

Sphere Pages

By Dn7 | Sphere documentation project

alpha

For Sphere 0.97.

Index

Chapter	Ex.	By	Page:
1	Introduction	Dn7	
2	About sphere	Dn7	
	2.1 Windows	'	
	2.2 Linux	'	
	2.3 Technical briefings	'	
3	The Editor briefly	Dn7	
	3.1 Image Editor	'	
	3.2 Sprite Editor	'	
	3.3 Map Editor	'	
	3.4 Script Editor	'	
	3.5 Other stuff	'	
4.	E.C.M.A Script	Dn7	
	4.1 Javascript in Sphere	'	
	4.2 Variables	'	
	4.3 Functions	'	
	4.4 Objects	'	
	4.5 Conditions (forks)	'	
	4.6 Structure	'	
	4.61 Knoty Knott	'	
	4.62 Comments	'	
	4.63 Position matters	'	
	4.64 The anti-knot	'	
4b	Advancing	Dn7	
	4b.1 Returning values	'	
	4b.2 Objects	'	
	4b.21 Functions in objects	'	
	4b.3 Arrays	'	
5.	Combining forces	'	
	5.1 Sphere specifics	'	
	5.2 Understanding the reference guide	'	
6.	First steps	Dn7	
	6.1 Designing code	'	
	6.2 Pseudo	'	
	6.3 The holy cookie	'	
7.	More scripting!	Dn7	
	7.1 Basic scripting tutorial	Eug.	
	7.2 Basic scripting tutorial part 2	Flik.	
	7.3 Control structures, loops etc	Eug.	
	7.4 Arrays	Eug.*	
8.	Game making	Dn7	
	8.1 Classics	Dn7	
	8.2 Jobs & Roles	Eug.*	
	8.3 Story development	Y	
9.	By hearth	Dn7	
	9.1 Revision	Dn7	
	9.11 Basics	Flik	
	9.12 Not-so-basics	Flik	
10.	The map editor	Dn7	
	10.1 Tile tab	Dn7	
	10.2 Map tab	Dn7	

11.		The sprites editor	Dn7
	11.1	Global	Dn7
	11.2	Frames tab	Dn7
	11.3	Edit tab ** (10.1)	Dn7
	11.4	Notes	
A.		Sphere functions reference	Riz*
B.		FAQ	Eug.
	I	Setup sphere	Eug.
C.		Sources	Var
D.		Compendium	Dn7

* = Some editing done

** = Redirection

1. Introduction

Thank you for using my guide for learning sphere, first of all, I want to tell you that I made this guide because of the lack of a complete document about sphere. I also noticed most of the documents are hard for newbies to understand, or were too detailed. Also, sphere is a professional product, it is very descent and very capable, thus needs good documentation. This guide is not a complete reference to sphere, it is as an introduction to how to use sphere. Most people download sphere and start of making a map and try to figure out how to actually use it (that is what I think).

Why use Sphere?

Sphere is meant for making those cool RPGs, like Final Fantasy 6, or Chrono Trigger. You can also use sphere for other things, actually, you can make anything. I recommend you only use sphere for scroller games.

As I said, sphere is capable of making games like Final Fantasy 6, this sounds very attractive, I know, but I don't think you will be able to do so. Making a RPG requires a team, and lots of patience. I know you can make a so-called 'rip off', but it is half the fun. Use sphere when you are sure you are going to make something!

Why is Sphere free?

Sphere is Open Source, and is located at an open source site called source forge. Open Source means that the code you write is accessible for everyone, and anyone can use it, or help you. You cannot use open source projects for commercial ends without asking the author, this is because of the GNU license (for protecting open source projects).

Sphere VS Speed

Sphere itself is fast; games made with it most times can run from PII 300+. But making a game with sphere takes longer, that is because you have to create everything from a scratch. Most RPGMakers see this as a negative point, but I don't. If you create something from a scratch, you'll learn from it, and it's more flexible. Sphere is at the moment one of the top ten most flexible RPGMakers around (Sphere, IKA etc).

When shouldn't I use sphere?

You should not use sphere when you are sure you are not very capable of engineering stuff like battle engines. You will develop development skills while using sphere, which are good for you. If you are used to programming with c++ or java etc, sphere will be pie for you. JavaScript is one of the easiest scripting languages, and it's highly recommended for beginning programmers.

Personally,

I think sphere is too slow. Yes, I know I said sphere is fast. Sphere IS fast, but it could be WAY faster. An example of this is when making screen transitions. I wanted to use bit wise to make those nice FF7 + screen to battle fades (the odd coloring), and it was SLOW. And, duo I have a fast computer, I think it could be faster yes. Compiled JS would be neat, but also imposable. I can imagine something to compress the interpreting to make it faster.

Contact Information

If you think anything in this guide is wrong and should be corrected, email smederij@home.nl. If you want to add something to this guide, you can also contact this address. This guide is in Alpha stages.

2. About sphere

Sphere for Windows consists of an editor and the engine. With the editor you create your RPG, and the engine reads what you made, and translates it in a game. If you use Linux, you will only have the Engine, there is no port of the editor. You don't actually NEED the editor, but it can be an important tool for rapid development.

What the editor does

A sphere project consists of a certain directory structure, everything has its place, and IE Images are placed in the 'Images' folder. The editor has this root structure implanted, and uses sub-programs to edit them. If you open an image, you will get the image editor etc. With it, you can add, insert, or edit files.

Windows

Windows has a pre-compiled version of the sphere engine and editor. It also has a configuration tool to select which driver you want to use. You can use several drivers, most of them are configurable. There are drivers for windowed mode, full screen, 16 bit, opengl, kreed2xsa1 etc etc.

Unix / Linux

If you are using Linux, you will have to either pull the cvs, or download the latest source tar ball. Note that the 0.97 version has it's autoconf file broken, so pull the CVS! After pulling it, open the readme file, it contains some info on what libraries you will need. There are problems compiling some libraries using an Open Linux box, you can try older versions. Also, an older version of audiere is needed. If you use audiere 1.0.3, you will also need Acoustic, which is included in the sphere projects file list.

The Unix version doesn't work (or does?) on big - endian machines. That means you can't run sphere on P4 etc. Sphere uses the SDL library for multimedia abstraction from your hardware, so it doesn't use any special / configurable drivers.

Technical briefing

Sphere is not done, at the moment its in v0.97. It is very close to completion, but you can count it as complete.

The script engine Sphere uses is Spider Monkey. Spider Monkey is a Java Script interpreter (now called ECMA). Java Script is easy to understand, and write. Web browsers also use Java Script. Java Script is NOT the same as Java, Java is a mature programming language and has a compiler. Java Script is NOT a programming language (...) and does NOT have a compiler. It might look a bit the same, but heck, shit isn't always pooped.

3. The editor



I don't think you will have much problems understanding how the Editor works, you open / make a new project and you are set.

Project Explorer

The project explorer has a directory structure similar to your game project folder. If you want to insert a file in a map, you right click on it, and click insert. You will now see a file explorer, if you want to create a new file, you will have to type a new filename, and click open. If you want to import a pre - made file, just select it. When sphere prompts for inclusion in project, click yes. If you want to remove a file, right click on the file, click remove.

You can double click on a file in the project explorer to edit it again. Remember to always save your file by pressing the save button before testing your game. Always do this, or you will get no result.

Toolbar

The casual stuff like open save... The winamp button is for testing your game.

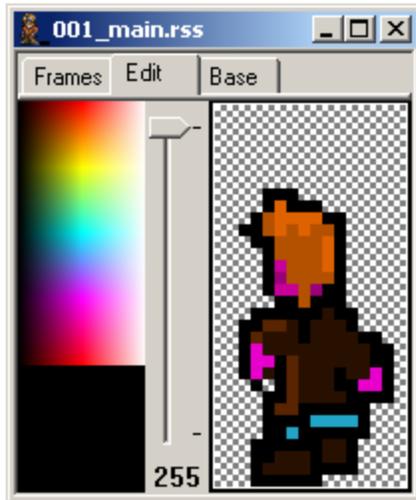
Menu bar

The menu bar has some useful tools; sometimes an extra menu item appears when editing certain files. You can also IMPORT other files like image > tileset. This way, you can import rm2k tilesets by clicking Import > Image -> Tileset, select the file, and enter 16 16. Etc.

Image editor

The image editor is not very good, or, it sucks. You should use software like PSP or IDraw for creating art for your game! DO NOT USE THE IMAGE EDITOR! : D

Sprite editor



The sprite editor is in fact pretty good. It isn't hard to use at all, when you want to insert a new picture, you right click, insert etc. When you want to edit one of the sprites, you click on the desired [FRAME] in the frame window, and click the tab 'edit'. Also note there is now a sprite resize option in the menu bar! There is no fixed npc size for sphere.

Drawing NPC frames

Drawing frames like walking up, left, right, right up etc required time and pixel skill. Click the edit tab to start editing a frame that you selected in the frame window. You can ADD [FRAME] by right clicking (you guessed) and click add frames. You wonder what that scroll bar does? It is the [ALPHA] value, the value of transparency for that pixel. After you did some frames, you will have to make the animation.

Animating the sprite

Click the first tab, the white-bordered 'thing' contains the directions. There are basic directions like 'north', south, east etc. Right click the white border, add, and type the desired direction name. Then, right from the border, right click, add to add frames to the direction animation. Click one of the frames, and then chose a frame from the frame window.

Setting obstructions

Click the obstruction tab, now draw a rectangle at the sprite his feet. The obstruction is for the computer, to calculate when the sprite is obstructed (so it can't move through walls etc).

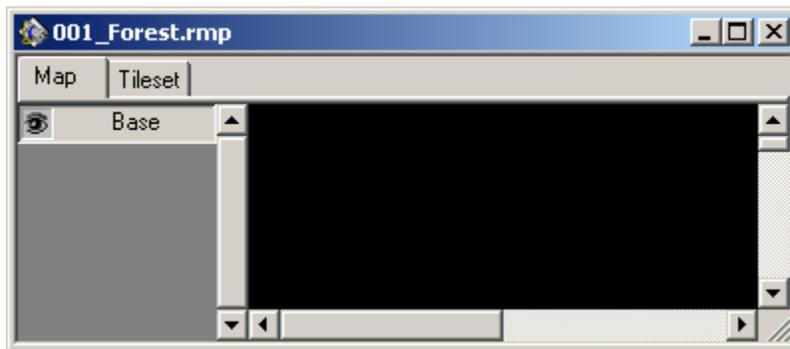
Don't forget to press the save button!

Map Editor

The map editor also includes a tile editor. Sphere saves maps with their Tileset. If you want to make a Tileset, which you want to use for different maps (like a town Tileset), you will have to edit one in an image editor, and import it as a Tileset. Then, you can insert a new map, and select the Tileset from the file list. Using a blank Tileset will pursue you to make your own tiles, or insert them later. Tilesets themselves do not contain obstructions (if its a wall etc); you will have to add them manually each time you make a map. Maybe a nice new feature for v1,0?

Editing tiles.

Right click the tile window > add to add tiles. You can also insert other tilesets and do other self-explaining things. When you inserted some tiles, select one. Then, click the 'Tile Edit' tab. The tile edit map is similar to all the other image editors in sphere. After finishing your tileset, don't forget to save your map, or even your tileset by right clicking it, and save it.



Placing tiles.

Select the desired layer, if you want to add a layer, right click the layer part, and click add. You will see lots of options and buttons and stuff. We won't talk about them yet. Just use the base layer. When you selected a tile, click on a place on the map editor etc etc you know the thing. There is also a toolbox, note that the filler does not fill your map, but it makes rectangles.

Obstructions

To add obstruction to your tiles, you have to right click them, and chose edit obstructions. You can there make a custom obstruction, or use a preset one.

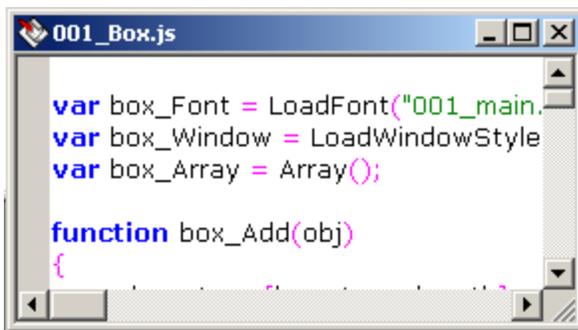
Animations

To use animations in your tiles, you take a note of what tiles should animate. I.e., tiles 10 11 12 and 13 contain a water animation. Right click tile 10, click animate, and fill in next tile = 11, and the desired interval (how long should this tile be used?), I recommend you use something like 45 on each tile. Then you click ok, right click tile 11, edit animation, enter 12 as next, 45 as interval ETC ETC. (At tile 13, you should go to tile 10! If you will go to a tile, which will not animate any further, it will stop the animation.).

Script editor

The script editor is an important asset of sphere. It has syntax highlighting and everything a programmer need: D. At the status bar is a line indent (what line you are editing), and in the menu some new options.

The script editor is just a plain text editor; we will need it later, and is easy to use. Tough, scripting itself is the hardest part.



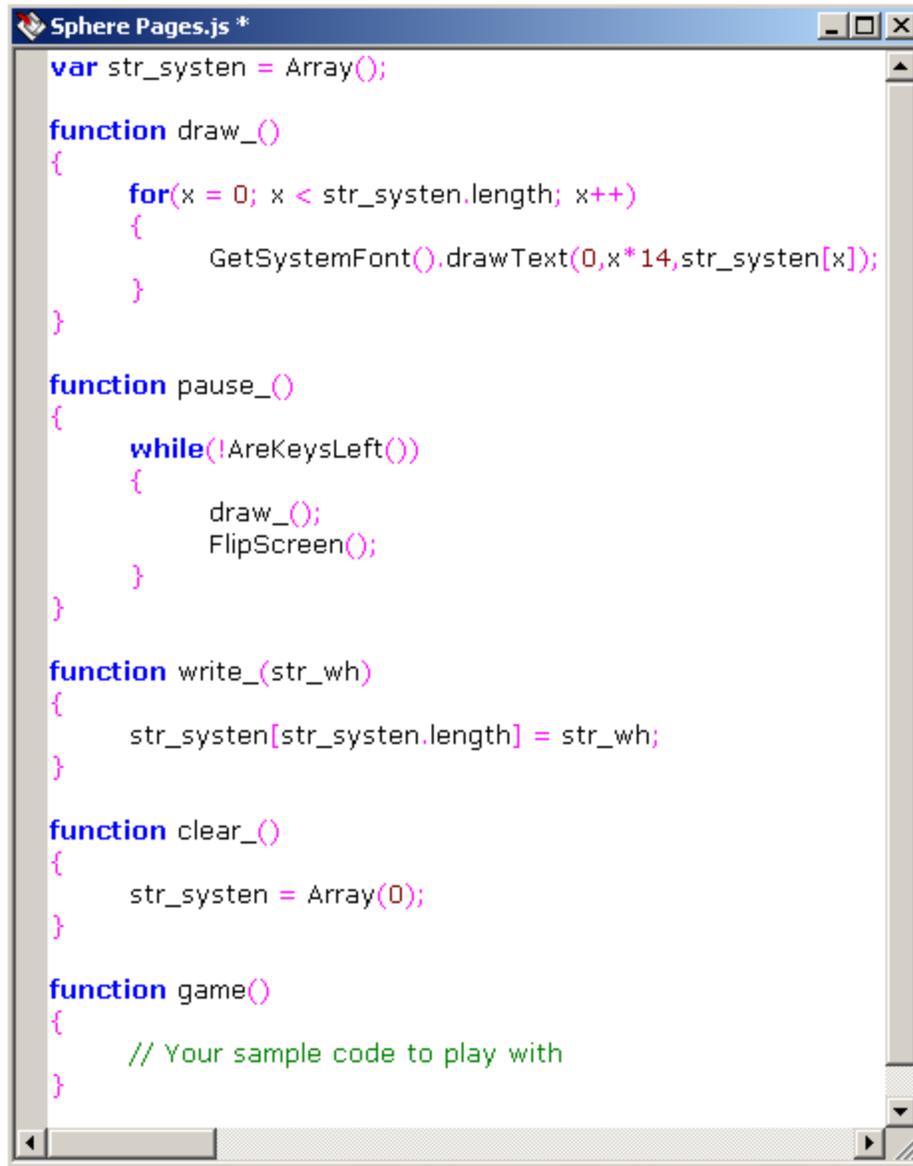
```
var box_Font = LoadFont("001_main.  
var box_Window = LoadWindowStyle  
var box_Array = Array();  
  
function box_Add(obj)  
{
```

Other stuff

I know I missed out ALOT of stuff here, but this are the basics, and you need to know them to use reference guides. Maybe later versions of this guide will provide full support of ALL the capabilities of Sphere.

4. ECMA Script

Welcome to the completely theoretical part of this guide >: D. Well, I don't want to be mean, so we will use both sphere and our known knowledge which we gained, USING a script I wrote, so you can see if you understood it etc. You must put this script in every new game you make, here is it:



```
var str_system = Array();

function draw_()
{
    for(x = 0; x < str_system.length; x++)
    {
        GetSystemFont().drawText(0,x*14,str_system[x]);
    }
}

function pause_()
{
    while(!AreKeysLeft())
    {
        draw_();
        FlipScreen();
    }
}

function write_(str_wh)
{
    str_system[str_system.length] = str_wh;
}

function clear_()
{
    str_system = Array(0);
}

function game()
{
    // Your sample code to play with
}
```

Save this file into 'template.js' into your project (make a new one if required). If you are trying one of the samples, you should place your sample code in 'game' function. Sphere will search for the game function and execute it. You will also have to change your project configuration, set the main script to template.js.

