und zu bisher nicht gegebenen Antworten zu bringen um die gebündelte Erfahrung der Antwortgeber auszuschöpfen. Überhaupt ist es die Erfahrung der Befragten (20 Jahre mindestens pro Person), die diese 460 Seiten interessant macht. Zwischen einigen vorhersehbaren Antworten, tauchen öfters Erinnerungen und Einsichten auf, welche die eigene Bildung bereichern können. Zum einen wirkte es anregend, die Ansichten der verdienten Veteranen zu verschiedenen technischen und sozialen Problemen zu erfahren. Zum anderen ist es klärend, die menschliche Natur der Autoren zu erkennen, welche das schrieben, was man selber täglich nutzt. Zum Beispiel wenn sie öfters auch "nur" `print` zum Debuggen verwenden. Zuletzt vertieft es das eigene Verständnis der Softwarewelt ungemein, je mehr man um das Wollen und Denken jener weiß, die sie Schicht um Schicht, Generation um Generation erschufen.

Deutscher Perl® Workshop 2012
in Erlangen
5. bis 7. März 2012

Schwerpunkte:
modernes Perl,
Einsteiger,
praktische Workshops

Call for Papers und Anmeldung

http://conferences.yapceurope.org/gpw2012

Veranstalter:
Kommunikationsnetz Franken
http://www.franken.de/gpw2012

KNF

Sebastian "tokkee" Harl

Perl in Multithreaded Programmen einbetten

Perl ist eine Skriptsprache, die sich einfach in die Programmiersprache C einbetten lässt. Dies ermöglicht Perl-Code direkt aus C zu nutzen und die Vorteile beider Sprachen (schneller Code in C gegenüber schnell geschriebenem Code in Perl) miteinander zu vereinen (bekannt ist dies insbesondere durch `mod_perl`, welches Perl in den Apache Webserver einbettet). Im Falle von parallelem Zugriff auf Perl aus einem multithreaded Programm ist die Einbettung jedoch nicht mehr trivial, da der Perl Interpreter selbst nicht thread-safe ist.

Im einfachsten Fall lässt sich dieses Problem dadurch lösen, dass der Zugriff auf den Perl Interpreter serialisiert wird. Damit kann das gleiche Objekt aus allen Threads verwendet werden. Die Kosten dafür sind eine Einschränkung des echten parallelen Betriebs des Programms. Dafür nötige Mechanismen, insbesondere zur abwechselnden Nutzung des Interpreters aus verschiedenen Threads, stellt Perl bereits zur Verfügung.

Mit einigen Tricks und Kniffen lassen sich diese Mechanismen allerdings erweitern und für einen echten parallelen Zugriff auf Perl nutzen. Dann allerdings unter Verwendung von mehreren Interpretern. Der hier gezeigte Ansatz baut auf dem Konzept der in Perl 5.6 eingeführten Interpreter-Threads (*ithreads*) auf und "bettet" dynamisch erzeugte ithreads in (C-)Threads ein. Der Artikel erläutert zunächst die nötigen Hintergrundinformationen und geht dann am Beispiel des sehr parallelisierten Statistik-Sammlers "collectd" auf eine konkrete Umsetzung ein.

Das grundsätzliche Vorgehen zur Einbettung von Perl ist in der Manpage *perlembed* beschrieben. Perl (der Interpreter) ist selbst in C geschrieben, wobei die eigentliche Umsetzung davon als die Bibliothek `libperl` verfügbar ist. Diese stellt das `PerlInterpreter` Objekt bereit, mit dessen Methoden das Verhalten und der Programmablauf gesteuert werden. Ein minimales Beispiel zur Einbettung von Perl schaut folgendermaßen aus:

```
#include <EXTERN.h>
#include <perl.h>

int
main(int argc, char **argv, char **env)
{
 PerlInterpreter *my_perl;
 PERL_SYS_INIT3(&argc, &argv, &en);
 my_perl = perl_alloc();
 perl_construct(my_perl);
 perl_parse(my_perl, NULL, argc, argv, NULL);
 perl_run(my_perl);
 perl_destruct(my_perl);
 perl_free(my_perl);
 PERL_SYS_TERM();
}
```

Das daraus resultierende Programm verhält sich im Wesentlichen wie das Programm `perl` selbst. Zu beachten ist, dass sich der Interpreter nicht von selbst beendet, sondern dies explizit durch den Aufruf von `perl_destruct` geschieht. Dies ermöglicht, durch die Vorgabe von Parametern durch das `argv` Array wie `-e 1`, einen Interpreter zu initialisieren, welcher erst später tatsächlich Code ausführt. Es ist dann möglich, belieben Perl-Code als Module nachzuladen und auszuführen.

Weiterhin ist es möglich, in einem Programm mehrere Interpreter gleichzeitig aktiv zu nutzen. Dabei sind einige Punkte zu beachten, die im Detail im Abschnitt **Maintaining multiple interpreter instances** in der Manpage *perlembed* dokumentiert sind. Perl bietet außerdem bereits die Möglichkeit, auf Interpreter-Objekte aus mehreren Threads zuzugreifen. Dazu später mehr.

In jedem Fall ist es essentiell, dass das gleiche Interpreter-Objekt nie von zwei Threads gleichzeitig verwendet wird, da dies zu einem inkonsistenten Zustand der Datenstrukturen und somit zum Absturz des Programms führen würde.

Einführung in Perl Interpreter-Threads

Beginnend mit Version 5.6 verwendet Perl das Konzept so genannter Interpreter-Threads, kurz **ithreads**. Dabei ist jedem Perl-Thread ein eigener Interpreter zugeordnet, welcher wiederum in einem eigenen Betriebssystem-Thread läuft. Das hat zur Folge, dass standardmäßig keine gemeinsame Daten-Nutzung zwischen den Threads möglich ist. Diese Möglichkeit wird jedoch durch das Modul *threads::shared* bereit gestellt. Dazu müssen gemeinsam genutzte Variablen explizit als solche deklariert werden. Sie werden dabei nicht im Kontext des jeweiligen Interpreters alloziert, sondern in einem speziell dafür vorgesehenen separaten `PerlInterpreter` Objekt. Der Zugriff darauf aus den Perl-Threads wird dadurch realisiert, dass eine so genannte "magische" Variable im Kontext der Threads erzeugt wird, die Zugriffe in den separaten Interpreter weiterleitet. Details zur Verwendung und auch zu Limitierungen sind in der Manpage des Moduls dokumentiert.

Analog zu diesem Konzept lässt sich ein paralleler Zugriff auf Perl dadurch realisieren, dass bei dem Zugriff auf Perl dynamisch ein Perl-Interpreter in dem entsprechenden Betriebssystem-Thread erzeugt wird und von diesem Moment an in den Betriebssystem-Thread eingebettet wird.

Pthreads und Thread-spezifische Daten

POSIX Threads (pthreads) werden von nahezu jedem UNIX/Linux System unterstützt. Im Gegensatz zu Perl-Threads teilen sich diese standardmäßig den kompletten Adressraum und damit alle vorhandenen Daten. Um eine Zuordnung eines Perl-Interpreters zu einem Betriebssystem-Thread zu ermöglichen, ist es daher notwendig, bestimmte Daten in den verschiedenen Threads in unterschiedlichen Varianten vorzuhalten. Dazu kann das von pthreads zur Verfügung gestellte Konzept der so genannten Thread-spezifischen Daten genutzt werden. Hierbei wird für jeden Thread ein eigener Speicherbereich, der so genannte **TSD** Bereich (*thread-specific data*), vorgehalten, welcher über einen Schlüssel indiziert wird. Der Schlüssel selbst steht dabei allen Threads zur Verfügung. Die Daten, die ihm zugeordnet sind, können aber von Thread zu Thread unterschiedlich sein. Bei Erzeugung eben dieser werden alle zugeordneten Werte auf `NULL` initialisiert. Damit ist es möglich, einen "neuen" Thread zu erkennen. Im Gegenzug dazu wird bei seiner Beendigung eine dem Schlüssel zugeordnete Funktion aufgerufen, die sich um notwendige Aufräumarbeiten kümmert und gleichzeitig die Beendigung signalisiert. Somit ist es möglich, den kompletten Lebenszyklus zu koordinieren. Details zu den Thread-spezifischen Daten der pthreads liefert die Manpages zu *pthread_key_create(3)* und *pthread_setspecific(3)* beziehungsweise *pthread_getspecific(3)*.

Bei der Verwendung von mehreren Perl-Interpretern in einem Prozess muss der jeweils Aktive durch den Aufruf von `PERL_SET_CONTEXT` als solcher markiert werden. Dieses Makro bindet den Interpreter dabei, unter Verwendung von Thread-spezifischen Daten, an den aktuellen Thread. Somit ist es möglich, eine feste Bindung zwischen Betriebssystem-Thread und Perl-Interpreter herzustellen. Durch die oben gezeigten Möglichkeiten können der Start und das Ende erkannt und entsprechend behandelt werden.

Schnittstellen zwischen Perl und C

Bei der Einbettung von Perl in C gibt es diverse Kontaktpunkte zwischen den beiden Sprachen. Zum einen wird das C-Programm Aufrufe von Perl-Code durchführen und zum anderen ist es auch möglich, aus Perl wiederum C-Code aufzurufen.

Letzteres geschieht unter Verwendung der **XS** (*eXternal Subroutine*) Schnittstelle, welche von einer Vielzahl von Perl-Modulen verwendet wird, die eine Schnittstelle zu C-Bibliotheken bieten. Beim Aufruf des C-Codes muss aus Perl-Sicht auch in einem multithreaded Programm kein besonderes Vorgehen beachtet werden. Der aufgerufene Code muss lediglich thread-safe sein, was an dieser Stelle durch die parallele Natur des umgebenden Programms bereits gegeben sein sollte. Die Verwendung von XS ist in den Manpages *perlxs* und *perlxstut* ausführlich dokumentiert.

Der Aufruf von Perl-Code aus C heraus kann auf unterschiedliche Art und Weise geschehen. Zum Einen wird nach Erzeugung eines Perl-Interpreter Objekts und Aufruf der `perl_parse` Methode der durch die Argumente spezifizierte Code ausgeführt. Dies geschieht äquivalent zum Aufruf des Pro-

gramms `perl` entweder durch Angabe einer Perl-Quellcode-Datei oder durch Angabe von Perl-Statements mittels des Schalters `-e` (vgl. die Manpage *perlrun*). Weiterhin kann Perl-Code durch die Verwendung von `load_module` nachgeladen und dabei ausgeführt werden. Zum anderen lassen sich Teile des Perl-Codes durch den Aufruf von Subroutinen zu jedem beliebigen Zeitpunkt mittels `call_pv` ausführen.

Im Folgenden wird davon ausgegangen, dass das Programm, welches von Perl Gebrauch machen möchte, wie folgt aufgebaut ist: Zunächst wird eine Initialisierung durchgeführt. Aus Perl-Sicht wird hier erster Code ausgeführt und gegebenenfalls Module nachgeladen. Danach wird die tatsächliche Arbeit des Programms verrichtet, wozu (teilweise) auf Perl zurückgegriffen wird. Dies kann durch Verwendung von registrierten Callback-Funktionen oder durch Kommunikation mit einem weitestgehend getrennt laufendem Perl-Thread geschehen. Der letzte Fall ist in diesem Kontext nicht näher interessant, da hierbei beim Einbetten keine besonderen Maßnahmen zu treffen sind und die Kommunikation über übliche Interprozesskommunikation abgewickelt werden kann. Ohne Beschränkung der Allgemeinheit kann im Wesentlichen davon ausgegangen werden, dass jedes Programm, das Perl einbettet, nach diesem Schema funktioniert.

Überblick über collectd

Im Folgenden wird der in der Software **collectd** verwendete Ansatz beschrieben, um Perl in einem multithreaded Programm einzubetten. Dabei handelt es sich um einen in C geschriebenen UNIX-Daemon, welcher periodisch Leistungsdaten eines Rechners oder von Rechenzentrumshardware erfassen, verarbeiten und speichern kann. Diese Daten, wie CPU-Auslastung, Speicherverbrauch, uvm., lassen sich dann zur Performance-Analyse und Kapazitätsplanung nutzen. Damit lassen sich Flaschenhälse bei der Auslastung eines Systems identifizieren und künftige Probleme durch geeignete Vorausplanung vermeiden. Zu diesem Zweck wird der zeitliche Verlauf der Daten aufgespeichert, damit beispielsweise eine graphische Aufbereitung möglich wird.

Der Aufbau der Software ist sehr modular. Der Daemon selbst übernimmt lediglich die Verwaltung von Plugins, in denen die eigentliche Funktionalität zur Datenabfrage und Weiterverarbeitung umgesetzt ist. Damit werden externe Abhängigkeiten im Daemon selbst vermieden und man erreicht ein hohes Maß an Flexibilität und Erweiterbarkeit. Da der Daemon dauerhaft im Speicher läuft und somit regelmäßiger Startup-Overhead entfällt, lässt er sich mit einer Standardauflösung von zehn Sekunden betreiben, ohne eine nennenswerte Auswirkung auf das umgebende System zu haben. Die Software läuft auf allen üblichen UNIX- und Linux-Systemen und wird als freie Software verteilt.

In der Standard-Distribution enthält collectd bereits eine Vielzahl an mitgelieferten Plugins. Diese umfassen die Datenabfrage von Standardkomponenten, wie CPU und Speicher, über sehr spezialisierte Plugins, die detaillierte Werte zu speziellen Systemeigenschaften, wie beispielsweise Detailinformationen zu einzelnen QDiscs des Linux netlink Subsystems abfragen, bis hin zu generischen Plugins, die ein Framework anbieten, über das der Benutzer nahezu beliebige Werte aus verschiedenen Datenquellen wie SNMP oder SQL abfragen kann. Eine Übersicht über vorhandene Plugins und weitere Features ist auf der Website, unter *http://collectd.org/* zu finden.

Die Kommunikation zwischen dem Daemon und den Plugins geschieht durch diverse Callback-Funktionen, welche das Plugin beim Daemon registriert. Die grundsätzliche Arbeitsweise ist wie folgt: Der Daemon läuft in einer Endlosschleife und ruft nach Ablauf des jeweiligen Intervalls die Lese-Callbacks der Plugins auf. Diese übermitteln die abgefragten Daten an den Daemon, welcher sie wiederum an alle registrierten Schreib-Callbacks weiterleitet. Zusätzlich stehen weitere Möglichkeiten, wie die Filterung der Daten vor der Übermittelung an die Schreib-Plugins, zur Verfügung. Die Datenabfrage und alle darauf folgenden Schritte passieren parallel in mehreren Threads, damit Wartezeiten auf Ein-/Ausgabeoperationen eine möglichst geringe Auswirkung auf die Datenerhebung haben. Weiterhin betreiben einige Plugins eigene Threads, um gewissen Operationen, wie das Empfangen von Daten über das Netzwerk, asynchron auszuführen. Dabei können Threads jederzeit dynamisch entstehen und beendet werden ohne das eine zentrale Stelle diese verwaltet. Die Ausführung dieser Threads kann weiterhin in beliebten Plugins passieren, beispielsweise durch Protokollierung von Fehlermeldungen, was ebenfalls durch registrierte Callbacks umgesetzt ist.

