

---

# Lucidity Documentation

*Release 0.1*

**Mikus S**

**Jul 29, 2017**



---

# Contents

---

<b>1</b>	<b>Contents:</b>	<b>3</b>
1.1	Global variables . . . . .	3
1.2	Map making . . . . .	10
1.3	States . . . . .	17
1.4	Cards . . . . .	22
1.5	Ghosts . . . . .	29
1.6	Inbuilt Packages . . . . .	34
1.7	Modules . . . . .	50
1.8	Assets . . . . .	59



**Latest Version: Alpha 0.1B**

Get the latest version here:

- Windows

32Bit

64Bit

- Linux

All

- Mac

Coming soon.

Lucidity is an asymmetric game, combining platforming and turn based strategy.

Below is the documentation directed towards developers or enthusiasts willing to create maps and modules for lucidity.



## Global variables

### Functions

**getPackageAlt** ()

Returns **string \_PACKAGE** the current location of the file.

**getEnv** ()

Gets the current environment used for setfenv()

**copyFiles** (*string dir, string dest*)

copies files from one directory to the other, recursively.

**saveLog** ()

saves the log file to %appdata%/LOG

**RctToWH** (*userData image, int rows, int columns*)

converts rows and columns to frame dimensions.

**printStats** (*float x, float y*)

print statistics at x,y

**loopRange** (*var root, float range, string name, float speed, string mode*)

short hand for a recursive tween.

**parseColor** (*string t*)

used for parsing color from text as such; "255,255,255,255"

**checkPlayerCol** (*HC shape a, HC shape b*)

checks the shapes type and returns player and part and other

**tableLenght** (*table t*)

returns table lenght of a non-indexed table.

**findDistance** (*float x, float y*)

finds the distance between two points

**findDimensions** (*(HC shape or float x), float y, float x1, float y1*)

**drawRelease** ()

draws overlay of version

**incrementPos** (*float x, float y, float ofx, float ofy, float degree*)

allows for locking to grid of cordinates.

**showFPS** ()

shows fps on the screen

**inRange** (*float x, float item, float y*)

returns boolean if item is in range of x and y

**inRangeHigher** (...)

same as inRange, but returns true if it equals to higher.

**inRangeLower** (...)

same as inRange, but returns true if it equals to lower.

**hsvToRgb** (*float h, float s float v*)

converts HSV to RGB color space, returns table: {r,g,b}

**MinMaxFromTable** (*table*)

returns the maximum numeric value from a table.

**crop (image, imagex, imagey, fw, fh, scalex, scaley, directon) \*\***

crops an image and returns an image as result

**removeFromTable** (*table, item*)

**mergeTable** (*table1, table2, mode*)

Modes: [add, force] add will add the values while force will set it.

**getFactors** (*float number*)

**nokey** ()

empty key

**keyboardpress** (*key, block*)

key is the name of the key, block overrides LOCKCONTROLS

**mousepress** (*key, block*)

**gamepadpress** (*joystick, key, block*)

experimental.

**findMap** (*name*)

finds a map on the top layer, aka maps/ and inbuilt\_maps/

**saveGame** (*string name, optional map*)

saves the game.

**getSaves** ()

returns saves in format:

```
{
  name = string,
  mapName = string, Note: only the first line without path.
  map = string,
  playTime = {h=h,m=n.s=s} Note: broken.
  cards = {} Note: players cards, table of names.
  mobs = {}, Note: players mobs, table of names.
  date = {date,time},
}
```

**saveGameShort** (*name*)

saves a map to playerData.children\_maps.

**loadGameShort** (*name*)

returns the playerData.children\_maps table for the map of the name var.

**loadGame** (*string name, bool dontswitch*)

dont switch prevents the switching of state, name is the savefile name.

**queryKey** (*string name, KeyControler c, function fn*)

returns bool if found, bool isDown, int position in table

**deepCopy** (*table*)

copies a table, please use deepCopy2 instead

**deepCopy2** (*table*)

properly copies a table including metatable, recursive.

**reverseTable** (*table*)

Reverses the table index.

**precentOf** (*int full, int current*)

same as findPercent, kept for legacy.

**findItem** (*mode, (bool or string) name*)

finds an item of Mode with the name of Name, searches alongside maps so if mode is “ghost” it will search at maps/your\_map/ghosts and ghosts/

if name is bool then it will return a table all items of the same mode.

Otherwise it will return path, [set, setPath] = if you are looking for a card.

**getParents** (*string map, string(optional) tail*)

returns a table of parents for a map with string format.

**imagestencil** (*function fn*)

returns a function within which a shader is applied to render a stencil mask.

## Standard lib extensions

### **table.zsort(table t)**

sorts a table by the z index when a.zmap < b.zmap

### **math.round(int number,int decimalPlaces)**

## Data

### Colors

Contains defined colors, allows the use of setColor(color), eg setColor(colors.yellow)

defined colors are;

```
red,darkRed,  
green,darkGreen  
blue,darkBlue,cyan,  
lightBlue,paleBlue,  
yellow,orange,  
ambient,black,  
gray,lightGray,darkGray,  
violet,white
```

### **debug\_text**

table full of the text printed using print(t,t2,..)

### **crystals**

table full of crystals from the main menu.

basic,blue,cyan,green,inverted,pink,player,silver

structure is as follows

```
{  
  w = w, (width)  
  h = h, (height)  
  delay = delay,  
  anim = Animation,  
  data = {image,w,h,delay}  
}
```

### **effectColors**

Effect data used internally in battle mode. contains:

```
Poison,fire,curse,drown
```

### **stock\_effects**

contains stock effects.

```
{
  jump = {
    image image,
    Animation anim.
  }
}
```

**mColors**

contains colors used in battle mode.

**icons**

contains icons as seen here:

**playersMobs**

contains all of the player mobs for the current active run.

**playersCards****playerData**

```
{
  saveName = string,
  essence = int,
  playTime = string,
  children_maps = {},
}
```

**keys**

Long table of keys, described here:

**alt\_keys**

Long table of keys, alternative edition.

**misc**

Misc settings:

```
{
  showFps = bool
  scaleFactor = int
  cameraSlide = bool
  cameraSlideB = bool
  battleCamera = bool
  mouseLock = bool
  trapMouse = bool
  textSpeed = int -- Not implemented yet!!
  sfvol = int -- sound effect volume. may be nil at times
  bgmvol = int -- bgm vol, may be nil at times
  voicebol = int -- voice volume, may be nil at times
  mastervol = int -- may be nil at times.
  seenIntro = bool
}
```

**video** Video settings

```
{
  width = int
  height = int
  trapMouse = bool
  mouseVisible = bool
  flags = {
    fullscreenType = string
    fullscreen = bool
    vsync = bool
    highdpi = bool
    display = int
    borderless = bool
    srgb = bool
  }
}
```

## States

**Intro**

**Credits**

**Editor**

**Battle**

**PlayMap**

## Constructors

**net**

work in progress do not use.