JavaScript in Sphere

JavaScript itself is very bony. The sphere version has some RPG like functions bound with it, including map handling, to make making a game easier. These functions tempt to change in later version; the guide will be updated then...

Keywords

You aren't allowed to use keywords as INT FLOAT BOOL etc, when you use one, it will get the color red!

Writing JS

At the moment, you will place everything in the game function. You should keep some things in mind while writing JS:

Lines end with a ';' (
(...)

Variables

Defending a variable in JavaScript can be done using the keyword 'var'. var is NOT needed, but it makes it easier to read. Variables are used in every rpg maker to store values. A value for instance, that indicated if a chest has been opened or not.

Initializing variables

I recommend you first INITIALATE a variable before you use one. IE:

```
var picked_chest = false;
```

There are different kinds of values variables can have:

Type	Exp	IE
NUMBER:	Numeric value	1242.12
STRING:	Word/sentence between ""	"Hello World"
BOOLEAN:	True or False	true
NULL:	Empty	0
OBJECT:	Variable is an object	font object
FUNCTION:	Variable is a function	makepie();

Setting values

You can set the value of a variable by using varname = something.

Operators

You can use operators to add two or more variables with each other, or just multiply it etc. IE:

$x + y$: Result x plus y
 $x + y$: If its a string, and x would be hello, and y would be world, this results in hello world
 $x - y$: Duh
 $x * y$: Result x times y
 x / y : Result x/y in decimal.
 $x \% y$: Result 'additive' or something from x/y. (not important) (Note by Arek: This returns the remainder from x/y. Can be very useful in some situations.)
 $x++,++x$: Add one to x
 $--x,x--$: Remove one from x
 $-x$: Result negative of x

Declaring

Declaring values can change their values. They are needed to actually change something. IE:

$x = y$: x will become the same as y
 $x += y$: x will be x plus y
 $x -= y$: x will be x minus y
 $x *= y$
 $x /= y$
 $x \% = y$

Conditions / Checking if

You can also check if value somehow makes sense. Like 10 is bigger then 4 makes sense, and thus is true. 4 is bigger then 10 would give false.

$x == y$: True if x equals y
 $x != y$: True if x does not equal y
 $x > y$: True if x is bigger then y
 $x >= y$: True if x is bigger of equals to y
 $x \&\& y$: True if both x and y are true.
 $x \|\| y$: True if one of them is true.
 $!x$: True if x isn't true.

Knots

Knots (I call them knots, dun about the real name :P) are very useful things. Between knots ({ and }) you can put lines of code. Some functions require knots, or most of them anyway. Like the function game() {}, everything between the knots gets executed... You could say everything between a knot is a box with code, which will be run at a certain point.

Functions

Functions are called like: game();. First, the identifier (game). Then the two brackets (), and then the semicolon ;.

Identifier

The identifier is the variable (the name) of the function. If you want to 'call' one, you will have to use functionname();.

Brackets

The brackets are there to tell the functions things, if we had like this function:

```
callname(whatname)
{
}
```

we would call callname this way: callname("Booba!");. Then, the variable 'whatname' of the function 'callname' will equal to "Booba!".

Semicolon

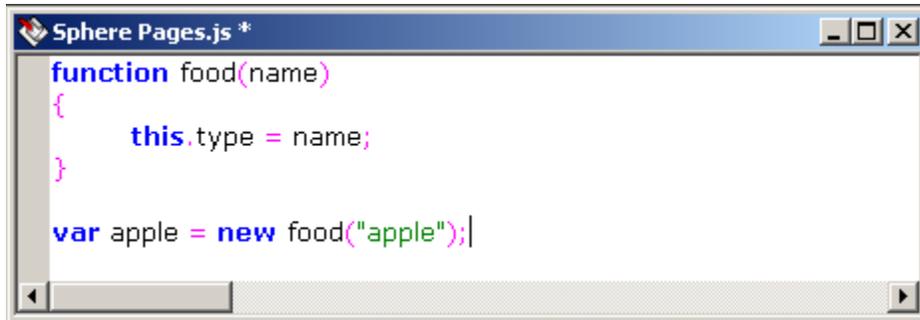
You always put semicolons at end of lines.

Example: (put this between game())

```
write_("Boobaaaa!!!!");
pause_();
```

Objects

Objects are also a special kind of variable. They are defined as functions. To create a new object, you do something like this: (you will have to put this outside of game(!))



```
function food(name)
{
    this.type = name;
}

var apple = new food("apple");
```

Function

Functions are put between knots, and are required for creating objects.

this.

This refers to a non static variable, this.type for instance created a new type for each NEW food made. If you do not use this. you will get a static variable, and will be the same for EVERY food object, instead if only one.

Colon

A colon is used to refer to a variable INSIDE an object. Like, apple.type would give apple, and if you want to change the type of variable apple, you would to apple.type = "froot!". Don't worry about objects yet.

New

New creates a new object from an object template. Here it is the object food.

()

You can give the new object some values just like normal functions, check the brackets explanation.

Conditions

Conditions are used inside knot functions, knot functions (they are NOT called this way, don't try to be smart and tell some older programmer: Hey, I have a knot function :D) are things like if, while, for. IE:

```

if(1==1)
{
    write_("One equals one!");
}
if("a"!="b")
{
    write_("A isn't the same as B!");
}
if(1==2)
{
    write_("One equals two!");
}
pause_();

```

If

The if is one of the knot functions. If the condition is true, if will execute the knot.

(stuff)

Contains the condition.

{ Code }

Code to run.

If also has some nice stuff included, you can also use else after an if knot function. IE

```

if("a" == "b")
{
    write_("A equals B");
}
else
{
    write_("A does not equal B");
}
pause_();

```

else is used when it ISNT true (false).

Loops

Loops are knots, which repeat themselves until a certain condition changes. You can use while, or for. There are some others, but you will only need these two.

While

While will continue the knot until a condition is FALSE (that is NOT TRUE).

ie:

```
var wh = 1;
while(wh < 10)
{
    wh++;
    write_(wh);
}
pause_();
```

For

For is a somewhat more complex knot function. It consist if 3 PARAMETERS. Parameters are things you put between brackets, like bla("Nwa","Something"); bla is the function, the parameter together is "Nwa", "Something", and the first ARGUMENT is "Nwa", and the second is "Something". As you can see, you can add more parameters due to the argument you can use.

IE:

```
write2x_(str1, str2)
{
    write_(str1);
    write_(str2);
}
write2x_("EVIL!", "This is!");
pause_();
```

Back to the for knot function... For used this parameter:

```
For(variable, condition, action){}
```

Variable/For

Here you can initialize a variable, i.e., for(x = 0 ...

Condition For

Here you put a condition which should be TRUE, if it will be false, the knot ends.
IE: for(x = 0; x < 10; ...

; Semicolon

The semicolon is used in for, it's for specific, and makes my head hurt. Anyway, do NOT use a ','.

Action

The stuff in action will be a one-line action that gets executed each time the script reached the end of the knot. IE: `for(x = 0; x < 10; x++){}` would repeat something 10 times, and x would have the value how many times it is already completed.

IE:

```
for(wh = 0; wh < 0; wh++)
{
    write_(wh);
    pause_();
}
```

Structure

Structure is a very important thing in JS, if you do not know it works, you are fucked. There are some rules in structures, which will be very handy dandy to know, DUH.

Knotty knot

Do not make functions in functions

WRONG

```
function pie(){
    function pie3(){
    }
}
```

This is redundant, function pie3 would do as good as in public (aside pie). Do not use the same variable names. Use a tab per knot, if you are two knots deep, which means you had type { two times, use 2 tabs at the beginning of your new line. If you type a } again, you lower on knot level.

Comments

Comments are placed in code to make clear some stuff, you should comment your code well. It is very useful. Commented code or comments will not be executed. You use comments using // or using a special /* */ knot.

```
/* This is comment */
// this will not be executed
```

It might seem useless, but you could place notes in your code if you have to.

Position matters

It does, when you put / declare a variable in a function, the variable is owned by that function, so other functions can't touch it. If you want to avoid this, declare you variable outside (most times at the beginning) the function. If you want the variable to be private (per function, only HE can touch it) you MUST declare it inside the function (between the knots).

Variables are read and defined by the engine from line 1 - ~. Don't do:

WRONG

```
function pri()
{
    write_(myname);
}
var myname = "Booba bybba!";
```

The anti knot

You don't need to use knots if you only require one line of code, you can use the ; instead (the ANTI KNOT! :D). IE:

```
if(1 == 1) write_("One is one.");
```

4b. Advancing

Now that you understand the basic, let's start talking about more advanced things of js.

Returning values

What does returning values mean? Well, actually, it's a function that 'returns' a value. The function 'GetSystemFont()' RETURNS a font object. IE:

```
function reverse(bole)
{
    return !bole;
}
```

```
var bla = true;
bla = reverse(bla);
write_(bla);
pause_();
```

Returning values is a VERY important part of using JS, you can create very complex things with it. Later we will continue using returning values.

Objects

Objects can be anything, they are like boxes containing several things, like functions, variables etc. As we told before, objects require the 'new' word for creating a new one. Variables or functions etc stored inside an object require a '.' to be called when the variable was DECLARED using 'this.'

Functions inside objects.

You can have any type of function inside object, we will not talk about prototypes. You can even have returning functions inside objects. IE:

```
Function oven(hotnes)
{
    this.degree = hotnes;
    this.bake = function bake(what)
    {
        write_("Baked a " + what + " on " + this.degree + " degrees!");
    }
}
```

```
var myoven = new oven(125);
myoven.bake("poop");
pause_();
```

Arrays

Arrays are variables, which contain multiple variables using a certain pattern. You could make a list (one dimension), or a matrix (2d) or even worse... In sphere, you are more likely to only use 1d arrays.

What can I do with it?

You could make an array containing all the days of the weeks and then like the number incorporate with it. It is easier then it sounds...

```
var weekdays = Array(7);  
weekdays[0] = "Monday";  
weekdays[1] = "Tuesday";  
(...)  
var today = 1;  
write_(weekdays[today]);
```

The [] are to identify WHICH part you want of your array. You can even store OBJECTS in arrays, or even complete FUNCTIONS.

!!! An array itself is an object, and has variables like .length .push() etc, the array is only counted as an object when you don't use the [] !!!

5. Combining forces.

Now that you know the Java Script bones (HOORAY!) we can start using our knowledge! Now first, why is this chapter called combining forces? Well, it is because the ECMA thing we just talked about was the 'mere' JavaScript, sphere itself has added some game making functionality to it, and combined them into the engine (it is JavaScript with a custom set of standard functions etc).

Sphere specifics

With sphere specifics I mean non-standard JS objects, functions etc. They are things like Images, Sound, Maps, and Sprites. They 'make' your game. I think you should take a peek in the reference guide to check some stuff out, or check some tutorial on your first game, or of course, keep reading.

Understanding the reference guide

The reference guide is made up like this:

OBJECT TYPE

Object functions etc

object function(argument1, argument 2);
this does ____. Argument1 is for ___ 2 is for __

You should refer to it often.

6. First steps

This chapter will cover the grounds up part of sphere, yay. You'll learn how to design code, and how to use it. We will also cover some functions, and actually draw stuff on the screen etc.

Designing code

An important step before starting to write code, is to design your code. Designing means you put a basic structure in it, which you will stick to until you finish your game.

You can make a design in two ways: You can make a visual design, or you can make a typed out design. A typed design looks like this:

```
Window show
| Do while there is no key pressed
| | Draw the map
| | Draw a window
| | Draw text in the window
| | Show it to the user
| -----
-----
```

Or something like this:

```
Window show
[
  Do while there is no key pressed
  [
    Draw the map
    Draw the window
    Draw text in the window
    Show it to the user
  ]
]
```

You should use the one, which you think is the easiest to understand, the second one is better for learning!

Pseudo

Later, you should be able to translate pseudo code (this is code like you use when designing code) into Java Script code. If you read the ECMA chapter (which was very brief), you will notice that you might already try some things. Ok, first get to the 'logic thinking'. Uhm... Yes, that was a whole chapter.

The holy cookie.

On a day, the holy cookie of Nazareth wants to make a game with sphere. He was thinking of some dude running around on a map, picking up apples to add to his item inventory. To the cookies amazement, sphere did not have item inventory included! NOT EVEN AN ITEM EDITOR! Then the holy cookie looked up in the sky and asked, WHY?! WHY IS THERE NO ITEM EDITOR! Then, a holy mitten came from the sky and told the cookie he didn't need an item editor, all he needed was a dildo. (took from a sphere quote)

Item demo

So, the cookie wants some item script for his game. If you want to make such things, you need some thinking'. We would need something for:

- Holding items
- Adding items
- Removing items
- Using items

Well, first we start with HOLDING ITEMS. What do you need for holding items? It is (Word 2000 corrected!) no dildos... No, it's a list! And of course, we need items! So... Item would be an object, what would the object have?

- A name (duh)
- A script (when you use it)
- A type (weapon, item etc)

Ok, knowing this, we will make the item object, we do this by designing the item object.

This is an Item

```
[
    It has a name as a word
    It has a script as a script
    It has a type as a number
]
```

And lo', there is an item... Empty of course, but we do not yet need it. Then, we need the item list, it is like 'the bag' of the user.

This is a bag

```
[
    It has an itemlist as a list
]
```

Joy, we have a bag type! Now we need a design on how to add items to the bag...

AddItem is a function

> You can pass the item you want to add as what_item

> You can pass the bag you want the item to be added to as what_bag

```
[
  do until I checked all items in what_bag
  [
    if what_bag.itemlist at the point I am
    checking is empty
    [
      stop checking
    ]
  ]
  add what_item to what_bag.itemlist at the point where I stopped
]
```

So, what does this do huh? Well, you should carefully read it, and you will understand that it will add the item at an empty place in the itemlist.

Now for removing items...

RemoveItem as a function

> You can pass the item you want to delete as what_item

> You can pass the bag you want to delete the item from as what_bag

```
[
  do until the list of what_bag.itemlist ends
  [
    if the item has been found at this point
    [
      empty the bag at that place
      tell the engine everything went OK
    ]
  ]
  tell the engine you didn't find the item
]
```

I am not going to comment this, try to understand this piece of pseudonyms!

Why are the dots?

The DOTS my friend, are to indicate that I am referring to an object's other 'thing'. You could read a dot as 'its'.

Translating it to code

If you haven't done yet, read chapter 5. Chapter five will give you information about programming JS.

[/]

The [and] can be translated as { and }, of course.

Do while ...

This can be translated as While(condition)

If...

If can be translated as if(condition)

x as a function...

Function x (using ...

using...

Using passes some arguments, i.e. function x using bla: Function x(bla)

tell...

Use return(value)

list...

A list can be used as an array. I.e., make itembag a list : itembag = Array()

as...

How should it be used? Should it be used as a number, word, or function? Note that JS takes care of this part most of the time.

The Scripts | Example

Itembag

```
function Item(name, type, function)
{
    this.name = name;
    this.type = type;
    this.recall = function;
}

function ItemBag()
{
    this.list = Array();
}

function addItem(item, bag)
{
    found_empty_place = false;
    for(x = 0; x < bag.list.length; x++)
    {
        if(bag.list[x] == 0)
        {
```

```

        bag.list[x] = item;
        found_empty_place = true;
        break;
    }
}
if(found_empty_place)
{
    return true;
}else
{
    bag.list[bag.list.length] = item;
    return true;
}
}

```

```

function removeItem(name_item, bag)
{
    found_one = false;
    for x = 0; x < bag.list.length; x++)
    {
        if(bag.list[x].name == name_item)
        {
            bag.list[x] = 0;
            found_one = true;
            break;
        }
    }
    if(found_one)
    {
        return true;
    }else
    {
        return false;
    }
}

```

```

// Creating some items
apple = new Item("Apple", 1, useApple());
sword = new Item("Sword", 2, useApple());
peach = new Item("Peach", 1, useApple());
// Creating a bag
player_bag = new ItemBag;
// Give it some items
AddItem(apple, player_bag);
AddItem(peach, player_bag);
function useApple(){
    RemoveItem("Peach",player_bag);
}

```

That's all! My code might be a bit different than yours, and might not even 100% fit in our design. Sometimes you need to adjust things, and when you have a big project, the design will come in handy when you are not that good in programming.

7. More Scripting

Yeah, I think it's good for you. You cannot make games without knowing how to code, either, you CAN know how to code, and don't know how to use it with sphere. Here I gather some Scripting tutorials from Sphere Suite Network in an order for you to read it (easy things first). Enjoy!

A basic scripting tutorial

A basic scripting tutorial, to teach a beginner how to create a simple game with Sphere's JavaScript.

Sphere's JavaScript like syntax is simple and rather easy to catch, this tutorial will teach you how to script your first game, a simple one but working one.

