A short introduction to EGSL

Easy **G**ame **S**cripting **w**ith **L**ua

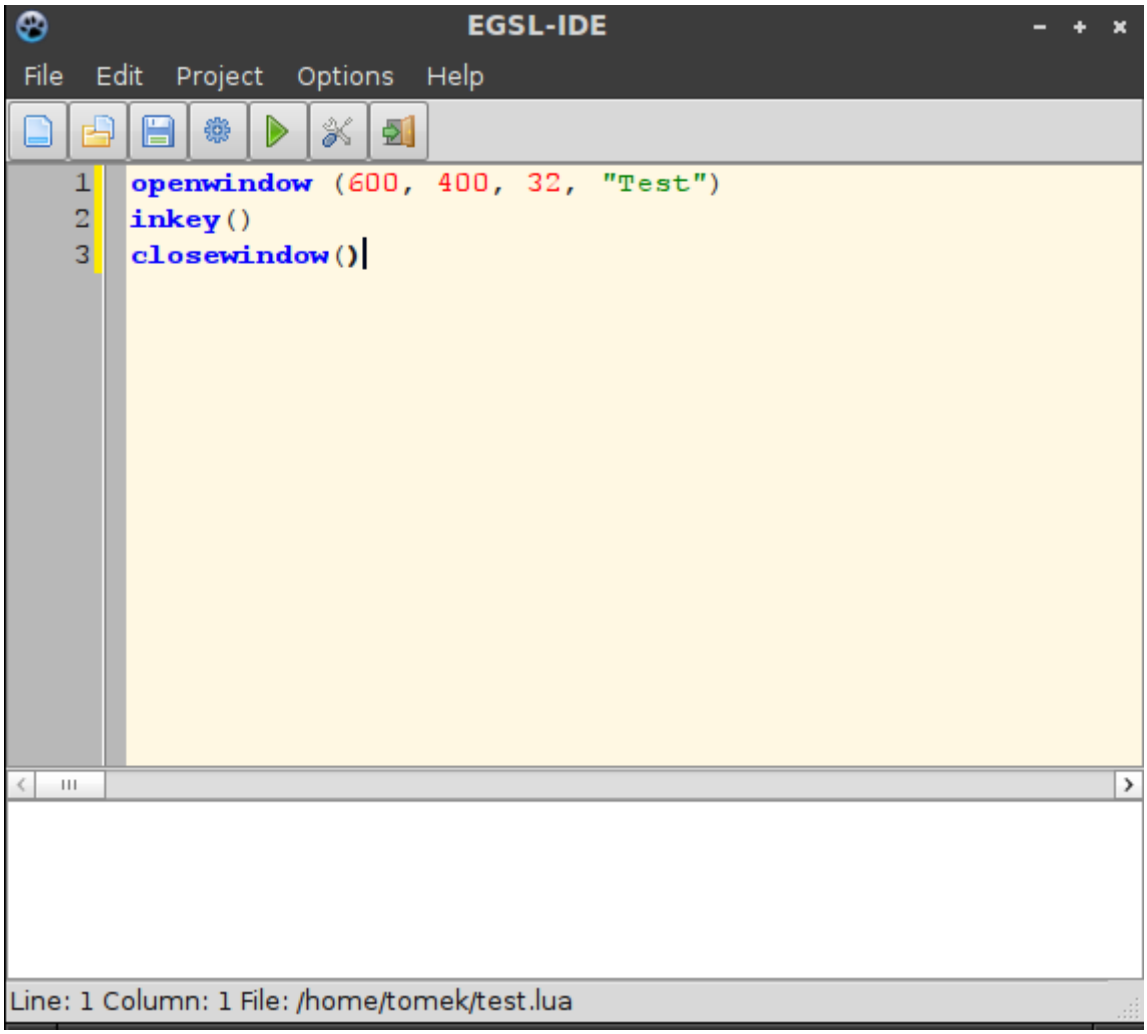
What is EGSL?

EGSL stands for ***E**asy **G**ame **S**cripting with **L**ua*. It's an easy to use interpreter designed for creating 2D retro games. It's been written in Pascal, but the language it's based on is Lua – small, fast and easy to learn scripting language that is often used in game industry. Basically, EGSL can do everything Lua can and even more, as many graphics and sound functions has been implemented in it. The engine is based on SDL. EGSL is multi platform. Versions for Windows, Linux, Max OS and Haiku are available. The interpreter is free and open source software.

To get EGSL, go to *http://www.egsl.retrogamecoding.org//pages/downloads.php* and download a version suitable for your system. EGSL comes with simple editor that let you

write first programs straight after installing it. There is no need for any configuration, set-up etc.
To run your first program you need to open EGSL editor, type and save your code and press **Execute Script** button. The simplest example written in EGSL looks like that:

```
openwindow(600, 400, 32, "Test")
inkey()
closewindow()
```



When you run this code, the interpreter will open a new graphic window. The size of the window is set to 600x400px, its title to „Test". The fourth parameter in **openwindow()** command set the graphic mode to 32 bit. It can be changed to different value, but for now we will leave it like that. The program opens a new window, then waits for any key to be pressed (responsible for that is **inkey()** in the second line) and when it happens, closes the window (what is done by **closewindow()** function). Not much, but this will be a start point we will use in almost every example.