**Flux**

flux instance. [Documentation](#).

**Gamestate**

hump gamestate. [Documentation](#).

**Timer**

hump timer, [Documentation](#).

**Vector**

hump vector, [Documentation](#).

**HC**

Hardon Collider, [Documentation](#).

**ui\_package**

the inbuilt UI,

**msgQue**

the message que for npc's and other

### **loadGhost**

ghost system, used for loading:

### **loadCards**

card system, used for loading:

### **keyControl**

controls the keys

**Profiler** the profiler, [Documentation](#).

**Tserial** packs table to string. [Documentation](#).

## **Other**

### **eCurrentMap**

The current map, full length string of the current map directory.

### **mapAdmin**

map admin module.

### **MaxCapture**

is part of the capture timer before engagement starts

### **playerMaskCategory**

the box2D category for player

### **uniBody**

uniBody font to be used alongside love.graphics.newFont

### **Glyphs**

Image font used for writing strange text

### **basicFrameColor**

### **basicButtonColor**

### **LOCKCONTROLS**

locks all the controls (bool)

### **\_RELEASE**

string of version

### **SHOWUI**

bool, shows or hides the ui, please use removeHud to remove HUD only works in play mode.

### **DEBUG**

bool, enables debug mode

### **PROFILER**

bool, allows to profile if P key is pressed

### **STATS**

bool, shows stats

## globals

the global gamestate.

## Map making

You can produce maps for lucidity through the editor.

This guide will help you to create your first map.

A tutorial video is in production.

## Basic controls

You can move using the W A S D keys and  to zoom,

Alternatively use the  button with either  to zoom in and  to zoom out.

Use the  icon or the G key to toggle grid.

Use the  icon or the R key to toggle ruler

You can move the mouse to the sides to the screen to drag it around, RTS style.

If you do not like this use the  icon to prevent it.

## Layers

The very basic principle of all maps of shapes and sizes is its layers.

Layers are a way to present depth and draw things atop and before the player.

To view your active layers go to the Editor from the main menu and click the  at bottom right

You will see one new layer colored in blue. Layers are color coded depending on what they do.

## Colors

Here is a table of colors and their meanings:

Color	Meaning
Blue	Has colision
Gray	None

More will be added as complexity of map making increases.

## Layer dialog

### What are those other buttons?

*UP* a layer, it moves a layer by one step.

*N*ew, makes a new layer above the current layer.

*M*erge, WIP doesn't work yet.

*D*elete, deletes a layer.

*P*roperties, views the properties of current layer:

```
Name: [Layer 1]
Mod X:    [0] Used for parallax, this is an offset based on camera position
Mod Y:    [0] Used for parallax, this is an offset based on camera position
Colision: [on/off] indicates that shapes on this layer will have physics shapes
Sound:    [on/off] Note:work in progress
Transp:   [255] Transparency of the layer.
```

When you are finished press the Apply button otherwise swap to another layer or close the layers dialog.

*DOWN* a layer, it moves a layer down by one step.

### Where to next?

Press the *N* button to create a new layer.

This layer will be your current layer, everything that you create will be placed on this layer unless you select another one.

So now we know about layers so lets move onto placing things in them.

## Shapes

To access to shapes take a look at the Dock at the bottom center of the window, you should see the following icons:



### Rectangle

Rectangles are fairly simple, press on the  icon and then  drag on the map and you should see a white rectangle form between the two points

Once you let go of the mouse the  will be finalised, however you can continue making rectangles until you 

### Circle

Circles are simple too, press on the  icon and then  drag on the map, it will make a circle from the mouse click.

## Polygon

Polygons are a little more complex, to create one click on the  icon and then  to place a point.

To move a point use  to set it into the “Edit” mode and move the points around, when you are done press  again.

To cancel the creation press  however to finish the process make a cycle by clicking on the last point to form a cycle.

---

**Note:** The colors of the points change depending on what you are doing, for example selected point will be yellow. If the polygon intersects then the whole shape will turn red so make sure it doesn't or you won't be able to form a cycle. Both convex and concave polygons are supported.

---

## Shape properties

**Go ahead and make one or two of the shapes you like, I recommend a simple rectangle**

At this point you may wonder how we can change the mundane white color to something more interesting.

Place your mouse atop of the rectangle until its edges turn red and 

A menu should pop up:

```
Change color
Set texture
Delete
Set Layer
Add Trigger
Remove Trigger
Properties
```

Press the Change color button to open a new dialog where you can change the color

Currently it is in text to rgb mode only so input the color like such: r,g,b,a

For red do:

255,0,0

Transparency works too, just try:

255,0,0,100

Now open the layer dialog from before and select the layer beneath the one we were working in.

Create a few new shapes atop of our old ones, afterwards play with the layers to see how they work I recommend that you move the layers up and down or even try parallax!

## Textures

As you have noticed the rectangle has an option to Set Texture, this is done from the Image interface that gets its images from various spritesheets.

The sprite will continuously wrap alongside the size of the shape in both X and Y axis.

Setting a texture will allow you to either mirror or reflect the texture, clicking on Set Texture again will remove the current texture.

## SpriteSheets

Spritesheets are complex however really convenient once you figure it out so I will try to keep it as simple as possible.

Before we do anything give the map a name and save it, if you pressed CTRL-S it will save it as New-<date><time>

To save the game press the “——” button to open up the drop down menu and go File -> save or CTRL-S

Now open the map directory through Map -> Open dir , this should automatically open the map directory.

Navigate your way to the “Image” folder and add your texture/spritesheet there, PNG is recommended.

**Now go back to Lucidity and go sprites -> New, this will open a menu with two lists, the first list is images in resources the other please select the name of the image you placed there.**

A new dialog will open with the following items:

Properties, Image preview with selection, list of sprites from this image and preview of the sprite.

To create a sprite you can use the properties window or the preview by dragging a square and moving it to the desired position.

---

**Note:** To create animations use the properties window to type out the properties of the animation(delay,(frame)start,(frame)finish) and when you hit *make* it will produce the animation in the preview.

---

Assign the sprite a name and then press *Add*, this will add the new sprite into the sprite list.

Once you are done press *Save*

## Images

Okay so now we have made a sprite sheet, lets create some cool images, press the  button.

Press on the name of your map and select the name of your spritesheet, you will see the sprites drawn there,  on

one to place them on the map.  again to finish placement, now you can move it around like anything else.

You may want to play around with this and try applying a sprite as texture on some shapes to see how it works.

## Selection tool

The selection tool is different from the other selection tools you have seen around, the main difference is that you can have as many of them as you want at the same time.

Press  and  drag a square the same way you create rectangles, but make sure that you are on the same layer that the items you want to select.

