Online unter http://lua.bumuckl.com - Lua Tutorials, Codebase & Forum

Weblinks:
http://www.bumuckl.com
http://lua.bumuckl.com
http://homebrewdump.dumpspot.net
http://www.parabella.org
http://www.pspsource.de
http://www.evilmana.com
http://www.psp-programming.com
http://www.luaplayer.org
http://pgelua.luaplayer.org
http://luawiki.bumuckl.com

Inhaltsverzeichnis

1. Bevor es losgehen kann

1.1. Vorwort

Ich persönlich "programmiere" LUA für die PSP seit des Releases des LUAPlayer für die Sony PSP.
LUA ist eine Scriptsprache, was den Vorteil hat, dass man seinen Code nicht kompilieren muss, d.h. nicht in Maschinencode umwandeln oder eben für das Gerät verständlich machen muss. Zudem ist LUA sehr einfach zu erlernen und gibt auch Leuten eine Chance, die noch nie in ihrem Leben Code geschrieben haben. Soll heißen, man braucht absolut keine Erfahrung mit anderen Programmiersprachen. Allerdings sind die Möglichkeiten von LUA begrenzt. Wer absolut grenzenlos programmieren will, muss den schweren, steinigen Weg zu C/C++ einschlagen, eine waschechte, recht anfängerunfreundliche Programmiersprache.
Dieses eBook soll einen kleinen Einblick in LUA geben. Es werden die Grundlagen vermittelt, ein Beispielprojekt wird geschrieben und es dient ebenso als große Funktionsrefrenz zum Nachschlagen.
Genug gelabert, wollen wir doch loslegen...

1.2. Was braucht man alles

Man braucht eine homebrewfähige PSP, einen LUA-Interpreter für die PSP (ja, es gibt verschiedene!), und zum Coden reicht der Notepad-Editor von Windows vollkommen aus. Dennoch empfehle ich Notepad++, welcher weitaus mächtiger ist,.
Wenn man nicht jedes mal seinen Code auf die PSP schaufeln und dort testen will, kann man das auch auf dem PC machen, mit dem "LUAPlayer for Windows". Voraussetzung ist, dass man eben Luaplayer-konformen Code hat.
Es gibt verschiedene "Interpreter", die den Lua-Code "interpretieren" und ausführen. Und es gibt leider bei den Befehlen nicht durchgehend eine einheitliche Syntax. Beim sogenannten Luaplayer lauten die meisten Befehle nämlich anders als bei PGELua.

Den "Luaplayer for Windows gibt" es hier:
http://www.bumuckl.com//index.php?option=com_remository&Itemid=2&func=fileinfo&id=391

Um den "LUAPlayer for Windows" auch benutzen zu können, muss man ein paar Kleinigkeiten einstellen, bzw erstellen.:
Extrahiere den Inhalt des LUAPlayer for Windows. Öffne dann den Ordner und erstelle darin eine´Datei mit dem Namen "script.cmd". Diese öffnest du dann mit einem Texteditor und fügst folgendes ein:

```
luaplayer script.lua
pause
```

Speicher dann ab. Jetzt kannst du deine LUA-Programme auch auf deinem PC starten, ohne jedes mal alles auf die PSP zu kopieren. Es muss sich allerdings um Code für den Luaplayer handeln, nicht für PGELua.

1.3. Der richtige LuaInterpreter

Wie gesagt, es gibt jede Menge verschiedene LUAInterpreter für die PSP:
- LUAPlayer 0.16 - 0.20 : Das ist der Standard LUAPlayer mit Standardfunktionen, den man normalerweise verwendet (ist in der Regel aber nur lauffähig bis CFW 3.52 M33 und PSP FAT)
- LUAPlayer Mod 2 - Mod 4 : Ein gemoddeter LUAPlayer von Cools, mit massenweise nützlichen Funktionen...
(Allerdings auch nur lauffähig bis CFW 3.52 M33 und PSP FAT)
- Luaplayer HM, ein gemoddeter LuaPlayer von HomeMister91, bietet noch mehr Funktionen als der Luaplayer von Cools, läuft auch auf allen CFW's und auf PSP Fat sowie PSP Slim. Wird allerdings mit zunehmender Versionszahl inkompatibler und instabiler, und sogar auch umständlicher. Ich empfehle LuaplayerHM 2., der läuft noch am stabilsten.
- PGELua, die Phoenix Game Engine (von InsertWittyName und Mk2k) in Verbindung mit Lua. Momentan der beste Lua Interpreter den es gibt. Er ist enorm schnell und bietet enorm viele klasse Funktionen bei sehr viel Stabilität und Sicherheit. Achtung: die Syntax des PGELua sieht ein wenig anders aus als die der "gewöhnlichen" Luaplayer.

1.4 Nützliche Tools

Notepad++...
...die wohl beste alternative zum Notepad Editor, unterstützt Syntax Highlighting und ist überhaupt sehr gut zum coden geeignet.
PBP Unpacker...
...dient zum Extrahieren des Inhalts von Eboots oder kann auch komplett neue Eboots erstellen.

2. Die Arbeit mit dem LuaPlayer

2.1. Hallo Welt

Nun beginnen wir mit unserem Programm. Dabei soll der Text "Hello world" auf dem PSP Bildschirm dargestellt werden. Das Programm ist auch bekannt als "Hello World"-Programm und ist so ziemlich immer der erste Schritt in einer Programmiersprache.

Der Befehl zum Ausgeben von Text auf dem Bildschirm lautet
screen:print(x,y, "dein Text hier", farbe)

Zum Einstieg nun einmal der komplette Code des "Hello world"-Programms:

```
farbe = Color.new(255,255,255)

screen:print(1,1,"Hallo welt", farbe)
screen.flip()
screen.waitVblankStart(300)
```

Tippe den Code vollständig ab und speichere ihn als script.lua im LuaPlayer-Ordner ab.

Starte dann den LUAPlayer und bewundere das Ergebnis: ein weißes "Hallo welt" auf schwarzem Hintergrund...

Jetzt zur Erklärung:
screen:print gibt an, welcher Text mit welcher Farbe wo dargestellt werden soll. "1,1" steht dabei für die x und y-koordinaten auf dem PSP -Bildschirm. Danach folgt der auszugebende Text, hier ist es "Hallo welt". Zu guter letzt müssen wir dem LUAPlayer noch mitteilen, welche Farbe die Schrift haben soll. Ohne Farbe geht es gar nicht. Darum muss man vor dem screen: print noch eine Farbe definieren, mit dem Befehl Color.new(rot,grün,blau). Rot, Grün und Blau müssen selbstverständlich als Zahlenwerte angegeben werden. Rot, Grün und Blau bilden zusammen eine Farbe. 255,255,255 ist dann zum Beispiel Weiß. Du kannst die Werte beliebig ändern.

Nach dem screen:print folgt dann der Befehl screen.flip(). Was hat es damit auf sich? Nun, screen:print sagt nur aus, wo und wie der Text dargestellt werden SOLL. screen.flip() klatscht es dann auf den Bildschirm. Ohne screen.flip() läuft in LUA nichts. Diesen Befehl braucht man immer.

Am Ende ist dann noch ein screen.waitVblankStart(300). Das gibt an, wie lange alles auf dem Bildschirm angezeigt werden soll, bevor das Script endet. 300 sind 5 Sekunden. 60 Einheiten in LUA entspricht einer Sekunde. Sollte der Text nur 2 Sekunden angezeigt werden, müsste 120 in der Klammer stehen.

Jetzt wirst du dich wahrscheinlich fragen, wie man es macht, dass der Text für
immer angezeigt wird. Dazu gibt es einen Codeblock mit einer while-schleife. Ich möchte dazu aber nicht mehr erklären oder sagen, da ich dich nicht verwirren möchte. Wie genau das funktioniert wird später erläutert.

Das "Hello world"-Programm sieht dann so aus:

```
farbe = Color.new(255,255,255)

screen:print(1,1,"Hallo welt", farbe)
```

```
while true do
	screen.flip()
	screen.waitVblankStart()
end
```

Wir haben die beiden Befehle screen.flip() und screen.waitVblankStart() in eine while-schleife gepackt. Das ganze funktioniert denkbar einfach: Die while-Schleife wird durchgehend wiederholt, daher werden die Befehle extrem schnell aufeinanderfolgend ausgeführt. Wenn es nach dem screen.waitVblnakStart() ginge, würde der Text " Hallo welt" nicht einmal eine Sekunde angezeigt werden. Da es aber in der while-Schleife steht, die in einem wahnsinnigen Tempo durchgehend wiederholt wird, wird "Hallo Welt" in einer Sekunde so oft angezeigt, dass es für unser Auge so aussieht, als ob das Bild durchgehend "stehen" würde. In Wirklichkeit wird der Text ganz schnell hintereinander ausgegeben, bei jedem neuen Durchlauf der Schleife...

2.2. Variablen

Variablen braucht man in jeder Programmiersprache, man kann Werte aus ihnen auslesen oder sogar Werte in ihnen speichern. Das kann sehr nützlich sein. In LUA sind Variablen gleich doppelt einfach zu benutzen, denn man muss ihren Typ nicht mitangeben. Soll heißen, mann muss nicht festlegen, welche Wertemenge sie einschliessen. Zudem, kann man in LUA normale Variablen kinderleicht in Zeichenketten, auch "Strings" genannt, umwandeln. Beginnen wir doch gleich einmal mit einem String:

```
farbe = Color.new(255,255,255)
text1 = "Hallo"
text2 = "Welt"

while true do

	screen:print(1,1,text1.." "..text2, farbe)
	screen.flip()
	screen.waitVblankStart()
end
```

Dieser Code gibt ebenfalls "Hallo Welt" auf dem PSP Bildschrim aus (wer hätte das gedacht ?!^^). Allerdings sind die auszugebenden Zeichketten nun in Variablen gespeichert. Die werden nun ausgelesen. text1 und text2 sind die Variablen. Sie beinhalten jeweils eine Zeichenkette bzw String. Strings müssen immer in Anführungszeichen stehen, damit der LUAPlayer weiss, es handelt sich dabei um eine Zeichenkette. Ohne Anführungszeichen würde der LUAPlayer meinen, dass z.B. die Variable text1 gleich der Variable Hallo ist. Er wird aber eine Fehlermeldung ausgeben, weil die Variable Hallo gar nicht vorhanden oder gar nicht definiert ist. Hier haben wir gleich mehrere Fliegen mit einer Klappe geschlagen: Du hast du gelernt, wie man Stringvariablen definiert, und wenn du dir mal screen:print anschaust, auch gleich, wie man 2 Strings innerhalb eines Befehls verbindet. Jetzt wollen wir aber "Hallo Welt" einmal hinter uns lassen und noch eine Sorte anderer Variablen anschauen:

```
farbe = Color.new(255,255,255)
a = 5
b= 3
c=a*b

while true do

	screen:print(1,1,a*b+c, farbe)
	screen.flip()
	screen.waitVblankStart()
end
```

a,b und c sind jetzt einmal "normale" Variablen. Sie beinhalten einen richtigen Wert. a beinhaltet 5, b beinhaltet 3 und c beinhaltet a*b, also 5 mal 3. Im screen:print() befehl haben wir dann einen kleinen Term, nämlich a*b+c, was das Gleiche ist wie a*b+a*b. Dort sollte am Ende dann 30 stehen.