Drawing

In EGSL many commands that let you draw on the window has been implemented. Ten of the most important ones are described below.

dot(x, y) – draws a dot at given coordination

line(x1, y1, x2, y2) – draws a line from the point (x1, y1) to the point (x2, y2)

box(x1, y1, x2, y2) – draws a rectangle where the point (x1, y1) represents upper left corner of the rectangle and the point (x2, y2) represents its lower right corner.

fillbox(x1, y1, x2, y2) – the same as box, but draws a filled rectangle

circle(x, y, radius) - draws a circle with given radius where the point (x, y) represents the centre of the circle

fillcircle(x, y, radius) – the same as circle, but draws a filled circle

colour(r, g, b) – sets the colour for drawing in RGB format

backcolour (r, g, b) – sets the colour for the background in RGB format

clearscreen() - clears the screen with the current background colour

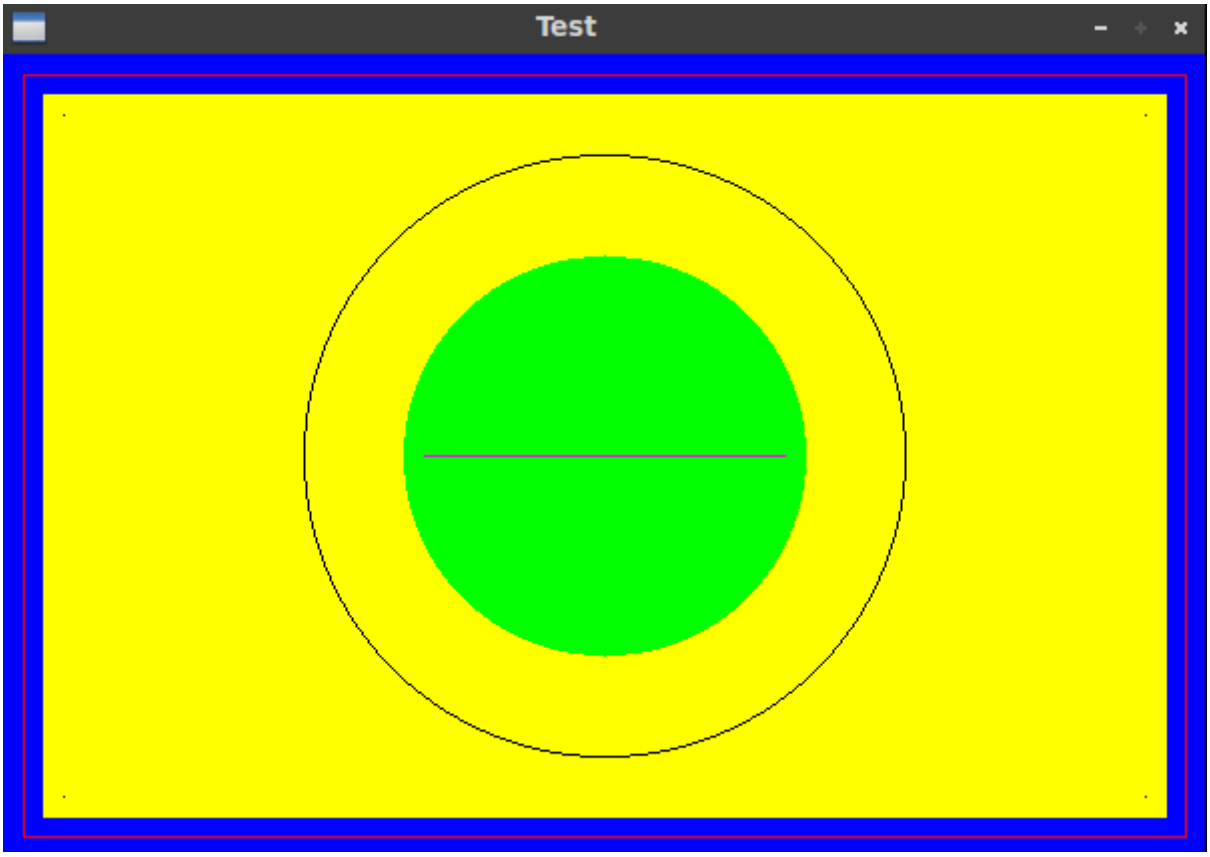
redraw() - this command may be a bit complicated for beginners. When you use drawing commands the computer remembers what you has drawn, but don't place it on the window automatically. To place your drawings on the window you must use **redraw()**. This is very useful feature. It let programmer to do smooth animations and makes drawing much faster.

Here as an example program that uses all commands explained above:

```
openwindow(600, 400, 32, "Test")
backcolour(0, 0, 255)
clearscreen()
colour(255, 0, 0)
box(10, 10, 590, 390)
colour(255, 255, 0)
fillbox(20, 20, 580, 380)
colour(0, 0, 0)
```

```
circle(300, 200,  150)
colour(0, 255, 0)
fillcircle(300, 200,  100)
colour(255, 0, 255)
line(210, 200, 390, 200)
colour(0, 0, 0)
dot(30, 30)
dot(570, 30)
dot(30, 370)
dot(570, 370)
redraw()
inkey()
closewindow()
```

When you run this example, you should see something like that:



The example is so simple that can be, hopefully, understand without any further explanation.

