

The top half of the cover features a series of overlapping, wavy bands of blue, ranging from a deep navy blue at the top to a lighter sky blue at the bottom, creating a sense of depth and movement.

Nspire LUA and its applications

First edition

Nick Steen

Nick Steen
Nspire LUA and its applications

© 2011, Nick Steen

ALL RIGHTS RESERVED. This book contains material protected under International and Federal Copyright Laws and Treaties. Any unauthorized reprint or use of this material is prohibited. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system without express written permission from the author.

Inhoud

Introduction.....4

TI-Nspire.....4

General.....5

Wiki5

Lua.....5

Data types.....6

Numbers6

Strings6

Lists7

Comments.....7

Boolean8

Events / functions8

General syntax8

Triggering a function.....9

Self-made functions.....10

Graphical Context..... 11

Math..... 12

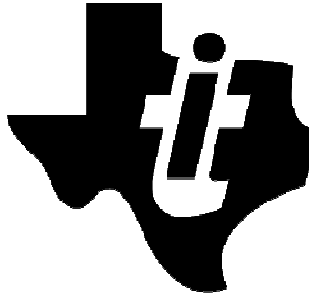
General.....12

Assignment and comparison12

Other math. functions12

Flow control..... 14

Tutorials 15



Introduction

TI-Nspire

The TI-Nspire is a calculator which got the possibilities to run Lua scripts. This possibility is not unlimited, but they are big enough to set up some nice working programs. Because Lua already existed before the Nspire supported it, the language has made a huge development. Many commands have been made possible within the language. Though Lua has a large library of functions and commands, the Nspire usage is limited for a simple reason: speed.

Speed is the only real problem you might encounter, because there are no possibilities to solve this 'native' problem. But when you'll start using it, you will see that this won't become an issue unless you're wanting to make heavy graphical games with e.g. 3D functions.

Or to say all this in terms of inspired-Lua:

"This is a new era for TI-Nspire programming !"



General

Wiki

On <http://www.inspired-lua.org> and <http://wiki.inspired-lua.org> you're able to see an overview of the functions, events, tutorials and more. This guide has been built upon the wiki, as this was the only place where an overview of the API was given.

If you want to learn more after this guide, or you only want the tutorials, don't hesitate on visiting it, it is surely worth your visit.

Lua

Lua in general is a language that has been in continuous development since 1993. It can easily be described:

"Lua is a powerful, fast, lightweight, embeddable scripting language."

"Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping."

It's an interpreted language that consist of an easy syntax. Every event can be caught with a specified function, instead of having to make a listener for that event (java, actionscript..). Because Lua is small, yet powerful, the possibilities are huge.

There's only one special remark. Lua sees a difference between lowercase an uppercase, so all functions and variable must be typed in the correct case! So variable a is not the same as variable A. Be aware of this when programming, since those thing are hard to find if you don't keep it in mind.

Data types

Numbers

Number

```
newvar = 5  
  
newvar returns 5
```

What is a program without numbers? Every program uses numbers, so that's why it is the start of the guide. In Lua, no declaration is needed and there is no limit on the value of the number. You can directly initialize the variable by giving it its standard form or directly a value. This saves a lot of space and programming in comparison with other languages. So when you start a program, you only have to type this for example:

```
newnumber = 5
```

Globals Locals

There's a difference between global variables and local variables. Global variables can be initialized anywhere, and it keeps its value. You can use those globals anywhere in the program. Locals are, as the name says, only initialized and useable within the function or loop it is defined. So when you define a local within a function and you want to read it somewhere else, you'll have to store it in a global, or use `return` to make it being returned to the place where the function was called. This will be explained later. To initialize a global variable, nothing special is needed, so just do it the way you did in the first example. To make a variable local, just write `local` in front of it. Locals reduces the usage of the memory to store the variables, since it only stores it for a short moment.

