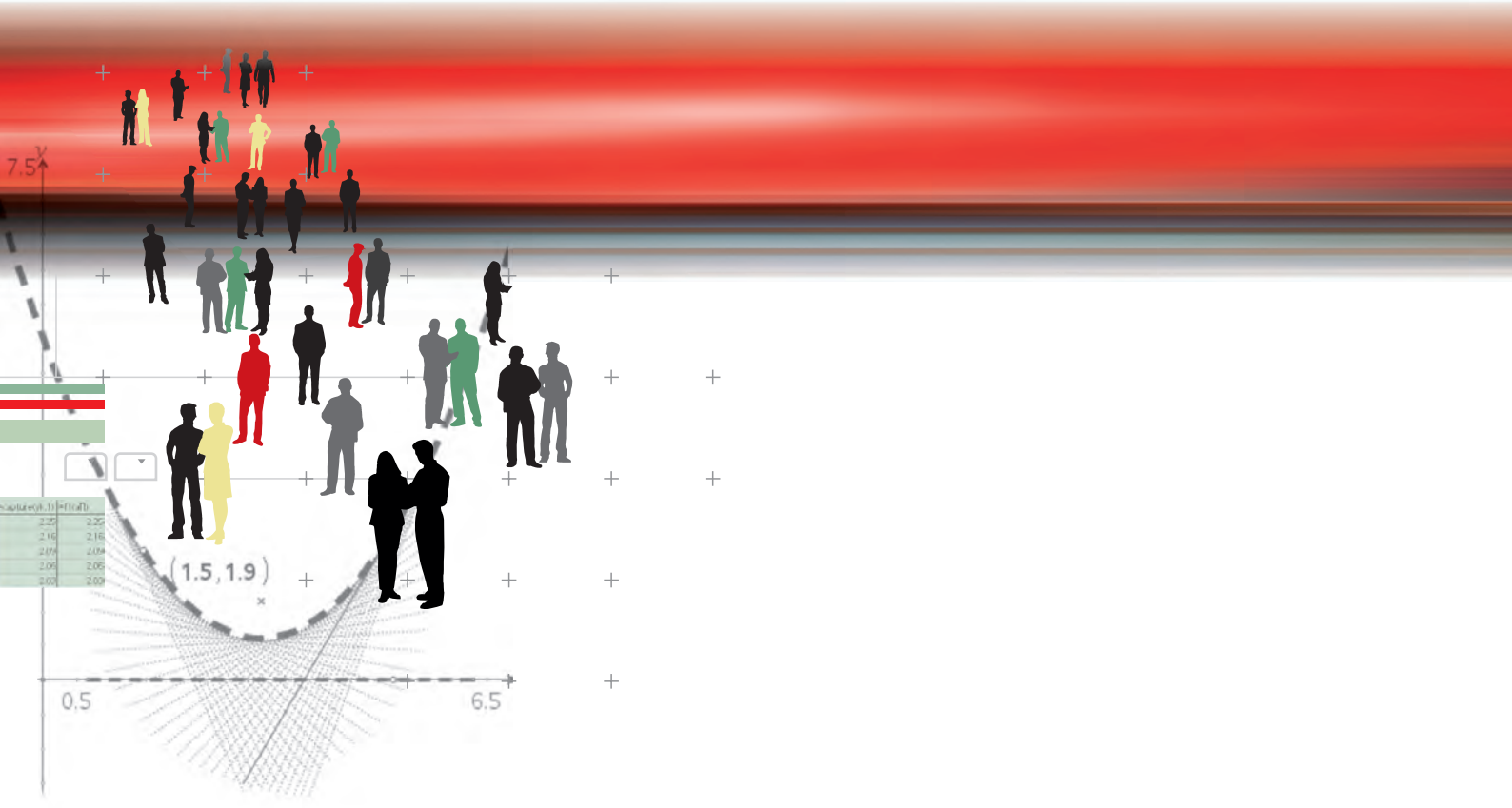
T³ EUROPE

T³- MATHEMATIK

Einführung in die LUA Programmierung

LUA-Skripts mit TI-Nspire™ Technologie

Steve Arnold, An-Sofie Bruggeman, Gregory Deroo, Jan-Klaas D'hulster, Joline Strubbe, Virginie Vileyn
bearbeitet für TI-Nspire™ CX CAS 4.0 von Josef Böhm (ACDCA und DUG)



Texas Instruments

Das vorliegende Material bezieht sich auf das T³ Cahier 35, erstellt von Mitgliedern von T³ Vlaanderen (www.t3vlaanderen.be).
Es wurde bearbeitet und ergänzt für Version 4.0 des TI-Nspire™ CX CAS von Mag. Josef Böhm, ACDCA und DUG.

Dieses und weiteres Material steht Ihnen zum pdf-Download bereit: www.ti-unterrichtsmaterialien.net

Inhalt

Vorwort 3

1 Zum Layout mit LUA 4

1.1 Neue Begriffe 4

1.2 Ein Text auf dem Display 5

1.3 Mehr Text mit einer Tabelle 10

1.4 Diktat in die Tastatur! 13

1.4.1 Im Zusammenspiel zum Quiz 14

1.5 Ein „Hüpfpunkt“ im Grafikfenster 17

1.5.1 Noch etwas Feinarbeit 18

2 Nach dem Schreiben folgt das Zeichnen 19

2.1 Ein einfaches Rechteck – mit Inhalt 21

2.2 Mit Grafik zu den „Figurierten Zahlen“ 22

2.2.1 Wir stellen das Gitter zusammen 23

2.2.2 Das Gitter wir variiert 24

2.2.3 Unsere Zahlen machen eine gute Figur 25

2.2.4 Letzte Kosmetik

3 Wir importieren fertige Grafiken 28

3.1 Wir bringen Bewegung(en) rein! 31

3.1.1 Mit Schiebereglern auch für das iPad 32

3.1.2 Auch eine Menüsteuerung ist möglich 33

3.1.3 Im Flug über das Display 36

4 Ab jetzt mit Klasse(n)! 40

4.1 Klasse Rechtecke 40

4.1.1 Die Maus packt zu 42

4.1.2 Tastatur & Maus United 44

4.2 Noch mehr Klasse(n) 45

4.2.1 Quadrat, Kreis, Tastatur und Maus 47

4.2.2 Aus zwei mach viele 48

5 Auf verschiedenen Plattformen 49

5.1 Auf die Skalierung kommt es an 49

5.2 Mit Maus und/oder Cursor 52

5.3 Schaltflächen und Tastatur 52

5.4 Kontrolle über die Pfeiltasten 54

6 Steve's Tipps and Tricks 55
6.1 Suchen, Ersetzen und Zerlegen 55
6.2 Lua sucht die Power des TI-Nspire 56
6.3 Mehr über Menüs 58

7 Etwas Mathematik mit viel Grafik 62

8 Text und Mathe in der Kiste 64
8.1 Texte innerhalb und außerhalb von Lua 65
8.2 Textbox im Doppelpack 67
8.3 Mathematik mit Chemie 69

9 Die Lua Physikmaschine 72
9.1 Wir starten die *Physics Engine* 72
9.2 Ein „Ding" tanzt am Display 73
9.3 Noch ein Tanz – jetzt mit Struktur 75
9.3.1 Pause, Neustart und Abbruch 78
9.4 Wir jonglieren mit mehreren Bällen 78
9.4.1 Unsichtbare Mauern 83
9.5 Polygone im „magischen Glitzern" 86

Unterlagen und Materialien zu Lua 89

Vorwort

Die „Cahiers“ der flämischen T^3 (Teachers Teaching with Technology) Gruppe sind eine reiche Quelle für Anwender moderner Technologie im Mathematikunterricht [1]. Mit Zustimmung der belgischen Kollegen haben wir bereits drei ihrer umfangreichen Hefte ins Deutsche übersetzt und ein wenig bearbeitet: „Einführung zum TI-Nspire CX CAS“, „Aufgaben zur Analysis mit TI-Nspire CX CAS“ und „Mathematik in wirtschaftlichen Zusammenhängen“.

So ist mir auch das Heft „Aan de slag met LUA“ aufgefallen. Da mein Interesse immer dem Programmieren gegolten hat, habe ich mir das näher angesehen. Ich war beeindruckt, aber auch abgeschreckt von der vorerst so komplex anmutenden Materie. Die Freunde in Belgien waren damit einverstanden, als ich ihnen mitteilte, dass ich versuchen wollte, auch dieses Heft den deutschsprachigen Anwendern zugängig zu machen.

Es stellte sich dann heraus, dass dieses *Cahier 35* eine ganze Serie von „Lua-Scripting Tutorials“ zur Grundlage hat, die Steve Arnold im Internet [2] veröffentlicht hat. Ich kenne Steve seit der TIME Konferenz, die 2008 in Südafrika stattgefunden hat. Es entwickelte sich eine sehr enge und intensive Kommunikation mit Steve. Ohne seine freundschaftliche, geduldige und so kompetente Hilfe wäre dieses Papier nicht zustande gekommen. Vielen Dank nach Australien. Seine Homepage mit den originalen „Scripting Tutorials“ steht auch nochmals an erster Stelle unter den *Referenzen zu Lua* am Ende des Skriptums. Weitere nützliche Links sind im Verlauf dieses Skriptums eingestreut.

Besonderer Dank gilt meinem lieben Freund Wolfgang Pröpper, den ich seit den ersten DERIVE-Tagen kenne und schätze. Er hat sich der Mühe unterzogen, dieses Skriptum auf Schreibfehler und Verständlichkeit zu überprüfen. Für allfällige noch vorhandene Fehler und Unklarheiten übernehme ich alleine die Verantwortung.

Viele Anwenderprogramme und Spiele basieren auf Lua. Die Verfügbarkeit mit TI-Nspire ist erst später dazu gekommen. Viele Ressourcen zu Lua finden sich im Internet. Unter den Referenzen sind einige Links angegeben.

Lua-Programme können bald sehr umfangreich werden. Die Programme, welche die *Physics Engine* einsetzen sind ein gutes Beispiel dafür. Ich muss auch hier auf die Materialien verweisen, die im Internet zu finden sind.

Steve hat erst kürzlich seine Tutorials erweitert. Er zeigt, wie der TI-Innovator mit Lua erfolgreich programmiert werden kann. Vielleicht gibt es diese Unterlagen in Zukunft ebenfalls in deutscher Sprache?

Damit wünsche ich viel Spaß mit Lua. Alle Programme sind verfügbar.

Josef Böhm

[1] http://www.t3vlaanderen.be/cahiers

[2] http://compasstech.com.au/TNS_Authoring/Scripting/index.html

1 Zum Layout mit Lua

Wichtig: Der Lua Script Editor ist **nicht** am Handheld verfügbar. Programme, die mit Lua erstellt wurden können aber auf den Handhelds verwendet werden. Sie müssen allerdings mit der TI-Nspire Teacher Software oder TI-Nspire Student Software erstellt und fallweise auch angepasst werden.

Bevor wir größere und praktisch nutzbare Applikationen erzeugen können, müssen wir die Grundlagen von Lua kennen lernen. Beim Schreiben eines Skripts ist große Sorgfalt angebracht. Groß- und Kleinschreibung sind zu beachten.

Wir beginnen mit der Darstellung eines Textes, danach besprechen wir die Behandlung von Tabellen, Abbildungen, … Aber vorerst sollen ein paar grundlegende Begriffe erklärt werden.

1.1 Neue Begriffe

Eine Funktion

So wie auch in anderen Programmiersprachen werden in Lua Funktionen verwendet: mit ihrer Hilfe drücken wir aus, was geschehen soll. Eine Funktion besteht aus dem Funktionskopf, wobei dann (mindestens) ein Argument zwischen Klammern folgt. In unserem Beispiel ist das Argument **gc**, das für *Graphical Context* steht. Im Funktionskörper steht dann der Inhalt. Mit **end** wird das Ende der Funktion angezeigt.

In diesem Kapitel wollen wir etwas Grafisches auf den TI-Nspire Schirm bringen:

```
function on.paint(gc)
    [Körper]
end
```

Kompilieren

Der im Script Editor verfasste Programmcode wird durch den Compiler in die Maschinensprache übersetzt.

Kommentare

Im Editor ist es auch möglich, in das Skript Kommentare einzufügen, die vom Compiler nicht übersetzt werden. Das ist sehr nützlich, da sich durch Kommentare Erläuterungen zum Programm einbringen lassen, die den Code auch anderen Personen verständlich zu machen. Außerdem lässt sich durch Kommentare das Programm leicht strukturieren.

Ein Kommentar wird durch zwei Bindestriche eingeleitet:

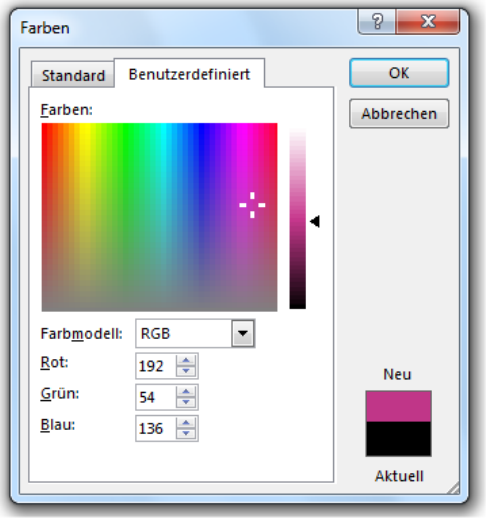
```
-- Hier kann ein Kommentar stehen
```

RGB-Codes

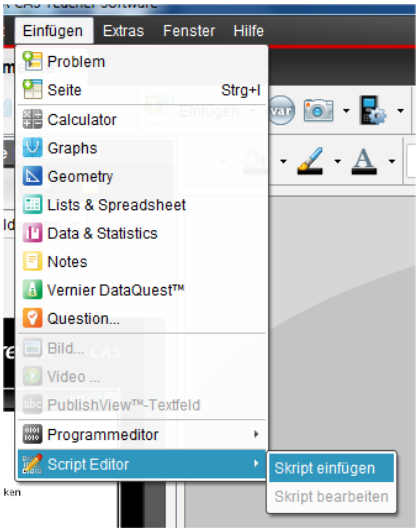
Für die Farbdarstellung (Text oder Grafiken) arbeiten wir mit RGB-Codes (Rot – Grün – Blau).

Um den Code einer gewünschten Farbe zu ermitteln, kann man Grafikprogramme oder auch, wie rechts gezeigt wird, MS-Word verwenden: um die Farbe Aubergine zu erhalten, ist der Code (192,54,136) zu wählen.

Die Codezahlen reichen von 0 bis 255: Rot = = (255,0,0), Grün = (0,255,0), Blau = (0,0,255), Gelb = (255,255,0), Schwarz = (0,0,0) und Weiß = (255,255,255).



1.2 Ein Text auf dem Display

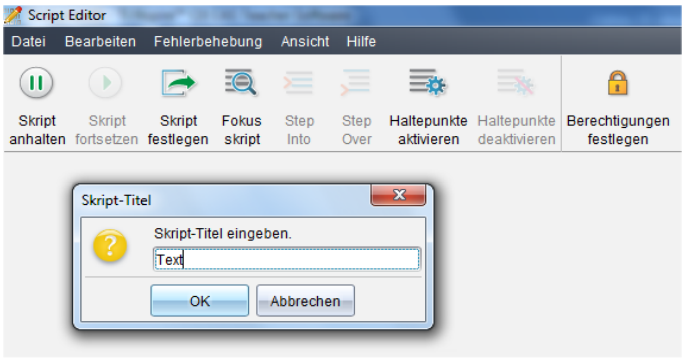


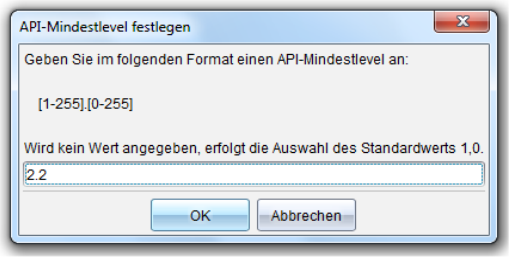
Nun wollen wir mit dem Programmieren in Lua beginnen. Laden Sie die TI-Nspire CX Software (am PC oder Notebook) und öffnen Sie über das Einfügen Menü den Script Editor.

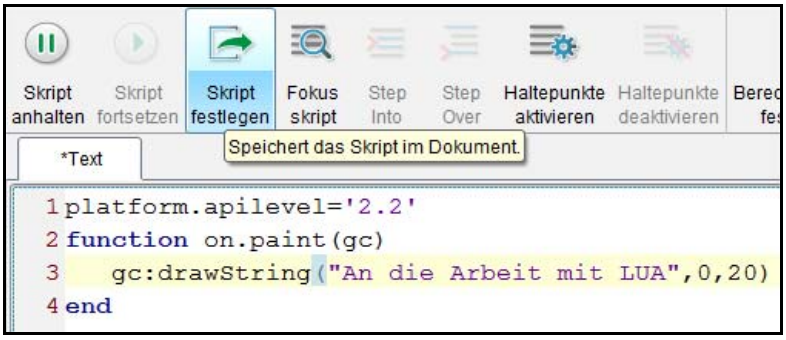
Zuerst bringen wir den gewünschten Text überhaupt auf den Schirm, dann wollen wir ihn gezielt platzieren und formatieren.

Das Skript muss einen Namen (= Titel) erhalten und anschließend legen wir über das Datei-Menü den „API-Mindestlevel fest“.

Dann kann es los gehen mit dem ersten Programm:





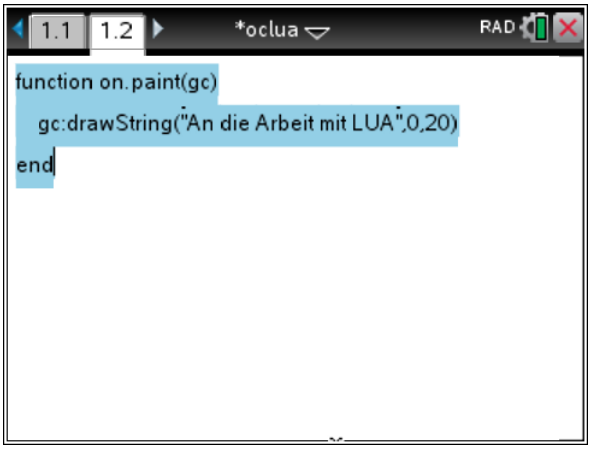


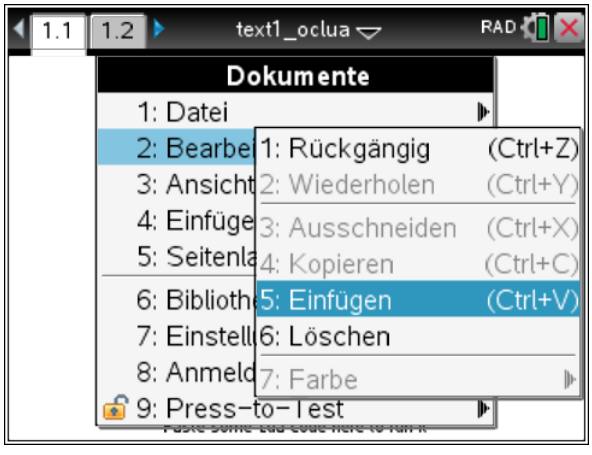
Der Programmkörper betrifft grafische Inhalte, daher beginnen wir mit „gc“. Dann wird mitgegeben, dass wir eine Zeichenkette (String) darstellen wollen. Nach dem String (wie üblich in Anführungszeichen - " ") stehen die *x*- und *y*-Koordinate der linken unteren Ecke unseres Textes (des A).

Der Koordinatenursprung liegt in der linken oberen Ecke, *x* wird nach rechts und *y* nach unten (in Pixels) gezählt (318 × 212 für Handheld und 942 × 628 für - meinen - PC).

Wenn wir mit dem Editieren fertig sind, klicken wir auf „Skript festlegen“, dann verschwindet auch das Sternchen im Skriptnamen und es erscheint hoffentlich keine Fehlermeldung im unteren Teil des Fensters. Mit „Fokus Skript“ bringen wir das Ergebnis unseres ersten Programms auf den Schirm. Hier wurde die Handheld-Ansicht gewählt. In der Computeransicht steht natürlich viel mehr Platz zur Verfügung.

Es soll aber auch die Möglichkeit gezeigt werden, ganz ohne Nspire-Software (d.h. ohne PC) mit Lua zu arbeiten. Es gibt ein Werkzeug namens *oclua*, das von Olivier Armand entwickelt wurde. (http://www.ticalc.org/archives/files/fileinfo/440/44075.html) Der Programmcode kann hier auf einer Notes-Seite geschrieben und mit *Copy & Paste* in die erste Seite des Dokuments kopiert werden.





Wir erhalten das gleiche Ergebnis wie oben.

Am Computerschirm sitzt der Text ebenfalls in der linken oberen Ecke, sieht aber verhältnismäßig viel kleiner aus.

(Wenn Ihnen das so gelungen ist und Sie auch mit der Software arbeiten, dann öffnen Sie bitte das Skript der mit *oclua* erzeugten Ausgabe. Da können Sie dann ein „richtiges“ Lua-Programm sehen und Sie werden staunen, versprochen!)

Wir werden in diesem Text hauptsächlich mit der Software arbeiten und dort wo es Sinn macht, auf allfällige Unterschiede zum Handheld hinweisen.

Um Platz zu sparen, werden wir aber möglichst die Screenshots vom Handheld zeigen.

Unser Text steht in der linken oberen Ecke des Displays. Wir wollen ihn nun in seine Mitte setzen, Zeichensatz, Größe und Farbe verändern.

Beschreibung der Aktion	Lua-Skript
Angabe der Lua-Version	platform.apilevel='2.2'
Funktionskopf	function on.paint(gc)
Lokale Variable h: Höhe des Displays (Schirms)	local h = platform.window:height()
Lokale Variable b: Breite des Displays (Schirms)	local b = platform.window:width()
Angabe des Zeichensatzes	gc:setFont("sansserif","b",16)
Angabe der Ausgabefarbe des Textes	gc:setColorRGB(41,184,120)
Lokale Variable txt für die Zeichenkette	local txt="An die Arbeit mit LUA"
sb: Textbreite	local sb=gc:getStringWidth(txt)
sh: Texthöhe	local sh=gc:getStringHeight(txt)
Ausgabe des Textes an der gewünschten Stelle	gc:drawString(txt,b/2-sb/2,h/2+sh/2)
Ende der Funktion	end

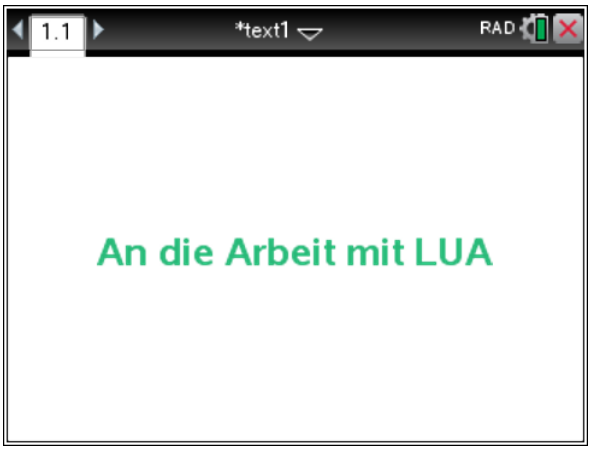
Die Groß- und Kleinschreibung der Funktionen muss peinlich genau eingehalten werden.

Das Ausgabefenster folgt auf der nächsten Seite. Hier kann mit dem Cursor gefuhrwerkt werden wie auch immer, das Display bleibt unverändert. Wir werden später sehen, dass die Ausgabe auch ohne Eingriff ins Programm variiert werden kann.

Um den Text in der Mitte des Displays darzustellen, werden zuerst die Höhe und Breite des Schirms und des Textes als lokale Größen definiert.

Diese Werte werden zur Berechnung der *x*- und *y*-Koordinaten des Textanfangs verwendet.

Testen Sie nun die Arbeitsweise des Skripts indem Sie die verschiedenen Parameter verändern: dann *Skript festlegen* und *Fokus Skript*.



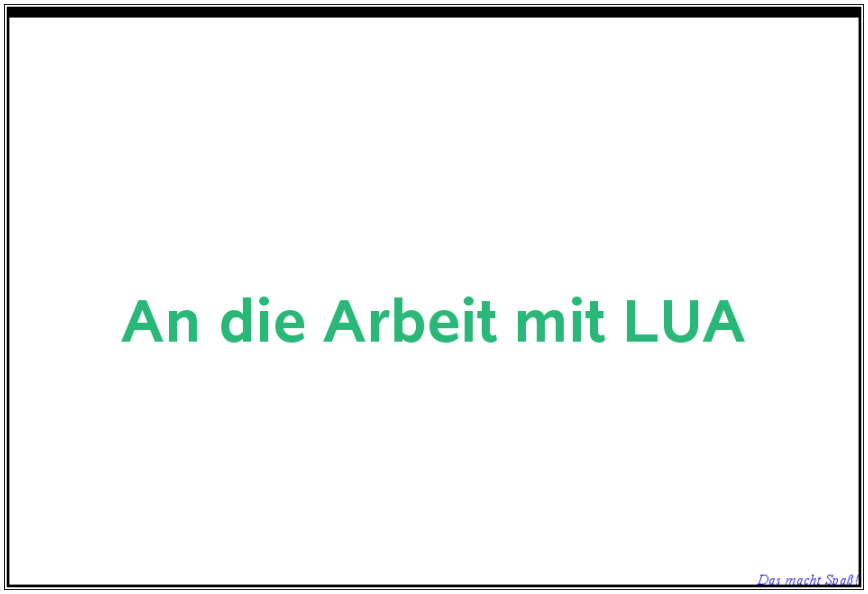
Es ist zu beachten, dass zuerst Schriftart und -größe (in der setFont-Funktion) festzulegen sind, dann wird erst die Position – abhängig von der Größe des Textfeldes – berechnet.

Im nächsten Schritt wollen wir zwei Textfelder in unterschiedlichen Formatierungen darstellen. Unser bestehendes Programm wird erweitert:

Beschreibung der Aktion	Lua-Skript
Angabe der Lua-Version	platform.apilevel='2.2'
Funktionskopf	function on.paint(gc)
Lokale Variable h: Höhe des Displays (Schirms)	local h = platform.window:height()
Lokale Variable b: Breite des Displays (Schirms)	local b = platform.window:width()
	local txt1="An die Arbeit mit LUA"
	local txt2="Das macht Spaß!"
	-- Ausgabe txt1
Parameter für das erste Textfeld	gc:setFont("sansserif","b",50)
	gc:setColorRGB(41,184,120)
	local sb1=gc:getStringWidth(txt1)
	local sh1=gc:getStringHeight(txt1)
Ausgabe des ersten Textfelds	gc:drawString(txt1,b/2-sb1/2,h/2+sh1/2)
	-- Ausgabe txt2
Parameter für das zweite Textfeld	gc:setFont("serif","i",12)
	gc:setColorRGB(20,25,240)
	local sb2=gc:getStringWidth(txt2)
	local sh2=gc:getStringHeight(txt2)
Ausgabe des zweiten Textfelds	gc:drawString(txt2,b-sb2,h)
Ende der Funktion	end

Das Ergebnis des Programms sehen wir rechts. Die Größe der Schrift im ersten Textfeld ist für die Ausgabe am Handheld zu groß gewählt. Während man für die Handheldausgabe mit den Schriftgrößen limitiert ist, kann man für die Computerausgabe stufenlos von 6 bis 255 mit ganzen Zahlen arbeiten.

Die Ausgabe am Computer folgt. Zuerst mit den oben gewählten Schriftgrößen, dann mit den Größen 70 bzw. 30.



Für den zweiten Text haben wir eine andere Farbe und den Schriftsatz „serif“ gewählt. Außerdem wird er kursiv (Stil = „i“ = italic) gedruckt. Die normale Ausgabe wird mit „r“ (= regular) erzwungen.

1.3 Mehr Text mit einer Tabelle

Im vorigen Abschnitt lernten wir, wie man ein Textfeld codiert. Für kurze Texte mag das ausreichen, für längere hingegen, sollten wir Tabellen verwenden. Wir werden zuerst lernen, wie man eine Tabelle erstellt, dann werden wir sie mit Inhalt versehen.

Wir arbeiten wieder mit einer on.paint(gc)-Funktion, in der wir über lokale Variable die Abmessungen des Schirms, die Anzahl der Zeilen, die Eigenschaften des Textes und eine noch leere Tabelle (Liste) angeben.

Es soll noch angemerkt werden, dass die lokalen Größen **h** und **b** sich dynamisch an die Größe des jeweils verwendeten Displays anpassen (Computer- oder Handheld View, geteilter Schirm, …).

Die Anzahl der Zeilen kann im Quellcode festgelegt werden. Wesentlich flexibler sind wir allerdings, wenn sie über einen Schieberegler verändert werden kann.

Dazu teilen wir das Display in zwei nebeneinanderliegende Fenster, links kommt die Tabelle hin und rechts eine Geometry-Seite. Diese Seite wird aktiviert, dann gehen wir auf *Dokumentwerkzeuge > 1:Aktionen > A:Schieberegler einfügen*. Der nun erscheinende Schieberegler muss über die entsprechende Variable mit der Tabelle – und auch mit dem Lua-Skript – verknüpft werden. Mit einem rechten Mausklick auf den Schieberegler gelangen wir zu dessen Einstellungen, geben den Variablennamen zeilen und den Bereich von 1 bis 10 mit einer Schrittweite 1 an. Die Gestalt des Reglers kann nach Vorliebe angepasst werden. Die rechts oben vorhandene Skala kann über einen rechten Mausklick verborgen werden.

Beschreibung der Aktion	Lua-Skript

screen ist eine Hilfsvariable.
 	
Lokale Variable für die Tabelle und für die Zeilenanzahl.
 	
var.recall stellt die Verbindung zur Nspire-Variablen her (die Variablen müssen nicht die gleichen Namen haben).
 	
Über den Schieberegler wird die Anzahl der dargestellten Zeilen definiert.
 	
Der Bildschirm wird aufgefrischt.
	platform.apilevel='2.2'
	screen=platform.window
	function on.paint(gc)
	  local h,b = screen:height(),screen:width()
	  local tab = {}
	  local zeilen = (var.recall("zeilen") or 1)
	
	  var.monitor("zeilen")
	-- Damit wird der Schieberegler „beobachtet“
	-- und eine Veränderung sofort berücksichtigt.
	
	  function on.varChange()
	  zeilen=var.recall("zeilen") or zeilen
	  screen:invalidate(); end
	-- ; kann einen Zeilenvorschub ersetzen

Die Schleife wird von *k* = 1 bis zur Zeilenanzahl durchlaufen und jede Zeile auf dem Display geeignet positioniert. Für ".." siehe die Anmerkungen. Damit ergibt sich das obige Bild.	gc:setFont("sansserif","r",10) gc:setColorRGB(0,128,0) for k = 1, zeilen do tab[k]="Zeile"..k zb = gc:getStringWidth(tab[k]) zh = gc:getStringHeight(tab[k]) gc:drawString(tab[k],b/2-zb/2,h*k/(zeilen+1)+zh/2) end; end

Die Tabelle steht nun. Jetzt wollen wir die Ausgabe „verschönern“: Schrift, Farbe und Größe sollen sich abhängig von der Zeilennummer verändern.

Jeweils erste und letzte Zeile sollen bleiben wie bisher, die dazwischenliegenden werden herausgehoben.

Dafür verwenden wir hier erstmalig eine if-then-else-Konstruktion.

Für die erste und jeweils letzte Zeile (Variable zeilen) lassen wir Größe und Farbe (grün), für alle anderen Fälle wird die Größe 16 und die Farbe Rot (RGB = (255,0,0)) und ein anderer Zeichensatz gewählt.

Beschreibung der Aktion	Lua-Skript

Hier beginnt die if-Abfrage: Wenn …, dann
 	
…, anderenfalls …
 	
Ende des if-Blocks
Zu beachten ist das doppelte Gleichheitszeichen „==“ im Quellcode. Es entspricht dem Gleichheitszeichen in einer Gleichung. Das einfache Gleichheitszeichen „=“ wird für eine Zuweisung (= Definition) verwendet und entspricht dem „:=“ beim TI-NspireCAS.
	-- Der erste Teil wie oben
	for k = 1, zeilen do tab[k]="Zeile"..k
	if k ==1 or k == zeilen then
	gc:setFont("sansserif","r",10)
	gc:setColorRGB(25,184,120)
	else
	gc:setFont("serif","b",16)
	gc:setColorRGB(255,0,0)
	end
	zb = gc:getStringWidth(tab[k])
	zh = gc:getStringHeight(tab[k])
	gc:drawString(tab[k],b/2-zb/2,
	h*k/(zeilen+1)+zh/2)
	end; end

Wenn uns das Layout gefällt, können wir uns mehr dem Inhalt widmen: Bis jetzt haben wir über die Quellcode-zeile for k = 1, zeilen do tab[k]="Zeile"..k den Auftrag gegeben, in der Tabelle *k* Textfelder mit dem Inhalt „Zeile“+Zeilennummer zu erzeugen.

Ab nun können wir direkt Inhalte in die Tabelle setzen, indem wir tab[k]="Zeile"..k ersetzen durch

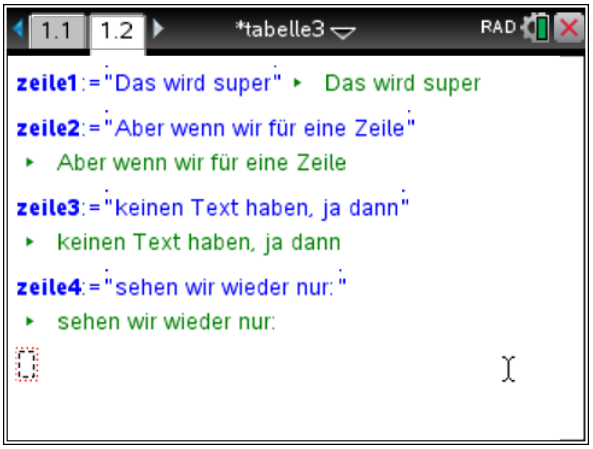
tab[k]=(var.recall("Zeile"..k) or "Zeile"..k).