Images

Another great thing about EGSL is that it let you to use external graphics very easily. The interpreter supports BMP, JPEG, JPEG2000, PNG/APNG, MNG, JNG, GIF, TGA, DDS, PBM, PGM, PPM, PAM, PFM, TIFF, PSD,PCX and XPM files as long as given images are in RGB formats. To display graphics on the window you need to use two EGSL commands.

loadimage(filename) – this command loads an image from the file and keep it in memory. You can choose a name for the image:

```
your_name = loadimage(filename)
```

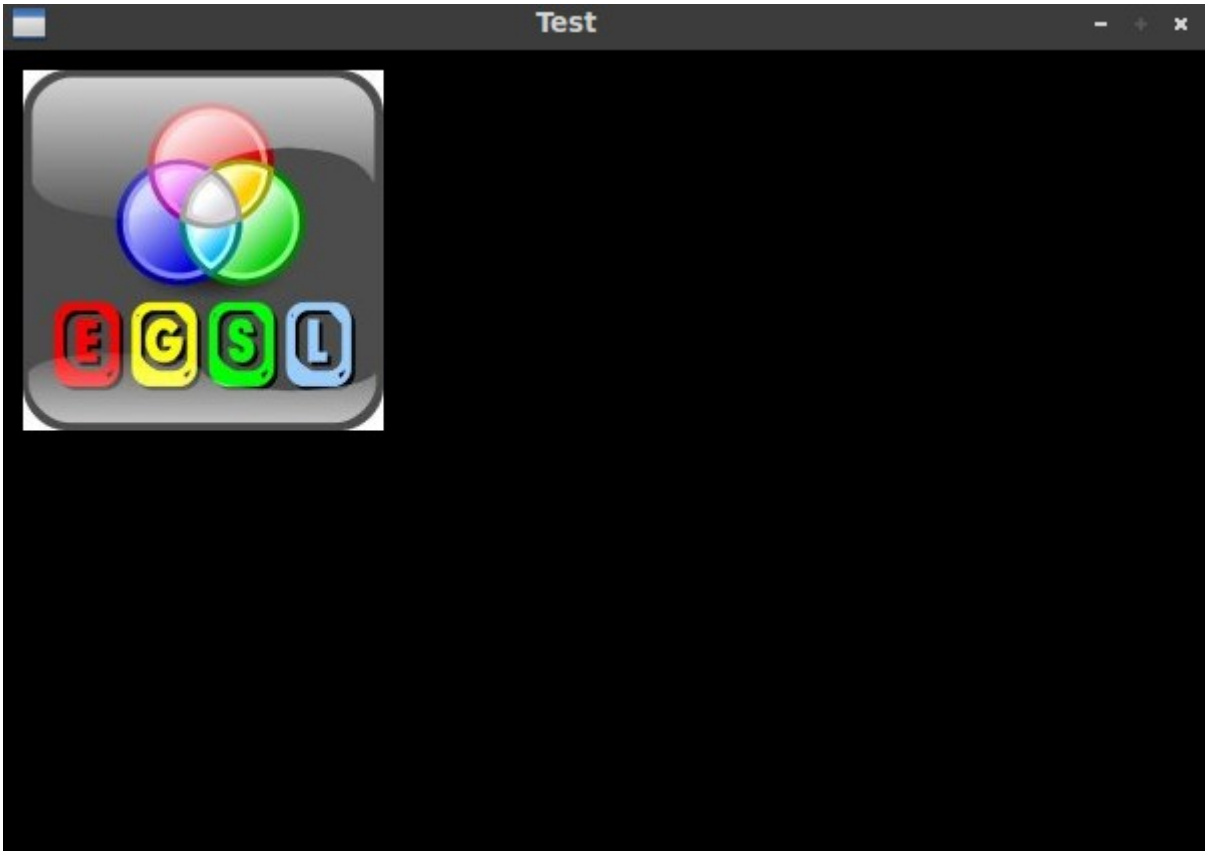
By default, the directory where your script is placed is used, so if you place the graphic file in the same directory you can load it simply by giving its name. If you want to load an image from other directory you need to give a full path to it.

putimage(x, y, your_name) – draws a loaded image onto the screen where (x, y) represents a position of the left upper corner of the image.

After using **putimage()** you need to use **redraw()**, exactly the same way like with drawing commands. If you won't do this, the image will be kept in the memory, but it won't be placed on the window. Here is a simple example. As a graphic file I'm using EGSL icon, but you can use any image as long as it's in proper format and has a proper size (should be smaller than a window). Remember to place your file in the same directory your script is saved or to give full path to your file.

```
openwindow(600, 400, 32, "Test")
first_image = loadimage("egsl.jpg")
putimage(10, 10, first_image)
redraw()
inkey()
closewindow()
```

After running this example you should see something like this (of course, instead of EGSL icon you will see a image you've used).



In EGSL many functions for transforming images (scaling, rotating, making transparent) have been implemented. You can find more information about them on EGSL website - *http://egsl.retrogamecoding.org/* .