Dynamische Einbettung von Interpreter-Threads

Die Einbindung von Perl in collectd geschieht ebenfalls durch ein Plugin (ähnlich mod_perl im Apache Webserver). Dieses kann weitere, in Perl geschriebene Plugins nachladen und stellt diesen die interne API von collectd zur Verfügung. Diese Schnittstelle ist mittels XS direkt im Plugin und etwas Perl-Code implementiert und wird allen geladenen Perl-Modulen automatisch zur Verfügung gestellt.

Die Perl-Plugins sind als Perl-Modul ausgelegt und werden beim Starten des Daemons geladen und initialisiert. Wie bei einem Perl-Modul üblich wird dazu der komplette Code übersetzt und ausgeführt. Der dabei erzeugte Parse-Tree wird (in einem `PerlInterpreter` Objekt) über die komplette Laufzeit im Speicher des Daemons vorgehalten, wodurch bei folgenden Aufrufen von Perl-Code der zum Übersetzen nötige Overhead entfällt. Die Plugins registrieren während der Initialisierung, ebenso wie in C geschriebene Plugins, Callback-Funktionen. Diese werden vom Daemon über das "perl" Plugin in den oben beschriebenen Phasen aufgerufen.

Da nicht klar ist, aus welchem Thread zu welchem Zeitpunkt ein Aufruf einer Callback-Funktion geschieht, muss die Einbettung von Perl in einen Thread vollständig transparent und automatisch geschehen. Wie oben beschrieben ist hierbei nur der Aufruf von Perl aus C relevant.

Initial wird ein einzelner Perl-Interpreter gestartet, der zunächst keinen weiteren Code ausführt, außer den zur collectd API gehörenden Code zu laden. In diesem Interpreter werden dann alle Plugins geladen (vgl. Abschnitt "Limitierungen" unten). Im Folgenden wird dieser Interpreter als Basis für alle weiteren erzeugten `PerlInterpreter` Objekte verwendet.

Bei einem Übergang von C nach Perl wird zunächst der zum entsprechenden Thread gehörende Interpreter ermittelt. Dazu wird das Makro `dTHX` (*thread context declaration*) verwendet, welches wiederum `PERL_GET_CONTEXT` aufruft, um die Variable `aTHX` (*thread context argument*) zu initialisieren. Sollte einem Thread noch kein Interpreter zugeordnet sein, das heißt das entsprechende Thread-spezifische Datum ist `NULL`, dann ist `aTHX` entsprechend ebenfalls mit `NULL` initialisiert. In diesem Fall wird auf Grundlage des anfangs erzeugten Interpreters (im Folgenden `initial_perl` genannt) ein neuer Interpreter erzeugt. Zu diesem Zweck wird die Funktion `perl_clone` verwendet, wodurch der Interpreter mit allen Daten und dem kompilierten Zwischencode kopiert wird. Der neue Interpreter wird von Perl automatisch dem aktuellen Thread zugeordnet und somit ab diesem Zeitpunkt in diesem Thread verwendet. Ein Zugriff aus einem anderen Thread wird auf diese Art und Weise verhindert, so dass es zu keinem parallelen Zugriff auf das `PerlInterpreter` Objekt kommt.

```
static void
call_perl(void)
{
    dTHX;

    if (aTHX == NULL) {
        perl_thread_t *t = NULL;
        pthread_mutex_lock(perl_mutex);
        t = perl_thread_create(
                            initial_perl);
        pthread_mutex_unlock(perl_mutex);
        aTHX = t->interp;
    }
    /* call registered Perl callback */
}
```

Zur Verwaltung der eingebetteten Perl-Threads werden diese in einer verketteten Liste gespeichert. Dies ermöglicht, dass alle beim Beenden des Daemons sauber heruntergefahren werden können. Der dazu erstellte Datentyp (im Beispiel `perl_thread_t`) enthält einen Zeiger auf den zum Thread gehörenden Interpreter und die zur Verkettung nötigen Pointer. Um parallelen Zugriff auf die verkettete Liste zu verhindern, wird das Erstellen des Perl-Threads mit einer Mutex geschützt (siehe *pthread_mutex_lock(3)* für Details).

Neben der verketteten Liste wird das neu erzeugte `perl_thread_t` Objekt auch im TSD des Threads (vgl. "Thread-spezifische Daten") gespeichert. Bei der Erzeugung des zugehörigen Schlüssels wird eine Funktion zum ordnungsgemäßen Beenden des Perl-Threads angegeben. Dadurch wird dieser automatisch beendet sobald der umgebende Betriebssystem-Thread beendet wird.

```
static void
perl_thread_destructor(void *arg)
{
    perl_thread_t *t = arg;

    pthread_mutex_lock(perl_mutex);
    destroy_perl_thread(t);
    pthread_mutex_unlock(perl_mutex);
}
```

Die Funktion `destroy_perl_thread` kümmert sich dabei um das Beenden des Interpreters mittels `perl_destruct` und `perl_free` und das Aushängen aus der verketteten Liste. Entsprechend wird diese Aktion wiederum mit der Mutex geschützt.

Gemeinsame Verwendung von Subroutinen in mehreren Threads

Wie in der Manpage beschrieben, gibt es einige Einschränkungen, welche Art von Variablen durch das Modul `threads::shared` mehreren Threads gleichzeitig zur Verfügung gestellt werden können. Insbesondere ist es nicht möglich, Referenzen auf Subroutinen gemeinsam zu nutzen. Daher werden in collectd die Callback-Funktionen von Perl-Plugins an Hand ihres Namens identifiziert. Zu diesem Zweck bietet die von dem "perl" Plugin exportierte API die Funktion `call_by_name` an. Diese erwartet den Namen der aufzurufenden Subroutine in der Variablen `Collectd::cb_name` und ruft diese dann auf, ohne den Stack zu verändern. Dadurch werden die an `call_by_name` übergebenen Argumente direkt an die Subroutine weitergereicht und der Rückgabewert von `call_by_name` entspricht dem der aufgerufenen Subroutine.

```
$Collectd::cb_name = "mysub";
$v = Collectd::call_by_name(1, 2, 3);

# dies entspricht:
$v = mysub(1, 2, 3);
```

Gemeinsame Verwendung von komplexen Datentypen in mehreren Threads

Analog zum vorherigen Abschnitt ist die gemeinsame Verwendung von komplexen Datentypen unter Umständen schwierig oder nicht möglich. Dies kann in einigen Fällen umgangen werden, falls die Datenstruktur in C implementiert werden kann. Äquivalent zu der Umsetzung von `threads::shared` können Perl-Variablen deklariert werden, die mit "Magie" belegt sind. Bei einem lesenden oder schreibenden Zugriff auf eine solche Variable können im Hintergrund beliebig komplexe Operationen ausgeführt werden (vgl. die Perl-Funktion `tie`). Dies ermöglicht ganz allgemein den Zugriff auf Variablen im C-Code und wird beispielsweise in collectd verwendet, um Perl-Plugins kompletten Zugriff auf globale Variablen zu geben, die auch in anderen Plugins verwendet werden. Eine ausführliche Dokumentation zu dem Thema findet sich im Abschnitt "Magic Variables" und den darauf Folgenden in der *perlguts* Manpage.

Die Magie hinter den Zugriffen auf eine solche Variable wird durch eine so genannte "Magic Virtual Table" definiert. Hier werden Funktionen angegeben, die bei verschiedenen Zugriffsarten ausgeführt werden und die hinter der Operation steckende Funktionalität umsetzen. Ein einfaches Beispiel, welches `set` und `get` (die ersten beiden Elemente in der Tabelle) verwendet, um Zugriff auf skalare C-Variablen umzusetzen, zeigt das Listing 1.

Mit diesem Code wird ein Zugriff auf die Perl-Variable `$var` auf einen Zugriff auf den C-String `c_string` abgebildet.

Limitierungen

Der gezeigte Ansatz ermöglicht die vollkommen transparente Einbettung von Perl ithreads in die bestehenden Threads eines (in C geschriebenen) Programms. Damit ist die Einbettung von Perl in nahezu beliebige multithreaded Programme möglich. Der Ansatz hat jedoch auch Limitierungen, die sich aus der Natur der ithread-Implementierung ergeben (vgl. auch den Abschnitt "CAVEATS" in der *collectd-perl(5)* Manpage).

Die Initialisierung des Perl-Codes muss vor der Erzeugung weiterer (Perl-)Threads abgeschlossen sein, da sonst die unterschiedlichen Perl-Threads nicht den gleichen Basis-Interpreter zu Grunde liegen haben. Dies betrifft insbesondere die Deklaration von gemeinsam genutzten Variablen und das Laden von weiteren Modulen. Letzteres wird durch keinen Mechanismus zu späteren Zeitpunkten unterbunden. Sollte dieser Punkt nicht beachtet werden, dann unterscheidet sich der Parse-Tree der unterschiedlichen Interpreter und damit der zur Verfügung stehende Code.

Da jeder Perl-Thread in einem eigenen Interpreter läuft, dürfen keine "Tricks" ("Hack" wäre an dieser Stelle noch angebrachter) mit der Symboltabelle angewendet werden. Bei

```
static int
my_get(pTHX_ SV *variable, MAGIC *mg)
{
    char *pv = mg->mg_ptr;
    sv_setpv(variable, pv); /* setze den String-Wert der magischen Variablen */
    return 0;
}

static int
my_set(pTHX_ SV *variable, MAGIC *mg)
{
    char *pv = mg->mg_ptr;
    /* den String-Wert der magischen Variablen auslesen und in die C-Variable kopieren */
    sstrncpy (pv, SvPV_nolen(var), DATA_MAX_NAME_LEN);
    return 0;
}

/* "magic virtual table" für Zugriff auf Strings */
static MGVTBL pv_vtbl = { my_get, my_set, NULL, NULL, NULL, NULL, NULL, NULL };

    ...
    /* Perl-Variable '$var' mit Magie belegen (und ggf. erzeugen) */
    SV *var = get_sv("var", 1);
    sv_magicext(var, NULL, PERL_MAGIC_ext, &pv_vtbl, c_string, 0);
```

Listing 1

Verwendung der im vorletzten Abschnitt gezeigten Methode zur gemeinsamen Nutzung von Subroutinen in mehreren Threads muss der Name der betroffenen Subroutinen zur Compile-Zeit feststehen (genauer: vor Erzeugung eines weiteren Threads). Die Verwendung von Code-Stücken wie `register_callback("foo"); *foo = \&bar;` ist daher nicht möglich, da der Callback über seinen Namen aufgerufen wird, die Symboltabelle aber nicht mit den anderen Threads geteilt wird. Dies würde daher zu inkonsistentem Verhalten führen.

Renée Bäcker

$foo als ebook

Ende letzten Jahres habe ich mir ein Sony Ebook Reader gekauft, weil ich viel unterwegs bin und gerne lese. Da kam dann sehr schnell die Idee, auch das Perl-Magazin als ebook zu veröffentlichen. Das hörte sich nach einer spannenden Idee an. Natürlich sollte kein kommerzielles Produkt genutzt werden und natürlich wollten wir unsere Vorgehensweise nicht komplett ändern. Darum musste etwas Eigenes her.

Erstmal ganz grob unsere Vorgehensweise bei der Erstellung einer Ausgabe von "$foo": Wir suchen Autoren für die einzelnen Themen und bekommen deren Beiträge im Pod-Format. Warum Pod? Wir arbeiten (fast) alle mit Perl und da wir es ja gewohnt sind, im Programm fleißig zu dokumentieren, kennen wir Pod. Es ist ein einfaches Textformat und ist zur Not auch schnell zu erlernen.

Wir mussten es nur an einer Stelle "aufbohren": Pod kann nicht mit Bildern umgehen. Es gibt also keine Direktive um Bilder anzuzeigen. Von daher sagen wir den Autoren immer noch, wie sie Bilder einbauen können, ohne den Umweg über externe Formate innerhalb des Pods zu gehen.

Wenn wir die Artikel bekommen haben, landen diese in einem Wiki. Wir benutzen Kwiki, weil es ganz schlank ist und die wichtigsten Sachen kann. Wir benutzen ein Wiki, weil es mehrere Leute gibt, die an einem Artikel arbeiten. Anfangs hatten wir auch einige Freiwillige, die Korrektur gelesen haben. Leider sind einige wieder abgesprungen. Für richtige Wikis in Unternehmen oder im Internet würde ich Kwiki nicht unbedingt nehmen, da gibt es sicherlich bessere. Aber für unsere Zwecke ist das Wiki absolut ausreichend und es kann Pod rendern - nicht immer einwandfrei, aber egal.

Ein paar Leute lesen Korrektur bis zum Stichtag, der ca. 1 bis 1,5 Monate vor dem Erscheinungsdatum liegt. Dann werden die Artikel in Adobe InDesign gesetzt. Hier ist auch noch Verbesserungspotential vorhanden, aber das könnte dann mal Thema eines weiteren Artikels sein. Wenn die Artikel gesetzt sind, bekommen die Autoren den Artikel als PDF. Damit bekommt der Autor einen Eindruck wie der Artikel später im Magazin aussehen wird. Jetzt ist auch die letzte Chance für Korrekturen. Diese Korrekturen werden nur in InDesign eingepflegt.

Dann geht das Magazin ca. 3-4 Wochen vor dem Erscheinungsdatum in die Druckerei. Wenn die Hefte aus der Druckerei kommen, haben wir in der Regel drei bis vier Tage Zeit für Eintüten, Etikettieren, Frankieren und zur Post bringen.

Das ebook kommt hinzu...

Jetzt kommt also noch das Produzieren des ebooks hinzu. Wir haben uns dazu entschieden, das Magazin im epub-Format anzubieten, da das von den meisten Readern gelesen werden kann. Aber was ist das epub-Format überhaupt?

epub steht für "electronic publication" und ist ein offener Standard. Viele Verlage nutzen auch die Möglichkeit Digital Rights Management Mechanismen zu implementieren, die bei epub erlaubt sind. Der große Vorteil von epub gegenüber ebooks im PDF-Format ist die dynamische Anpassung des Textes an die Bildschirmgröße, was man gerade bei kleineren Geräten sehr zu schätzen lernt.