2.3. Schleifen und Blöcke

Schleifen gehören ebenso zu den Dingen, um die man in LUA nie herumkommen wird, sind aber sowieso recht einfach. Zu den Schleifen & Blocks in LUA gehören die if-Blocks, die for-Schleifen, die while-Schleifen und die repeat-until-Schleifen. Ich werde hier nur näher auf die if-Blocks eingehen, aber trotzdem kurz erklären, welche Schleife was bedeutet:

```
for a=1,5 do
	screen:print(0,0,"a ist kleinergleich 5 und groeßergleich 1 !", farbe)
end
```

->für z.B eine Zahl a von 1 bis 5 tue blablabla...

```
while a==2 do
	screen:print(0,0"a ist 2",farbe)
end
```

->während a gleich 2 ist, tue blablabla...(der Unterschied zu den if-blocks: die while-schleife wird mehr als einmal durchlaufen, sie kann sogar dauerhaft wiederholt werden. Die if-blocks gelten außerhalb von Schleifen nur einmal)

```
repeat a = a + 1 until a = 5 end
```

->wiederhole a ist a + 1 bis a gleich 5 ist...

Das waren ein paar kleine Beispiele. Jetzt zum if-Block, den man am häufigsten braucht:

```
farbe = Color.new(255,255,255)
a=5
b=3
c=a*b

while true do
	if a*b+c == 30 then
		screen:print(1,1,a*b+c, farbe)
	end
	if a*b+c > 30 then
	screen:print(1,1,"da stimmt was nicht", farbe)
	end

screen.flip()
screen.waitVblankStart()
end
```

Die Bedeutung ist denkbar einfach: Falls a*b+c gleich 30 ist, wird das Ergebnis 30 auf dem PSP-Bildschirm ausgegeben. Falls a*b+c > 30 sein sollte, was aber nicht der Fall ist, erscheint der Text "da stimmt was nicht"... Es gibt noch andere Wege, das auszudrücken, man kann das alles in eine if-Schleife packen:

```
if a*b+c == 30 then
	screen:print(1,1,a*b+c, farbe)
else
	screen:print(1,1,"da stimmt was nicht", farbe)
end
```

Hier wird für alle Werte ungleich 30 der Text ausgegeben "da stimmt was nicht". Es gibt aber noch eine andere Methode:

```
if a*b+c == 30 then
	screen:print(1,1,a*b+c, farbe)
elseif a*b+c > 30 then
	screen:print(1,1,"da stimmt was nicht", farbe)
end
```

..."elseif" ermöglicht sogar mehrere Alternativen innerhalb eines Blocks.

2.4. Tasten

Jetzt kannst du schon beachtlich viel, du hast schon eine ganze Menge an Code hinter dir...aber so ein richtiges Programm kann man ja noch nicht machen, oder? Ohne Tasten geht es einem ziemlich schlecht...
Dem können wir Abhilfe schaffen. Denn in LUA ist das auch wieder ziemlich leicht. So wie alles andere auch.
Zuerst, was man zur Tasten-Benutzung wissen sollte:
Mit pad = Controls.read() öffnet man sozusagen die Schnittstelle zu den Tasten. Es gibt folgende Tasten:

```
pad:cross()
pad:triangle()
pad:circle()
pad:square()
pad:up()
pad:down()
pad:left()
pad:right()
pad:l()
pad:r()
pad:start()
pad:home()
pad:select()
pad:note()

pad:analogX()
pad:analogY()
```

Wie du bemerkt haben solltest, ist "pad" nur eine Variable. Wir hätten es auch so vereinbaren können:

```
müsli = Controls.read()
```

dann würde die Tastenfunktion dazu so heißen:

```
müsli:cross()
```

...klingt jetzt wahrscheinlich erst mal lustig, aber das sollte man wissen. Es handelt sich ja doch nur um eine Variable. Anstelle von "müsli" hätte man auch "Apfelbaum" oder "Toastbrot" nehmen können...oder was auch immer du willst.
Am meisten Sinn macht allerdings immer noch pad. Da weiß man sofort, was gemeint ist...
Die Tastenfunktionen sind extrem wichtig. Ein Programm ohne Eingabe- und Manipulation ist nämlich langweilig.
Stell dir nur einmal ein PC-Programm vor, wo man weder Maus noch Tastatur nutzen kann. Das wäre auch sinnlos. Aber genug um den heißen Brei geredet. Wagen wir uns an den Code...

```
black = Color.new(0,0,0)
white = Color.new(255,255,255)

while true do
	screen:clear(black)
	pad = Controls.read()

	if pad:cross() then
		screen:print(10,10,"Es geht!",white
	end

	screen.flip()
	screen.waitVblankStart()
end
```

Wissen sollte man auch, dass Controls.read() nur in Verbindung mit einer Schleife Sinn macht. Da die ständig wiederholt wird, werden auch so ständig die Tasten abgefragt. Ausserhalb einer Schleife würden die Tasten nur ein einziges Mal abgefragt. Nach der Abfrage würde dann auchs chon der weitere Code folgen, und mit der Abfrage kann man gar nichts mehr anfangen, weil das Controls.read() längst verjährt ist. Also, Controls.read() immer in Verbindung mit einer Schleife ;)

2.5. Bildausgabe

Soweit so gut, mit deinen bisherigen Kenntnissen solltest du mittlerweile in der Lage sein, ein eigenes Programm mit LUA zu schreiben. Aber ein Programm nur mit Schrift? Das ist doch ein wenig arg langweilig...viel schöner wären doch auch Bilder. Dann könntest du deine Anwendung auch grafisch gestalten. Und das geht natürlich auch.

Der Befehl zur Bildausgabe sieht so aus:

```
screen:blit(x,y,bild, transparent=ja/nein[true/false])
```

Der ähnelt dem der Textausgabe natürlich sehr. Die ersten beiden Variablen x und y sind wieder die X&Y-Koordinaten, auf denen das Bild angezeigt werden soll. Danach folgt der Name des Bildes. Und zuguterletzt noch die Einstellung, ob das Bild (nicht) transparent ist. Der Befehl selbst ist genaugenommen noch ein wenig komplexer, es gibt noch mehr Parameter, die lassen wir aber ersteinmal ausser acht.

Hier folgt einmal ein Beispielcode zur Ausgabe eines Bildes:

```
bild = Image.load("wasweissich.png")

while true do
	screen:blit(0,0,bild,true)

	screen.waitVblankStart()
	screen:flip()
end
```

Ist nicht sonderlich kompliziert, oder? Und lass dich ja nicht von diesem true verwirren. Das sagt einfach nur aus, dass das Bild transparent angezeigt werden soll. Bei einer Anzeige ohne Transparenz würde dort nun "false" stehen müssen...oder man lässt diesen letzten Parameter einfach weg ;)

2.6. Funktionen

Da du jetzt alles Grundlegende durchgekaut hast, können wir uns nun an das "optionale" wagen: die Funktionen.
So richtig "optional" sind sie nun doch nicht. Aber für einfache Anwendungen braucht man sie dann auch nicht unbedingt.
Mit Funktionen kann man den Code übersichtlicher und etwas aufgeräumter gestalten. Und man kann sich mit ihrer Hilfe auch viele Zeilen Code sparen. Eine Funktionen beinhaltet einen bestimmten Code, der irgendetwas bewirkt. Wenn man zum Beispiel in einer Anwendung mit verschiedenen Tasten immer ein und den selben Code ausführen will, müsste man ja jedesmal für jede Taste den Code schreiben. Mit Funktionen muss man diesen Code nur einmal schreiben.
Man ruft die Funktion halt dann einfach mit den entsprechenden Tasten auf.

Vielleicht ist es verständlicher, wenn ich mal ein konkretes Codebeispiel bringe:

```
bild = Image.load("bild.png")

function Viereck()
	screen:blit(0,0,bild,false)
end

while true do
	pad = Controls.read()
	if pad:cross() then
		Viereck()
	end
	if pad:square() then
		Viereck()
	end

	screen.flip()
	screen.waitVblankstart()
end
```

Das sollte mal meine die Funktionsweise veranschaulichen. Bis hierher sollte das ganze doch sehr leicht verständlich sein. Der Hauptcode einer Funktion lautet also so:

```
function name()
	dein code hier
end
```

Ganz einfach oder? Es geht aber natürlich auch etwas komplizierter:

```
bild1 = Image.load("bild1.png")
bild2 = Image.load("bild2.png")
bild3 = Image.load("bild3.png")

function Viereck(x,y,bild)
	screen:blit(x,y,bild,false)
end

while true do
	pad = Controls.read()
	if pad:cross() then
		Viereck(0,0,bild1)
	end
	if pad:circle() then
		Viereck(150,150,bild2)
	end
	if pad:triangle() then
		Viereck(300,120,bild3)
	end

	screen.flip()
	screen.waitVblankstart()
end
```

Das dürfte beim ersten Überfliegen des Codes ersteinmal für Verwirrung sorgen. Ist aber alles halb so wild. Am Anfang laden wir einfach 3 verschiedene Bilder in den RAM-Speicher der PSP. Dann folgt unsere eigentliche Funktion mit dem Namen Viereck. Der Funktionscode ist hier screen:blit(x,y,bild,false). Das sollte keine Probleme bereiten. x und y sind die beiden Koordinaten, bild steht für das anzuzeigende Bild, und Transparenz ist deaktiviert (weil false). Aber warum steht in den Klammern hinter "Viereck" so komisches Zeug? Das ist grob gesagt nichts anderes wie ein "Variablenwerte-Übermittler". Die Funktion wird ja z.B. mit der X-Taste aufgerufen. Dabei soll ein Bild angezeigt werden. Dieses Bild braucht aber ja auch richtige Koordinaten und ein richtiges, vorhandenes Bild. Mit "Viereck(0,0,bild1)" weisen wir den Variablen "x","y" und "bild" Werte zu. So würde im Falle eines X-Tastedrucks das Bild "bild1.png" auf den Koordinaten 0 und 0 angezeigt. Wie muss man also die Variablen innerhalb einer Funktion verstehen? Man legt schlicht und einfach fest, welche Werte man in die Funktion an welcher Stelle einsetzen kann. Diese Werte können dann innerhalb der Funktion genutzt werden.

Achtung: Die Werte gelten auch bloß innerhalb der Funktion und sind nicht außerhalb gültig.

Ich hoffe ihr habt es verstanden^ ^...bei Problemen unf Fragen bitte ins Forum (http://www.bumuckl.com) posten. Es gibt dann schnellstmöglich Hilfe..