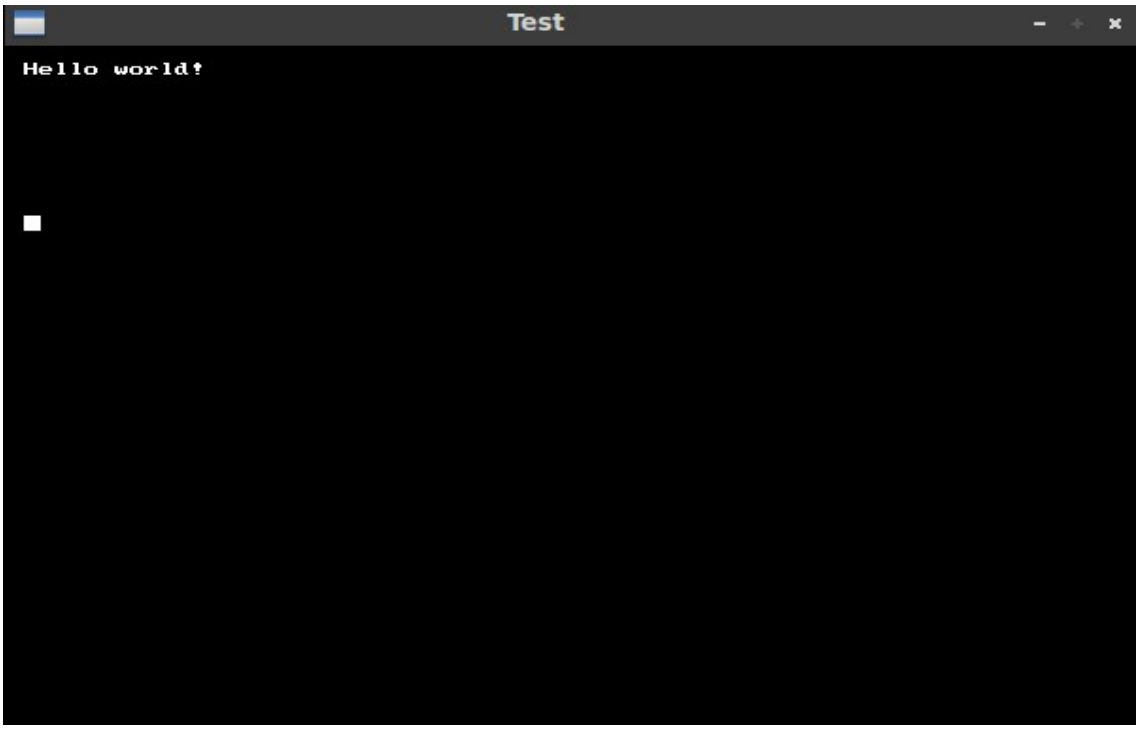
If you have any difficulties with installing or using EGSL, please visit EGSL section on *http://forum.retrogamecoding.org/*.

What is Lua?

All the previous examples were very simple. With that set of commands it's impossible to write a real program. To do that computer must make decisions, repeat some commands in loops or change the parameters of drawing etc. For that we need a programming language. And here comes Lua.

Lua is an interpreted programming language. It's free, multi-platform, lightweight, fast and easy to learn. It is widely used by game programmers. It shouldn't take more than 10 min. to write your first program in Lua, but before you do that I will introduce one useful EGSL command that let us to write a classic “Hello World” example.

drawtext(x, y, text) - writes a text to the graphic window at (x, y) position




```
openwindow (600, 400, 32, "Test")
drawtext(10, 10, "Hello world!")
redraw()
```

```
inkey()
closewindow()
```

Data types and values

Lua is a dynamically typed language. It means that you don't have to worry about data declarations or conversions. There are eight basic types of data in Lua, but from our point of view the most important are for of them: number, string, boolean and nil.

Numbers

The number type represents any kind of numbers, for example: 5; 567.65; 0.0002; 125000. Lua doesn't have a special type for integers numbers. All of numbers are real with double precision floating point. It simply means that you don't have to worry about what numbers you are using at the moment. Lua will manage it for you.

String

String type represents sequences of characters: single characters, words etc. Strings are in single or double quotes, for example: „t”, 'home', „My name is John”. String can contain a single character or a book. Lua doesn't have problems with handling a very long strings.

Boolean

The boolean type has only two values: true or false. It helps to make decisions.

Nil

Nil is a type that has only one value – nil. It is used mostly with variables. In Lua all variables that weren't assigned any other value by default has a value nil.

Operators

Arithmetic operators

'**+**' - addition

'**-**' - subtraction

'*****' - multiplication

'**/**' - division

'**^**' - exponentiation

'**%**' - modulo

Logical operators

Lua supports the usual logical operators: **and**, **or** and **not**.

Relational operators

'**==**' - equal

'**<**' - less than

'**>**' - more than

'**<=**' - less than or equal

'**>=**' - more than or equal

'**~=**' - not equal

String operators

'**..**' - concatenation operator

Variables

Variables are places that can store values. Values of variables can change during the execution of the program. In Lua variables don't need to be declared before they are used. There is also no need to define types of variables. Names of variables can be any string of letters, digits and underscores not beginning with a digit. To assign a value to the variable, you need to use assignment operator ('**=**'). Examples:

```
number = 127
name = 'John'
second_number = 0.345
name2 = „Paul”
```

To change variable value you need to assign a new value to it. For example:

```
number = 5
name = 'Marcus'
second_number = number + 8
name2 = name2 .. „ Smith”
```

To discard a variable you need to change its value to **nil**. This is the only way to discard a **global** variable. Global variable is a variable that is visible anywhere in the program. There are also **local** variables, that are visible only inside the block they were given values (for example – function). Local variables are discarded automatically when the program

execution leaves the block they were used in. By default all variables in Lua are global, however it is possible to declare local variables by using keyword **local**. It's a good practice to use local variables when possible.

Lua provides automatic conversion between a string and number values. Any arithmetic operation tries to convert string to a number, while when string is expected Lua tries to convert a number to the string. There are also two built-in functions that convert between a string and number values.

tostring(number) – converts number to string

tonumber(string) – converts string to number

It's a good idea to use those function when possible to avoid wrong automatic conversion. Lua is case sensitive. **Number** and **NumbeR** are two different variables.

Arrays

In fact, there are no arrays in Lua. In Lua there are tables that are more complex data type, but it is possible to use them as arrays. By using a numerical key, the table resembles an array data type.