Im Grunde sind epub-Dateien einfach gepackte Inhalte, die man auch mit Programmen wie 7zip entpacken kann. Damit kann man sich einen Eindruck von den Inhalten machen. Welche Dateien enthalten sein müssen und welche Struktur eingehalten werden muss, ist im OEBPS Container Format beschrieben.

So sieht eine Struktur aus:

```
/
|- mimetype
+ META-INF
|    |- container.xml
+ OPS
   |- toc.ncx
   |- content.opf
   |- <inhalt.xhtml>
```

In der mimetype-Datei ist nur der Inhalt "application/epub+zip" zu finden, und teilt den Endgeräten mit, dass es sich um ein epub handelt.

Da nicht genau festgelegt ist, wie die Stammdatei (im Beispiel `content.opf` heißt), muss der Name dem Lesegerät mitgeteilt werden. Das wird mit der Datei `container.xml` im Ordner `META-INF` erledigt.

In dieser Stammdatei ist das epub genau beschrieben, ein paar Metainformationen, enthaltene Dateien und die Reihenfolge des Inhalts. Zu den Metainformationen gehören die Sprache, der Autor, der Titel, Bilder und bei meinem Programm auch die verwendeten Perl-Module. Ein Ausschnitt aus diesen Metainformationen ist in Listing 1 zu finden:

Die Struktur ist im Open Packaging Format (daher auch die Endung .opf) beschrieben. Im Abschnitt `manifest` werden alle Dateien aufgelistet. Ich mache es so, dass es nicht nur eine `inhalt.xhtml` gibt, sondern für jeden Artikel gibt es ein eigenes Kapitel. Das ermöglicht es mir, auch einzelne Kapitel nachträglich zu ändern oder für eigene epubs herauszuziehen - siehe Listing 2.

Jede Datei bekommt hier eine eigene ID, die im Minimal-Inhaltsverzeichnis eine Rolle spielt. Dieses Minimalinhaltsverzeichnis ist im `spine`-Abschnitt zu finden. Dort wird über das Attribut `toc` (`<spine toc="ncx">`) das richtige Inhaltsverzeichnis festgelegt.

```
<spine toc="ncx">
  <itemref idref="ch1" linear="no" />
  <itemref idref="ch2" linear="no" />
</spine>
```

Das Inhaltsverzeichnis ist im NCX-Format und ist ausführlicher als das Inhaltsverzeichnis in der Stammdatei. Hier sind die neben der Reihenfolge der Inhalte auch die Unterpunkte der Kapitel zu finden - mit Sprungmarken, damit man aus dem Inhaltsverzeichnis direkt zum gewünschten Inhalt gehen kann.

```
<navMap>
  <navPoint id="navPoint-1" playOrder="1">
    <navLabel>
      <text>Vorwort</text>
    </navLabel>
    <content src="text/1.xhtml#Vorwort" />
    <navPoint id="navPoint-2" playOrder="2">
      <navLabel>
        <text>Kamel</text>
      </navLabel>
      <content src="text/1.xhtml#Kamel" />
    </navPoint>
    <navPoint id="navPoint-3" playOrder="3">
      <navLabel>
        <text>Links</text>
      </navLabel>
      <content src="text/1.xhtml#Links" />
    </navPoint>
```

Die einzelnen Inhaltsdateien sind im `XHTML`-Format und man kann auch CSS-Dateien zum stylen nehmen. Die einzelnen Seiten sind also nichts Besonderes.

```
<dc:title>$foo - Perl-Magazin "Frühjahr 2011"</dc:title>
<dc:creator opf:role="aut">Renee Baecker</dc:creator>
<dc:language>en</dc:language>
<meta name="cover" content="img1" />
<meta name="EPublisher version" content="0.3" />
<meta name="EBook::EPUB version" content="0.5" />
<meta name="image1" content="img2" />
<meta name="image1" content="img3" />
<dc:identifier id="BookId">urn:uuid:89339E83-9A6F-3DEF-8642-3F105D729DD4</dc:identifier>
```

Listing 1

```
<manifest>
  <item id="ncx" href="toc.ncx" media-type="application/x-dtbncx+xml" />
  <item id="img1" href="images/CoverAusgabe17.jpg" media-type="image/jpeg" />
  <item id="ch1" href="text/cover.xhtml" media-type="application/xhtml+xml" />
</manifest>
```

Listing 2

EPublisher

Um das Ganze nicht immer wieder händisch machen zu müssen, habe ich angefangen den "EPublisher" zu programmieren. Mit diesem Programm soll der Aufwand für die Umwandlung der Ausgaben in ein ebook minimiert werden. Außerdem soll das Programm so flexibel sein, dass es nicht nur für das Perl-Magazin sondern auch für andere Dinge verwendet werden kann. Auch sollen unterschiedliche Quell- und Zielformate unterstützt werden - aber das ist noch Zukunftsmusik.

Zur Konfiguration wird YAML mit dem Modul `YAML::Tiny` verwendet. Damit ist eine einfache Konfiguration, die leicht kopiert und verändert werden kann, möglich. In dieser Konfiguration legt man die Formate und die Ausgangsdateien fest.

```
---
Test:
  source:
    -
      type: Dir
      path: D:/Foo/Ausgabe20/
  target:
    type: EPub
    author: Renee Baecker
    output: ./foo_20.epub
    title: "$foo Nr 20 - Winter 2011"
    encoding: iso-8859-1
```

Das Projekt "Test" hat als Quelle ein Verzeichnis. Hier werden dann alle Dateien mit dem Suffix `.pod` bzw `.pm` genommen und für das ebook herangezogen. Hierbei ist dann darauf zu achten, das die Dateien so benannt sind, dass sie in der endgültigen Kapitelreihenfolge zurückgeliefert werden.

Jede Datei wird dabei zu einem eigenen Kapitel.

Weitere Quellen sehen so aus:

```
source:
  -
    type: File
    path: D:/Foo/Ausgabe20/test.pod
  -
    type: Module
    name: CGI::Application
```

Damit kann man dann beliebige Zusammensetzungen des ebooks machen. Eine Quelle vom Typ "File" ist eben ein einzige Datei und ein "Module" sucht im `@INC` nach dem Modul und holt sich daraus die Dokumentation. Damit ist es dann auch leicht möglich, ebook-Handbücher aus größeren Distributionen zu machen.

Als Ziel gibt es bisher nur EPub. Hierfür können noch einige Parameter übergeben werden - wie den Pfad zur Ausgabedatei oder das Encoding, in dem die Quelldateien vorliegen.

Der EPublisher liest diese YAML-Datei aus und jede Quelle muss eine Liste an Pod-Strings liefern, die die einzelnen Kapitel darstellen. Die bisherigen Quellplugins verwenden dazu `PPI`, mit dem man sehr leicht die Pod-Dokumentation aus den Dateien bekommt:

```
sub extract_pod_from_code {
  my ($code) = @_;

  return if !$code;

  my $parser    = PPI::Document->new(
      \$code,
  );
  my $pod_nodes = $parser->find(
      sub {
          $_[1]->isa( 'PPI::Token::Pod' );
      },
  );

  my $merged = PPI::Token::Pod->merge(
      @{$pod_nodes || []},
  );

  return '' if !$merged;
  return $merged->content;
}
```

Wurden die ganzen Quellen eingelesen, bekommt das Ziel-Plugin alle Pods übergeben und das Plugin ist für die korrekte Verarbeitung zuständig.

Für die Umwandlung von Pod-Dateien in ein ebook gibt es - wie schon fast zu erwarten war - Module auf CPAN, die den Vorgang unterstützen: `Pod::Simple::XHTML` und `EBook::EPUB`. `Pod::Simple::XHTML` ist für die Umwandlung von Pod nach XHTML zuständig und `EBook::EPUB` für die ebook-Infrastruktur.

Als erstes werden Metainformationen zu dem ebook hinzugefügt:

```
# Create EPUB object
my $epub = EBook::EPUB->new();

# Set the ePub metadata.
$epub->add_title( $title );
$epub->add_author( $author );
$epub->add_language( $language );
```

Desweiteren wird ein Cover hinzugefügt, wenn es gewünscht ist:

```
my $cover_id = $epub->copy_image(
    $cover_filename,
    "images/$basename"
);
$epub->add_meta_item( 'cover', $cover_id );
```

Das `copy_image` ist notwendig, damit die Grafik später auch in der epub eingepackt wird. Damit ist es aber noch nicht als Cover wirklich sichtbar. Wir halten uns an die typische Struktur aus dem Web - alle Grafiken landen in einem Verzeichnis "images".

Danach wird das Cover in den Metadaten eingefügt. In der Datei `content.opf` ist dann so eine Zeile zu finden:

```
<meta name="cover" content="img1" />
```

Das reicht aber immer noch nicht für die Anzeige des Covers. Auch für das Cover muss eine XHTML-Seite angelegt werden. In der steht - neben dem notwendigen XHTML-Rumpf nur

```
<img src="../images/$basename" alt="" />
```

Den Rest der XHTML-Datei spare ich mir hier, da reicht ein Blick in die epub-Datei.

Es folgen noch weitere Metadaten, unter anderem die UUID, für die das Modul `Data::UUID` genommen wird. Da `EBook::EPUB` eine Warnung ausgibt wenn man die UUID überschreibt, wird ein lokaler Signalhandler installiert. Die Warnung ist sonst mit der Zeit nervig.

Weiterhin wird noch ein Standard-Stylesheet kopiert, für den Fall, dass kein eigenes Stylesheet angegeben wurde. Danach kommt dann der interessante Teil - die Umwandlung von Pod nach XHTML und die Einbindung der einzelnen Kapitel in das ebook (Listing 3).

```
for my $pod ( @{$pods} ) {
    my $parser = Pod::Simple::XHTML->new;
    $parser->index(0);

    $parser->accept_directive_as_processed( 'image' );

    my ($in_fh_temp,$in_file_temp) = tempfile();
    binmode $in_fh_temp, ":encoding($encoding)";
    print $in_fh_temp $pod;
    close $in_fh_temp;

    my $in_fh;
    open $in_fh, "<:encoding($encoding)", $in_file_temp;

    my ($xhtml_fh, $xhtml_filename) = tempfile();

    $parser->output_fh( $xhtml_fh );
    $parser->parse_file( $in_fh );

    close $xhtml_fh;
    close $in_fh;

    $epub->copy_xhtml( $xhtml_filename, "text/$counter.xhtml", linear => 'no' );

    # cleaning up...
    unlink $xhtml_filename;
    unlink $in_file_temp;

    $self->add_to_table_of_contents( $counter, $parser->{to_index} );

    # add images
    my @images = $parser->images_to_import();
    for my $image ( @images ) {
        my $path     = $image->{path};
        my $name     = $image->{name};
        my $image_id = $epub->copy_image( $path, "images/$name" );
        $epub->add_meta_item( "image$image_counter", $image_id );
    }

    $counter++;
}
```

Listing 3

Nachdem das Parserobjekt erzeugt wurde, werden Einstellungen vorgenommen: Zuerst wird das "Inhaltsverzeichnis" für die Seiten ausgeschaltet. Dieses "Inhaltsverzeichnis" kennt man von den Dokumentationen auf CPAN, bei denen über dem eigentlichen Text die Links zu den ganzen Unterpunkten zu finden sind. Dies ist hier nicht erwünscht.

Wie weiter oben bereits erwähnt, wurde das Pod so erweitert, dass mittels neuer Pod-Direktive auch Bilder eingebunden werden können. Die Direktive ist sehr einfach aufgebaut:

```
=image Pfad/zu/Bild.png Abb x: Text
```

Damit der Parser weiß, dass diese neue Direktive zum "gültigen" Pod gehört und dass es geparst werden muss, wird dem Parser die Direktive "image" bekannt gemacht:

```
$parser->accept_directive_as_processed(
    'image'
);
```

Das Verarbeiten dieser Direktive muss auch noch implementiert werden.`Pod::Simple::XHTML` kennt zwei "Stellen", die man abgreifen kann: `start_<Direktive>` und `end_<Direktive>`. Da wir den Text zu der Direktive (den Pfad und die Bildunterschrift) benötigen und keine Vorbereitungen brauchen, müssen wir nur die Subroutine `end_image` implementieren (Listing 4).

Die Subroutine baut den XHTML-String für das Einbinden des Bildes zusammen. Wer eigene Direktiven einbinden will, muss sich mit den Innereien des Parsers beschäftigen und auch etwas ausprobieren. Die Dokumentation von `Pod::Simple` ist dahingehend etwas dünn. Woher soll man wissen, dass der Ergebnisstring in `$self->{scratch}` landen muss? Und dass man diesen String mit `emit` freigeben muss?

Ein paar weitere Methoden des Parsers müssen angepasst werden, damit der Verbatim-Teil des Pod als `<div>` mit der entsprechenden CSS-Klasse ausgegeben wird und dass die Dokumente den richtigen Header haben.

```
sub Pod::Simple::XHTML::end_image {
    my ($self) = @_;

    my %regexe = (
        # =image "C:\path with\whitespace.png" alt text
        path_quoted => qr/"([^"]+)"(?:\s+(.*))?/s,
        # =image C:\path\img.png alt text
        path_plain  => qr/([^\s]+)(?:\s+(.*))?/s,
    );

    my $text  = $self->{scratch};
    my $regex = $text =~ /^\s*"/ ? $regexe{path_quoted} : $regexe{path_plain};

    my ($path,$alt) = $text =~ $regex;
    $alt = '' if !defined $alt;

    return if !$path;

    if ( !-e $path ) {
        warn "Image $path does not exist!";
        return;
    }

    my $filename     = basename $path;

    # save complete path in $self->{images_to_import}
    push @{$self->{images_to_import}}, { path => $path, name => $filename };

    $self->{scratch} = qq~<p><img src="../images/$filename" alt="$alt" /></p>~;

    $self->emit;
};
```

Listing 4

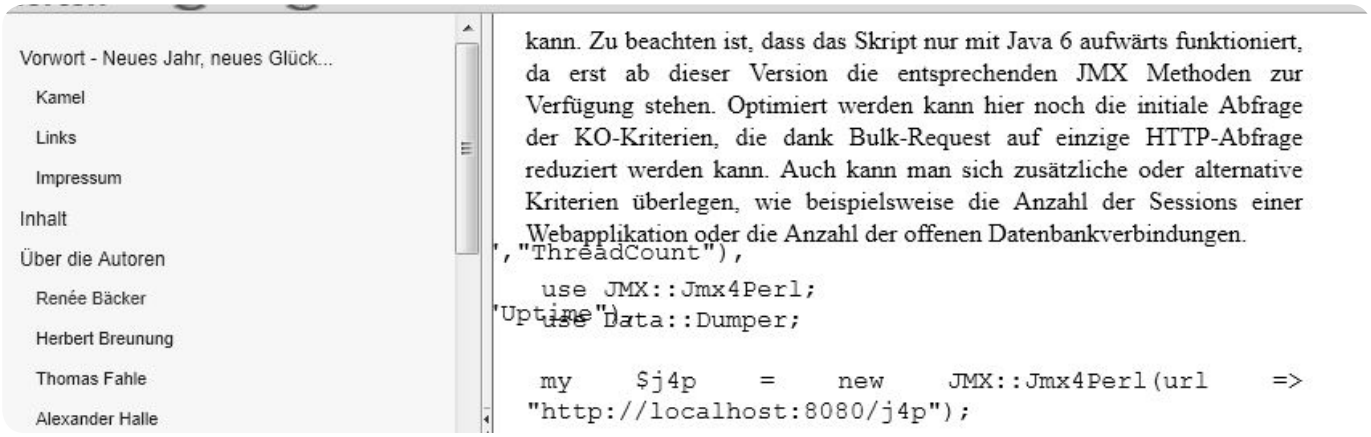


Abbildung 1: Darstellung im Firefox mit "EPUBReader"

Probleme bei der ebook-Erzeugung

Neben den dedizierten Ebook-Readern wie Sonys Reader PRS-350, kann man die Ebooks auch auf vielen anderen Geräten lesen - unter anderem auch mit Plugins direkt im Firefox. Dadurch ergeben sich unterschiedliche Probleme - wie es häufig bei Browsern der Fall ist: Nicht jeder Reader unterstützt alles aus dem CSS bzw. die Anweisungen werden unterschiedlich interpretiert. Dadurch kommt es zu unschönen Darstellungsunterschieden.