2.7. Arbeiten mit Dateien

Für ein hochwertiges Programm braucht man meist auch andere Dateien,sei es der Übersicht halber oder aus reiner Funktionalität. Aufjedenfall gibt es dafür wieder einige verschiedene Methoden. Wenn man einfach nur eine weitere Datei, die LUA-Code enthält, ins Script einbinden will, dann eignet sich "dofile" ganz gut. Dieser Befehl muss dann an gewünschter stelle eingebunden werden:

```
dofile ("hallo.lua")
```

...bei mir könnte ich z.B. in meiner script.lua auf die Datei"hallo.lua", die sich im selben Ordner befindet, zugreifen. In "hallo. lua" könnten dann z.B Funktionen enthalten sein. Da dofile aber etwas "buggy" ist, also auch zu Programmabstürzen führen kann, bevorzuge ich diese Methode:

```
datei = loadFile("hallo.txt")
datei()
```

...dieser kleine Codeblock ist stabiler als "dofile". Mit loadFile() lade ich erst eine Datei und erstelle dabei eine Art Funktion, die ich sofort danach mit "datei()" ausführe. Die eleganteste Methode meiner Meinung nach ist allerdings folgende:

```
require "datei"
```

Damit wird eine .lua Datei namens "datei" ins Script miteingebunden. Wichtig ist dabei, dass das nur mit .lua-Dateien oder mit so genannten Libraries funktioniert und dabei die Dateiendung nicht mitangegeben werden darf.

Mit diesen Methoden kann ich also Datein einlesen, aber nur funktionierenden, fehlerfreien LUA-Code. Wenn man jetzt selber Dateien erstellen will oder bestimmte Dateiformate mit genauer Zeile auslesen will, gibt es folgende Methode:

```
datei = io.open("huhu.txt", "r")
zeile = datei:read()
datei:close()
```

...hierbei wird die erste Zeile der Datei "huhu.txt" gelesen. Beim nächsten "datei:read()" wird die folgende Zeile gelesen usw. Für "datei:read()" gibt es aber auch Parameter (Werte) für innerhalb der Klammer, die wie folgt aussehen:

```
*n - Liest eine Zahl ein. Bsp.: datei:read("*n")
*a - Liest die ganze Datei ab der Momentanen Position, bsp.: datei:read("*a")
*l - Liest die nächste Zeile, ist standardmäßig auch ohne Parameter eingestellt., bsp.: datei:read("*l")
zahl - Liest eine Zeichenkette mit der angegebenen Zeichzahl ein, bsp.: datei:read(5)
```

Jetzt kann man Dateien aber nicht nur lesen, sondern auch schreiben. Dazu vorweg eine kleine Liste mit den dazugehörigen Commands. Wer sich vorher gefragt hat, was das "r" in der Klammer von io.open soll, erfährt es jetzt:

```
r - Lesemodus (read mode)
w - Schreibmodus, überschreibt kompletten Inhalt (write mode)
a - Fügt zum existierenden Inhalt hinzu (append mode)
b - Binärmodus (binary mode)
r+ - Updatemodus (existierende Daten bleiben)
w+ - Updatemodus(existierende Daten bleiben)
a+ - Zufüg -und Updatemodus (existierende Daten bleiben, fügt nur am Ende hinzu)
```

Wie kann man dann etwas in eine Datei schreiben?

```
datei = io.open("huhu.txt","w")
datei:write("hallo")
datei:close()
```

Zuguterletzt noch eine kleine Funktion von mir, mit der man alle Zeilen einer Datei auslesen kann. Die Werte werden in einer Table gespeichert:

```
function readAllLines(datei)
	zeilen = {}
	datei = io.open(datei,"r")
		for line in datei:lines() do
			zeilen[line] = line
		end
	datei:close()
end
```

Wer alle Zeilen auslesen will und dann noch gleich anzeigen lassen will kann das mit dieser Funktion machen:

```
function printAllLines(datei,color)
	y=10
	zeilen = {}
	datei = io.open(datei,"r")
		for line in datei:lines() do
			zeilen[line] = line
			y=y+10
```

```
                screen:print(0,y,zeilen[line],color)
            end
        datei:close()
end
```

Zu beachten ist hier besonders, dass die Filehandling-Methoden nicht generell io.open / io.close etc. sind. Die Befehle sind auch abhängig vom jeweiligen Luaplayer.

2.8. Musik und Sound

Wer ein Spielchen programmiert, wird wohl nicht auf Musik oder Sounds verzichten können. Deshalb kümmern wir uns hier darum, wie und wo man Sounds oder Musik einbindet. Zudem gibt es auch gleich ein kleines Tutorial, wie man in das gewünschte Format konvertiert. Vorweg sei aber gesagt, dass so ziemlich jeder Luaplayer andere Befehle für Musik und Sound hat, sowie auch andere Dateiformate unterstützt.

Der "normale" LUAPlayer (hier speziell Luaplayer Mod1-4 von Cools) kann folgende Formate abspielen:
UNI, IT, XM, S3M, MOD, MTM, STM, DSM, MED, FAR, ULT oderr 669 für Musik. WAV kann nur für Sounds verwendet werden.
Die obigen Formate können direkt aus dem MIDI-Format konvertiert werden. Dazu gibt es ein schönes Programm namens MadTracker, erhältlich auf http://www.madtracker.org.
Nun zum eigentlichen Code (Achtung, diese Befehle gelten NUR für die Luaplayer Mod1-4 von Cools)

```
Music.playFile("lied.xm", true)
```
...spielt Musik ab, true/false aktiviert/deaktiviert den Loop-Modus.
```
Music.pause()
```
...pausiert ein Lied, also hält es nur an.
```
Music.stop()
```
...beendet ein Lied.
```
Music.resume()
```
...setzt die Wiedergabe fort nach der Pausierung.
```
Music.playing()
```
...ermittelt, ob gerade ein Lied läuft.
```
Music.volume(Zahl)
```
...legt die Lautstärke fest, Zahl kann einen Wert von 0 bis 128 enthalten.

Soviel zum Abspielen von Musik, wollen wir uns doch nun um Sounds kümmern. Wie schon gesagt, für Sounds eignet sich nur das Format WAV. Ins WAV-Format kann man z.B. mit Audacity konvertieren, dass es auf http://audacity.sourceforge.net gibt.

```
sound = Sound.load("bing.wav",false)
```
...lädt eine WAV-Datei, der Loop ist deaktiviert(false!)
```
sound:play()
```
...spielt eine Sounddatei ab.
```
sound:stop()
```
...beendet die Wiedergabe der Sounddatei
```
sound:pause()
```
...pausiert die Soundwiedergabe
```
sound:resume()
```
...setzt die Soundwiedergabe fort
```
sound:playing()
```
...ermittelt, ob der Sound gerade wiedergegeben wird.

Speziell beim LUAPlayer Mod 4 kann man aber dann doch MP3 abspielen mit den folgenden Befehlen:

```
Mp3.load("song.mp3")
```
...lädt eine MP3-datei
```
Mp3.play()
```
...spielt den MP3-Song ab
```
Mp3.pause()
```
...pausiert das Abspielen
```
Mp3.stop()
```
...stoppt das Abspielen und löscht den Song aus dem RAM

Mp3.EndOfStream()
...ermittelt, ob der Song vorüber ist, gibt true oder false zurück
Mp3.getTime()
...zeigt die MP3-Spielzeit an, Anzeige mittels einem String (Zeichenkette)
Mp3.volume(zahl)
...stellt die MP3-Lautstärke ein

Das waren jetzt spziell die Funktionen des mittlerweile veralteten LuaPlayer Mod4 von Cools. Für neuere Firmwares benötigt man natürlich dann den LuaPlayerHM, ab Version 2 ist dieser auch Audio-tauglich. Version 2 hat zwar Befehle für Mp3, Sounds im .wav-Format können erst ab LuaPlayerHM 7 wieder problemlos benutzt werden.

Mp3/Ogg-Befehle (Standard Mp3-Funktionen) sind:

```
Mp3.load("datei.mp3")      --Achtung, es kann maximal eine Mp3 geladen und zur selben Zeit abgespielt werden
Mp3.stop()
Mp3.pause()
Mp3.play()                 --braucht keine Parameter, da sowieso nur eine Mp3 abgespielt werden kann
Mp3.EndOfStream()
Mp3.getTime()
Mp3.songTime()
Mp3.artist()
Mp3.title()
Mp3.album()
Mp3.genre()
Mp3.year()
Mp3.trackNumber()
Mp3.layer()
Mp3.kbit()
Mp3.mode()

Ogg.load()                 --Achtung, es kann maximal eine Ogg geladen und zur selben Zeit abgespielt werden
Ogg.stop()
Ogg.pause()
Ogg.play()                 --braucht keine Parameter, da sowieso nur eine Ogg abgespielt werden kann
Ogg.EndOfStream()
Ogg.songTime()
Ogg.artist()
Ogg.title()
Ogg.album()
Ogg.genre()
Ogg.year()
Ogg.trackNumber()
Ogg.layer()
Ogg.kbit()
Ogg.mode()
```

Dazu gibt es noch eine spezielle Mp3-Bibliothek, die um einiges stabiler und schneller läuft und daher bevorzugt zu verwenden ist:

```
Aa3me.load()               --Achtung, es kann maximal eine Aa3 geladen und zur selben Zeit abgespielt werden
Aa3me.play()               --braucht keine Parameter, da sowieso nur eine Aa3 abgespielt werden kann
Aa3me.stop()
Aa3me.eos()
Aa3me.gettime()
Aa3me.percent()
Aa3me.pause()
Aa3me.songTime()
Aa3me.artist()
Aa3me.title()
Aa3me.album()
Aa3me.genre()
Aa3me.year()
```

```
Aa3me.trackNumber()
Aa3me.layer()
Aa3me.kbit()
Aa3me.mode()
Aa3me.rawSongTime()
Aa3me.instantBitrate()
Aa3me.vis()

Mp3me.load()            --Achtung, es kann maximal eine Mp3 geladen und zur selben Zeit abgespielt werden
Mp3me.play()            --braucht keine Parameter, da sowieso nur eine Mp3 abgespielt werden kann
Mp3me.stop()
Mp3me.eos()
Mp3me.gettime()
Mp3me.percent()
Mp3me.pause()
Mp3me.songTime()
Mp3me.artist()
Mp3me.title()
Mp3me.album()
Mp3me.genre()
Mp3me.year()
Mp3me.trackNumber()
Mp3me.layer()
Mp3me.kbit()
Mp3me.mode()
Mp3me.rawSongTime()
Mp3me.instantBitrate()
Mp3me.vis()
```

Für Sounds speziell im .wav-Format ersteinmal ein Befehl ausgeführt werden, der das Abspielen aktiviert:

```
System.oaenable()
```
Aktiviert die Wiedergabe von Sounds

```
System.oadisable()
```
Deaktiviert die Wiedergabe von Sounds

Zwischen diesen beiden Befehlen können dann Sounds wiedergegeben werden:

```
Sound.play(meinsound)
```

Der "Sound" meinsound muss aber natürlich zuvor geladen werden:

```
meinsound = Sound.load("achso.wav")
```

Weitere Befehle wären:

```
SoundSystem.SFXVolume()
SoundSystem.reverb()
SoundSystem.panoramicSeparation()
sound:gc()
sound:tostring()
```

Im Laufe der Zeit werden immer mehr Sound-Funktionen hinzukommen bzw evt. auch hinzugekommen sein. Für nähere Infos kann man dann einfach in der readme.txt des jeweiligen LuaPlayers nachlesen, oder auch im Entwicklerforum von

http://luaplayerhm.xtreemhost.com

nachsehen. Selbstverständlich darf auch im Forum auf http://www.bumuckl.com nachgefragt werden.