An array is a group of indexed values. For example:

```
first_array = {1, 20, 300, 41, 58, 62}
second_array = {'cat', 'dog', 'horse', 'monkey'}
```

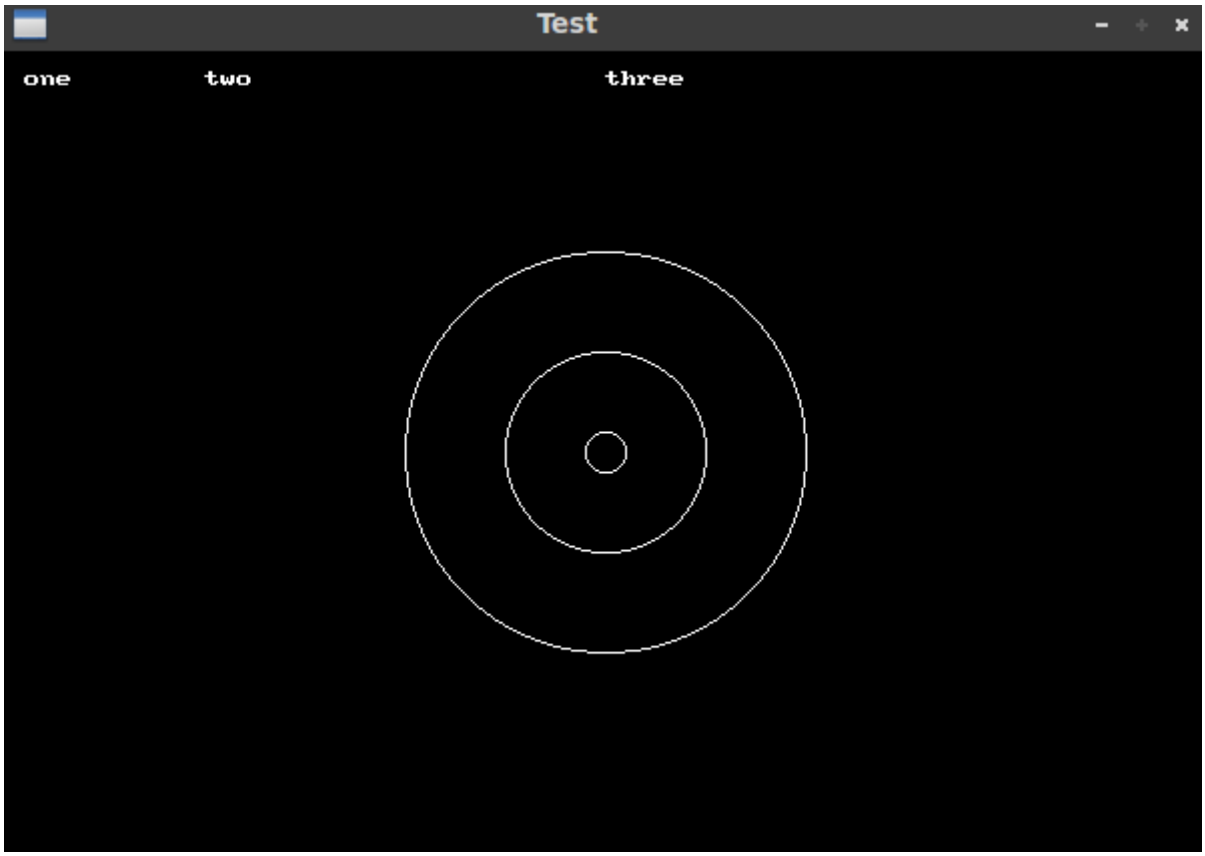
To access an element of the array, you need to give its index. Lua arrays, unlike as other languages, are 1-based. The first index is 1 rather than 0. So, how does it work?

To get the first element of **first_array**, you need to use command `first_array[1]`, while `second_array[3]` will return a string 'horse'. To change element of the array you need to assign a new value to it. For example `second_array[3] = 'bird'` will replace 'horse' with 'bird'.

Below you will find an example that uses arrays and variables:

```
openwindow(600, 400, 32, "Test")
first_array = {10, 50, 100}
second_array = {"one", "two", "three"}
variable1 =  first_array[1]
variable2 =  first_array[2]
variable3 =  first_array[3]
circle(300, 200, variable1)
circle(300, 200, variable2)
circle(300, 200, variable3)
variable4 = second_array[1]
drawtext(10, 10, variable4)
```

```
variable4 = second_array[2]
drawtext(100, 10, variable4)
variable4 = second_array[3]
drawtext(300, 10, variable4)
redraw()
inkey()
closewindow()
```



Loops

In almost every computer program commands are executed thousands or millions times. Of course, it doesn't mean that every program has a million lines. It's achieved by looping, in other words – executing the same lines and block of code many times.

In Lua there are three kinds of loops.

For
This is a very traditional loop that can be found in many old programming languages like C or BASIC. It is great if we know how many times we need to repeat our code.

```
openwindow(600, 400, 32, "Test")
for x = 1, 1000 do
    clearscreen()
    drawtext(10, 10, x)
    redraw()
```