In Abbildung 1 ist die Darstellung mit dem Firefox-Plugin zu sehen während in Abbildung 2 die Darstellung der gleichen Seite im Sony Reader zu sehen ist.

Ein Beispiel ist auch das Inhaltsverzeichnis: Während der Sony Reader sehr pflegeleicht bei der Darstellung von Umlauten ist, kommen andere Reader damit weniger zurecht - weil sie strikter in der Umsetzung der Anweisungen sind.

Als sehr nützlich hat es sich erwiesen, dass während der Entwicklung zum Einen unterschiedliche Desktop-Programme und die Firefox-Plugins zum Testen herangenommen wurden und dass zum anderen viele Leser sich bereiterklärt haben, das epub Probe zu lesen. Dadurch wurden viele der oben dargestellten Probleme während der Entwicklung behoben.

Leider ist aber nicht ausgeschlossen, dass es doch noch Probleme gibt.

Das fertige Produkt

Mit dem epublisher bekomme ich recht schnell - wenn die Artikel dann letztendlich im Pod-Format vorliegen - das fertige Produkt. Bei der Umsetzung hat sich auch wieder gezeigt,

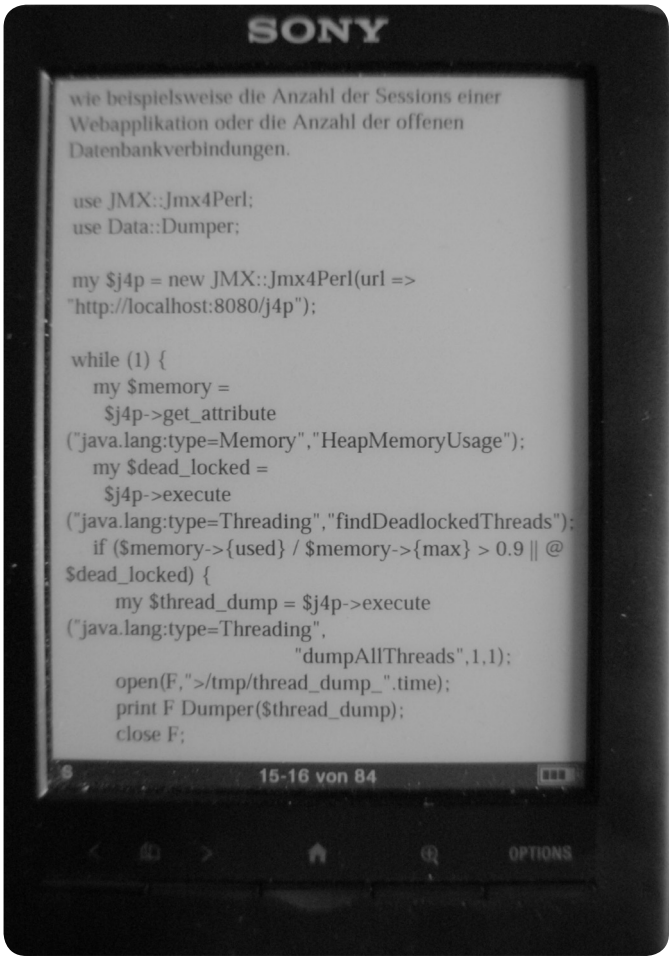


Abbildung 2: Darstellung der gleichen Seite im Sony Reader

dass man auf CPAN für die meisten Aufgaben schon fertige Module bekommen.

Fazit

Insgesamt ist das Produzieren der ebook-Version von "$foo" ein zusätzlicher Aufwand von zwei Tagen pro Ausgabe. Es gibt noch einige Stellen, an denen es Verbesserungspotential gibt, z.B. das Exportieren der Artikel aus InDesign in das Textformat. Das wird thematisiert, wenn der Leidensdruck zu groß wird.

Zusätzlich soll der epublisher dahingehend ausgebaut werden, dass LaTeX-Quellen in das epub-Format umgewandelt werden können sollen. Damit ließen sich ganz schnell wissenschaftliche Texte, die ja sehr häufig in LaTeX gesetzt sind, umwandeln.

Auch wenn viele - oder vielleicht auch alle - ebook-Reader mit PDF umgehen können, wurde die epub-Version eingeführt, weil das neue Format Vorteile z.B. bei der Skalierung bringt.

Werden Sie selbst zum Autor...
... wir freuen uns über Ihren Beitrag!

info@foo-magazin.de

Renée Bäcker

Perl für Vortragende

Der 13. Deutsche Perl-Workshop ist gerade vorbei. An drei Tagen wurden Vorträge gehalten und viel diskutiert. So wie auf dem Deutschen Perl-Workshop gibt es noch viele andere Gelegenheiten, bei denen man Vorträge halten kann. In den meisten Fällen sind es "statische" Vorträge, das meine ich die Vorträge, bei denen es nur Folien gibt.

Diese Vorträge sind auch in den meisten Fällen vollkommen in Ordnung. Oft ist es aber für den Zuschauer interessanter, wenn noch etwas gezeigt wird - ein Stück Code, eine Vorführung einer Anwendung oder etwas Ähnliches.

So eine Dynamik in seine Vorträge zu bekommen, kann aber für den Vortragenden "gefährlich" sein. Gerade ungeübte Redner sind häufig sehr nervös und machen dadurch Fehler. Wenn sich die Fehler häufen, assoziieren die Zuhörer diese Fehler mit dem Code/der Anwendung. Oder als Vortragender vergisst man, was man zeigen wollte. Hinterher ärgert man sich, dass die wichtigsten Features der Anwendung nicht gezeigt wurden.

Aber es liegt ja nicht immer am Vortragenden selbst wenn bei solchen Live-Demos etwas schief geht. Auf der YAPC::Europe 2007 habe ich einen Lightning Talk gehalten, der voll automatisiert war. Ich wollte zeigen, wie man mit Perl automatisiert Vorträge erstellen kann und auch Online-Spiele erfolgreich meistern kann. Kurz vor der Session kam ein anderer Teilnehmer zu mir, um seinen Lightning Talk mit meinem Laptop zu präsentieren. Dazu fehlte ein Plugin im Firefox, also noch schnell installiert.

Zum Glück habe ich dann meinen Talk nochmal schnell zur Probe laufen lassen und dann der GAU - das Online-Spielen funktionierte nicht mehr. Das Plugin hat eine neue Toolbar eingeblendet, so dass die Positionen von Elementen des Spiels an einer anderen Position zu finden waren. Man sieht, dass durch kleine Änderungen am System, der ganze Vortrag scheitern kann.

Wenn etwas aus der laufenden Entwicklung gezeigt werden soll, dann kann eine kurze Änderung am Code, der fehlerhaft z.B. im SVN eingecheckt wurde, vieles kaputtmachen und die Live-Demo platzen lassen.

Probleme vermeiden

Die Probleme bei solchen Vorträgen lassen sich zum Teil ganz einfach vermeiden: Kurz vor dem Vortrag sollten keine Änderungen mehr am System vorgenommen werden, bei Vorführungen von aktuellen Entwicklungen, sollte man sich merken, bei welcher Version alles funktioniert hat.

Man kann sich aufschreiben was man machen will - was aber ganz klar nicht so "cool" aussieht - und das oberste Gebot ist natürlich "üben, üben, üben". Dazu eignen sich Perlmonger-Treffen, die eigenen Kollegen oder auch ein kleinerer Workshop wie den Frankfurter Perl-Community Workshop.

Eine Person, die extrem gut bei Vorträgen ist, ist Damian Conway. Er kennt seine Folien fast auswendig, er ist witzig und hat extrem gute Ideen. Auch wenn er selbst immer wieder sagt, dass man Live-Demos nicht machen sollte, zeigt er immer wieder Code.

Mit einem Modul von Damian Conway kann man solche Live-Demos mit Code auch Vortragssicher machen: `IO::Prompt`

IO::Prompt to the rescue

Es ist schon toll, wenn man live auf der Bühne programmiert und es funktioniert alles. Nur wenn man es wirklich live macht, vertippt man sich wahrscheinlich. Deshalb kann man so tun als würde man live programmieren, dabei ist alles schon vorbereitet.

Das Programm an sich ist ganz simpel:

```
use IO::Prompt;

system( 'clear' );
print 'perl@ubuntu:~/foo$ ';

while (prompt '') {
}

__DATA__
__PROMPT__
```

Das `system()` wird nur benutzt um den Programmaufruf zu verstecken. Und danach wird ein Prompt ausgegeben. So merkt der Zuhörer nicht, dass alles nur ein Fake ist. Das Entscheidende ist dann die `while`-Schleife. `prompt` ist eine Funktion von `IO::Prompt`. Dieser Funktion kann man mehrere Parameter übergeben, die praktisch für den eigentlichen Einsatz des Moduls ist - Benutzerinteraktion. Für die Fake-Demo muss ein leerer Prompt übergeben werden.

Im `__DATA__`-Bereich gibt es dann wieder einen extra Bereich, den `IO::Prompt` als Quelle für die Ausgaben nutzt. Dieser wird mit `__PROMPT__` gekennzeichnet. Ein komplettes Beispiel sieht somit folgendermaßen aus:

```
use IO::Prompt;

system( 'clear' );
print 'perl@ubuntu:~/foo$ ';

while (prompt '') {
}

__DATA__
__PROMPT__
cat > test.pl
#!/usr/bin/perl

use strict;
use warnings;
print "Hallo Welt!";
```

Wenn das Programm gestartet wird, wird mit jedem Tastaturanschlag ein Zeichen aus dem Prompt-Bereich ausgegeben. Nur am Zeilenende muss man zweimal die Return-Taste drücken, damit es weiter geht. Aber Vorsicht! Wenn man mitten in der Zeile (also z.B. nach "use") die Return-Taste drückt, wird der Rest der Zeile ganz schnell ausgegeben. Da könnte es passieren, dass der Fake auffällt.

Das Programm aus dem Prompt-Bereich sollte man schon als Skript gespeichert haben (hier: test.pl), da `IO::Prompt` nur die Ausgabe macht und nicht den Befehl ausführt.

Nachdem die letzten Zeichen aus dem Fake-Programm ausgegeben wurden, kann man mit Ctrl+D das Programm beenden und man ist wieder ganz normal in der Shell unterwegs. Jetzt kann man test.pl ausführen und keiner merkt, dass das Programmieren der Anwendung schon vorher erledigt wurde.

Aber auch wenn man jetzt die Vertipper nicht im Programm haben kann, sollte man den Vortrag üben, damit man keine Überraschungen erlebt was denn als nächstes kommt.

Leider kann man das Modul nicht mit ActivePerl oder StrawberryPerl nutzen, da es auf `IO::Tty` aufbaut. Wer es unter Windows dennoch nutzen will, muss Cygwin nehmen. Unter Linux funktioniert das Modul einwandfrei.

GUIs automatisiert bedienen

Aber nicht jede Live-Demo beinhaltet Programmierung; viel häufiger werden es Anwendungen mit GUIs sein. Diese Anwendungen automatisiert zu bedienen ist an sich kein Problem - auch hier liefert CPAN alle notwendigen Tools. Für Windows gibt es `Win32::GuiTest` und für X11 gibt es `X11::GUITest`. Für letzteres gibt es sogar ein Modul, mit dem man die Interaktionen aufnehmen kann um sie anschließend wieder abspielen zu können. Das Modul `X11::GUITest::record` wurde in der Ausgabe "Sommer 2009" von `$foo` vorgestellt.

Die entstehenden Programme sind aber nicht besonders hübsch. Ein Beispiel:

```
#!/usr/bin/perl

use strict;
use warnings;

use X11::GUITest qw/
  StartApp
  WaitWindowViewable
  SendKeys
/;

# Start application
StartApp('padre');

# Wait for application window to come up
# and become viewable.
my ($GEditWinId) =
    WaitWindowViewable('Padre');
if (!$GEditWinId) {
  die("Couldn't find window in time!");
 }

sleep 11;

# session <ctrl+alt+o>

SendKeys( "^(%(o))" );
sleep 3;
SendKeys( "\n" );
```

Ich habe hier einige `sleep`s eingebaut, weil das Programm in einer VM läuft, die nicht gerade die schnellste ist. Damit aber immer die notwendigen Informationen da sind, ist ein Warten unabdingbar.

Nachdem die Zielanwendung (hier: padre) gestartet ist, kann mit der automatischen Bedienung angefangen werden. In diesem Beispielprogramm werden nur ein paar Tastatureingaben simuliert. `SendKeys` wird dabei ein String übergeben, der die Tastatureingaben darstellt. Hier muss man sich erst an die Darstellungsweise gewöhnen: `^(...)` ist die Ctrl-Taste, `%(...)` die Alt-Taste und für weitere "Spezialtasten" wie die Pfeiltasten gibt es zusätzliche Syntax. In dem String können auch ganze Abfolgen von Tasten enthalten sein:

```
^(g){PAUSE 1000}5\n{END}{LEF 13}
```

Das sendet Ctrl+g, macht dann eine Sekunde Pause, sendet dann eine 5 und einen Zeilenumbruch. Schickt die "Ende"-Taste und geht dann 13 Positionen nach links.