Every game requires at least one function. It is called "game". A function is a set of operations grouped together. This "game" function is the function the engine runs when first started, like C's main function. So this function is coded like this:

```
function game() {  
}
```

When you run this script, nothing but a blank screen is seen. Now lets get down to putting something on that screen:

```
function game() {  
    //Function to load a map  
    MapEngine("map.rmp", 60);  
}
```

The script runs the map engine and loads the map "map.rmp". The // lines are called comments, and they allow you to document what the script is doing, the engine does not take them into consideration when compiling the script and so there is no need to worry about comments affecting your script.

```
function game() {  
    //Function to load a map  
    CreatePerson ("my_name", "character.rss", false);  
    AttachCamera ("my_name");  
    AttachInput ("my_name");  
    MapEngine("map.rmp", 60);  
}
```

This script does what the same thing as the one before but now adds the character "character.rss" and allows you to control the character and move the character around. The AttachInput functions attaches the input buffer to the character "character.rss" and so when the up key is pressed, the character moves up. The AttachCamera function attaches the screen's spotlight to your character and thus when your character moves the screen moves too. Try the same script without the AttachCamera function and see what happens. Then try the script without the

AttachInput statement.

Now, you might want to add an introduction sequence. So let's add an intro sequence and add multiple functions. Examine this script and see if you can figure it out:

```
function game() {
    intro();
    start_game();
}
function intro() {
    // loads the image into a variable
    // var statements declare variables that you can store data in
    var intro_image = LoadImage("intro.png");
    intro_image.blit(0, 0);
    FlipScreen();
    Delay(1000);
}
function start_game() {
    CreatePerson ("my_name", "character.rss", false);
    AttachCamera ("my_name");
    AttachInput ("my_name");
    MapEngine("map.rmp", 60);
}
function Delay(time) {
    var until = GetTime() + time;
    while (GetTime() < until);
}
```

That short script will show "intro.png" for 1000 milliseconds and then do the same as the 2nd script. The delay function is to make the system wait for a certain amount of time.

Ok, now here's something to add to this, Aegis just told me that the new Sphere is just like C and has header files called System Scripts, cool huh! So Delay is declared as a system script in the script "time.js" and thus there is an alternative way of doing the delay function and that is by including the system script by typing in EvaluateSystemScript("time.js"); above and then just using the Delay(1000); function without needing the function Delay(time); Like below:

```

EvaluateSystemScript("time.js");
function game() {
    intro();
    start_game();
}
function intro() {
    // loads the image into a variable
    // var statements declare variables that you can store data in
    var intro_image = LoadImage("intro.png");
    intro_image.blit(0, 0);
    FlipScreen()
    Delay(1000);
}
function start_game() {
    CreatePerson ("my_name", "character.rss", false);
    AttachCamera ("my_name");
    AttachInput ("my_name");
    MapEngine("map.rmp", 60);
}

```

A basic scripting tutorial part 2

Sphere games are located in the "sphere/games/" directory. So if your game is called "fun" there'll be a file called game.inf at "sphere/games/fun/game.inf". This file has something like this in it:-

```

name=fun, the game
script=startup.js
screen_width=320
screen_height=240

```

This means that fun, the game has a startup script called "startup.js". So if you open up a text editor and type this below, you make a simple startup script that goes straight into the game.

```

function game()
{
    ChangeMap("worldmap.rmp");
    AddSpriteSet("aegis.rss");
    MapEngine();
}

```

Of course, we don't want to go straight into the game, we'd like to go over some simple techniques, like drawing a title.

There's a function called DrawText(x, y, text) which does this.

The problem is it only draws the text to the back buffer. You want it to be on the screen. Now, there are several ways of getting to the screen, the best two being:-

```

FadeIn(ms);
FlipScreen();

```

But we want the results of it to stay on screen for a set period of time via:-

```
Delay(ms);
```

So with that in mind, we can change the "startup.js" to this:-

```
function game()
{
    DrawText(0,0, "Fun, the game.");
    FadeIn(750);
    Delay(1000);
    ChangeMap("worldmap.rmp");
    AddSpriteSet("aegis.rss");
    MapEngine();
}
```

FadeIn(ms) slowly puts the back buffer to the screen, whilst as FlipScreen() is instant. FadeIn(0); would be the same as FlipScreen(); Although, what we want to do is check some variable. A variable is like a "box in memory" where you can store information. So:- var swords = 0; Would be a perfectly fine variable for a RPG.

```
function game()
{
    var swords = 0;
    DrawText(0,0, "Fun, the game.");
    FadeIn(750);
    DrawText(16,16, "You have " + swords + ", good luck." );
    FlipScreen();
    Delay(1000);
    ChangeMap("worldmap.rmp");
    AddSpriteSet("aegis.rss");
    MapEngine();
}
```

You maybe wondering why we want a variable to be automatically set to 0 everytime the game is set? Well, swords will be saved to a file, and the basic idea of this tutorial is to make a simple in-game shop. Actually, I'm making this up as I go along, so bear with me. Right, we want to make the variable swords to load itself from a file. So:-

```
var items = OpenFile("items.ini");
var swords = items.read("swords",0);
```

So basically, this sets a variable called items, to point to the file "items.ini". The swords variable allocates itself to be the result of the function ReadFileInt(items,"swords",0); Which basically says read the file "items.ini" an int type variable from the "swords" section. And if the "swords" section doesn't exist the default value for it is 0. Now, we just add a very simple shop, so...

```

function shop()
{
    var items = OpenFile("items.ini");
    var swords = items.read("swords",0);
    if(swords == 0)
    {
        ClearScreen();
        DrawText(0,0, "Here's a sword, free of charge.");
        swords = 1;
        FadeIn(250);
        Delay(750);
    }
    items.write("swords",swords);
}

function game()
{
    DrawText(0,0, "Fun, the game.");
    FadeIn(750);
    shop();
    {
        var items = OpenFile("items.ini");
        var swords = items.write("swords",0);
        DrawText(16,16, "You have " + swords + ", good luck." );
        FlipScreen();
        Delay(1000);
    }
    ChangeMap("worldmap.rmp");
    AddSpriteSet("aegis.rss");
    MapEngine();
}

```

The interface of the functions isn't hard to understand. It just takes a little getting used to. If you wanted to make your own function to return the variable you wanted you could write this:-

```

function getValue(section, default)
{
    var data = OpenFile("data.ini");
    var value = data.read(section,default);
    return value;
}

```

This function `getValue(section, default)` presumes all your saved data will come from the file "data.ini". The good thing about this function is it simplifies the coding involved. We always use `data.read(section,default)` as we don't have to worry about different types of variables in sphere. If a variable can be converted to a number (int) from a string it will be if it needs to be.

Like:-

```

var zero = "0";
if(zero == 0)
{
    DrawText(0,0, "Zero is a number, in the form of a string.");
    FadeIn(500); // Fade in is only here to show the results.
}

```

Also, `getValue` needs another function called `setValue` to complete the simplifying of our script.

So:-

```

function setValue(section, value)
{
    var data = OpenFile("data.ini");
    data.write(section,value);
}

```

The only real differences inbetween our `getValue(section, default)` and `setValue(section, value)` functions are:- That `getValue` takes a default parameter and `setValue` takes a value parameter. This shouldn't be too hard for you to understand. Anyhow, lets remake our "startup.js" file with what we know so-far.

```

function getValue(section, default)
{
    var data = OpenFile("data.ini");
    var value = data.read(section,default);
    return value;
}

```

```

function setValue(section, value)
{
    var data = OpenFile("data.ini");
    data.write(section,value);
    CloseFile(data);
}

```

```

function shop()
{
    var swords = getValue("swords", 0);
    if(swords == 0)
    {
        ClearScreen();
        DrawText(0,0, "Here's a sword, free of charge.");
        swords = 1;
        FadeIn (250);
        Delay(750);
    }
    setValue("swords",swords);
}

```

```

function game()
{
    DrawText(0,0, "Fun, the game.");
    FadeIn(750);
    shop();
    {
        var swords = getValue("swords",0);
        DrawText(16,16, "You have " + swords + ", good luck.");
        FlipScreen();
        Delay(1000);
    }
    ChangeMap("worldmap.rmp");
    AddSpriteSet("aegis.rss");
    MapEngine();
}

```

This tutorial was originally by Flik and rewritten by Eugene Siow for my site. For Questions or comments email Flik at flik@r67.net or to kyo_116@hotmail.com.

Control structures

I've been keeping my eyes opened and the most undocumented part of Sphere are the if...else statements and loops, control structures. The sound of this words may make beginners shudder, but there's nothing to be afraid of really. These control structures are one of the most important parts of programming. Loops can be found in all C competition programs. Loops are even more important in algorithm programming and are the things that make a programmers life simple. As you should know I do lots of C, VB, C++, Delphi and Pascal (I don't do Java and not much Asm, which I did but gave up because it was too tedious). And of course you may say, what are all this silly stuff, I want to learn Sphere. What I would reply is that Sphere is connected to lots of these languages in terms of syntax and in algorithms. C is a structured language and is simplistic and thus we will relate a lot with it during this course of this Sphere tutorial. The conditionals in Sphere are very much like those in C.

This is a switch statement that works a lot like the if statement and is in fact better than the if statement for large decisions. It work like this.

```

switch (x) {
    case (y): {
    }
    case (z): {
    }
    default: {
    }
}

```

This should be simple enough to understand, but without any variables inside, it might be hard. So let me illustrate this better with finding the key pressed.

```

var keycode;
keycode = GetKey();
switch (keycode) {
    case (KEY_ESCAPE): {
        Exit();
    }
    case (KEY_ENTER): {
        Next();
    }
    default: {
        Next2();
    }
}

```

Ok, what happens is this. First you get the key entered from the users keyboard by using the GetKey() function and then you store this key in the variable keycode. Next, you use a switch statement to decide what the keycode is. Next you search for KEY_ESCAPE and KEY_ENTER in from the key entered and if the keycode is KEY_ESCAPE then you exit the program and if the keycode is KEY_ENTER then you run the function Next. The next statement is the default statement which tells Sphere that if the key entered is neither KEY_ESCAPE or KEY_ENTER then Sphere should run the function Next2(). The variable keycode must be put in the switch statement to tell Sphere that the variable you want to search is keycode and the case statements tell Sphere what to look for in the variable.

Next, we will look at the if statement which is similar to the switch statement.

```

var keycode;
keycode = GetKey();
if (keycode == KEY_ESCAPE) {
    Exit();
}
if (keycode == KEY_ENTER) {
    Next();
}
else {
    Next();
}

```

The if statement is even simpler to understand as it is a lot like the English. just represented here and there by symbols and stuff. Now to break down the code. like the above switch statement, this if statement does the same thing. It gets the users input key and stores it in key code. Then it tells Sphere, if the value keycode is KEY_ESCAPE then, it will exit the program. Otherwise, if the keycode is KEY_ENTER then the Next function is run. the last statement works like the switches default statement. The else statement, runs the function next if the key pressed is not enter or escape. One more thing. When doing control structures, it would be best to indent your code. (Using the TAB button of course). The code within the control should be indented, making the code look neat and easy to debug and easy to understand. One more very important thing is that in C and Sphere, to compare, one must use == and not =. This will be something new to Basic programmers. The use of = in C and Sphere is strictly for assignment like i = 1 and so on. So to compare whether keycode is equivalent to KEY_ESCAPE a Sphere programmer must use the == operator. (by the way, are you freaked out by my language? if you are, just grab any C book and look through it and you should

understand me better.)

```
while (x==2) {  
}
```

Next, one of the most favorite loops used in Sphere and C. (Of course there are only 3 kinds of loops. My personal preference is the for loop since it was the first loop I used and learned.) The while loop is very simple it loops until the condition is reached, which in this case is x is equivalent to 2. But how do we use this loop when doing our Sphere programming?

```
var x = 0;  
while (x==2) {  
    GetKey();  
    x++;  
}
```

I couldn't think of a better example so I just used this lame one, bare with it. This shows the power of a loop. (Which can be destructive too. Do you know that a high speed loop can burn out your processor? These loops are called infinite loops, which obviously, loop forever. Kids, don't try this at home.) Ok, the loops I'll teach you are just basic loops and these won't burn out your processor, so don't worry. Anyway, with today's advanced Pentium technology, it isn't as dangerous as before. Now, back to the stuff, this while loop gets the key the user pressed for 3 times. this is done by using x as a sort of counter and so when x reaches 2 the loop will exit. x is increased each time the loop is ran so from being 0, it will become 1 and then 2 and Sphere will exit the loop. Simple enough. How about doing something more useful like doing something with the data.

```
var x = 0;  
var keycode;  
while (x==10) {  
    keycode = GetKey();  
    if (keycode == KEY_ESCAPE) {  
        Exit();  
    }  
    x++;  
}
```

This loop, gets 11 keystrokes and then exits the loop, but this loop can also be broken and the program exited when the escape key is pressed anytime in the loop. But this decision making is done in the loop itself. Can store what I loop in in a variable and then use it after I've exited a loop, the answer, sadly, is no. But you can store it in an array of variables. This will be talked about in the tutorial on arrays, but I'll feed your code-hungry minds with some code on arrays in loops first.

```

var x = 0;
var keycode = new Array(11);
while (x==10) {
    keycode[x] = GetKey();
    x++;
}

```

I shan't explain this first so you can try to figure it out on yourself or check out my arrays tutorial. This isn't all, all this does is store all the keys pressed in an array. We need to do another loop to get all these out and scan them.

```

var x = 0;
var keycode = new Array(11);
while (x==10) {
    keycode[x] = GetKey();
    x++;
}
x=0;
while (x==10) {
    if (keycode[x]==KEY_ESCAPE) {
        Exit();
    }
    x++;
}

```

Ok, I must admit this is rather useless and can be done with a much simpler method, but what I'm trying to do is illustrate the theory of it all. This reads in the keystrokes into an array and then scans the array and finds all the KEY_ESCAPES and if a KEY_ESCAPE is found the game is exited. This is explained in detail in the array tutorial.

The next kind of loop is the Do While loop, a little like the while loop but with different priorities. This Do While loop does the certain statement then checks whether the condition is met. Whereas the While loop checks the condition first and then does the statement. This different in priorities is normally not taken into consideration and so I shall not talk much about it but delve straight into the syntax.

```

do {
    GetKey();
} while (x==true);

```

Simple enough. The do while loop is not so commonly used and is quite like the while loop, so you should just take note of the syntax. The for loop, a loop which is more difficult to understand as it has all its increments, declarations and conditional all in the same loop. That's why I like it. It looks rather neat, neater than a while loop and functions about the same.

```

for (start; conditional; update) {
}

```

This is what is a version of it that is closer to your heart. Notice I like to use i as a variable, I don't know why but I just use i, j and k in loops, maybe because of my C background. Of course, you can use any variable.

```

for (i=0; i<=10; i++) {

```

```
}
```

This is what it does. First, it sets *i* to 0 and then it starts looping until the condition is met, which is *i* is equals to 10 and then it does the increments automatically. Of course you can use the increments *i*++, ++*i*, *i*--, *i* + 1 and whatever else.

```
var keycode;
for (i=0;i<=10;i++) {
    keycode = GetKey();
    if (keycode == KEY_ESCAPE) {
        Exit();
    }
}
```

This is the for loop implementation of our earlier while loop and should be easy to understand. First, I declare the keycode to store the name of the key entered. Then i loop it with a for loop, first setting *i* to 0 and then saying that *i* will stop only when *i* is equals to 10. *i* will be increased each time the loop runs. That's it for controls. Check out the next few tutorials and find out all about coding in Sphere.

Arrays

Arrays are very useful things in Sphere and are very good for storing data. These can be used with ease and are actually very simple, even though some beginners don't get the concept of arrays. Arrays are like variables, variables with multiple storage spaces. These array are declared under the same name like the test variable if defined as `var test;` can only store 1 number but if defined as an array, can store any amount of numbers. These arrays are easily accessible by `test[0]` and `test [1]` which in this example store 1 and 2. This may seem a little difficult to understand at first but I'll slowly make it clearer.

```
var x = 0;
var keycode = new Array(11);
while (x==10) {
    keycode[x] = GetKey();
    x++;
}
x=0;
//second part
while (x==10) {
    if (keycode[x]==KEY_ESCAPE) {
        Exit();
    }
    x++;
}
```

This simple loop stores dynamic variables into an array and was mentioned in the earlier tutorial on control structures. Again it may seem way beyond your level but I'll first talk about this then go to more basic stuff, just bare with it and you may end up a better programmer then if I just thought you the very basics first. Ok, now to analyze

the code. A new array is defined by `var keycode = new Array(11);` and 11 is the number of storage spaces in the array. Remember that the 0 character is also part of a character in code, `keycode[0]` is valid. This is a loop to store keycodes in the array `keycode[]`. This loop stores the first key in `keycode[0]` and then loops again and stores the second keycode in `keycode[1]` and this goes on till the loop stops at 10. The reason why I declared 11 for the array as I'm a C programmer and I'm used to the fact that the last storage space in the array is a null character. The second part of the array does this it scans out all the keycodes and then checks each to see if the keycode is the escape key.

```
var keycode = new Array(11);
```

This is how an array is defined. Remember that the first space in the array is 0.

```
keycode[0] = GetKey();
```

This script stores the keycode in the first storage space in the array `keycode`.