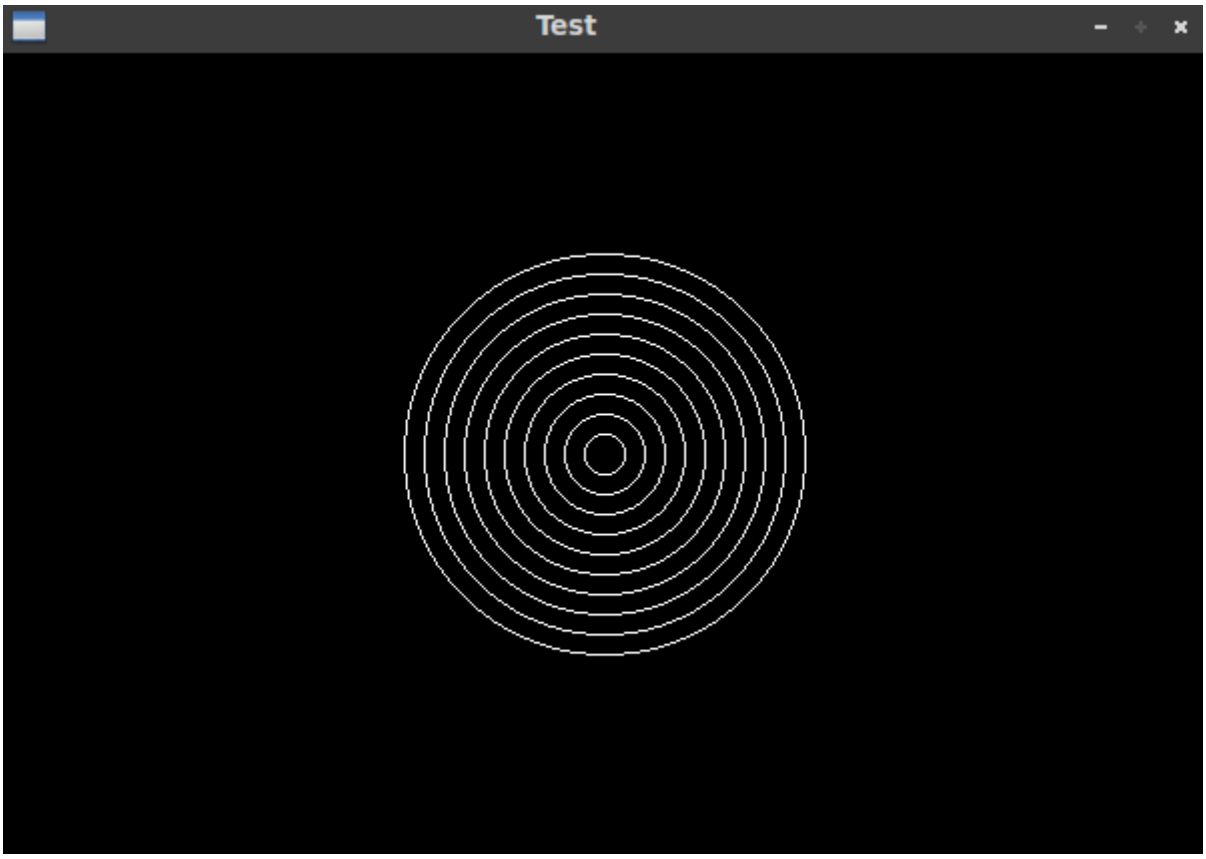
```
end
inkey()
closewindow()
```

This simple example demonstrates how **for** loop works. The code between **for** and **end** is executed 1000 times (every single time it writes the number on the screen that represents an actual value of the variable x).

It is possible to add a third parameter that will set a step for counting.

```
openwindow(600, 400, 32, "Test")
for x = 10, 100, 10 do
    circle(300, 200, x)
    redraw()
end
inkey()
closewindow()
```

This time code inside a **for/end** block is executed only 10 times.



While
Another very popular and widely used loops. It can be used instead of **for** loop, but also in

the situation when we don't know how many times a block of the code must be executed.

```
openwindow(600, 400, 32, "Test")
x = 10
while x <= 100 do
    circle(300, 200, x)
    redraw()
    x = x + 10
end
inkey()
closewindow()
```

When you run this example, you will see exactly the same result. In this case Lua check if the condition is true (if x <= 0) and if it is execute the code inside a **while/end** block. When a value of x becomes 110 the condition return false and the program execution leaves **while/end** block.

Another example of using **while** loop. This time the program is looping until it gets all elements of an array. When a value of x becomes 11 the condition return false (there is only 10 elements in an array, so a[11] return nil what is treated by Lua as false). The results is the same like in the example above.

```
openwindow(600, 400, 32, "Test")
a = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
x = 1
while a[x] do
    circle(300, 200, a[x])
    redraw()
    x = x + 1
end
inkey()
closewindow()
```

Repeat/until

This loop is very similar to the **while** loop. The most important difference is that the condition is checked after executing the code, so unlike with **while** loop, the code is executed at least one time.