Neben dem Senden von Tastatureingaben kann man auch die Maus steuern. Dazu gibt es die Funktionen `MoveMouseAbs` und `ClickMouseButton`. Das Problem bei der Verwendung der Mausfunktionen ist, dass die Position nur absolut angegeben werden kann. Das heißt, immer wenn an der GUI etwas geändert wird oder sich am System etwas ändert, müssen die Skripte überprüft werden.

Auf jeden Fall muss die Automatisierung von GUIs gut geplant werden.

Zu den offenen Punkten gehört sowohl bei `IO::Prompt` als auch `X11::GUITest`, wie man es am geschicktesten umsetzt, dass man wirklich noch auf Rückfragen aus der Zuhörerschaft eingehen kann. Gerade bei der Automatisierung von GUIs mit `X11::GUITest` kommt man schwer aus dem ursprünglich geplanten Ablauf raus, um evtl. nochmal auf eine andere Einstellung zurückzugehen.

Ich selbst habe es bisher so gelöst, dass ich Fragen erst am Ende des Vortrags akzeptiert habe. Es wäre natürlich schöner, wenn man den Ablauf unterbrechen kann und dann einfach wieder am Ausstiegspunkt einsteigen kann.

Diese zwei Beispiele zeigen, dass man durchaus "Live-Demos" in Vorträgen zeigen kann und man einige Unsicherheiten ausschließen kann.

Tom Kirchner

2D Spieleprogrammierung mit OpenGL - Teil 2

Diese Artikelreihe gibt eine kleine Einführung in die Spieleprogrammierung mit OpenGL und GLUT. Dabei wird das Spiel PONG programmiert und in mehreren Ausbaustufen verschönert.

Animation des Balls

Aus Teil 1 dieser Artikelreihe wissen wir, dass es so etwas wie ein "Objekt", etwa ein Cubus oder eine gekrümmte Oberfläche in OpenGL nicht gibt. Es gibt nur Primitive, also Dreiecke und Linien. Weiterhin wurde festgestellt, dass die Informationen, also die Vektoren, die diese Primitive definieren, sofort von OpenGL vergessen werden, kurz nachdem sie ihre Spuren im Framebuffer hinterlassen haben.

Das heißt, alle Informationen, z.B. Position, Rotation etc. und welche Primitive zu einem konkreten Objekt gehören, müssen vom Programmierer verwaltet werden. Um nun ein Objekt zu animieren, verändert man einfach diese Objekteigenschaften über die Zeit und zeichnet die Objekte für jedes Frame. Macht man dies schnell genug, üblicherweise min. 30 mal pro Sekunde, so entsteht eine Animation.

In unserem Programm "PONG" ist z.B. der Ball ein animiertes Objekt. Da er nur seine Position verändert, ist dies auch die erste Information des Objektes die man verwalten und über die Zeit verändern muss, um den Ball zu animieren. An dieser Stelle sollte man sich Gedanken darüber machen, welche Informationen man noch benötigt, um ausgehend von der aktuellen Ballposition dessen neue Position zu berechnen. Bei unserem Spiel bewegt sich der Ball auf einer geraden Linie mit gleich bleibender Geschwindigkeit, zumindest im einfachsten Fall. In der Mechanik heißt das "gleichförmige Bewegung" und die Formel zur Berechnung der neuen Position s aus der vorherigen Position so lautet: `s = v * t + s0` mit dem Geschwindigkeitsvektor v und der verstrichenen Zeit t. v ist also die zweite Information, die wir abspeichern müssen - t theoretisch auch, aber dafür findet sich gleich eine elegantere Lösung, sodass wir t abrufen können, sobald wir es brauchen.

Anmerkung: v ist nur auf den ersten Blick konstant, denn eigentlich handelt es sich bei v um einen Vektor, dessen Betrag (Länge) zwar konstant ist, dessen Richtung sich aber ändert, z.B. wenn der Ball an einem anderen Objekt abprallt.

Nun zur Implementierung: Es wurde bereits eine Funktion geschrieben, die den Ball an einem festen Ort zeichnet. Wir definieren jetzt eine globale Variable, die die Position des Balls speichert, und nehmen dafür eine zweidimensionale Arrayreferenz. Die `display()` Funktion benutzt nun beim Zeichnen die Position aus dieser globalen Variable.

Wie verändern wir nun die Position nach einer bestimmten Zeit und zeichnen die Szene neu? GLUT bietet uns dafür die Funktion `glutTimerFunc()`, die eine Funktion setzt, die nach n Millisekunden aufgerufen werden soll. `glutTimerFunc()` übergeben wir nun die Referenz auf unsere `timer()` Funktion - siehe Listing 1.

Um die `timer()` Funktion wiederholt aufzurufen, müssen wir in der `timer()` Funktion erneut `glutTimerFunc()` aufrufen, denn `glutTimerFunc()` ruft die an sie übergebene Funktion wirklich nur einmal auf.

`glutTimerFunc()` hat eine interessante Eigenschaft: Die n Millisekunden, nach denen die registrierte Funktion aufgerufen wird, sind eher ein Wunsch, d.h. die Realität kann stark davon abweichen.Um nun die wirklich verstrichene Zeitdifferenz in der `timer()` Funktion zu erhalten, um sie

```
my $TimerCallsPerSecond = 30;
my $BallPosition = [250,250];
my $BallVelocity = [0.1,0.1];

sub main {
  # [...]
  glutTimerFunc(int(1000 / $TimerCallsPerSecond), \&timer, 0);
  # [...]
  glutMainLoop();
}

sub timer {
  my ($lastCallTime) = @_;
  my $thisCallTime = glutGet(GLUT_ELAPSED_TIME);
  my $delta_t = $thisCallTime - $lastCallTime;
  $BallPosition = [
    $BallVelocity->[0] * $delta_t + $BallPosition->[0],
    $BallVelocity->[1] * $delta_t + $BallPosition->[1],
  ];
  glutTimerFunc(int(1000 / $TimerCallsPerSecond), \&timer, $thisCallTime);
  glutPostRedisplay();
}

sub draw {
  # Ball
  glPushMatrix();
    glTranslatef($BallPosition->[0], $BallPosition->[1], 0);
    glScalef(10, 10, 1);
    ball();
  glPopMatrix();
}
```

Listing 1

in der Formel zur Positionsberechnung `s = v * t + s0` zu verwenden, wenden wir einen kleinen Trick an: `glutTimerFunc()` erhält drei Parameter: die Anzahl Millisekunden, nach denen der Callback wunschgemäß ausgeführt werden soll, die Funktionsreferenz auf den Callback und einen dritten Parameter, der eine Ganzzahl ist und beliebig sein kann. Dieser dritter Parameter wird an die `timer()` Funktion übergeben. Wir benutzen ihn dazu, um die Zeit des letzten Aufrufs der `timer()` Funktion zu übergeben, um damit dann t zu berechnen. Zur Berechnung der Zeitdifferenz verwenden wir nun noch die Variable `GLUT_ELAPSED_TIME`, die die Anzahl Millisekunden seit Programmstart enthält und mit der Funktion `glutGet()` abgerufen wird.

Bewegung der Spieler

Anders als der Ball, bewegen sich die Spieler nicht aufgrund mechanischer Gesetze, sondern gesteuert durch Eingabegeräte, z.B. indem eine Taste auf der Tastatur gedrückt wird. Die Funktion `glutSpecialFunc()` registriert eine Funktion, die ausgeführt wird, sobald eine "spezielle", d.h. nicht alphanumerische, Taste gedrückt wird. Der Keycode der Taste wird an die aufgerufene Funktion übergeben. Speichert man die aktuelle Position, die aktuelle Höhe jedes Spielers, als globale Variable, kann man diese in dieser Callback Funktion verändern. Sinnvoll ist es bei PONG auf die "Pfeil nach oben" Taste (Keycode 101) zu reagieren und die Höhe eines Spielers um einen Fixbetrag zu erhöhen. Analog dazu die "Pfeil nach unten" Taste (Keycode 103) - siehe Listing 2.

Physik: Kollisionsberechnung

Wir haben bereits eine einfache Kollisionserkennung für die Spieler eingebaut, nämlich mit der `clip()` Funktion, die die Bewegungsfreiheit der Spieler auf eine bestimmte Strecke beschränkt. Für den Ball benötigen wir allerdings eine etwas ausgefeiltere Kollisionserkennung, da der Ball sich kreuz und quer über das Spielfeld bewegt und an beweglichen Objekten, den Spielern, und den Wänden abprallen soll.

Um die Kollision festzustellen, ist das Konzept der Boundingboxen nützlich. Eine Boundingbox ist das geometrische Gebilde eines Objektes, dessen Grenzen mit denen anderer Gebilde kollidieren, in diese also nicht eindringen. Bounding-

```
my $Player1Y = 100;
my $Player2Y = 300;

sub main {
  # [...]
  glutSpecialFunc(\&keyboard);
  # [...]
  glutMainLoop();
}

sub keyboard {
  my ($key, $x, $y) = @_;
  if ($key == 101) {
    $Player1Y =
       clip($Player1Y + 10, 20, 430);
  }
  elsif ($key == 103) {
    $Player1Y =
       clip($Player1Y - 10, 20, 430);
  }
}

sub draw {
  # [...]
  # Spieler 1
  glPushMatrix();
    glTranslatef(20, $Player1Y, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();

  # Spieler 2
  glPushMatrix();
    glTranslatef(460, $Player2Y, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();
}
```

Listing 2

boxen sind oft vereinfachte Formen, der dem eigentlichen, oft komplexeren Objekt, möglichst nahe kommen. Mit diesen einfachen Geometrien lassen sich Kollisionsberechnungen schneller durchführen. Da wir allerdings in PONG nur Kreise (Ball) und Rechtecke (Spieler) haben, können wir direkt deren Form als Boundingboxen verwenden.

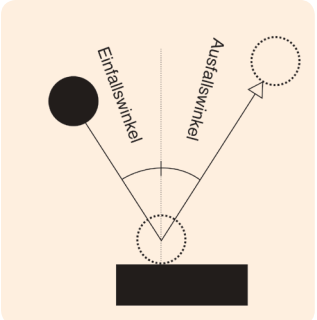


Abbildung 1: Einfallswinkel gleich Ausfallswinkel

Wir müssen in unserem Spiel also Kollisionen von einem Kreis mit einem Rechteck und einer Linie (Wand) erkennen und korrekt darauf reagieren. Da es sich bei den Wänden und den Spielern um starre Objekte, so genannte "Rigid Bodies" handelt, besteht die Reaktion auf die Kollision darin, dass der Ball abprallt, und zwar gemäß dem Gesetz "Einfallswinkel am Lot auf der Abpralloberfläche gleich Ausfallswinkel" (Abbildung 1).

Damit die `moveBall()` Funktion nicht unnötig komplex wird, ist die Kollisionsberechnung etwas approximiert: Der Ball prallt direkt von der Stelle aus ab, an der er sich *vor* der Kollision befindet. Dieser Ort ist u.U. sichtbar ein kleines Stück vom Spieler (oder Spielfeldrand) entfernt, was aber bei großen Geschwindigkeiten nicht auffällt.

Um eine realistischere Kollision zu berechnen, kann man folgendermaßen vorgehen: Ausgehend von der Position des Balls vor der Kollision in die Bewegungsrichtung berechnet man kleine Zwischenschritte, bis man die Kollisionsbarriere (Spieler oder Spielfeldrand) erreicht und spiegelt dann den Bewegungsvektor an der Kollisionsbarriere. Hat man die durch `$delta_t` berechnete Strecke zurückgelegt, ist die neue Position des Balls erreicht (Abbildung 2).

Texturen

Texturen sind eine einfache Möglichkeit, um strukturierte Oberflächen zu simulieren, z.B. eine Steinmauer oder auch einen Himmel mit Wolken. Im Fall von PONG wollen wir als Hintergrund einen Sommerhimmel erzeugen. Der Ball soll einem Tennisball ähneln und die Spieler sollen eine Holzoberfläche erhalten.

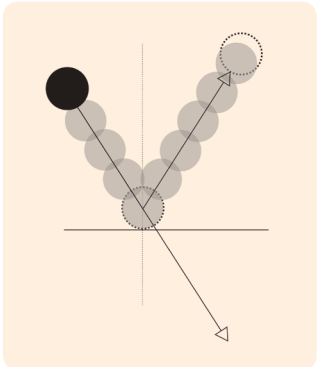


Abbildung 2: Wird die Kollision registriert (schwarze Position), werden ab dann kurze Teilschritte berechnet; Der Geschwindigkeitsvektor wird gespiegelt, sobald die Barriere erreicht ist.

```
sub gameOver {
  print "You missed the Ball - GAME OVER\n";
  exit;
}

sub moveBall {
  my ($delta_t) = @_;
  my $oldPos = [ @{$BallPosition} ];
  my $newPos = [
    $BallVelocity->
      [0] * $delta_t + $oldPos->[0],
    $BallVelocity->
      [1] * $delta_t + $oldPos->[1],
  ];

  # Abprallen des Balls am oberen Spielfeldrand
  if ($newPos->[1] + 5 >= 470) {
    $BallVelocity-> [1] = -$BallVelocity->[1];
  }

  # Abprallen des Balls am unteren Spielfeldrand
  if ($newPos->[1] - 5 <= 20) {
    $BallVelocity-> [1] = -$BallVelocity->[1];
  }

  # Abprallen des Balls am (linken) Spieler 1
  if ($newPos->[0] - 5 <= 40) {
    gameOver() if
      $newPos->[1] > $Player1Y + 50 ||
      $newPos->[1] < $Player1Y;
    $BallVelocity->[0] = -$BallVelocity->[0];
  }

  # Abprallen des Balls am (rechten) Spieler2
  if ($newPos->[0] + 5 >= 450) {
    gameOver() if
      $newPos->[1] > $Player2Y + 50 ||
      $newPos->[1] < $Player2Y;
    $BallVelocity->[0] = -$BallVelocity->[0];
  }

  return [
    $BallVelocity-> [0] * $delta_t + $oldPos->[0],
    $BallVelocity-> [1] * $delta_t + $oldPos->[1],
  ];
}

sub timer {
  my ($lastCallTime) = @_;
  my $thisCallTime = glutGet(GLUT_ELAPSED_TIME);
  my $delta_t = $thisCallTime - $lastCallTime;

  # der Ball prallt am Spieler oder Spielfeldrand ab
  $BallPosition = moveBall($delta_t);

  # Spieler 2 automatisch auf Ballhoehe bewegt
  $Player2Y = $BallPosition->[1] - 25;

  glutTimerFunc(int(1000 / $TimerCallsPerSecond), \&timer, $thisCallTime);
  glutPostRedisplay();
}
```

Listing 3

Dies sind die einzelnen verwendeten Texturen:

Abbildung 3:
Himmeltextur

Abbildung 4:
Tennisballtextur

Abbildung 5:
Holztextur

Abbildung 6:
zweite Holztextur

In OpenGL besteht eine Textur aus ein oder mehr quadratischen Pixelbildern, deren Breite (gleich Höhe) eine Potenz von 2 ist, also z.B. 256, 512 usw. Pixel Kantenlänge. Dasselbe "Bild" der Textur wird dabei in mehreren Auflösungen in der Grafikkarte platziert, so dass OpenGL je nachdem, wie weit das texturierte Objekt entfernt ist, die zur Auflösung passende Version der Textur auswählen kann.