2.9. Timers

Für manche Games kann man auch einen Timer gut gebrauchen, SuperMario ist ein gutes Beispiel. Timer sind recht einfach zu benutzen.
Aber auch wie bei den meisten anderen LuaPlayer-abhängigen Befehlen muss man hier beachten, dass die Befehle teilweise etwas anders lauten können. Hier wird die Funktion von Timern anhand der LuaPlayer Mod 4 Funktionen erläutert.

Zuerst erstellen wir einen neuen Timer:

```
zaehler = Timer.new()
```

Wenn der Timer nun gestartet werden soll, ergänzen wir folgenden Befehl:

```
zaehler:start()
```

Man kann auch die Zeit des Timers auslesen und anzeigen lassen:

```
runtime = zaehler:time()
```

Zuguterletzt kann man den Timer auch zurï¿½cksetzen:

```
zaehler:reset(zahl)
```

...oder auch stoppen:

```
zaehler:stop()
```

Wie kann man den Timer nun einbauen und anwenden? Dafür schreiben wir doch einfach mal eine kleine Anwendung:

```
color = Color.new(255,255,255)
zaehler = Timer.new()

while true do
zaehler:start()
runtime = zaehler:time()
screen:print(0,0,runtime,color)

if runtime > 1000 then
zaehler:reset(0)
end

screen.flip()
screen.waitVblankStart()
end
```

Um sicherzustellen, dass es die Timerfunktionen auch in deinem LuaPlayer gibt, genügt ein Blick ins LuaWiki ;)

2.10. Tables

Bei Tables denkt sich so manch einer "Für was sollen die bitte gut sein?". Ich hab mir das zu Beginn meiner Lua-Karriere zumindest gedacht. Mit der Zeit wird man aber nicht nur Älter, sondern auch klüger. Damit meine ich, dass man irgendwann ihren Sinn und ihre Nützlichkeit begreift.

Zuallererst bekommt man immer wieder eingetrichtert, dass sie etwas Übersicht in den Code bringen und man deshalb mit "tables" arbeiten sollte. Nun, ich arbeitete generell gern unübersichtlich und total durcheinander, drum kamen "tables" bei mir gar nicht in Frage. Bis ich dann begriffen habe, dass man mit ihnen noch viel mehr machen kann. Man kann nach sie nämlich, im Gegensatz zu if-Blöcken, mittels for-Schleifen komplett durchlaufen lassen. Sprich man kann automatisiert jeden Wert auslesen und mit ihm anstellen, was man will. Das spart viel Zeit und ist in manchen Fällen sogar zwingend erforderlich.

Wie sehen Tables denn eigentlich aus?

```
table = {}
```

...das wäre jetzt mal eine leere "table" ohne Einträge. Nun füllen wir sie doch einfach mal ein wenig:

```
table = {"hallo","," du","lernst ","gerade ","tables ","kennen."}
```

...nun könnten wir den Inhalt der "table" darstellen mit screen:print():

```
color= Color.new(255,255,255)
table = {"hallo,", " du", "lernst ", "gerade ", "tables ", "kennen."}
screen:print(0,0,table[1]..table[2]..table[3]..table[4]..table[5]..table[6],color)
```

...dann würde auf dem Bildschirm stehen "hallo, du lernst gerade tables kennen". Nun gut, das ist noch nicht besonders praktisch. Aber jetzt steigen wir mal tiefer in die Materie ein, ich liste hier jetzt mal alle wichtigen Befehle zur Arbeit mit "tables" auf:

```
table.concat (table , zwischentext , ab wann , bis wo)
...verbindet die Werte einer table.
table.foreach (table, funktion)
...geht alle Felder einer table durch und übergibt sie ggf an eine Funktion.
table.getn (table)
...ermittelt die Anzahl der Tablefelder.
table.sort (table , funktion)
...sortiert eine table. Ist keine Funktion gegeben wird nach groesser/kleiner Prinzip sortiert.
table.insert (table, position, wert)
...fügt ein Feld in eine vorhandene table ein.
table.remove (table , position)
...loescht ein Feld einer table
table.setn (table, felderanzahl)
...setzt die Anzahl der Tablefelder.
```

Da wir das jetzt wissen, möchte ich euch nun zeigen, warum "tables" so praktisch sind. Mit einer for-schleife kann man alle Tablefelder einmal durchgehen und sie z.B. nacheinander bzw untereinander anzeigen lassen:

```
limit = table.getn(table)
y = 0
for i=1,limit do
y=y+10
screen:print(0,y,table[i],color)
end
```

...lasst euch nicht von "i" verwirren. "i" ist eine Variable, die hier mit dem Wert 1 versehen wird. Bei jedem Neudurchlauf des Scriptes wird "i" dann um 1 größer. Das kann man in for-Schleifen ausnutzen. Man könnte anstatt i auch "klabautermann" verwenden, nur zur Veranschaulichung. Dieser kleine Codeblock tut das gleiche wie:

```
screen:print(0,0,table[1],color)
screen:print(0,10,table[2],color)
screen:print(0,20,table[3],color)
screen:print(0,30,table[4],color)
screen:print(0,40,table[5],color)
screen:print(0,50,table[6],color)
```

...hier ist das nicht so gravierend, aber wenn man mal 100 Tablefelder hat, ist die for-Schleife die elegantere Lösung.

2.11. IrDA

Die PSP hat ja bekanntlich auch einen Infrarotport. Und mit dem kann man auch so allerhand Sachen anstellen. Was damit alles möglich ist zeigt IrDA Games von alex705 (http://www.rabbitcoder.de). Man kann spannende Multiplayerspiele erstellen, oder auch multifunktionaleHomebrews. Und es ist sogar ziemlich einfach. Zu beachten ist lediglich, dass die IrDA-Funktionen vom jeweiligen LuaPlayer abhängen. Dies hier sind die IrDA-Funktionen des LuaPlayer Mod4 von Cools.

Zuallererst muss einmal das Ir-Modul aktiviert werden:

```
System.irdaInit()
```

Da das Modul nun aktiviert ist, kann man auch sofort loslegen. Man kann z.B. Signale von anderen Geräten, wie z.B. Fernbedienungen empfangen, oder auch mit anderen PSPs kommunizieren.

Arbeiten mit Fremdgeräten(z.B. Fernbedienung):
Die PSP muss Signale vom Fremdgerät bekommen. Falls dann ein Signal kommt, kann man dann mittels if-Blöcken die Aktionen Regeln. Ein kleines Codebeispiel:

```
color=Color.new(255,255,255)
add = 0
System.irdaInit()

while true do

remote = System.irdaRead()
remotelength = string.len(remote)
screen:print(0,0,add,color)

if remotelength>1 then
add = add+1
end

screen.flip()
screen.waitVblankStart()
end
```

**Zuerst wird das Ir-Modul initialisiert und wir legen unsere Variablen fest. Im Mainloop wird der Ir-Port dann ständig überwacht bzw eingelesen. Somit wartet er auf Signale. Gleichzeitig ermittelt eine andere Variable die Länge der mit dem Signal übermittelten Zeichenkette. Wenn die Zeichenkette aus mehr als 1 Zeichen besteht, wird die Variable "add" vergrößert.
Und "add" geben wir auf dem Bildschirm aus. Somit wird bei jedem Tastendruck auf der Fernbedienung "add" vergrößert.**

Arbeiten mit anderen PSPs:
Das läuft auch nicht viel anders ab als mit einer Fernbedienung. Es ist genau das selbe Schema, nur muss eine andere PSP ein Signal senden. Dafür gibt es auch einen ganz simplen Befehl. Die Signale kann man auch wieder mittels if-Blöcke auswerten:

```
color=Color.new(255,255,255)
add = 0
System.irdaInit()

while true do
pad=Controls.read()

remote = System.irdaRead()
remotelength = string.len(remote)

screen:print(0,0,add,color)

if pad:cross() then
System.irdaWrite("hallo")
end

if remotelength>0 then
add = add+1
end

screen.flip()
screen.waitVblankStart()
end
```

Hierbei wird exakt dasselbe Ergebnis wie bei den Fernbedienungen erzeugt. Ihr seht schon, die Verwendung des Ir-Ports ist wirklich sehr einfach.

2.12. Wlan

Der Luaplayer bietet über den Ir-Port hinaus noch Befehle für den WLAN-Port. Somit kann man auch auf das Internet zugreifen bzw. Online arbeiten. Das kann man z.B. für ein Multiplayergame ausnutzen. Ich werde nun die einzelnen Wlan-

WLAN-Befehle nacheinander auflisten und erläutern. Das ganze natürlich in der Reihenfolge, wie sie in einem Script verwendet werden muss. Zu beachten ist wiederum, dass die Befehle je nach LuaPlayer unterschiedlich lauten. Dazu hilft dann auch die jeweilige readme.txt des Luaplayers weiter.

Wlan.init()
Initialisiert das WLAN-Modul. Dieser Befehl wird immer vor Gebrauch weiterer WLAN-Befehle benötigt.

Wlan.getConnectionConfigs()
Ermittelt alle verfügbaren Wifi-Verbindungskonfigurationen der PSP und speichert sie in einer table ab.
Bsp.: verbindungen = Wlan.getConnectionConfigs()
...das ergebnis davon ist dann:
verbindungen = {verbindung1,verbindung2,verbindung3}
...und wenn man dann alle Verbindungen auf dem Bildschirm anzeigen lassen will, und zusätzlich ein Auswahlmenü dazu haben will, der darf sich folgendes Script anschauen (ist nicht so kompliziert wie es aussieht):
http://www.bumuckl.com//index.php?option=com_content&task=view&id=257&Itemid=120

Wlan.useConnectionConfig(1)
Dieser Befehl sagt dem Luaplayer, welche Verbindung er benutzen soll. In diesem Fall will er Verbindung 1 benutzen. Vor weg muss ich allerdings erwähnen, dass es in Firmwares ab 3.xx eine kleinen Bug mit dem Luaplayer gibt. Wenn man Verbindung 1 benutzen will, muss man 2 eingeben. Für Verbindung 2 dann 3. Das gilt aber nur für Firmwares ab 3.xx. In früheren Firmwares funktioniert es ganz normal.