Example:

```
function on.paint(gc)  
    local localvar = 5    --will return 5 within on.paint(gc), nil out of it  
    globalvar = 10        --will always return 10  
end
```

Strings

String

```
stringvar = "Hello!"  
  
stringvar returns "Hello!"
```

Strings are series of characters. They are used to display text in programs, or to manipulate a certain succession of characters. There are many instructions available to manipulate them, but before you can do that, you have to know how to build a string. Strings are simple to use, and can be stored in a variable with any name you want. The only name you may not use is "string", since that word is used by Lua itself as a constructor. We'll explain that later.

Example:

```
newstring = "Hello World!"
```

All characters, except for accentuated ones are supported. With the coming SDK, those characters will be supported too. Also the same counts for global and local strings as was explained in the number part. When writing `local` in front of the initialization, a string will only return its value within the function or loop in which it is created, and `nil` (error) when outside the function.

Nil

List

```
list = {"Hello",2,"j"}
list[1] returns "Hello"
```

```
list = {"Hello",2,{"Lua",2}}
list[3][2] returns 2
list[2] returns 2
```

Lists

Lists are used to store values together in a kind of table. They are easy to make, read and manipulate. As you can see in the example below, every list element is found by the name, followed by the place of the value in the list.

Example:

```
newlist = {2,"j","2","?"}
```

You can easily read from the list by entering the name of the list, followed by [number]. Number is the place of the value you're looking for, starting with 1 as first entry of the list. As you can see, all datatypes can be stored in the lists as they can be stored in other variables. When we follow that reasoning, we should be able to insert a list into a list...

```
newlist = {2,"j","2","?",{8,7,"hello"}}
```

The way you read those variables is the same as you do from a single list. You only have to add new [number]'s to reach the most nested value. So when you have three lists in each other, and you want the 1st value of the 3rd list of the 2nd list, you'll have to do listname[2][3][1]. When listname={{1,0,1},{0,1,0},{0,1,0}},{0,1,0},{1,1,0},{**0**,1,0}}, then listname[2][3][1] will return 0, from the place that is in red bold.

Comments

Comment

When writing a program, It's very important to comment what you're doing. Certainly when a lot of different functions or calculations are used. When you're wanting to understand the code in an certain amout of time, you'll have to be sure that every difficult step is commented. You can comment on two ways: a full line comment, or a partial comment.

Example full line comment:

```
newvar = 5 --this is a comment till the end of the line
```

Example of a partial comment

```
newvar = {--[this is a partial comment]]5,2}
```

```
-- This is a comment
```

```
--[[this is a partial comment]]
```

CONCLUSION CHAPTER 1

As seen in this first chapter, there's no need to declare variables. The initialization can directly happen with a value or a nil. That means you can initialize a table with e.g. {"Hello"} or {}. You can simply get the value of a variable by using the name of it, and when it's a table, followed by the place of the value in the list.

Commentsyntax is -- for a full line comment and --[[...]] for a partial comment.

Boolean

Boolean variables are variables that only have two states: true or false. Like the state says, it can keep track of something, and return true or false. This is useful when obtaining a

Event - function

Events / functions

Everything in lua is based on events. Events are functions that are called when something happens. This something can be a keypress, the redrawing instruction etc. Every event has the same general syntax, but some have arguments, and other don't. In this chapter, all possible events will be explained.

NOTE: remember that "A" is different to "a" in Lua, so every function has its capitals for a reason!

General syntax

Every event calls a function. A function follows a certain syntax, as you can see in the example.

```
function on.event([arguments])  -- with event the name of a function from below
    --write instructions here
End
```

```
Function on.paint(gc)
    local varnr = 10
    -- varnr returns 10
End
Varnr returns nil
```