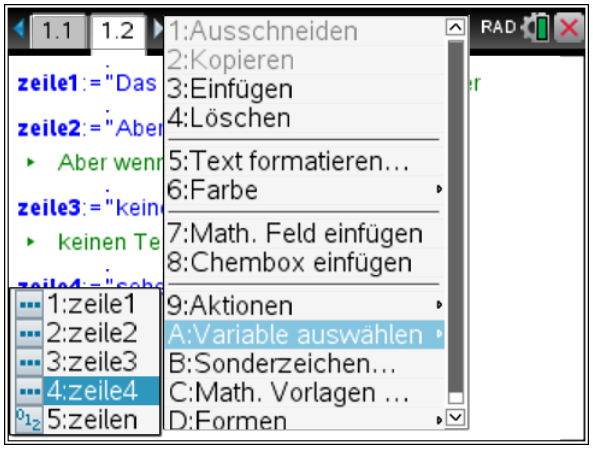
Diese Änderung veranlasst das System nach einem Wert für die Variable zu suchen. Falls keiner zu finden ist, wird wieder nur „Zeile“+Nummer ausgegeben. Wo sind aber die Inhalte – falls es welche gibt – zu finden?

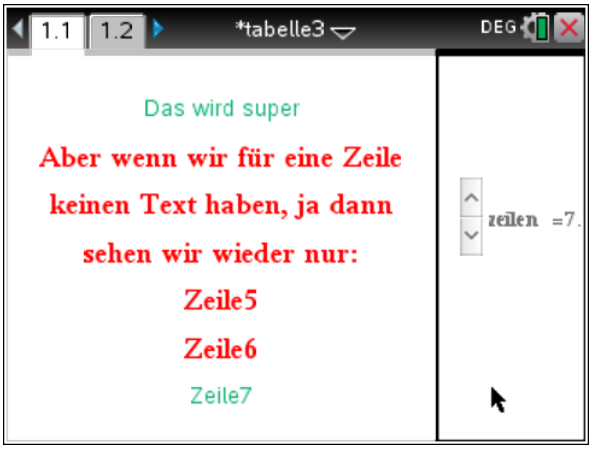
Wir fügen eine Notes-Seite ein und belegen die Zeilen unserer Wahl mit neuen Inhalten. Das muss in einem mathematischen Feld (Math Box) erfolgen.

Sobald die Definition mit der Eingabetaste bestätigt ist, erscheint die neue Zeichenkette im Lua-Fenster.

So können rasch Änderungen erfolgen.







Es gibt noch eine zweite Art, diese Änderungen direkt durchzuführen. Man muss wieder eine Notes-Seite einfügen. Wenn man mit der rechten Maustaste auf die leere Seite klickt, öffnet sich ein Menü, in dem man die Option *A:Variable auswählen* wählt. Da werden Sie dann nur „zeilen“ finden. Korrigieren Sie das auf die gewünschte Zeile und weisen Sie ihr einen String zu, wie z.B. zeile5:=“noch eine neue Zeile!“.

Auch auf diese Weise geht das sehr rasch und einfach.

Anmerkungen:

Versuchen Sie, die Formel zur dynamischen Positionierung der Textfelder zu erklären.

Die beiden Punkte „..“ hängen zwei Strings aneinander, daher „Zeile“..k für die Ausgabe.

Für die Ausgabe der roten Zeilen am Handheld muss die Zeichengröße von 16 auf 12 herabgesetzt werden.

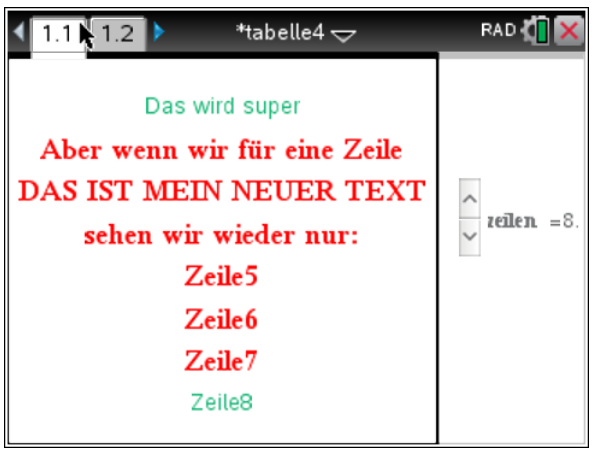
1.4 Diktat in die Tastatur!

Lua verfügt über eine Reihe von einfachen und nützlichen Werkzeugen, um verschiedene Eingaben (Zeichen, Pfeiltasten, Maus, Touchpad, …) zu verarbeiten. Wir werden vorerst Zeicheneingaben mittels on.charIn(char) akzeptieren lernen. Dazu definieren wir die Variable antwort, der alles, was wir über die Tastatur eingeben (zumindest Buchstaben, Ziffern und Sonderzeichen) übergeben wird.

Beschreibung der Aktion	Lua-Skript

Wir definieren die Variable antwort als einen Leerstring. Sobald die Funktion on.charIn aktiv wird, fügt sie jedes Zeichen, das über die Tastatur eingegeben wird hinten dran. (Beachten Sie nochmals das Zeichen .. zum Aneinanderreihen von Zeichenketten.) Der so neu gebildete String wird als Variable Zeile3 gespeichert. Dazu dient das Gegenstück von var.recall, das eine Variable aus dem TI-NspireCAS-Speicher holt, nämlich var.store("Var. Name", Wert). Sollte es die angeführte Variable nicht geben, dann wird sie somit erzeugt.
 	
platform.window:invalidate() frischt nach jeder Eingabe das Display auf und berücksichtigt alle Änderungen.
 	
Über die Backspace-Taste (←) können wir auch Zeichen löschen und die Eingabe korrigieren. Dazu hilft der string.usub-Befehl, der die ganze Antwort ohne das letzte Zeichen wiederholt.
 	
Wir fügen die beiden rechts gezeigten Funktion vor on.paint(gc) in unser Programm von vorhin ein und es sollte funktionieren.
	platform.apilevel='2.2'
	antwort=""
	function on.charIn(char)
	  antwort=antwort..char
	  var.store("Zeile3",antwort)
	  platform.window:invalidate()
	end
	
	function on.backspaceKey()
	  antwort=string.usub(antwort,0,
	  string.len(antwort)-1)
	  var.store("Zeile3",antwort)
	  platform.window:invalidate()
	end
	
	function on.paint(gc)
	…
	…

Wir wechseln ins Lua-Fenster und beginnen zu schreiben. Zeichen für Zeichen erscheint der neue Text in der dritten Zeile. Alle anderen Zeilen werden davon nicht berührt. Gleichzeitig ändert sich auch der Wert der Variablen zeile3 in den Notes.

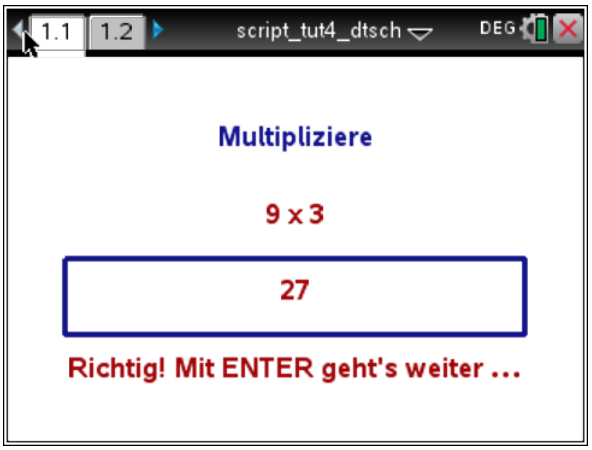


Die Abbildung nebenan zeigt ein schon aufwändigeres Beispiel, das gleich drei Fragen aufwirft:

Woher stammen die Eingaben, die hier zu sehen sind?

Wie zeichnen wir den netten blauen Kasten um die Lösung der Aufgabe?

Wie können wir die Antwort beurteilen und einen entsprechenden Kommentar abgeben?



1.4.1 Im Zusammenspiel zum Quiz!

Wir wollen demonstrieren, wie die Werte für Variable zwischen Notes, TI-Nspire Programm und Lua-Skript wechselseitig ausgetauscht werden können. Das im Folgenden entstehende Quiz für die Grundrechenarten kann auch komplett in Lua codiert werden (siehe script_tut4.tns in http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut4.html).

Die Strings für die Zeilen 1, 2 und 4 des Lua.Skripts werden jetzt im TI-Nspire-Programm vorbereitet und gespeichert.

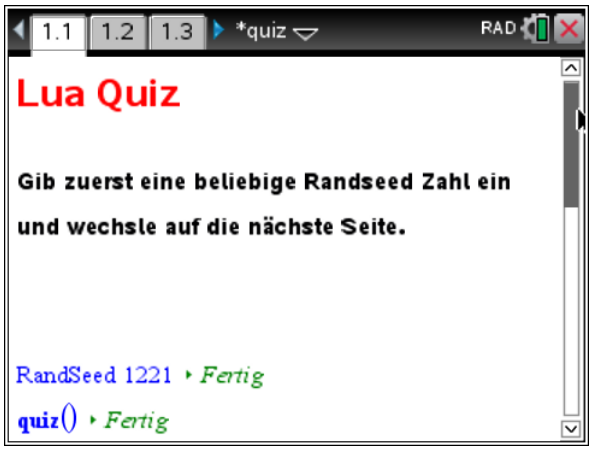
Die Zufallszahlen für die Aufgaben im Skript und für die Auswahl der Rechenart (typ) erzeugen wir in einer Notes-Seite.

Für Addition und Subtraktion werden die Zahlen *a* und *b* mit $0 \le a, b \le 20$ und *c*, *d* mit $1 \le c, d \le 10$ erzeugt. Die Zufallsgröße typ geht von 1 bis 4.

Wir wollen bei jedem Start des Trainingsprogramms neue Werte erhalten. Zu diesem Zweck soll der Benützer in den Notes eine beliebige Zahl als RandSeed eingeben. Das Programm quiz() muss nur einmal aufgerufen werden.

In den Notes wurde schon die Erzeugung der Zufallszahlen vorbereitet. Der Trick mit + neu – neu sorgt dafür, dass immer neue Zufallszahlen bereit gestellt werden.

```
Define quiz()=
Prgm
:zeilen:={"Addiere","Subtrahiere","Multipliziere",
          "Dividiere"}
:zeile1:=when(typ≤2,zeilen[typ],zeilen[typ])
:If typ=1 Then
:   zeile2:=string(a)&" + "&string(b)
:ElseIf typ=2 Then
:   zeile2:=string(max(a,b))&" -
            "&string(min(a,b))
:ElseIf typ=3 Then
:   zeile2:=string(c)&" × "&string(d)
:Else
:   zeile2:=string(c*d)&" / "&string(c)
:EndIf
:If expr(zeile3)=expr(zeile2) Then
:   check:=1
:   zeile4:="Richtig! ENTER für nächste
            Aufgabe."
:Else:   check:=0
:   zeile4:="Gib' die Antwort ein - oder ESC!"
:EndIf
:
:EndPrgm
```



Rechts stehen die Hilfsfunktionen zur Erzeugung der Zufallszahlen *a, b, c* und *d*. Links sieht man nur den oberen Teil der Notes-Seite.

Es folgt das kommentierte Lua-Skript.

Beschreibung der Aktion	Lua-Skript

Wir starten mit dem schon üblichen Beginn.
screen steht für platform.window (Abkürzung).
 	
neu holen wir von den Notes, antwort wird mit einem Leerstring belegt.
 	
Die nächsten beiden (Routinen on.charIn on.backspaceKey) und kennen wir schon von vorhin.
 	
Das Skript muss wissen, ob die Antwort richtig ist. Es kontrolliert daher die Variable check (aus dem TI-Nspire-Programm) und veranlasst den nächsten Schritt (Leeren der 3. Zeile und der Variablen antwort, sowie Übergabe eines neuen Werts für neu, womit auch gleich neue Zufallsgrößen erzeugt werden.
Immer wieder wird das Display aufgefrischt.
	platform.apilevel='2.2'
	local screen=platform.window
	local h=screen:height(); local b=screen:width()
	neu=var.recall("neu")
	antwort=""
	
	function on.charIn(char)
	  antwort=antwort..char
	  var.store("zeile3",antwort)
	  screen:invalidate()
	end
	
	function on.backspaceKey()
	  antwort=string.usub(antwort,0,string.
	  len(antwort)-1)
	  var.store("zeile3",antwort)
	  screen:invalidate()
	end
	
	function on.enterKey()
	  local ch=(var.recall("check") or 0)
	  local neuer=(var.recall("neu")or 1)
	  if ch == 1 then
	  var.store("zeile3",""); var.store("neu",neuer+1)
	  antwort=""; var.store("check",0)
	  end
	  screen:invalidate(); end

Beschreibung der Aktion	Lua-Skript

Wenn die Antwort falsch ist, dann geschieht überhaupt nichts.
Mit der Backspace-Taste können wir sie löschen und den nächsten Versuch unternehmen.
Mit der ESC-Taste rufen wir eine neue Aufgabe auf. (Siehe später Hinweise auf mögliche Erweiterungen.)
Und jetzt wird ausgegeben:
s ist ein Skalierungsfaktor, der seine Wirkung zeigt, wenn wir vom PC-Schirm auf das Handheld wechseln.
 	
Eine ähnliche Ausgabe haben wir schon im vorigen Beispiel schon kennen gelernt.
 	
Hier wird der blaue Rahmen um das Antwortfeld gezeichnet.
	function on.escapeKey()
	local ch=(var.recall("check") or 0)
	local neuer=(var.recall("neu")or 1)
	var.store("zeile3",""); var.store("neu",neuer+1)
	antwort=""; var.store("check",0)
	screen:invalidate(); end
	
	function on.paint(gc)
	local h = screen:height(); local b = screen:width()
	local s=math.floor(h/212+0.5)
	local tab = {}
	for k = 1, 4 do tab[k]=(var.recall("Zeile"..k)
	or "Zeile"..k)
	gc:setFont("sansserif","b",12*s)
	if k ==1 then
	gc:setColorRGB(20,20,138)
	else
	gc:setColorRGB(158,5,8)
	end
	zb = gc:getStringWidth(tab[k])
	zh = gc:getStringHeight(tab[k])
	gc:drawString(tab[k],b/2-zb/2,
	h*k/(5)+zh/2)
	gc:setColorRGB(20,20,138)
	gc:setPen("medium","smooth")
	gc:drawRect(0.1*b,h*3/5-s*15-5,0.8*b,0.2*h)
	end
	screen:invalidate(); end

Mögliche Erweiterungen:

- Erweiterung der Zahlenfelder (auch negative Zahlen)
- Einführung eines Schiebereglers zur Wahl der Rechenart
- Textausgabe, wenn das Ergebnis falsch ist
- Ausgabe des richtigen Ergebnisses nach drei Fehlversuchen
- …

Wie schon oben erwähnt, lässt sich das komplette Programm in Lua erstellen. Die Zufallszahlen werden z.B. mit einer Funktion aus der Mathematik-Bibliothek von Lua erzeugt:

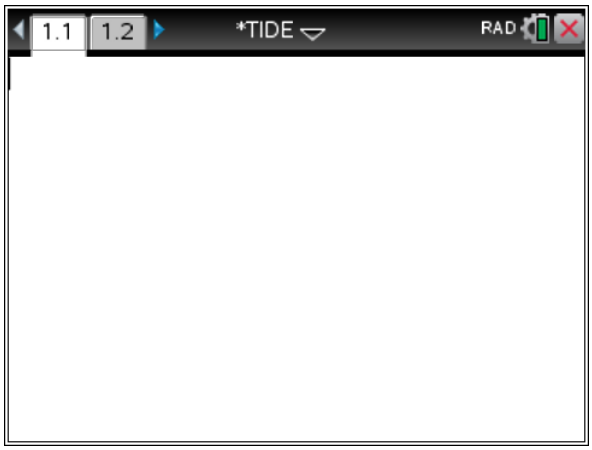
```
a = math.random(0,20).
```

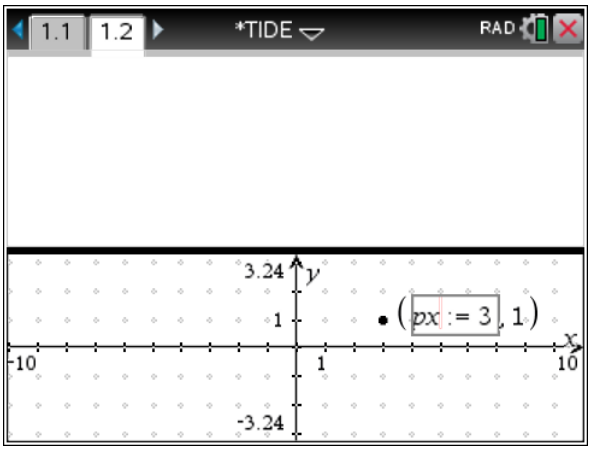
1.5 Ein „Hüpfpunkt“ im Grafikfenster

Bevor wir wirklich in die Grafikprogrammierung mit Lua einsteigen, wollen wir uns ansehen, wie Lua mit der TI-NspireCAS-Grafik kommuniziert. Wir werden ein Skript erzeugen, mit dessen Hilfe wir die Bewegung eines Punktes in der Koordinatenebene nur mit den Pfeiltasten steuern können.

Weil es schnell gehen soll, verwenden wir die TIDE.tns-Datei (TIDE ist ein Lua-Editor (geschrieben Nadrieril Feneanar, https://tiplanet.org/forum/archives_voir.php?id=19303). TIDE.tns kann von http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut6.zip bezogen werden (sowohl leer als auch ausgefüllt).

Wir können das Skript sofort auf die erste Seite 1.1 des Dokuments eintragen und es wird automatisch auf der zweiten Seite 1.2 ausgeführt.

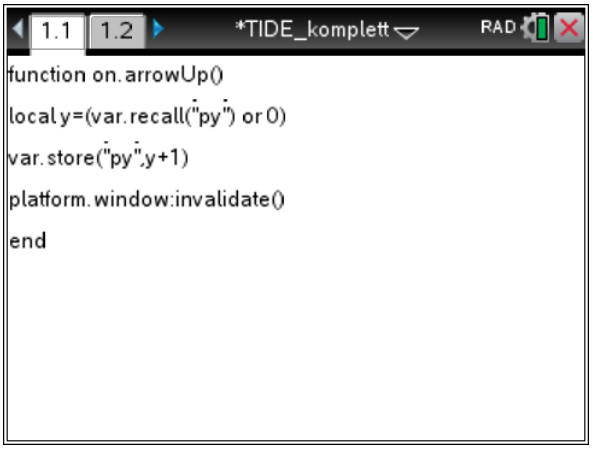




Wir markieren einen Punkt auf dem Koordinatengitter, erhalten mit einem rechten Mausklick seine Koordinaten, die wir dann als ***px*** und ***py*** speichern. Jetzt kann es losgehen!

Ich weiß nicht, wie es Ihnen geht. Ich finde es lästig, einen Punkt im Grafikfenster zu fassen und dann auf einen anderen Gitterpunkt zu verschieben. Ich habe es auch schon mit Schiebereglern versucht, aber da muss ich wieder den Regler anklicken und dann mit den Pfeiltasten ansprechen.

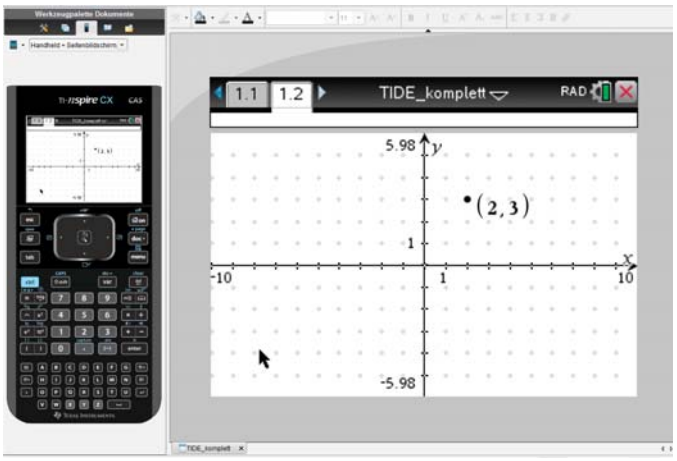
Hier werden gleich nur die Pfeiltasten verwendet: wir gehen zurück auf Seite 1.1 und geben das nebenstehende Skript ein.



Was geschieht hier? Lua „holt sich“ (*recalls*) den Wert der Variablen ***py*** und speichert ihn lokal als ***y***. Wenn es keinen Wert gibt, wird 0 gespeichert. ***y*** wird um 1 erhöht und auf ***py*** zurück geschrieben. Damit rückt der Punkt um eine Einheit nach oben. Nach jeder Änderung soll das Display aufgefrischt werden.

Nun können wir dieses Skript dreimal kopieren und für die anderen Bewgungen anpassen: `arrowDown` für $y-1$, `arrowLeft` für $x-1$ und `arrowRight` für $x+1$. Beachten Sie bitte die Groß- und Kleinschreibung.

Eleganter ist es, alle Bewegungen in einer Funktion zusammenzufassen:




```
function on.arrowKey(key)
  if key == "down" then
    local y = (var.recall("py") or 0)
    var.store("py",y-1)
  elseif key == "up" then
    local y = (var.recall("py") or 0)
    var.store("py",y+1)
  elseif key == "left" then
    local x = (var.recall("px") or 0)
    var.store("px",x-1)
  else
    local x = (var.recall("px") or 0)
    var.store("px",x+1)
  end
platform.window:invalidate()
end
```

Die obere Hälfte wurde verkleinert, aber diese muss aktiviert sein, wenn die Pfeiltasten gedrückt werden.

In der Geometrie-Seite kann ganz normal gearbeitet werden, die Steuerung des Punktes muss aber von Lua aus erfolgen.

Wie wir mit der Steuerung mit einer Maus umgehen, werden wir im Kapitel über die Klassen erfahren.

1.5.1 Noch etwas Feinarbeit

An dieser Stelle muss nochmals erwähnt werden, dass bei der Entwicklung von Skripts, die jegliche Art von Änderungen im Display beinhalten, die Bildwiederholungsrate des Handheld Unterstützung benötigt. Skripts, die perfekt am Computer laufen, können am Handheld scheitern, da es über ein wesentlich einfacheres Beriebssystem mit weniger Signalen und Abfragen im Hintergrund verfügt. Grundsätzlich müssen wir das **platform.window:invalidate()**-Kommando für jede Funktion, die eine sichtbare Veränderung bewirkt einsetzen.(invalidate heißt eigentlich „ungültig machen“, d.h. offensichtlich, dass der alte Schirm „erneuert“ werden soll.) Für viele Anwendungen wird es günstig sein, die folgenden beiden Funktionen am Beginn zu definieren:

```
function on.construction()
timer.start(1/5)
end
function on.timer()
platform.window:invalidate()
end
```

Können wir erkennen, was sie bewirken? Sobald die Seite erzeugt ist, wird eine Zeitschaltuhr (oder Taktgeber) aktiv, die fünfmal in der Sekunde tickt. Die zweite Funktion erneuert damit fünfmal pro Sekunde das Display. All dies passiert im Hintergrund, und man kann die Häufigkeit dieses Auffrischens beliebig definieren. Allerdings kann dies bei besonders dichten Skripts zu Darstellungsproblemen führen, also gehen wir damit vorsichtig um.

Das wäre so weit alles.

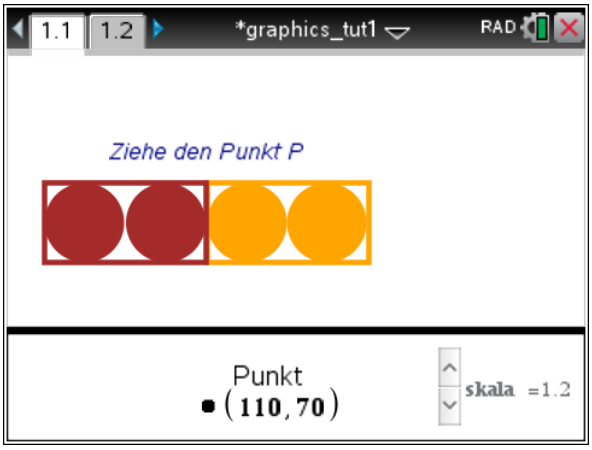
Es folgen drei Ideen für mögliche Erweiterungen:

- Speichern Sie die Schrittweite in beiden Richtungen auf den Achsen anstelle von 1.
- Setzen Sie **enterKey** für einen sinnvollen Zweck ein. Im Beispielprogramm wird mit on.enterKey() der Punkt in den Ursprung zurück geschickt.
- Sie können das Display in etwa gleichen Hälften lassen und in der Notes-Seitenhälfte einen informativen Text zeigen (z.B. die Koordinaten des Punktes, den Quadrant, in dem sich der Punkt befindet, oder eine andere Eigenschaft.)

(Siehe auch TIDE_script_tut6.tns und TIDE_script_tut6_1.tns in http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut6.html).

2 Nach dem Schreiben folgt das Zeichnen

Bevor wir fertige Grafiken und Bilder importieren wollen wir Grafiken, die aus geometrischen Figuren bestehen selbst erzeugen. Wir erinnern uns, dass die Auflösung des TI-NspireCAS Handheld Displays 318 × 212 Bildpunkte (Pixels), des PC-Schirms (auf meinem z.B.) 942 × 628 Pixels beträgt.



Jeder Grafikbefehl muss innerhalb des **graphic contexts** aufgerufen werden. D.h., dass also innerhalb der uns bereits bekannten **function on.paint(gc)** jeder Befehl mit **gc:** eingeleitet werden muss.

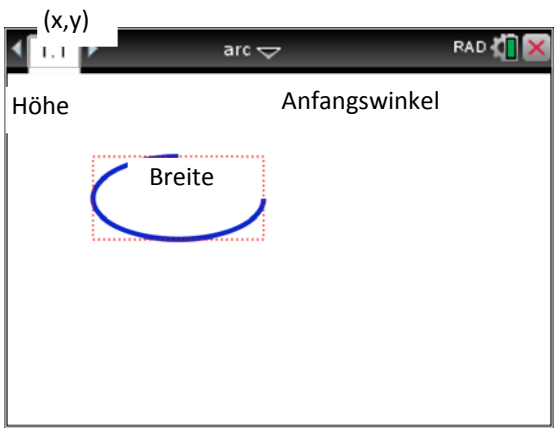
Man kann einen Grafikbefehl auch innerhalb einer selbst definierten Funktion mit gc als Argument verwenden. Diese Funktion muss aber auch innerhalb von on.paint(gc) aufgerufen werden.

Graphic context stellt uns die folgenden Befehle zur Verfügung:

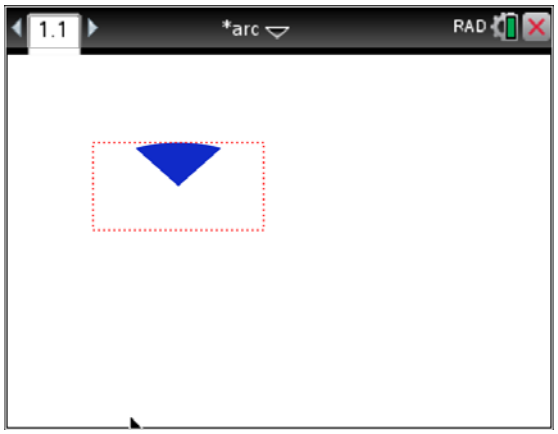
- **drawLine**(xstart, ystart, xende, yende) – zeichnet eine Strecke zwischen den Punkten mit den Koordinaten (*xstart*,*ystart*) und (*xende*,*yende*).
- **drawRect**(x, y, xbreite, yhöhe) – zeichnet ein Rechteck *xbreite* × *yhöhe* mit der linken oberen Ecke im Punkt (*x*,*y*).

- **fillRect**(x, y, xbreite, yhöhe) – füllt das Rechteck in der zuletzt definierten Farbe.
- **drawPolyLine**({x1, y1, x2, y2,..,xn, yn, x1, y1}) – zeichnet einen Polygonzug, das die als geordneten Paare gegebenen Punkte verbindet. Um ein geschlossenes Vieleck zu zeichnen, muss der Anfangspunkt am Ende der Liste wiederholt werden.
- **fillPolygon**({x1, y1, x2, y2,..,xn, yn, x1, y1}) – füllt das Vieleck – es muss geschlossen sein.
- **drawArc**(x, y, Breite, Höhe, Anfangswinkel, Drehwinkel) – zeichnet einen (elliptischen) Bogen mit gegebener Breite und Höhe an der Stelle (*x*,*y*), der den gegebenen Winkel umfasst. Ein Kreis wird geformt, wenn Breite = Höhe und der Winkel von 0 bis 360 reicht.
- **fillArc**(x, y, Breite, Höhe, Anfangswinkel, Drehwinkel) – füllt den gegebenen Sektor oder Kreis.

  Die beiden Abbildungen illustrieren die Bedeutung der Parameter.



drawArc(50, 50, 100, 50, 90, 270)



fillArc(50, 50, 100, 50, 60, 60)

- **drawString**(string, x, y [, Position]) – platziert eine Zeichenkette an die Stelle (*x*,*y*). Position ist der Verankerungspunkt des Strings mit den Möglichkeiten "bottom","middle" oder "top".
- **getStringWidth**(string) – gibt die Breite (Länge) des Strings in Pixel zurück.
- **getStringHeight**(string) – gibt die Höhe des Strings in Pixel zurück.

  (Der String muss natürlich zwischen " " angegeben werden.)

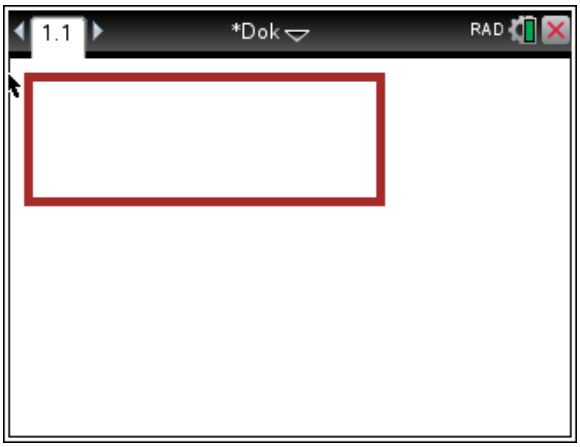
Weitere nützliche Grafikbefehle sind:

- **isColorDisplay()** – gibt den Wert 1 zurück, wenn eine Farbe vorliegt, sonst 0.
- **setAlpha**(0 ≤ Ganzzahl ≤ 255) – setzt die Transparenz.
- **setColorRGB**(rot, grün, blau): die RGB-Werte sind ganze Zahlen von 0 bis 255.
- **setFont**(font, typ, Größe) - font: ("sanserif", "serif",..), typ: ("b", "r", "i", "bi") für fett (*bold*), normal (*regular*), kursiv (*italic*) und kursiv fett (*bold italic*). Größe ist eine ganzeZahl.
- **setPen**(Stärke, Stil): Größe ("thin", "medium", "thick"), Stil ("smooth", "dotted", "dashed").

2.1 Ein einfaches Rechteck – mit Inhalt)

Wir haben zwar schon einmal – unkommentiert – ein Rechteck gezeichnet, aber jetzt soll das genauer erklärt werden: wir verwenden oben angeführte gc-Funktionen.

```
-- neue Funktion Rechteck
-- basiert auf drawRect
function Rechteck(x,y,breite,hoehe,gc)
  gc:drawRect(x,y,breite,hoehe)
end
-- jetzt wird gezeichnet
function on.paint(gc)
  gc:setColorRGB(165,42,42)
  gc:setPen("thick","smooth")
  Rechteck(10,10,200,70,gc)
end
```

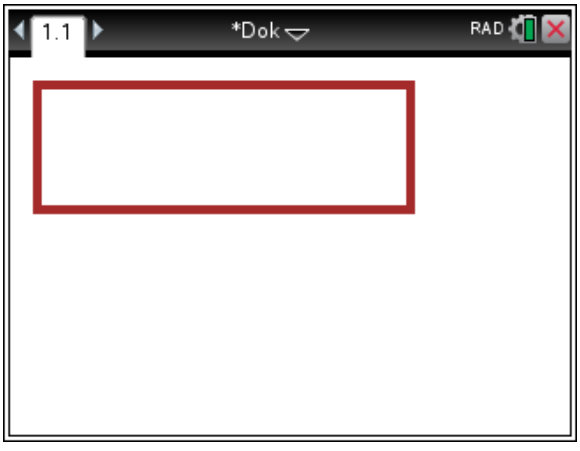


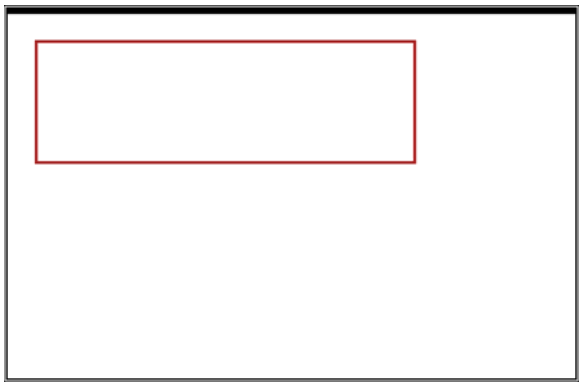
Wenn wir mit diesem Lua-Skript auf einem PC arbeiten, sehen wir ein verhältnismäßig kleines Rechteck in der linken oberen Ecke des Displays. Das ist der Nachteil, wenn wir die Parameter in absoluten (Pixel-) Größen angeben. Wollen wir eine „Multi-Plattform-Anwendung“ erzeugen, müssen wir die Ausgabe flexibler machen. Am einfachsten ist es, die „window“-Größen width() und height() als Referenzgrößen zu wählen. Das könnte dann etwa so aussehen:

Die beiden Abbildungen unten zeigen das Display des Handheld (links) und des PC (rechts). Leider geht die Strichstärke nicht entsprechend mit.

Ein – allerdings umständlicher – Umweg wäre, das Rechteck seinerseits aus vier gefüllten Balken zusammen zu setzen.