Wlan.getIPAddress()
Dieser Befehl ermittelt die IP-Adresse der PSP und speichert sie in einer Zeichenkette (String).
Bsp.:
```
ip = Wlan.getIPAddress()
screen:print(0,0,ip,farbe)
```
Im Falle, dass die WLAN-Verbindung fehl schlägt, kann man es alle 100 ms nochmal probieren:
```
if not Wlan.getIPAddress() then
    System.sleep(100)
end
```
Dieser if-Block kommt nur dann in Funktion, falls der Luaplayer die IP-adresse nicht ermitteln kann.

Socket.connect(host, port)
Erstellt eine neue Basis für Datenempfang und Versand mit einer direkten Verbindung zum Host über einen bestimmten Port. Der HTTP-Port, also für das WWW, läuft normalerweise über Port 80.
Bsp.: neuersocket = Socket.connect("http://lua.bumuckl.com", 80)

neuersocket:isConnected()
Dieser Befehl prüft, ob der Socket verbunden ist und kann genauso wie Wlan.getIPAddress() benutzt werden.
Bsp.:
```
if neuersocket:isConnected()==false then
    System.sleep(100)
end
```

neuersocket:recv()
Dieser Befehl liest die erhaltenen Daten des Sockets aus und speichert sie in einer Zeichenkette. Falls keine Daten vorhanden sind, ist auch die Zeichenkette leer.
Bsp.: data = neuersocket:recv()
Achtung: Der Socket erhält nicht von selbst Daten. Er muss erst einer Adresse mitteilen, dass er Daten empfangen kann bzw will.

neuersocket:send(zeichenkette)
Dieser Befehl sendet daten an unseren Host. Diese Daten müssendem Host allerdings mitteilen, dass er Daten zurücksenden soll.
Bsp.: neuersocket:send("GET /index.html HTTP/1.0\r\n")
GET /index.html teilt dem Host mit, dass wir die index.html haben wollen. Das HTTP/1.0\r\n sagt dem Host, dass wir das HTTP-Protokoll benutzen, also WWW.

neuersocket:close()
Dieser Befehl schliesst einen Socket bzw beendet ihn und damit auch die Verbindung zum Host.

Wlan.term()
Deaktiviert das WLAN-Modul und trennt somit alle WLAN-Verbindungen.

2.13. 3D Gu

Lua unterstützt sogar die Arbeit mit 3D Objekten, insofern im LuaPlayer eine "3D Gu" enthalten ist. Ich empfehle allerdings, dass Anfänger die Finger davon lassen, denn die Arbeit mit der 3D Gu ist recht schwer. Zuerst einmal sollte man wissen, dass es in 3D nicht nur die x und y Achsen gibt, sondern x,y, und z Achse. Also Höhe, Breite und Tiefe. Zudem kann man in der 3D Gu auch nicht von "blit" reden, da ja kein Bild in der 2 Dimensionalen Ebene dargestellt wird, sondern ein 3-dimensionales Objekt. Die Funktion hierfür heisst Gum.translate(x,y,z). Desweiteren läuft eine 3D-Darstellung auch etwas anders ab als eine 2-dimensionale.

Grundlegend kann man in jedes Lua-Script die 3D Gu, insofern der LuaPlayer es abspielen kann, einbetten. Die 3D Gu muss gestartet und auch wieder geschlossen werden:

```
Gu.start3d()
        --der Code für das Objekt
Gu.end3d()
```

Nun, wir müssen systematisch vorgehen. Zuallererst benötigen wir eine Farbe und die Farbtiefe für unser Objekt, dass wir noch gar nicht erstellt haben. Also weisen wir der 3D Gu eine Farbe und eine Farbtiefe zu:

```
Gu.clearColor(farbe)
...legt die Füllfarbe fest
Gu.clearDepth(0)
...legt die Farbtiefe fest, von 0 bis 255(0=kein Farbtiefe,255=größte Farbtiefe)
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
...weist dem Objekt die eingegebene Farbe und Farbtiefe zu.
```

Also sieht der Code bisher so aus:

```
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gu.end3d()
```

Nun, weiter geht es. Nachdem jetzt die Farbe und Farbtiefe festgelegt ist, müssen wir dem LUAPlayer mitteilen, wie wir auf das Objekt schauen werden. Das ganze nennt man auch "Kamera", also die Sicht auf das "Model" (Objekt). Das wird so festgelegt:

```
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
```

Dieser Codeblock legt die Kameraposition fest. Der Wert 50 legt fest, wieviel man von der 3D Gu sehen kann. 16/9 ist die Bilddarstellung. 0.5 legt fest, wie nah man das Objekt sehen kann. 1000 legt fest, bis in welche Entfernung man das Objekt sehen kann.

Folglich sieht der gesamte 3D Gu Code so aus:

```
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
Gu.end3d()
```

...das war es aber noch lange nicht, denn nun folgt, nachdem die Farbdarstellung und die Perspektive festgelegt wurde, der Anzeigemodus:

Jetzt sieht der Code so aus:

```
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
Gum.matrixMode(Gu.VIEW)
Gum.loadIdentity()
Gu.end3d()
```

...was jetzt noch fehlt ist/sind das/die eigentliche(n) Objekt(e). Und die sind im großen und ganzen nur am Anfang schwer. Wenn man das Konzept dahinter einmal verstanden hat, dürfte es nicht mehr schwer sein. Vorweg, es gibt 7 Arten von Formen & Figuren:

```
Gu.POINTS- zeichnet Punkte.
Gu.LINES- zeichnet Linien.
Gu.LINES_STRIP- Zeichnet aneinandergereihte Linien.
Gu.TRIANGLES- Zeichnet Dreiecke.
Gu.TRIANGLES_STRIP- zeichnet aneinandergehängte Dreiecke.
Gu.TRIANGLES_FAN- zeichnet ebenfalls aneinandergehängte Dreicke, allerdings an 2 andere Eckpunkte.
Gu.SPRITES- zeichnet Quader.
```

Zusätzlich können solche Objekte nur im Gu.matrixMode(Gu.MODEL) angezeigt werden. Das bedeutet, es gibt wieder ein Codeupdate, und wir nähern uns langsam dem Ziel!

```
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
Gum.matrixMode(Gu.VIEW)
Gum.loadIdentity()
Gum.matrixMode(Gu.MODEL)
Gum.loadIdentity()
Gum.translate(0, 0, -3)
Gu.disable(Gu.TEXTURE_2D)
Gu.end3d()
```

Ganz wichtig, Gum.translate legt die Position des Objektes im 3D-Koordinatensystem fest! In diesem Fall sind Texturen zusätzlichn deaktiviert, das Objekt bekommt nur Farbe. Soweit so gut, wollen wir doch endlich mal ein Objekt erstellen. Dafür sollte man wissen, wie Tables funktionieren.

```
triangle = {
        {red,3,-2,0}, --eckpunkt 1
        {red,-3,-2,-2}, --eckpunkt 2
        {red,3,3,-1}, --eckpunkt 3
}
```

Diese Table liefert nachher die Werte für das zu zeichnende Dreieck. Grundsätzlich muss die Table so aussehen:

```
table = {
        {farbe,x,y,z}
}
```

Beim Dreieck brauchen wir eben 3 von diesen "innertable Tables", daein Dreieck eben 3 Eckpunkte hat. Bei einem Punkt

Bei einem Punkt braucht man nur eines davon. Nun gut, da wir jetzt unser Dreieck haben, kommt der finale Schritt, nämlich das Stückchen Code in der Gu, dass unsere Table "triangle" verwendet, um ein Dreieck zu zeichnen:

```
Gum.drawArray(Gu.TRIANGLES, Gu.COLOR_8888+Gu.VERTEX_32BITF+Gu.TRANSFORM_3D, triangle)
```

...diese sehr kompliziert aussehende Zeile gibt unser Dreieck aus! Und so extrem kompliziert ist sie auch gar nicht. Der erste Teil in der Klammer "Gu.TRIANGLES" teilt dem LUAPlayer mit, dass ein
Dreieck gezeichnet werden soll. Für einen Punkt müsste hier "Gu.POINTS" stehen. Der zweite Teil sagt, dem LUAPlayer, dass wir uns im 32 Bit Modus befinden. Den zweiten Teil muss man nicht wirklich verstehen, den kann man einfach so stehen lassen. Im dritten Teil informieren wir den LUAPlayer, wo er die Eckpunkte des Dreiecks finden kann. Unsere Table oben heisst "triangle", darin befinden sich die Koordinaten der Eckpunkte. Und die verwenden wir auch hier. Hier also mal der komplette Code

```
red = Color.new(255,0,0)
triangle = {
	{red,3,-2,0}, --eckpunkt 1
	{red,-3,-2,-2}, --eckpunkt 2
	{red,3,3,-1}, --eckpunkt 3
}
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
Gum.matrixMode(Gu.VIEW)
Gum.loadIdentity()
Gum.matrixMode(Gu.MODEL)
Gum.loadIdentity()
Gum.translate(0, 0, -3)
Gu.disable(Gu.TEXTURE_2D)
Gum.drawArray(Gu.TRIANGLES, Gu.COLOR_8888+Gu.VERTEX_32BITF+Gu.TRANSFORM_3D, triangle)
Gu.end3d()
```

...das waren die Grundlagen der 3D Gu. Es gibt da jetzt noch ein paar weitere Funktionen, und Texturen darf man natürlich auch nicht vergessen:

```
Gum.rotateXYZ(x* (Gu.PI/180),y* (Gu.PI/180),z* (Gu.PI/180))
```
...rotiert ein Objekt nach den eingegebenen Werten. Der Befehl sollte nach Gum.translate() und vor Gum.drawArray() platziert werden. Werte kann man für x,y und z einsetzen.

```
Gum.lookAt(vonwox,vonwoy,vonwoz,nachx,nachy,nachz,0,1,0)
```
...ändert die Kameraposition von der Standardposition zu den eingegebenen Positionen. Das 0,1,0 am Ende muss man nicht verändern.

Und nun noch das Verwenden von Texturen:

```
Gu.enable(Gu.BLEND)
Gu.blendFunc(Gu.ADD, Gu.SRC_ALPHA, Gu.ONE_MINUS_SRC_ALPHA, 0, 0)
Gu.enable(Gu.TEXTURE_2D);
```

Diese 3 Befehle aktivieren die Verwendung von Texturen.

```
Gu.texImage(bild)
```
...verwendet das angegebene Bild als Textur.