Die folgenden kleinen Funktionen werden verwendet, um die Texturen, die z.B. im TGA Format vorliegen, zu laden. Es wird dabei das Modul `OpenGL::Image` verwendet, was Bestandteil der POGL (CPAN: OpenGL) Distribution ist. Die Funktion `loadTextures()` geht davon aus, dass die Texturdateien auf ".tga" enden und sich im "textures" Verzeichnis im aktuellen Verzeichnis befinden - siehe Listing 4.

Um die Texturen zu laden, müssen wir einmal in der `main()` Funktion die Funktion `loadTextures()` mit allen zu ladenden Texturen aufrufen:

```
sub main {
  # [...]

  glEnable(GL_TEXTURE_2D);
  loadTextures(
      'sky','tennisball','wood1','wood2');

  glutMainLoop();
}
```

```
use OpenGL::Image;
my %Textures;

sub loadTexture {
  my ($filename, $texID) = @_;
  my $tex = OpenGL::Image->new(source => $filename);
  return unless $tex->IsPowerOf2();
  my($ifmt, $fmt, $type) = $tex->Get('gl_internalformat','gl_format','gl_type');
  my($w,$h) = $tex->Get('width','height');
  glBindTexture(GL_TEXTURE_2D, $texID);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  glTexImage2D_c(GL_TEXTURE_2D, 0, $ifmt, $w, $h, 0, $fmt, $type, $tex->Ptr());
  return $texID;
}

sub loadTextures {
  my (@filenames) = @_;
  foreach my $id (glGenTextures_p(scalar @filenames)) {
    my $name = shift @filenames;
    $Textures{$name} = loadTexture('textures/'.$name.'.tga', $id);
  }
}

sub useTexture {
  my ($name) = @_;
  glBindTexture(GL_TEXTURE_2D, $Textures{$name});
}
```

Listing 4

OpenGL::Image kann von Hause aus Texturen im Bildformat TGA (Targa) laden, die aber 4 Kanäle enthalten müssen, sowie nicht komprimiert sein dürfen. Wenn OpenGL::Image auf Ihrem System mit einer Meldung abbrechen sollte z.B. "Bad file descriptor", dann hilft meist eine kleine Konvertierung des Bildes mit dem Tool "convert" (Teil von ImageMagick):

```
convert meinBild_other.tga meinBild.tga
```

Um eine Textur auf ein Objekt zu "kleben", müssen wir einzelnen Koordinaten in der Textur Vektoren der Geometrie zuweisen. Durch diese Zuweisung kann OpenGL die Textur auf die Geometrie mappen. Die Texturkoordinaten befinden sich dabei in x- und y-Richtung im Wertebereich von 0.0 bis 1.0. Die Texturkoordinate (0.5,0.5) markiert also die Mitte der Textur. Die Fläche zwischen zwei Vektoren interpoliert OpenGL automatisch.

```
sub ball
 {
   my $segments = 16;
   my @verts = ();
   foreach my $a (0..$segments) {
     my $angle = $a * (360/$segments);
     push @verts, [sin($angle*3.1415/180),
         cos($angle*3.1415/180)];
   }

   glColor4f(1,1,1,0);
   glBegin(GL_TRIANGLE_FAN);
     glTexCoord2f(0.5,0.5);
     glVertex3f(0,0,0);

     map {
       glTexCoord2f(
       ($_->[0] + 1) / 2, ($_->[1] + 1) / 2);
       glVertex3f(@{$_}, 0);
     } @verts;
   glEnd();
 }

sub player {
  glColor3f(1,1,1);
  glBegin(GL_QUADS);
    glTexCoord2f(0,0);
    glVertex3f(0,1,0);

    glTexCoord2f(1,0);
    glVertex3f(0,0,0);

    glTexCoord2f(1,1);
    glVertex3f(1,0,0);

    glTexCoord2f(0,1);
    glVertex3f(1,1,0);
  glEnd();
}
```

Um die Textur jetzt zu verwenden, also für die kommenden Vertices zu aktivieren, rufen wir die `useTexture()` Funktion mit dem entsprechenden Texturnamen auf, z.B. `useTexture('sky')`:

```
sub draw {
  # Hintergrund
  useTexture('sky');
  glColor3f(1,1,1);
  glBegin(GL_QUADS);
     glTexCoord2f(0,0);
     glVertex3f(0,0,-1);
     glTexCoord2f(1,0);
     glVertex3f(500,0,-1);
     glTexCoord2f(1,1);
     glVertex3f(500,500,-1);
     glTexCoord2f(0,1);
     glVertex3f(0,500,-1);
  glEnd();

  # Ball
  useTexture('tennisball');
  glPushMatrix();
    glTranslatef($BallPosition->[0],
       $BallPosition->[1], 0);
    glScalef(10, 10, 1);
    ball();
  glPopMatrix();

  # Spieler 1
  useTexture('wood1');
  glPushMatrix();
    glTranslatef(20, $Player1Y, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();

  # Spieler 2
  useTexture('wood2');
  glPushMatrix();
    glTranslatef(460, $Player2Y, 0);
    glScalef(20, 50, 1);
    player();
  glPopMatrix();
}
```

Shader

Die Philosophie von OpenGL ist, die grundlegenden Funktionen zur Verfügung zu stellen, die ausgeführt werden müssen, um ein Bild zu erzeugen, und dies möglichst in Hardware, der Grafikkarte, implementiert. Diese Funktionen werden auch als *Fixed Function Pipeline* bezeichnet, also die Ausführung von festgelegten Abläufen. Z.B. enthält diese Pipeline die Schritte *Transformation der Vertices* sowie *Einfärben eines Pixels*. Daneben gibt es noch viele weitere Schritte, z.B. das Darstellen von Texturen. Für eine genaue Darstellung der Pipeline, sei auf *http://www.songho.ca/opengl/gl_pipeline.html* verwiesen.

Um einen leichten Einstieg in OpenGL zu erhalten, bietet sich an, die Standardfunktionen der Pipeline einfach zu benutzen. Will man aber mehr, und das wollen vor allem Spieleprogrammierer, so bietet OpenGL die Möglichkeit, einzelne Teile der Pipeline zu programmieren. Diese Programme, die Teile der Fixed Function Pipeline ersetzen, heißen Shader und sind in GLSL (OpenGL Shading Language) geschrieben, einer C-ähnlichen Sprache. GLSL wird ebenfalls von der Khronos Group entwickelt, welches auch schon den OpenGL Standard eingeführt hat. Unter *http://www.opengl.org/sdk/docs/manglsl/xhtml/*ist GLSL dokumentiert.

Es gibt bisher drei Typen von GLSL Shadern, die jeweils einen anderen Teil der Pipeline ersetzen:

- **Vertexshader:** Diese werden für jeden Vertex ausgeführt und übernehmen in der Standardpipeline die Transformation und Projektion der Vertices.
- **Fragmentshader:** Diese werden für jeden Pixel ausgeführt und übernehmen in der Standardpipeline das Einfärben der Pixel sowie das Darstellen von Texturen.
- **Geometryshader:** Diese erzeugen neue Vertices, was die Vertexshader nicht können. Geometryshader wurden mit GLSL 3.2 eingeführt und sind deshalb noch nicht überall verfügbar.

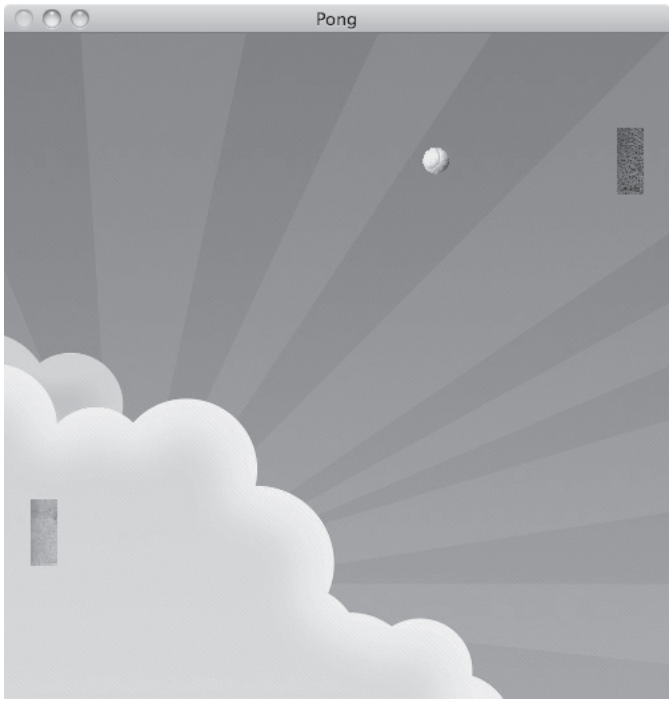


Abbildung 7: PONG mit Texturen und Animation

Nachfolgend wollen wir unser Programm mit Hilfe von einem Vertexshader und einem Fragmentshader etwas hübscher gestalten.

Das OpenGL Perl Modul unterstützt GLSL Shader leider nicht vollständig, so dass wir eine kleine Erweiterung vornehmen müssen. Zuerst sollte das Modul OpenGL::Shader installiert werden, worin sich auch das Modul OpenGL::Shader::GLSL befindet, welches wir um die Methode `SetSampler2D()` erweitern müssen, um später auch Texturen im Shader verarbeiten zu können:

```
package MyShader;

use OpenGL qw(:glfunctions :glconstants :
    glufunctions :gluconstants);
use OpenGL::Shader;
use base 'OpenGL::Shader::GLSL';

sub SetSampler2D {
  my($self, $var, $texID) = @_;
  # Die ID der Uniformvariable namens
  # $var abrufen.
  my $id = $self->Map($var);
  return 'Unable to map $var'
    if (!defined($id));
  # Den Wert der Uniformvariable setzen.
  glUniform1iARB($id, $texID);
  return '';
}
package main;
# [...]
```

Jetzt gehen wir ähnlich wie bei den Texturen vor und erzeugen eine globale Variable `$Shader`, die eine Instanz der Klasse OpenGL::Shader::GLSL ist. In der neuen Funktion `initShader()` (siehe Listing 5) erzeugen wir nun den Shader, indem wir die beiden GLSL Programmquelldateien laden. Der erste Dateiname im Aufruf von `LoadFiles()` benennt den Fragmentshader und der zweite den Vertexshader. In `initShader()` binden wir außerdem noch jede Textur, die wir vorher geladen haben, an einen bestimmten Texturbuffer, so dass wir diese später im Shader verwenden können.

Die beiden folgenden Aufrufe binden eine bestimmte Textur an einen Texturbuffer, der dann im Shader verwendet werden kann. Dabei aktiviert der Aufruf von `glActiveTextureARB()` den Texturbuffer, hier den vierten (`GL_TEXTURE4`) und der Aufruf von `glBindTexture()` bindet eine entsprechend (vor geladene) Textur an diesen Buffer. Da wir vier Texturen im Shader verwenden wollen, belegen wir einfach die ersten vier Texturbuffer mit den entsprechenden Texturen (s.o.).

```
my $Shader;

sub initShader {
  $Shader = new MyShader();
  my $stat = $Shader->LoadFiles("shaders/fragment.glsl","shaders/vertex.glsl");
  # Alle Texturen in einen separaten Texturbuffer laden.
  foreach my $tex (keys %Textures) {
    my $texID = $Textures{$tex};
    glActiveTextureARB(GL_TEXTURE1) if $texID == 1;
    glActiveTextureARB(GL_TEXTURE2) if $texID == 2;
    glActiveTextureARB(GL_TEXTURE3) if $texID == 3;
    glActiveTextureARB(GL_TEXTURE4) if $texID == 4;
    glBindTexture(GL_TEXTURE_2D, $texID);
  }
}
```

Listing 5

```
sub useShader {
  my ($texname) = @_;
  my $texID = $Textures{$texname};
  $Shader->Enable(); # Den Shader aktivieren.
  # Die Uniformvariablen des Shaders mit Werten belegen.
  $Shader->SetVector('color',0.3,0,0,0.1); # setze die uniform-Variable "color"
  $Shader->SetSampler2D('texture', $texID); # setze die uniform-Variable "texture"
}
sub unuseShader {
  $Shader->Disable();
}
```

Listing 6

```
glActiveTextureARB(GL_TEXTURE4);
glBindTexture(GL_TEXTURE_2D, $texID);
```

Unser Ziel ist es erst einmal die Funktionalität der Fixed Function Pipeline weitestgehend durch selbstgeschriebene Shader zu ersetzen, die wir anschließend dann modizifieren können, um diverse Bildeffekte zu erzeugen. Anstatt der `useTexture()` Funktion benötigen wir eine Funktion `useShader()` (siehe Listing 6), die ebenfalls einen Texturnamen als ersten Parameter erhält. Diese Funktion aktiviert den Shader und setzt globale Variablen, die der Shader benötigt, in unserem Fall sind das die Objektgrundfarbe *color* und die Objekttextur *texture*. Für letztere verwenden wir die oben erweiterte Methode `SetSampler2D()`. `unuseShader()` ist das Gegenstück zu `useShader()` und deaktiviert den Shader.

In der `main()` Funktion (Listing 7) wird jetzt `initShader()` aufgerufen und die `display()` und `draw()` Funktionen werden ebenfalls leicht modifiziert, so dass `useShader()` statt `useTexture()` aufgerufen wird.