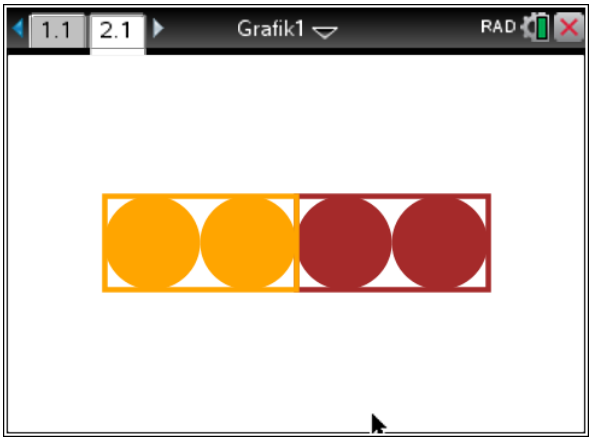
```
local screen=platform.window
function Rechteck(x,y,breite,hoehe,gc)
  gc:drawRect(x,y,breite,hoehe)
end
function on.paint(gc)
-- b und h müssen hier definiert werden
  local b,h=screen:width(),screen:height()
  gc:setColorRGB(165,42,42)
  gc:setPen("thick","smooth")
  Rechteck(b/20,b/20,2*b/3,2*b/9,gc)
end
```





```
local screen=platform.window
function Rechteck(x,y,breite,hoehe,gc)
  gc:drawRect(x,y,breite,hoehe)
end
function on.paint(gc)
  local b,h=screen:width(),screen:height()
  gc:setColorRGB(165,42,42)
  gc:setPen("medium","smooth")
  Rechteck(b/2,3*h/8,h/2,h/4,gc)
  gc:fillArc(b/2,3*h/8,h/4,h/4,0,360)
  gc:fillArc(b/2+h/4,3*h/8,h/4,h/4,0,360)
  gc:setColorRGB(255,165,0)
  Rechteck(b/2-h/2,3*h/8,h/2,h/4,gc)
  gc:fillArc(b/2-h/4,3*h/8,h/4,h/4,0,360)
  gc:fillArc(b/2-h/2,3*h/8,h/4,h/4,0,360)
end
```

Wir werden nun, angeregt durch das Bild am Beginn dieses Abschnitts, die beiden Rechtecke mit den gefüllten Kreisen in die Mitte des Displays setzen:



Anregungen: Lassen Sie sich zu weiteren Darstellungen von ihrer Fantasie oder von Bildern aus Ihrer Umwelt inspirieren und erzeugen Sie komplexere Grafiken, die aus einfachen geometrischen Formen bestehen, wie z.B. Firmenlogos (Mastercard, …) oder zeichnen Sie eine Verkehrsampel oder …

Zwei Herausforderungen: Versuchen Sie, wie schon in einem vorigen Abschnitt geschehen, die Position der Box mit einem variablen Punkt im Grafikfenster zu verknüpfen, so dass Sie mit dem Bewegen des Punkts auch die Box im Lua-Fenster bewegen können.

Sie können auch eine Skalierungsvariable einführen und damit die Box vergrößern und verkleinern.

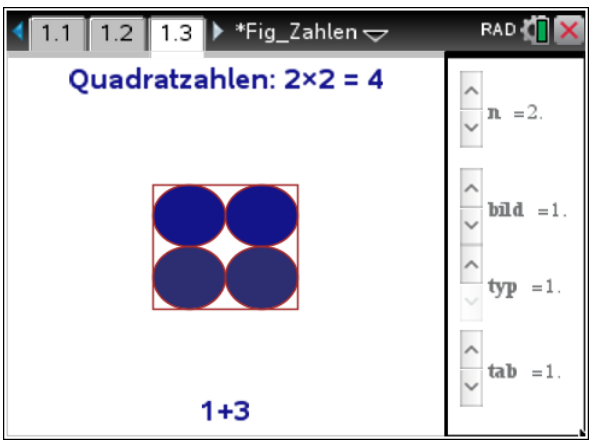
In http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut6.html ist die Realisierung einer dieser Aufgaben zu finden.

2.2 Mit Grafik zu den „Figurierten Zahlen“

Jetzt wollen wir alles das, was wir bis jetzt gelernt haben zusammenfassen. Wir beginnen mit der grafischen Darstellung von Quadrat-, Rechtecks- und Dreieckszahlen.

Dabei werden wir neben den Schiebereglern auch parallel die Pfeiltasten, ENTER, ESC und TAB verwenden, dass wir auf allen Plattformen arbeiten können.

↑ und ↓ steuert den Wert von *n*, ← und → wechseln den Typ (Quadrat-, Rechtecks- und Dreieckszahlen).



ENTER wechselt die Darstellung (*bild*) und mit TAB kann man die Entstehung der figurierten Zahlen verfolgen. So sind die Quadratzahlen die Summen der ungeraden Zahlen, die Rechteckszahlen die Summen von geraden Zahlen und die Dreieckszahlen ergeben sich als die Summen der natürlichen Zahlen (alle ohne 0). Mit ESC können wir Entwicklung der Reihen wieder zurückverfolgen. Laden Sie die Datei Fig_Zahlen.tns (erzeugt von Steve Arnold) und experimentieren Sie damit.

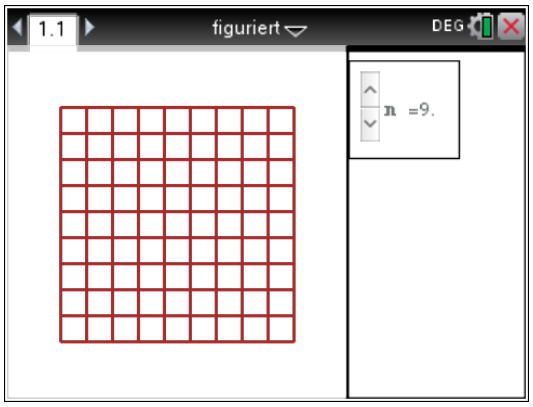
2.2.1 Wir stellen das Gitter zusammen

Wir beginnen mit der Definition einer Funktion zeichneGitter, die ein Gitter von $n \times n$ Quadraten zeichnet. Das Gitter soll in der Mitte des Displays zu liegen kommen. Ein schmaler Rand um das Gitter soll frei bleiben. Daher teilen wir die Breite und Höhe des Displays durch *n* (ein wenig mehr, wegen des Rands). Für *n* führen wir einen Schieberegler ein – weitere werden folgen. Das Display wird geteilt in ein Geometry-Fenster (für den Schieberegler) und das Lua-Fenster.

Beschreibung der Aktion	Lua-Skript

Betrachten wir zeichneGitter doch etwas genauer: als Argumente werden der aktuelle Wert für *n*, die Koordinaten für den Ankerpunkt des ersten Quadrats, sowie Breite und Höhe jeder Zelle übernommen.
drawLine arbeitet dann mit den Laufvariablen *k* und *m* für die Zeilen und Spalten des Gitters.
Erst mit paint wird das Ergebnis der Funktion gezeichnet. Mit Breite und Höhe des Displays und der Zellenanzahl werden der Startpunkt, sowie die Dimensionen der Zellen erst berechnet und dann nach Wahl der Farbe und Strichstärke durch den Aufruf von zeichneGitter auf den Schirm gebracht.
	platform.apilevel = '2.2'
	local screen=platform.window
	function on.construction()
	  timer.start(1/5)
	end
	function on.timer()
	  screen:invalidate()
	end
	function zeichneGitter(nummer,x,y,br,hoe,gc)
	  for k=0, nummer do
	  for m = 0, nummer do
	  gc:drawLine(x,y+hoe*m,x+br*nummer,
	  y+hoe*m)
	  gc:drawLine(x+br*k,y,x+br*k,
	  y+hoe*nummer)
	end; end; end
	function on.paint(gc)
	  br=screen:width();h=screen:height()
	  num=var.recall("n") or 1
	  xval=math.floor(br/(num+4))
	  yval=math.floor(h/(num+4))
	  x=br/2-num*xval/2;y=h/2-num*yval/2
	  gc:setPen("thin","smooth")
	  gc:setColorRGB(165,42,42)
	  zeichneGitter(num,x,y,xval,yval,gc)
	end

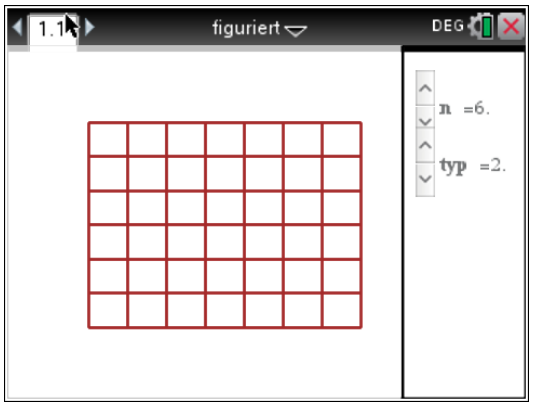
Dieses Skript erzeugt nun ein dynamisches quadratisches Gitter, dessen Größe durch die Variable *n* (definiert in der TI-Nspire-Umgebung) gesteuert wird.



2.2.2 Das Gitter wird variiert

Wir werden jetzt dieses Gitter in zweifacher Weise abändern: es soll auch ein Rechtecks- und ein Dreiecksgitter geben. Bei den Rechteckszahlen ist eine Rechtecksseite um jeweils eins länger als die andere.

Wir führen für den Typ der darzustellenden figurierten Zahlen den Schieberegler *typ* mit den möglichen Werten 1, 2 oder 3 ein. Im Programm verzweigen wir dann nach dem Typ ab: if typ ==1 then (Quadratzahlendarstellung) elseif typ == 2 then (Rechteckszahlendarstellung mit Zeilen der Länge n+1 untereinander) else (Dreieckszahlendarstellung mit Zeilen jeweils um ein Quadrat gekürzt und nach rechts verschoben) end.

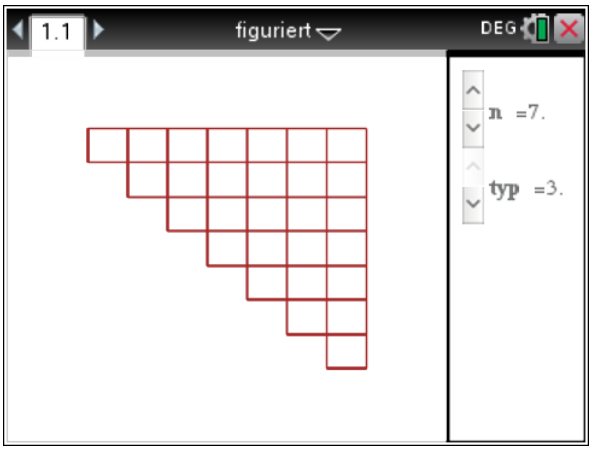


Bei den Dreieckszahlen läuft die zweite Variable von *k* bis *n*. Damit wird die Figur aber nicht geschlossen. Die obere und rechte Begrenzung fehlen. Es müssen zwei Strecken zusätzlich mitgezeichnet (mit drawLine) werden. Studieren Sie die bis-herigen Befehle und versuchen Sie, die beiden Strecken selbst zu erzeugen, bevor Sie das im Skript nachlesen.

```
function zeichneGitter(nummer,x,y,br,hoe,gc)
  typ=(var.recall("typ")or 1)
  if typ == 1 then
… wie oben
  end
  elseif typ == 2 then
  for k=0, nummer+1 do
    for m = 0, nummer do
    gc:drawLine(x,y+hoe*m,x+br*(nummer+1),
    y+hoe*m)
    gc:drawLine(x+br*k,y,x+br*k,y+hoe*m)
    end
  end
  else
  gc:drawLine(x,y,x+br*(nummer+1)-br,y)
  gc:drawLine(x+br*nummer,y,x+br*nummer,
  y+hoe*nummer)
  for k=1, nummer do
    for m = k, nummer do
    gc:drawLine(x+br*(m-1),y+hoe*m,
    x+br*nummer,y+hoe*m)
    gc:drawLine(x+br*(k-1),y,x+br*(k-1),
    y+hoe*k)
    end
  end
  end
end
function on.paint(gc)
…
…
```

Im nächsten Schritt werden wir das Programm weiter verbessern, indem wir die Zahlen zusätzlich durch Kreise dynamisch entstehen lassen.

Außerdem werden wir die Steuerung auch über die Tastatur (Pfeil- und andere Tasten) möglich machen.



2.2.3 Unsere Zahlen machen eine gute Figur

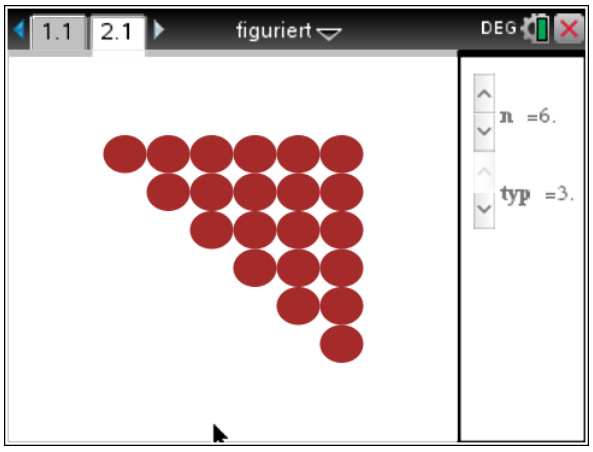
Nachdem es uns gelungen ist, einige figurierte Zahlen zu visualisieren, wollen wir weitere bereits bekannte Techniken anwenden, um die Darstellungsmöglichkeiten zu erweitern, die Steuerung auch über die Tastatur zu ermöglichen und das Entstehen der Zahlenfolge dynamisch zu verdeutlichen.

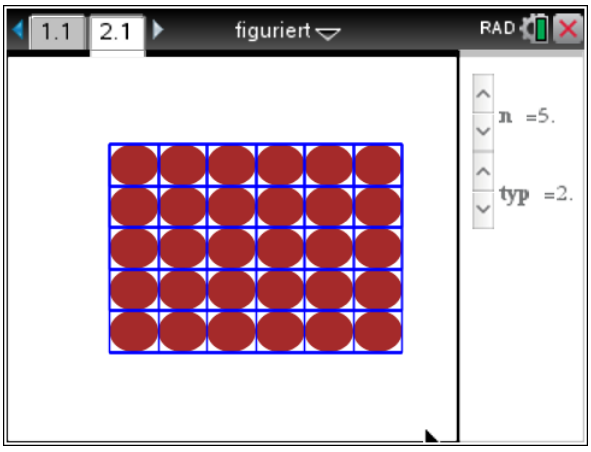
Angenommen, wir wollen anstelle des Gitters Kreise zur Darstellung der entstehenden Muster verwenden? Was müssten wir da ändern? Im Musterbeispiel, das bezogen werden kann unter http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut9.html kann man mit einem Schieberegler (*view*) zwischen den Präsentationsarten wechseln. (Das soll als Übung für den Einsatz einer if-Konstruktion für Sie offen gelassen bleiben). Hier wollen wir der Einfachheit halber das Gitter durch Kreise ersetzen, bzw. Kreise in die Quadrate setzen.

Das erfordert relativ wenig Änderungen in unserem Skript: am Einfachsten ist es, die zeichneGitter-Funktion zu kopieren und unter die erste anzufügen, sie umzubenennen, z.B.in zeichneKreise und alle drawLine-Kommandos in drawArc- oder fillArc-Kommandos mit den entsprechenden Parametern zu ersetzen.

Innerhalb der on.paint-Funktion können wir dann zeichneGitter entweder durch zeichneKreise ersetzen (linkes Bild unten) oder beide Funktionen bestehen lassen, wobei auch die Farbe für das Gitter oder für die Kreise geändert werden kann .

```
function zeichneKreise(nummer,x,y,br,hoe,gc)
  typ=(var.recall("typ")or 1)
  if typ == 1 then
  for k=0, nummer-1 do
    for m = 0, nummer-1 do
      gc:fillArc(x+br*m,y+hoe*k,br,hoe,0,360)
  end; end
  elseif typ == 2 then
  for k=0, nummer-1 do
    for m = 0, nummer do
      gc:fillArc(x+br*m,y+hoe*k,br,hoe,0,360)
  end; end
  else
  for k=1, nummer do
    for m = k, nummer do
      gc:fillArc(x+br*(m-1),
        y+hoe*(k-1),br,hoe,0,360)
end: end; end; end
```



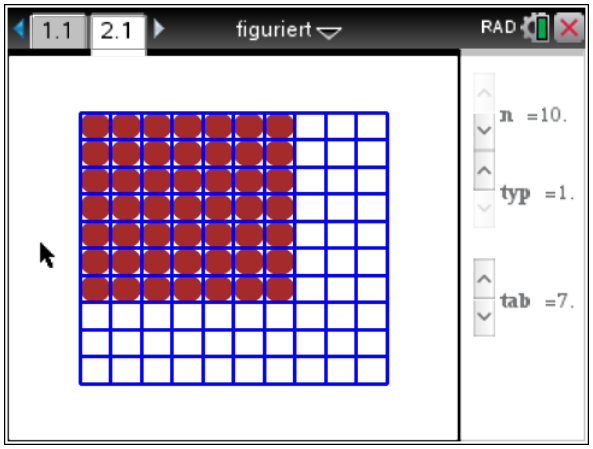


Nun geht es weiter zum dynamischen Display.

Um die Enststehung der figurierten Zahlen im Muster zu verfolgen bedarf es nur weniger Änderungen im Skript.

Bis jetzt wurden alle Kreise bis zur Nummer *n* gefüllt – eben im Umfang des Gitters. So führen wir eine neue Variable *tab* ein, die die Anzahl der auszuführenden Schritte steuert.

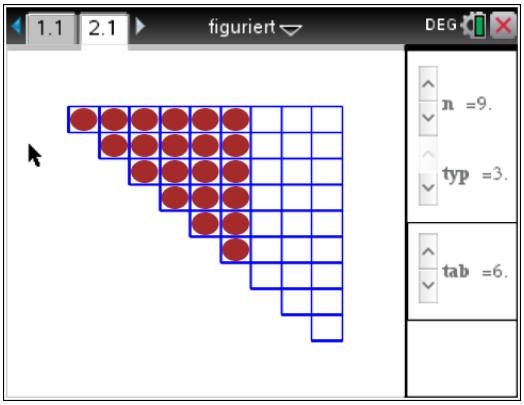
Gemeinsam mit einem Schieberegler für *tab* reichen die beiden neuen, bzw. geänderten Zeilen (in rot) innerhalb der on.paint-Funktion, um unser Ziel zu erreichen.




```
function on.paint(gc)
  br=screen:width();h=screen:height()
  num=var.recall("n") or 1
  tabs=var.recall("tab") or 1
  xval=math.floor(br/(num+4))
  yval=math.floor(h/(num+4))
  x=br/2-num*xval/2;y=h/2-num*yval/2
  gc:setPen("thin","smooth")
  gc:setColorRGB(165,42,42)
  zeichneKreise(tabs,x,y,xval,yval,gc)
  gc:setColorRGB(0,0,255)
  zeichneGitter(num,x,y,xval,yval,gc)
end
```

Sie können versuchen, die Kreise etwas kleiner zu machen, dann würde das Bild nicht so gedrängt aussehen.

Alles, was noch zu tun übrig bleibt, ist die Erweiterung der Steuerung über die Tastatur (und eventuell die Hinzunahme einer Beschriftung, wie zu Beginn des Abschnitts 2.2 gezeigt).



2.2.4 Letzte Kosmetik

Die Eingabe über die Tastatur macht die Eingabe möglicherweise bequemer als über die Schieberegler.

Wir wählen ↑ und ↓ für die Steuerung von *n*, mit → und ← wechseln wir den *typ*, mit der Tabulatortaste erhöhen wir mit dem Wert für *tab* die Anzahl der gezeigten Kreise und mit der ESC-Taste nehmen wir diesen wieder zurück.

Wir müssen beachten, dass wir bei der Eingabe über die Tastatur – anders als beim Schieberegler – Grenzen für die Werte eingeben müssen.

Wenn wir das nicht tun, dann führt die ↓-Taste bald in den negativen Bereich und die ↑-Taste weit über unser Gitter hinaus. Auch für den *typ* sind nur sehr beschränkte Werte möglich.

Wie man einen dynamischen Text eingeben kann, können wir dem Lua-Skript in Shape_patterns.tns auf Steve Arnolds Webseite entnehmen.

Der Code der restlichen Tastatursteuerung folgt:

```
function on.arrowDown()
  num=var.recall("n") or 1
  if num > 0 then
    var.store("n",num-1)
  else
    var.store("n",1)
  end
end
function on.arrowUp()
  num=var.recall("n") or 1
  var.store("n",num+1)
end
```

```
function on.arrowLeft()
  typ=var.recall("typ") or 1
  if typ > 1 then
    var.store("typ",typ-1)
  else
    var.store("typ",1)
  end
end
function on.arrowRight()
  typ=var.recall("typ") or 1
  if typ < 3 then
    var.store("typ",typ+1)
  else
    var.store("typ",3)
  end
end
```

```
function on.tabKey()
  tabs=var.recall("tab") or 1
  num=var.recall("n") or 1
  if tabs < num then
    var.store("tab",tabs+1)
  else
    var.store("tab",tabs)
  end
end
function on.escapeKey()
  tabs=var.recall("tab") or 1
  if tabs>0 then
    var.store("tab",tabs-1)
  else
    var.store("tab",0)
  end
end
```

Mögliche Erweiterungen wären: Eine Erweiterung auf „allgemeine“ Rechteckzahlen, wobei man mit einem Rechteck $n \times (n+k)$ arbeitet und für k einen weiteren Schieberegler einführt.

Studieren Sie die figurierten Zahlen (siehe die Referenzen) und überlegen Sie, ob und wie es möglich wäre z.B. andere Darstellungen für die vorliegenden Zahlen bzw. auch Pentagonal- und andere Zahlen zu visualisieren.

Unterlagen zu den figurierten Zahlen finden Sie z.B. in:

https://www.lern-online.net/mathematik/arithmetik/zahlen/figurierte-zahlen/

http://www.mayhematics.com/p/figurate.pdf

http://www.ziegenbalg.ph-karlsruhe.de/materialien-homepage-jzbg/materials-in-english/FigurateNumbers.pdf

3 Wir importieren fertige Grafiken

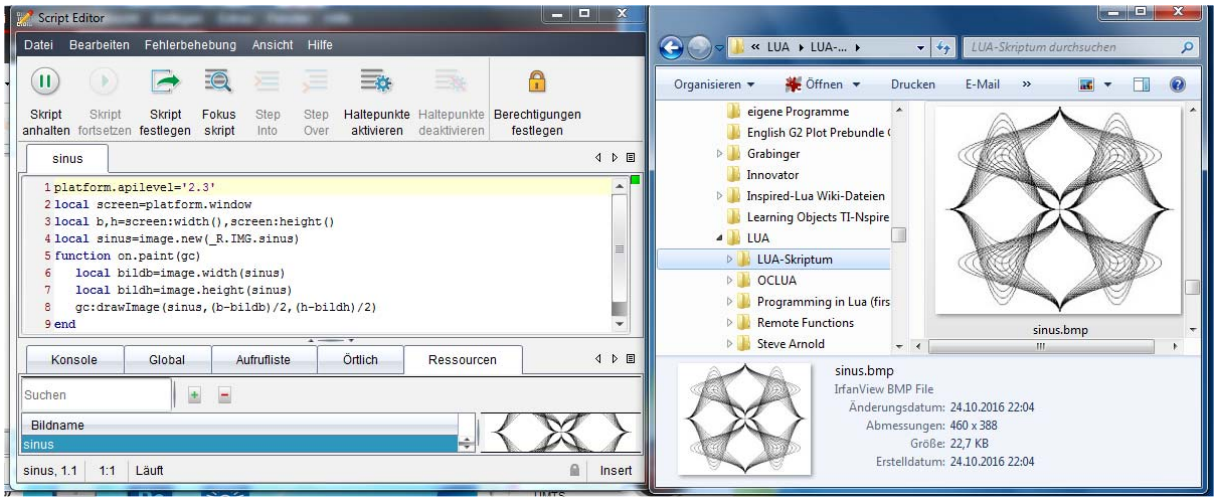
Steve Arnold dankt Adrien Bertrand für seine Unterstützung bei der Vorbereitung dieses Tutorials (http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut7.html).

Adrien Bertrand's Website ist https://inspired-lua.org/index.php/author/adriweb/.

Lua verfügt über einfache, aber sehr mächtige Kommandos zur Bildmanipulation. Man kann wohl mit TI-NspireCAS auch Bilder importieren, jedoch gibt es dann keine weiteren Möglichkeiten, mit diesen Bildern zu arbeiten – sie stehen fix als Hintergrund. Mit Lua lässt sich die Größe verändern, die Bilder können auftauchen und wieder verschwinden, ja, und sie können sogar bewegt werden.

In früheren API-Versionen wurden die Bilder in einem digitalen Code als String eingelesen und verarbeitet. Dies verbrauchte viel Speicherplatz und verlangsamte den Ablauf der Programme.

Jetzt werden Bilder als Dokument „Ressourcen" behandelt. Sie haben es möglicherweise noch nicht bemerkt – außer es wurden Ihnen schon Fehler bei der Eingabe angezeigt -, dass sich am unteren Rand des Editors fünf Tabellenreiter befinden: *Konsole* (mit der hat man es beim Verfassen eines Skripts am meisten zu tun), *Global*, *Aufrufliste*, *Örtlich* (eine etwas missglückte Übersetzung von Locals und ***Ressourcen***. Zurzeit sind als einzige Ressourcen Bilddateien möglich. Die meisten Grafikformate (JPG, TIFF, PNG, BMP, …) werden unterstützt. Klicken Sie einfach auf den "+"-Knopf, navigieren Sie zur gewünschten Bilddatei und vergeben Sie für das Bild einen Namen, unter dem das Bild innerhalb des Programms angesprochen wird.



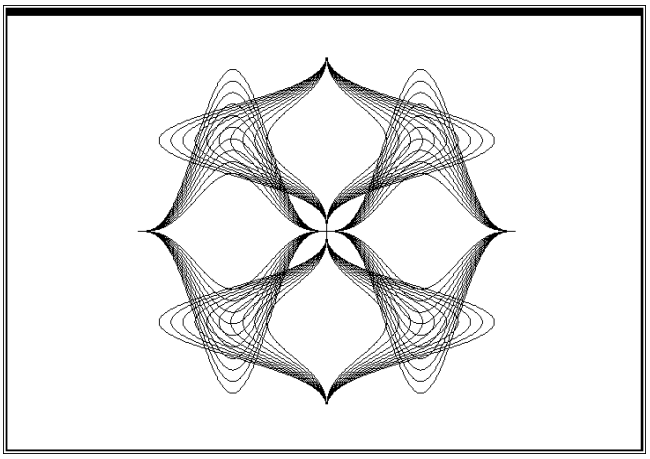
Das Bild darf höchstens 3145728 Pixels groß sein, muss also eventuell in seiner Größe (in Pixel) reduziert werden.

Wenn das Bild in die Ressourcen aufgenommen ist, muss es ins Skript übernommen werden, wie unten gezeigt. Der Namen der Bilddatei ist mit "_R.IMG." einzuleiten. Wir wollen das Bild in die Mitte des Bildschirms setzen und arbeiten mit den Dimensionen der Grafik, die wir mit image.width(bild) und image.height(bild) ermitteln könnten. (Siehe dazu das Tutorial 7 in Steve Arnold's Scripting Tutorials). Hier wird eine andere Möglichkeit gezeigt.

Wie Sie unten im Skript sehen, wird das Bild im Rahmen der schon bekannten on.paint-Funktion mit dem Kommando drawImage in das Skript importiert.

Nachdem uns kein Fehler im Skript (in der Konsole) angezeigt wird, gehen wir auf Fokus Skript und sehen uns das Ergebnis an, zuerst am PC und dann auf dem Handheld – oder in der Handheld-Ansicht auch am Computer:

Das hat offensichtlich funktioniert, aber am Handheld sieht das nicht so gut aus:



Uns ist schon vorhin gelungen, die Ausgabe an die Ausgabeplattform anzupassen. Außerdem sollen Skalierung und Bewegung der Bilder dynamisiert werden.

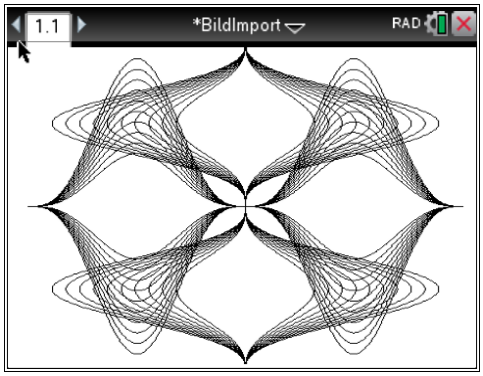
An dieser Stelle soll darauf hingewiesen werden, dass diese Vorgangsweise nicht die effektivste ist, aber sie ist ein geeigneter Anfang. In weiterer Folge werden wir neben der paint.on-Funktion, die bei großen Bilddateien den Programmablauf sehr verlangsamt, andere Möglichkeiten finden. Ein besserer Weg ist es, das copy-Kommando mit zugehörigen Ereignissen zu verbinden wie z.B. on.resize, on.arrowKey und anderen.

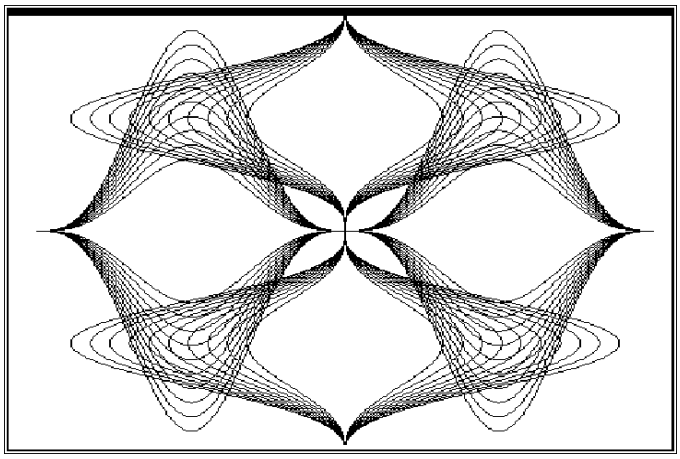
Beschreibung der Aktion	Lua-Skript
Angabe der Lua-Version (ab 2.3) Wir definieren lokale Variable. on.resize übernimmt Aufgaben von on.paint. Eine Kopie des Bilds, das genau auf das Display passt wird erzeugt. resize wird im Gegensatz zu paint nur dann aufgerufen, wenn die Seite erzeugt oder umskaliert wird.	platform.apilevel='2.3' local screen=platform.window local b,h local sinus=image.new(_R.IMG.sinus) local bildb,bildh function on.resize() b,h=screen:width(),screen:height() sinus=sinus:copy(b,h) -- 2. Möglichkeit: --sinus=sinus:copy(h,0.9*h) bildb=sinus:width(sinus) bildh=sinus:height(sinus)

Für Handheld und größere Bilder wird diese Methode empfohlen.	screen:invalidate() end function on.paint(gc) gc:drawImage(sinus,(b-bildb)/2,(h-bildh)/2) end

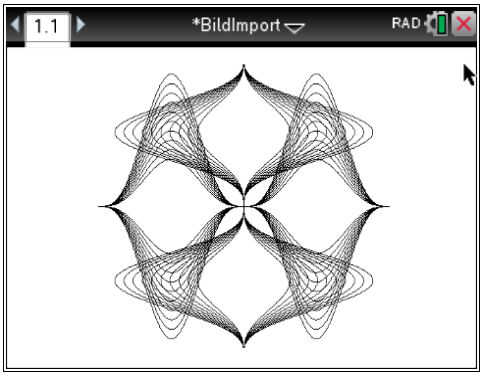
In einem Kommentar (in rot) ist eine zweite Möglichkeit bereits codiert, die eine „schönere" Ausgabe am Display nach sich zieht, wie wir auf der nächsten Seite sehen können.

Nun passt alles ins Display, wird aber auf Schirmgröße verzerrt (vergleichen Sie mit der Bildausgabe oben):

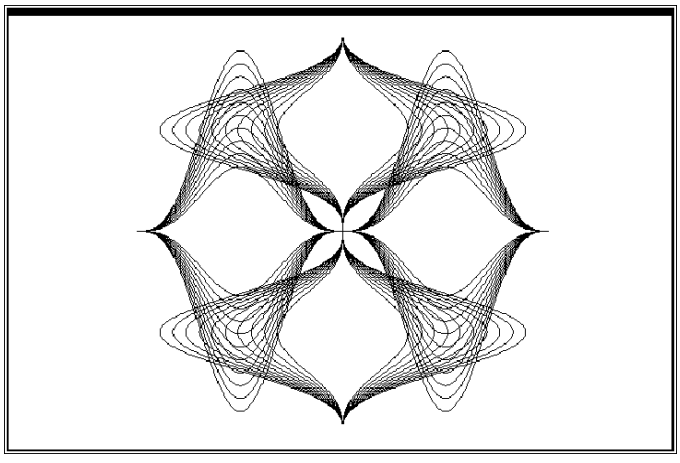




Dies ist die Ausgabe mit der zweiten Möglichkeit:



Das sieht ohne Verzerrung schon viel besser aus!



Natürlich könnten wir nun im Skript so lange herumprobieren, bis das Bild schön aufs Display passt, aber befriedigend ist diese Lösung sicher nicht.

Nun, bis jetzt haben wir noch keine Bewegung in die Grafik(en) gebracht. Das wird nun ausgiebig nachgeholt. Wir werden drei Bilder in unseren Vorrat (die Ressourcen) aufnehmen und diese Bilder – vorerst über die Tastatur (←,→) – hintereinander auf unser Display bringen und sie auch mit ↑, ↓ vergrößern und verkleinern können.

3.1 Wir bringen Bewegung(en) rein!

Ich habe drei eigene Bilder von Alpenblumen – geeignet verkleinert, um das Pixelvolumen nicht zu überschreiten – in die Ressourcen geladen: Enzian, Almrausch (Alpenrose) und natürlich Edelweiß.

Betrachten Sie bitte vorerst die roten Programmteile nicht.