Release  and move the selection about, you may see that the selection causes the shapes to become out of shape, I recommend using the *lock* option from its *Menu* this option will lock the selection to grid.

Once you are happy with the changes you have made select the *Clear* option to remove the selection.

---

**Note:** Selection rectangles are NOT saved.

---

## Text

Text is the fastest way to deliver information, please use the Text tool to create it.

The text tool will automatically adjust to its size, so if you make additional spaces then the shape will be larger, the longer the text the longer the box. To edit the text use the Set text option, the dialog has options for;

color, font name,font size,text.

---

**Note:** Rich text is not implemented so please use fonts such as uniBody\_italic and so on as your font. Fonts can be found in your\_map/fonts/ [work in progress.] or define it in a script file as `_G[font_name] = love.graphics.newFont(location)`

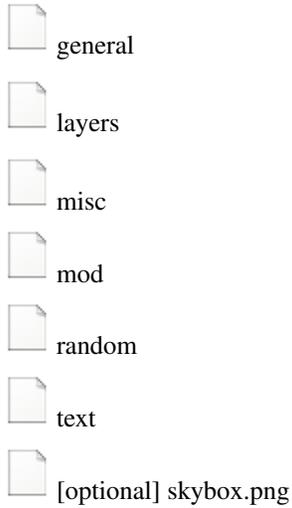
---

## File structure

Okay so here the serious stuff comes in, if you were uncomfortable with the previous parts I recommend that you revisit them before continuing.

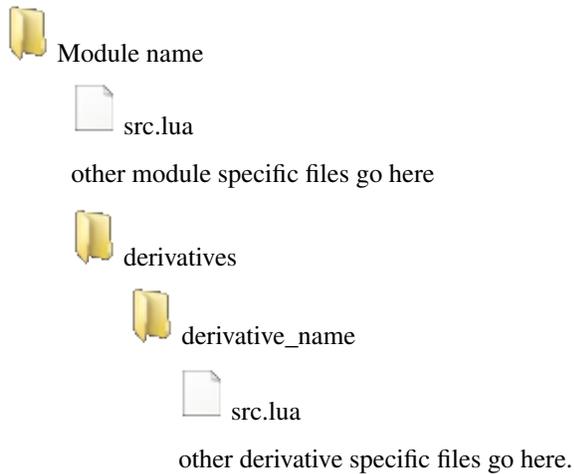
The map has the following structure:

-  Map\_name
  -  children
  -  image
  -  modules
  -  sounds
  -  spritesheets
  -  [optional] ghosts
  -  [optional] cards



## Modules

Modules go here with the following structure:



## Resources

Okay so the resources are rather simple, it contains all the resources.

This includes modules, images, spritesheets, skyboxes, sounds

place all of your resources at `%appdata%/Lucidity/resources` for global use across maps.

## Children

Children are one of the most important parts of the mapping process, it is responsible for saving space and resources and persistent gameplay.

Use the Zdoor derivative under the Station Module to automatically link and copy children.

To manually add a child map just copy a map to the `your_map/children` folder from your parent map.

**Note:** All of the resources between the child and parent are shared however if you keep any resources in the child map they will not be removed. So to properly create a child map make a blank map, and move it to *children* folder and then right click on your parent map in the *file -> load* menu and select your map.

Children can be added recursively!

---

## Triggers and scripts

All items, except some rare cases, have triggers multiple triggers available, these are events that are triggered based on

a condition. You can attach a trigger to a shape by  on it and *Add Trigger* then you will see an input box for code, you can type short scripts here.

It is highly inconvenient to use the in-game text editor I highly recommend to make a script file and place your scripts there to do this write a short script like this:

```
--The global table of the Gamestate is named _G in this case so you must define_  
↪things in the _G table  
--Otherwise the global variables are local, this means that you dont need to use_  
↪Local and you wont interfere with the globals.  
  
_G["HelloWorld"] = function()  
  print("Hello world")  
end
```

Save this as *myscripts.lua* in the *your\_folder/scripts/* folder, and press the *Utilities -> Reload scripts* button, check the console for errors! Now attach this script to a shape by *Add trigger* by typing *HelloWorld()* in the script menu and pressing *Apply*

Each of the triggers have different global variables returned from events and properties:

```
Player-in      none  
Player-out     none  
Hplayer-in   none  
Hplayer-out    none  
Load          none  
Draw          bool debug mode  
Update        float dt [delta time]
```

All of the triggers will have the *self* variable defined as the shape the trigger is called on.

The collision based events will require that you return *true* to continue executing after the player has colided, otherwise it will be a *Once only* event Pointers =====

To point at an object you need to assign it a name first, to do that:



on the object and *properties* then input a name and press *done*

This will add the item to the global table that you can access through scripts.

## Modules

Modules are a complex subject so I have assigned a large category to this and programing then so if thats what you are looking for please visit: <link>

To place a module use the **M**, this is a similar dialogue as that of the images so you can use it the same way.

Once you find the module you want select it and then its derivative, for example to make player you have to select:

Your\_map -> Player -> Yume

This will create Yume, which is a derivative of Player.

---

**Note:** Each of the modules have their own settings and their own way of doing things so please refer to the developers documentation on how to use them.

---

So to finish off this tutorial create Yume and place her in the middle of the map and press *file->run* to play the map.

## Skybox

Skyboxes are a rather simple implementation, you place the skybox.png in your map folder and then *Utilities -> skybox* and this will set the skybox.

if you want an ambient color instead of a skybox use *Utilities -> Ambient color* and type the color out.

## Scenes

Scenes are a different beast but operate on the same editor, the main difference is that they have a viewing range.

I calculated the maximum space that the scene can take and created simple placements of the sky and ground of the scene with Green and Blue respectively.

You can operate them like shapes and set texture to them but you cannot erase them.

There is a pedestal for each of the teams captains marked with a shape of its own, please use that to set an image for the pedestal.

Otherwise it is no different, please use the layers carefully though and make sure not to place things on the ground where the ghosts walk, they might trip.

## Additional help

You may require additional help so you can contact me on Lucidity's forums at <link>

## States

The states used are from [Hump Gamestate](#), so make sure to read its documentation before you start. To get the current gamestate use: `Gamestate.current()`

**Gamestates Editor,Play and Battle have the following in common**

### Variables

*x,y*

The current positions of the camera.

*current\_zmap*

the active zmap.

*scale*

*layers*

Layers module link.

*textureManager*

Texture manager module link.

*textureMapper*

Texture mapper module link.

*imageAdmin*

The image module link.

*shapeAdmin*

Shape module link.

*moduleAdmin*

Module Admin link.

*textAdmin*

Text module link.

*tool*

The current active tool.

*mainCanvas*

The main canvas where everything except UI is drawn upon.