Jetzt kommt der eigentlich interessante Teil, die Shader selbst. GLSL ähnelt sehr C, d.h. es gibt eine Funktion `main()`, die die Shaderlogik beinhaltet. Es können weitere Funktionen definiert werden. An den verfügbaren Datentypen sieht man

```
sub main {
  # [...]
  initShader();
  # [...]
  glutMainLoop();
}
sub display {
  # [...]
  draw();
  unuseShader();
  glutSwapBuffers();
}
sub draw {
  # Hintergrund
  # statt useTexture('sky');
  useShader('sky');
  glColor3f(1,1,1);
  glBegin(GL_QUADS);
     # [...]
  glEnd();
  # Ball
  # statt useTexture('tennisball')
  useShader('tennisball');
  glPushMatrix();
    # [...]
  glPopMatrix();
  # Spieler 1
  # statt useTexture('wood1');
  useShader('wood1');
  # [...]
  # Spieler 2
  # statt useTexture('wood2');
  useShader('wood2');
  # [...]
}
```

Listing 7

die Spezialisierung von GLSL auf grafische Algorithmen: Es gibt Datentypen für Vektoren bzw. Farben (vec2, vec3, vec4) sowie Datentypen für Texturen (sampler2D) und Matrizen. Daneben gibt es noch alte Bekannte aus C: float und int.

Zuerst schreiben wir den Code für den Vertexshader, da dieser in unserem Beispiel bewusst einfach gehalten ist und nur die Funktionalität der Standardpipeline implementiert:

```
void main () {
  gl_Position =
    gl_ModelViewProjectionMatrix * gl_Vertex;
  gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

Im Vertexshader gibt es die vordefinierte Variable `gl_Vertex`, die die Position des Vektors beinhaltet, für die der Shader aktuell ausgeführt wird, also z.B. der mit `glVertex3f()` definierte Vektor in der `draw()` Funktion. Die Variable `gl_ModelViewProjectionMatrix` beinhaltet die Projektionsmatrix inklusive Modeltransformation. Die Multiplikation der von `gl_Vertex` und `gl_ModelViewProjectionMatrix` ergibt die Position im so genannten *Weltkoordinatensystem* ("finale" Position). Diese wird nun `gl_Position` zugewiesen, was im Grunde dem "Ausgabewert" des Shaders entspricht.

Ein weiterer "Ausgabewert" des Shaders ist die Variable `gl_TexCoord`, ein Array aus Texturkoordinaten. Darin speichern wir die Texturkoordinate, die dem aktuellen Vertex zugeordnet ist, also der Wert, der mit `glTexCoord2f()` in der `draw()` Funktion gesetzt wird und der in der Variablen `gl_MultiTexCoord0` zur Abholung bereit steht.

Der Vertexshader macht also nichts weiter, als den Vertex in das Weltkoordinatensystem zu transformieren sowie die Texturkoordinate zu speichern, so dass wir diese im Fragmentshader verwenden können.

Der Fragmentshader hat einen wichtigen "Ausgabewert", nämlich die Farbe des aktuellen Pixels (`gl_FragColor`), die er berechnen soll. Um die Pixelfarbe zu berechnen braucht der Shader jetzt die Shadervariablen (Uniforms) "color" und "texture", die wir im Hauptprogramm in `useShader()` mit den aktuellen Werten belegen. Die Pixelfarbe setzt sich also aus der Texturfarbe an einer bestimmten Stelle (an der Stelle der Texturkoordinate) und der Objektfarbe zusammen. Erstere erhalten wir über die Variable `gl_TexCoord[0]`, die im Vertexshader gesetzt wurde. Die praktische GLSL Funktion `texture2D()` sucht nun einfach nur die Farbe aus der Textur `texture` an der Position `gl_TexCoord[0]` heraus und durch Addition mit der Objektfarbe `color` erhalten wir die finale Pixelfarbe.

```
uniform sampler2D texture;
uniform vec4 color;

void main () {
  vec4 col = color + texture2D(
               texture, vec2(gl_TexCoord[0]));
  gl_FragColor = col;
}
```

Die genannten Shadervariablen werden auch *Uniforms* genannt und müssen im Shader mit dem Schlüsselwort `uniform` global deklariert werden und können dann, wie beschrieben, per `SetVector()` oder `SetSampler2D()` mit Werten belegt werden.

Es mag verwundern, dass wir die Texturkoordinate für einen bestimmten Pixel aus `gl_TexCoord[0]` erhalten, denn wir haben ja die Texturkoordinate einem *Vertex* und keinem Pixel zugewiesen. Aber hier interpoliert OpenGL einfach die Texturkoordinaten zwischen mehreren Vertices. Das sieht man beispielsweise daran, dass ja auch auf der Fläche zwischen den Punkten eines Dreiecks die Textur dargestellt wird. Würde keine Interpolation stattfinden, wäre die Textur nur an denjenigen Pixeln definiert, die einen Vertex beinhalten.

Der nachfolgende Fragmentshader stellt nun eine Modifizierung der Standardpipeline dar. Die Funktion `brightness()` erhöht eine Farbe um einen bestimmten Wert. Die Funktion `posterize()` erzeugt den "Posterisierungseffekt", d.h. die Farbwerte werden diskretisiert (es werden weniger Farben verwendet), was durch ein einfaches Casten erreicht wird. Diese beiden Funktionen sollen nur die Möglichkeiten von Fragmentshadern andeuten - siehe Listing 8.

Wer sich näher mit Shadern auseinandersetzen möchte, dem seien die beiden GLSL Tutorials unter *http://wiki.delphigl.com/index.php/Tutorial_glsl* und *http://wiki.delphigl.com/index.php/Tutorial_glsl2* empfohlen, sowie die offizielle Spezifikation unter *http://www.opengl.org/sdk/docs/manglsl/xhtml/*

```
float posterizeColor (float x, int numOfColors) {
  int y = int(x * float(numOfColors));
  return float(y) / float(numOfColors);
}
vec4 posterize (vec4 color, int numOfColors) {
  return vec4(
    posterizeColor(color.r, numOfColors),
    posterizeColor(color.g, numOfColors),
    posterizeColor(color.b, numOfColors),
    posterizeColor(color.a, numOfColors)
  );
}
vec4 brightness (vec4 color, float value) {
  return color * value;
}
void main () {
  vec4 col = color + texture2D(texture, vec2(gl_TexCoord[0]));
  col = brightness(posterize(col, 5), 1.3);
  gl_FragColor = col;
}
```

Listing 8

Beschreibung der neuen verwendeten OpenGL Funktionen

Im Nachfolgenden Abschnitt werden folgende Kurzbezeichnungen für die Parameter-Datentypen verwendet: i für Ganzzahl, r für reelle Zahl zwischen 0 und 1, sub für Funktionsreferenz, c für Konstante und s für String (Zeichenkette).

glutTimerFunc(*i* Verzögerung, *sub* Funktion, *i* Argument

`glutTimerFunc)` setzt eine Funktion, die mit einer bestimmten Verzögerung ausgeführt wird. Der Funktion wird beim Aufruf das angegebene Ganzzahl-Argument übergeben.

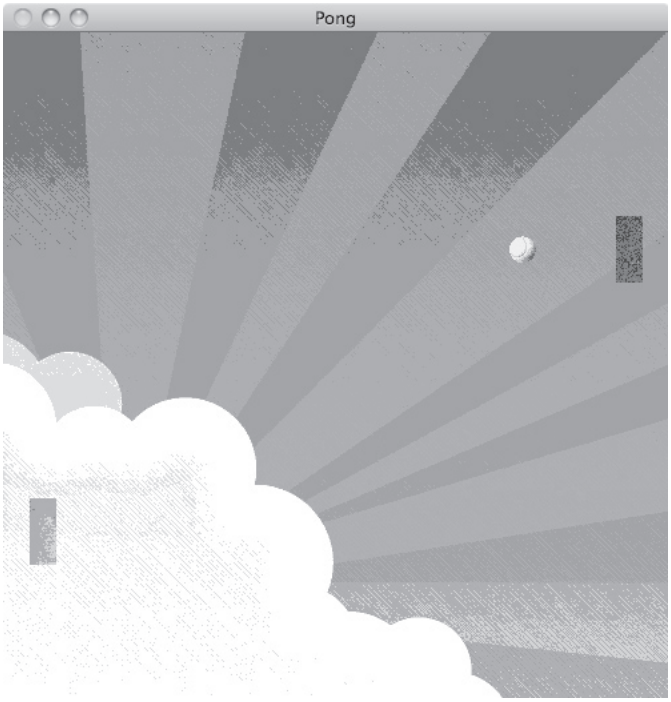


Abbildung 8: Der Posterisierungseffekt des Fragmentshaders reduziert die Farben, so dass die Farbverläufe rauer wirken.

glutSpecialFunc(*sub* Funktion)

`glutSpecialFunc()` setzt die Funktion, die ausgeführt wird, wenn eine spezielle Taste, keine alphanumerische Taste, ausgeführt wird.

glBindTexture(*c* Ziel, *i* ID)

`glBindTexture()` aktiviert die Textur, die mit der angegebenen Textur-ID verknüpft ist. Das Ziel ist in unserem Fall `GL_TEXTURE_2D`.

glTexParameteri(*c* Ziel, *c* Parametername, *c* Wert)

`glTexParameteri()` setzt Texturparameter für die aktuell aktive Textur. Das Ziel ist in unserem Fall `GL_TEXTURE_2D` und folgende Parameter sind u.a. definiert (`Parametername =Wert`):

- `GL_TEXTURE_WRAP_S = GL_REPEAT`: Textur in s-Richtung wiederholen
- `GL_TEXTURE_WRAP_T = GL_REPEAT`: Textur in t-Richtung wiederholen
- `GL_TEXTURE_MAG_FILTER = GL_LINEAR`: Texturvergrößerungsfunktion wird verwendet, sofern nötig
- `GL_TEXTURE_MIN_FILTER = GL_LINEAR`: Texturverkleinerungsfunktion wird verwendet, sofern nötig

Viele weitere Optionen sind möglich. Es sei auf das hervorragende DGL Wiki verwiesen: *http://wiki.delphigl.com/DGL Wiki* und *http://wiki.delphigl.com/index.php/glTexParameter.*

glTexEnvf(*c* Ziel, *c* Parametername, *c* Wert)

`glTexEnvf()` setzt globale Texturkonfigurationen. Ziel muss `GL_TEXTURE_ENV` sein, Parametername muss `GL_TEXTURE_ENV_MODE` und der Wert kann u.a. `GL_MODULATE` sein. `GL_MODULATE` definiert dabei, dass eine bestimmte Funktion verwendet wird, um aus der Texturfarbe an einer bestimmten Texturkoordinate eine Vertexfarbe zu berechnen. Es gibt verschiedene Ansätze dafür, die im DGL Wiki gut beschrieben werden (*http://wiki.delphigl.com/index.php/glTexEnv|glTexEnv()*).

glTexImage2D_c(*c* Ziel, *i* Level, *i* internes Format, *i* Breite, *i* Höhe, *i* Rahmenbreite, *c* Format, *c* Typ, *C-Pointer* Pixeldaten)

`glTexImage2D_c()` lädt Pixeldaten in die aktuell aktive Textur. Das Ziel ist in unserem Fall `GL_TEXTURE_2D` und Level 0. Neben den anderen Parametern, die alle im DGL Wiki dokumentiert sind, ist der letzte ein Pointer auf die eigentlichen Pixeldaten. Um diesen in Perl zu erhalten, muss die Methode `Ptr()` der OpenGL::Image Instanz aufgerufen werden: `$tex->Ptr()`.

glGenTextures_p(*i* Anzahl)

`glGenTextures_p()` generiert eine gegebene Anzahl von Textur und gibt ein Array mit den IDs dieser Texturen zurück. Dies schafft nur Platz für die Texturen. Sie müssen vor Verwendung noch konfiguriert und mit Pixeldaten gefüllt werden.

glTexCoord2f(*r* s, *r* t)

`glTexCoord2f()` setzt die aktuelle Texturkoordinate. s und t sind für Texturen das, was x und y im Raum sind und liegen immer zwischen 0 und 1. Wenn anschließend z.B. mit `glVertex()` ein neuer Vertex definiert wird, so wird die mit `glTexCoord2f()` gesetzte Texturkoordinate für diesen Vertex verwendet.

glUniform1iARB(*i* Id, *i* Wert)

`glUniform1iARB()` setzt eine Uniformvariable vom Typ Integer auf den Wert `Wert`. `Id` ist dabei nicht der Name der Uniformvariable, sondern deren ID, die OpenGL ihr zugewiesen hat.

glActiveTextureARB(*c* Texturbuffername)

`glActiveTextureARB()` aktiviert den angegebenen Texturbuffer für die nachfolgenden Aufrufe, die diesen Buffer z.B. beschreiben oder konfigurieren. Je nach Grafikkartenmodell gibt es mehr oder weniger verfügbare Texturbuffer, wobei meist mindestens vier verfügbar sind, namens `GL_TEXTURE1` bis `GL_TEXTURE4`. Auf aktuellen Grafikkarten sind bis zu 80 Textureinheiten möglich.

YAPC::EU 2012
August 2012 in Frankfurt

Die YAPC::EU ist mit rund 300 Teilnehmern die größte europäische Veranstaltung rund um Perl...

Interesse an Sponsoring?
sponsoring@yapc2012.de

TPF News

Antrag auf Hague-Grant: Macros für Rakudo

Carl Mäsak hat bei der Perl Foundation einen Antrag für einen Hague-Grant gestellt. Er möchte in Rakudo Makros implementieren. Diese Makros werden auf dem Abstract Syntax Tree von Rakudo basieren. Dieser Grant soll auch dazu dienen, die Spezifikation von Makros zu festigen. Im Moment sind die Definition und Spezifikation noch eher vage - auch weil noch keine Implementierung existiert, an der man Probleme sehen und neue Ideen entwickeln kann.

Von Anfang an will Mäsak auf eine gute Abdeckung durch Tests achten.

Perl 5 Core Grants: "Improving Perl 5"-Antrag wurde angenommen

Die Perl Foundation hat den Grant-Antrag von Nicholas Clark angenommen. Clark wird in den nächsten Monaten daran arbeiten, Teile des Perl 5 Kerns zu refactorn, Bugs zu fixen und neue Features hinzuzufügen. Dazu werden 400 Stunden während der nächsten drei Monate eingeplant. Ingesamt beläuft sich der Grant auf 20.000 US$.

Beim Refactoring wird darauf geachtet, dass der Code kleiner wird (ohne zu golfen), der kompilierte Code kleiner und/oder der verbrauchte Speicher geringer wird. Beim Bugfixing wird Nicholas Clark sich auf die Bugs konzentrieren, die "keiner" machen möchte und tiefen Eingriff in den Core benötigen.

Der Grant wird mit Geldern aus dem Perl 5 Core Maintenance Spendentopf bezahlt.

Google Summer of Code 2011 - der Abschluss

Der Google Summer of Code 2011 ist vorbei und Google hat mit Hilfe der teilnehmenden Projekte/Organisationen die Arbeiten evaluiert.

Die Perl Foundation hat in diesem Jahr mit 6 Studenten teilgenommen und alle Projekte wurden erfolgreich abgeschlossen.

Diese Perl-Projekte waren mit dabei:

1. André Walker – Rework Catalyst component setup code
betreut von Tomas Doran und Eden Cardim
2. Brian Neil Fraser – Making the Perl Core UTF-8 clean
betreut von Florian Ragwitz und Zefram
3. Carlos Ivan Sosa – Removing the upgrading necessity of the Dancer script with a module
betreut von Sawyer X und Franck Cuny
4. Marc Green – Standardization of core documentation parsing tools
betreut von Ricardo Signes und David E. Wheeler
5. Moritz Onken – CPAN search for the modern web
betreut von Clinton Gormley und Olaf Alders
6. Tadeusz Sosnierz – Pod parser for Rakudo
betreut von Moritz Lenz und Carl Mäsak

Wir gratulieren zu diesem Erfolg!

Perl 5 Spendentopf: Dijkmat, Oslo.pm und Net-a-Porter spenden...

Auf der diesjährigen YAPC::Europe wurden weitere Spenden für den Perl 5 Spendentopf verkündet: Die Oslo Perlmongers spenden 1.000 US$, Dijkmat spendet 5.000 US$ und Net-a-Porter spendet 10.000 US$.

Damit sind bis jetzt über 48.000 US$ zusammengekommen. Mit diesem Geld werden Dave Mitchell und Nicholas Clark für Bugfixing und andere Maintenancearbeiten am Perl 5 Kern bezahlt.

Grant-Update: Fixing Perl 5 Core Bugs

Im Juni 2011 hat Dave Mitchell durchschnittlich nur 4 Stunden pro Woche an seinem Grant gearbeitet. Das ist aber dadurch bedingt, dass er an einigen Bugfixes gearbeitet hat, die nicht unter diesen Grant fallen.

Zu den Arbeiten am Grant gehörte das Bearbeiten von Bugs, die mit der Sicherheit von Perl zu tun haben: Das Zusammenspiel von Konstanten, index() und open() hat im Taint-Modus unerwartet einen Fehler produziert.

Perl 5 Core Grants: Bis zu 10.000 US$ von Vienna.pm

Am Tag nachdem die Perl Foundation den Perl 5 Spendentopf angekündigt hatte, haben die Wiener Perlmongers verkündet, bis zu 10.000 US$ zu spenden. Sie werden für jeden Dollar, der von anderen gespendet wird, nochmal 1 Dollar dazugeben - bis zu maximal 10.000 US$.

Perl 5 Spendentopf: cPanel und booking.com spenden jeweils 10.000 US$

Nach den Wiener Perlmongers haben jetzt auch cPanel und Booking.com angekündigt, jeweils 10.000 US$ für den "Perl 5 Core Maintenance"-Spendentopf. Dazu kommen noch unzählige kleinere (und größere) Spenden von Privatleuten und anderen Unternehmen. Mit den über 30.000 US$ werden Programmierer wie Dave Mitchell und Nicholas Clark dafür bezahlt, die eher unbeliebten Aufgaben bei der Perl 5 Kernentwicklung übernehmen.

CPAN News XX

Game::TextPacMonster

Das gute alte Pacman! Eine ganz einfach Textversion für die Konsole stellt Game::TextPacMonster bereit. Man muss über die Tastatur versuchen, so viele "." wie möglich aufzufressen. Nach dem Start über `perl -MGame::TextPacMonster -e 'Game::TextPacMonster->level1->run'` erscheint auf der Konsole das Spielfeld. Neben `level1` gibt es noch `level2` und `level3`.