Beschreibung der Aktion	Lua-Skript
Der übliche Beginn: Wir deklarieren die Variablen als „local“.	platform.apilevel='2.3' local screen=platform.window local b,h,skala,wahl,bildbild,bildb,bildh local bilder={image.new(_R.IMG.Alm), image.new(_R.IMG.Enz),image.new(_R.IMG.Edel)}

Die Bilder werden in eine „lokale“ Liste geladen.	var.monitor("skala"); var.monitor("wahl")
	
	function on.varChange()
	  skala=var.recall("skala") or skala
	  wahl=var.recall("wahl") or wahl
	  neuskal(bilder[wahl],skala)
	end

Wir passen die Bilder an die Größe des Displays an und legen Anfangswerte für skala und wahl fest. Wir starten mit der halben Bildgröße. Das muss eventuell adaptiert werden.
#bilder gibt die Dimension der Bilderliste an.
	function on.resize()
	  b,h=screen:width(), screen:height()
	  skala=1; wahl=1
	  for k=1,#bilder do
	  bilder[k]=bilder[k]:copy(0.5*b,0.5*h)
	  end
	
	  var.store("skala",skala); var.store("wahl",wahl)
	  neuskal(bilder[wahl],skala)
	end

neuskal erzeugt ein „Display-Bild“, das mit dem gewählten Skalierungsfaktor skala – innerhalb sinnvoller Grenzen – erzeugt wird.	function neuskal(bild,skala)
	  if skala > 0.1 and skala < 4.9 then
	  bildbild=bild:copy(skala*bild:width(),
	  skala*bild:height())
	  bildb,bildh=bildbild:width(),bildbild:height()
	  end
	  screen:invalidate()
	end

Die Esc-Taste fungiert praktisch als ein Reset, um den Anfangszustand wieder herzustellen	function on.escapeKey() on.resize() end

Beschreibung der Aktion	Lua-Skript

Hier wird die Steuerung durch die Pfeiltasten festgelegt:
Up und Down für die Skalierung (Zoom) und Right und Left für den Bildwechsel.
 	
Das %-Zeichen entspricht dem mod in anderen Sprachen. Mit dieser Formulierung werden die drei Bilder in zyklischer Folge auf den Schirm gebracht.
 	
Das gewählte Bild wird entsprechend der Skalierung aufbereitet und …
 	
… endlich in der Mitte des Schirms dargestellt.
 	
Wenn wir nun die Datei öffnen (hier heißt sie BildImport.tns) zeigt sich das erste Bild und nach ein paarmal ↑ füllt die Alpenrose das ganze Display.
In der Computeransicht kann man noch einige Male vergrößern.
 	
Die anderen Blumen zeigen wir gleich.
	function on.arrowKey(taste)
	--skala=var.recall("skala") or skala
	if taste=="up" then
	if skala <4.9 then
	skala=skala+0.1
	end
	elseif taste=="down" then
	if skala > 0.1 then
	skala=skala-0.1
	end
	elseif taste=="right" then
	wahl=(wahl+1)%3+1
	elseif taste=="left" then
	wahl=wahl%3+1
	end
	var.store("skala",skala); var.store("wahl",wahl)
	neuskal(bilder[wahl],skala)
	end
	
	function on.paint(gc)
	gc:drawImage(bildbild,(b-bildb)/2,(h-bildh)/2)
	end
	 

3.1.1 Mit Schiebereglern auch für das iPad

Dieses Skript funktioniert einwandfrei sowohl am Handheld als auch am PC. Auf dem iPad gibt es keine Pfeiltasten, daher ist diese Steuerung nicht möglich. Eine einfache Lösung bietet sich mit der Installierung von Schiebereglern an. Dazu sind nur geringe Ergänzungen im vorliegenden Skript einzufügen – und natürlich die Schieberegler einzurichten.

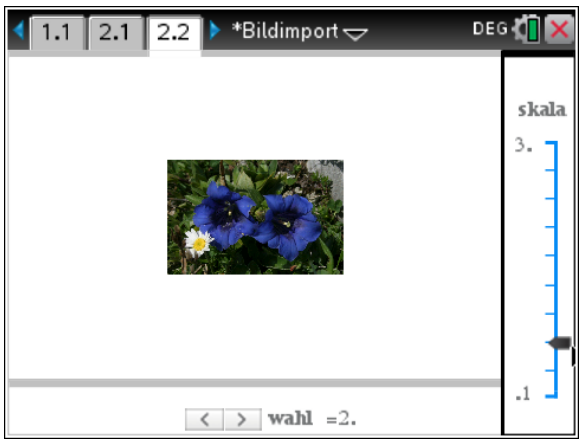
Diese Ergänzungen sind im obigen Lua-Skript rot geschrieben.

var.monitor veranlasst das Programm, diese Variablen (*skala* und *wahl*) zu beobachten, während on.varChange veranlasst, was bei einer Änderung dieser Variablenwerte zu tun ist. Die geänderten Werte werden mit var.store gespeichert und mit var.recall bei Bedarf wieder zum Gebrauch abgeholt.

Natürlich muss der Schirm geteilt werden, da wir zumindest eine Geometry-Seite für die Schieberegler brauchen.

Ich nehme zwei Geometry Seiten (rechts und links unten) und passe deren Größe an. Ein Regler steuert die Skalierung, der andere die Bildwahl.
(0.1 ≤ *skala* ≤ 3 und 1 ≤ *wahl* ≤ 3)

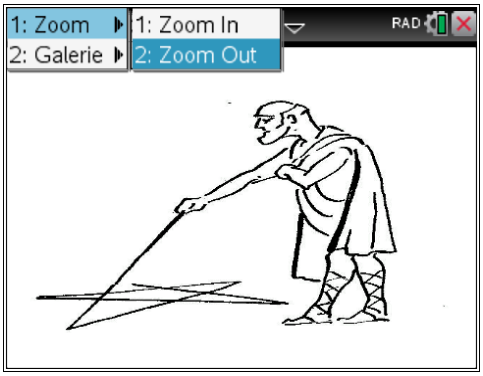
Jetzt zeigen wir unsere Miniversion der Alpenflora:

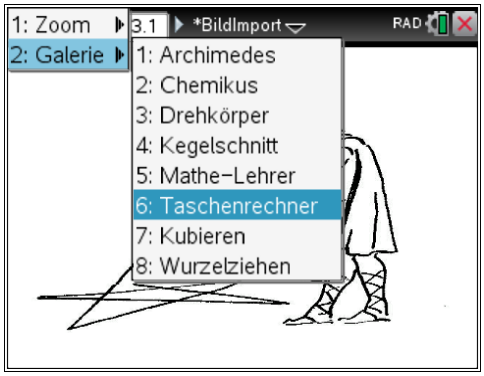


Die Steuerung über die Pfeiltasten bleibt aktiv – wenn wir das Fenster mit dem Bild anklicken.

3.1.2 Auch eine Menüsteuerung ist möglich

Aus den Zeiten eines TI-92 weiter bis zum Voyage 200 waren wir es gewohnt, Programme auch über Menüs zu steuern. Das ist auch mit Lua realisierbar:





Ich habe acht Zeichnungen (meines Vaters) eingescannt und sie unter den Namen c1 bis c8 in die Ressourcen aufgenommen.

Beschreibung der Aktion	Lua-Skript
Nach Definition der lokalen Variablen wird die Bildliste geladen.	platform.apilevel='2.3' local screen=platform.window local b,h,bild,bildb,bildh,skala local bilder={image.new(_R.IMG.c1), image.new(_R.IMG.c2),image.new(_R.IMG.c3), image.new(_R.IMG.c4),image.new(_R.IMG.c5), image.new(_R.IMG.c6),image.new(_R.IMG.c7), image.new(_R.IMG.c8)}
Das sind das Startbild (könnte auch ein Titelbild sein) und der Anfangswert für die Skalierung.	bild = bilder[1]
Die Funktionen wahl1() bis wahl8() werden im Menü „Galerie“ aufgerufen.	function wahl1() bild=bilder[1] on.resize() end function wahl2() bild=bilder[2] on.resize() end function wahl3() bild=bilder[3] on.resize() end function wahl4() bild=bilder[4] on.resize() end function wahl5() bild=bilder[5] on.resize() end function wahl6() bild=bilder[6] on.resize() end function wahl7() bild=bilder[7] on.resize() end function wahl8() bild=bilder[8] on.resize() end
Hier beginnen die Zoom-Funktionen.	skala=var.recall("skala") or 1 function ZoomIn() skala=(var.recall("skala")+0.1) or 1 var.store("skala",skala); on.resize() screen:invalidate(); end

on.resize() kennen wir schon.

Das ist nun wiederum neu:

So wird das Menü (in zwei „Gängen“) mit Unterteilungen erzeugt:

Zuerst das Menü „Zoom“ …

… und dann das Menü „Galerie“:

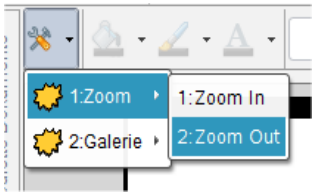
Jeder String ist mit einer Funktion gekoppelt, die wir oben definiert haben.

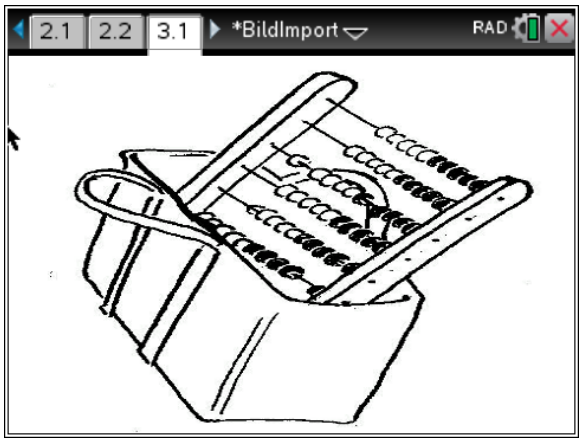
Und damit wird das Menü dann endgültig hergestellt.

```
function ZoomOut()
  skala=(var.recall("skala")-0.1) or 1
  var.store("skala",skala); on.resize()
  screen:invalidate(); end

function on.resize()
  b,h=screen:width(),screen:height()
  bild=bild:copy(skala*b,skala*h)
  bildb,bildh=bild:width(),bild:height()
  screen:invalidate(); end

menu={
    {"Zoom",
     {"Zoom In", function() ZoomIn() end},
     {"Zoom Out", function() ZoomOut() end},
    },
    {"Galerie",
     {"Archimedes", function() wahl1() end},
     {"Chemikus", function() wahl2() end},
     {"Drehkörper", function() wahl3() end},
     {"Kegelschnitt", function() wahl4() end},
     {"Mathe-Lehrer", function() wahl5() end},
     {"Taschenrechner", function() wahl6() end},
     {"Kubieren", function() wahl7() end},
     {"Wurzelziehen", function() wahl8() end},
    },
   }
toolpalette.register(menu)

function on.paint(gc)
  gc:drawImage(bild,(b-bildb)/2,(h-bildh)/2)
end
```

Ich habe das nun so in den Script-Editor eingegeben – endlich gab es keine Fehlermeldung mehr und das erste Bild ist tatsächlich am PC-Schirm aufgetaucht – aber wo sind die Menüs?

Die Menüs „verstecken“ sich unter den Werkzeugen. Am Handheld müssen sie die „Menütaste“ – welche sonst? – drücken und dann erscheinen beide wie gewünscht und funktionieren auch

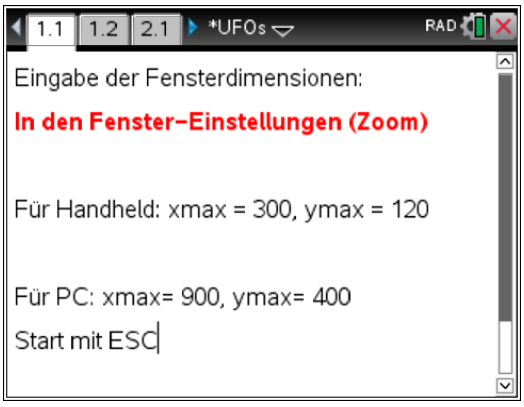


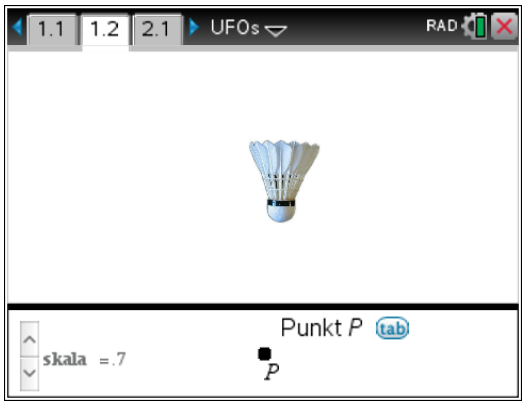


Hier sehen wir dann den Taschenrechner gezoomt auf Handheldgröße.

3.1.3 Im Flug über das Display

Wie können wir ein Objekt am Display bewegen? Mit einem Punkt ist uns dies mit Hilfe der Pfeiltasten bereits gelungen (1.4.1). Eine weitere feine Möglichkeit ist, den Schirm zu teilen und im unteren Teil ein Graphs-Fenster zu installieren. Die Bewegung der Grafik kann jetzt über Ziehen des Punktes *P* erfolgen. Dazu müssen die Koordinaten des Punktes (z.B. *px* und *py*) gespeichert und geeignet verknüpft werden.





Wenn das Lua-Fenster aktiviert ist, erfolgt die Steuerung über die Pfeiltasten. Die Werte für *xmax* und *ymax*, bzw. für die Schrittweite werden über die Fenstereinstellungen des Graphs-Fensters vorgenommen.

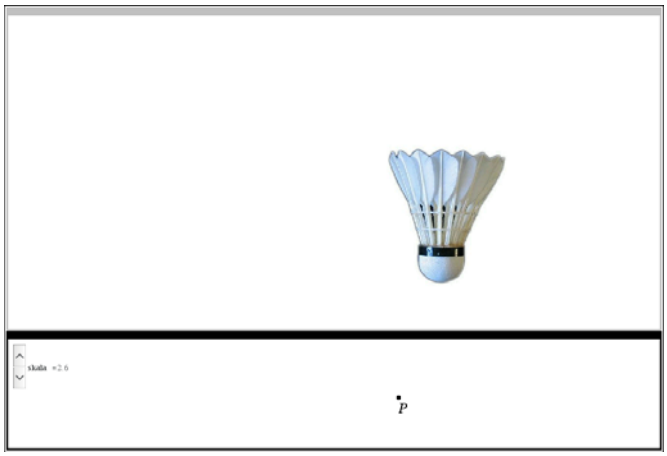
Mit dem Schieberegler variieren wir die Größe des Objekts.

Wir lassen hier den Federball von Steve Arnold über den Bildschirm huschen. Die englische Bezeichnung für dieses Objekt ist recht lustig: „*shuttlecock*“.

Rechts sehen wir die Darstellung am PC. Wenn wir den Punkt hinaufbewegen, dann kommt der Federball nach unten und umgekehrt.

Mit der ESC-Taste setzen wir den „*Schüttelhahn*“ etwa in die Mitte des Feldes.

Für die Codierung müssen wir nichts Neues lernen.



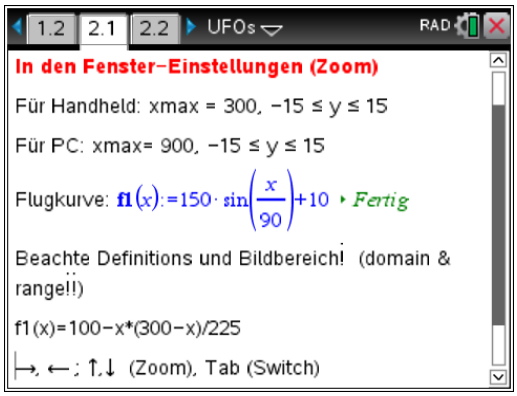
```
platform.apilevel='2.3'
local screen=platform.window
function on.timer()
  screen:invalidate();end
function on.create()
  timer.start(1/10);end

local b,h,skala,bildb,bildh,bildbild
local bild0=image.new(_R.IMG.c00)
skala = 0.5

var.monitor("skala")
var.monitor("px"); var.monitor("py")

function on.varChange()
  skala = var.recall("skala") or skala
  x=var.recall("px") or 0; y=var.recall("py") or 0
  xt=var.recall("xtick") or 0
  yt=var.recall("ytick") or 0
  neuskal(bild0, skala)
end

x=var.recall("px") or 0; y=var.recall("py") or 0
xt=var.recall("xtick") or 10
yt=var.recall("ytick") or 10

function on.arrowKey(taste)
  if taste == "down" then
    local y = (var.recall("py") or 0)
    var.store("py",y+yt)
  elseif taste == "up" then
    local y = (var.recall("py") or 0)
    var.store("py",y-yt)
  elseif taste == "left" then
    local x = (var.recall("px") or 0)
    var.store("px",x-xt)
  else
    local x = (var.recall("px") or 0)
    var.store("px",x+xt)
  end
screen:invalidate()
end

function on.escapeKey()
   local b,h=screen:width(),screen:height()
   neuskal(bild0,0.5)
   imb=0.75*image.width(bild0)
   imh=image.height(bild0)
   var.store("px",(b-imb)/2)
   var.store("py",(h-imh)/2)
   screen:invalidate()
end

local b,h=screen:width(),screen:height()
bild0=bild0:copy(skala*0.55*b,skala*h)

function neuskal(bild,skala)
  bildbild=bild:copy(0.75*skala*bild:width(),
                     skala*bild:height())
  bildb,bildh=bildbild:width(),bildbild:height()
screen:invalidate()
end

function on.paint(gc)
  neuskal(bild0, skala)
  b,h=screen:width(),screen:height()
  x=var.recall("px") or 0; y=var.recall("py") or 0
  gc:drawImage(bildbild, x,y)
end
```

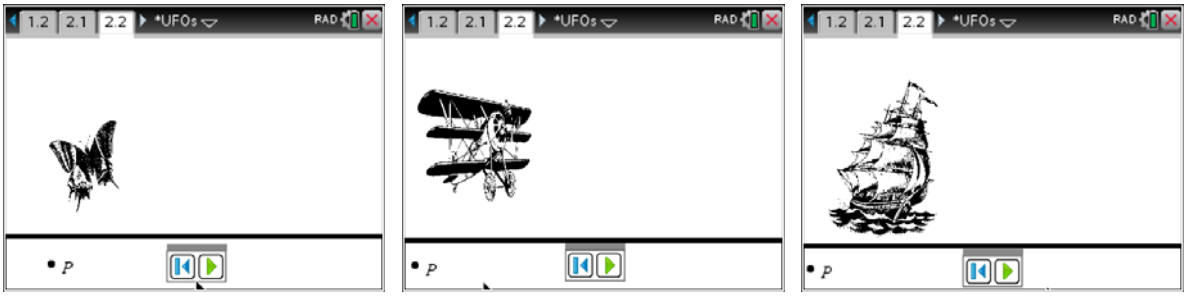
Zum Abschluss dieses Abschnitts lassen wir einige „UFOs“ auf einer vorgegebenen Bahn über das Display fliegen.

Ich habe 13 Clipart-Grafiken in die Ressourcen aufgenommen, die ich der Reihe nach mit der Tabulator-Taste aufrufen kann.

Die Gleichung der Flugbahn wird über die Notes eingegeben.

Sehen wir uns drei Figuren am Handheld an:



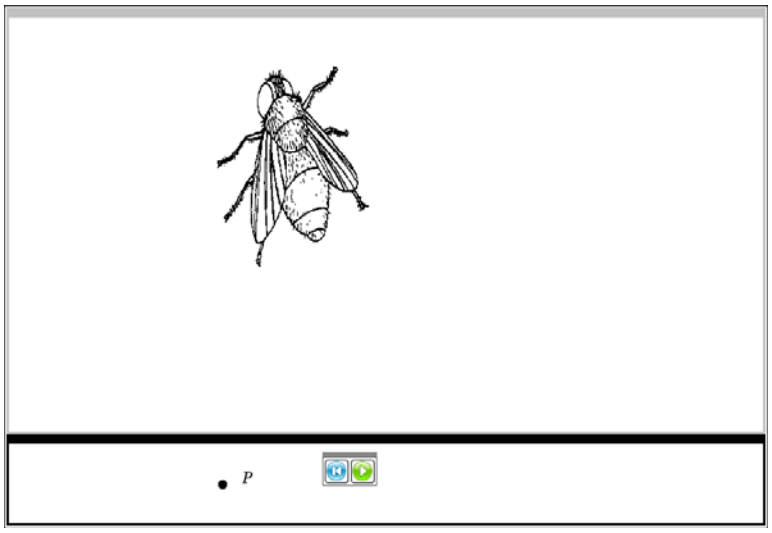


Die Bewegung kann auf drei Arten bewirkt werden: durch Ziehen des Punkts *P*, mit Links- und Rechts-Pfeiltaste im Lua-Fenster oder durch Start der Animation.
Auch hier gibt es keine wesentlichen Neuheiten im Lua-Code.

```
platform.apilevel='2.3'
local screen=platform.window
function on.timer()
  screen:invalidate();end
function on.create()
  timer.start(1/10);end

local b,h,skala,wahl,bildb,bildh,bildbild
local bilder={image.new(_R.IMG.c00),image.new(_R.IMG.c01),image.new(_R.IMG.c02),
image.new(_R.IMG.c03),image.new(_R.IMG.c04),image.new(_R.IMG.c05),image.new(_R.IMG.c06),
image.new(_R.IMG.c07),image.new(_R.IMG.c08),image.new(_R.IMG.c09),image.new(_R.IMG.c10),
image.new(_R.IMG.c11),image.new(_R.IMG.c12)}

skala=0.5; wahl=1

var.store("skala",skala);var.store("wahl",wahl);var.monitor("px")

function on.resize()
  b,h=screen:width(),screen:height()
  for k=1,#bilder do
    bilder[k]=bilder[k]:copy(0.55*b,h); neuskal(bilder[wahl],skala)
end;end

function on.varChange()
  x=var.recall("px") or 0;y=var.recall("py") or 0;xt=var.recall("xtick") or 10
neuskal(bilder[wahl],skala);end
```

```
skala = var.recall("skala") or skala
x=var.recall("px") or 0; y=var.recall("py") or 0; xt=var.recall("xtick") or 10

function on.arrowKey(taste)
  if taste == "down" then
    if skala >0.2 then
    skala = skala-0.1;end
  elseif taste == "up" then
    if skala < 2 then
    skala = skala+0.1;end
  elseif taste == "left" then
    local x = (var.recall("px") or 0); var.store("px",x-xt)
  else
    local x = (var.recall("px") or 0)
  var.store("px",x+xt);end
  var.store("skala",skala); neuskal(bilder[wahl],skala)
screen:invalidate();end

function on.tabKey()
  wahl=(wahl+1)% 13 +1;var.store("wahl",wahl);neuskal(bilder[wahl],skala)
screen:invalidate(); end

function neuskal(bild,skala)
  bildbild=bild:copy(skala*0.75*bild:width(),skala*bild:height())
  bildb,bildh=bildbild:width(),bildbild:height()
screen:invalidate();end

function on.paint(gc)
  neuskal(bilder[wahl],skala); b,h=screen:width(),screen:height()
  x=var.recall("px") or 0; y=var.recall("py") or 0
gc:drawImage(bildbild, x,y);end
```

Und zum Schluss machen wir noch die Fliege am PC-Display.



4 Ab jetzt mit Klasse(n)!

In Kapitel 2 haben wir ein ganz gut brauchbares Dokument zur Darstellung der figurierten Zahlen entwickelt. Die Steuerung duch den Benutzer erfolgte ausschließlich über die Tastatur. Das muss für das Arbeiten am Handheld auch ausreichen.

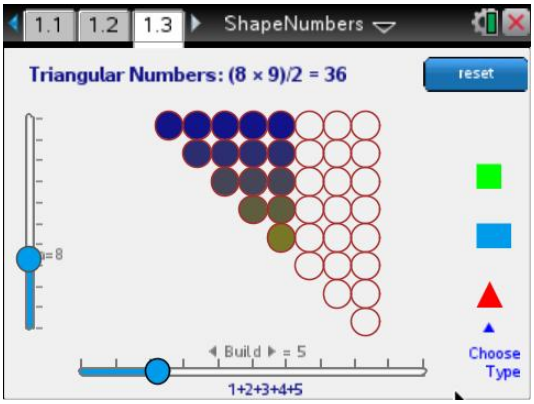
Natürlich haben wir dann auch noch die Schieberegler. Aber, wie Sie sicher gemerkt haben, das Hin- und Herschreiben der Werte zwischen TI-Nspire und Lua ist möglicherweise nicht die effektivste Art für die Erstellung der Programme und ihre Ausführung am Gerät.

Sicherlich haben Sie bis jetzt den Einsatz der Maus vermisst. Sie wird wohl bei der Arbeit am Computer am häufigsten verwendet. Wäre es nicht ideal, wenn unsere Produkte auf allen Plattformen optimal eingesetzt werden könnten. Das heißt mit Tastatursteuerung am Handheld und mit der Maus am PC. Dazu kommt der Vorteil, dass wir, wenn wir auf die Schieberegler verzichten können, keine Variablen mehr übertragen müssen und wir können komplett in Lua arbeiten. Das macht die meisten Aufgaben deutlich einfacher.

Unser Projekt zu den figurierten Zahlen könnte dann so aussehen wie rechts abgebildet.

Um dieses Ziel zu erreichen, müssen wir eine Ebene höher in die Lua Programmierung einsteigen und ein wichtiges und mächtiges Werkzeug einführen, die **Klassen**.

(Abbildung aus dem Tutorial#11 von Steve Arnold)



4.1 Klasse Rechtecke

Wir müssen uns *Klassen* quasi als „Superfunktionen“ vorstellen. Die Klassen bringen zusätzliche Vorteile gegenüber den „einfachen“ Funktionen mit sich: so „weiß“ z.B. ein auf diese Weise definiertes Objekt, wo es sich auf dem Schirm befindet, ob es aktiviert ist oder nicht, welche Farbe es haben soll und vieles mehr. Es kann aussagen, ob es ein anderes Objekt oder dessen Koordinatenposition enthält. Können Sie sich jetzt schon vorstellen, wie das nützlich sein kann?

Wenn wir eine Klasse definieren, sorgen wir dafür, dass alle Objekte, die zu dieser Klasse gehören, etwas über sich selbst „wissen“. Der Einsatz von Klassen macht dann richtig Sinn, wenn man viele gleichartige Objekte verwenden will. Da erzeugt man eine Klasse und legt ihre Eigenschaften fest, dann ruft man einfach die Klasse auf und muss nicht wieder für jedes Objekt alle Details angeben.

Wir wollen den Einsatz von Klassen an einem einfachen Beispiel demonstrieren:

Unser Ziel ist, eine paar verschieden große grüne Rechtecke auf das Display zu bringen.

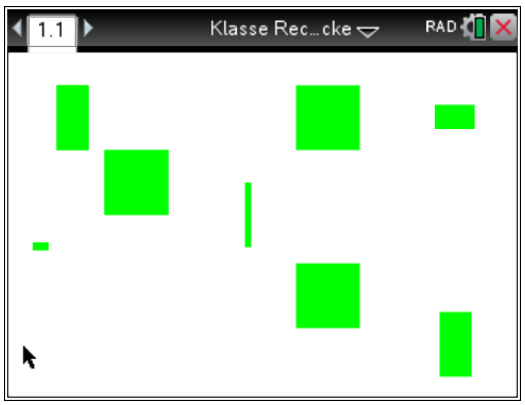
Die Farben werden nicht über die RGB-Werte in bekannter Form definiert, sondern über die entsprechenden Hexadezimalzahlen aus einer bereits bestehenden Klasse geholt.

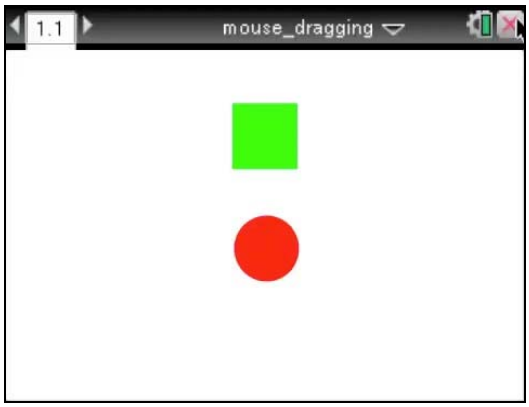
Beschreibung der Aktion	Lua-Skript

Grün aus der Klasse „farbe“	platform.apilevel = '2.3'
	
	farbe = {
	gruen = {0x00, 0xFF, 0x00},
	}

Die Klasse „**Rechteck**“ wird gebildet. Funktionskopf: Initialisierung (init) x ist x in der Klasse y ist y in der Klasse breite ist breite in der Klasse hoehe ist hoehe in der Klasse farbe ist farbe in der Klasse (gruen)	Rechteck = class() function Rechteck:init(x, y, breite, hoehe) self.x = x self.y = y self.breite = breite or 20 self.hoehe = hoehe or 20 self.farbe = farbe.gruen end
Funktionskopf: Inhalt (contains) contains wird später für die Kontrolle der Position des Rechtecks benötigt, wenn wir wissen wollen, ob es (von der Maus) angeklickt wurde. Lokale Variable für breite und hoehe und Grenzen für diese Variablen	function Rechteck:contains(x, y) local b = self.breite local h = self.hoehe return x >= self.x - b/2 and x <= self.x + b/2 and y >= self.y - h/2 and y <= self.y + h/2 end
Funktionskopf: Zeichnen (paint) Der Code für die Farbe wird „ausgepackt“ und in den RGB-Code umgewandelt. Das Rechteck wird eingefärbt.	function Rechteck:paint(gc) gc:setColorRGB(unpack(self.farbe)) gc:fillRect(self.x - self.breite / 2, self.y - self.hoehe / 2, self.breite, self.hoehe) end
Ende der Klasse (alle Eigenschaften wurden beschrieben)	R01 = Rechteck(80, 80, 40, 40) R02 = Rechteck(40, 40, 20, 40) R03 = Rechteck(80, 80, 40, 40) R04 = Rechteck(40, 40, 20, 40) R05 = Rechteck(150, 100, 4, 40) R06 = Rechteck(280, 180, 20, 40) R07 = Rechteck(200, 40, 40, 40) R08 = Rechteck(200, 150, 40, 40) R09 = Rechteck(20, 120, 10, 5) R10 = Rechteck(280, 40, 25, 15)
Hier werden Objekte aus der Klasse „Rechteck“ gebildet.	
Das endgültige Zeichnen auf den Schirm ist nun sehr einfach. Die Dimensionen der Rechtecke sind für den Handheld ausgelegt.	function on.paint(gc) R01:paint(gc); R02:paint(gc); R03:paint(gc) R04:paint(gc); R05:paint(gc); R06:paint(gc) R07:paint(gc); R08:paint(gc); R09:paint(gc); R10:paint(gc); end

Das Ergebnis unseres Skripts ist links zu sehen. Der rechte Screenshot stammt aus den Tutorials von Steve Arnold. Versuchen Sie, das Quadrat und den Kreis (eine zweite Klasse) zu malen.





Da sich hier am Schirm nichts bewegt, brauchen wir die Funktion invalidate() auch nicht. Das wird sich aber gleich ändern, wenn wir ein Rechteck mit der Maus „packen“ und am Display verziehen wollen.

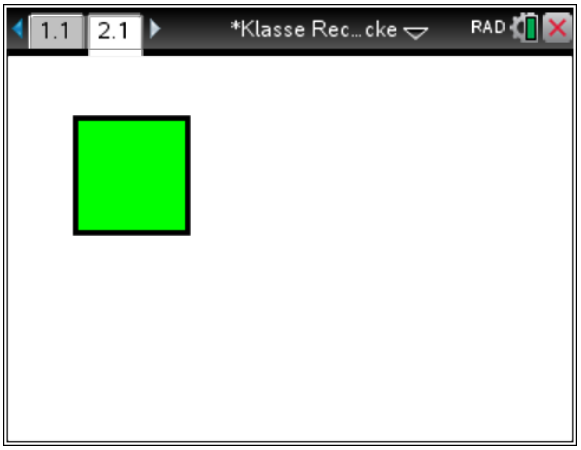
4.1.1 Die Maus packt zu

Um Platz zu sparen, wird der Code wird hier etwas gedrängter geschrieben. Das heisst, dass Befehle anstelle in einer neuen Zeile oft in der selben Zeile durch einen Strichpunkt getrennt geschrieben werden. Das funktioniert im Skript-Fenster ebenfalls.

Beschreibung der Aktion	Lua-Skript

on.resize wird sowohl beim Aufbau der ersten Seite aufgerufen als auch immer dann, wenn die Dimensionen der Seite geändert werden – perfekt.
 	
Die neu geschaffene Variable *ObjWahl* ist am Anfang *leer*. Sie wird dann wichtig, wenn wir mehrere Objekte verfolgen wollen.
 	
Hier beginnen wir mit der Maussteuerung:
 	
Es gibt zwei Aktionen mit der Maus: Klicken (= *Down*) und loslassen (= *Up*).
 	
Jedes Mal wird geprüft ob das gewählte Objekt existiert. Wenn ja, wird der Zustand zuerst wieder auf *leer* gestellt. Bei *Down* wird das gewählte Objekt aktiviert und so bleibt es, solange wir mit der gedrückten Maustaste innerhalb des Objekts bleiben.
 	
Up stellt den *leer*-Zustand wieder her.
	platform.apilevel = '2.3'
	screen=platform.window
	function on.resize()
	  B=screen:width();H=screen:height()
	  Farbe={gruen={0x00, 0xFF, 0x00},}
	  ObjWahl=leer; Qu=Rechteck(B/2,H/2,B/5,B/5)
	end
	
	function on.mouseDown(x,y)
	  if Qu:contains(x,y) then
	  if ObjWahl ~= leer then
	  ObjWahl.gew=false; end
	  ObjWahl=Qu; Qu.gew=true
	  screen:invalidate();end;end
	
	function on.mouseUp(x,y)
	  if ObjWahl ~= leer then
	  ObjWahl.gew=false;end
	ObjWahl=leer;screen:invalidate();end

Beschreibung der Aktion	Lua-Skript

Move verändert laufend die Position des Objekts solange wir die Maustaste gedrückt (= *Down*) lassen.
 	
Wann immer wir die Maus verwenden, wird das Display aufgefrischt. Daher brauchen wir hier den timer nicht.
 	
Das kennen wir schon von vorhin, außer: self.gew ist neu und dient später zur Auswahl, wenn mehrere Klassen zur Verfügung stehen.
 	
In dem Augenblick, in dem wir das Quadrat anklicken wird ein schwarzer Rand um das Quadrat gezeichnet.
 	
Das ist eine nette Ergänzung. Wenn mehrere Objekte vorhanden sind, kann man das jeweils aktive deutlich erkennen.
 	