Function name	Arguments	Called when
on.paint(gc)	GC is the graphical context (explained later)	platform.window:invalidate() is called
on.arrowKey(arrow)	Arrow returns 'left','right','up' or 'down'	An arrow key had been pressed
on.arrowLeft()	-	The left arrow key has been pressed
on.arrowRight()	-	The right arrow key has been pressed
on.arrowUp()	-	The up arrow key has been pressed
on.arrowDown()	-	The down arrow key has been pressed
on.enterKey()	-	The enter key has been pressed
on.escapeKey()	-	The escape key has been pressed
on.tabKey()	-	The tab key has been pressed
on.charIn(char)	Char can return any char, e.g. char = '+'	Any character key has been pressed
on.mouseMove(x,y)	x and y return the position of the mouse	The mouse moves
on.mouseDown(x,y)	x and y return the position of the mouse	The mouse has been pressed
on.mouseUp(x,y)	x and y return the position of the mouse	The mouse has been released
on.create()	-	The file gets opened (made)
on.destroy()	-	The file gets closed (destroyed)
on.resize()	-	The screen is resized (when in the pc software)
on.timer()	-	The timer is called
on.deleteKey()	-	The delete key has been pressed
on.backspaceKey()	-	The backspace key has been pressed

on.returnKey()	-	The return key has been pressed
on.contextMenu()	-	The [ctrl][menu] key has been pressed
on.help()	-	The [ctrl][trig] has been pressed
on.clearKey()	-	The clear key has been pressed
on.activate()	-	The tab/page gets activated
on.deactivate()	-	The tab/page gets inactivated
on.blink()	-	The focus gets lost on the page

Triggering a function

Triggering a function means you call it by changing something, by using a command, or by a keypress. Every function can be called with its original name. so if you want to trigger the `on.arrowKey(arrow)` function when no arrowkey is pressed, but e.g. 4 is pressed, then you just have to insert `on.arrowKey('left')` within the `on.charIn(char)` function. Some functions, like `on.paint(gc)` and `on.timer()` have special triggers.

```
function on.create()
  timer.start(1)
end

function on.paint(gc)
  gc.drawString("Hello",0,0)
End

on.timer()
platform.window:invalidate()
end
```

`platform.window:invalidate()` triggers the `on.paint(gc)` function. This way it can be used within any function to redraw the screen. The only way you mustn't use it is within `on.paint(gc)` itself. Sometimes you want the screen to be updated continuously, but you better use a timer that calls `platform.window:invalidate()` every 0.05 seconds. Of course this amount of time is relative. If huge functions are being calculated in between the counting, it's impossible for the calc to redraw the screen every 0.05 seconds. It's better to use a value that is realistic. This needs some experimenting, thought It will work when you use 0.05). It is recommended that you don't repaint the screen continuousle. It's better to use the optional redrawing: only redraw when a change to the buffer has been done.

These are the instructions:

<i>timer.start(value)</i>	<code>timer.start(value)</code>	-- triggers the <code>on.timer()</code> event every -- 'value' seconds -- Can be stopped with <code>timer.stop()</code> -- triggers the <code>on.paint(gc)</code> function
<i>Platform.window:invalidate()</i>	<code>platform.window:invalidate()</code>	

other functions are called with their name:

```
on.arrowKey('left')
on.help()
on.escapeKey()

etc.
```

Every time a function is called within another function, it returns to the point where the functions was called when no `platform.window:invalidate()` is called. So that way you can do calculation after each other, each in a different function, and placing the functions below each other in the same main function. This will be explained in Self-made Functions.

```
function on.create()
    myvar = 10
end

function on.paint(gc)
    printstring(gc,myvar)
End

function printstring(gc,string)
    gc:drawstring(string,0,0)
end
```

Self-made functions

The same way you can call a function, you can make you own. It uses the same syntax, but it gets called when you trigger it. If you want to do a heavy mathematical calculation that exists out of different steps, or every part of the function must be able to be called in another calculation, then writing your own function is the best option. It's not hard after all. You can call it the way you want, but again, no words from the Lua language itself might be used (see Appendix A).

Another specialty about the on.paint(gc) is that that is the only function where gc must be called. Every command has to be called withing on.paint(gc) or a function to which the gc is passed from within on.paint(gc). This might sound difficult, but it isn't.