*ui\_keyControler*

The ui key controller

*CamerBox*

The camera box HC shape.

*ui*

The ui\_package used.

## Functions

**worldPos** (*x, y*)

**Parameters** **x, y** (*numbers*) – Position on screen

**Returns** **numbers x,y** Position in the world.

**camPos** (*x, y*)

**Param** floats **x,y**: Position in the world

**Returns** **floats x,y** Position on the camera.

**render** ()

Draws all of the items on the mainCanvas.

**getTween ()**

**Returns** Flux group; The flux group used for tweens within the gamestate.

**parse ()**

**Returns table** Editor items including shapes and modules etc.

## Editor

**Editor:renderRuler ()**

**Returns** draws the ruler on the ruler canvas.

**Editor:getMousePos ()**

**Returns floats x,y** the mouse position.

**Editor:getMod ()**

**Returns floats x,y** modifiers for parallax from the current layer.

**Editor:remakeCamera ()**

Recreates the camera shape, use it if you are setting scale etc, no need to use it when changing position.

**Editor:navigation (KeyControler keycontrol1, keyControler keycontrol2, float dt)**

Controls all of the navigation.

**Editor:save (name)**

**Parameters name** (*string*) – saves the map with the name.

**Editor:run ()**

Runs the game into Play mode.

**Editor:load (name)**

**Parameters name** (*string*) – loads the map at name

**Editor:refreshImageList ()**

Refreshes the image list if it is open.

**Editor:getMapDir (tail)**

**Parameters tail** (*string*) – the tail string such as “/children”

**Returns string** directory.

**Editor:pickImage (function)**

**Param function** (*sheet, quad*), executed when the appropriate sprite is selected.

**Editor:pickLayer (function)**

**Param function** (*zmap, layer*), executed when the appropriate layer is selected.

**Editor:pickSound (function) -- WORK IN PROGRESS!**

**Param function** (*name, dir*), executed when the appropriate sound is selected.

**Editor:getUi ()**

**Returns** the ui package used in the state.

**Editor:newMap (name)**

**Parameters name** (*string*) – the name of the new map.

## Variables:

*dock*

the Dock ui portion.

*dropDown*

The drop down menu.

*quickSave*

The quick save timer if it reaches zero the game is saved.

*mx,my*

The mouse position.

*all\_active*

All of the active items, within and outside of CameraBox

## Play

Play state is the main state, it is the platformer part of the game.

## Functions

**Play:getPlayer ()**

**Returns** the active camera holder, aka the player.

**Play:getScenes (map, initial)**

**Parameters**

- **map** (*string*) – the map to query (optional).
- **initial** (*table*) – the initial table filled with maps or empty (optional)

**Returns table scenes** the table full of scene locations in string format.

**Play:OnBattleMode (team, player, ownTeam, cards)**

**Parameters**

- **team** (*table*) – table with Ghosts, max: 5.
- **player** (*table*) – the player instance
- **ownTeam** (*table*) – Optional, the players team.
- **cards** (*table*) – Optional, the players cards.

The table full of Ghosts should look like this:

```
{
  {name = string name
   dir = string directory
   _host = ghost -- Instance of loadGhost data}
}
```

**Play:save (string name, Team)**

**Params string name** The name of the save.

**Params Team** The team, same syntax as `OnBattleMode()`

**Play:tweenCapture()**

It is the capture tween, the effect that happens when you are about to enter the Battle mode.

**Play:clampCamera()**

**Returns Camera** The camera unique to the current game.

**Play:getCamera()**

**Returns Camera** The camera unique to the game, Doesn't block the existing one though.

**Play:releaseCamera()**

releases the camera.

**Play:switchMap(dir)**

**Parameters dir** (*string*) – the directory of the map.

## Camera

The camera is a new class created when you either `Play:getCamera()` or `Play:releaseCamera()`

- Variables
  - x,y* the map position of the Camera.
- Functions

**Cam:updateCamera()**

**Returns numbers x,y** new x and y positions in the world.

You must redefine this function to set the appropriate positions.

**Example usage:**

```
local game = Gamestate.current()

local Cam = game:clampCamera(true)
local finished = false
Cam.modx = 0
Cam.mody = 0
game.tweenGroup:to(Cam,2,{modx = 100})
function Cam:updateCamera(dt)
    local x,y = self.x + self.modx,self.y + self.mody
    return x,y
end

-- some where else when finish = true,

if finish then
    game.tweenGroup:to(Cam,2,{modx = 0}):oncomplete(function()
        game:releaseCamera()
    end)
end
```

## Battle

Battle is the RPG part of the game,

Battle has little to no functions at the moment, but please note that is soon subject to change.

As of right now it has little modding compatibility and will soon be rewritten.

## MainMenu

The main menu doesn't share the afro mentioned functions but instead has the following:

**pralaxPos** (*factor*)

**Parameters** **factor** – the parallax factor.

**Returns numbers** **x,y** the new position.

As you may have guessed MainMenu is currently off-limits too, however that is subject to change.

## Options and Menu

Options is a sub-state of Menu while Menu itself is a state that is found across the states.

Menu will be used in most states to allow for navigation, just press ESC.

## Cards

### File structure



- draw.lua

This contains code to draw the cards of the same set, it is used together with load.lua to draw the card.

**Example:**

```

local _PACKAGE = getPackage(...)
local image = love.graphics.newImage(_PACKAGE.."../.."image.png")
local func
local Shader = love.graphics.newShader("s/outline.glsl")
local TitleF = love.graphics.newFont(uniBody,12)
local TextF = love.graphics.newFont(uniBody,8)
local largeFont = love.graphics.newFont(60)
func = function(card)
    local CardColor = card.cColor or {105,105,255}
    local oldFont = love.graphics.getFont()

    local lg = love.graphics
    local rareCol = card.rarityCol
    local w,h = image:getDimensions()

    -- Corner crop for the beautiful coloring
    local stencil = function()
        local h = h - 2
        lg.polygon("fill",0,0, 13,0, 0,13 )
        lg.polygon("fill",w,0, w - 13,0, w,13)

        lg.polygon("fill",0,h, 13,h, 0,h - 12 )
        lg.polygon("fill",w,h, w - 14,h, w,h -14 )
    end

    -- coloring only the sides so crop needs to be applied here

    lg.setInvertedStencil(stencil)
    lg.setColor(rareCol)

    -- drawing a rectangle with rarity color.

    lg.rectangle("fill",1,1,w - 2,h - 2)
    lg.setColor(255,255,255)
    lg.setInvertedStencil()

    -- if its not an animation draw the image, otherwise draw the
    ↪animation.

    if not card.activeAnim then
        lg.draw(card.image,20,16)
    else
        card.activeAnim:draw(19,16)
    end

    lg.draw(image,-1,-1)

    -- drawing of text happens here.
    lg.setFont(TitleF)
    -- sets new font.

    local text = card.name

    -- retrieves the card name

    local tw = TitleF:getWidth(text)
    local th = TitleF:getHeight()
    local w,h = 146,18