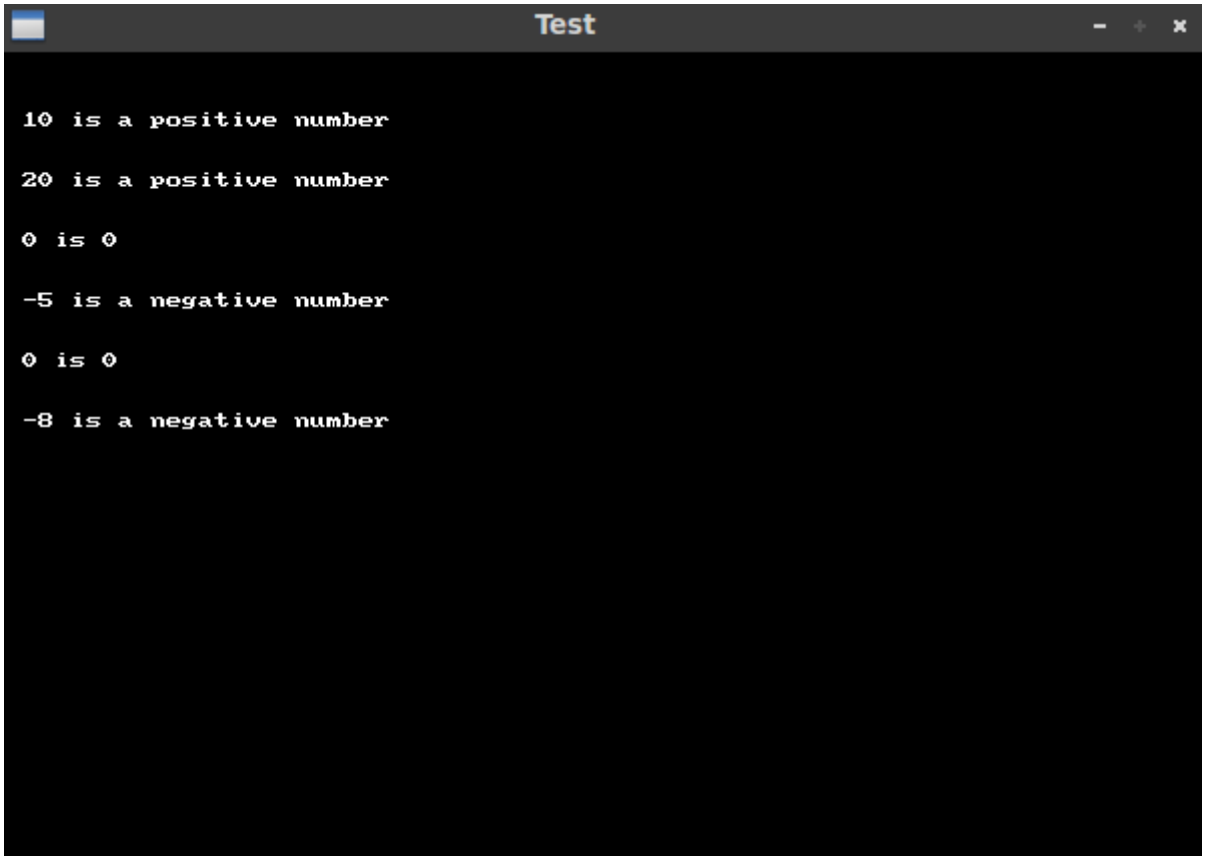
```
openwindow(600, 400, 32, "Test")
x = 10
repeat
    circle(300, 200, x)
    redraw()
    x = x + 10
until x > 100
inkey()
closewindow()
```

As you can see this example produce exactly the same graphic result.

Making decisions

This is another very important aspect of programming. Without making decision there would be not possible to write any useful program. How it works? The program tests a given condition and depends on a result makes a decision. To do this you need to use **if/elseif/else** block.

```
openwindow(600, 400, 32, "Test")
a = {10, 20, 0, -5, 0, -8}
x = 1
while a[x] do
    if a[x] > 0 then
        drawtext(10, x * 30, a[x] .. " is a positive number")
    elseif a[x] < 0  then
        drawtext(10, x * 30, a[x] .. " is a negative number")
    else
        drawtext(10, x * 30, a[x] .. " is 0")
    end
    redraw()
    x = x + 1
end
inkey()
closewindow()
```



How does it work? It's very simple. The program checks all elements of the array. If the first condition (a[x] > 0) is true and element of the array is a positive number the code inside **if** block is executed and **elseif** and **else** blocks are ignored. If the first condition is false, the code inside **if** block is ignored and the second condition (a[x] < 0) is tested. If it's true (element of an array is a negative number) the code inside **elseif** block is executed and **else** block is ignored. If the second condition is false, the code inside **elseif** block is ignored and block **else** is executed.

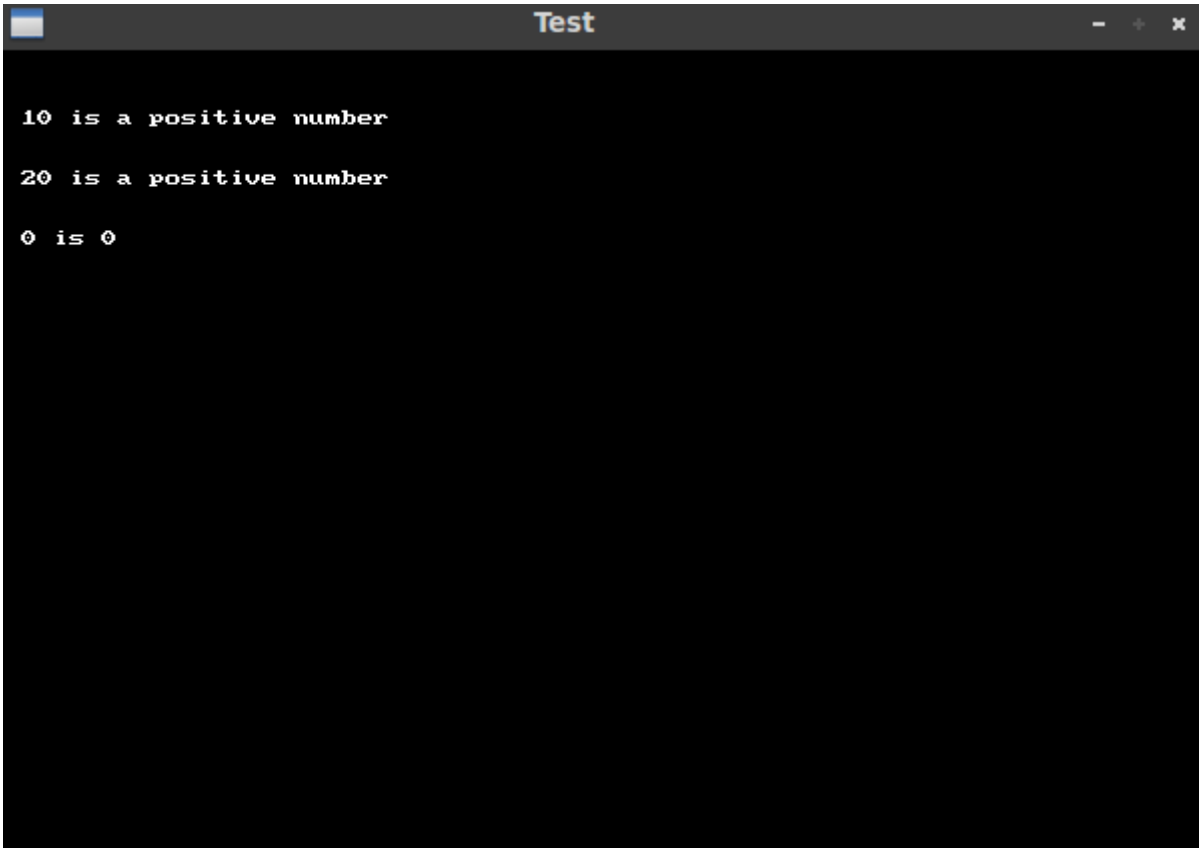
Now I can introduce another useful command – **break**. This command is used to leave a loop at any time, even if by definition looping should be still continued. Let's make a small change in a previous example.

```
openwindow(600, 400, 32, "Test")
a = {10, 20, 0, -5, 0, -8}
x = 1

while a[x] do
    if a[x] > 0 then
        drawtext(10, x * 30, a[x] .. " is a positive number")
    elseif a[x] < 0  then
        break
    else
        drawtext(10, x * 30, a[x] .. " is 0")
    end
    redraw()
    x = x + 1
end

inkey()
closewindow()
```