Der Screenshot zeigt das aktive Quadrat gerade unterwegs über den (Handheld-) Schirm.
	function on.mouseMove(x,y)
	  if ObjWahl ~= leer then
	  ObjWahl.x=x; ObjWahl.y=y
	screen:invalidate();end;end
	
	Rechteck = class()
	function Rechteck:init(x, y, breite, hoehe)
	  self.x = x;self.y = y
	  self.breite = breite or 20
	  self.hoehe = hoehe or 20
	  self.farbe = Farbe.gruen; self.gew=false; end
	
	function Rechteck:contains(x, y)
	  local b = self.breite;local h = self.hoehe
	  return x >= self.x - b/2 and x <= self.x + b/2
	  and y >= self.y - h/2 and y <= self.y + h/2
	end
	
	function Rechteck:paint(gc)
	  gc:setColorRGB(unpack(self.farbe))
	  gc:fillRect(self.x - self.breite / 2,
	  self.y - self.hoehe / 2, self.breite, self.hoehe)
	  if self.gew then
	  gc:setPen("medium","smooth")
	  gc:setColorRGB(0, 0, 0)
	  gc:drawRect(self.x - self.breite / 2,
	  self.y - self.breite / 2, self.breite, self.hoehe)
	end;end
	
	function on.paint(gc)
	  Qu:paint(gc);end
	
	 

4.1.2 Tastatur & Maus United

Nun soll auch noch die Steuerung über die Tastatur integriert werden. Die wesentlichen Befehle kennen wir schon. Sie müssen nur den Einsatz einer oder mehrer Klassen berücksichtigen.

Als Aufputz schreiben wir die Koordinaten (der linken obere Ecke in Pixel) des beweglichen Quadrats ebenfalls aufs Display.

Beschreibung der Aktion	Lua-Skript

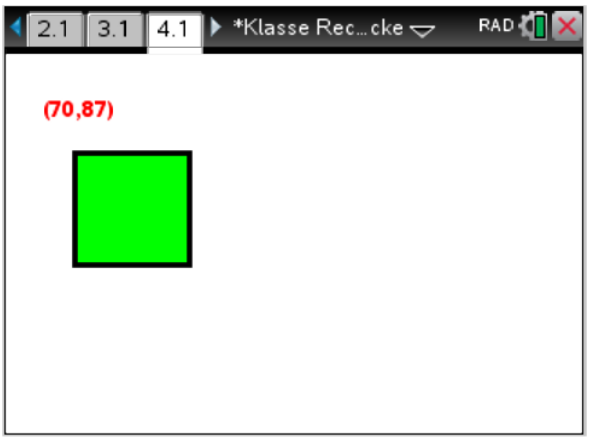
Zuerst berücksichtigen wir die üblichen Pfeiltasten.
 	
Mit der Maus können wir das Quadrat ganz aus dem Schirm schieben. Mit den Tasten wollen wir das nicht erlauben, daher die die „Bremsen“ über die if-Abfragen.
	function on.arrowRight()
	  if ObjWahl ~= leer then
	  if ObjWahl.x < B - ObjWahl.breite/2 then
	  ObjWahl.x = ObjWahl.x + 5
	end; end; end
	
	function on.arrowLeft()
	  if ObjWahl ~= leer then
	  if ObjWahl.x > ObjWahl.breite/2 then
	  ObjWahl.x = ObjWahl.x - 5
	end; end; end

Da helfen uns wieder die anderen Tasten, wie z.B. die TAB-Taste:	function on.arrowDown()
	  if ObjWahl ~= leer then
	  if ObjWahl.y < H - ObjWahl.hoehe/2 then
	  ObjWahl.y = ObjWahl.y + 5
	end; end; end
	
	function on.arrowUp()
	  if ObjWahl ~= leer then
	  if ObjWahl.y > ObjWahl.hoehe/2 then
	  ObjWahl.y = ObjWahl.y - 5
	end; end; end

So weit, so gut – aber wie wechseln wir von der Maus zur Tastatur?
 	
Mit der TAB-Taste wechseln wir zwischen Maus und Tastatur.
	function on.tabKey()
	  if ObjWahl ~= leer then
	  ObjWahl.gew = false
	  end
	  ObjWahl = Qu; Qu.gew = true
	screen:invalidate(); end

Mit ESC geben wir das gewählte Objekt frei.	function on.escapeKey()
	  if ObjWahl ~= leer then
	  ObjWahl.gew= false; ObjWahl = leer
	end; end

Beschreibung der Aktion	Lua-Skript

Und so sollte das aussehen:
 
 	
	function on.paint(gc)
	  Qu:paint(gc)
	  if ObjWahl ~= leer then
	  gc:setFont("sansserif", "b", 10)
	  gc:setColorRGB(255,0,0)
	  gc:drawString("("..ObjWahl.x..",
	  "..ObjWahl.y..")",20,20)
	end; end

4.2 Noch mehr Klasse(n)

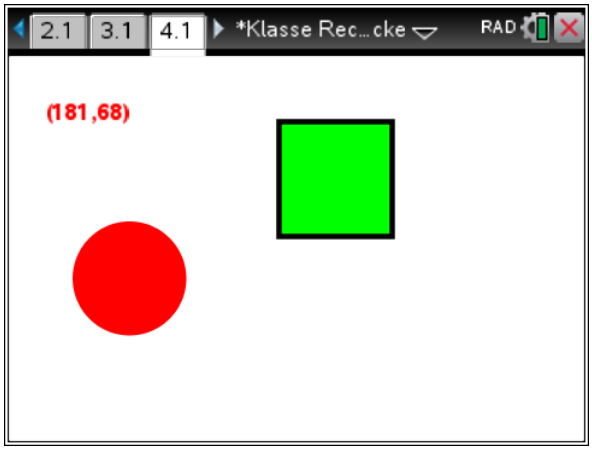
„Und jetzt beginnt der Spaß“ schreibt Steve Arnold in seinem Tutorial#14. Wir werden mit mehreren Objekten arbeiten können. Dazu brauchen wir zumindest eine weitere Klasse. Zusätzlich zu unserer Klasse **Rechteck** definieren wir die Klasse **Kreis**.

Vorerst wollen wir nur die Tastatureingabe (mit „lahm gelegter“ Maus) programmieren:

Beschreibung der Aktion	Lua-Skript

Jetzt sind wir beim Schwerpunkt dieses Abschnitts: der Erzeugung einer Liste der vorhandenen Objekte. (Hier wird sie Objekte genannt.) Um dann von einem Objekt zum anderen wechseln, müssen wir nur einen Schritt in der Tabelle machen.
Jetzt wird auch die Bezeichnung ObjWahl von vorhin klar.
Diese Liste wird gemeinsam mit der zusätzlichen Farbe rot in resize aufgenommen.
Ab hier wird die Klasse „**Kreis“** erzeugt und zudem haben wir zwei Farben:
Die Klasse „Rechteck“ von vorhin gehört natürlich auch in das Skript (und wird hier nicht nochmals abgedruckt).
Das gilt auch für die „arrow“-Funktionen von Abschnitt 4.2.2 (siehe auch in lua_4.tns).
	screen = platform.window
	function on.resize()
	  Farbe = {gruen = {0x00,0xFF,0x00},
	  rot = {0xFF,0x00,0x00},}
	  B = screen:width(); H = screen:height()
	  ObjWahl = leer
	  Objekte = {Kreis(B/2,H/2,B/10,B/10),
	  Rechteck(B/2,H/2,B/5,B/5)}
	End
	
	Kreis = class()
	function Kreis:init(x, y, breite, hoehe)
	  self.x = x; self.y = y
	  self.breite = 2*breite
	  self.hoehe = 2*hoehe
	  self.radius = hoehe
	  self.farbe = Farbe.rot
	  self.gew = false
	end

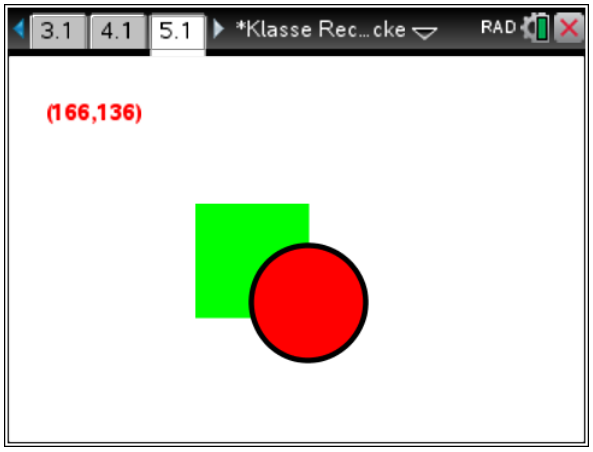
Beschreibung der Aktion	Lua-Skript

Wir halten die Klasse Kreis allgemein, d.h., dass das auch einmal Ellipsen werden könnten. Gemeinsam mit Zufallsfunktionen könnte so einmal eine Software zur Gestaltung von unterschiedlichen Fragestellungen entstehen.
 	
Beachten Sie math.sqrt. Damit rufen wir eine mathematische Funktion aus einer schon bestehenden Lua-Bibliohek auf.
 	
Die Funktion on.tabKey() wird für die Tastatur den Übergang von einem Objekt zum anderen steuern. Jetzt wird aus der Liste das entsprechende Element (hier Rechteck oder Kreis) ausgewählt.
 	
Das break in der Schleife sorgt dafür, dass der Wechsel von einem zu nächsten Element nicht unendlich oft weiterläuft.
(Für die Maus wird diese Aufgabe dann on.mouseDown übernehmen.)
 	
 
 	
Eine fiktive Variable „_“ arbeitet alle Elemente der Schleife (obj[1], obj[2], …) mit seinen spezifischen Eigenschaften (Farbe, Platzierung, ...) ab.
 	
Ein Screenshot zeigt das Quadrat aktiv.
 	
Der Start erfolgt über die TAB-Taste.
	function Kreis:contains(x, y)
	  local r = self.radius
	  local d = math.sqrt((self.x-x)^2+(self.y-y)^2)
	  return d <= r
	end
	
	function Kreis:paint(gc)
	  local cx = self.x - self.radius
	  local cy = self.y - self.radius
	  local durchm = 2*self.radius
	  gc:setColorRGB(unpack(self.farbe))
	  gc:fillArc(cx,cy,durchm,durchm,0,360)
	  if self.gew then
	  gc:setPen("medium","smooth")
	  gc:setColorRGB(0, 0, 0)
	  gc:drawArc(cx, cy, durchm,durchm,0,360)
	end;end
	
	function on.tabKey()
	  if ObjWahl ~= leer then ObjWahl.gew = false
	  end
	  for i = 1,#Objekte do
	  local obj = Objekte[i]
	  if obj == ObjWahl then ObjWahl = Objekte[i+1]
	  break
	  end; end
	  if ObjWahl == leer then ObjWahl = Objekte[1]
	  end
	  ObjWahl.gew = true
	screen:invalidate(); end
	
	function on.paint(gc)
	  for _,obj in ipairs(Objekte) do
	  obj:paint(gc);end
	  if ObjWahl ~= leer then
	  gc:setFont("sansserif", "b", 10);
	  gc:setColorRGB(255,0,0)
	  gc:drawString("("..ObjWahl.x..",
	  "..ObjWahl.y..")", 20,20)
	  end
	end

4.2.1 Quadrat, Kreis, Tastatur und Maus

Es folgen die abgeänderten Routinen für die Maus (Klicken, Loslassen und Ziehen). Im Wesentlichen werden hier die Objekte der Liste angesprochen.

Beschreibung der Aktion	Lua-Skript

Die zu tabKey analoge Mausfunktion:
 	
Die globalen Offset-Variablen verhindern einen kleinen Sprung des mit der Maus angeklickten Objekts an die Position des jeweiligen Objektmittelpunkts.
 	
table.remove und table.insert stellen jedes vorher gewählte Objekt frei und ersetzen es durch jenes, das durch Anklicken aktiv wird.
 	
 
 	
Hier schiebt sich gerade der Kreis vor das Quadrat.
	function on.mouseDown(x,y)
	for i=1,#Objekte do
	local obj = Objekte[i]
	if obj:contains(x,y) then
	if ObjWahl ~= leer then
	ObjWahl.gew = false
	end
	ObjWahl=obj; obj.gew = true
	Offsetx = ObjWahl.x – x
	Offsety = ObjWahl.y - y
	table.remove(Objekte,i)
	table.insert(Objekte,obj)
	screen:invalidate()
	break
	end; end; end
	
	function on.mouseUp(x,y)
	if ObjWahl ~= leer then
	ObjWahl.gew = false
	end
	ObjWahl = leer
	screen:invalidate(); end
	
	function on.mouseMove(x,y)
	if ObjWahl ~= leer then
	ObjWahl.x = x + Offsetx
	ObjWahl.y = y + Offsety
	screen:invalidate()
	end; end

4.2.2 Aus zwei mach viele

Nun kommt noch eine nette Draufgabe. Wir können Quadrat und Kreis mit Maus und Tastatur am Display frei bewegen und wir wollen die Objekte „clonen“. Wenn ein Objekt gewählt ist (erkenntlich an der schwarzen Umrandung) wird durch Drücken der ENTER-Taste eine Kopie dieses Objekts erzeugt – und in die Liste der Objekte aufgenommen.

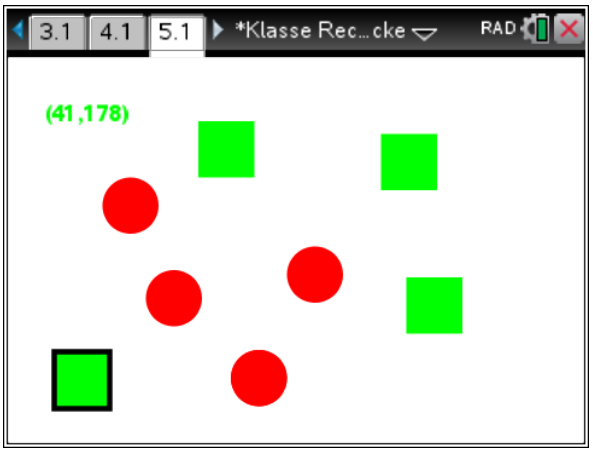
Der rot geschrieben Code ist in die on.mouseDown-Funktion aufzunehmen:

```
ObjWahl=obj;obj.gew = true
    function on.enterKey()
      if ObjWahl ~= leer then
        if ObjWahl.farbe == Farbe.gruen then
          table.insert(Objekte,
          Rechteck(ObjWahl.x, ObjWahl.y, B/10, B/10))
        else
          table.insert(Objekte,
          Kreis(ObjWahl.x, ObjWahl.y, B/20, B/20))
        end
      end
    end
    Offsetx = ObjWahl.x-x
```

Um den Bildschirm nicht zu überladen wurden die Objekte verkleinert.

Außerdem können die Koordinaten in der jeweiligen Objektfarbe angezeigt werden.

Diese kleinen Änderungen können Sie vielleicht selbst einbauen.



Die nächste Routine löscht angeklickte Elemente wieder.

```
function on.backspaceKey()
  for i = #Objekte,1,-1 do
    local obj = Objekte[i]
    if obj.gew == true then
      table.remove (Objekte,i)
  end; end
screen:invalidate(); end
```

Sie sind nun eingeladen, die „Figurierten Zahlen“ ohne Schieberegler – und daher auch ohne var.store und var.recall u.s.w. ähnlich wie auf der Abbildung auf Seite 40 zu erzeugen. Viel Glück dazu! (http://www.compasstech.com.au/TNS_Authoring/Scripting/script_tut11.zip)

5 Auf verschiedenen Plattformen

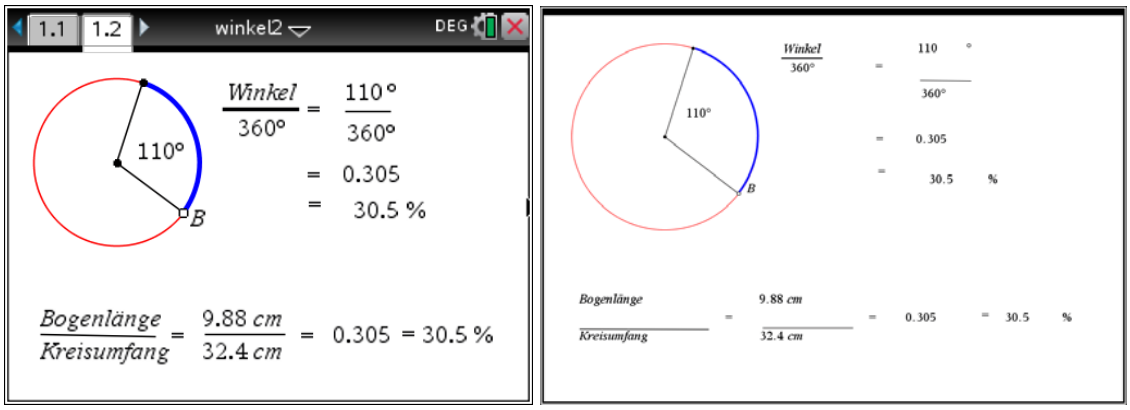
Wenn wir Dokumente erzeugen, die nicht nur für den Eigengebrauch bestimmt sind, ist es notwendig, dass sie auf allen Plattformen wir Handheld, Notebook, PC, … ohne weitere Modifikationen laufen. Eine wichtige Rolle spielt dabei die API-Version. Eine niedrige Version (z.B. apilevel 1.0) ist zu mehreren Plattformen kompatibel, hingegen verfügen höhere Versionen über bessere Funktionalitäten.

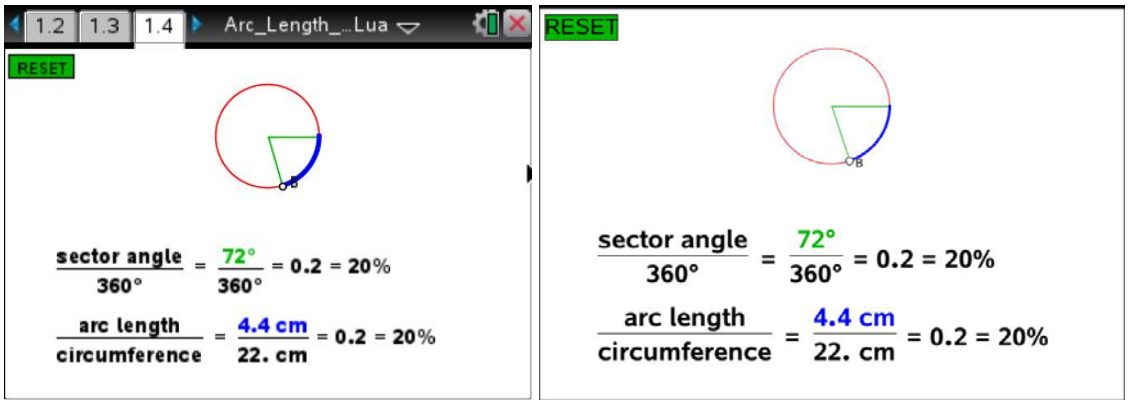
5.1 Auf die Skalierung kommt es an

Ein typisches Beispiel ist die geeignete Skalierung für alle Displays. So sollen sich z.B. unsere Programme automatisch an die Handheld- oder Computer-Ansicht von TI-Nspire anpassen.

Dazu benötigen wir einen Skalierungsfaktor, der den Abmessungen des jeweils eingesetzten Displays Rechnung trägt.

Bevor wir genauer erklären, wie das erreicht wird, soll ein eindrucksvolles Beispiel gezeigt werden (Kreisbogen.tns und Arc_Length_and Sectors.tns).





Hier zeigen wir eine TI-Nspire- Illustration von Kreisbogen und -sektor. Die ersten beiden Screenshots stammen von TI-Nspire-Original, links vom Handheld – ganz nett – und rechts vom PC – ist schon weniger nett. Die Abbildungen darunter zeigen den gleichen Inhalt, aber in einer aufbereiteten Lua-Version, für die wir uns bei John Hanna recht herzlich bedanken (http://www.johnhanna.us/Lua.htm).

Die Bewegung des Punkts am Kreisumfang lässt sich leicht über die Pfeiltasten bewerkstelligen und muss nicht mühsam durch Ergreifen und Herumführen des Punkts erfolgen. Darüber werden wir im nächsten Abschnitt mehr lernen.

Im Vergleich der beiden Abbildungen sehen Sie den Vergleich zwischen Handheld und Computerdisplay, wenn eine entsprechende Skalierung im Skript vorgenommen wird.

Laut Steve Arnold müssen wir eine goldene Regel beachten:

Alle Abmessungen und Positionen müssen relativ zu den Displaydimensionen gegeben sein. Verwende NIE absolute Werte für Abmessungen und Koordinaten, auch nicht am Anfang, denn diese werden zumeist in einer Ansicht gut wirken, in anderen aber schlecht.

Ein gutes „schlechtes“ Beispiel ist die Ansammlung von Rechtecken auf Seite 42 links oben. Am Handheld sind die grünen Rechtecke schön verteilt, während der PC-Schirm größtenteils leer bleibt.

Die Vorgangsweise, die wir hier vorstellen, wird von Alfredo Rodriguez vorgeschlagen. Er geht von einer „intended page size“ aus, die dann entsprechende skaliert wird.

Vorschlag von Steve Arnold und Alfredo Rodriguez:

Beschreibung der Aktion	Lua-Skript
Wir deklarieren die Anfangsgrößen (für das Handheld), die bei Bedarf angepasst werden. iB, iH und der Skalierungsfaktor sf (Startwert = 1) gelten global im ganzen Skript. Diese Routine wird einmal (am Start) durchlaufen. Sie enthält Verweise („handlers“) zu weiteren Funktionen.	Platform.apilevel = ‘3.2’ screen = platform.window local iB = 318; local iH = 212 -- Pixel am Handheld local sf = 1 function on.paint(gc) on.paint = onPaint; on.resize = onResize onResize(screen:width(),screen:height()) onPaint(gc) end

Beschreibung der Aktion	Lua-Skript

Diesen Zugang verdanken wir Nevil Hopley.
Die Schriftgrößen müssen innerhalb der zulässigen Werte bleiben. Siehe dazu einen Kommentar am Ende.
 	
Das ist die Skalierungsfunktion. Damit erhalten wir am PC-Schirm das gleiche Bild wie am Handheld.
 	
Mit Hilfe der lokalen Variablen werden die Objekte (Hier ein Rechteck, zwei Strecken und eine Zeichenkette) auf den Bildschirm gebracht.
	function onResize(b, h)
	  sf=math.min(b/iB, h/iH)
	  fontSize = 10*sf
	  fontSize = fontSize >= 6 and fontSize or 6
	  fontSize = fontSize <= 255 and fontSize or 255
	end
	
	function s(a)
	  return math.floor(a*sf+0.5)
	end
	
	function onPaint(gc)
	  local b,h = screen:width(), screen:height()
	  local startx, starty = s(iB/4), s(iH/4)
	  local b,h = s(iB/2), s(iH/2)
	
	  gc:setColorRGB(200, 200, 250)
	  gc:fillRect(startx, starty, b, h)
	  gc:setColorRGB(150, 150, 250)
	  gc:drawRect(startx, starty, b, h)
	  gc:drawLine(startx, starty, startx + b, starty + h)
	  gc:drawLine(startx, starty + h, startx + b, starty)
	  gc:setColorRGB(150, 30, 30)
	  gc:setFont("sansserif", "b", fontSize)
	  local str =
	  "Wir geben Technologie in Deine Hände"
	  local sw = gc:getStringWidth(str)
	  gc:drawString(str, s(iB/2) - sw/2, s(iH*0.9),
	  "bottom")
	
	end

Ein andere Möglichkeit ist, über die globalen Variablen B = platform.window.width() und H = platform.window.height() zu arbeiten. Alle Positionsgrößen müssten dann über B und H definiert werden. Ein Objekt wäre dann in der Schirmmitte mit B/2, H/2 zu platzieren. Auch die Schriftgrößen hängen davon ab. Dann wäre z.B. fontSize = W/32 etwa die Schriftgröße 10 am Handheld. Der Wert wird automatisch gerundet. Gemeinsam mit math.floor und zwei Bedingungen sorgen wir, dass die Schriftgrößen im Rahmen bleiben:

```
fontSize = math.floor(W/32 + 0.5)
fontSize = fontSize >= 6 and fontSize or 6
fontSize = fontSize <= 255 and fontSize or 255
```

Während man am Computer Größen zwischen 6 und 255 pt anwenden kann, stehen am Handheld nur die Größen 7, 9, 10, 11, 12 und 24 zur Verfügung.

5.2 Mit Maus und/oder Cursor

Für jede Plattform gibt es natürlich bevorzugte Eingabe- und Steuermöglichkeiten. Am PC erfolgt die Interaktion zumeist über die Maus. Anklicken und Ziehen ist schnell und einfach und erfordert i. A. nur wenig Erklärungen.

Am Handheld, selbst mit guter Übung im Gebrauch des Cursors (Touchpad), ist das bei weitem nicht so perfekt wie am Computer. Wenn es uns gelingt, der Tastatur am Handheld eine besserer Funktionalität zu geben, werden wir unsere Dokumente wesentlich benutzer-freundlicher machen.

Der Schlüssel zum Erfolg ist die Verdoppelung der Funktionalität von Maus und Tastatur, so dass der Benutzer ganz nach seiner persönlichen Vorliebe das Eingabemedium wählen kann.

Obwohl es ein wenig schwierig scheint, ist der beste Weg, den Umgang mit der Maus zu optimieren, alle anklickbaren Objekte über Klassen zu definieren. Gemeinsam mit dem mächtigen „contains“ können damit viele Parameter kontrolliert werden.

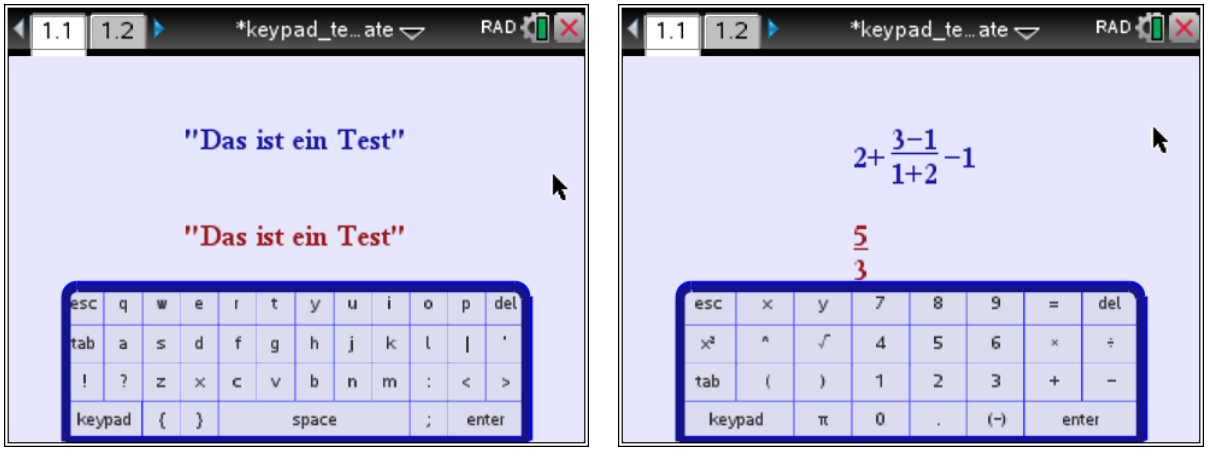
Ein Problem mit der Maus/dem Cursor ergibt sich immer wieder am Handheld: Die Maus „taucht unter“ und erscheint erst wieder nach ein paar Sekunden.Es wird empfohlen, den cursor.show()-Befehl in das Skript aufzunehmen. Dies erfolgt im Rahmen dieser Funktion:

```
function on.activate
  cursor.show()
end
```

Es empfiehlt sich, diesen Befehl z.B. in die mouseMove-Funktion aufzunehmen, um sicher zu stellen, dass der Cursor während der Interaktion mit dem Skript sichtbar bleibt.

5.3 Schaltflächen und Tastatur

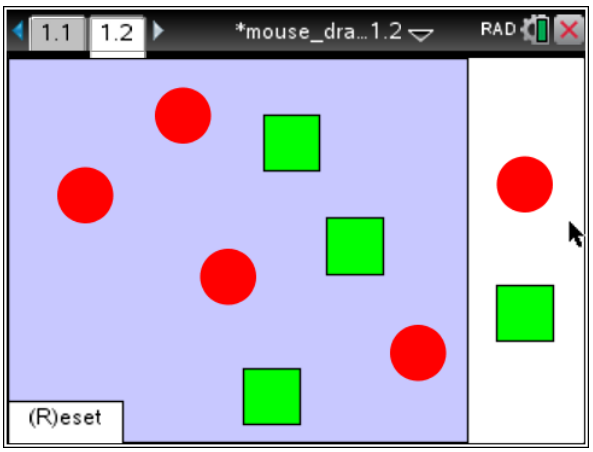
Manchmal ist die Tastatur kein geeignetes Eingabemedium. Dann müssen Schaltflächen ihre Aufgabe übernehmen. Ein mögliches Beispiel ist im unten gezeigten KeyPad zu sehen.



Die tns-Datei ist unter den Materialien zum Tutorial #20 auf der Scripting Webseite von Steve Arnold zu finden. (Die Großbuchstaben müssen hier noch zwischen Anführungszeichen über die Tastatur eingegeben werden.)

Wie schon erwähnt, wenn wir mit dem Handheld arbeiten, ist die Maus nicht immer das bequemste Werkzeug, um ein Objekt auszuwählen und zu bewegen. Da sind oft auch Tastaturmethoden viel effizienter. Die Datei mouse_dragging_v1.2.tns demonstriert ein einfaches Beispiel für dieses Konzept.

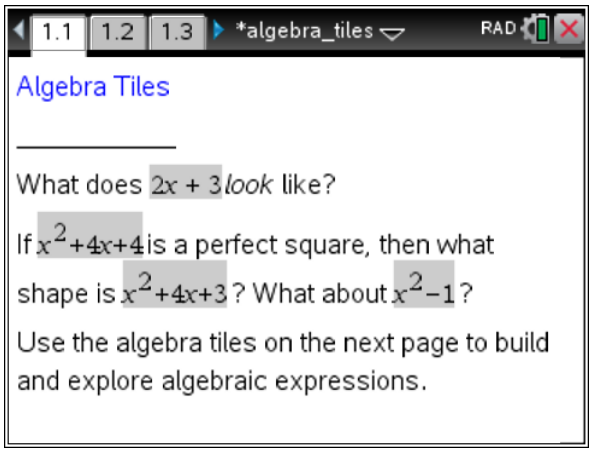
- Mit der Tabulator-Taste werden die Objekte der Reihe nach ausgewählt
- Mit den Pfeiltasten werden sie kopiert und einfach und schnell bewegt. Das kann auch mit der Maus erfolgen.
- Die ESC-Taste lässt ein aktiviertes Element wieder los.
- Über die (R)eset-Schaltfläche oder „r“/ „R“ wird der Anfangszustand wieder hergestellt.

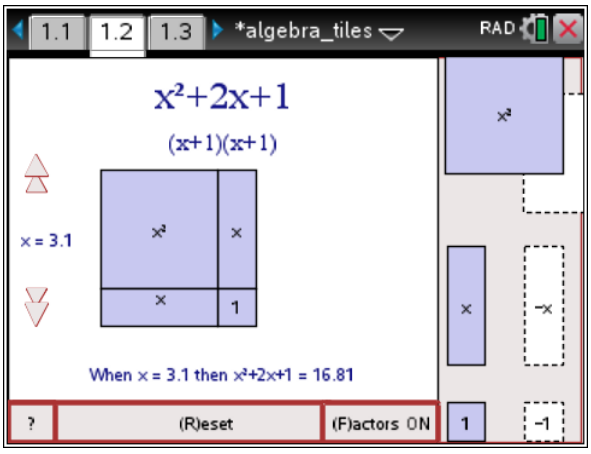


Meistens ist eine Kombination von Tastaur und Maus die optimale Lösung. *Wähle mit dem Tabulator und ziehe mit der Maus!*

Was geschieht, wenn wir mehr als zwei oder drei Objekte wählen und kontrollieren wollen (müssen)? Das Durchklicken durch mehrere verschiedenartige Elemente ist nicht vorteilhaft.

Das Dokument algebra_tiles.tns – die Algebra-Kacheln – ist ein ziemlich hoch entwickeltes Beispiel für all die Ideen, die wir bisher diskutiert haben. Man kann alle sechs verschiedenen Objekte über die Tastatur ansprechen: mit „x“ die x-Kachel, mit „s“ das Quadrat (square) und mit „u“ die Einheitskachel. Sie können dann mit den Pfeiltasten oder mit der Maus ins „Rechenfeld“ gezogen werden. Mit einem einleitenden „n“ rufen wir „negative Kacheln“ auf. Das „?“ liefert Bedienungsanweisungen und anderes mehr.





Sie sind eingeladen, das Skript, das sich auch unter den zum Tutorial #20 gehörigen Dateien befindet, zu studieren und Teile daraus zu kopieren und in ihr eigens Skript zu übertragen, bzw. sie dort Ihren Bedürfnissen anzupassen.

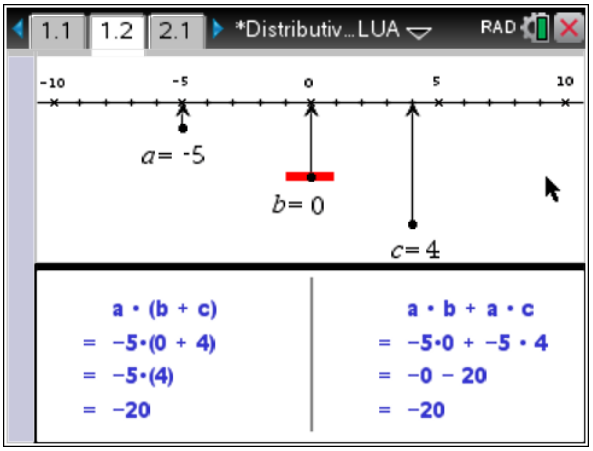
5.4 Kontrolle über die Pfeiltasten

Wir haben bereits erwähnt, dass das Ergreifen und Verziehen von Objekten am Handheld oft frustrierend sein kann. Eine der Möglichkeiten, die Steve Arnold veranlasste, sich näher mit Lua zu befassen, war die Chance, Objekte am Display mit den Pfeiltasten bewegen zu können, unabhängig davon, ob sie sich im Lua-Fenster oder „außerhalb“ in der „Nspire-Welt“ befinden. Darauf wurde ja schon hingewiesen, aber das ist auch wichtig genug, um ein eigenes Leitprinzip zu formulieren:

Dokumente sollten, wenn immer möglich, so entworfen werden, dass die Bewegung von Punkten mit den Pfeiltasten gesteuert werden können und das zusätzlich zum Fassen und Verziehen mit der Maus.