Example:

```
function on.create()
    myvar = 10
    multiplier = 5
    addition = 90
    multiply(myvar,multiplier)
    add(newvar,addition)
end

function multiply(var,multi)
    newvar = var * multi
end

function add(var,adder)
    newvar = var + adder
end

function on.paint(gc)
    printstring(gc,newvar,10,10,"top")
end

function printstring(gc,string,x,y,pos)
    gc:drawString(string,x,y,pos)
end
```

Is the same as:

```
function on.create()
    myvar = 10
    multi = 5
    addit = 90
    newvar = myvar*multi+addit
end

function on.paint(gc)
    gc:drawString(newvar,10,10,"top")
end
```

In Lua, whole documents are built upon events. You don't need to assign values to the keys and continuously check if a key is pressed, only add a function that gets called when something happens. This makes a Lua program very simple to learn, and easy to read.

Functions can be called with their name, on.paint(gc) with platform.window:invalidate(), on.timer with timer.start(value)

The Graphical Context can be passed from on.paint(gc) to any self-made function by just giving gc as argument in both the trigger and the name.

Graphical Context

Graphical Context

buffer

If you have read the last chapter, you'll have noticed the gc as argument for the on.paint(gc) function. This gc is called the Graphical Context and represents a sort of a buffer where every gc instructions gets draw to, wherafter the buffer is copied to the screen. To fulfill this copy, nothing special must be done, the function on.paint(gc) only need to get closed.

We have several instructions to manipulate this buffer, such a writing text, drawing images and lines, circles, rectangles, setting a specified color etc. Every instruction has its own arguments, which will be shown below. And again, watch the capitals.

Instruction	Arguments
gc:drawString(string,x,y,positioning)	Positioning is "middle", "top", "bottom" or "baseline", where the positioning is placed
gc:drawRect(x,y,w,h)	w and h are the width and height of the rectangle, measured in pixels
gc:fillRect(x,y,w,h)	w and h are the width and height of the rectangle, measured in pixels
gc:drawArc(x,y,w,h,start,angle)	start is the start angle (all in degrees) for the arc, a circle is e.g. (0,0,30,30,0,360)
gc:fillArc(x,y,w,h,start,angle)	start is the start angle (all in degrees) for the arc, a circle is e.g. (0,0,30,30,0,360)
gc:drawPolygon({x1,y1,x2,y,2...})	A list as argument, it connects the points and gets back to the start
gc:fillPolygon({x1,y1,x2,y,2...})	A list as argument, it connects the points and gets back to the start
gc:drawLine(x1,y1,x2,y2)	Draws a line from (x1,y1) to (x2,y2)
gc:drawPolyLine({x1,y1,x2,y,2...})	A list as argument, it connects the points and does not get back to the start
gc:setColorRGB(Red,Green,Blue)	Sets the color, with R, G and B values for red, green and blue from 0 to 255
gc:setFont(font,type,size)	Sets font, font = "sanserif" or "serif", type = "r", "b", or "l" and size from "6" to "11"
gc:drawImage(image,x,y)	Draws an image on (x,y), image is a special formal, see Appendix B
gc:getStringHeight(string)	This does not draw anything! It returns the height of the passed string
gc:getStringWidth(string)	This does not draw anything! It returns the width of the passed string

11

Example:



```
function on.create()
    newvar = 5
    newstring = "Hello World"
end

function on.paint(gc)    -- This is always called one time when the document
                           -- made, so no call to it is
    gc:drawString(newvar,10,10,"top")
    gc:setFont("sanserif","b","11")
    gc:drawString(newstring,80,40,"bottom")
    gc:setColorRGB(50,150,50)
    gc:fillArc(190,170,40,40,30,90)
end
```

*Assignment
Statement
Comparison*

```
5 > 6 returns false
5 < 6 returns true
5 == 6 returns false
5 >= 6 returns false
5 <= 6 returns true
5 ~= 6 returns true
```

Math

General

Mathematics are simple to use. Every mathematical function can be called with `math.something`. When using notepad++, this becomes violet, when using SciTE this will become blue. One important thing is to watch out for the use of '=', since they must be used alone to assign a value to a variable, but must be typed twice as statement for comparison. We'll explain this later.