Well, arrays are are simple and I'm sure you should agree with me, but they are certainly very, very useful and help a lot when programming. Now you can write your own arrays and store game data and inputted data. Now, to get on with more advanced stuff about arrays. And if you're not sure about arrays then keep on reading.

```
var x = 0;
var keycode = new Array(11);
while (x==10) {
    keycode[x] = GetKey();
    x++;
}
x=0;
//second part
while (x==10) {
    if (keycode[x]==KEY_ESCAPE) {
        Exit();
    }
    x++;
}
```

Commands Entered:

C

A

S

D

F

G

H

Esc

Enter

Backspace

Enter

Stored in arrays:
keycode[0] is C
keycode[1] is A
keycode[2] is S
keycode[3] is D
keycode[4] is F
keycode[5] is G
keycode[6] is H
keycode[7] is Esc
keycode[8] is Enter
keycode[9] is Backspace
keycode[10] is Enter

Does the above illustrate what an array does? I really hope so. Now to get on with more about arrays.

```
var text = new Array(5);  
text[0] = "Hello";  
text[1] = "Bye";  
text[3] = "Hello World";  
text[4] = "Bye World";  
text[5] = "see you soon world";  
DrawText(0,0, text[0]);  
FlipScreen();  
Delay(3500);
```

```
function DrawText(x,y, txt)  
{// this function is available in system scripts...  
GetSystemFont().drawText(x,y,txt);
```

Well, guess what the above is the way to declare variable in an array. Really Simple, then the program draws the text Hello onto the screen. Play around with the above code and try to add a loop function to display all the variables in the array onto the screen. When you do the program, remember to take note of the draw text function and the width and height that the text is drawn.

```
var text = new Array("Hello","Bye","Hello World","Bye World");  
DrawText(0,0, text[0]);  
FlipScreen();  
Delay(3500);
```

```
function DrawText(x,y, txt)  
{// this function is available in system scripts...  
GetSystemFont().drawText(x,y,txt);
```

This is an alternative way to declare variables in an array. Now, for some extra stuff on arrays.

length

returns number of elements in the array

Note :: The length of an array is dynamic.

::var drivers = new Array("opengl", "8bit", "grayscale") has a length of 3

Example :: length_of_array_variable = drivers.length

concat(array)

- returns the array with the specified array concatenated onto it

Note :: Simple example is... three = one.concat(two);

join()

- returns a string of all the elements of an array concatenated together

Note :: This inserts a "," between all elements

reverse()

- returns the array reversed

slice(a,b)

- returns a specified part of the array,
from array_object[a] to array_object[a + b]

Note :: array_object(0,0) will return undefined...

sort()

- returns a sorted array

Note :: A sort function can be passed as a parameter...

:: the sort function must take two values, a and b

:: the array is then sorted according the sort functions return values

:: If the return value is less than 0, the element a should be appear before element b...

:: If the return value is 0, then the two elements being compared should not be switched.

:: If the return value is greater than 0, the element b should be appear before element a...

Example :: array_object.sort(SortByNumber);

```
function SortByNumber(a,b)
```

```
{  
    return a - b;  
}
```

push()

- returns the array with an item pushed onto the end of the array

pop()

- returns the array with an item removed from the end of the array

shift()

- returns the array with the first element at the beginning of the array, removed

unshift()

- returns the array with elements inserted at the beginning of the array

splice(StartIndexNumber, DeleteCount, NewElements)

- allows insertion and deletion of array elements...

Note :: `myArray.splice(3, 2, "jim");`

Starts at the 3 element, deletes 2 elements, and inserts at the element 3, an element with the value of "jim"

Example :: Say you want to remove the 3 element of the array Sphere, you would do ::

`sphere = sphere.splice(3, 1);`

So, one can do a lot with arrays and all it's functions, the most useful ones being sorting, slicing and length and of course not forgetting the rest like push and pop which work like C's stacks. Th sorting functions alike C's bubble sort and Quick Sort algorithms are easily done. So Sphere really is one powerhouse scripting language and if you're getting real furious at it, keep your cool and go on with Sphere bearing the thought, that Sphere is a powerful language, in mind.

8. Game Making

Welcome to the game making part of this tutorial, this part is still incomplete, or lacks quality. It contains some things you should keep in mind, some things that can be used as tricks etc. Enjoy!



Classies!

Most successful RPG games come from square, square actually 'set the standards' for classic Role Gaming (1991–1996). Square was also the most experienced in RPG Making and programming, they started early, also on the nes and the gameboy. They have learned a-lot from it, as well you while creating RPG's.

How it begun

The first thing a classy-rpg needs is a role model, the person which the story follows. Also, for this 1995 feeling, you will need a tile grid with those tilesets :P. Then, the thing that made RPG big was that they were huge for their time. Most people were used to asteroids or some breakout games those days, RPG was something new and exciting.

Mage + Monster = Encounter

Most self made RPGs try to resemble those old classy ones, I also prefer them. But some, you will just need for you OWN good rpg:

- Nice tilesets containing loads of nice animations and shadows.
- Characters which are equippable with weapons etc.
- A unique battle system
- Items, remember that some items must have certain effects, like fire etc.
- You need lots of balance in your game
- You need much maps containing villages etc, also, maps must evaluate and be big.
- There must be enough NPC's & clues so the user doesnt feel bored or alone while

playing the game.

- Plentyfull of 2d cut screnes!
- A story that actaully *works* for a RPG, dont try Harry Potter please! :)
- A good framework for you menus
- Skill !
- Patience (not the computer game!)
- Other people
- Will power
- A bag of cheese snacks.
- You must pay taxes!

We will talk about some of them later. Except the last three. I know this sounds very familiar with you, and you know this is kinda lame because you know this, but I also know you dont keep it in mind enough!

Oooh soloooo miooooo

There is a team tutorial on this site (EDIT: in this booklet) , check it out! A team is probably the most important thing of your RPG. Yes, ofcourse you can fix a RPG yourself, but it will take longer (duh), and you will find it hard for yourself to keep anticipating on it.

The standards

Some of the standards include a splash intro (it's like a short movie or banner before you start the game) containing your teams name, your name, the games name etc. After it, you get the title screen (dunno), or it skips the title screen if the user never started the game before. If so, the user gets a intro of some sort, just dropping in the story or fully explaining etc, if you had your school, you can probably remember how to proper write a storyboard and make up characters and the chronologic and non-chronological, if you don't, we will talk about them later on.

So what's important?

Personally:

1. Story / Characters / Setup
2. Graphics
3. Framework like battling
4. Length
5. Map design / Art / Fonts / Music etc

The story

Before you even start writing a story, you should ask yourself some interesting questions :)....

Midiaz Rez:

Do you want that the 'reader' drops in the story, or will he be informed about what happen, or even make it happen? le, you can make a cutscene about something that happens in the future.

Chronologics:

Will time have a role? Won't the game 'skip parts' like for instance, 100 years? Or will it? Will the user both play in the future and the past? What kind of role has time?

Characters:

Will there be lots of main characters? Or will there only be one?

Self/Him:

Will the user 'see' the story, does the user follow the main character? (you will hear him talk) Or maybe you want him to BE the user (own decisions, no talking), or maybe even a mix!(chrono trigger!).

Good/Bad:

I still see a overdo on good roles.

Time:

Will the user be in 1999, or 60mil BC? Or maybe even unexcisting?

How it plays:

Do you see through the characters eyes or does it seems like someone is telling the story? Still, this has not yet been done much.

--> *Language use:*

What kind of language (rude, irish, joker etc) will the story teller have then?

Single lined:

Will your story be based on one line, or will it have multiple endings? Try something original!

Ok, there are some things I might be missing, but this will do. Dont forget to describe everything in your story! (characters, items etc).

Graphics

2d graphic are harder then 3d graphics, no really. 3d Is just a bunch of preemtives with some texture on it. 2d pixel art requires skill, and talent. But, of course you can also rip them off other games. Do NOT do this! Well, how to do pixel art then? First, you need a descent pixel art software. Tilestudio works great, it has lots of nice functions and allows exportation. I will not cover how to setup your preffered tool, but rather tell you some things first.

What style do I want to use?

If you just start off drawing, you will just see how crap it can get. No really, style is required for good graphics in your game, thus giving it that extra 'feeling' with it:

Manga: This style doesnt use that much color, only for the backgrounds, also, remember to put countour over objects like characters, trees etc.

Reall: You could rip some grass and tile them and make something that looks reall. This stuff can get crap tough.

Fur: Use a fur filter on your graphics or just make them fur. Fur graphics make your game look more like a picture instead of a tilegrid with tiles.

SNES: This kind of style uses 16 color sprites and 256 colour tiles.

NES: Only 16 colours :)

More stuff to make it nice

Reflexion, Dynamic Lights, mirroring, Fire, Flares, 2xSal or other render methods, Rippling, Shadowing, Double Mapped (2 maps, 1 image containing the mask, 256 [ALPHA] = draw pixel from map1, 0 = draw pixel from map2, 125 = blend them etc).

Jobs and Roles

In Sphere, it is very important to organize a role in a team to work on an RPG. Unlike RPGMaker, Sphere is much more powerful and flexible and thus it would be difficult to make a full RPG by yourself in limited time. However, this is still possible and many have proven that.

Even if you're working alone on your RPG project, it is important to define various roles you would take up and thus reading through this tutorial will make you more organised and enable you to plan for your RPG.

These roles are even more important when you are working as a team. Reading this tutorial will teach you the organisation of a team and the specific roles of team members. In large commercial game companies, teams have huge budgets and large teams of professional working together in various positions and this is important even when developing small games.

Story / Plot

An RPG centres around its story and the story and plot enables you to captivate the user and make him continue playing your RPG. The story in an RPG must be interesting, and original, it must also be refreshing and creative. Someone with a good imagination is needed to think of a story and also someone with good writing skills is needed to add interesting dialogue to make the story flow smoothly.

Character Development

One of the main reasons why RPGs are so popular is that you control characters with feelings and personalities. As the game advances the player should get to know the character, and be able to relate to them. He should feel that he is the character and he is living in the world you created in the RPG. Character development is a very important part of an RPG. If the player does not relate with the characters, they will not enjoy the game. The game cannot be called a "Role-Playing Game" anymore.

Coding

Coding includes things like creating battle systems, designing menu systems, making cut scenes, creating NPCs, placing treasure chests, building mini games, creating triggers, etc.. These often add to the gameplay value of an RPG and make up the main skeleton of the game. As Sphere is already a full-blown RPG Engine, this coding is simplified into scripting where most functions required are already defined for you.

Heroes/Monsters Statistics Management

Statistics for the heroes, monsters and NPCs need to be set and tested out so that the game is well balanced and the difficulty level is appropriate.

Item Management

Someone to name and create items like potions, scrolls, swords, shields, etc.

Skills Management

Someone to name and create original skills like magic, strength, dexterity, etc.

Maps / Level Design

Someone to design interesting and detailed maps to explore and puzzles on the map to solve and special triggers to give the user a surprise.

Music / Sound

Someone to compose music that fits in well with the atmosphere throughout the RPG.

Charsets

Someone to create/edit/rip character graphics for the heroes and NPCs of the game / Someone to create poses for the heroes of the game / Someone to search the internet for good charsets for heroes and NPCs.

Chipsets

Someone to create/edit/rip chipsets for the game / Someone to search the internet for good chipsets, that will fit in with how the story writer invisioned certain places to look like.

Custom Systems Graphics

This is someone to create graphics for the custom systems in the game. Graphics are extremely important in a menu system, to make the menu look nice and have an easy layout. Also, as the battle system will be used a lot, it is good to make it as visually pleasing as possible.

All Other Graphics

Someone to create things like pictures and panoramas for the game. / Someone to search the internet for things like good pictures and good panoramas for the game.

Game Tester

Someone to test the game and look for bugs. This person shouldn't just simply play through the game, they should try everything that is possible in the game, test everything and try and make the game break by doing things that people wouldn't normally do.

Story development

Stories for games are nearly always integral to the fun. A good story will keep a player at the game; a great story will keep the player on for too long at a time. I say they are nearly always important, because occasionally a game will come along that is so silly as to have a stupid story, just to increase the humor. Also, the story isn't the key factor in a game; gameplay, graphics, music, and curve of experience all are factors as well. However, my article is not bent towards making a game in general; rather, I'm talking about writing a story itself.

First Tip

Avoid making clichés if at all possible. They make your game seem repetitive and predictable in its storytelling, and that is not good. Rather, try to invent a story off of the top of your head as you write it out. This does not mean you forcefully run from clichés; if you encounter a cliché that is ultimately required for your story, then use it. If the hero's father absolutely has to be the main villain in your story, then use it.

Second Tip

Do not try to write a story as you go along with making your game on whatever gamemaker you are using. Although this process may prevent you from making holes in the story, it can also prevent you from making incredible plot twists. Also, writing as you go may force you to edit or redo your previous work on the game, meaning more work in the end.

Third Tip

Make sure you are at top shape when writing a story. Write when you are fully conscious; writing at 1:00 A.M. isn't a good idea, because you will be tired and might not realize what sorts of plot twists you can implement. Be willing to write when you sit down—work for at least an hour at a time, as you will get more work done than three 20-minute spurts of work. Finally, make sure you are comfortable at your workspace. Play some good music on the radio or off of a CD you have, and maybe sip a glass of water or pop/soda if you want. However, don't listen to anything that makes you think too much— I prefer to listen to music I have gathered for my game, to set moods for myself when writing.

Fourth Tip

Please do not steal material from other games. Many storywriters claim they are 'inspired' to make story parts based on other games they have played. However, I will not enjoy a story whose author was 'inspired' to make a blond mercenary named Clod Stride with a dark past. That was just a joke folks; most of the time, we don't actually

intentionally steal material. Be careful when writing a story that you don't steal events or ideas that worked good in another game.

Fifth Tip

Take your time when writing a story, and make it thorough. I typically take 1–2 months to write a game story [working from 1 to 6 hours in a day], but when I write I draw out everything– world map, every event and outcome, and every plot twist. Don't start working on maps in your game with only a paragraph of information about what your story's about. If you feel you must absolutely start work on the game itself, do something such as designing characters, items, monsters, battle animation, etc. If you feel you can't find the time to write your story, you can try to find someone online who will write one. Just be careful when having others write your story– you may not be happy with the end result.

Writing a game

The following is my exact process of steps for how I write a game. You may use this exact approach, or you may make your own. I am simply showing mine to help others.

I. World Map

I draw out how the world map looks, showing all of the major nations and the important locations. I can always change the map later, if I need to; the important thing is that you begin somewhere.

II. Light Character Information

I make the basic characteristics of how the characters are– their moods, beliefs, and ideals, as well as other information such as how they look and what weapons they use. These are for player characters [PCs] only; other non–player characters [NPCs] come with the story itself.

III. World History and Other

I write out all of the important events that occurred before the game begins, in the history of the game's world. After doing that, I write out what the goals and intentions of the governments and world groups are. Next I'm writing the history of the PCs, as well as getting a general idea of who the greater enemy is to the PCs. I finish by writing a summary of the story in general, showing what basically happens from start to finish.

IV. Introduction to Game

At last, I write out how the game begins. The player is introduced to the main character, or at least tricked into thinking they've met the main character. Remember with introductions to make them unique and interesting– no one likes sitting through 20 minutes of a single character's rant.

V. Finish Earlier Work

Time to make any last changes on parts I, II, or III. They need to become established fact, so that the game can go on smoothly. Of course, they may become edited later for story purposes, but we want to avoid that so that we can avoid holes in the plot.

VI. The Goal of the Game

What will the player be trying to do for most of the game? This is the hidden goal that pops up after the first 5–25% of your game, and is what the player is trying to

accomplish. Many times the goal is to defeat an evil enemy, such as the Greater Enemy, so making your game unique in this will help. Examples of part VI are...

- The following of Sephiroth in Final Fantasy 7
- The seeking of the Zodiac Stones in Final Fantasy Tactics
- The attempt at making a new nation in Legion Saga.

VII. The End Result

Next I make the ending of the game, including plenty of plot twists and a good series of final battles.

VIII. The Filler in the Middle

Now I work on all of the story that goes between parts V and VII. Some may disagree with this, saying that the ending should go after the middle in a fashion of start to finish. However, I find this process easier for making great plot twists.

IX. Editing Your Work

The longest step of the 10-part process, and rightfully so. Now I go back over everything I've done, looking for extra plot twists I can add to the game. I also look for any corrections I need to make, and I make sure there aren't any holes in the story. Basically, this step is for finishing the document and making it look its prettiest.

X. Get Reviews of Story

After I'm done with writing a story, I find some willing fellows on Gaming World Forums to read it and check for errors. I also tell them to give me a review of the story in general, so that I know what to correct next time I write.

And, um... That's the end! Please comment on what you've read; feel free to write about your own story-making habits. (EDIT: Adresses in the resoure reference at final version!)

9. Learnt by hearth

I pro. spelled tha wron! :D. This chapter covers stuff we learned, it might even learn you something new. ALL documents are taken from Flikky. Flikky's tutorials are easy to read and understand thus fit in this chapter. Enjoy! ... Cookie!

Revision

Stuff we will review (is that a word?)