Wenn wir nur einen Punkt bewegen müssen, ist das einfach. Wenn es jedoch mehrere gibt, dann bedarf es einiger Überlegung, wie wir die Punkte auf bequeme Weise ansprechen können.

Wir können mit Klassen aber auch ohne diese arbeiten. Im nebenstehenden Beispiel (Quelle: Daniel Ilaria von Math Nspired) sind drei Punkte a, b und c zu steuern, die allerdings im Nspire-Graph-Fenster liegen. Daher sind Klassen hier keine Option.



Distributive_Property LUA.tns

Wir müssen das Lua-Fenster aktivieren, dann können wir mit der Tabulatortaste von einem Punkt zum anderen springen, wobei der jeweils aktivierte Punkt mit einem roten Strich markiert ist. Die Koordinaten werden in einer Lua-Tabelle/Liste gespeichert und der Tabulator „schreitet“ einfach durch diese Liste.

Dokumente, die so oder so ähnlich gestaltet sind, ermöglichen es dem potentiellen Benutzer, sofort in der geöffneten Seite zu arbeiten. Er/Sie muss nicht erst einen Punkt erfassen und weiter bewegen, dass etwas passiert. Dies mag einem PC-Anwender trivial erscheinen, aber für einen Handheld-Anwender ist das weit weg von trivial. Es ist immer notwendig, die Anwendungen so benutzerfreundlich wie möglich anzubieten.

Im nächsten Abschnitt verrät uns Steve Arnold noch einige Tipps und Tricks.

6 Steve's Tipps and Tricks

Es gibt noch sehr viel zu lernen. Wir empfehlen besonders Lua Nspired Wiki [1], das von vielen klugen und interessierten Personen betreut wird. (Da kann jedermann auch etwas dazu beitragen.) Ein „Muss" sollte ein Durchblättern der exellenten Tutorials von Nspired Lua [2] sein. Eine ganze Menge von netten Beispielen findet sich auch in den guten alten TI-Calc.org Lua archives [3].

[1] (https://wiki.inspired-lua.org/Main_Page)
[2] https://www.inspired-lua.org/ (betreut bis 2013)
[3] http://www.ticalc.org/pub/nspire/lua/

6.1 Suchen, Ersetzen und Zerlegen

Diese drei Einträge stammen von einem Lua-Guru, John Powers. Er war maßgeblich an der Entwicklung von Lua für TI-Nspire beteiligt.

https://inspired-lua.org/index.php/author/jppowers/

find

Wenn wir ermitteln wollen, ob ein String einen Teilstring enthält, und wo sich dieser befindet, dann können wir die Funktion **find** verwenden. Sie liefert zwei Zahlen, mit denen sie Anfang und Ende des ersten gefundenen Teilstrings anzeigt. Diese Funktion hat eine vielseitige Fähigkeit, Muster zu erkennen. So zeigt uns z.B. **str:find("%b()")** Anfang und Ende des ersten Paares von zusammengehörigen Klammern.

```
formula = "2*(3+(4+5)) "
formula:find("%b()")
```

gibt die Antwort 3, 11.

replace

Wenn wir einen Teilstring durch einen anderen String ersetzen wollen, dann geht das folgendermaßen:

```
newstring = oldstring:gsub("=","->")
```

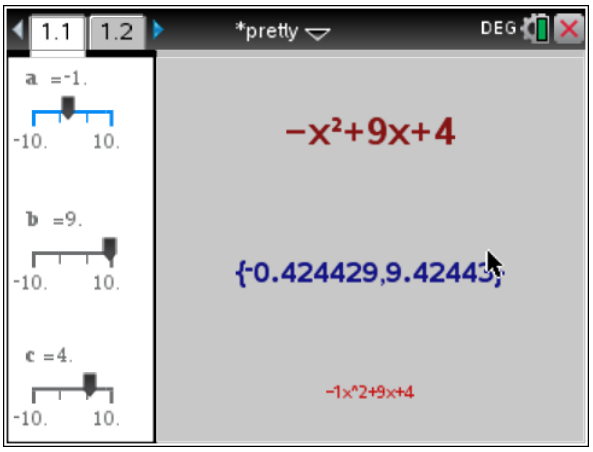
Die neue Zeichenkette ist eine Kopie der alten, bei der alle „=" durch „->" ersetzt wurden. Wenn wir nur das erste Gleichheitszeichen ersetzen müssen, dann fügen wir noch den Parameter 1 an: newstring = oldstring:gsub("=","->",1). Wird der zu ersetzende Teilstring nicht gefunden, bleibt der Originalstring unverändert. **gsub** kann auf gleiche Weise Muster erkennen, wie **find**.

Eine nette Anwendung ist z.B. das hässliche „^2" oder „3" durch das viel schönere „^{2}" und „^{3}" zu ersetzen.

Dabei ist zu beachten, dass gewisse Symbole, wie „^" und „+" Teile von Lua definitionen sind. Der Einsatz von „%" ignoriert dies und verwendet, was darauf folgt.

Die folgende Funktion konvertiert einen quadratischen Term, wie z.B. „-1x^2+-0x+-4“ in „$-x^2-4$“.

```
function pretty(input)
  input = tostring(input)
  input = input:gsub("%+%-", "-")
  input = input:gsub("0x%^2%+", "")
  input = input:gsub("0x%^2%-", "")
  input = input:gsub("0x%+", "")
  input = input:gsub("0x%-", "")
  input = input:gsub("%+0", "")
  input = input:gsub("%-0", "")
  input = input:gsub("1x", "x")
  input = input:gsub("%^2","²")
  return input
end
```



Die komplette pretty-Funktion im Verbund mit dem Skript finden Sie unter den begleitenden Materialien zum Tutorial #16, pretty.tns. Das ganze Skript umfasst fast 200 Zeilen! Lua-Programmieren ist sehr aufwändig!

Jetzt fehlt noch

split

Damit können wir einen String in zwei Teilstrings zerlegen. Nehmen wir an, dass wir eine Gleichung in ihre linke und rechte Seite zerlegen wollen. Dann würde das so aussehen:

```
teile = gleichung:split("=")
links = teile[1]
rechts = teile[2]
```

Oder in einer Skriptzeile:

```
links, rechts = unpack(gleichung:split("="))
```

6.2 Lua sucht die Power des TI-Nspire

math.eval

Ich – Steve Arnold – habe viel Spaß – aber auch nicht wenig Frust – erlebt, als ich lernte, wie man diese wunderbare und mächtige Funktion einsetzen muss/kann.

Es ist ja so, dass wir nicht wie die meisten Lua-Programmierer auf einem blanken Papier in einem Vakuum arbeiten. Wir haben ja ein mächtiges mit vollem mathematischem Rüstzeug ausgestattetes Umfeld um uns – den TI-Nspire. Anstelle, dass wir nach Algorithmen zur Bestimmung des ggT im Internet recherchieren, können wir auf die TI-Nspire gcd-Funktion zurückgreifen. Und da kommt math.eval ins Spiel.

Angenommen, wir wollen wissen, ob die beiden Zahlen **zae** und **nen** gemeinsame Faktoren haben. Wenn ja, dann wollen wir beide durch diese dividieren und den Bruch zae.. "/"..nen in gekürzter Form darstellen. Dafür benötigen wir den größten gemeinsamen Teiler.
Den TI-Nspire-Anteil schreiben wir in roter Farbe.

```
var.store("zae",zae)
varstore("nen",nen)
```

(Es ist kein Problem in Lua und TI-Nspire gleiche Variablennamen zu verwenden, sie werden sich nie … wirklich treffen.) Dann geht es weiter mit

```
local ggt = math.eval("gcd(zae,nen)")
```

Wenn wir das analysieren, finden wir, dass das Argument von math.eval (unter ") zum Nspire geschickt wird. Dieser versteht das Kommando und schickt die Antwort zurück zu Lua, ist doch nett?

Es ginge auch noch etwas besser:

```
local ggt = math.eval("gcd("..zae..","..nen..")"),
```

weil man hier keine Variable über die Lua/ti-Nspire Grenze schickt. Versuchen Sie auch das zu analysieren.

So können wir den TI-Nspire dazu bringen, alle seine Fähigkeiten im Hintergrund voll auszuspielen. Das funktioniert auch mit selbst erzeugten TI-Nspire-Funktionen UND es funktioniert auch für CAS – allerdings mit einer wichtigen Einschränkung: es lässt sich nur mit Variablentypen arbeiten, die Lua kennt: mit Zahlen und Zeichenketten, nicht mit Listen, Termen und Matrizen.

Also, angenommen, wir wollen gerne das Ergebnis einer *PolyRoots*-Funktion einer vorliegenden Funktion, wie z.B. **fn = "x^2-5"** übernehmen. Wir könnten annehmen, dass dies so erfolgen müsste:

```
ergeb = math.eval("polyroots("..fn..",x)")
```

Aber leider geht das nicht. Warum nicht?

Das Ergebnis von *PolyRoots* ist eine Liste. Wir müssen die Liste in eine Zeichenkette konvertieren, dass sie Lua empfangen kann. Mehr Erfolg hätten wir mit

```
ergeb = math.eval("string(polyroots("..fn.. ",x))")
```

Ähnlich müssen wir vorgehen, wenn wir mit Hilfe des CAS einen Term faktorisieren wollen. Das Ergebnis ist wieder ein Term – und Lua kann diesen nicht verarbeiten. Es heisst also wieder eine Zeichenkette zurück nach Lua zu schicken:

```
ergeb = math.eval("string(Factor("..fn..",x))")
```

Und wenn wir schließlich unsere Funktion wieder zurück zu TI-Nspire schicken, erfolgt dies über einen String. Wir müssen sie daher rückkonvertieren, damit das Faktorisieren durchgeführt werden kann, dann wieder zurück in einen String und zurückschicken. Das scheint kompliziert, aber bei näherer Betrachtung ergibt das auch einen Sinn. Eine Alternative wäre:

```
var.store("fn",fn)
ergeb = math.eval("string(Factor(expr(fn),x))")
```

Steve bevorzugt den eleganteren Weg und vermeidet, Variable über die “Grenze” zu schicken, außer wir benötigen sie dort wirklich für besondere Aufgaben. So, z.B., wenn wir den Graph unserer Funktion **fn** sehen wollen. Wenn wir sie im TI-Nspire speichern, können wir **f1(x) = expr(fn)** definieren und der Graph kann dargestellt werden.

Alle diese Routinen und Ideen haben in die Datei pretty.tns Eingang gefunden. Zusätzlich zur Darstellung von algebraischen Ausdrücken enthält das Skript auch eine Funktion, um Brüche darzustellen und math.eval-Prozeduren um Wurzeln von Polynomgleichungen zu finden und numerische Ausdrücke zu berechnen.

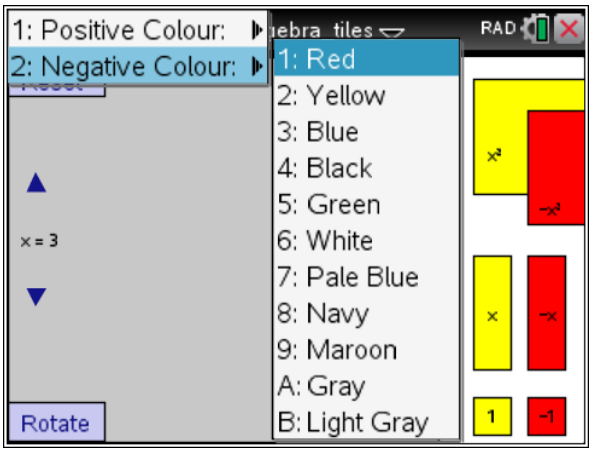
6.3 Noch mehr über Menüs

In 3.1.2 wurde bereits einiges über die Erzeugung eigener Menüs gesagt. Hier sollen noch zwei besonders schöne und lehrreiche Beispiele (von Marc Garneau und Andy Kemp) gezeigt werden.

Dieses Menü erlaubt es, dem Benutzer die positiven und negativen Flächen mit verschiedenen Farben zu versehen.

Den Anfang macht eine Funktion, die diese Wahl erlaubt. Die menu-Funktion stellt das Menü zusammen und schließlich lässt toolpalette.register alles zusammenwirken.

Wir belassen hier die Originalbezeichnungen in englischer Sprache.



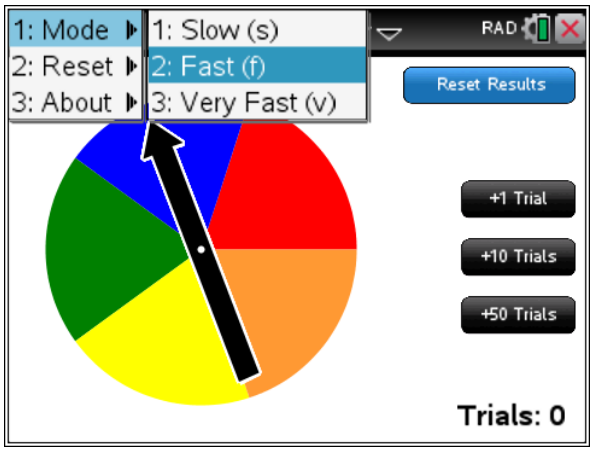
Es folgen die für das Menü entscheidenden Passagen von algebra_tiles.tns. Das komplette Skript umfasst mehr als 1600 Zeilen – allerdings mit Kommentaren.

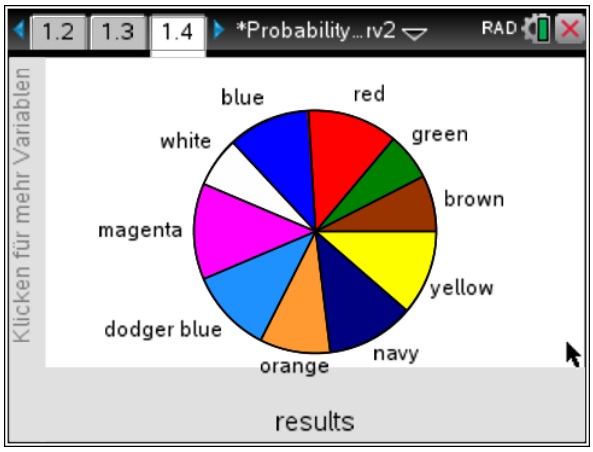
```
-- SECTION 7: [Optional] Custom menu for Tile colours

function choosecolour(kind,clr)
  if kind == "pos" then
    poscolour=clr
  else
    negcolour=clr
  end

  reset()
  platform.window:invalidate()
end
```

```
menu={
  {"Positive Colour:",
    {"Yellow", function() choosecolour("pos","yellow") end},
    {"Red", function() choosecolour("pos","red") end},
    {"Blue", function() choosecolour("pos","blue") end},
    {"Black", function() choosecolour("pos","black") end},
    {"Green", function() choosecolour("pos","green") end},
    {"White", function() choosecolour("pos","white") end},
    {"Pale Blue", function() choosecolour("pos","paleblue") end},
    {"Navy", function() choosecolour("pos","navy") end},
    {"Maroon", function() choosecolour("pos","maroon") end},
    {"Gray", function() choosecolour("pos","gray") end},
    {"Light Gray", function() choosecolour("pos","lightgray") end},
  },
  {"Negative Colour:",
    {"Red", function() choosecolour("neg","red") end},
    {"Yellow", function() choosecolour("neg","yellow") end},
    {"Blue", function() choosecolour("neg","blue") end},
    {"Black", function() choosecolour("neg","black") end},
    {"Green", function() choosecolour("neg","green") end},
    {"White", function() choosecolour("neg","white") end},
    {"Pale Blue", function() choosecolour("neg","paleblue") end},
    {"Navy", function() choosecolour("neg","navy") end},
    {"Maroon", function() choosecolour("neg","maroon") end},
    {"Gray", function() choosecolour("neg","gray") end},
    {"Light Gray", function() choosecolour("neg","lightgray") end},
  },
  }
toolpalette.register(menu)
Color = {
        red = {255, 0, 0}, orange = {255, 165, 0}
        yellow = {255, 255, 0}, green = {0, 255, 0}
        blue = {0, 0, 255}, white = {255, 255, 255}
        black = {0, 0, 0}, paleblue = {200, 200, 240}
        navy = {20, 20, 138}, maroon = {150, 50, 50}
        gray = {120, 120, 120}, lightgray = {200, 200, 200},
        }
```

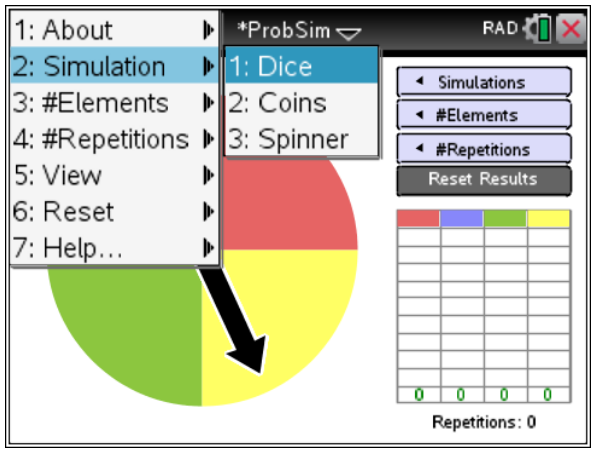
Die Farbpalette mit den passenden RGB-Werten können wir sicher auch anderswo gut einsetzen. Sie kann durchaus am Beginn des Skripts eingefügt werden.

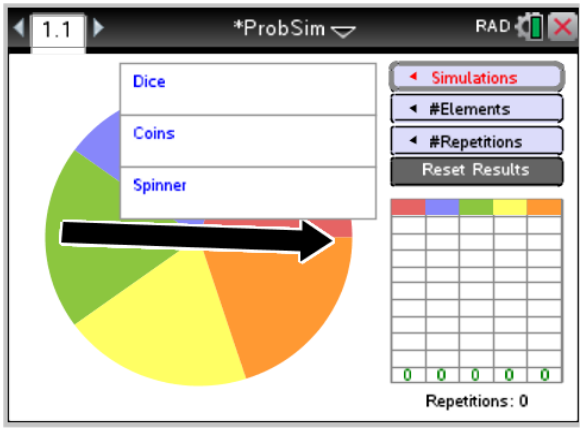
Wir zeigen drei weitere schon hoch entwickelte Beispiele für Skripts zur Simulation von Wahrscheinlichkeitsexperimenten, die von Andy Kemp entwickelt wurden. Steve Arnold hat sie auf den neuesten Stand gebracht (ProbSim.tns, und ProbDice.tns).





Hier sind zwei Versionen eines „Glücksrads“ abgebildet. In der oberen Fassung wird das Ergebnis auf einer TI-Nspire-Seite in Form eines separaten Tortendiagramms ausgegeben. In der unteren – neuen – Fassung fällt auf, dass es zwei Arten von Menüs gibt: diejenige, die wir schon kennen (links oben und wird über die Menü-Taste aufgerufen – oder über das Werkzeugsymbol am PC) und eine zweite rechts oben. Die doppelte Form der Menüs ist für den Einsatz des Programms auf einem IPad gedacht.

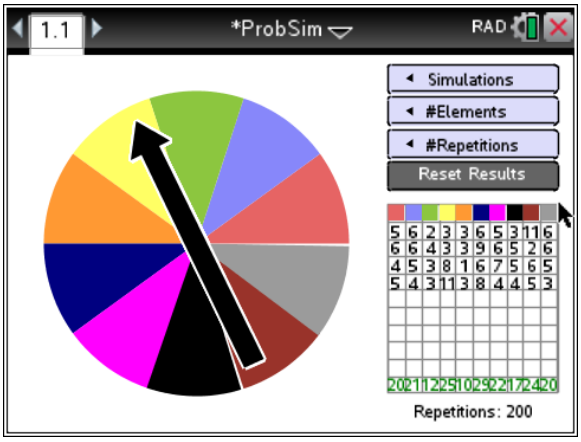


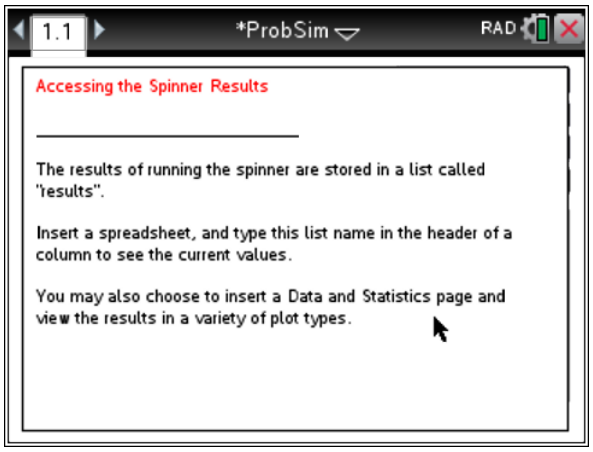


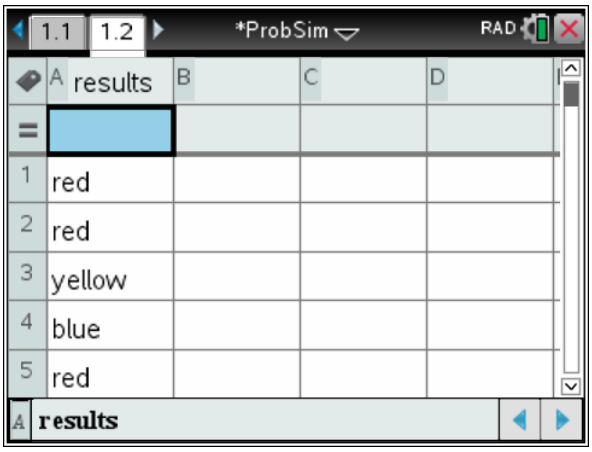
Das Ergebnis der Simulation wird hier gleich im Lua-Fenster angezeigt.

Über die „*Hilfe*“ erhalten wir Hinweise, wie wir das Simulationsergebnis selbst nach eigenen Vorstellungen darstellen können.

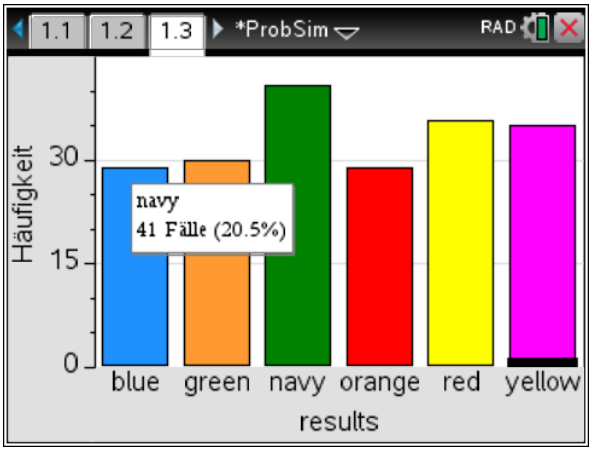
Wir können die Ergebnisse in das Spreadsheet übetragen und auch auf einer Data & Statistics-Seite in Diagrammform ausgeben wie auf der nächsten Seite gezeigt.







Ich habe 200 Umdrehungen erzeugt und das Ergebnis in Form eines Balkendiagramms dargestellt.



Wir zeigen hier noch den Teil des Skripts für die erste Fassung, der das Menü betrifft.

```
menu={
        {"Mode",
        {"Slow (s)", function() mode="slow" on.enterKey() end},
        {"Fast (f)", function() mode="fast" on.enterKey() end},
        {"Very Fast (v)", function() mode="veryfast" on.enterKey() end},
        },
  {"Reset",
  {"Reset (r)", function() resetresults() end},
  },
  {"About",
  {"About", function() about=1 platform.window:invalidate() end},
  }
}
toolpalette.register(menu)
```

Auf diesen Websites sind viele Beispiele, Tipps und Tricks zu finden:

https://sites.google.com/site/tinspiregroup/ti-nspire-script-application-files

http://www.piman.ca/Nspire/PimanNspire/Blog/Archive.html

7 Etwas Mathematik mit viel Grafik

Ich – der Übersetzer der Tutorials – möchte zum Abschluss dieses Teils noch ein eigenes Beispiel erfolgreicher Lua-Programmierung anfügen, das das Zusammenwirken von TI-Nspire CX CAS und Lua schön demonstriert.

Es geht darum, räumliche Lissajous-Gebilde als Scharen von Strecken zu präsentieren. Leider lassen sich mit TI-Nspire Geraden- und Streckenscharen nicht darstellen, da die entprechenden Grafikbefehle nicht programmierbar sind. Wenn wir diese Gebilde trotzdem auf den TI-Nspire-Schirm bringen wollen, bietet sich Lua mit seinen grafischen Fähigkeiten an.

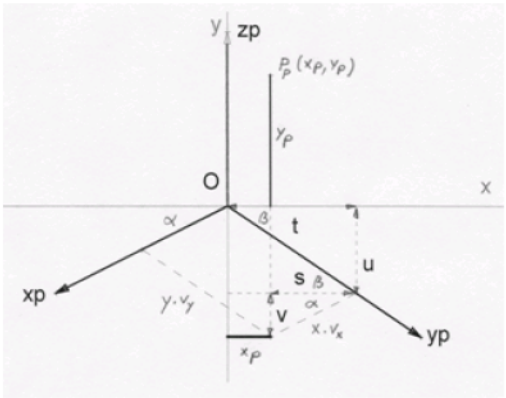
Die Mathematik dahinter findet sich in http://www.austromath.at/dug/dnl97.pdf, die TI-Nspire-Dateien sind in der zip-Datei enthalten.

```
Define spider3d(r,a,b,c,nn)=
Prgm
Local pts
x1:={}:y1:={}:x2:={}:y2:={}
n:=nn
For i,1,n
  pts:=[ r·cos(i·π/180)·cos(i·π/180)   r·sin(i·π/180)·cos(i·π/180)   r·sin(i·π/180)
         r·cos(a·i·π/180)·cos(a·i·π/180)   r·sin(b·i·π/180)·cos(b·i·π/180)   r·sin(c·i·π/180) ]·aff(vx,vy,vz,α,β)
  x1:=augment(x1,{pts[1,1]}):y1:=augment(y1,{pts[1,2]})
  x2:=augment(x2,{pts[2,1]}):y2:=augment(y2,{pts[2,2]})
EndFor
EndPrgm
```

```
Define aff(vx,vy,vz,α,β)=
Func
[ -vx·cos(α·π/180)   -vx·sin(α·π/180)
  vy·cos(β·π/180)   -vy·sin(β·π/180)
  0                  vz              ]
EndFunc
```

Hier sehen wir die Grundlagen der axonometrischen Projektion:

Axonometrische Abbildung mit den Verkürzungsverhältnissen vx, vy und vz
und den Winkeln alpha und beta:



$$x_p = t - s = y \cdot v_y \cdot \cos\beta - x \cdot v_x \cdot \cos\alpha$$

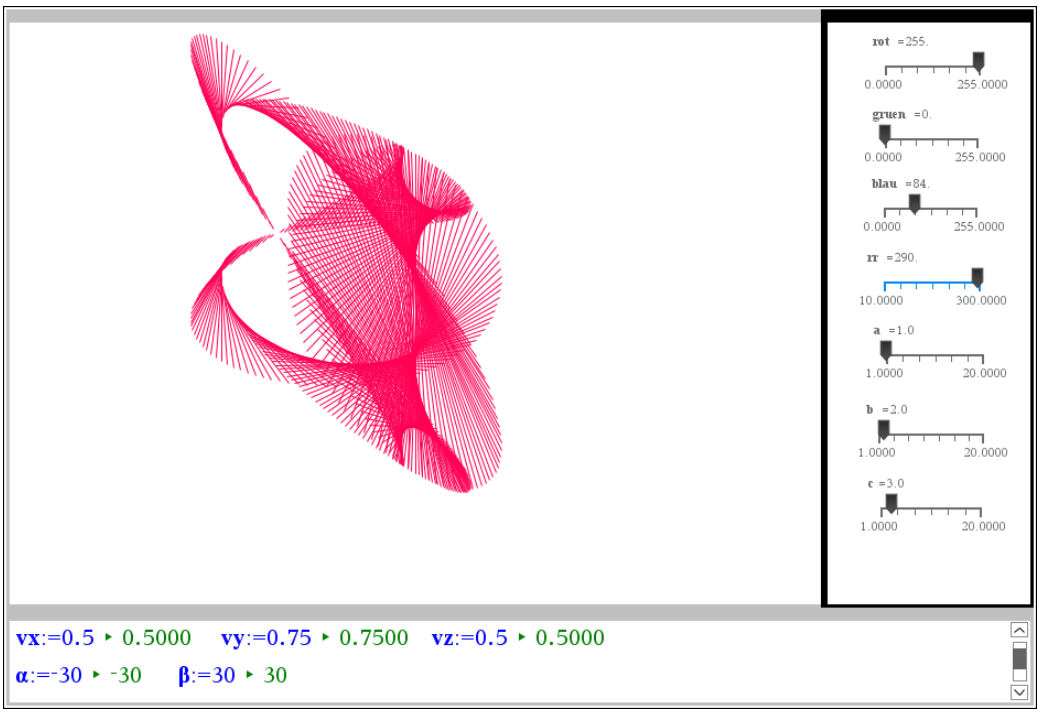
$$y_p = -u - v + z \cdot v_z = -y \cdot v_y \cdot \sin\beta - x \cdot v_x \cdot \sin\alpha + z \cdot v_z$$

oder in Matrizenschreibweise:

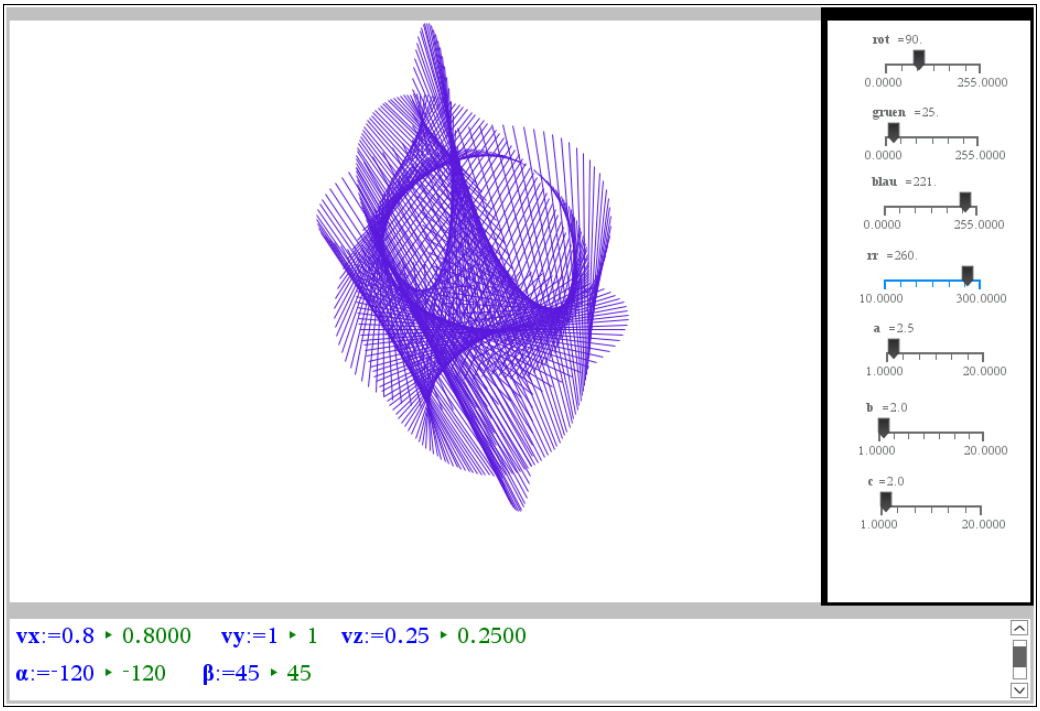
$$(x_p, y_p) = (x, y, z) \cdot \begin{pmatrix} -v_x \cos\alpha & -v_x \sin\alpha \\ v_y \cos\beta & -v_y \sin\beta \\ 0 & v_z \end{pmatrix}$$

spider3d(rr,a,b,c,360) ▸ *Fertig*

Ursprünglich hatte ich nur Schieberegler für die RGB-Werte und konnte so kleine „Farbspiele“ veranstalten. Das hat auch nicht zu 100% funktioniert, da die Reaktion erst erfolgte nachdem von den Reglern ins Lua-Fenster gewechselt ist.



Die Parameterwerte für *rr*, *a*, *b* und *c* waren über die Notes-Seite einzugeben. Jetzt bin ich über Steve‘s Tutorials schon etwas klüger geworden und kann die Schieberegler über var.monitor und on.varChange wesentlich effektiver einsetzen.



Schieberegler für die Parameter der axonometrischen Abbildung hätten das Teilfenster mit den Reglern sicherlich überladen, daher werden sie nach wie vor in den Notes festgelegt.

8 Text und Mathe in der Kiste

Seit dem TI-Nspire OS 3.2 gibt es den eingebauten Lua Script Editor, der über das Einfügen-Menü geöffnet werden kann. Diesen haben wir schon ausgiebig verwendet. Wir kehren wieder zu diesem gut arbeitenden 2D-Editor zurück und erinnern uns wieder an das Thema der ersten Schritte mit Lua: die Eingabe und Ausgabe von Text (= Zeichenketten).

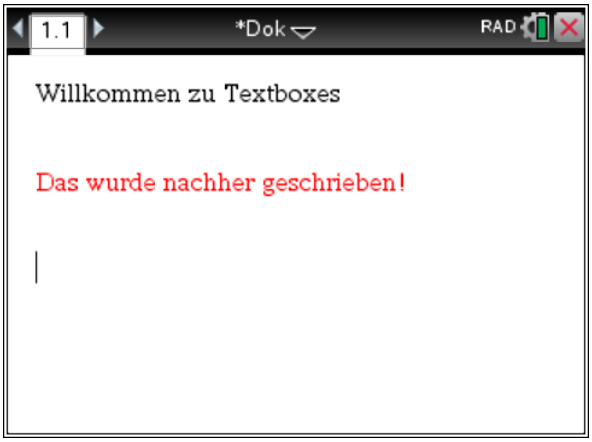
Ab nun können nicht nur Text, sondern auch mathematische und sogar chemische Ausdrücke eingegeben werden, und dazu noch ein paar geometrische Symbole.