Assignment and comparison

As seen in the chapter before, we assign a value to a variable using '='. It's as easy as could be. Comparison is simple too, just use the `<`, `>`, `==`, `>=`, `<=` and `~=` to compare. `<` and `>` return true if the left variable is respectively smaller and bigger than the right one. With `==` you do any check you want, the variables mustn't be from the same type. No comparisons can be done on total lists. Therefore you'll need to check every single value from it and compare with the values from the other list. `>=` and `<=` compare two variables if respectively 'bigger or same as' and 'smaller or same as'. The last one, the `~=`, checks if a variable is not the same as.

Other math. functions

A lot of mathematical functions are available through the constructor `math.`. Sine, cosine, square roots, rounding etc. This list is probably not complete, so if you find new functions, please let us know about them. The list can be found on the next page.

CONCLUSION CHAPTER 3 & 4

The Graphical Context is a buffer that contains all the data that has to be drawn to the screen. You can easily add data to it by using one of the commands of the list, no own commands can be made to draw to the screen.

The buffer is drawn to the screen when the `on.paint(gc)` closes, no command is needed for that.

In Lua, a lot of mathematical functions can be used to determine a status, or to get a value returned.

Comparisons are the same as in other languages, except for the `~=` that means not equal to.

Function	Result
math.random(start,end)	Start is 0 by default, returns a random integer between start and end When no arguments, it returns a random value between 0 and 1
math.randomseed(number)	Randomseed is used to obtain a series of pseudo random numbers, based on the seed, and can be used to generate a random.
math.floor(number)	Returns the ceiling of a value, e.g. math.ceil(2.928) returns 2
math.ceil(number)	Returns the ceiling of a value, e.g. math.ceil(2.128) returns 3
math.pi	Returns 3.1415926535898
math.sin(degree)	Calculates the sine upon the degree
math.cos(degree)	Calculates the cosine upon the degree
math.tan(degree)	Calculates the tangent upon the degree
math.asin(ratio)	Calculates the angle upon the ratio of 2 sides of the triangle
math.acos(ratio)	Calculates the angle upon the ratio of 2 sides of the triangle
math.atan(ratio)	Calculates the angle upon the ratio of 2 sides of the triangle
math.atan2(side1,side2)	Calculates the angle upon the length of the sides of the triangle
math.sinh(number)	Returns the hyperbolic sine of number
math.cosh(number)	Returns the hyperbolic cosine of number
math.tanh(number)	Returns the hyperbolic tangent of number
math.sqrt(number)	Returns the square root of number like \sqrt{number}
math.abs(number)	Returns the absolute value of a number like $ number $
math.deg(rad)	Converts radian to degrees
math.rad(degree)	Converts degree to radian
math.exp(power)	Returns e to the 'power'th power
math.log(number)	Returns the natural log of number
math.log10(number)	Returns the 10 th log of number
math.min(nr1,nr2,nr3...)	Returns the minimum of the numbers
math.max(nr1,nr2,nr3...)	Returns the maximum of the numbers
math.pow(x,y)	Returns the value x^y
math.modf(number)	Returns the integral and fractional parts of number, e.g. math.modf(-10.318) returns -5 and -0.318 (approximated) You can store these values by doing var1, var2 = math.modf(number)
math.fmod(number,math.modf(number))	Returns the fractional part of number, e.g. math.modf(-10.318) returns -0.318 (approximated)
math.frexp(number)	Returns a number split in fraction and exponent e.g. math.frexp(2) returns 0.5 and 2 $number = fraction * 2^{exponent}$
math.ldexp(var1,var2)	Returns the floating point from var1 and var2 e.g. math.ldexp(0.785,2) returns 3.14 $floating\ point = var1 * 2^{var2}$
math.huge	Represents infinity and returns infinity e.g. $1/0 == math.huge$ returns true

Flow control

Flow control is using statements to execute a specified function, or return a value. Multiple controls can be used, each one with different specifications and (dis)advantages. A flow control executes a piece of code when a statement returns true. That means that if you don't want to execute a part of the code you'll have to make a variable (most likely a Boolean) or a statement (comparison) that returns true or false, depending on what you want.