```

```

lg.setShader(Shader)

lg.push()
lg.scale(1.01)
love.graphics.printf(text,16 + w/2 - tw/2,146 + h/2 -th/2 - 3,tw)
lg.pop()

-- shader does some recoloring on the text to make it stand out a
↳ little.

Shader:send("color",{0,0,0})
lg.setShader()

lg.setColor(CardColor)

-- print the text again this time as overlay of the shaded text.

love.graphics.printf(text,16 + w/2 - tw/2,146 + h/2 -th/2 - 3,tw)
lg.setFont(TextF)
local x,y = 26,168
local w,h = 127,42

-- print the description in its appropriate box.

local text = card.desc
lg.setColor(255,255,255)
love.graphics.printf(text,x,y,w,"center")
lg.setFont(oldFont)
lg.setColor(255,0,0)

lg.setColor(255,255,255)

end

return func

```

- load.lua

Load contains information required such as description,name and functions of the card.

**Example:**

```

local _PACKAGE = getPackageAlt(...)
local image = love.graphics.newImage(_PACKAGE .."/image.png")
local card = {
  name = "Healing",
  desc = "targets team \n heals [30%] of max hp each",
  cost = 10,
  rarity = 1, -- out of 10
  target = "team", -- team, none, single
  onUse = function(target,stack,item,areal,area2) -- Target is the
↳ target :: stack: {team1, team2, finished} :: item is the item that you
↳ can draw with, it will allow you to draw things to screen
    for i,v in ipairs(target) do
      v:applyDamage((v.maxHp/100)*30,"heal")
    end
    stack[4] = true
end,
end,

```

```

externalUse = function()
  for i,v in ipairs(playersMobs) do
    local hp = v.info.hp
    local maxHp = v.info.maxHp
    v.info.hp = hp + (maxHp/100)*30
    if v.info.hp > maxHp then
      v.info.hp = maxHp
    end
  end
end,
qualifier = function(stack) -- stack is same as above
  return true
end,
image = image,
anim = {}, -- if animation then place here, else will display image
↳without anim. Manage states using functions above.
}

return card

```

## Sets

To lower the workload of creating new cards a *Set* system was created.

The set system works by providing a way to draw the base of the cards so that it doesn't have to be duplicated across.

It works by its main **draw.lua**, it is a function that accepts *Card* as a Variable.

You can draw any amount of images and perhaps none at all.

## Function

The cards have two types of functions: *onUse* and *externalUse*.

- onUse

This is used in Battle mode. passes: target, stack, item, area1, area2

- externalUse

This is used in the Card selection in Play mode.

- Qualifier

The qualifier is the function that either allows or prevents the user from using a card.

- Stack

Stack is a table with both teams and the finishing condition which is **!! Stack[4]**

### Advanced Example:

```

onUse = function(target,stack,item,area1,area2) -- Target is the target :: stack:
↳{team1, team2, finished} :: item is the item that you can draw with, it will allow
↳you to draw things to screen
  -- Area polygon is only there if target = "team". ITs the target area for
↳the selected team in points eg [x,y,x1,y1,x2,y2]
  -- the draw item has the following func: drawBefore() draw() update()
  local element = "ice"
  -- for i ==1,10 do

```

```

        -- local x,y = math.random(0,)
    -- end
    local x,y = areal:bbox()
    local aw,ah = findDimensions(areal)
    local anims = {}
    local grid = {}
    for x=1,aw,w do
        for y=1,ah,h do
            local block = {x,y}
            table.insert(grid,block)
        end
    end
    local no = 0
    local bno = 0
    for i,v in ipairs(grid) do
        local c = love.math.random(1,3)
        if c == 1 and not v[3] then
            bno = bno + 1
            local anim = icePick:addInstance()
            anim:setMode("once")
            anim._px = v[1]
            anim._py = v[2]
            local dx,dy = x + v[1],y + v[2]
            local speed = 1000
            local distance = math.sqrt((dx + i)^2 + dy^2)
            Timer.tween(distance/speed,anim,{_px = dx,_py = dy},"in-quad",
↪function()
                no = no + 1
                if no >= #anims then
                    for i,v in ipairs(target) do
                        v:applyDamage(20,element)
                    end
                    stack[4] = true
                end
            end)
            anim._resG = grid[i]
            grid[i][3] = true
            table.insert(anims,anim)
        end
    end
    item.drawBefore = function()
        for i,v in ipairs(anims) do
            if (v._px ~= x + v._resG[1] or v._py ~= y + v._resG[2]) ↪
↪then
                v:draw(v._px,v._py)
            end
        end
    end
    item.update = function(dt)
        for i,v in ipairs(anims) do
            v:update(dt)
        end
    end
end,

```

- **item** Item has the following functions:

*drawBefore()*

*draw()*

*update()*

## Graphics

When producing graphics take in consideration the set system, it is your friend, use it well.

As an artist you should look at the different card designs and draw up similar features and then categorise them.

These will be your Sets.

After you have drawn this write a simple draw script, and extract all the unique parts of the card, this will go into the new Image.

This image is now your unique card part of the new set that you have made, this way you save both space and effort when adding new cards.

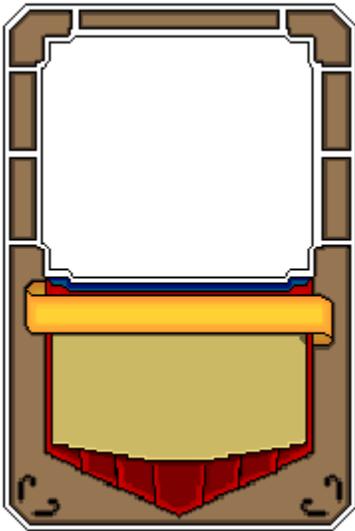
---

**Note:** When designing cards keep in mind the somewhat limited space for writing text.

---

### Example

Set picture:



Unique card picture:



Combined:



Gaps are to be filled by computer generated graphics, primarily the rarity colors on the sides.



---

**Note:** You can have as many images and computer graphics as you wish but DO NOT exceed the image size limit of: 180 x 267

---

Animations are welcome too but untested as of yet.

## Using your card

---

**Note:** The name of your card in playersCards and saves will match that of your folder name.

---

To use your card you must at first load it in, however the easiest way to test it is to play skirmish.