```
###########
#.V..#..H.#
#.##...##.#
#L#..#..R.#
#.#.###.#.#
#....@....#
###########
feed(s) left: 28
time left: 50
log:
Your turn [k/j/h/l/.]:
```

Test::Flatten

Yuji Shimada sieht Probleme mit Subtests, z.B. dass diese die Zahl der Tests verfälscht, weil immer nur die Zahl der "Gruppen" genommen wird und die Zahl der Subtests nicht mitgezählt werden. Um diese Probleme zu beseitigen, hat er Test::Flatten geschrieben.

```
use Test::More;
use Test::Flatten;

subtest 'foo' => sub {
    pass 'OK';
};

subtest 'bar' => sub {
    pass 'ok';
    subtest 'baz' => sub {
        pass 'ok';
    };
};

done_testing;
```

Email::Queue

Wer viele Mails aus seiner Anwendung heraus verschickt, kann - z.B. wenn der Mailserver nicht erreichbar ist - die Anwendung sehr stark ausbremsen. Mit Email::Queue kann man die Mails in der Anwendung erstellen, aber den Versand auf einen späteren Zeitpunkt verschieben bzw. von einem anderen Programm verschicken. Das Modul arbeitet mit Email::Simple-Objekten und serialisiert diese auf der Platte.

```
use File::Queue;

my $fq = File::Queue->new(
    path   => '/tmp/emails',
    mkpath => 1
);

# Create a message
my $message = Email::Simple->create(...);

# Schedule the message 50 times (for the sake of the example)
$fq->schedule($message) for ( 0 .. 49 );

# ... much later ... from another script ....

# Process the scheduled messages
while ( my $message = $fq->next ) {

    # sendmail is an imaginery sub that sends RFC2822 formatted emails.
    sendmail( $message->as_string );
}
```

HTTP::BrowserDetect

Welcher Browser wird da benutzt? In vielen Webanwendungen werden je nach Browser unterschiedliche Funktionalität geladen oder für bestehende Funktionalität Workarounds ausgegeben. Das ist notwendig, weil die Browserhersteller teilweise sehr unterschiedliche Auffassungen davon haben wie der Browser mit gewissen Dingen umzugehen hat. Mit HTTP::BrowserDetect ist es möglich, den Browser zu "identifizieren" und dann im Code zu reagieren.

```
use HTTP::BrowserDetect;

my $browser_string =
    "Mozilla/5.0 (iPad; U; CPU OS 4_3_4 like Mac OS X; en-us)
      AppleWebKit/533.17.9 (KHTML, like Gecko) Version/5.0.2 Mobile/8K2 Safari/6533.18.5";
my $browser = HTTP::BrowserDetect->new( $browser_string );

for my $type ( qw(ie firefox safari mac windows) ) {
    printf "%s: %s\n", $type, $browser->$type();
}

for my $level (qw(version major minor)) {
    my $method = 'public_' . $level;
    printf "Version $level: %s\n", $browser->$method();
}
__END__
ie:
Use of uninitialized value in printf at browser_detect.pl line 11.
firefox:
safari: 1
mac:
windows:
Version version: 5
Version major: 5
Version minor: .0
```

Parse::BBCode

Aus Foren und Kommentarfelder verschiedenster Webseiten kennt man sie - BBCodes. Häufig sind Tags wie [code] oder [url] dabei. Parse::BBCode ist ein Modul, mit dem man solchen BBCode in verschiedene Zielformate wie HTML oder Markdown umwandeln kann. Das Modul ist dabei sehr flexibel was die Tags angeht - man kann sehr leicht eigene Tags hinzufügen - oder auch beim Handling von Escapes

```
use Parse::BBCode;

my $text = 'Ich will so ein Magazin:
[url]http://perl-magazin.de[/url].
[perl]$foo[/perl]
[magic]$bar[/magic]';

my $parser = Parse::BBCode->new({
    tags => {
        Parse::BBCode::HTML->defaults,
        perl  => '<pre>%s</pre>',
        magic => '<pre>%{euro}s</pre>',
    },
    escapes => {
        Parse::BBCode::HTML->default_escapes,
        euro => sub{
            my ($parser, $tag, $text) = @_;
            $text =~ s/\$/\x{20ac}/g;
            return $text;
        },
    },
});

print $parser->render( $text );
__END__
Ich will so ein Magazin:<br>
<a href="http://perl-magazin.de" rel="nofollow">http://perl-magazin.de</a>.<br>
<pre>$foo</pre><br>
<pre>€bar</pre>
```

Termine

November 2011

01. Treffen Frankfurt.pm
Treffen Vienna.pm
03. Treffen Dresden.pm
04.-05. TwinCity Perl-Workshop Wien/Bratislava
07. Treffen Hamburg.pm
08. Treffen Stuttgart.pm
12. London Perl-Workshop
12.-13. OpenRheinRuhr
14. Treffen Ruhr.pom
Treffen Hannover.pom
16. Treffen Darmstadt.pm
Treffen Munich.pm
18.-20. Foswiki Camp
21. Treffen Erlangen.pm
29. Treffen Bielefeld.pm

Dezember 2011

01. Treffen Dresden.pm
05. Treffen Hamburg.pm
06. Treffen Frankfurt.pm
Treffen Vienna.pm
06. Treffen Dresden.pm
12. Treffen Ruhr.pm
Treffen Hannover.pm
13. Treffen Stuttgart.pm
19. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
27. Treffen Bielefeld.pm
28. Treffen Berlin.pm

Januar 2012

02. Treffen Hamburg.pm
03. Treffen Frankfurt.pm
Treffen Vienna.pm
05. Treffen Dresden.pm
09. Treffen Ruhr.pm
Treffen Hannover.pm
10. Treffen Stuttgart.pm
16. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
Trefen Munich.pm
25. Treffen Berlin.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.perl-community.de

Perl-Community.de ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perlmongers.de/
http://www.pm.org/

Das Online-Zuhause der Perl-Mongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Außerdem kann man auch neue Gruppen gründen und bekommt Hilfe...

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern.

http://www.perl-foundation.org

Die Perl-Foundation nimmt eine führende Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundationen geleistet. Jedes Jahr werden Studenten beim "Google Summer of Code" betreut.

http://www.Perl.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

einraumwerk

Die Individual-Programmierung für OTRS-Anwender

Leistungen

Spezialisierung auf die individuelle Entwicklung von Modulen für OTRS.

Langjährige Erfahrung mit OTRS und der Programmierung von speziellen Anforderungen ermöglicht eine effiziente und kostensparende Arbeitsweise.

Mit der Bewältigung großer Aufgaben ebenso vertraut wie mit der sorgfältigen Abwicklung kleinerer Problemlösungen.

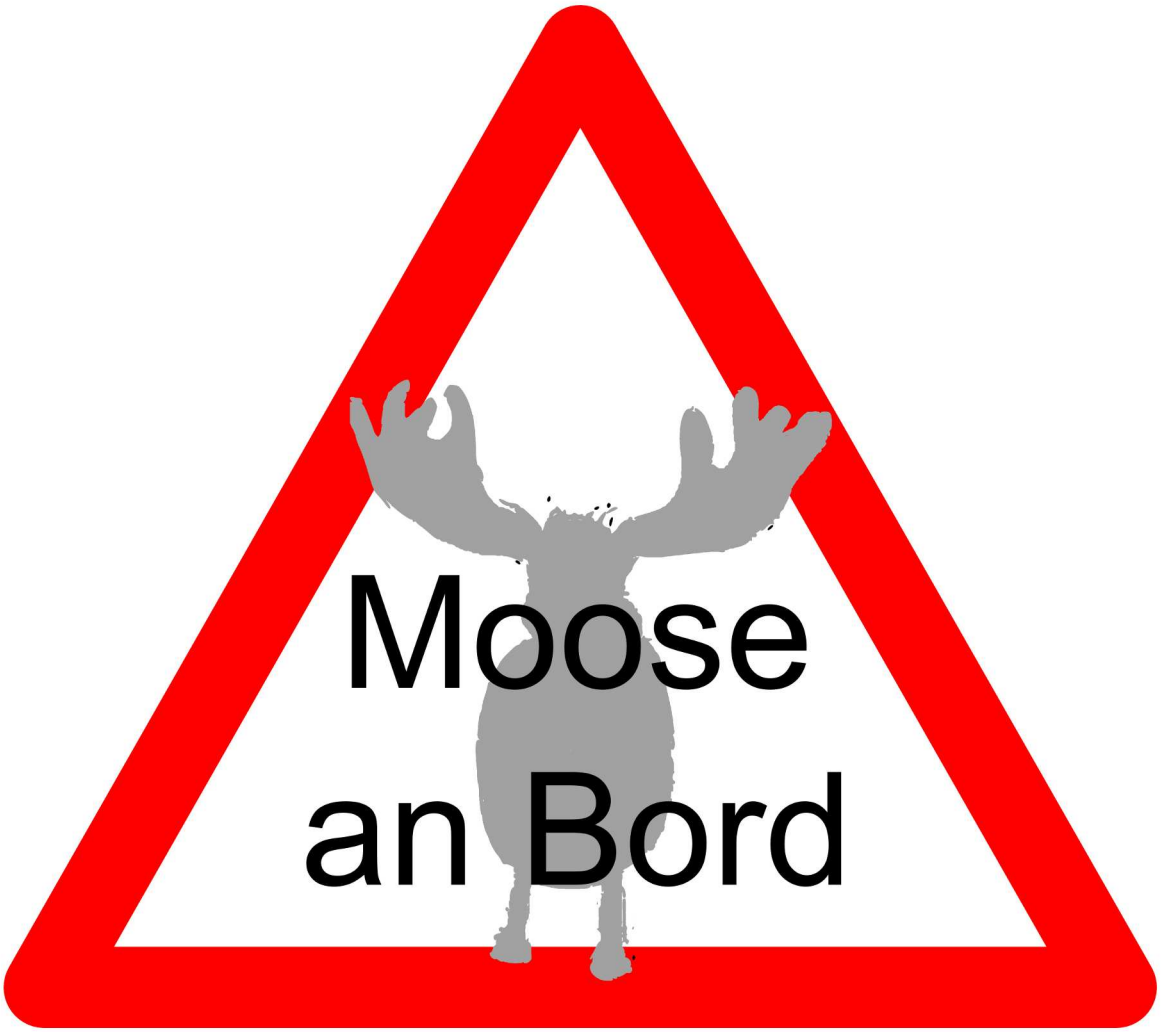
Bewährte Vorgehensweise:

* Erstellung technischer und funktionaler Spezifikationen
* Auf Basis der Spezifikation die Sicherstellung hoher Qualität
* Alternative Lösungsszenarien werden wenn möglich aufgezeigt
* Erstellung des verbindlichen Zeitplans und die dazugehörige Kostenschätzung

Leistungen:

* Entwicklung neuer Module und Funktionen für OTRS
* Anpassung vorhandener Module bei Versions-Wechsel

Kontakt:
Uwe Dieckmann
ud@einraumwerk.de



Perl-Services.de

Programmierung - Schulung - Perl-Magazin

info@perl-services.de