Example:

```
mystatement = true
myexecutable = false
myvar = 5

function on.paint(gc)
  if mystatement then
    --code here
  elseif not myexecutable then
    --code here
  elseif myvar == 10 then
    --code here
  End
end
```

14

As you can see in the example, you don't have to use ==true or ==false in statements. The Boolean variables here already contain true or false, so that's what they will return. mystatement returns true, and myexecutable returns false. Notice that in the example we can see elseif **not** myexecutable. This means that the opposite value is returned. So myexecutable returns false, but thanks to the not, it returns true. The original value is not affected by this operation, myexecutable stays false.

```
If..then..[else]..end
For i=1,10[,step] do..end
Repeat..until..
While..do..end
```

Although the use is very simple, and you'll use it continuous when writing programs, there are a few things to consider. When only using a statement like in the example, you must be sure that the variables are Booleans (true or false). Otherwise the code won't be executed, since a number variable returns a number, and not a Boolean. We can easily add things seen in the math chapter (p 12) 'assignment and comparison'. For this statements, we need more variables or fixed values. In the margin of this page you see the possible controls, with '..' standing for code or statement.

Tutorials

First program: cube

In this first tutorial, we're going to make a game in which you have to avoid a block, guided by the AI, by moving it with the arrow keys. First, we need to initialize the variables when the program start. We do not necessarily have to place it within the `on.create()` function, but it's good to keep an overview, and if we want to reset the whole game, we just call `on.create()`. The values we declare are `x` and `y`, the position of our block. This game is originally written by epic7.

```
function on.create()  
    x, y = 0, 0  
end
```

Now we want to draw our block on the screen, on our own position. We do this by calling `gc:drawRect` within the newly created `on.paint(gc)` function.

```
function on.paint(gc)  
    gc:drawRect(x,y,30,30)  
end
```

If you'll try this program, you'll see a black cube (grey in the emulator) that does nothing, now let's add some movements. The goal of the game is to avoid an AI guided block, so we want to move ours with the arrow keys:

```
function on.arrowKey(arrow)  
    if arrow=='up' and y>9 then          -- we set limits to the y so it stays in the  
field                                     field  
        y = y - 8                        -- this makes the block go upwards  
    elseif arrow=='down' and y<170 then  
        y = y + 8                        -- this makes the block go downwards  
    elseif arrow=='left' and x>9 then  
        x = x - 8                        -- this makes the block go left  
    elseif arrow=='right' and x<280 then  
        x = x + 8                        -- this makes the block go right  
    end  
    platform.window:invalidate()         -- we call this to be sure the screen getsend
```

We want it to look flashy, so every time `on.paint(gc)` gets called, we want to give the block a different color. We do this by using `gc:setColorRGB(Red,Green,Blue)`. Since it has to be different, and we don't want to keep a table with possible colors, we just use the random function to make a new color every loop, so add this to the `on.paint(gc)`, before the `drawRect`.

```
gc:setColorRGB(math.random(0,255),math.random(0,255),math.random(0,255))
```

This function chooses for every color (red, green or blue) a random number between 0 and 255. Those are the limits of the numbers. We call this every loop, so the color will be changed continuously.

Now let's add the AI. This might sound difficult, but it surely isn't. First of all, we want to give it speed. We can do this by making it do something every loop, but until now, the only times the game loops, is when the player presses an arrow key. So we have to find another way: the timer.

By using the timer, we can easily give him a certain speed, while it is till perfectly controllable, and we have a function we can call with it: `on.timer()`. Before we actually can do something, we have to do some mathematics. Because we want the AI to follow you wherever you go, we need to find out how to do this.