Nun noch ein paar Zeilen Code, die man vorerst nicht verstehen muss und die man so lassen kann. Lediglich die Farbe "white" sollte auch richtig definiert sein:

```
Gu.texFunc(Gu.TFX_MODULATE, Gu.TCC_RGBA)
Gu.texEnvColor(white)
```

```
Gu.texFilter(Gu.LINEAR, Gu.LINEAR)
Gu.texScale(1, 1)
Gu.texOffset(0, 0)
Gu.ambientColor(white)
```

Das mit den Texturen geht ja nun auch recht schnell. Es muss dann nur noch eine kleine Änderung bei Gum.drawArray() vorgenommen werden.

```
Gum.drawArray(Gu.TRIANGLES, Gu.TEXTURE_32BITF+Gu.COLOR_8888+Gu.VERTEX_32BITF+Gu.TRANSFORM_3D,
table)
```

...hier sieht der 2. Parameter (Wert) in der Klammer etwas anders aus. Er wurde für die Verwendung von Texturen angepasst. Zur Übersicht und Veranschaulichung hier noch der gesamte Code mit Textur:

```
white = Color.new(255,255,255)
red = Color.new(255,0,0)
bild = Image.load("bild.png")
triangle = {
        {0,0,red,0,0,0},
        {1,0,red,1,0,0},
        {1,1,red,1,-1,0},

        {0,0,red,0,0,0},
        {0,1,red,0,-1,0},
        {1,1,red,1,-1,0}
}
farbe = Color.new(255,0,255)
Gu.start3d()
Gu.clearColor(farbe)
Gu.clearDepth(0);
Gu.clear(Gu.COLOR_BUFFER_BIT+Gu.DEPTH_BUFFER_BIT)
Gum.matrixMode(Gu.PROJECTION)
Gum.loadIdentity()
Gum.perspective(50, 16/9, 0.5, 1000)
Gum.matrixMode(Gu.VIEW)
Gum.loadIdentity()
Gu.enable(Gu.BLEND)
Gu.blendFunc(Gu.ADD, Gu.SRC_ALPHA, Gu.ONE_MINUS_SRC_ALPHA, 0, 0)
Gu.enable(Gu.TEXTURE_2D);
Gu.texImage(bild)
Gu.texFunc(Gu.TFX_MODULATE, Gu.TCC_RGBA)
Gu.texEnvColor(farbe)
Gu.texFilter(Gu.LINEAR, Gu.LINEAR)
Gu.texScale(1, 1)
Gu.texOffset(0, 0)
Gu.ambientColor(farbe)
Gum.matrixMode(Gu.MODEL)
Gum.loadIdentity()
Gum.translate(0, 0, -3)
Gum.drawArray(Gu.TRIANGLES, Gu.TEXTURE_32BITF+Gu.COLOR_8888+Gu.VERTEX_32BITF+Gu.TRANSFORM_3D,
triangle)
Gu.end3d()
```

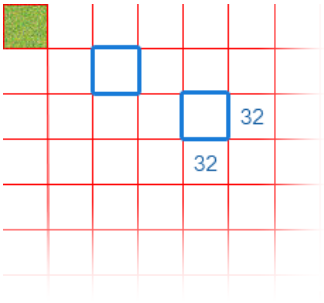
Noch ein nachträglicher Hinweis: die Auflösung der Texturbilder muss durch 8 Teilbar sein und darf maximal 512x512 betragen. Ich habe es zwar versucht einigermaßen verständlich zu erklären, aber trotzdem kann es ein wenig unverständlich sein, da ich mich persönlich nicht richtig intensiv mit der 3D Gu beschäftige. Am besten fängt man hiermit an und lernt dann Stück für Stück dazu, "learning by modding" nenn ich das.

Fragen bitte ins Forum posten, Danke!

2.14. Tilemaps

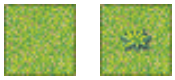
"Tile Maps - was zur Hölle ist denn das?"...das werden sich wohl einige von euch denken. Die Erklärung ist denkbar einfach:

Tile Maps sind Maps, die aus aneinandergereihten Grafiken mit allesamt der gleichen Größe aufgebaut sind. Man kann/braucht sie zum Großteil in RPG's (Roleplay Games). Das ganze kann man ganz einfach mit folgender Grafik veranschaulichen:



Eine Tilemap ist sozusagen ein Raster bzw eine Tabelle. Jedes Feld in diesem Raster kann man beispielsweise mit einer Grafik "belegen". Dabei entsteht eine Tilemap, wie man sie eben von RPG's kennt. Für Lua bedeutet das: man kann damit beispielsweise eine Karte für ein 2D-Spiel erstellen. Wollen wir uns nun doch jetzt einmal an den Code wagen...

In diesem kleinen Projekt sieht die Sache so aus, wir haben 2 Grafiken, beide mit der Auflösung 32x32.



Wir wollen den ganzen Bildschirm mit der Gras-Grafik füllen und an ein bestimmten Positionen das Gras mit der Pflanze auch noch unterbringen.

```
a=Image.load("tilebild1.png")
b=Image.load("tilebild2.png")

map={
	{a,a,a,a,a,b,a,a,a,b,a,a,a,b,a}, --Zeile 1
	{b,a,a,a,a,a,a,a,a,a,a,a,a,b,a}, --Zeile 2
	{a,a,b,a,a,b,a,a,a,a,a,b,a,b,a}, --Zeile 3
	{a,a,a,a,a,a,a,a,a,a,a,a,a,b,a}, --Zeile 4
	{a,a,b,a,a,b,a,a,a,a,a,b,a,b,a}, --Zeile 5
	{a,a,a,b,a,a,a,a,a,b,a,a,a,b,a}, --Zeile 6
	{a,b,a,a,a,b,a,b,a,a,a,b,a,b,a}, --Zeile 7
	{a,a,a,b,a,a,a,b,a,a,b,a,a,b,a} --Zeile 8
}

while true do

	for c=1,8 do
		for d=1,15 do
			screen:blit(d*32,c*32,map[c][d],false)
		end
	end

	screen.flip()
	screen.waitVblankStart()
end
```

Ich vermute, jetzt wird große Verwirrung herrschen...aber was hier extrem kompliziert erscheint, ist wahrhaftig recht einfach. Hierbei begründet sich auch die Notwendigkeit der for-Schleife: Vor dem Mainloop werden ersteinmal die beiden Bilder geladen. Anschließend erstellen wir eine Tabelle (in Fachkreisen Array genannt^^). Dabei handelt es sich allerdings um ein "mehrdimensionales" Array bzw Tabelle. Soll heißen, die Tabelle "map" besteht aus 8 Feldern:

```
map = { , , , , , , , }
```

In jedes dieser 8 Felder wird aber nun noch eine weitere Tabelle mit 15 Feldern eingebettet. Wir haben sozusagen eine Tabelle mit Untertabellen. Was ist der Sinn der Sache? Nun, die Tabelle "map" ist sozusagen unser Raster. Die Tabelle "map" definiert unsere Tilemap. Die 8 Felder der Tabelle "map" stehen für die Anzahl der Zeilen der Tilemap. Der Inhalt jedes Feldes, die "Untertabelle", steht für den Inhalt jeder Zeile. Bei genauem Betrachten sollte einem zudem auffallen, dass der Inhalt der "Untertabellen" nur a und b ist, genau so, wie unsere Tilebilder heißen.

Der Mainloop sollte einem bekannt sein, aber was zur Hölle soll diese kompliziert ausschauende for-Schleife? Nun, genau diese for-Schleife erstellt unsere Tilemap, mithilfe der Tabelle "map". Hier arbeiten wir außerdem mit 2 for-Schleifen, die 2. ist nämlich in die 1. eingebettet. Um das zu verstehen, müssen wir die Schleife mal zerstückeln:

```
for c=1,8 do
end
```

Diese for-schleife durchläuft jede einzelne der 8 Zeilen(8 Hauptfelder in der Tabelle "map"!). Da wir aber mit einer 2-dimensionalen Tabelle arbeiten, brauchen wir noch eine for-Schleife, die zusätzlich zur passenden Zeile noch den Inhalt der jeweiligen Zeile durchläuft und dazu die Bilder ausgibt:

```
for c=1,8 do --durchläuft alle 8 Zeilen der Tabelle "map"
        for d=1,15 do --durchläuft den Inhalt der jeweiligen Zeile
                screen:blit(d*32,c*32,map[c][d],false) --gibt die Bilder aus
        end
end
```

Womöglich gibt es jetzt noch Unklarheit über folgenden Befehl in der inneren for-Schleife:

```
screen:blit(d*32,c*32,map[c][d],false)
```

...screen:blit sollte ja schon lang ebekannt sein, nur der Inhalt, die sogenannten "Argumente" bzw "Parameter" sehen ein wenig verwirrend aus. Das ganze lässt sich leicht durch folgende Annahme klären: c=3 (also die Zeile ist 3), d = 7(der dazugehörige Zeileninhalt ist 7). Dann würde der Befehl zu diesem Zeitpunkt folgendermaßen aussehen (bitte mit dem Inhalt der Tabelle "map" vergleichen, hilft zum Verständnis):

```
screen:blit(7*32,3*32,map[3][7],false)
```

ist dasselbe wie:

```
screen:blit(224,96,a,false)
```

Bei Verständnisproblemen bitte ins Forum posten! Danke.

2.15. Kollisionen

Eine Kollision im echten Leben, das wäre zum Beispiel der Zusammenprall zweier Gegenstände. Auch beim Programmieren kann bzw. braucht man manchmal "Kollisionen". Was Kollisionen in Lua sind und wann man sie braucht, das erfahrt ihr in diesem kleinen Tutorial:

Nehmen wir einfach mal an, wir haben das Bild eines Balles geladen und geben den Ball mit den vorher definierten Variablen x und y aus. Im Mainloop vergrößern wir x ständig, das hat zur Folge, dass der Ball immer weiter nach rechts gelangt, solange, bis wir einschreiten und ihm einen Riegel vorschieben. Wir teilen dem Luaplayer mittels eines if-Blocks mit, dass der Wert von x ab einer bestimmten Position nicht mehr vergrößert wird bzw ihm sogar im Gegensatz wieder etwas abgezogen wird. Dadurch entsteht sozusagen eine unsichtbare Mauer, an der der Ball abprallt. Unsere erste kleine "Kollision"!

```
if x > 480 then
        x=480
end
```

In diesem Fall nennt man das eine "Kollisionsabfrage". Wenn man nun will, dass sich x beim Erreichen des Grenzwertes wieder andauernd verkleinert, der muss sich eines Tricks bedienen. Wir bauen eine Variable für den "Modus" ein.

```
ball = Image.load("ball.png")
x = 100
```

```
y = 100
modus = 0

while true do

	if modus == 0 then
		x = x + 1
	end

	if x>480 then
		modus = 1
	end

	if modus == 1 then
		x = x - 1
	end

	if x < 0 then
		modus = 0
	end

	screen.flip()
	screen.waitVblankStart()
end
```

Der Trick ist ganz einfach: Am Anfang ist die Variable Modus gleich 0. Wenn der Modus gleich 0 ist, dann wird x ständig vergrößert. Wenn der Modus gleich 1 ist, dann wird x ständig verkleinert. Wenn x nun größer als 480 wird, dann wechselt auch der Modus, was zur Folge hat, dass sich der Wert ständig verkleinert, bis zur Kollisionsabfrage für x kleiner 0.
Dann wechselt der Modus wieder zu 0 und der Wert von x wird wieder vergrößert.

Hier sieht das alles noch recht einfach aus, was es zweifellos auch ist. Die Kollisionsabfrage wird erst bei verschiedenen Flächenformen wie Kreise, Ellipsen oder Dreiecken relativ schwer...