Anstelle mit Hilfe der on.paint Funktion alles mühsam auf den Schirm „zu malen" (wie aufwändig ist es, einen Bruch darzustellen?) können wir jetzt eine "TextBox" an jeder beliebigen Stelle des Displays erzeugen, in der wir nach Belieben editieren und formatieren können, wie in einer Notes-Seite. Ja wir können auch eine "Mathbox" (oder eine "ChemBox") erzeugen.

Und, was dabei am wichtigsten ist: wir können alle Eingaben aus diesen „Kisten" entnehmen und mit ihnen weiterarbeiten.

Die TextBox ist *dynamisch*, d.h., ihr Inhalt kann in das Lua-Skript übernommen werden. So lässt sich z.B. Feedback auf die Eingabe geben: die Antworten der SchülerInnen auf gestellte Fragen können sofort überprüft bzw. korrigiert werden. Wir werden bald merken, dass diese "boxes" viele Einsatzmöglichkeiten bieten.

Beschreibung der Aktion	Lua-Skript

Den ersten Teil kennen wir schon.
 	
Mit dem Befehl D2Editor.newRichText() wird eine Textbox mit der Bezeichnung TBox definiert.
 	
Alle Kommandos innerhalb von on.resize() werden beim Öffnen der Seite aufgerufen. Die linke obere Ecke der Textbox und ihre Dimensionen werden festgelegt.
 	
Die beiden letzten Zeilen erzeugen die Schreibmarke und den Begrüßungstext.
 	
Der rote Text wurde in der Nspire-Umgebung anschließend geschrieben – ganz normal.
	platform.apilevel="3.2"
	local screen = platform.window
	local b,h = screen:width(),screen:height()
	local TBox=D2Editor.newRichText()
	function on.resize()
	  b=screen:width(); h=screen:height()
	  TBox:move(0.05*b,0.05*h)
	  TBox:resize(0.9*b,0.9*h)
	  TBox:setFocus(true)
	  TBox:setText("Willkommen zu Textboxes")
	end
	 

8.1 Texte innerhalb und außerhalb von Lua

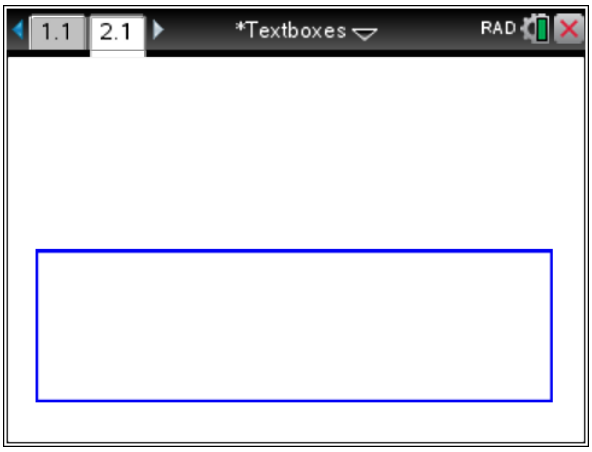
Beschreibung der Aktion	Lua-Skript

Jetzt wird es etwas aufwändiger – und nicht mehr ganz so einfach: wir verwenden on.construction() und das gute alte on.resize().	platform.apilevel='3.2'
	local screen=platform.window
	local b,h = screen:width(), screen:height()
	local TextBox1
	local fontSize
	
	function on.construction()
	  timer.start(0.4)
	  TextBox1=D2Editor.newRichText()
	end
	
	function on.resize()
	  b = screen:width(); h = screen:height()
	  TextBox1:move(b*0.05,h*0.5)
	  TextBox1:resize(b*0.9,h*0.4)
	  fontSize=math.floor(b/32)
	  fontSize = fontSize >6 and fontSize or 7
	  TextBox1:setFontSize(fontSize)
	  TextBox1:setBorderColor(500)
	  TextBox1:setBorder(1)
	  TextBox1:setTextColor(8000000)
	  TextBox1:setFocus(true)
	end
	
	function on.getFocus()
	  TextBox1:setFocus()
	end
	
	function on.timer()
	  cursor.show()
	  Eingabe = TextBox1:getExpression()
	  if Eingabe then
	  var.store("eingabe",Eingabe)
	  end
	  screen:invalidate()
	end
	
	function on.escapeKey()
	  TextBox1:setExpression(" ")
	  screen:invalidate()
	end

Positionierung der Textbox	
Farbparameter von 0 bis 16777215 (= hexFFFFFF)	
Rahmendicke von 1 bis 10	
Damit wird die Textbox aktiviert und wir können sofort etwas eintragen.	
Der Cursor wird angezeigt und die Texteingabe wird in der Variablen eingabe gespeichert.	
Über die Esc-Taste wird die Textbox wieder geleert – und die Variable mit dem Leerstring belegt.	

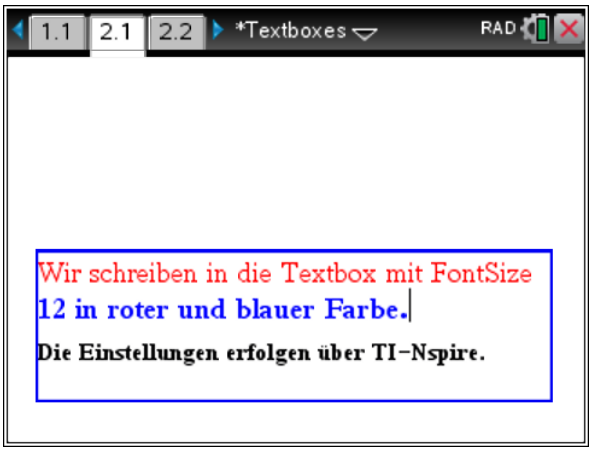
Wenn wir die Datei öffnen bietet sich vorerst nur das Lua-Fenster mit der Textbox an. Wir können beliebige Texte eintragen und auch nach Belieben formatieren, wie oben.

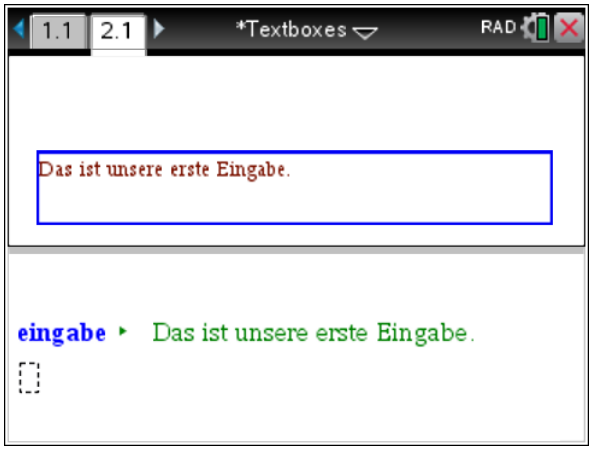
Die beiden Schirme sind rechts abgebildet.

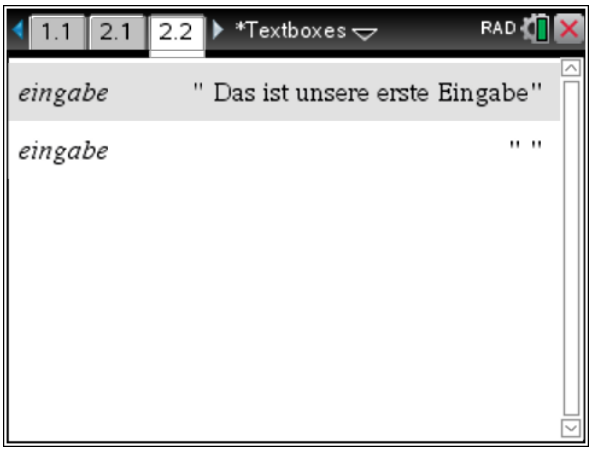


Jetzt teilen wir den Schirm und installieren in der unteren Hälfte eine Notes-Seite.

Wenn wir eine Math Box einfügen und die Variable *eingabe* aufrufen, erscheint sofort die Zeichenkette von oben, die sich jeder Veränderung in der Textbox sofort anpasst.







Die ESC-Taste leert die Textbox und auch den Variableninhalt, wie schon oben erklärt.

Der rechte Screenshot zeigt den Calculator vor und nach Gebrauch der ESC-Taste in der Textbox.

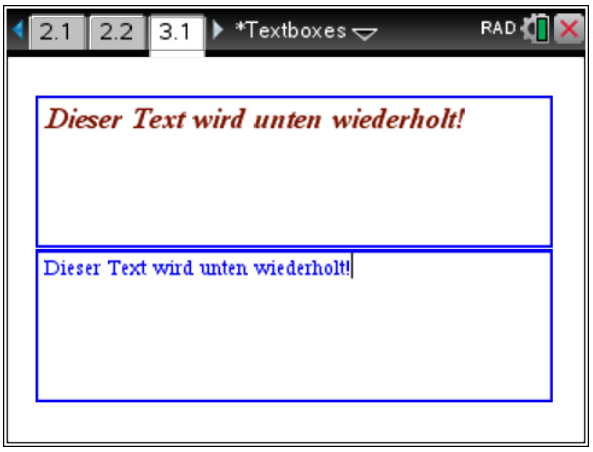
(Die Textbox ist nun leer und die Variable *eingabe* hat den Wert “ “.)

Ergänzend sei bemerkt, dass die *eingabe* natürlich auch im Calculator aufgerufen werden kann.

8.2 Textbox im Doppelpack

Versuchen Sie es zuerst alleine: Erzeugen Sie eine zweite Textbox auf dem Display und arrangieren sie diese so, wie unten abgebildet.

```
platform.apilevel = "3.2"
local screen=platform.window
local b,h = screen:width(), screen:height()
local TextBox1, TextBox2
local fontSize

function on.construction()
   timer.start(0.4)
   TextBox1 = D2Editor.newRichText()
TextBox2 = D2Editor.newRichText(); end
function on.resize()
  b = screen:width(); h = screen:height()
   fontSize = math.floor(b/32)
   fontSize = fontSize > 6 and fontSize  or 7
   TextBox1:move(b*0.05, h*0.1)
   TextBox1:resize(b*0.9,h*0.4)
   TextBox1:setFontSize(fontSize)
   TextBox1:setBorderColor(500)
   TextBox1:setBorder(1)
   TextBox1:setTextColor(8000000)
   TextBox1:setFocus(true)
   TextBox2:move(b*0.05, h*0.5)
   TextBox2:resize(b*0.9,h*0.4)
   TextBox2:setFontSize(fontSize)
   TextBox2:setBorderColor(500)
   TextBox2:setBorder(1)
TextBox2:setTextColor(500); end


function on.timer()
   cursor.show()
   Eingabe = TextBox1:getExpression()
   if Eingabe then
      var.store("eingabe", Eingabe)
      TextBox2:setExpression(Eingabe); end
screen:invalidate(); end
function on.escapeKey()
  TextBox1:setExpression(" ")
  TextBox2:setExpression(" ")
screen:invalidate(); end
```



Textbox1 ist oben, Textbox2 ist darunter.

Im Skript links sind alle notwendigen Änderungen gegenüber dem vorigen Skript in Rot geschrieben.

Der Schluss bleibt gleich.

Damit steht einmal die zweite Textbox, in der sich allerdings noch nichts tut.

Wir wollen weiter so ergänzen, dass der in die obere Box geschriebene Text – nur dort kann geschrieben werden – in der unteren wiederholt wird.

In der Eingabebox kann beliebig in der TI-Nspire-Umgebung formatiert werden, in der Ausgabebox nicht.

Spielen Sie ein wenig mit den Boxparametern (Strichstärke, Farben) herum und beobachten Sie die Änderungen.

Nur zwei kleine Ergänzungen sind dazu notwendig, die links gezeigt sind (in dern on.timer() und on.escapeKey()) Funktionen.

Hinweis: Die Kommandos getText und setText können anstelle von getExpression und setExpression verwendet werden.

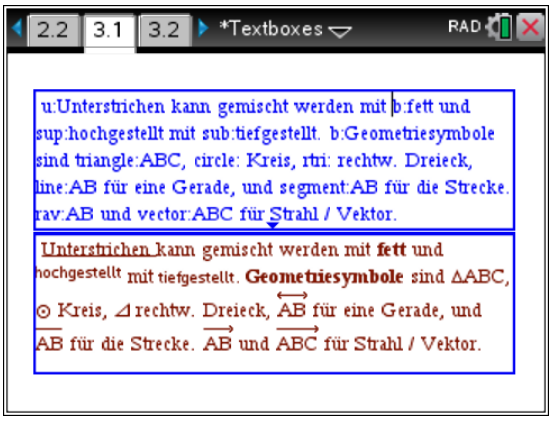
In Abschnitt 6.1. haben wir mit pretty(input) bereits einige mathematische Sonderzeichen darstellen können. pretty (input) vermag noch einiges mehr.

```
function pretty(eingabe)
  eingabe = eingabe:gsub("circle:", "\\1circle ")
  eingabe = eingabe:gsub("triangle:", "\\1tri ")
  eingabe = eingabe:gsub("angle:", "\\1angle ")
  eingabe = eingabe:gsub("ray:", "\\1ray ")
  eingabe = eingabe:gsub("line:", '\\1line ')
  eingabe = eingabe:gsub("segment:", "\\1lineseg ")
  eingabe = eingabe:gsub("rtri:", "\\1rtri ")
  eingabe = eingabe:gsub("vector:", "\\1vector ")
  eingabe = eingabe:gsub("sup:", "\\1supersc ")
  eingabe = eingabe:gsub("sub:", "\\1subscrp ")
  eingabe = eingabe:gsub("b:", "\\1keyword ")
  eingabe = eingabe:gsub("u:", "\\1title ")
  eingabe = eingabe:gsub("i:", "\\1subhead ")
  return eingabe
end
```

TextBox2:setExpression(Eingabe); end

Wird geändert zu:

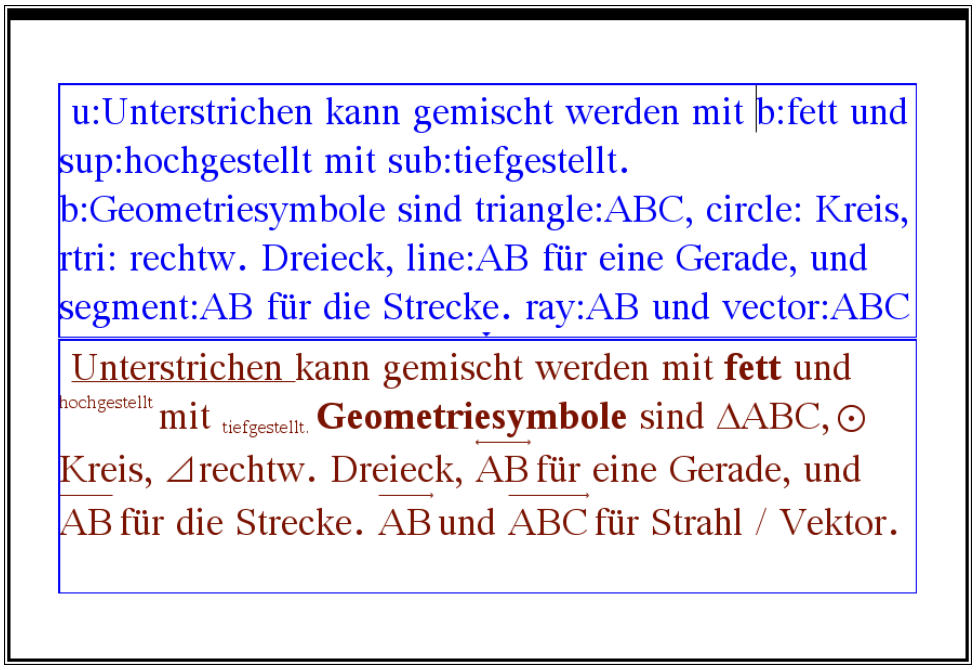
TextBox2:setExpression(pretty(Eingabe))
end



In der Computer-Ansicht kann man die Eingabe (oben) und Ausgabe (unten) noch besser sehen:

Die Funktion pretty ist einfach in das vorige Skript einzufügen und in der Textbox2 im Rahmen der on.timer-Funktion zu berücksichtigen.

Textboxes 1 und 2 in der Computer-Ansicht:



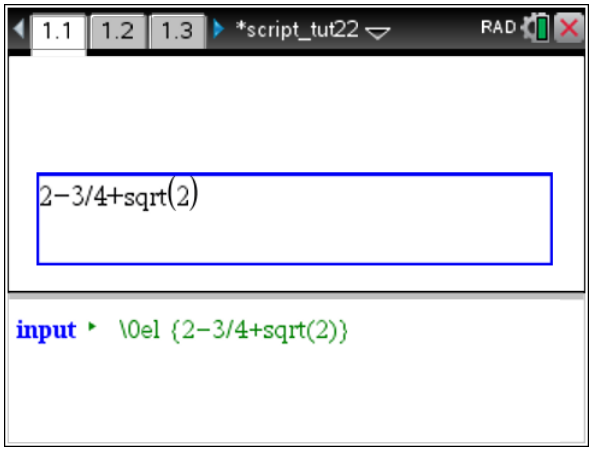
8.3 Mathematik mit Chemie

Den Anfang des Skripts wiederholen wir hier nicht. Die Unterschiede zur „gewöhnlichen“ Textbox schreiben wir wieder in Rot. Es sind nur ein paar Ergänzungen in der on.resize- und on.timer-Funktion zu machen, sowie eine neue Funktion uwrap hinzuzufügen.

Beschreibung der Aktion	Lua-Skript

Textbox1 wird als Mathbox mit der Länge des Zeilenumbruchs (in Pixel) definiert.
 	
Der ReadOnly modus wird ausgeschaltet.
 	
Analoges für die Textbox2:
 	
Schaltet das 2D-Layout einer mathematischen Eingabe aus und zeigt das Objekt des Texteditors.
 	
Wir holen die Eingabe aus Textbox1 und werten diese aus. math.eval hatten wir schon in 6.2. Aber hier ist die Ausgabe wahrscheinlich kein gewöhnlicher String, sondern vielleicht ein Bruch oder ein anderer algebraischer Ausdruck. Daher verwenden wir math.evalStr.
 	
Aber halt, die Eingabe ist ja in einem speziellen Code „verpackt“, den wir erkennen, wenn wir nur eine Mathbox erzeugen und ihren Inhalt in den Notes ausgeben.
	function on.resize()
	b=screen:width(); h=screen:height()
	TextBox1:move(b*0.05, h*0.1)
	TextBox1:resize(b*0.9,h*0.4)
	TextBox1:setFontSize(b/24)
	TextBox1:createMathBox()
	TextBox1:setWordWrapWidth(0.875*b)
	TextBox1:setBorderColor(500)
	TextBox1:setBorder(1)
	TextBox1:setReadOnly(false)
	TextBox1:setTextColor(8000000)
	
	TextBox2:move(b*0.05, h*0.5)
	TextBox2:resize(b*0.9,h*0.4)
	TextBox2:createMathBox()
	TextBox2:setWordWrapWidth(0.875*b)
	TextBox2:setFontSize(b/24)
	TextBox2:setBorderColor(500)
	TextBox2:setBorder(1)
	TextBox2:setReadOnly(false)
	TextBox2:setTextColor(8000000)
	
	TextBox1:setDisable2DinRT(false)
	TextBox1:setFocus(true)
	end
	function on.timer()
	cursor.show()
	Eingabe = TextBox1:getText()
	if Eingabe then
	Ausgabe = math.evalStr(unpretty(Eingabe))
	var.store("eingabe", unpretty(Eingabe))
	if Ausgabe then
	var.store("ausgabe", Ausgabe)
	TextBox2:setExpression("\\0el {"..Ausgabe.."}")
	end; end
	screen:invalidate()
	end

Das würde dann so aussehen:

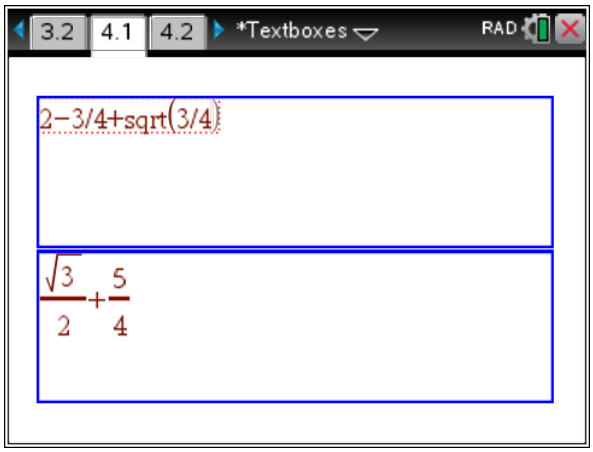


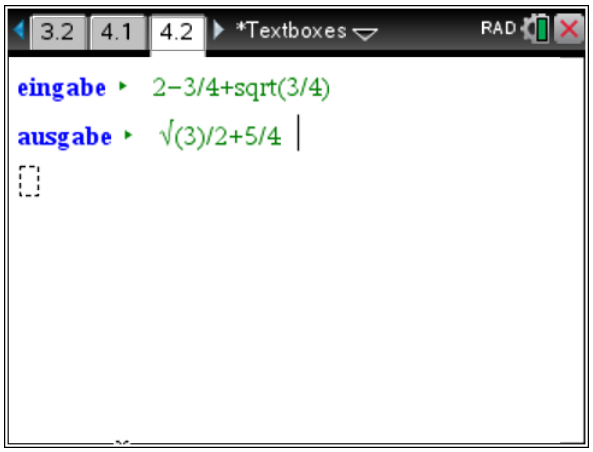

```
function unwrap(eingabe)
   eingabe = eingabe:gsub("\\0el {", "")
   eingabe = eingabe:gsub("}", "")
   eingabe = eingabe:gsub(" ", "") or ""
   return eingabe
end
```

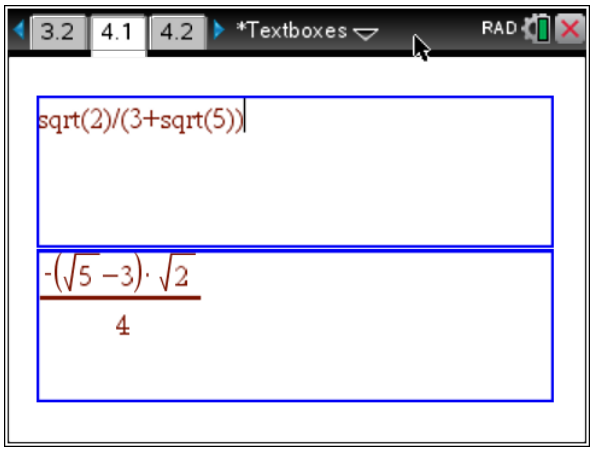
Man sieht hier nur einen Backslash. Dieser ist ein reserviertes Zeichen in Lua, daher müssen wir, dass es erkannt wird, diese Zeichen doppelt in der unwrap-Funktion angeben.

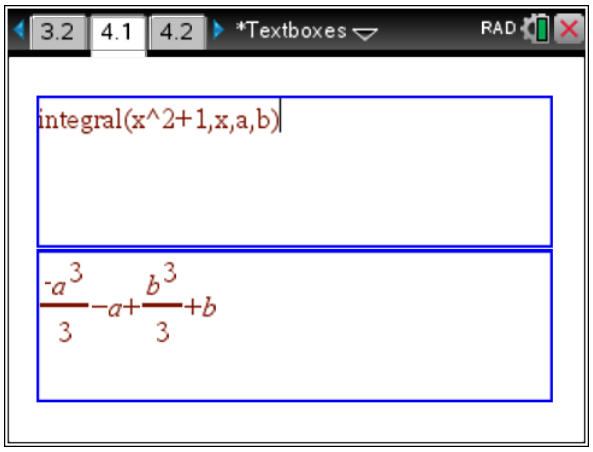
Die kleine Funktion unwrap entfernt diese Verpackung "\\0el {"Ausgabe"}" und macht den Inhalt für TI-Nspire lesbar. Für die schöne mathematische Ausgabe in Textbox2 müssen wir die „Verpackung“ für Lua wieder anbringen, ist doch findig?

Es folgen vier Illustrationen zur Anwendung der Mathbox.



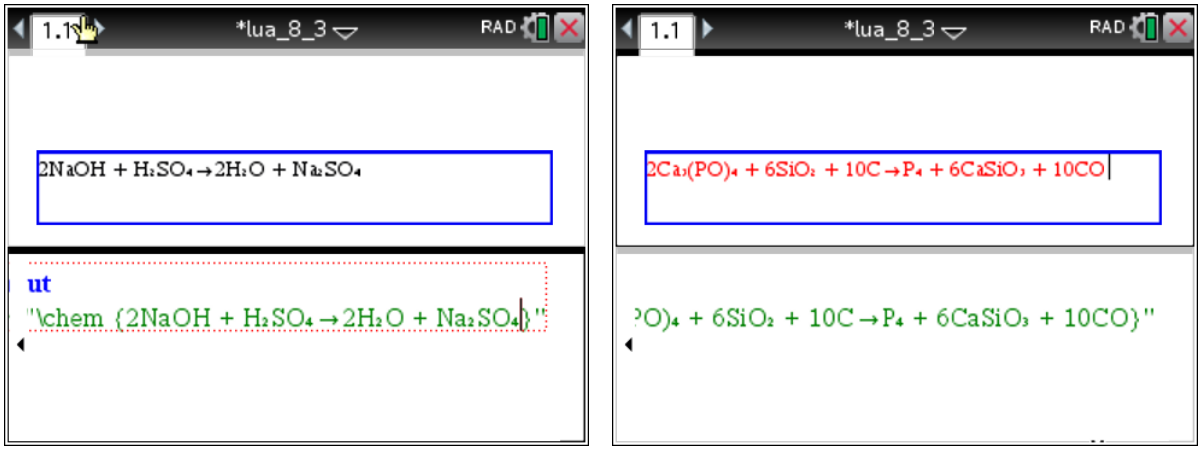






Das Arbeiten mit Mathboxes ist sicherlich wesentlich bequemer als das mühsame „Basteln“ von ordentlichen Ausgaben für Brüche und verschiedene Terme. So wünschen wir viel Spaß damit.

Wir haben in der Überschrift auch von Chemie geschrieben. Anstelle der MathBox lässt sich auch eine ChemBox einrichten.



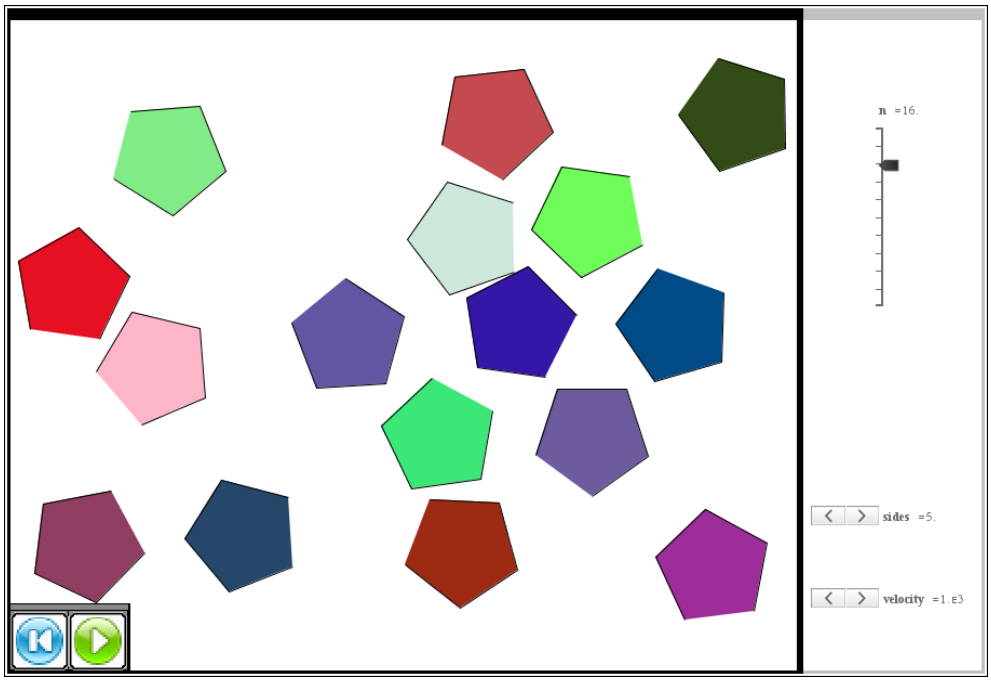
Das Kommando ist einfach, nämlich: TextBox1:createChemBox(). Damit wird die Textbox zur „Chemiekiste“. Darin lassen sich chemische Formeln schön darstellen.

Auch hier ist die Eingabe „verpackt“, in "\chem{meine Formel}".

Da die zweite Formel zu lang ist, muss man die Schriftgröße im Skript auf b/36 reduzieren.

9 Die Lua Physikmaschine

Die wahrscheinlich wichtigste Erweiterung der Lua-Möglichkeiten gemeinsam mit der TI-Nspire-Version 3.2 liegt in der neuen „*Lua Physics Engine*“. Sie basiert auf der Open Source „*2D Chipmunk Engine*“. Mehr darüber findet man auf http://chipmunk-physics.net/. Diese neue Bibliothek stellt ungeheuer mächtige Simulationsmöglichkeiten zur Verfügung. Mit diesen Werkzeugen lassen sich nicht nur Bewegungen von Objekten unter der Einwirkung von verschiedenen Kräften modellieren, sondern sie können für Aufgaben der Mechanik, der allgemeinen Physik und vieles mehr verwendet werden.



9.1 Wir starten die *Physics Engine*

Unser erstes Chipmunkprojekt wird natürlich deutlich weniger ambitioniert sein als das oben abgebildete quirlige Display. Wir werden beginnen, indem wir einen Ball auf dem Schirm von einer Seite auf eine andere prallen lassen.

Sie werden – völlig mit Recht – sagen, dass wir dazu nicht unbedingt die *Physics Engine* brauchen. Die Zauberei mit dem Timer kann Objekte ohne viel Aufwand in Bewegung bringen und wir müssen nur eine „Klasse Kreis“ schaffen, deren Position über den Timer kontrolliert wird.

So, was kann uns diese Physikmaschine bieten, dass es die ganze Aufregung wert ist?

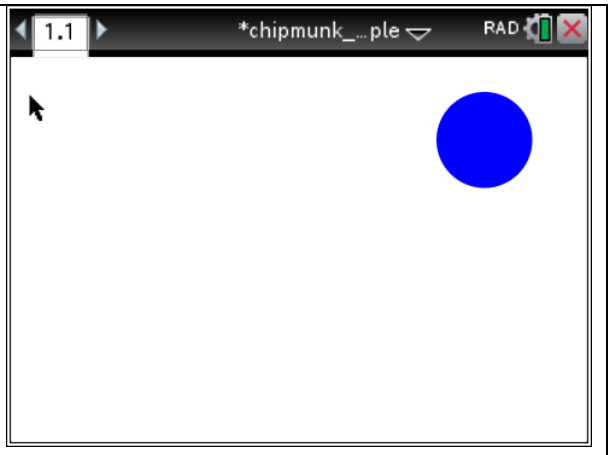
Erinnern wir uns zurück an unsere erste Begegnung mit den *Klassen*. Eine ihrer mächtigen Eigenschaften ist, dass diese Objekte jeweils „wissen“, wo sie am Display auftreten. Das öffnet uns viele Türen und macht Programme, die sonst nur sehr kompliziert wären, um sehr vieles einfacher.

Zurück zu Chipmunk. Auch hier definieren wir – wie die Klassen – Objekte, die innerhalb eines Raums (= *space*) existieren, wobei vor allem die Eigenschaften, die diese Objekte besitzen von Wichtigkeit sind. Sie besitzen einen **Körper** (= *body*) und eine **Form** (= *shape*). Bei-

des kann mit Eigenschaften versehen werden, von einfachen Dingen wie Masse, Schwerkraft und Geschwindigkeit bis hin zu interessanteren Dingen wie Trägheitsmoment, Reibung, Elastizität, Drehfedern, Gelenksverbindungen und einfachen Motoren.

Wir wollen hier wirklich nicht bis in diese weit entfernten und „exotischen“ Anwendungen vordringen – aber wir wollen so viel kennen lernen, dass ein Anfang gemacht wird. Wie weit Sie dann weiter gehen wollen, das liegt dann ganz bei Ihnen.

9.2 Ein „Ding“ tanzt am Display

Nun folgt ein erstes schnelles – nicht ganz sauberes – Programm, das einen Kreis (Ball) oder ein Quadrat am Display von Rand zu Rand reflektieren lässt. Der API-Level muss auf 2.0 gesetzt werden.	

Wichtig ist der **require**-Befehl, der den Zugriff auf eine Reihe von Bibliotheken öffnet: `require "physics"` und `require "color"` sind für uns interessant.

Beschreibung der Aktion	Lua-Skript

Wir definieren den *Raum*, in dem unser Körper existieren soll. Später werden wir weitere Eigenschaften wie z.B die dort herrschende *Schwerkraft* definieren. Vorerst ist diese 0.
 	
`physics.Body(100,0)` verlangt zwei Parameter, Masse und Trägheit, die hier mit 100 und 0 festgelegt werden.
 	
Der Körper (body) ist kein sichtbares Objekt. Wir können uns eine leere Hülle vorstellen, die mit Eigenschaften wie z.B. einer Form (shape), Elastizität oder Reibung gefüllt werden kann.
 	