First of all, we have to give the AI cube his own variables: `a` and `b`. So the `on.create()` becomes:

```
function on.create()  
  x, y, a, b = 0, 0, 280, 170  
end
```

Now let's take a look at the math. We want the AI to move our way when it's moving, so when the block is left from us, it has to go right, and when it's right it needs to go left, get it?

```
if x<a then      -- a is the x of the AI, x is our x value  
  a=a-2          -- if it is our x is smaller (more left) a has to shrink  
else  
  a=a+2          -- and the other way around of course  
end
```

So here we see that it will move left when we are left and right when we are right AND at the same x-value. This means that it will move right when at the same x, which is weird of course, let's change that:

```
if x<a then  
  a=a-2  
elseif x>a then      -- now it will only move when bigger, no longer when the  
same                 same  
  a=a+2  
end
```

now the same for the y value:

```
if y<b then  
  b=b-2  
elseif y>b then  
  b=b+2  
end
```


We want this code to be executed within an specified amount of time, so we add the on.timer function and the timer.start in on.create().

```
function on.create()
    timer.start(0.1)
    x, y, a, b = 0, 0, 280, 170
end

function on.timer()
    if x<a then
        a=a-2
    elseif x>a then
        a=a+2
    end
    if y<b then
        b=b-2
    elseif y>b then
        b=b+2
    end
    platform.window:invalidate()    -- don't forget this! It will update the
screen                                -- every loop, otherwise nothing will happen
end
```

Now we added the movement of the AI, we're able to test it! Use the Nspire scripting tools which you can find on <http://education.ti.com/calculators/downloads/US/Software/Detail?id=6840> . Just save your file to a directory of your choice, open the TI scripting tool, and click the 2nd A. This pastes the script to the clipboard in a understandable manner for the software. Now open the TI student software, set the view on calculator, close the current opened file, open a new one by clicking new document, and just do ctrl-v. Now the program should be running in the emulator.

Now let's add the most important piece of code: checking when you die.

Again, this is simple. We only have to look if there's a collision between our block and the one from the AI. We know that both our blocks are 30 px, and we know both the x and y values. So, when do we die?

Hitting the other block means a part of the drawing is overlapping (or at least the pixel next to it). So we could check if the pixels around our block are black, like we could do in TI 83/84+ basic, asm and asm. This isn't possible in Lua. We have to rely on what we know.

We know the x values, so when we calculate the difference and the result is less than 30, we know that the cubes are intersecting on the x-axis. This does not yet mean they are totally intersecting, they just have one or more common x-values. To be able to check this in one time, we have to take the absolute value of this subtraction. Now again the same with the y values. That gives this code:

```
if math.abs(x-a)<=30 and math.abs(y-b)<=30 then
  --[[some code here]]
end
```

on the place of the 'some code here' we want to make something happen so we can check later if he's dead or not. Let's take the Boolean variable alive. At the beginning it gets the value true, and when the cubes intersect, we change that to false. Add this to on.create() and add a statement to on.paint(gc) for conditional drawing. Now the whole program should look like the one on the next page.

Now, the task's up to you to add a pause function. You can use any key you want, but keep it useful. Example: you could use the on.charIn command and check if 'p' has pressed. Or escape, since that's close to the arrows... your choice. Only keep in mind that we have Boolean variables to use. Those variables have state changers like **not**, use them!

```

function on.create()
    timer.start(0.1)
    x, y, a, b, alive = 0, 0, 280, 170, false
end
function on.timer()
    if x<a then
        a=a-2
    elseif x>a then
        a=a+2
    end
    if y<b then
        b=b-2
    elseif y>b then
        b=b+2
    end
    if math.abs(x-a)<=30 and math.abs(y-b)<=30 then
        alive=false
    end
    platform.window:invalidate()
end
function on.paint(gc)
    gc:setColorRGB(0,0,0)
    if alive then
        gc:setColorRGB(math.random(0,255),math.random(0,255),math.random(0,255))
        gc:drawRect(x,y,30,30)
        gc:fillRect(a,b,30,30)
    else
        gc:drawString("You died!",0,0,"top")
    end
end
function on.arrowKey(arrow)
    if arrow=='up' and y>9 then
        y = y - 8
    elseif arrow=='down' and y<170 then
        y = y + 8
    elseif arrow=='left' and x>9 then
        x = x - 8
    elseif arrow=='right' and x<280 then
        x = x + 8
    end
    platform.window:invalidate()
end

```