- Variables
- Functions
- Loops
- Objects
- Arrays

We will also learn some new things from Flik:

- Files
- Network
- Map rendering
- Bitmaps
- Animations
- Movement
- Events
- Input

And some practical demonstrations (make while you read stuff):

- Making a dungeon
- Making an ice cave (weee!)
- Making a castle (in sphere...)
- Making a town

Basic revision

Variables

Here's my explanation of how to use variables.

A variable is how you store information. So let's have this code...

`var total = 0;` So that's declaring a variable called total and giving it a value of 0.

Declaring?

Declaring is telling the computer we are going to use a variable.

You know we're declaring something by the fact that we have the word "var".

Also sometimes we have an equals sign after the variable name and a value for the variable.

`var variable_name = value;` To read this out loud you might say:

Create a variable called 'variable_name' and set it equal to 'value'.

`x = 4;` Since there is no 'var' in front of this line, we presume the variable 'x' already exists And we set x to 4.

`left = right;` Whatever is on the left hand side of the equals sign is set to whatever is on the right hand side of the equals sign.

Strings?

A string is something else that you can put into a variable.

```
var str = "Hello";
```

 That declares a variable called str with the string 'Hello'

```
str = "Hi";
```

 // str is now 'Hi' e.g. not 'Hello' anymore

So say we want to store someone's name in a variable...

```
var name = "Jim";
```

```
var str = "Hi " + name + ", how're you?";
```

Weird looking expressions?

```
++number
```

```
number++
```

```
--number
```

```
number--
```

```
number += number
```

```
number /= number
```

```
number *= number
```

```
number -= number
```

++number = add 1 to number and return the new number

number++ = add 1 to number but returning the old number

--number = minus 1 to number and return the new number

number+- = minus 1 to number but returning the old number

```
var c = 6;
```

```
var d = ++c; // a is 7 and b is 7
```

```
var b = 6;
```

```
var a = b++; // a is 6 and b is now 7
```

+ is the plus sign

/ is the divide sign

- is the minus sign

* is the times sign

a += b could also be written as a = a + b

a /= b could also be written as a = a / b

a *= b could also be written as a = a * b

a -= b could also be written as a = a - b

```
var c = a % b;
```

 a mod b meaning c is now the remainder when a is divided by b

Functions

This is how to make user defined functions... A function is a group of lines of code, that can be called upon whenever needed. An example function call is:-
ShowText(0,0,"Hi there!", 3000);

This calls the function ShowText with the parameters 0, 0, "Hi there!" and 3000...

ShowText is defined like this:-

```
function ShowText(x,y,txt, ms)
```

```
{
```

```
    GetSystemFont().drawText(x,y, txt);
```

```
    FlipScreen();
```

```
    var start = GetTime();
```

```
    while(start + ms < GetTime());
```

```
}
```

Now as you can see, ShowText actually takes parameters of x,y, txt and ms... But in calling ShowText like:- ShowText(0,0,"Hi there!", 3000) we've substituted x,y, txt and ms with 0 for x, 0 for y, "Hi there!" for txt and 3000 for ms.

We could call ShowText and make it look exactly like:- ShowText(x,y,txt, ms) which is known as the syntax, simply by doing something like this:-
var x = 0, y = 0, txt = "Hi there!", ms = 3000;
ShowText(x,y,txt,ms);

Ok, that's that done with. Now, let's return a value from our functions.

```
function GetRandomText() {  
    me = GetRandomText.arguments;  
    // me is now pointing to GetRandomText.arguments  
    return me[Math.floor(Math.random()*me.length)];  
}
```

Huh?! No parameters are defined though?! Yeah, you can pass more parameters into a function than the defined parameters... A defined parameter looks like:- ShowText(x,y,txt,ms) ShowText has 4 parameters defined in its header. But in the body of the function, you could access ShowText(x,y,txt,ms, ?, ??, ???, ... etc) by thinking of them as:-

```
ShowText.arguments[0] == x  
ShowText.arguments[1] == y  
ShowText.arguments[2] == txt  
ShowText.arguments[3] == ms  
ShowText.arguments[4] == ?  
ShowText.arguments[5] == ??  
ShowText.arguments[6] == ???
```

ShowText only exists within the ShowText function!
ShowText.arguments is an array of the parameters passed to the function.

So basically, GetRandomText randomly returns one of its parameters.
Let's throw it into the ShowText function as the txt parameter.

```
x = 0, y = 0, ms = 3000;  
ShowText(x,y, GetRandomText("Hey!", "Hi there!", "Yo!"), ms);
```

This will show some random text on the screen for 3000 milliseconds...

Now let's show how to create your own functions...
This is a function with no parameters, that returns nothing:-

```
function blah(){  
    { // body of the function here...  
}
```

This is a function with one parameter, that returns twice the parameter:-

```
function wee(x){  
    { // body of the function here...
```

```
return (2 * x); }
```

Final issue with functions, scope.

If a variable is created within a function, it is destroyed when the function ends. Its scope is local to the function that defined it. A variable that is defined outside a function, is not destroyed when the function ends. Its scope is global, and all functions can access and modify its value.

In this example x is global, and temp is local.

```
var x = 25;
```

```
function blah()
{ var temp = 30; // blah!!
} // temp has been destroyed
// x is still in scope, thus is not destroyed
```

That's it folks, soon, I will cover loops and special statements, like if, with, etc...

Loops

This is how to repeat certain parts of code, over and over... Say you want to do something 20 times, but that something doesn't actually change much each time... An example of this is, showing all the text from an array of text.

You have an array of text strings, defined like so:-

```
var txt = new Array();
txt[0] = "Once upon a time...";
txt[1] = " In a land far far away.";
txt[2] = " A boy met the girls of his dreams...";
txt[3] = " This girl was in trouble though, and needed help.";
// this is a tutorial, not a story, heh. ;)txt[7] = " He got ready to start his quest...";
txt[8] = " Hoping nothing was going to go wrong...";
txt[9] = " This was his destiny!";
```

Now to me it seems obvious how to draw this to the screen, in succession. Simply make a loop.

You could do this:-

```
var index = 0;
while(index < 10)
{
    var txt = txt[index];
    GetSystemFont().drawText(0,0, txt);
    var start = GetTime();
    while(start + 2000 < GetTime());
    index = index + 1;
}
```

Now if you know arrays, this is easy to understand. All you are doing is putting the indexed element of the array into the txt parameter of the font.drawText method.

However, you could write it out the long way round. But this way saves you all that time and effort.

Now, the problem with this is using that damn index variable. ;) Its just a pain to do, so lets use the for loop.

```
for(var index = 0; index < 10; ++index)
{
    var txt = txt[index];
    GetSystemFont().drawText(0,0, txt);
    var start = GetTime();
    while(start + 2000 < GetTime());
}
```

Now this does exactly the same, but in a easier to manage way. The index has been defined in the for loop's header, and given the value of 0. For variable index is 0, to when index is less than 10 is not true, increase index by one. Huh?! Well, for(initialization; condition; increment) is the syntax of the for loop. initialization is var index = 0
condition is index < 10
increment is ++index

This may not seem helpful, but it is.
Note loops take this syntax:

```
loopname(loopdefinition)
statement
This means that:-
while(blah);
```

Does nothing!

The statement preceding this while loop is a null statement, or a semicolon. Semicolons go at the end of lines of code, loops are more like the beginning of a new paragraph. ;)

Alright, next thing.

Notice that if your condition in the while loop is false it will not run?

```
var condition = false;
while(condition)
{
    // this will not run, because the condition is evaluated to false
}
```

To make a loop run once no matter what, you can do this:-

```
do {
    // this will run, at least once
} while(condition);
```

Alright, an example of a cool condition that is designed to be ended in the body of the loop.

```

var done = false;
while(!done) // while not done... ;)
{
// when you are ready to stop using this code over and over...
// set done equal to true
// like so:- done = true;
if(blah() == true)
{
    done = true;
}
}

```

Finally, I think I'll cover break statements. Sometimes in a loop you want it to end before it runs all the way through and back to the header. Loops only end when the header is evaluated. This means that I might have a lot of code that gets executed when it shouldn't! The solution to this problem, is to just break from the loop altogether.

```

var done = false;
for(var i = 0; i < 30; ++i)
{
    // blah...
    if(done == true)
        break;
}

```

This will break from the code. The other thing is continue statements. In a loop if you put:-

```
continue;
```

It will go back to the header of the loop and evaluate it. Now you know the basics of loops, breaks and continues. ;)

Objects

Understand what a variable is?
Understand what a function is too?

Okay good.

An object is essentially a collection of variables (and methods – more on this later) that are stored in one place.

Say we have a battle system, most of the time we'll need a character, or player object or something like that.

```

var Jim = new Object();
Jim.name = "Jimmy";
Jim.hp = 50;
Jim.maxhp = 100;
Jim.atk = 6;

```

```

var Blob = new Object();
Blob.name = "Blob";
Blob.hp = 25;
Blob.maxhp = 50;
Blob.atk = 7;

// let's fight Jim against Blob shall we?

function simple_battle()
{
    var turn = 0;
    while (Jim.hp > 0 && Blob.hp > 0)
    {
        // change the turn
        if (++turn > 1)
            turn = 0;

        // handle attacking, deal damage
        switch(turn)
        {
            case (0):
                Blob.hp -= Jim.atk;
                break;
            case (1):
                Jim.hp -= Blob.atk;
                break;
        }

        // draw stats
        GetSystemFont().drawText(16,32, Jim.name + " " + (Jim.hp / Jim.maxhp));
        GetSystemFont().drawText(16,48, Blob.name + " " + (Blob.hp / Blob.maxhp));

        FlipScreen();
        Wait();
    }

    if (Jim.hp > 0)
        return true;
    return false;
}

function Wait()
{
    while (AreKeysLeft()) GetKey(); GetKey();
}

```

There you have your first simple text based battle system as such.
Using objects!

But why would I want to use an object? Simply, to keep things that should be together, together. Such as in my example there, all the player (or monster) stats, such as hp, maxhp and atk were in the same place.

Now, if we want to improve that above, we do:-

```
function PlayerObject(name, hp, maxhp, atk)
{
  this.name = name;
  this.hp = hp;
  this.maxhp = maxhp;
  this.atk = atk;
}
```

And:-

```
var Jim = new PlayerObject("Jimmy", 50, 100, 6);
var Blob = new PlayerObject("Blob", 25, 50, 7);
```

So instead of using new Object, and then listing out all the properties we need. We use an object called PlayerObject to help us be lazy. In short, if we ever have more than 1 object that is essentially the same thing, we're not being lazy enough.

So now we want to change our simple_battle so that the so called attack code is better.

```
PlayerObject.prototype.take_hp = function(amount)
{
  this.hp -= amount;
  if (this.hp < 0) this.hp = 0;
  if (this.hp > this.maxhp) this.hp = this.maxhp;
}
```

```
PlayerObject.prototype.give_hp = function(amount)
{
  this.hp += amount;
  if (this.hp < 0) this.hp = 0;
  if (this.hp > this.maxhp) this.hp = this.maxhp;
}
```

Notice how the give_hp method and the take_hp method are almost the same? Only one line is different? Let's make it better still...

```
PlayerObject.prototype.take_hp = function(amount)
{
  this.hp -= amount;
  this.on_stats_change();
}
```

```
PlayerObject.prototype.give_hp = function(amount)
{
  this.hp += amount;
  this.on_stats_change();
}
```

```
PlayerObject.prototype.on_stats_change = function()
```

```

{
  if (this.hp < 0) this.hp = 0;
  if (this.hp > this.maxhp) this.hp = this.maxhp;
}

```

Okay, let's go back to our simple_battle function...

And we change it to:-

```

function simple_battle()
{
  var turn = 0;
  while (Jim.hp > 0 && Blob.hp > 0)
  {
    // change the turn
    if (++turn > 1)
      turn = 0;

    // handle attacking, deal damage
    switch(turn)
    {
      case (0):
        Blob.take_hp(Jim.atk);
        break;
      case (1):
        Jim.take_hp(Blob.atk);
        break;
    }

    // draw stats
    GetSystemFont().drawText(16,32, Jim.name + " " + (Jim.hp / Jim.maxhp));
    GetSystemFont().drawText(16,48, Blob.name + " " + (Blob.hp / Blob.maxhp));

    FlipScreen();
    Wait();
  }

  if (Jim.hp > 0)
    return true;
  return false;
}

```

So we've covered objects, creating user defined objects and prototyping object methods... I know I haven't done anything great, I've just done the basics.

Arrays

The array part of this tut is completely the same as of the scripting part of this manual. Read it!

Not so basics

Files

```
var file = OpenFile("filename");
var default_value = 0;
var value = file.read("counter", default_value);
value += 1;
file.write("counter", counter);
```

So that opens the file "filename" from the games 'save' directory.
(It creates the file if it doesn't already exist.)

It then looks in this file for the line like:-
counter=

And file.read("counter", default_value) returns what is after the equals sign. If that line doesn't exist in the file it'll return the default value that you gave it. So the value written after counter= gets assigned to 'value'.

We then add one value. And write to our file the new 'value' for counter.
That's it I guess. So say we have something written under the header 'custom_name' we could do:-

```
GetSystemFont().drawText(16, 16, "Hi " + OpenFile("filename").read('custom_name', "Jimmy") + ", how're you?");
```

Although you'd probably want to do:-

```
var name = OpenFile("filename").read('custom_name', "Jimmy");
GetSystemFont().drawText(16, 16, "Hi " + name + ", how're you?");
```

So say we have a player object like this:-

```
function Player(name, hp, mp)
{
    his.name = name;
    his.hp = hp;
    his.maxhp = hp;
    his.mp = mp;
    his.maxmp = mp;
}
```

```
Player.prototype.load_stats = function() { var file = OpenFile("filename");
for (var i in this) this[i] = file.read(this.name + "_" + i, this[i]);
(EDIT sorry for the layout here!)
```

```
Player.prototype.save_stats = function() { var file = OpenFile("filename");
for (var i in this) file.write(this.name + "_" + i, this[i]);
```

With that we could do:-

```
// create bob
var bob = new Player("bob", 50, 50);
// restore bob to what he once was? ^^
bob.load_stats();
// level up I guess :)
bob.hp += 30;
```

```
bob.maxhp += 60;
bob.mp += 30;
bob.maxmp += 50;
// save bob for the next game
bob.save_stats();
```

Network

Ever thought of creating a multiplayer game? Via the internet, maybe... this is how...

You have 2 different scripts... one person has a client, and the other has a server script... First, we make the server script...

```
var port = 1234;
var data = ListenOnPort(port);
```

So now data is a port object, err well, almost anyway. You need to wait for someone to connect to it (thus making a connection), so you might want something like this:-

```
while(!data.isConnected())
{
    if(IsKeyPressed(KEY_ESCAPE)) break;
    GetSystemFont().drawText(0,0, "Connecting...");
    FlipScreen();
}
```

Which would wait until someone connects to 'data'... but allowing you to press escape if you want to cancel.

Now that a connection is made, send a hello message, like so:-

```
if(data.isConnected())
{ // send a hello message...
data.write(CreateByteArrayFromString("hello: Hi, you've connected to " +
GetLocalAddress());
}
```

Now pretend you've started playing multiplayer pong...

```
data.write(CreateByteArrayFromString("pongx: " + pong.x));
```

Now you need a client... almost the same thing, really.

```
var port = 1234;
var address = "213.13.5.94";
// your server's ip, you can get this from using irc, and the command:-
// /dns theirnickname
var data = OpenAddress(address, port);

while(!data.isConnected())
```

```
{
    if(IsKeyPressed(KEY_ESCAPE))break;
    GetSystemFont().drawText(0,0, "Connecting...");
    FlipScreen();
}
```

```
data.write(CreateByteArrayFromString("hello: Hi, I've just connected to you using: " +
GetLocalAddress());
data.write(CreateByteArrayFromString("pongx: " + pong.x);
```

Anyway, you now need to check for "pongx: " each time you are sent something...

```
if(data.getPendingReadSize())
{
    var txt = data.read(data.getPendingReadSize());
    txt = CreateStringFromByteArray(txt);
    if(txt.indexOf("pongx: ") == 0)
    {
        pong.x = parseInt(txt.substr(txt.indexOf ("pongx: "))); // I think, hehe
    }
}
```

I bet this method would be really, err bad, because I've made it so the server controls where pong.x is... oh well, you get the idea atleast.

A special thanks goes to fenix, without him, this doc wouldn't exist.

Map rendering

Today we will look at Sphere's RenderMap() function. Once you've started the map engine and are walking around in the map, you can start playing around with the render function... An example of this would be:-

```

function Earthquake()
{
    while(AreKeysLeft()) GetKey();
    var StartEarthQuakeTime = GetTime();
    var DoneEarthQuake = false;
    var InputPerson;
    var CameraPerson;
    if(IsInputAttached())
    {
        InputPerson = GetInputPerson();
        DetachInput(InputPerson);
    }
    if(IsCameraAttached())
    {
        CameraPerson = GetCameraPerson();
        DetachCamera(CameraPerson);
    }
    var x = GetCameraX();
    var y = GetCameraY();
    while(!DoneEarthQuake)
    {
        RenderMap();
        SetCameraX(x + ( Math.random() * 10) - Math.random() * -5);
        SetCameraY(y + ( Math.random() * 7) - Math.random() * -5);
        FlipScreen();
        if(GetTime() > 3000 + StartEarthQuakeTime) DoneEarthQuake = true;
    }
    if(InputPerson) AttachInput(InputPerson);
    if(CameraPerson) AttachCamera(CameraPerson);
    while(AreKeysLeft()) GetKey();
}

```

Ok, this may look complex but all it really does is shake the screen for 3000 milliseconds... Lets talk though the main area of focus...