Richtung und Geschwindigkeit des Körpers werden über einen Vektor definiert – hier durch den Vektor [(0,0), (500,500)].
	`-- ball1.tns`
	`platform.apilevel="2.0"`
	`require "physics"`
	`timer.start(0.01)`
	`local screen=platform.window`
	`local b,h = screen:width(), screen:height()`
	
	`function on.resize()`
	`  raum=physics.Space()`
	`  koerper_neu=physics.Body(100,0)`
	`  koerper_neu:setVel(physics.Vect(500,500))`
	`  raum:addBody(koerper_neu)`
	`end`

Beschreibung der Aktion	Lua-Skript

Position und Geschwindigkeit können jederzeit verändert werden. Und dies geschieht nun im Rahmen der on.paint-Funktion:
 	
Das meiste, das wir in diesem Funktionsblock finden, sollte uns von früher schon bekannt sein:
Wir vergeben Bezeichnungen für die Eigenschaften des Körpers. Nicht nur für seine Lage und Geschwindigkeit, sondern für die Koordinaten dieser Größen in Vektorform.
Eine vorher nicht nutzbare Schlüsselgröße, die nun von der *Physics Engine* unterstützt wird ist die Geschwindigkeit. Früher hatten wir nur den timer zur Verfügung, der nicht so flexibel Geschwindigkeit und Position kontrollieren kann.
Wir beschränken die Bewegung unseres Körpers indem wir ihn an den Rändern des Displays zurückprallen lassen (wie eine Billardkugel). setPos und setVel verwenden die jeweils angepassten neuen Koordinaten.
Schließlich verleihen wir dem noch immer nur virtuellen Körper eine Gestalt. In diesem ersten Fall soll dies ein blauer Kreis sein. Wir könnten aber jede andere Form oder sogar ein Bild wählen, solange wir nur die Lage mit posX und posY festsetzen.
Wir definieren den timer mit raum:step und erneuern das Display. Wegen eines bugs? im timer.start-Kommando setzen wir dieses nicht innerhalb der on.resize-Funktion sondern gleich an den Anfang des Skripts, so dass es nur einmal aufgerufen wird.
	function on.paint(gc)
	  local breite = b/6
	  local pos = koerper_neu:pos()
	  local vel = koerper_neu:vel()
	  local velX, velY = vel:x(),vel:y()
	  local posX, posY = pos:x(),pos:y()
	
	  if posX > b then
	  velX = -1*math.abs(velX)
	  posX = b
	  elseif posX < 0 then
	  velX = math.abs(velX)
	  posX = 0
	  end
	
	  if posY > h then
	  velY = -1*math.abs(velY)
	  posY = h
	  elseif posY < 0 then
	  velY = math.abs(velY)
	  posY = 0
	  end
	
	  koerper_neu:setPos(physics.Vect(posX,posY))
	  koerper_neu:setVel(physics.Vect(velX,velY))
	  gc:setColorRGB(0,0,255)
	  gc:fillArc(posX,posY,breite,breite,0,360)
	end
	
	function on.timer()
	  raum:step(0.01)
	  screen:invalidate()
	end

Jetzt wollen wir - wie üblich - ein wenig mit diesem Skript experimentieren:

Was passiert, wenn wir den Geschwindigkeitsvektor verändern? Welche Auswirkung hätte (1000,1000) oder (1000,0) oder (-100,-100) anstelle von (500,500)? Man sollte zuerst überlegen, was passieren wird und dann seine Überlegung überprüfen.

Wir könnten auch eine andere Startposition annehmen. Dazu setzen wir unter die setVel-Zeile etwas wie z.B. koerper_neu:setPos(physics.Vect(b/2,h/2).

Es fällt uns auf, dass unser Ball ganz richtig vom linken und vom oberen Rand zurückgeworfen wird. Er läuft aber über die beiden anderen Ränder hinaus. Wie könnten wir dies verbessern? Dies wären die notwendigen Anpassungen:

```
if posX > b-radius then
    velX = -1*math.abs(velX)
    posX = b-radius


if posY > h-radius then
    velY = -1*math.abs(velY)
    posY = h-radius
```

Radius und Farbe der Kreisscheibe sind so leicht zu ändern, dass wir das hier gar nicht herzeigen.

Wir wollen den Ball noch durch ein Quadrat ersetzen:

```
gc:fillPolygon({posX,posY,posX+radius,posY,posX+radius,posY+radius,posX,posY+radius})
```

9.3 Noch ein Tanz – jetzt mit Struktur

Das Skript von 9.2 wird nun nochmals geschrieben, aber mit einer besseren und effektiven Struktur. Das kann für weitere Aufgaben sehr nützlich sein

Beschreibung der Aktion	Lua-Skript

Wir definieren alle Variablen im Voraus als lokale Variable.
Lokale Variable sind immer wirkungsvoller als globale. Sie können dann in allen folgenden Funktionen verwendet werden.
 	
require "physics" wurde bereits vorgestellt. Hier verwenden wir auch die zweite Bibliothek mit require "color".
Damit entfällt das umständliche Hantieren mit den RGB-Zahlen, es reicht z.B. color.orange, color.yellow …
Unsere neue Struktur beginnt mit einer selbst definierten Funktion **init(gc)**. In ihr werden alle Startbedingungen festgelegt und die Variablen definiert.
	--Ball2
	platform.apilevel = '2.0'
	require "physics"
	require "color"
	timer.start(0.01)
	local b,h
	local raum
	local koerper_neu, form_neu
	local pos, posX, posY
	local vel, velX,velY
	local radius
	local schwerkr, masse, traegh, elast, reibung
	
	function init(gc)
	  b = platform.window:width()
	  h = platform.window:height()
	  raum = physics.Space()

Beschreibung der Aktion	Lua-Skript

Die schon übliche on.paint-Funktion wird durch unsere eigene definierte paint-Funktion (nächste Seite) ersetzt. Damit wird das tatsächliche Layout des Bildschirms inklusive Ball berücksichtigt. Zum Schluss definieren wir die timer-Funktion.
 	
Nun ist alles definiert – aber es wird noch keine Tätigkeit aufgerufen. Die allerletzte Funktion ist daher ein resize, das über on.paint auf init verweist, welches – wie erforderlich - als erstes läuft und somit alle Definitionen und Anfangsbedingungen herstellt.
 	
Alle on.paint-Aufrufe werden an unsere paint-Funktion geschickt. Das dient auch als ein mögliches Reset: jedes Mal, wenn resize aktiviert wird, verweist on.paint wieder auf init (nur einmal) und alles beginnt von Vorne.
 	
So viel dazu. Das heißt also, dass die init-Funktion nur einmal läuft und dass nachher alles zur paint-Funktion geht, wo der bewegte Ball gesteuert und kontrolliert wird.
 	
Darüber sollte man etwas nachdenken. Steve Arnold ist es ebenso ergangen.
 	
Wir definieren den **raum**, in dem sich der Körper bewegen soll und darin seine Erdanziehung (**schwerkr** = gravity = 9,8), die die Gegenstände zu Boden zieht.
 	
Experimentiere auch mit anderen Werten, wie z.B. 0, 100, -9.8, …
 	
Eine weitere Option betrifft die Trägheit, oder besser das Trägheitsmoment eines Kreises (oder jedes anderen Körpers). Die Argumente für diese Eigenschaft sind die Masse, der innere und äußere Radius des Körpers und die Entfernung der Form von ihrem Mittelpunkt (= SetOff).
	masse = 100;radius = b/10
	schwerkr = 9.8;elast = 1;reibung = 1
	raum:setGravity(physics.Vect(0,schwerkr))
	traegh = physics.misc.momentForCircle
	(masse,0,radius,physics.Vect(0,0))
	koerper_neu=physics.Body(masse,traegh)
	koerper_neu:setVel(physics.Vect(500,500))
	koerper_neu:setPos(physics.Vect(0,0))
	form_neu=physics.CircleShape(koerper_neu,
	radius,physics.Vect(0,radius/2))
	form_neu:setRestitution(elast) --*optional*
	form_neu:setFriction(reibung)
	raum:addBody(koerper_neu)
	raum:addShape(form_neu)
	on.paint=paint
	paint(gc); end
	function paint(gc)
	pos = koerper_neu:pos()
	vel = koerper_neu:vel()
	posX = pos:x(); posY = pos:y()
	velX = vel:x(); velY = vel:y()
	
	if posX > b-radius then
	velX = -1*math.abs(velX)
	posX = b-radius
	elseif posX < 0 then
	velX = math.abs(velX); posX = 0; end
	
	if posY > h - radius then
	velY = -1*math.abs(velY); posY = h-radius
	elseif
	posY < 0 then
	velY = math.abs(velY); posY = 0; end
	
	koerper_neu:setPos(physics.Vect(posX,posY))
	koerper_neu:setVel(physics.Vect(velX,velY))
	gc:setColorRGB(color.orange)
	gc:fillArc(posX,posY,radius,radius,0,360)
	end

Beschreibung der Aktion	Lua-Skript
Überlege, wie man das auch variieren könnte, so z.B., indem man inneren und äußeren Radius gleichsetzt und damit nur die Hülle beschreibt – wie wirkt sich dies aus? Die Startposition ist ebenso wie die Geschwindigkeit durch einen Vektor beschrieben. Jetzt, wo unser Körper definiert ist, können wir ihm eine Form (= shape) verleihen. Zur Wahl stehen Segment, Rechteck, Kreis und Polygon. Wie schon erwähnt, kann die Form die Eigenschaften Elastizität (= restitution) und Reibung (= friction) tragen. Wir werden dies in diesem Beispiel nicht berücksichtigen Elastizität = 1 bedeutet perfekte Elastizität. Elastizität > 1 erzeugt interessante Situationen, führt aber oft zu einem Fehler. Reibung > 1 führt zu einem Energieverlust bei jedem Zusammenstoß. Auch hier lässt sich mit verschiedenen Werten experimentieren. Schließlich nehmen wir den Körper samt seiner Form in den geschaffenen **raum** auf mit: raum:addBody(koerper_neu) raum:addShape(form_neu). Hier ist nochmals der Hinweis auf die on.resize-Funktion, in der on.paint auf init verweist.	function on.resize() on.paint = init end

9.3.1 Pause, Neustart und Abbruch

Unser Ball springt nun – für alle PC-Zeiten – von einer Wand zur anderen. Wir wollen ihm aber doch einmal eine Pause gönnen oder ihn wieder an seine Startposition zurückschicken.

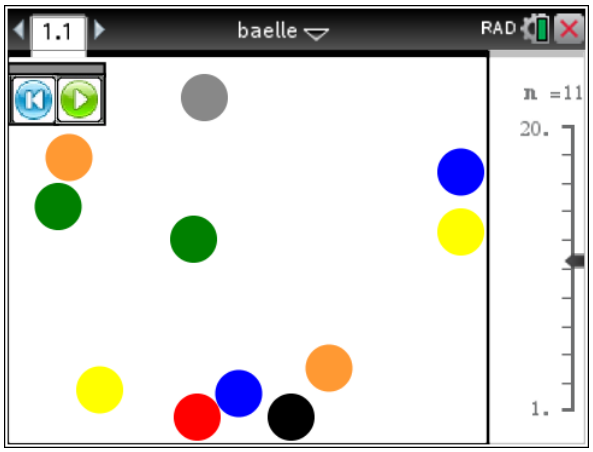
Der Ball ruht, sobald das Argument der timer-Funktion den Wert 0 annimmt. Momentan beim Wert 0,01 wird der Schirm 100-mal pro Sekunde aufgerufen, das ist das Maximum der Plattform. Wir erzeugen daher eine lokale Variable pause, der wir im Rahmen von init(gc) den Wert pause = 0 zuordnen. Das heißt, dass wir mit einem ruhenden Ball beginnen.

Dann wird über die schon bekannte on.enterKey()-Funktion der Wert für pause zwischen 1 und 0 hin- und her geschaltet. Die ESC-Taste schickt den Ball in seine Ausgangslage (linke obere Ecke) zurück.
 	
Mit ENTER schalten wir die Bewegung abwechselnd ein und aus.
	function on.enterKey()
	  pause = pause == 1 and 0 or 1
	platform.window:invalidate(); end
	
	function on.escapeKey()
	  on.resize()
	platform.window:invalidate();end
	
	function on.timer()
	  raum:step(0.01*pause)
	platform.window:invalidate(); end

9.4 Wir jonglieren mit mehreren Bällen

In diesem Abschnitt werden wir nicht nur mit mehreren Objekten (hier mit Bällen) arbeiten, sondern auch einige schon früher vorgestellte Techniken einsetzen.

Wir sehen einen Schieberegler für die Anzahl der Bälle, zwei Schaltflächen, die als Bilder eingefügt werden, die Steuerung wird über Maus und über die Tastatur erfolgen und auch das Klassenkonzept wird eingesetzt werden.



Die Symbole für die Schaltflächen wurden aus den Screenshots von einem anderen Programm ausgeschnitten und als Grafiken gespeichert. Dann konnten sie nach den Vorgaben des Kapitels 3 in die *resources* des Skripts unter den Bezeichnungen pause, play und reset aufgenommen werden.

Beschreibung der Aktion	Lua-Skript

Natürlich gibt es nun weitere lokale Variable:
 	
Für die Schaltflächensymbole, für die Farbenliste (colors), für die – zufälligen – Startpositionen der Bälle (initX, initY), für die Listen der Objekte und deren Formen u. a.
 	
grabX und grabY werden später beschrieben.
 	
Der größere Teil des Skripts sollte sich selbst erklären.
 	
Nochmals der Hinweis auf die on.resize-Funktion, in der on.paint auf init verweist.
 	
Die init-Funktion ist hier noch umfangreicher als vorhin.
 	
Sie umfasst die Liste der möglichen Farben, wobei zahl_farbe die Anzahl der möglichen Ballfarben und wahl_farbe die Nummer der gewählten Farbe angibt.
 	
Für die koerper und formen werden leere Listen (Tabellen) vorbereitet.
 	
Die Anzahl *n* der später tanzenden Bälle wird über einen Schieberegler kontrolliert. (Außerdem werden wir das auch über die Pfeiltasten ermöglichen).
	--baelle.tns
	-- Originalautor: Alfredo Rodriguez (2011)
	-- modifiziert von Steve Arnold (2012)
	-- bearbeitet von Josef Böhm (2017)
	
	platform.apilevel = '3.2'
	
	require "physics"; require "color"
	local b, h, raum
	local farben, zahl_farbe, wahl_farbe
	local zahl_koerper, koerper_neu, form_neu
	local koerper, formen, radius
	local initX, initY
	local masse, schwerkr, traegh, elast, reibung
	local pos, posX, posY, vel, velX,velY
	local im
	local pause_image=image.new(_R.IMG.pause)
	local play_image=image.new(_R.IMG.play)
	local reset_image=image.new(_R.IMG.reset)
	local pause
	local grabX = 0; local grabY = 10
	
	function init(gc)
	  b = platform.window:width()
	  h = platform.window:height()
	  zahl_koerper = (var.recall("n") or 5)
	  koerper = {}; formen = {}
	  radius = b/10
	  masse = 100; schwerkr = 9.8
	  elast = 1; reibung = 1
	  pause = 0
	  raum=physics.Space()
	  raum:setGravity(physics.Vect(0,schwerkr))
	  traegh=physics.misc.momentForCircle
	  (masse,radius,radius,physics.Vect(0,0))
	  farben = {color.gray,color.red,color.orange,
	  color.yellow,color.green, color.blue,
	  color.black}
	  zahl_farbe = 1

Beschreibung der Aktion	Lua-Skript

Herzstück ist die Schleife, in der die Objekte mit zufälligen Startpositionen und entsprechenden Farben aus der Liste erzeugt werden.
 	
Für die „Lager“ der Schaltflächen werden Rechtecke bereitgestellt, die über eine Klasse gegen Ende des Skripts definiert werden.
 	
Der „grabber“ ist ein umfassendes Bett für die Schaltflächen, das als Ganzes mit der Maus am Display verzogen werden kann.
 	
Die paint-Funktion ist der nächste größere Block im Skript.
 	
Für alle Objekte werden die Daten eingelesen und die Ränder des Displays berücksichtigt, wie im Skript zum Programm ball1.tns.
	for i=1,zahl_koerper do
	koerper_neu=physics.Body(masse,traegh)
	koerper_neu:setVel(physics.Vect(500,500))
	initX=math.random(b); initY=math.random(h)
	koerper_neu:setPos(physics.Vect(initX,initY))
	form_neu=physics.CircleShape(koerper_neu,
	radius/2,physics.Vect(0,0))
	form_neu:setRestitution(elast)
	form_neu:setFriction(reibung)
	table.insert(koerper,koerper_neu)
	table.insert(formen,form_neu)
	raum:addBody(koerper_neu)
	raum:addShape(form_neu); end
	
	grabX = grabX or 0; grabY = grabY or 10
	grabber = Rechteck(grabX,grabY,2*radius,
	0.2*radius,false)
	reset_button = Rechteck(grabber.x,
	grabber.y+radius, radius,radius,false)
	go_button = Rechteck(grabber.x+radius,
	grabber.y+radius, radius,radius,false)
	
	on.paint = paint
	paint(gc)
	timer.start(0.01); end
	
	function paint(gc)
	for k=1,zahl_koerper do
	pos = koerper[k]:pos()
	vel = koerper[k]:vel()
	posX = pos:x(); posY = pos:y()
	velX = vel:x(); velY = vel:y()
	if posX > b - radius then
	velX = -1*math.abs(velX)
	posX = b - radius
	elseif posX < 0 then
	velX = math.abs(velX); posX = 0; end
	if posY > h - radius then
	velY = -1*math.abs(velY)
	posY = h - radius
	elseif posY < 0 then
	velY = math.abs(velY);posY = 0; end

Beschreibung der Aktion	Lua-Skript

Sind die Positionen bezogen, dann können die Kreise mit der jeweils nächsten Farbe aus der Liste gezeichnet werden.	koerper[k]:setPos(physics.Vect(posX,posY))
	koerper[k]:setVel(physics.Vect(velX,velY))
	wahl_farbe=
	farben[((zahl_farbe+k)%#farben)+1]
	gc:setColorRGB(wahl_farbe)
	gc:fillArc(posX,posY,radius,radius,0,360); end
	
	grabber:paint(gc)
	reset_button:paint(gc)
	go_button:paint(gc)
	
	if pause == 0 then
	im = image.copy(play_image,radius,radius)
	else
	im = pause_image:copy(radius,radius); end
	
	imb,imh = im:width(), im:height()
	gc:drawImage(im,grabber.x+imb,
	grabber.y+radius-imh)
	im1=image.copy(reset_image,radius,radius)
	im1b,im1h=im1:width(),im1:height()
	gc:drawImage(im1,grabber.x,
	grabber.y+radius-im1h)
	end

Hier befindet sich die Routine zum Einfügen der Schaltflächen.	function on.timer() raum:step(0.01*pause) platform.window:invalidate(); end function on.resize() on.paint=init; end end
Nun folgen die bereits bekannten Teile, welche die ENTER- und ESC-Taste betreffen.	function on.escapeKey() on.resize() platform.window:invalidate(); end function on.enterKey() pause = pause == 1 and 0 or 1 platform.window:invalidate(); end

Beschreibung der Aktion	Lua-Skript

Mit der Maus sprechen wir die Schaltflächen und auch den Schieberegler an.
 	
Dieser muss in der nun vertikal geteilten Seite in einer Geometry-Seite für die Variable *n* eingerichtet werden.
 	
Mit den Pfeiltasten können wir die Anzahl der Bälle um jeweils eins erhöhen oder vermindern.
	function on.mouseDown(x,y)
	  if grabber:contains(x,y) then
	  grabber.selected = true
	  else
	grabber.selected = false; end; end
	
	function on.mouseUp(x,y)
	  grabber.selected=false
	  if go_button:contains(x,y) then
	  on.enterKey()
	  elseif reset_button:contains(x,y) then
	  on.escapeKey()
	  else
	  for k = 1,#formen do
	  koerper[k]:setPos(physics.Vect(x,y)); end
	  end; end
	platform.window:invalidate(); end
	
	function on.mouseMove(x,y)
	  if grabber.selected==true then
	  grabber.x=x; grabber.y=y
	  reset_button=Rechteck(grabber.x,
	  grabber.y+radius,radius,radius,false)
	  go_button=Rechteck(grabber.x+radius,
	  grabber.y+radius,radius,radius,false)
	  end
	grabX=grabber.x; grabY=grabber.y; end
	
	function on.arrowUp()
	  zahl_koerper=(var.recall("n") or 5)
	  zahl_koerper=zahl_koerper +1
	  var.store("n",zahl_koerper)
	on.resize();end
	
	function on.arrowDown()
	  if zahl_koerper > 1 then
	  zahl_koerper = (var.recall("n") or 5)
	  zahl_koerper = zahl_koerper - 1
	  var.store("n",zahl_koerper)
	on.resize(); end; end

Schließlich fehlt noch die Klasse Rechteck, die noch zu definieren ist:

Beschreibung der Aktion	Lua-Skript

Schluss des Skripts:	Rechteck = class()
	function Rechteck:init(x,y,width,height,selected)
	  self.x = x; self.y = y
	  self.width = width
	  self.height = height
	self.selected = selected; end
	
	function Rechteck:contains(x,y)
	  local sw = self.width
	  local sh = self.height
	  return x >= self.x and x <= self.x+sw and
	  y >= self.y-sh and y <= self.y; end
	
	function Rechteck:paint(gc)
	  local sw = self.width
	  local sh = self.height
	  gc:setPen("medium","smooth")
	  gc:setColorRGB(color.black)
	  gc:drawRect(self.x,self.y-sh,sw,sh)
	  gc:setColorRGB(color.gray)
	gc:fillRect(self.x,self.y-sh,sw,sh); end

Nun, das war ja eine ganz schöne Dosis an Programmierarbeit, an der man auch eine ordentliche Menge zum Nachdenken gezwungen wird, wenn man sich nicht nur mit dem Abtippen des Codes zufriedengeben will.

9.4.1 Unsichtbare Mauern

Wir ergänzen das Skript von oben so, dass vorerst keinerlei Veränderungen am Display sichtbar werden. Dabei wird ein neuer Typ der Form (= shape) – das **segment** – vorgestellt. Außerdem werden wir neue Skript-Techniken lernen, die in Zukunft hilfreich sein können.

Hier soll nicht das komplette Skript angegeben werden. Wir beschreiben nur die Änderungen, die am vorigen Skript vorzunehmen sind:

Wesentlich geändert werden die init- und die paint-Funktion:

Die Liste der lokalen Variablen wird ergänzt um local grenzen.

In die init-Funktion wird für die Grenzen eine leere Tabelle aufgenommen: grenzen = {}.

In der paint(gc)-Funktion können die Abfragen, ob der Bildschirmrand erreicht wird, gelöscht werden. Das ist der Block zwischen if posX > b - radius then und velY = math.abs(velY);posY = 0; end.

Anstelle dessen werden unsichtbare Mauern am Bildschirmrand gezogen. Wir geben ihnen keine Form, aber sie können Eigenschaften erhalten, wie z.B. hier ihre Stärke (1 Einheit) und ihre Elastizität (hier die gleiche wie die Bälle).

Unmittelbar im Anschluss an die init-Funktion definieren wir die Funktion raender(bb,hh), in der diese Begrenzungen als „Segmente“ definiert werden.

```
function raender(bb,hh)
  for _,rand in ipairs(grenzen) do
    raum:removeStaticShape(rand)
  end

  local grenze
  grenze=physics.SegmentShape(nil,physics.Vect(-radius/2,-radius/2),
            physics.Vect(bb-radius/2,-radius/2),1)
    :setRestitution(elast)
    :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)

  grenze=physics.SegmentShape(nil,physics.Vect(-radius/2,hh-radius/2),
            physics.Vect(bb-radius/2,hh-radius/2),1)
    :setRestitution(elast)
    :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)

  grenze=physics.SegmentShape(nil,physics.Vect(-radius/2,-radius/2),
            physics.Vect(-radius/2,hh-radius/2),1)
    :setRestitution(elast)
    :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)

  grenze=physics.SegmentShape(nil,physics.Vect(bb-radius/2,-radius/2),
            physics.Vect(bb-radius/2,hh-radius/2),1)
    :setRestitution(elast)
    :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)
end
```

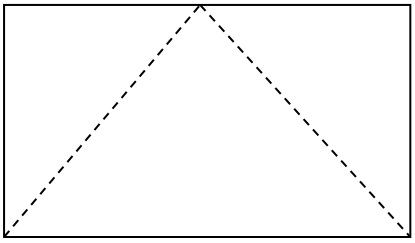
Die Segmente werden in die Tabelle grenzen (schon früher als lokale Variable und als leere Tabelle im Skript definiert) über **physics.SegmentShape** aufgenommen. Sie werden als „statische“ Formen (= StaticShape) deklariert, da sie nicht bewegt werden.

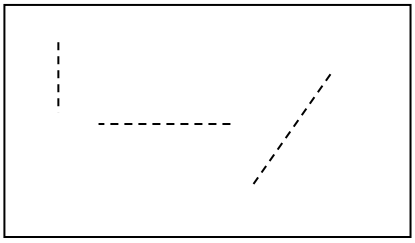
Zuerst werden eventuell schon vorhandene statische Formen gelöscht. Die ipairs-Schleife ist eine andere Form einer for-Schleife. Dann wird eine lokale Variable grenze definiert, die der Reihe nach mit den Segmenten belegt wird. Als Parameter dienen der Koerper (den es hier nicht gibt, daher schreiben wir nil), Anfangs- und Endpunkt der Strecke (als Vektoren definiert) und schließlich die Dicke der Strecke. Anschließend verpassen wir noch zusätzliche Eigenschaften wie Elastizität und Reibung. Da die Eigenschaften unmittelbar auf die Definition folgen, reicht ein Doppelpunkt.

Das sollte nun funktionieren!

Probieren Sie nun, ob die Bälle weiter vom Rand wegbleiben, wenn Sie die „Mauerdicke“ vergrößern. Ich empfehle aber, keine absoluten Werte, wie 10, 20, 100, … zu wählen, da dies beim Übergang zum Handheld Probleme machen kann. Anstelle dessen sind relative Werte wie b/20, h/20 usw. zu empfehlen.

Es ist nicht allzu schwierig, andere Mauern zu bauen. Versuchen Sie, eine dreieckige Barriere zu errichten. Auch ein Hindernisparcours mit mehreren Abschnitten kann Spaß machen. Dabei sollten die Bälle eher kleiner gehalten werden.





Natürlich lässt sich auch mit Reibung und Elastizität der Mauern experimentieren.

Es sollte auch gelingen, die Mauern sichtbar zu machen. Erinnern Sie sich bitte an Kapitel 2.

Einen Vorschlag für die entscheidenden Zeilen für das „Geisterdreieck“ finden Sie auf der nächsten Seite. Man darf aber nicht vergessen, die Bildschirmränder auch zu aktivieren (entweder als Segmente, oder so wie im letzten Skript), sonst verschwinden Bälle vom Schirm und kehren nie wieder.

Damit sind wir zum letzten Kapitel dieser Einführung ins Programmieren mit Lua angelangt. Wir werden die Körper verallgemeinern – siehe Abbildung am Beginn des Kapitels 9 – und Vielecke mit veränderlicher Eckenzahl sowie variabler Fluggeschwindigkeit auf den Bildschirm bringen.

Das „Geisterdreieck“:

Am einfachsten ist es, die beiden folgenden Segmente hinzuzufügen:

```
grenze=physics.SegmentShape(nil,physics.Vect(-radius/2,hh-radius/2),
          physics.Vect(bb/2,-radius/2),1)
   :setRestitution(elast)
   :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)

grenze=physics.SegmentShape(nil,physics.Vect(bb/2,-radius/2),physics.
         Vect(bb-radius/2, hh-radius/2),1)
   :setRestitution(elast)
   :setFriction(reibung)
  raum:addStaticShape(grenze)
  table.insert(grenzen,grenze)
```

Ein einfacher Vorschlag für die Visualisierung der Mauer sieht so aus:

```
gc:setColorRGB(color.red)
gc:setPen("medium","smooth")
gc:drawPolyLine({0,h,b/2,0,b,h})]]
```

Diese Zeilen setzen Sie ans Ende der paint(gc)-Funktion.

9.5 Polygone im „magischen Glitzern“

Die notwendige Erweiterung für die Darstellung von Vielecken ist überraschend einfach. Im Wesentlichen werden zwei neue Befehle benötigt, dass unsere Form als Vieleck erkannt wird, das **PolyShape**-Kommando: **physics.PolyShape(koerper, ecken, offset)** und eine andere Definition der Trägheit: **physics.misc.momentForPoly(masse, ecken, offset)**. Es ist zu beachten, dass Polygone durch eine Schar von Strecken begrenzt werden. Die umschlossene Fläche muss konvex sein und die Ecken sind im mathematisch positiven Sinn zu definieren.

Wir wollen nicht das ganze Skript zeigen, sondern werden nur auf die notwendigen Änderungen bzw. Ergänzungen eingehen.

Beschreibung der Aktion	Lua-Skript
Auf local farben, zahl_farbe, wahl_farbe können wir verzichten, da wir hier die Farbtabelle nicht brauchen werden. Dafür müssen wir neue lokale Variable aufnehmen: In der init(gc)-Funktion: Zwei zusätzliche Schieberegler für die Eckenanzahl ($3 \leq eck \leq 15$) und für die Geschwindigkeit ($100 \leq v \leq 1000$ mit Schrittweite 50) sind einzuführen. Auf sie muss im Skript verwiesen werden. Zwei zusätzliche Listen sind bereitzustellen. Der radius bekommt einen festen Wert. (Auch dieser könnte über einen Schiebregler variabel gehalten werden.) Diese Variablen werden für die Konstruktion der Polygone benötigt.	local zahl_ecken, ecken, eckpunkte, points zahl_ecken = var.recall("eck" or 5) vel = var.recall("v" or vel) ecken = {}; eckpunkte = {} radius = b*0.075 local winkelDiff=360/zahl_ecken local winkel = 0
Die Tabelle (Liste) ecken wird gefüllt. Das zusätzliche Argument 1 sorgt dafür, dass jeder Vektor jeweils als erster der Liste hinzugefügt wird. Damit wird die mathematisch positive Orientierung sicher gestellt.	for i=1,zahl_ecken do table.insert(ecken,1,physics.Vect (math.cos(math.rad(winkel))*radius, math.sin(math.rad(winkel))*radius)) winkel = winkel + winkelDiff end

Die Definition der Trägheit (*inertia*):
In der zahl_koerper-Schleife ist die neue Form der Polynome zu definieren:
Sonst ist in dieser for-Schleife nichts zu ändern.
Dafür aber umso mehr in der paint(gc)-Funktion:
In einer doppelten Schleife werden zuerst die formen durchlaufen und für diese dann die Koordinaten der eckpunkte für jede form (= jedes Polygon) ausgelesen.
Sowohl Steve Arnold als auch ich wissen nicht, warum Lua hier nur den Variablen-
	traegh=physics.misc.momentForPoly
	(masse,ecken,physics.Vect(0,0))
	form_neu=physics.PolyShape(koerper_neu,ecken,
	physics.Vect(0,0))
	
	for _,form in ipairs(formen) do
	local punktz = 1
	points = form:points()
	for _,punkt in ipairs(points) do
	eckpunkte[punktz] = punkt:x()
	eckpunkte[punktz] = eckpunkte[punktz] < b
	and eckpunkte[punktz] or b
	eckpunkte[punktz] = eckpunkte[punktz] > 0
	and eckpunkte[punktz] or 0

namen **points** akzeptiert. Jede andere Bezeichnung als points führt zu einer Fehlermeldung. Alle anderen bezeichnungen ließen sich problemlos ins Deutsche übertragen.	eckpunkte[punktz+1] = punkt:y() eckpunkte[punktz + 1] = eckpunkte [punktz + 1] < h and eckpunkte[punktz + 1] or h eckpunkte[punktz + 1] = eckpunkte[punktz + 1] > 0 and eckpunkte[punktz + 1] or 0 punktz = punktz + 2 end
Hier erzeugen wir das „magische Glitzern”: die Füllfarbe wird als Kombination von zufälligen RGB-Werten erzeugt und bei jedem resize frisch gebildet.	gc:setColorRGB(math.random(255), math.random(255), math.random(255)) gc:fillPolygon(eckpunkte) gc:setColorRGB(0) gc:drawPolyLine(eckpunkte) end

Schließlich wollen wir die neuen Schieberegler auch noch über die Tastatur kontrollieren:
Die Art des Vielecks – beginnend mit einem Dreieck bis zu einem 15-Eck wird mit den Pfeiltasten nach rechts und links gesteuert.
(Mehr Ecken machen wenig Sinn, da die Polygone, dann fast kreisförmige Gestalt annehmen. Die Zahl kann aber jeder beliebig erhöhen – in Skript und im Schieberegler.)
	function on.arrowRight()
	zahl_ecken = var.recall("eck") or 5
	if zahl_ecken < 15 then
	zahl_ecken = zahl_ecken + 1
	var.store("eck", zahl_ecken)
	on.resize(); end; end
	
	function on.arrowLeft()
	zahl_ecken = var.recall("eck") or 5
	if zahl_ecken > 3 then
	zahl_ecken = zahl_ecken - 1
	var.store("eck", zahl_ecken)
	on.resize(); end; end

Schön langsam gehen uns die Steuertasten aus. Die TAB-Taste steht noch zur Verfügung.
Für die Verringerung der Geschwindigkeit nehmen wir die "BackTab-Taste", die es eigentlich nicht wirklich gibt. Das ist die Tastenkombination Shift+TAB.
Damit haben wir auch dieses Beispiel genau beschrieben.
	function on.tabKey()
	vel = var.recall("v") or vel
	if vel < 1000 then
	--vel = var.recall("v") or vel
	vel = vel + 50
	var.store("v", vel)
	on.resize(); end; end
	
	function on.backtabKey()
	vel = var.recall("v") or vel
	if vel >= 50 then
	vel = vel - 50
	var.store("v", vel)
	on.resize(); end; end

Wir sind am Ende unserer doch recht ausführlichen „Einführung" ins Programmieren mit Lua angelangt.

Wir - die Kollegen aus Belgien, Steve Arnold und ich – hoffen, dass die Erklärungen so weit wie in der Kürze möglich, ausreichend waren, und dass Sie Lust bekommen haben, sich weiter mit Lua zu beschäftigen. Im Internet sind viele Schätze vorhanden. Sie warten darauf, gehoben zu werden. Ein paar Hinweise wurden bereits im Text gegeben. Einige weitere folgen hier.

Referenzen zu Lua

http://compasstech.com.au/TNS_Authoring/Scripting/index.html

http://www.inspired-lua.org/

http://www.lua.org/

http://lua-users.org/wiki/

https://education.ti.com/en/resources/lua-scripting

https://education.ti.com/html/webhelp/EG_TINspireLUA/EN/index.html

The Lua Scripting API Reference Guide:

https://education.ti.com/en/guidebook/details/en/59108CCE54484B76AF68879C217D47B2/ti-nspire_scripting-api-guide

Programming in Lua (1st and 2nd edition, Roberto Ierusalimschy):

https://www.lua.org/pil/contents.html

http://it-ebooks.directory/book-859037985x.html
https://scanlibs.com/programming-in-lua-3rd-edition/

Beginning Lua Programming (Kurt Jung und Aaron Brown):

http://crypto.cs.mcgill.ca/~simonpie/webdav/ipad/EBook/Programmation/Beginning%20Lua%20Programming.pdf

Speziell für die *Physics Engine*:

https://wiki.inspired-lua.org/Category:Physics_Engine

https://www.youtube.com/watch?v=K5PW4998zII (Basketballspiel)

https://www.omnimaga.org/lua-language/