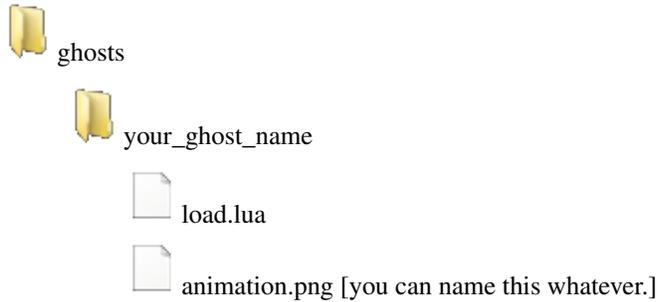
To access skirmish, make a save in the real house of the erisIntro map.

Another way to test this out is to edit the `playersCards` table using `loadCard` module, but I would recommend against this for now.

## Ghosts

### File structure

The file structure for ghosts is simple:



---

**Note:** The name of your ghosts as seen in `playersMobs` and `saves` will correspond to that of the folder name.

So if you have a folder called `Avictus`, you can do `loadGhost:byName(Avictus)` to get the ghost.

the `loadGhost` system is NOT case sensitive.

---

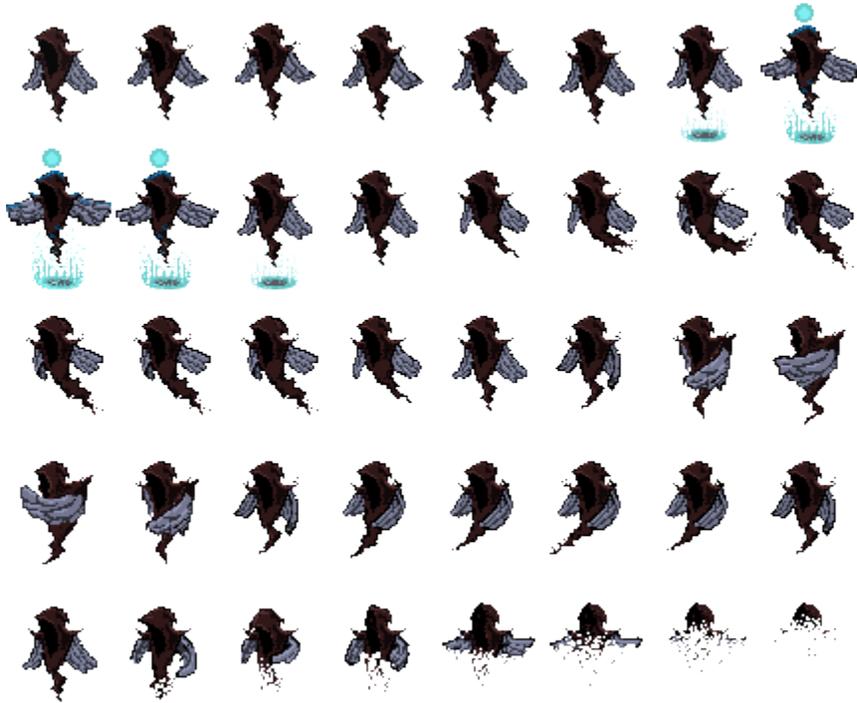
### Animation

---

**Note:** The maximum dimensions for a single frame is `94 x 94`, however this limit is not enforced but instead highly encouraged, ghosts above this resolution may escape the screen.

---

Here is the animation for `Avictus`:



The animation is coded like this:

```

local w,h = RCToWH(image,5,8)
-- we have 5 Rows and 8 columns, hence we divide the image by those values using the
↳convenient RCToWH function.

states = {
  {
    name = "run",
    anim = {w,h,0.14,13,15}, -- from frame 13 to 15.
  },
  {
    name = "attack",
    anim = {w,h,0.09,20,26}, -- from frame 20 to 26
  },
  {
    name = "death",
    anim = {w,h,0.09,33,40}, -- from frame 33 - 40
  },
  {
    name = "cast",
    anim = {w,h,0.14,6,12}, -- from frame 6 to 12
  },
  {
    name = "idle",
    anim = {w,h,0.14,1,6}, -- from frame 1 to 6
  },
},

```

**Note:** Animations accept Delay as their parameter, to convert this to FPS use 1/fps, for example if you want your animation at 30 fps you do 1/30 for delay.

## States

As you may have noticed animations are divided into states with respective names, there are the following states:

*run* When you move about.

*attack* When you attack.

*death* When you die.

*cast* When you cast a spell.

*idle* When you remain still.

*x\_change* When you change direction.

---

**Note:** You may see a *func* variable as part of the states, this is currently work in progress to allow for CG when drawing the state, but its going to change to allow for *update* and *draw*,

---

## Characteristics

All of the characteristics are kept within the *info* table:

### Example:

```
name = "Avictus",    -- Name on card
maxHp = 100,        --      100 or higher
maxMp = 100, -- 100 or higher
-- Below all stats can go up to 100
def = 5, -- defence
atk = 10, -- attack
luck = 10, -- the lower the luck the higher the chance of capture.
level = 3, -- the initial level.
sourceCrystal = "player", -- the source of the mob, you can leave this empty,
↳currently it doesn't do anything.
stillFrame = 1, -- the frame that shows up when you see the VS screen before battle.
story = "A guardian of sorts", -- The story
rarity = 4, --super common 1 to 10 is rareness (how hard it is to capture) try to
↳keep a good economy here, eg 1 = 30%, 2 = 25%.3 = 15%,4 = 10%,5 = 7%,6 = 5%, 7 = 3%,
↳ 8 = 2.5% , 9 = 1.5%, 10 = 1%
element = "divine", -- The element that shows up on the card when you capture it/buy
↳it.

weak = { -- is weak to, takes more damage from.
    "fire",
    "attack",
},
strong = { -- takes less damage from.
    "divine",
},
invin = { --Invincibility, you don't lose a heart with this
    "dark",
},
```

## Elements

The elements available are:

*attack*

*fire*

*water*

*ice*

*wind*

*divine*

*dark*

The following types are for healing or restoring mp

*heal*

*restore*

## Attacking

You may want your Ghost to attack the opposite team, not just with the usually physical attack but instead via *Skills* to do this you need to script a skill.

This process can be easy or challenging, so please consider it carefully when designing a mob.

### Example:

```
skills = {
{
  name = "Curse",
  desc = "Curse[10 ATK] a target for 3 turns",
  cost = 25,
  -- damage is used for the AI, in this case it should be 10*3 so 30.

  damage = 0,

  type = "dark",

  target = "foe", -- targets: fTeam, team, self, friend, foe

  func = function(target,stack) -- Target is enemy -- tables is the damage meter --
↳stack: {self, team1, team2, finished}
    local mob = stack[1]
    mob:cast()
    -- On Cast is called when the cast animation is finished.
    mob.OnCast = function()
      local no = 3
      local i = 0
      target.effect = "curse"
      -- in this case we set the effect to "curse", and we damage the enemy for
↳3 turns.
      target.OnChangeTurn = function()
        local dmg = 10
        local type = "dark"
        if i >= 3 then
          target.effect = nil
          target.OnChangeTurn = nil
        end
        i = i + 1
      end
    end
  end
}
```

```

        return dmg,type
    end

    -- this is where we say that we finished the skill and we should hand the
    ↪control back to the player.

    stack[4] = true
    mob.OnCast = nil
end
end,
}

```

**Note:** I am planning to include CG hooks to this, so you can draw fancy stuff as part of skills.

## Effects

Effects on their own do not do anything, so instead please apply damage yourself.