We have a picture of the map that we put into the backbuffer by calling:-

```
RenderMap();
```

Now we have the backbuffer filled with the map, we can call the camera functions, in combination with FlipScreen() to make the map move. And since we are using random x and y variations, it looks like an earthquake.

Enough said about that. Now lets put this into our game. Make a trigger on the map, by right clicking and selecting:- Insert Entity > Trigger.

In the trigger box, type:-
Earthquake();

Then press 'Check Syntax' and 'OK'.

And it will then be an earthquake spot. Remember, RenderMap() fills the backbuffer with the image. If you wanted to change the way the map looks before

showing it on screen, using FlipScreen(), you could do:-

```
RenderMap();
var map_image = GrabImage(0,0, GetScreenWidth(), GetScreenHeight());
// now change the map_image to your liking...
That should do it for an explanation of this function.
```

Animations

Here's my explanation of how to use animations. An animation is really just a collection of images that is played one after the other.

Animations go into your games 'animations' folder
(currently *.mng or *.flc and *.flic files are supported)

```
var ani = LoadAnimation("sphere.flc");
var next_update = GetTime() + ani.getDelay();
var current_frame = 0;
while (current_frame < ani.getNumFrames())
{
    if (GetTime() >= next_update)
    {
        ani.readNextFrame();
        next_update = GetTime() + ani.getDelay();
        current_frame += 1;
    }
    ani.drawFrame(0,0);
    FlipScreen();
}
```

That would load the animation sphere.flc and draw until there were no more frames to draw. next_update is the time when you should read the next [FRAME] for drawing. Easy when you know how... But because I'm nice, I'll make a PlayAnimation(ani) function for you too..

```
function PlayAnimation(ani) {
    var next_update = GetTime() + ani.getDelay();
    var current_frame = 0;
    while (current_frame < ani.getNum[FRAME]s())
    {
        if (GetTime() >= next_update)
        {
            ani.readNext[FRAME]();
            next_update = GetTime() + ani.getDelay();
            current_frame += 1;
        }
        ani.drawFrame(0,0);
        FlipScreen();
    }
}
```

Simply do `PlayAnimation(LoadAnimation("filename"))` and that's it.

Bitmaps

So you can draw a image to the screen, flip the backbuffer, and delay for a while... .. but thats just images, surfaces can do a lot more:-

```
function Delay(ms)
{
    var until = GetTime() + ms;
    while(until < GetTime());
}

function ShowPicture(filename,x,y, delay)
{
    var picture = LoadImage(filename);
    picture.blit(x,y);
    FlipScreen();
    Delay(delay);
}
```

Now you can show pictures using `ShowPicture...`

But really, you need to just understand what `ShowPicture` does...

`LoadImage(filename) ===` Loads the image 'filename' from the games' images directory.
`var picture = LoadImage(filename) ===` Loads the image, and allocates 'picture' as the image object.

`picture.blit(x,y) ===` Draws (blit is an old word meaning draw, heh) the 'picture' image at (x,y)

`FlipScreen() ===` Shows the drawing...

`Delay(delay) ===` Delays for 'delay' amount of time, so that you can view the image..

So now we've got a simple function... lets draw a neat image on the screen for 5 seconds...

```
function game()
{
    ShowPicture("neat.jpg",0,0, 5000);
}
```

Movement

Movement in sphere? Say we have one person, called Jim, and we want to move him down We do:-

```
QueuePersonCommand("Jim", COMMAND_MOVE_SOUTH, true);
```

And Jim will move south by 1 pixel.

Ok, so we want Jim to move 1 whole tile north then?

```
for (var i = 0; i < GetTileHeight(); ++i)
  QueuePersonCommand("Jim", COMMAND_MOVE_NORTH, false);
```

And Jim will move 1 tile north

... But say we want him to turn around before he moves north, we do:-

```
QueuePersonCommand("Jim", COMMAND_FACE_NORTH, true);
for (var i = 0; i < GetTileHeight(); ++i)
  QueuePersonCommand("Jim", COMMAND_MOVE_NORTH, false);
```

And by now we have something annoying to type out...

So we do something like:-

```
function Move(name, command, tiles, immediate)
{
  switch (command)
  {
    case COMMAND_MOVE_NORTH:
      QueuePersonCommand(name, COMMAND_FACE_NORTH, immediate);
      for (var i = 0; i < GetTileHeight() * tiles; ++i)
        QueuePersonCommand(name, command, false);
      break;

    case COMMAND_MOVE_SOUTH:
      QueuePersonCommand(name, COMMAND_FACE_SOUTH, immediate);
      for (var i = 0; i < GetTileHeight() * tiles; ++i)
        QueuePersonCommand(name, command, false);
      break;

    case COMMAND_MOVE_EAST:
      QueuePersonCommand(name, COMMAND_FACE_EAST, immediate);
      for (var i = 0; i < GetTileWidth() * tiles; ++i)
        QueuePersonCommand(name, command, false);
      break;

    case COMMAND_MOVE_WEST:
      QueuePersonCommand(name, COMMAND_FACE_WEST, immediate);
      for (var i = 0; i < GetTileWidth() * tiles; ++i)
        QueuePersonCommand(name, command, false);
      break;
  }
}
```

So now we do:-

```
Move("Jim", COMMAND_MOVE_NORTH, 5);
```

And Jim will move north 5 tiles.

Say we want to spawn a whole army, and move all of them...

```
function SpawnArmy()
{
  var base_x = 100;
  var base_y = 100;

  for (var x = 0; x < 5; ++i)
  {
    for (var y = 0; y < 5; ++i)
    {
      var c = "army_" + x + "_" + y;
      CreatePerson(c, "army.rss", true);
      SetPersonX(c, base_x + x * 20);
      SetPersonY(c, base_y + y * 20);
      Move(c, COMMAND_MOVE_SOUTH, y, false);
    }
  }

  for (var x = 0; x < 5; ++i)
  {
    for (var y = 0; y < 5; ++i)
    {
      var c = "army_" + x + "_" + y;
      // now issue commands to them all
      Move(c, COMMAND_MOVE_SOUTH, 5, false);
      Move(c, COMMAND_MOVE_EAST, 2, false);
      Move(c, COMMAND_MOVE_SOUTH, 10, false);
    }
  }
}
```

Huh?

Well, the first bit of that code is designed to create the whole army and move them all into a line...

```
YYYYY = 4
YYYYY = 3
YYYYY = 2
YYYYY = 1
YYYYY = 0
```

The ones at the back, have to move 4 tiles to get in line with those at the front...

Once they are all in line (they aren't actually in a line, just in terms of commands we give them) we can issue commands to all of them. Thus the second bit of the code. Move south 5 tiles, move east 2 tiles and then move south 10 tiles...

Then when they've all finished their commands, they're all back where they started. (I mean in their formations :))

Okay, so we've spawned an army and made them walk along..

"I want an guy to wander around" I hear you cry...

Easy.

```
function GenerateRandomMovement()
{
    var person = GetCurrentPerson();
    var commands = new Array(COMMAND_MOVE_NORTH, COMMAND_MOVE_EAST,
COMMAND_MOVE_SOUTH, COMMAND_MOVE_WEST);
    var r1 = Math.floor(Math.random() * commands.length);
    var r2 = Math.floor(Math.random() * 3); // 3 being the number of tiles

    Move(person, commands[r1], r2, false);
}
```

And then:-

```
SetPersonScript("Jim", SCRIPT_COMMAND_GENERATOR, "GenerateRandomMovement());
```

Jim will now wander randomly.

Input

Missing :(

...

Missing :(

Practical Demos

By Flik!

Dungeon

A 'how to make a dungeon' tutorial... First up, if you're really new to Sphere.. Create a New Project, then create a new map and save it as 'dungeon.rmp' and a new script saving that as 'main.js'. (We now have 3 windows open, a project window, a map window, and a script window) We also want to add a sprite to use, so we'll just take one from somewhere else and put it in our game's spritesets folder.

Open up main.js and type:-

```
function game()
{
    CreatePerson("Flik", "pacman.rss", false);
    AttachInput("Flik");
    AttachCamera("Flik");
    MapEngine("dungeon.rmp", 60);
}
```

And save the script. What does that script do? Well, when our game starts, it looks for the 'game' function and runs it. Within the game function we have the following instructions:-

```
CreatePerson("Flik", "pacman.rss", false);
AttachInput("Flik");
AttachCamera("Flik");
MapEngine("dungeon.rmp", 60);
```

The CreatePerson creates a person object on the map. The person is called "Flik" and is created from the spriteset "pacman.rss". AttachInput lets us take control of "Flik" AttachCamera makes the screen follow where-ever "Flik" goes. MapEngine starts the map engine using the map dungeon.rmp and at 60 frames per second.

Okay, now we're ready to start working on our map. A dungeon needs treasure chests, and freaky never ending tunnels. (Create a spriteset called chest - with two directions, 'closed' and 'open') Right click on your map, Insert Entity > Person..

Up pops a little window and we fill it in like so:-

Name: chest_1
Spriteset: chest.rss

Make sure 'On Activate (Talk)' is selected and then put:-

```

var f = OpenFile("save.dat");
if (f.read(GetCurrentPerson() == 0))
{
    // Talk("Ohh, a sword.");
    // give 'Flik' a sword
    SetPersonDirection(GetCurrentPerson(), 'open');
    f.write(GetCurrentPerson(), 1);
}

```

Into the dialogue box below.

I haven't bothered with any actual item code because that's dependent on your battle system. So all we do is change the direction of the chest to open and save it as open by saving it to a file.

You also need to add a map script...

Map > Properties

Click the Entity box and type:-

```

var p = GetPersonList();
var f = OpenFile("save.dat");
for (var i = 0; i < p.length; ++i) if (p[i].indexOf("chest_") == 0) SetPersonDirection(p[i],
(f.read(p[i], 0) == 0 ? 'closed' : 'open'));

```

Next a dungeon needs freaky never ending tunnels. So on your map, draw a repetitive pattern. (Make sure to get the pattern just right) At some point in the pattern, add a trigger:-

(Right click on the map, Insert Entity > Trigger..)

To make a corridor that never ends if you are going south, you insert the trigger of:-

```
SetPersonY("Flik", (GetPersonY("Flik") - (GetTileHeight() * 5)));
```

That code presumes you have a pattern repeating every 5 tiles.

Reasoning:

Imagine a screen of tiles like this:-

```

12345123451234T1234512345
  ^  X   ^

```

Where T is the trigger, and ^ mark the beginning and end of the screen. X marks where you are. When you hit T, we place everything back 5 tiles.

```

12345123451234T1234512345
  ^  X   ^

```

That creates the illusion of the corridor never ending.

Notice where T is now. The same place, just you've moved back 5 tiles.

Check my files page for the 'game' that goes with this tutorial. (EDIT:

<http://sphere.sourceforge.net/flik/>)

10. The map editor

The map editor is proly one of the 'powerful' features of sphere, it is very advanced, and many tricks can be applied. This chapter is for understanding how it works, and what things do. I hope you enjoy!

Tilesets

You do not instantly need tilesets when you make a new map (you can choose a blank one instead), you edit the tiles while you place them on the map. Though, after making a tileset, you can save it. If you wish to use a RM2k or Toolkit or whatever kind of tileset, you should import it. Click File, Import, Image to tileset.



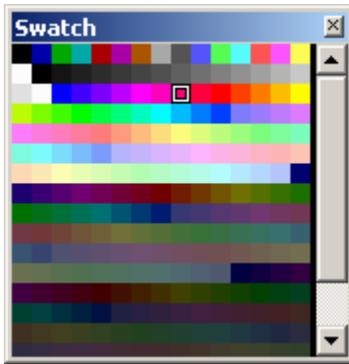
If you are using RM2k chipsets, you should set it to 16 x 16. Please note that you just save your tileset somewhere, you do not need to include your tilesets to your game, because they are implented in the map.

Maps

Insert a new map, open it, and click 'Yes'. Now you will get this dialog:



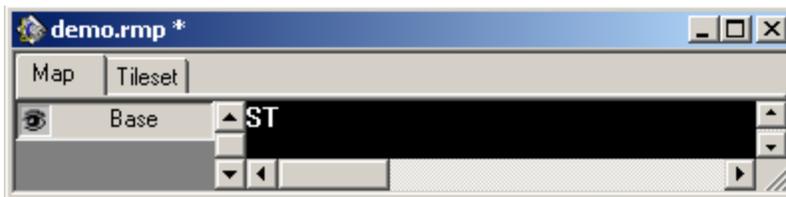
The width and height are the w and h of the 'Base layer.' You can leave the tileset blank just for now, or if you have imported a RM2k chipset, load it. Then press 'Ok'. You will now get this window (the map editor, tileset editor) :



The swatch is for easy color selection.

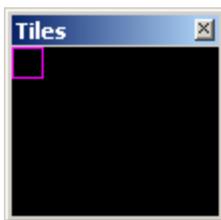
Alright, that is kind of a bunch of non saying windows for you, but I will explain.

Map tab



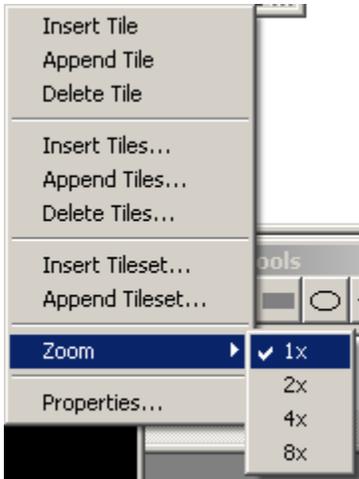
The map tab is for editing the layers on your map, at the left side of the window is the layer tab, the main frame contains the map editing.

Tiles

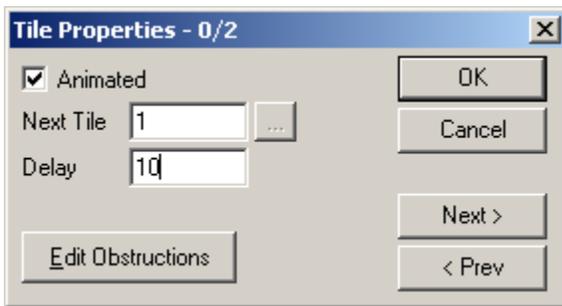


With this you can select the tile you want to place, OR the tile you want to edit in the Tileset tab.

Tileset tab



Insert tile insert tiles, do not use append, I can recall something bad from it ;). Insert tiles means you add multiple in one time. Insert tileset gives you the ability to mix tilesets. Zoom will zoom the tiles if they are not that clear to you. Properties has some interesting tile properties which are very important to know also.



If you have the animated button checked for a tile, the map will say it's an animation. Animations are like water, fire etc, anything that dynamically moves on a map. The map will change to 'next tile' in 'delay' frames. Let's say you have 3 water frames, you would have frame 1 point to frame 2 with a delay of let's say 10 frames. Then, you set the second frame to point to the third one also using 10 frames, and finally, you will point frame 3 back to frame 1, starting over again! (animation). Don't forget to press 'ok.'.

Obstructions

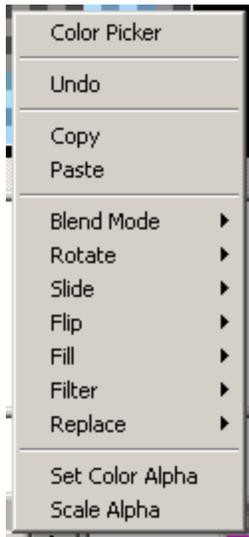
Obstructions are needed for the engine to tell if the hero is held back by a wall or a tree. You can draw your own obstructions or use pre-set ones.



The pink part is there the hero is held up.

More on drawing tiles.

There are more functions for drawing tiles, to make life easier to say it shortly. They are not needed in anyway!



Color picker: Right click a pixel and select this option, it will result in selecting the pixels color.

Undo: Undo your last changing

Copy / Paste: Copy and Paste tiles

Blend mode: Replace : Use the color in full (use the color itself), Blend: Blend the color until it's opaque (255 [ALPHA]).

Rotate: Rotate the tile

Slide: Pan the tile

Flip: Flip the tile

Filter: Filter the tile with a graph filter. Blend will blur the image, noise will make the tile look more naturel / colourful.

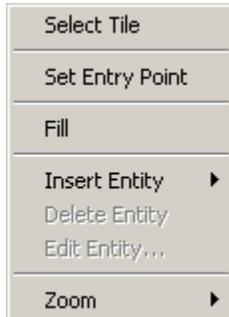
Replace : Same as blend mode, only for one time.

Set Color [ALPHA]: Set the [ALPHA] of this color, and all the pixels in the tile.