2.16. SIO

SIO steht für Serial Input/Output. An der PSP ist das die Schnittstelle neben der Kopfhörerbuchse. Mit dem SIO-Port der PSP hätte man ungeahnte neue Möglichkeiten, denn der Port eröffnet die Möglichkeit, mit anderen Geräten aller Art zu kommunizieren...so könnte man beispielsweise ein Modem an den SIO Port anschliessen und mit dem richtigem passenden Homebrew eine Verbindung herstellen. Es gibt auch schon zahlreiche Erfindungen für diesen Port, wie z.B Standardmäßig das HPRM (das Socom Headset) oder das PSP Motion Kit. Allerdings kann man in LUA nicht viel mit dem SIO Port anfangen. Man braucht nämlich immer den LuaPlayer mit der passenden Library dazu.

Trotzdem eine kleine Einführung in die Arbeit mit dem SIO-Port:
Bevor man Daten vom SIO-Port lesen bzw senden kann, muss man den Port ersteinmal initialisieren:

```
System.sioInit(Baudrate)
```

Die Baudrate oder Symbolrate ist eine physikalische Größe, welche die Schrittgeschwindigkeit einer Datenübertragung beschreibt. Aber woher man nun weiss, was die passende Baudrate ist, kann ich leider nicht sagen.

Nun kann man Daten an den Port senden, oder empfangen:

```
System.sioRead()
System.sioWrite("text")
```

Zu beachten ist hier aber, dass die Befehle und ihre Verfügbarkeit auch wieder vom jeweiligen LuaPlayer abhängen. Zudem kenne ich im Moment keinen einzigen LuaPlayer, bei dem die Arbeit mit dem SIO-Port richtig funktionieren würde.

2.17. Das erste kleine Projekt

Genaugenommen hast du jetzt die Basics von LUA drauf. Aber wie macht man denn nun ein Programm? Darauf möchte ich euch eine Antwort geben. Und zwar anhand eines simplen Malprogramms. Ich möchte jetzt allerdings nicht daraufhin hundert verschiedene Mini-Malprogramme auftauchen sehen.

Machen wir uns doch zuerst einmal Gedanken darüber, was man so alles brauchen könnte für das Malprogramm. Für ein simples Malprogramm brauchen wir:
1. Eine Malfläche
2. Einen Cursor
3. Tasten
4. Malfunktion

Vielleicht klingt das jetzt erstmal fürchterlich aufwendig, ist es aber gar nicht. Der Code sollte recht einfach zu verstehen sein. Der DPad-Code wird aber vielleicht für Verwirrung sorgen. Bei Fragen dazu einfach ins Forum posten. Den Cursor soll man nämlich mit dem DPad steuern. Für das DPad gibt es einen Standardcode, den wir auch hier verwenden. Nun aber zum Code:

```
weiss = Color.new(255,255,255) --definiert die farbe schwarz
schwarz = Color.new(0,0,0) --definiert die farbe weiss
malflaeche = Image.createEmpty(480,272) --erstellt ein neues, leeres Bild mit 480x272 Pixeln
malflaeche:clear(weiss) --färbt das Bild "malflaeche" weiss

cursor = Image.load("cursor.png") --lädt das Bild "cursor.png" in die Variable cursor
cursor_x = 100 --Variable mit dem Wert 100, später unsere X-Position des Cursors
cursor_y = 100 --Variable mit dem Wert 100, später unsere Y-Position des Cursors

while true do --Mainloop

	pad = Controls.read() --DPad-Code...muss man nicht verstehen. Fragen bitte ins Forum posten
	dx = pad:analogX()
	dy = pad:analogY()
	if dx > 30 then
		cursor_x = cursor_x + (math.abs(pad:analogX())/64)
	end

	if dx < -30 then
		cursor_x = cursor_x - (math.abs(pad:analogX())/64)
	end

	if dy > 30 then
		cursor_y = cursor_y + (math.abs(pad:analogY())/64)
	end
	if dy < -30 then
		cursor_y = cursor_y - (math.abs(pad:analogY())/64)
	end --DPad-Code ENDE

	screen:blit(0,0,malflaeche) --zeigt die Malfläche an
	screen:blit(cursor_x, cursor_y, cursor, true) --zeigt das cursor auf cursor_x und cursor_y an

	if pad:cross() then
		malflaeche:drawLine(cursor_x, cursor_y,cursor_x, cursor_y,schwarz)
	end

	screen.flip()
	screen.waitVblankStart()
end
```

Wenn ihr alles übernommen habt und abspeichert, solltet ihr ein sehr mageres Malprogramm besitzen^^...ich hoffe, ihr versteht, wie das Programm funktioniert. Wir haben erst 2 Farben definiert, weiss und schwarz. Dann habe wir ein leeres Bild mit der Auflösung 480x272 erstellt. Das ist dann später unsere Malfläche. Anschließend haben wir noch eine Grafik für den Cursor geladen und seine x und y Position bestimmt. Im Mainloop dann haben wir zuerst den Code für den DPad. Ihr fragt euch sicherlich, was das "> 30" usw bedeutet. Naja, die Erklärung ist denkbar einfach, das DPad kann ja mehr oder weniger sanft bewegt werden. Die DPad-Sensibilität reicht bis zu 128. Ohne das "> 30" würde sich der Cursor schon beim leichtesten

Berühren stark bewegen. In den If-Anweisungen werden je nach Bewegung des DPads die Koordinaten des Cursors verändert. Nach diesem Code wird noch die Malfläche und der Cursor selbst ausgegeben. Und zu guter letzt, wenn man X drückt, soll noch ein Punkt auf die Malfläche gezeichnet werden...

3. Die Arbeit mit dem PGELua Wrapper

3.1. Die Vorzüge von PGELua

Wer intensiv mit den "LuaPlayern" gearbeitet hat, wird manchmal sehr frustriert gewesen sein, da man viel zu schnell an seine Grenzen stößt. Und damit meine ich nicht eure Fähigkeiten, sondern die Leistungsgrenzen der "LuaPlayer". Je mehr auf dem Bildschirm ausgegeben wird, desto langsamer wird die ganze Anwendung und die FPS rutschen schlagartig in den Keller. Und genau damit muss man sich in PGELua nicht herumschlagen. PGELua ist verdammt schnell, selbst bei einer total zugemülltem Bildschirm. Das kommt daher, da PGELua komplett unabhängig von den anderen LuaPlayern entwickelt wurde, und zudem astrein programmiert ist. Die LuaPlayer nämlich bestehen aus einem gigantischem Sammelsorium aus "zusammenklaubtem" Code von verschiedensten Quellen. Und PGELua im Gegensatz dazu nicht. Wer es nicht glauben will, der soll sich selbst überzeugen.

http://pge.luaplayer.org

Eigentlich ist ein HelloWorld für PGELua gar nicht notwendig, da im PGELua-Package gigantische Mengen an professionell gecodetem und gut kommentiertem Beispielcode dabei sind.

3.2. Hallo Welt

PGELua erwartet von seinem "Meister" immer, dass er selbst eine Schriftart lädt und festlegt, die er danach für die Schriftausgabe verwenden kann. Zudem will PGELua auch vor der Text/Bild/GFX-Ausgabe immer wissen, was für ein "Modus" gerade läuft, nämlich ob er sich im Zeichenmodus befindet, oder nicht. Was vielleicht ein wenig umständlich erscheinen mag, ist ein wahres Goldstück für die Performance des Programmes. Auf gut deutsch: Es ist gut gemeint und ist auch gut so :D Genug um den heissen Brei gerdet, ich serviere euch jetzt einfach mal den kompletten Code.

```
font = pge.font.load("verdana.ttf", 12, PGE_RAM)  --es können auch mehrere versch. "Fonts" geladen werden!

white = pge.gfx.createcolor(255, 255, 255)
black = pge.gfx.createcolor(0, 0, 0)

while pge.running() do

	pge.controls.update()

	pge.gfx.startdrawing()
	pge.gfx.clearscreen(black)

	font:activate()
	font:print(10, 10, white, "Hallo Welt")

	pge.gfx.enddrawing()

	pge.gfx.swapbuffers()

end
```

Vielleicht sieht das auf den ersten Blick recht seltsam aus, aber das Prinzip bzw. die Funktionsweise sind doch ganz einfach. Zuerst laden wir eine Schriftart mit der Größe 12 in den Arbeitsspeicher der PSP. Alternativ zu PGE_RAM könnte man auch PGE_VRAM schreiben. VRAM ist der Grafikspeicher vom Grafikchip der PSP und generell etwas schneller als der Arbeitsspeicher, dafür aber auch kleiner. Die Variable, die die Schriftart beinhaltet heisst hier font. Wir könnten sie aber auch Wurstbrot nennen. Nur müssten wir dann später anstatt

```
font:activate()
font:print(10, 10, white, "Hallo Welt")
```

folgendes schreiben:

```
Wurstbrot:activate()
Wurstbrot:print(10, 10, white, "Hallo Welt")
```

Die Definition der Farben bedarf, denke ich, keiner weiteren Erklärung. Der Mainloop lautet hier

```
while pge.running() do
```

anstatt

```
while true do
```

, wobei letzteres ebenso funktionieren würde. Ein Nachteil, oder auch Vorteil, jenachdem wie man es betrachtet, ist die Sache mit den Tasten. Man kann die Tasten nicht überwachen und diesen Status in einer Variablen abspeichern, man bindet lediglich einmal

```
pge.controls.update()
```

ein, was die Tasten überprüft. Dementsprechend würde dann auch ein "Buttoncheck" anders aussehen:

```
if pge.controls.pressed(PGE_CTRL_CROSS) then ... end
```

Das tolle daran ist aber auf jeden Fall, dass der Hickhack mit oldpad und pad wegfällt. Das erledigt PGE automatisch. Bevor man nun Text etc ausgeben kann, muss zuvor der Zeichenmodus initialisiert werden:

```
pge.gfx.startdrawing()
```

Danach kann gezeichnet werden. Wenn man fertig ist mit zeichnen sollte man den Modus wieder schliessen:

```
pge.gfx.enddrawing()
```

Danach könnte man dann PSP Systemdialoge anzeigen, was hier jetzt abe rnicht mit veranschaulicht wird. Darüber geben die Codebeispiele von PGELua mehr Aufschluss. Und zuguterletzt hätten wir da noch

```
pge.gfx.swapbuffers()
```

, was grob gesagt das screen.flip() und screen.waitVblankStart() ersetzt. Und natürlich darf man nicht vergessen, die while-Schleife mit einem end wieder ordnungsgemäß zu schliessen. Viel Spaß beim Ausprobieren!