Here are the types of effects:

*poison*

*fire*

*curse*

*drown*

## Healing

Healing is little to no different than Attacking.

### Example:

```

{
  name = "Heal",
  desc = "Heals [50%] of current HP of the target.",
  cost = 45,
  damage = 0,
  type = "heal",
  target = "friend",

  func = function(target,stack)
    local mob = stack[1]
    mob:cast()
    mob.OnCast = function()
      target:applyDamage((target.hp/100)*50,"heal")
      stack[4] = true
      mob.OnCast = nil
    end
  end,
},

```

### Play mode

Adjustments for play mode are made automatically, a physics shape is generated based on the size of your frame, this means that the bigger the frame the larger the shape so please take that in consideration.

If you make your frame too large then the ghost may find it difficult to move about or glitch, that is if you exceed 94 x 94 soft limit.

### Using your ghost

To use your ghost you can use the NPC module and set the NPC name to that of your ghost.

Then place Yume and start the game, the AI should find its way towards you.

The AI is fairly basic but it will attempt some dangerous jumps so please be vary of it.

If you want to set the team yourself then use the save for the Skirmish mode as placed in the real house of the erisIntro level.

### After note

Some parts of the Ghost system are still under construction but the syntax of further builds is likely not to break backward compatibility.

## Inbuilt Packages

The inbuilt packages are used only internally but feel free to use them although please beware of modifying them as it will have global effects.

### User interface

The user interface can be accessed in two ways:

Use this to create a new UI package, use this in exceptional circumstances.

**ui\_package : new ()**

**Returns ui\_package**

OR

This is the main method of accessing UI

*Gamestate.current().ui*

### Parenting

The UI is based on a parent and child system, inother words if you create a button it is either the child of the UI itself or a frame or another item.

The use of this system should come intuitively.

All of the ui elements with the exception of a few follow this syntax:

```
ui:add<Item>(parent, x, y, width, height, ...)
```

**Example:**

```
-- Creating a frame with a button on it.
local width,height = 200,200
local Frame = ui:addFrame(nil,400,300,width,height)

local doneButton = ui:addButton(frame,5,20,100,100)
doneButton:setText("Done")
-- this will create a button at 405 and 320; the 5,20 coords are relative.
doneButton.OnClick = function()
    -- this will remove the Frame AND the button.
    Frame:remove()
end
```

The parent system is cumulative meaning that you can have a parent who's child is also a parent and so on, this is also used internally especially in lists.

**Items**

Here are all the currently implemented items and their functions.

- Frame

**ui:addFrame([parent or name], x, y, width, height, [parent])**  
**Returns Frame** the frame item.

**frame:addCloseButton(function)**  
**Params function** the function executed when the X button is pressed.

**frame:setShape(shape)**  
**Params HardonCollider shape** the shape of the new frame.

**frame:setImage(image)**  
**Params love2DImage** the image to draw on the frame

**frame:setTooltip(toolTip, delay)**  
**Params toolTip** the pre-created tooltip  
**Params number delay** time before the tooltip shows up.

**frame:addTooltip(text, delay)**  
**Params string text** the text the tooltip will display  
**Params number delay** time before the tooltip shows up

**frame:setSize(width, height)**

**frame:setTexture(image.wrap)**  
**Params Love2DImage** the image for the texture.  
**Params wrap** the wrap mode, see: [Wrap Modes](#).

**frame:setHeader(name, width, height, color)**  
**Params table color** color = {r,g,b,a}

**Important Variables:**

dontDrawSelf ; boolean, prevents the frame from being drawn.

**Events:**

*OnDrawTop(self)*

- Button

**button:setTooltip(toolTip, delay)**

Params **toolTip** the pre made tooltip

**button:addTooltip(text, delay)**

Params **string text** the text

Params **delay** the time required for the tooltip to appear.

**button:setShape(shape)**

Params **HardonCollider shape** the shape of the new button

**button:setSize(width, height)**

**button:setTexture(image.wrap)**

Params **Love2DImage** the image for the texture.

Params **wrap** the wrap mode, see: [Wrap Modes](#).

**Important Variables:**

dontDrawB ; boolean, prevents drawing of image on button.

dontDrawC ; boolean, prevents drawing of collision.

dontDrawText ; boolean, prevents drawing text.

dontDrawActive ; boolean, prevents drawing active/inactive gestures.

- Text

**ui:addText([parent], x, y, text, [font], [color])**

Params **font** love2d font for the text.

**text:setText(text, [font], [color])**

**Events:**

DrawPrefix(x,y)

DrawSuffix(x,y)

- Costum

Experimental, only use if you absolutely need to.

**ui:addCostum(x, y, colision)**

Only has *draw* , *getColision* and *update*

- Tooltip

**ui:addTooltip([parent], text, mode, [font], [color])**

**tooltip:setCenter(cx, cy, px, py)**

Params **numbers cx,cy** the center x and y

Params **numbers px,py** the parent x and y

**tooltip:setPoints()**

Resets the points.

- Menu

**ui:addMenu([parent], x, y, [table])**

Params **table** table filled in as such:

```
{ {"Run",function() self:save() self:run() end,10} }
-- Syntax:
{ {Text,function,gap} }
```

**menu:addChoice(text, function)**

**menu:addItem(item, function)**

**Params ui\_item item** A UI item to be an item within the list.

**menu:clear()**

**menu:refresh()**

- Slider

**ui:addSlider([parent], x, y, width, height, [mode])**

**Params string mode** horizontal or vertical, horiz or vert for short.

**slider:getFloat()**

**Returns** gets the float value in terms of percent of.

**slider:setFloat(d)**

**Params float d** the float number of d, use to set the current percentage.

**slider:setRange(min, max, isInteger)**

**Params bool isInteger** if you want integer only values.

**slider:getStepSizePX()**

**Returns number** the number of pixels per step.

**slider:getValue()**

**Returns number** the current value so if min = 100, max = 200 will return value in-between.

**slider:setValue(val)**

**Params number val** the value that lies within min and max set earlier.

**slider:OnMove(x, y)**

**Event:**

*OnChange(value)*

- Textinput

**ui:addTextinput([parent], x, y, width, height, text)**

**textinput:getLocalText()**

**textinput:getPosFromCords(x, y)**

**Params numbers x,y** the local coordinates

Sets the position of the indicator at X and Y.

**textinput:resetPos()**

**textinput:setMultiline()**

Enables multi-line.