Scale [ALPHA]: Unsure... :D

Map Editor

The map editor... Wow... Isn't it impressive? So functional... Ok, right click on the layer for editing, you will see this menu:



Select Tile: Select a tile from the map TIP: Add all the tiles to the map and use this to order your tilesets

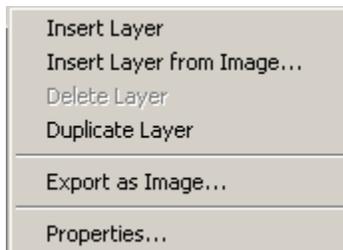
Set Entry Point: Set the place where all Entities will start when they are created manually with Java script.

Fill: Fill the area with the selected tile.

Insert Entity: Add a NPC.

Zoom: ZOoom! :).

Ok, first things first... Let's get to the layer part. I think you need at least 2 layers for a RPG, a base, for walking, and an upper, for overlapping. Notice that you must order you layers... Right click the layer bar... (the base layer bar).



Insert layer: Add a layer to your map.

Insert Layer from Image: Interesting option, draw maps in programs like Pain Shop Pro and put them in sphere with one click. You can also make panoramas with it.

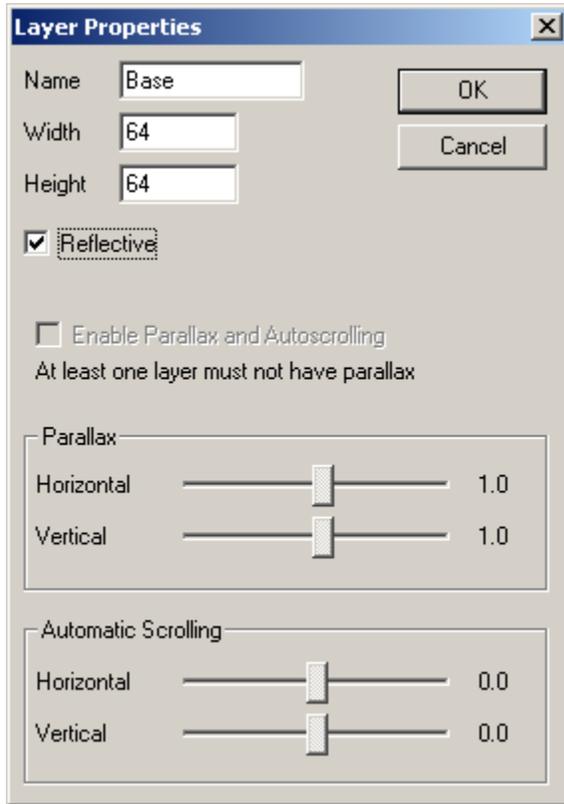
Duplicate Layer: Heh.

Delete Layer: You need at least ONE layer.

Export as image: Easy creation of snaps from your maps!

Properties

Yes, layers have properties, they are important! (Again...)... Wow, whasap with the dots?



Reflective: Use the layer for reflecting like a mirror on other layer, reflection will mirror the image and use the remaining [ALPHA] (thus 'unfilled' space) for filling it up. You can make nice water effects with it. (somewhere in this guide is a demo snapshot).

Parallax: Parallax is how fast the layer moves VS other layers.

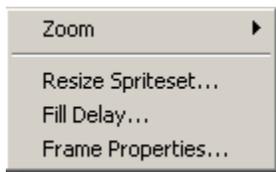
Auto Scroll: Scrolling without moving, it always does move. (if the user isn't walking in the opposite direction ;)).

11. Sprite Sets editor

With the sprite editor you can create / edit sprites! Wow...

Global

Menu Options



Zoom: May the gods be with you...

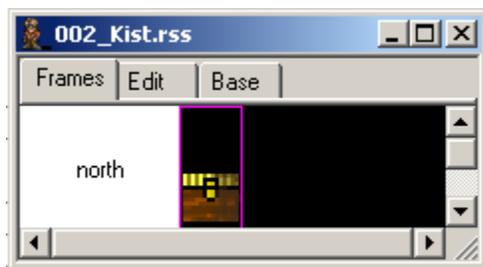
Resize Spriteset: Another useful sphere ability!

Fill Delay: The standard delay before changing frames in the engine.

Frame Properties: Delay for only THIS frame (which you selected).

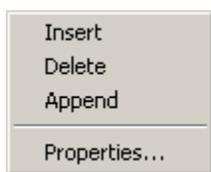
Frames tab

In the frames tab you add directions and frame you have created.



Direction bar options

You can get the direction bar options by right clicking the directions bar (it is white), in the empty space, or an existing direction to change the properties from.



Insert: Insert a direction.

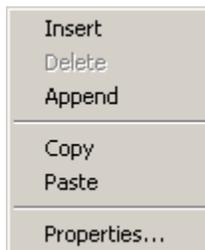
Delete: Delete a direction

Append: Append one.

Properties: Change the name of the directions.

Direction frame bar options

You can get this by right clicking on the space next to the direction bar.



Properties: Change the delay of only this frame, used only ONCE. (not global that is).

What are directions good for?

Well, north south etc are needed for movement. If you add another one, you will need to learn how to use it with Java Script. You can make stuff like 'laughing', 'crying', etc.

Notes

The edit tab is the same as in 10.1, it's the same editor. The base tab is for setting the obstruction. Have fun...

A. Sphere Functions

From now on, we will look at all functions and objects separately. Basic knowledge of all is needed to create a game.

Functions take different places in different programming languages. Structured programming languages, like C, are built on functions. Object Oriented Programming Languages, like C++, are not so dependent on functions. Languages like Basic name functions subs (even though they do have functions too). How does Sphere relate to functions? Sphere depends on functions to run. There are three main types of Sphere functions, user defined functions, system script functions and built-in function. the function "game" is the most basic function in Sphere and every working game should have a function game function.

```
function game() {  
}
```

Function game is the first and only function Sphere looks for by itself. Of course, you can tell Sphere to look for specific functions. Function game is run when the game starts, working a lot like void main and sub main.

User Defined functions

To call another function, one would simply define the function and then call it:

```
function game() {  
    second_function();  
}  
function second_function() {  
}
```

This script is simple it runs second_function when the game starts as it is called in function game. This program basically does nothing but tells you how to use functions. Functions can be named almost anything and should be named informatively like second_function and not just s or i. Functions can only be one word and if split into two words must be separated by an underscore _ . Functions also cannot begin with a numeral like 1 or 2 or 3. Functions in Sphere are case-sensitive and second_function cannot be called as Second_function or SECOND_FUNCTION or SeConDFunctlon. These functions you create on your own are called user defined functions. Other kinds of functions are built in functions and system script functions. Built in functions are those built into Sphere. System Script functions are those in the system scripts included in Sphere. I will talk about user defined functions first. These are defined by you and can span over a few scripts and included into a main script. The statement `evaluatescript(script)` will do the job well. Creating functions spanning over a few files is something wise as it will minimize compile time, make it easier to debug and etc. But this is also difficult to navigate different scripts with a text editor like notepad, so I suggest using the Sphere IDE or Sphere Suite, which has a project and file viewer that is docked on the left of the screen allowing easy editing of scripts in a project. Now this is a simple example of a game that spans over a few scripts. This system is similar to a common C++ one with plenty of header files, etc.

```

filename :: script1.js
function map_function() {
    //Function to load a map
    CreatePerson ("my_name", "character.rss", false);
    AttachCamera ("my_name");
    AttachInput ("my_name");
    MapEngine("map.rmp", 60);
}

```

```

filename :: script2.js
EvaluateScript("script1.js");
function game() {
    map_function();
}

```

The above script is a simple one that includes the map_function function from the script script1.js and runs it from script2.js. This is a real cool feature in Sphere and enables you to write different segments of code in different files. This would allow you to have a script named time.js that does all the time and delay functions and a battle.js that does the battle system functions. But including scripts from all over can make the game very messy and it only shows its power in large game designs, so if you're planning on a small game you could just write everything into one script file.

System Script defined functions

System scripts are also useful. Below is the list of functions that are included in System Scripts.

System Script :: audio.js

ChangeMusic(filename)

Change music changes the sound file that is being played the sound file to be played being *filename*

PlaySound(filename)

Play Sound plays the sound file *filename*

Remember to include EvaluateSystemScript("audio.js");

System Script :: color.js

Colors Defined

This system script creates keywords for the colors like black and red and is good for fast coding and for beginners to do the basic colors. Instead of using CMYK or RGB and remembering the codes or trying them out for common colors, you could use this system script. The color keywords defined are as follows.

Black
White
Red
Green
Blue
Cyan
Magenta
Yellow

Remember to include `EvaluateSystemScript("color.js");`

System Script :: menu.js

Menu()

This creates a menu for you, like the one you get when you start Sphere and it allows you to choose options. This menu is used in the following way.

```
var menu = new Menu();  
menu.addItem("new game", processmenu(), color);  
menu.execute(0, 0, 640, 480);
```

Ok the first line is simple, it creates a new Menu and stores it as the variable "menu". Then it adds the item "new game" which will show on the screen when the menu is executed, then there is a processmenu() function which will be called when the gamer presses enter on that particular item. The next color sets the color of the menu item if you don't specify it, Sphere will use the default color, white, which I feel looks rather nice. Next the menu.execute(); function is called. This executes and shows the menu on the screen. The first and second zeros represent the width from the left and the height from the top respectively. The next 640 and 840, represent the width and height of the menu respectively. So it's rather simple. Some other, special properties that can be added are the following.

menu.escape_function = my_function;

the function that is called when the escape(Esc) key is pressed

menu.font = LoadFont("whatever");

This is simple, it changes the menu font to the following font.

menu.arrow = LoadImage("whatever_else");

This changes the menu's arrow icon to whatever_else.

menu.up_arrow = LoadImage("whatever_else");

This changes the menu's up arrow icon to whatever_else. Note that the up arrow is up_arrow. Remember the underscore.

menu.down_arrow = LoadImage("whatever_else");

This changes the menu's down arrow icon to whatever_else. Note that the down arrow is down_arrow. Remember the underscore.

menu.window_style = LoadImage("whatever_else");

This changes the menu's window style icon to whatever_else. Note that the window style is window_style. Remember the underscore.

Remember to include EvaluateSystemScript("menu.js");

System Script :: oldsphere.js

ClearScreen()

This clears the screen to the default color, black. Also defined in screen.js. This function is a famous Sphere 0.87 one.

Delay(time)

This is the delay function also included in the time.js system script. It holds the system at the position for the certain number of milliseconds that are represented by *time*. This function is a famous Sphere 0.87 one.

Remember to include EvaluateSystemScript("oldsphere.js");

System Script :: screen.js

ClearScreen()

This clears the screen to the default color, black.

FadeOut(milliseconds)

This is the improved fade out function from Sphere 0.87. The fading is much smoother. The *milliseconds* representing the time it takes to fade out.

FadeIn(milliseconds)

This is the improved fade in function from Sphere 0.87. The fading is much smoother. The *milliseconds* representing the time it takes to fade in.

FadeToColor(msecs, clr)

This is the fade to color function used in the fade out prototype allows you to fade to a certain color. With msecs as the number of milliseconds the fade takes and clr as color it fades to. If color.js is included, you can easily specify common colors.

FadeFromColor(msecs, clr)

This is the fade from color function used in the fade in prototype allows you to fade from a certain color. With msecs as the number of milliseconds the fade takes and clr as color it fades from. If color.js is included, you can easily specify common colors.

Remember to include EvaluateSystemScript("screen.js");

System Script :: time.js

Delay(time)

This holds the system at the current position for the certain number of milliseconds that are represented by *time*. This function is a famous Sphere 0.87 one.

Remember to include EvaluateSystemScript("time.js");

System Script :: timed_animation.js

TimedAnimation(animation)

This is a function that plays an animation sequence which is defined as *animation*.

Remember to include EvaluateSystemScript("timed_animation.js");

System Script :: timer.js

Timer()

This is a powerful function that acts as a timer. This is how, it is used:

```
var a = new Timer();  
a.pause();  
a.unpause();  
a.getMilliseconds();
```

The function a.pause(); pauses the timer for a bit. a.unpause() resumes updating the timer each second and a.getMilliseconds() returns the number of milliseconds since the timer was created.

Remember to include EvaluateSystemScript("timer.js");

Ok, that's all with System Script functions. The last type of functions are Sphere's Engine Functions. These engine functions are built into Sphere and of course are the most important and basic functions.

Built-in functions

Script Functions

EvaluateScript(script)

- Reads the script in and uses it as if it were a part of the current script.

ex: EvaluateScript("myscript.js");

EvaluateSystemScript(script)

- Reads in one of the preset system scripts for use in the current script

ex: EvaluateSystemScript("menu.js");

GarbageCollect()

- invokes the JavaScript garbage collector

Engine Functions

Exit()

- Exits the Sphere engine unconditionally

Abort(message)

- Exits the Sphere engine unconditionally, displays the 'message' to the user

Debugging (the log object)

OpenLog(filename)

- opens a log file for use under the filename. If Sphere is unable to open the file for logging, the engine will give an error message and exit.

If Sphere is successful in opening the file, it will return a log object for use.

ex: var myLog = OpenLog("game.log");

`log_object.write(text)`

- writes a string of text under the current block.

ex: `myLog.write("Starting system...");`

`log_object.beginBlock(name)`

- creates a "block" which is indent inside the log with the name as the title of the block. Any subsequent write commands will go under the newly created block.

ex: `myLog.beginBlock("Video Information");`

`log_object.endBlock()`

- closes the current log block.

System Interfaces

Video

`FlipScreen()`

- displays the contents from the video buffer onto the screen. Then the video buffer is cleared. You **need** to call this to make anything you've drawn in code to appear on the screen.

`SetClippingRectangle(x, y, w, h)`

- Sets a clipping rectangle of width *w* and height *h* at (*x*, *y*) into the video buffer. Anything drawn outside the rectangle is not drawn into the video buffer.

`ApplyColorMask(color)`

- fills the whole screen with the color specified. Note that the color passed must have an [ALPHA] that is less than 255. Otherwise, it'll just make the screen solidly that color.

`SetFrameRate(fps)`

- allows you to set the maximum frames rendered per second that the engine is allowed to draw at most.

`GetFrameRate()`

- Returns the current fps (set by `SetFrameRate`)

`GetScreenWidth()`

- returns the width of the engine screen

`GetScreenHeight()`

- returns the height of the engine screen

Graphic Primitives

NOTE: Make sure to use FlipScreen() when you're done drawing

Point(x, y, color)

- plots a point onto the video buffer at (x, y) with the color

Line(x1, y1, x2, y2, color)

- draws a line from (x1, y1) to (x2, y2) with the color

GradientLine(x1, y1, x2, y2, color1, color2)

- Draws a line from (x1, y1) to (x2, y2) with a color fade from color1 to color2

Triangle(x1, y1, x2, y2, x3, y3, c)

- Draws a filled triangle with the points (x1, y1), (x2, y2), (x3, y3), with the color c

GradientTriangle(x1, y1, x2, y2, x3, y3, c1, c2, c3)

- Draws a gradient triangle with the points (x1, y1), (x2, y2), (x3, y3), with each point (c1 = color of (x1, y1), c2 = color of (x2, y2), c3 = color of (x3, y3)) having a color to generate the gradient of the triangle

Rectangle(x, y, w, h, c)

- Draws a rectangle at (x, y) of width w and height h, filled with color c.

GradientRectangle(x, y, w, h, c_ul, c_ur, c_lr, c_ll)

- Draws a gradient rectangle at (x,y) with the height h and width w. Each corner of a rectangle (c_ul = color of upper left corner, c_ur = color of upper right corner, c_lr = color of lower right corner, c_ll = color of lower left corner) accepts a color information to generate the gradient of the rectangle.

Input

Keyboard

AreKeysLeft()

- returns true or false depending if there are keys from the key input queue.

GetKey()

- returns the first key in the queue. If there are no keys in the queue, Sphere will wait until there is a key in the queue.

IsKeyPressed(key)

- checks if the key has been pressed. Returns true if 'key' is pressed....

Mouse

SetMousePosition(x, y)

- Sets the x and y of the mouse cursor

GetMouseX()

GetMouseY()

- returns the location of the mouse cursor within the engine screen

IsMouseButtonPressed(button)

- returns true if the button is pressed

allowed button values are: MOUSE_LEFT, MOUSE_RIGHT, MOUSE_MIDDLE

Time

GetTime()

- returns the number of milliseconds since some arbitrary time.

ex:

```
var start = GetTime();  
while (GetTime() < start + 1000) {}
```

ByteArray Object

ByteArrays are pretty much only used for networking (see below for Network stuff)

CreateByteArray(size)

- returns a ByteArray object of 'size' bytes

CreateByteArrayFromString(string)

- returns a ByteArray object from string 'string'

CreateStringFromByteArray(array)

- returns a string from a ByteArray

`bytearray_object[index]`

- returns the allows you to access the index in the array

Networking

`GetLocalName()`

- returns a string with the local name of your computer

`GetLocalAddress()`

- returns a string with the IP address of your computer

`OpenAddress(address, port)`

- attempts to open a connection to the computer specified with 'address' on 'port'
returns a socket object

`ListenOnPort(port)`

- listens for connections on port, returns a socket object if successful

`socket.isConnected()`

- returns true if the socket is connected

`socket.getPendingReadSize()`

- returns the size of the next array to be read in the socket

`socket.write(byte_array)`

- writes a ByteArray object into the socket

`socket.read(int size)`

- reads from the socket, returns a ByteArray object

Sphere Objects

Colors

`CreateColor(r, g, b [, a])`

- returns a color object with the color r is Red, g is Green, b is Blue,
and a is [ALPHA] (translucency of the color).