4. Allgemeine Lua Funktionen und Befehle

4.1. Operatoren

Hier eine Tabelle der Operatoren in Lua

=	Zuweisungsoperator ("a = 1" oder "b = 18939", weist der Variablen den Wert rechts vom = zu)

Vergleichsoperatoren:

==	Gleichheitsoperator ("hat a den selben Wert wie b")
~=	Ungleichheitsoperator ("hat a einen anderen Wert als b")
<	"Kleiner-Operator" ("a ist kleiner als b")
>	"Größer-Operator" ("a ist größer als b")
<=	"Kleiner-gleich-Operator" ("a ist kleiner oder gleich b")
>=	"Größer-gleich-Operator" ("a ist größer oder gleich b")

Logische Operatoren:

and, or, not	if (a = 1 and b = 5) or (a = 3 or a = 2) and c = 2 and not d = 1 then ... end

true, false if a == false then ... end oder
if a == true then ... end oder if a then ... end

4.2. Escape-Sequenzen

\" --Anführungszeichen
\' --Apostroph

screen:print("\"in anführungszeichen\"")

\a --???
\b --Backspace
\f --???
\n --Neue Zeile
\r --???
\t --Horizontaler Strich = -
\v --Vertikaler Strich = |
\\ --Backslash = \
\[--eckige Klammer "auf" = [
\] --eckige Klammer "zu" =]

4.3. String Operationen

Für Strings, also Zeichenketten, gibt es so allerhand Funktionen, die standardmäßig im Lua-Core dabei sind und daher mit jedem Lua-Wrapper funktionieren (sollten). Ich liste hier jetzt einfach mal einige auf:

string.byte
string.char
string.find
string.format
string.len
string.lower
string.rep
string.sub
string.upper
string.gfind
string.gsub
string.dump

string.byte liest den numerischen Wert eines Zeichens aus.
Beispiel: text = string.byte("hello",2)
screen:print(0,0,text,color)
Als text wird dann "101" ausgegeben, da der 2. Buchstabe in "hello" ein e ist, und e ist der 101. Buchstabe in der ASCII-Tabelle...

string.char wandelt ASCII-Werte in einen string um.
Beispiel: text = string.char(65,66,67)
screen:print(0,0,text,color)
Als text wird "ABC" ausgegeben.

string.find sucht innerhalb eines strings nach angegebenen Werten.
Beispiel: text = string.find("Hi wie gehts", "geht", 7, true)
screen:print(0,0,text,color)

string.format formatiert einen string z.B. in ASCII Code.
Beispiel: text = string.format("%s %q", "Hallo", "du da!")
screen:print(0,0,text,color)
%c Wandelt eine Zahl in ein ASCII - Zeichen um
%E, %e Gibt eine Zahl mit Exponentialwert aus E+00 oder e+00
%f Gibt eine Zahl mit maximal 6 Stellen hinter dem Komma aus
%g, %G Gibt eine Zahl mit insgesamt 6 Stellen aus. Wenn nötig mit e+00 oder E+00

%o Wandelt eine Zahl in octal um
%X, %x Wandelt eine Zahl in hexadezimal um.
%q Umgibt einen String mit Anführungszeichen
%s gibt den String unverändert aus

string.len ermittelt die Länge des Strings
Beispiel: text = string.len("hallo")
screen:print(0,0,text,color)
Ausgegeben wird hier 5, da "hallo" aus fünf Zeichen besteht.

string.lower wandelt einen String in Kleinbuchstaben um.
Beispiel: text = string.lower("HALLO")
screen:print(0,0,text,color)
Ausgegeben wird hier "hallo".

string.rep vervielfältigt den String.
Beispiel: text = string.rep("hi ", 3)
screen:print(0,0,text,color)
Ausgegeben wird dann "hi hi hi".

string.sub schneidet einen bestimmten Teil eines strings aus.
Beispiel: text = string.sub("hiho", 3,4)
screen:print(0,0,text,color)
Hier erscheint dann "hi" auf dem Bildschirm, h und o wurden entfernt...

string.upper wandelt einen string in Großbuchstaben um.
Beispiel: text = string.lower("hallo")
screen:print(0,0,text,color)
Ausgegeben wird hier "HALLO".

string.gfind dursucht einen String auf das Vorkommen eines Schlüsselwortes und merkt sich bei einem Fund die Fundposition, beim nächsten Suchdurchlauf wird dann ab der 1. Fundstelle weitergesucht. Der Rückgabewert ist eine Funktion, die einen String zurückgibt.
Beispiel: textsuche = string.gfind("hallo", "l")
screen:print(0,0,textsuche, color)
screen:print(0,10,textsuche, color)
Ausgegeben wird hier l
l

string.gsub sucht in einem String nach einem Schlüsselwort und ersetzt es. Wenn der 4. Parameter mitangegeben ist wird das Schlüsselwort nur so oft ersetzt, wie es der Parameter angibt.
Beispiel: text = string.gsub("Hallo", "a", "e")
text = string.gsub("Hallo", "l", "s", 2)
Rückgabe wäre dann Hello oder Hasso.

string.dump wandelt eine Luafunktion in ihre Binärform um.
Beispiel:
```
function test(a,b)
        return a*b
end
string.dump(test)
```

4.4. Mathe Operationen

math.abs(zahl)
Der absolute Wert

math.acos(zahl)
Der Arcus Cosinus (Umkehrfunktion des Cosinus)

math.asin(zahl)
Der Arcus Sinus (Umkehrfunktion des Sinus)

math.atan(zahl) — **Der arcus tangens**

math.atan2(zahl1, zahl2) — **Der arcus tangens, meldet keinen Fehler, wenn zahl2 = 0 ist**

math.ceil(zahl)
Aufrunden

math.cos(winkel_in_rad)
Der Cosinus eines Winkels, Rückgabewert in Radians

math.deg(zahl)
Umwandlung von Radians zu Grad

math.floor(zahl)
Abrunden

math.frexp(zahl)
Normalisierungsfunktion, Beispiel: 5 * 4^3 = 320 -> faktor, exponent = math.frexp(320) -> faktor = 5, exponent = 3

math.ldexp(faktor, exponent)
Umkehrfunktion von math.frexp

math.log(zahl)
Der natürliche Logarithmus zur Basis e (= eulersche Zahl ~ 2.71828)

math.log10(zahl)
Der Logaritmus zur Basis 10

math.max(zahl1, zahl2,...)
Liefert die groesste Zahl einer Gruppe von Zahlen, Beispiel math.max(3,8,2) -> 8

math.min(zahl1, zahl2,...)
Liefert die kleinste Zahl einer Gruppe von Zahlen

math.mod(zahl1, zahl2)
Ganzzahliger Rest von Zahl1 / Zahl2

math.pi
Konstante Pi ~3,1415926535898

math.pow(x, n)
Fuer x hoch n

math.rad(zahl)
Umwandlung von Grad zu Radians

math.random(anfangszahl, endzahl)
Zufallszahl generieren

math.randomseed(os.time())
Einen Zahlenpool generieren für bessere Zufallszahlen

math.sin(zahl)
Der Sinus eines Winkels

math.sqrt(zahl)
Die Quadratwurzel aus einer Zahl ermitteln

math.tan(zahl)
Der Tangens eines Winkels

4.5. Table Operationen

table.concat (table , zwischentext , ab wann , bis wo)
Verbindet die Werte einer Table.

table.foreach (table, funktion)
Geht alle Felder einer Table durch und übergibt sie ggf an eine Funktion.

table.getn (table)
Ermittelt die Anzahl der Tablefelder.

table.sort (table , funktion)
Sortiert eine Table. Ist keine Funktion gegeben wird nach groesser/kleiner Prinzip sortiert.

table.insert (table, position, wert)
Fügt ein Feld in eine vorhandene Table ein.

table.remove (table , position)
Loescht ein Feld einer Table

table.setn (table, felderanzahl)
Setzt die Anzahl der Tablefelder.

5. Nützliche Funktionen & Codes

Die gesamten Lua Codesnippets und Funktionen, und vor allem noch einige mehr, gibt es auch unter

http://codebase.parabella.org

Ein paar wenige nützliche Codesnippets möchte ich auch hier zur Verfügung stellen. Ich beginne hie rmal mit dem Analogpad-Code, da man den wirklich oft brauchen kann:

```
sens = 64
x0 = 0
y0 = 0

--ab hier muss der Code dann in den Mainloop

pad = Controls.read()
dx = pad:analogX()
dy = pad:analogY()
if dx > 30 then
        x0 = x0 + (math.abs(pad:analogX())/sens)
end

if dx < -30 then
        x0 = x0 - (math.abs(pad:analogX())/sens)
end

if dy > 30 then
        y0 = y0 + (math.abs(pad:analogY())/sens)
end

if dy < -30 then
        y0 = y0 - (math.abs(pad:analogY())/sens)
end
```

Desweiteren finde ich folgende Funktion für das Zeichnen eines Kreises sehr nützlich, da esnicht immer standardmäßig Befehle für das Zeichnen von Kreisen gibt:

```
function drawCircle(where, x, y, radius, color)

        for angle=0, 360 do
```

```
                where:drawLine((math.sin( angle)*radius)+x, (math.cos(angle)*radius)+ y, (math.sin(angle+1)*radius )
                +x, (math.cos(angle+1)*radius )+y, color)
        end
end
```

Und zum Füllen des Kreises kann man folgende Funktion verwenden:

```
function fillCircle(where, x, y, radius, colour)

        drawCircle(where, x, y, radius, colour)
        for i=1, radius do
                drawCircle(where, x, y, (i), colour)
        end
end
```

Und wenn man mal eine Datei einliest und deren Zeilen speichern will, lohnt sich folgende Funktion:

```
function readAllLines(datei)
        zeilen = {}
        datei = io.open(datei,"r")
        for line in datei:lines() do
                zeilen[line] = line
        end
        datei:close()

        return zeilen
end
```

6. Links

Es ist niemals falsch, eine ordentliche Liste mit wichtigen Links rund ums Thema PSP Programmierung, PSP Programmierung mit Lua und Lua allgemein zu haben. Daher gibt es zum krönenden Abschluss noch eine Linkliste für all eure Bedürfnisse in Punkto PSP-Development, und natürlich, damit es sich auch für mich lohnt, ein paar private Links zu meinen Webseiten und Partnerwebseiten.

http://www.bumuckl.com
http://lua.bumuckl.com
http://pspdev.bumuckl.com
http://www.pspsource.de
http://www.parabella.org
http://codebase.parabella.org
http://homebrewdump.dumpspot.net
http://www.evilmana.com
http://www.luaforge.net
http://www.lua.org
http://www.luaplayer.org
http://www.psp-programming.com
http://www.ps2dev.org
http://lua.gts-stolberg.de
http://forums.qj.net
http://www.xtreamlua.com

Das war es soweit, ich würde mich natürlich über Besuche auf http://www.bumuckl.com, http://www.parabella.org und http://www.dumpspot.net serh freuen, vielleicht traut sich auch der ein oder andere, sich dort zu registrieren. Wenn ihr Fehler in diesem Tutorial findet, etwas Feedback oder euch sonst irgendetwas bezüglich dem Thema Lua (oder was auch immer :D) auf dem Herzen liegt, dürft ihr es gerne auf einer meiner Websites posten, und auf http://www.pspource.de bin ich auch regelmäßig anzutreffen.

Mit freundlichen Grüßen, euer bumuckl