As you can see, this time when the second condition was true, the **break** command was executed, the program left the **while** block and rest of the array elements weren't checked.



Functions

Functions are defined by programmer blocks of code that can be called at any time with some parameters. Functions perform some action and are also able to return a value. Here is a simple example:

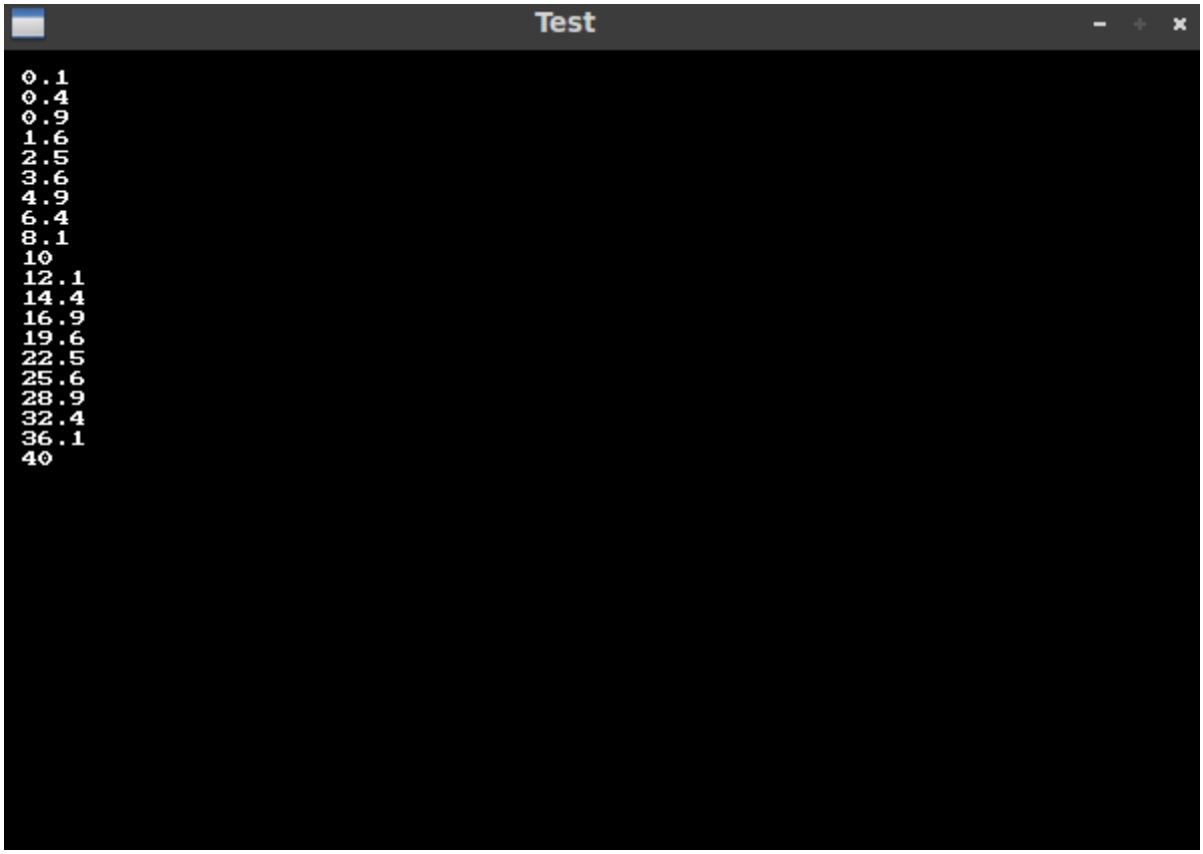
```
openwindow(600, 400, 32, "Test")

function calculate(number)
     new_number = (number ^ 2) / 10
     return new_number
end

for x = 1, 20 do
    drawtext(10, x*10, calculate(x))
end

redraw()
inkey()
```

```
closewindow()
```



In this program we define a function **calculate()** that rise the number to the power of 2, reduce it by ten and then return the result of the calculation that is in main program printed on the screen.

To find more information about programming in Lua, please visit: http://www.lua.org/manual/5.2/ and http://www.lua.org/pil/ . Of course you can always ask a question on http://forum.retrogamecoding.org/ .