Note + [ALPHA] of 0 = transparent, [ALPHA] of 255 = opaque

+ [ALPHA] is optional, and defaults to 255 if not specified

`BlendColors(c1, c2)`

- returns a color object that is the blended color of color c1 and c2

`BlendColorsWeighted(c1, c2, c1_weight, c2_weight)`

- blends two colors together, allowing you to specify the amount of each color

ex:

`BlendColorsWeighted(a, b, 1, 1) // equal amounts (like BlendColors())`

`BlendColorsWeighted(a, b, 1, 2) // 33% a, 66% b`

`color_object.red`

- the red component of a color object

color_object.green

- the green component of a color object

color_object.blue

- the blue component of a color object

color_object.alpha

- the [ALPHA] (translucency) component of a color object

+ all color components are from 0-255 (unsigned 8-bits each, 4x8bit == 32bit)

Maps Engine

MapEngine(map, fps)

- starts the map engine with the map specified and runs at 'fps' frames per second

ChangeMap(map)

- changes current map

ExitMapEngine()

- Exits the map engine

UpdateMapEngine()

- updates map engine (state of entities, color masks, etc.)

Maps

GetNumLayers()

- get number of layers on map

GetLayerWidth(layer)

- get width of 'layer'

GetLayerHeight(layer)

- get height of 'layer'

GetNumTiles()

- return number of tiles in map

SetTile(x, y, layer, tile)

- changes tile on map to 'tile'

GetTile(x, y, layer)

- returns tile on map

RenderMap()

- Renders the map into the video buffer

SetColorMask(color, num_frames)

- applies a color mask to things drawn by the map engine for 'num_frames' frames

SetDelayScript(num_frames, script)

- in 'num_frames' frames, runs 'script'

ex: SetDelayScript(60, "ChangeMap('forest.rmp')");

this tells the map engine to change to forest.rmp after 60 frames

Input

BindKey(key, onkeydown, onkeyup)

- runs the 'onkeydown' script when the 'key' is pressed down and runs 'onkeyup' when the 'key' is released

ex: BindKey(KEY_SPACE, "mode = 'in';", "mode = 'out;");

refer to keys.txt for a list of key names

UnbindKey(key)

- unbinds a bound key

AttachInput(person_entity)

- makes the 'person_entity' respond to the input

(up = KEY_UP, down = KEY_DOWN, left = KEY_LEFT, right = KEY_RIGHT)

DetachInput()

- releases input from the attached person entity

IsInputAttached()

- returns true if a person is attached to the input

GetInputPerson()

- returns a string with the name of the person who currently holds input

SetUpdateScript(script)

// calls 'script' after each frame (don't draw stuff in here!)

SetRenderScript(script)

// calls 'script' after all map layers are rendered

SetLayerRenderer(layer, script)

- calls the rendering 'script' after 'layer' has been rendered. Only one rendering script can be used for each layer of the map

Camera

AttachCamera(person_name)

- Attaches the camera view to specified person

DetachCamera()

- Detaches camera so it can be controlled directly

IsCameraAttached()

- returns true if the camera is attached to a person, false if the

camera is floating

GetCameraPerson()

– returns a string with the name of the person whom the camera is attached to

SetCameraX(x)

SetCameraY(y)

– set the top-left corner of the view (in pixels) of the camera

GetCameraX()

GetCameraY()

– returns the co-ordinates of the top-left corner of the view (in pixels) of the camera

Entities

Persons

CreatePerson(name, spriteset, destroy_with_map)

– returns a person object with 'name' from 'spriteset'. If Sphere is unable to open the file, the engine will give an error message and exit. destroy_with_map is a boolean (true/false value), which the spriteset is destroyed when the current map is changed if the flag is set to true.

DestroyPerson(name)

– destroys the person with the name

SetPersonX(name, x)

SetPersonY(name, y)

SetPersonLayer(name, layer)

– sets the position of the person on the map

SetPersonDirection(name, direction)

SetPersonFrame(name, frame)

– sets which frame from which direction to display

GetPersonX(name)

GetPersonY(name)

GetPersonLayer(name)

– gets the position of the person on the map

GetPersonDirection(name)

GetPersonFrame(name)

– gets the frame and direction that are currently being displayed

FollowPerson(name, leader, pixels)

– makes the sprite 'name' follow 'pixels' pixels behind sprite 'leader'

SetPersonScript(name, which, script)

– sets 'script' as the thing 'name' does in a certain event

the five events are

- + SCRIPT_ON_CREATE
- + SCRIPT_ON_DESTROY
- + SCRIPT_ON_ACTIVATE_TOUCH
- + SCRIPT_ON_ACTIVATE_TALK
- + SCRIPT_COMMAND_GENERATOR

(SCRIPT_COMMAND_GENERATOR will be called when the command queue for the person runs out (for random movement thingies, etc))

CallPersonScript(name, which)

- calls a person's script from code
- 'which' constants are the same as for SetPersonScript()

GetCurrentPerson()

- best when called from inside a PersonScript handler
- it will return the name of the person for whom the current script is running

QueuePersonCommand(name, command, immediate)

- adds a command to the person's command queue
- the commands are:

- + COMMAND_WAIT
- + COMMAND_FACE_NORTH
- + COMMAND_FACE_NORTHEAST
- + COMMAND_FACE_EAST
- + COMMAND_FACE_SOUTHEAST
- + COMMAND_FACE_SOUTH
- + COMMAND_FACE_SOUTHWEST
- + COMMAND_FACE_WEST
- + COMMAND_FACE_NORTHWEST
- + COMMAND_MOVE_NORTH
- + COMMAND_MOVE_EAST
- + COMMAND_MOVE_SOUTH
- + COMMAND_MOVE_WEST

(note: these *might* change in a future release
'immediate', if true, will execute the command go right away
if false, it will wait until the next frame)

ClearPersonCommands(name)

- clears the command queue of sprite with the 'name'

IsPersonObstructed(name, x, y)

// returns true if person 'name' would be obstructed at (x, y)

SetTalkActivationKey(key)

- set key used to activate talk scripts

SetTalkDistance(pixels)

- set distance to check for talk script activation

Spritesets

LoadSpriteset(filename)

– returns a spriteset object from 'filename'. If Sphere is unable to open the file, the engine will give an error message and exit.

spriteset_object.getNumDirections()

– returns the number of directions that the spriteset has

spriteset_object.getNumFrames(direction)

– returns the number of frames in a given direction

spriteset_object.getFrame(direction, frame)

– returns an image object of a given frame from the direction

Sounds

LoadSound(filename)

– returns a sound object from 'filename'. If Sphere is unable to open the file, the engine will give an error message and exit.

sound_object.play(repeat)

– plays the sound. repeat is a boolean (true/false), that indicates if the sound should be looped

sound_object.stop()

– stops playback

sound_object.setVolume(volume)

– sets the volume for the sound (0–255)

sound_object.getVolume()

– returns the sound's volume (0–255)

sound_object.getLength()

– returns the total length of the sound. The number returned is a arbitrary number, depending on the sound plugin.

sound_object.getPosition()

– returns the position in the sound currently being played. The number returned is an arbitrary number, depending on the sound plugin.

sound_object.setPosition(position)

– allows you to set the position currently being played. Since the sounds have no uniform measurement due to the audio plugins, be careful on how you set the position of the sound.

sound_object.isPlaying()

– returns true if the sound is currently playing

Fonts

GetSystemFont()

– returns a font object of the font that the engine currently uses.

LoadFont(filename)

- returns a font object from 'filename'. If Sphere is unable to open the file, the engine will give an error message and exit.

font_object.setColorMask(color)

- Sets the colors mask for a font (see ApplyColorMask)

font_object.drawText(x, y, text)

- draws 'text' at x, y with the font

font_object.drawTextBox(x, y, w, h, offset, text)

- draws a word-wrapped text at (x, y) with the width w and height h. The offset is the number of pixels which the number of pixels from y which the actual drawing starts at.

font_object.getHeight()

- returns the height of the font, in pixels

font_object.getStringWidth(string)

- returns the width of a given string, in pixels

Window Styles

GetSystemWindowStyle()

- returns a windowstyle object of the windowstyle that the engine currently uses.

LoadWindowStyle(filename)

- returns a windowstyle object from 'filename'. If Sphere is unable to open the file, the engine will give an error message and exit.

windowstyle_object.drawWindow(x, y, w, h)

- draws the window at (x, y) with the width and height of w and h.

Note that window corners and edges are drawn outside of the width and height of the window.

Images

GetSystemArrow()

- returns an image object of the System Arrow that the engine currently uses.

GetSystemUpArrow()

- returns an image object of the System Arrow(up) that the engine currently uses.

GetSystemDownArrow()

- returns an image object of the System Arrow(down) that the engine currently uses.

LoadImage(filename)

- returns an image object from 'filename'. If Sphere is unable to open or read the image, the engine will give an error message and exit. The

image type that the engine supports are either PCX, BMP, JPG, and PNG.

`GrabImage(x, y, w, h)`

– returns an image object from a section of the video buffer defined by the parameters

`image_object.blit(x, y)`

– draws the image onto the video buffer at x,y

`image_object.blitMask(x, y, mask)`

– draws the image into the video buffer, except that the color passed as 'mask' tints the image

`image_object.rotateBlit(x, y, radians)`

– draws the image into the video buffer, except that the image is rotated in anti-clockwise in radians, which have a range of $0-2\pi$. (x,y) is the center of the blit.

`image_object.zoomBlit(x, y, factor)`

– draws the image into the video buffer with zooming, with the scaling depending on factor. Normally a factor of 1 will blit a normal looking image. Between 0 and 1 will shrink the image. Any values greater than 1 will stretch the size of the image.

`image_object.transformBlit(x1, y1, x2, y2, x3, y3, x4, y4)`

– draws the image into the video buffer with "transformation", where (x1, y1) is the upper left corner, (x2, y2) the upper right corner, (x3, y3) is the lower right corner, and (x4, y4) is the lower left corner.

`image_object.width`

– the width of the image

`image_object.height`

– the height of the image

Surfaces

`CreateSurface(width, height, color)`

– returns a surface object with width × height, filled with color

`LoadSurface(filename)`

– returns a surface object with an image with the 'filename'

`GrabSurface(x, y, w, h)`

– returns a surface object captured from an area of the video buffer, at (x, y) with the width w and height h.

`surface_object.blit(x, y)`

– draws the surface to the video buffer at (x,y)

`surface_object.blitSurface(surface, x, y)`

– draws the surface into the video buffer at (x,y)

surface_object.createImage()

- returns an image object from the surface object

surface_object.setBlendMode(mode)

- pass it with either BLEND or REPLACE

REPLACE mode will make the surface erase the pixels needed when doing a drawing operation (setpixel, line, rectangle...)

BLEND will blend the pixels needed when doing a drawing operation

surface_object.getPixel(x, y)

- returns the color of the pixel at (x,y)

surface_object.setPixel(x, y, color)

- sets the pixel at (x,y) to 'color'

surface_object.setAlpha(alpha)

- sets the [ALPHA] of the surface

surface_object.line(x1, y1, x2, y2, color)

- draws a line onto the surface starting from (x1, y1) to (x2, y2) with the color

surface_object.rectangle(x, y, w, h, color)

- draws a filled rectangle onto the surface at (x, y) with the width w and height h with the color

surface_object.rotate(radians, resize)

- rotates the surface anti-clockwise with the range 0 - 2*pi. The resize flag is a boolean to tell the engine to resize the surface if needed to accommodate the rotated image.

surface_object.resize(w, h)

- resizes the surface images. This does not stretch or shrink the image inside the surface.

surface_object.rescale(w, h)

- stretches or shrinks the surface to the new width w and height h

surface_object.flipHorizontally()

- flips the surface horizontally

surface_object.flipVertically()

- flips the surface vertically

surface_object.clone()

- returns a surface object, which is a copy of this surface object

surface_object.cloneSection(x, y, w, h)

- returns a new surface object with the height and width of h and w, with part of image at (x,y) from the surface_object with the width w and height h.

Animations

LoadAnimation(filename)

- Returns an animation object with the filename. If Sphere is unable to open the file, the engine will give an error message and exit. Sphere supports animation formats of .flic, .fli, .flc and .mng

animation_object.getWidth()

- returns the width of the animation

animation_object.getHeight()

- returns that height of the animation

animation_object.getNumFrames()

- returns the number of frames the animation contains

animation_object.getDelay()

- returns the delay between frames, in milliseconds

animation_object.readNextFrame()

- readies the next frame for drawing

animation_object.drawFrame(x, y)

- draws the current frame into the video buffer

Files

GetFileList()

- returns an array of strings, which contains the filenames that resides in the "save" directory of the game.

OpenFile(filename)

- returns a file object with the filename. The file is created/loaded from the "save" directory of the game. Note that any changes in the keys will not be saved until the file object is destroyed.

file_object.read(key, default)

- reads a value from the key

the value type returned depends on the default value.

+ if the default is a number, read will return a number.

+ if the default is a text or string, read will return a string.

+ if the default is a boolean, read will return a boolean

+ if the key is not present in the file, it will return the default value.

file_object.write(key, value)

- writes a value (string, number, boolean) to the file under the key name

Raw Files

OpenRawFile(filename)

- opens a file with the filename. The file **must** exist and reside in the "other" directory of the game otherwise Sphere will give an error. If the file is opened successfully, the function will return a rawfile object.

rawfile_object.setPosition(position)

- sets the position that the file will be read from

rawfile_object.getPosition()

- returns the current position which the data is read from

rawfile_object.getSize()

- returns the number of bytes in the file

rawfile_object.read(num_bytes)

- reads the number of bytes that is specified by num_bytes. It will create and return an array of data the rawfile object has read. The array of data are numbers representation of each byte (0-255). Note that if the number of bytes that will be read, exceeds the filesize from the current position, it will only return an array of data of that is actually read.

This I copied from my Reference tutorial page. And is not written by me but included in the Sphere 0.90 package. This was written by Rizen and edited and updated by Darklich and AegisKnight. I did some formatting and editing too. (Eugene Siow)

Lastly to sum things up, functions are the most basic element of a Sphere program. Sphere is function driven, unlike VB which is object driven (object-oriented and whatever else). C++ is also object-oriented. So in Sphere, it is essential to realize what functions are and how they are used. If you can't master this, you'll have to start the tutorial all over again or check out a better tutorial if you think my way of explaining sucks.

B. FAQ

Sphere Setup

Sphere works most of the time on most computers without needing you to configure any of your settings. Sometimes though, you just can't get Sphere to work on your computer and this can be frustrating. This might also cause you to lose interest in Sphere without even tried it and made a game in it. If you are one of these people, this tutorial might prove to be beneficial.

Before we get started lets talk about some technical details. Sphere is a powerful engine and is able to use a variety of different video drivers to run. Thus, you could configure your video drivers to optimize the performance of Sphere on your computer even if it already works on your computer.

To do so, run the config.exe file which resides in the same directory as the sphere.exe.

The first screen you see shows you the various video graphics drivers you can use. It is set by default to the standard16.dll or standard32.dll. These are the standard drivers and work well with most computers. The 32 bit driver supports 32 bit true colour graphcs and require quite a lot of system resources. However, graphics look better in 32 bit.

The sphere_gl.dll, is the Sphere Open GL driver which tends to run faster on most systems. If you have a graphics card that supports opengl and acceleration, use this driver.

Clicking on configure driver will display a dialog allowing you to configure the driver.

The input tab has an option which says use joystick that allows Sphere to be used with a joystick.

To further configure Sphere, run the edior, editor.exe and choose, File > Options... which will display an options dialog. This allows you to register the .rmp, map extension, .rss, spriteset extension, .rws, window style extension and .rfn, Sphere font extension with windows. Double clicking on any of these file types in explorer will then bring up the Sphere editor. This file association allows for easy editing of Sphere files in the future.

C. Resources

The manual was set up by Dn7 and put together by Dn7. I also wrote some content (about 33%), the rest is written by various people.

Resources

<http://www.aegisknight.com/>

Here I found info about some other project combining with sphere

<http://spheresuite.sourceforge.com/>

Many thanks, many tutorials.

<http://sphere.sourceforge.net/>

More tutorials and resources

<http://sphere.sourceforge.net/flik>

:D

People

Eugene Siow,
Rizen,
Flik,
Mister Y,
Aegis,
Me,
some people I have forgotten.

email: dragonnumber7@hotmail.com, or better, smederij@home.nl.

Channels

#sphere on irc.esper.net, put some quotes from it in it.

D. Compendium

The compendium is for quick search of 'stuff' you don't know, and get a quick answer. It's like a dictionary.

A.

- Alpha

Alpha for color means the $\text{Alpha} / 255 * 100\%$ translucent. 255 Means it's 'Opaque' and 0 is so you can see through it.

•B.

- Braces

Braces are the '{' and '}' they are used to put code between.

•

•C.

- C.

•

•D.

- D.

•

•F.

- Frame

A frame contains a picture, a frame is put in a list. Having one frame does not always make sense.

•