**textinput:getText()**

**Returns text** the concatenated text.

**textinput:setAllowed(a)**

**Params table a** the table full of strings you don't want to see in your input.

**textinput:moveLine(bool)**

**Params** `bool` false for down, true for up

`textInput.setText(text)`

**Params** `string text` the text as a long string.

**Events:**

*OnChange(text)*

- List

`ui.addList([parent], x, y, width, height, [items], [mode])`

**Params** `bool [mode]` True if you want horizontal mode.

`list.setHeader(text)`

`list.addChoice(text, function)`

`list.addItem(item, position)`

**Params** `ui_Item item`

`list.refresh()`

`list.setMode(horizontal, vertical)`

**Params** `booleans horizontal, vertical`

`list.getStencil()`

**Returns** `function`

`list.goMax()`

Sets the position to maximum.

`list.clear()`

`list.getLeftOver()`

**Returns** `number` the leftover space at either the sides or top and bottom.

`list.removeItem(item)`

**Params** `ui_item item`

- Shared

`<item>.getColision()`

**Returns** `HardonCollider shape`

`<item>.addItem(item)`

**Params** `ui_item item`

`<item>.setInactive(t)`

**Params** `bool t`

`<item>.moveTo(x, y)`

`<item>.remove()`

**Events:**

*OnUpdate(dt)*

*OnDraw(self)*

*OnFocusLost()*

*OnFocus()*

*OnMove(x,y)\*<sup>0</sup>*

---

<sup>0</sup> This might be used internally so be careful when assigning.

---

**Note:** Some items have internal items:

Slider: sliderButton [button]

Menu, List, textInput: mainFrame [frame]

---

- Base[UI]

**ui:getItemCount ()**

**ui:clear ()**

**ui:getZorder ()**

**ui:draw (debug)**

**ui:update (dt)**

**ui:mousepressedList (x, y, button)**

**ui:textinput (t)**

**ui:getmxb ()**

**Returns HardonCollider\_shape** the shape used for the mouse.

**ui:init (Collider)**

**ui:getFont ()**

**Returns love2Dfont**

**ui:setMousePos (x, y)**

**ui:setTitleIncrement (degree)**

**ui:setLeftClick (main, alt)**

**Params booleans main,alt** the main and alt left clicks.

Run at update.

**ui:setRightClick (main, alt)**

**Params booleans main,alt** the main and alt right clicks.

Run at update.

**ui:setMiddleMouse (down, up)**

**Params booleans down,up** the down and up for middle mouse

Run at update.

**ui:setFont (font)**

**Params Love2dFont font**

---

**Note:** Some internal use functions are omitted.

---

**Important Variables:**

dontDraw = boolean; do not draw self.

inList = boolean; is it in the list?

## Editor specific

### Module admin

Module admin primarily designed for internal usage, please use with care!

to get module admin:

```
local ma = Gamestate.current().moduleAdmin
```

**ma:newModule(directory, name)**

**Returns module**

**ma:seekInstance(module, own)**

**Params numbers module,own** the registry numbers that the module and instance owns.

**Returns instance**

**ma:getModules()**

**Returns** table of modules.

**ma:getModulesFrom(dir, make)**

**Params string dir** the directory where the modules are.

**Params boolean make** if the module should be created.

**Returns table** in format {dir = string,name = string,derivatives = table}

**ma:make(table)**

**Params table** in format {dir = string,name = string,derivatives = table}

**ma:save(map, [source or filter].parents)**

**Params string map**

**Params string source**

**Params filter** the filter function(module), return module.

**Params table parents** Use: *getParents()*

**ma:saveModule(module, map, source, parents)**

**Params Module module**

**Params string map**

**Params string source** source directory.

**Params table parents** Use: *getParents()*

**Returns table** the table with the saved module.

**ma:checkModule(src, name, instances, parents)**

**Params src** source directory

**Params name** name of the module

**Params table instances** the modules instances

**Params table parents** Use: *getParents()*

**ma:load(map, table, parents)**

**Params string map**

**Params table** the table where all of the modules were saved into.

**Params table parents** Use: `getParents()`

**ma:getInstance()**

**Returns table** of instances

**ma:clear()**

## Module

**module:seek(derivative)**

**Params string derivative** the name of the derivative.

**module:getInstance()**

**Returns table instances**

**module:addDeriv(dir, name)**

**Params string dir** the source directory of the derivative

**Params string name** the name of the derivative.

**module:addInstance(name, x, y, zmap)**

**Params string name** the name of the source derivative

**Params numbers x,y** the position in game world

**Params number zmap** the z position in game world

**Returns instance**

**module:remove()**

## Trigger admin

To get the trigger admin handle:

```
local ta = Gamestate.current().triggerAdmin()
```

**ta:setEnvironment(global)**

**Params table global** the pairs table with globals. :returns table environment: use in coalition with setenv.

**ta:getEnvironment()**

**Returns table environment**

**ta:load(t)**

**Params table t** the table with saved triggers.

**ta:getModes()**

**Returns table** table of strings of trigger modes.

**ta:exec(function, var)**

**Params function** a function to execute

**Params var** the self variable.

**ta:addTrigger(item)**

**Params Editor\_item item** the item that exists within the editor.

**Returns Trigger** the trigger module used below.

## Trigger

**Trigger:setActiv(mode, fns)**

**Params string mode** the mode of action

**Params function fns** the function to execute.

**Trigger:save()**

**Returns table**

**Trigger:load(t)**

**Params table t** the table previously made using `Trigger:save()`

## Image admin

To get the image admin use:

```
local ia = Gamestate.current().imageAdmin
```

**ia:addImage(sheet, quad, x, y, rotation, scale\_x, scale\_y)**

**Params Sheet sheet** the sheet from texture manager.

**Params Quad quad** from the sheet.

**Returns Image**

**ia:clear()**

**ia:getItems()**

**ia:save()**

**ia:load(tables, tm, gamestate)**

**Params TextureManager tm** link to texture manager

**Params gamestate** the current gamestate

## Image

**Image:unpack()**

**Image:mirror()**

**Image:flip()**

**Image:setScale(sx, sy)**

**Image:setZmap(z)**

**Image:setRotation(r)**

Params number r In radians!

**Image:setMod(modx, mody)**

Params numbers modx,mody the modifiers in x and y directions, used for parallax.

**Image:makeColision()**

**Image:update(dt)**

**Image:getPos()**

Returns numbers x,y position

**Image:rotate(r)**

Params number r in radians

**Image:draw(DEBUG)**

Params bool DEBUG debug for drawing wireframe etc

**Image:remove()**

**Image:OnMenu(menu, ui)**

Params ui\_item menu

Params ui\_package ui

## Shape admin

To get the shape admin do:

```
local sa = Gamestate.current().shapeAdmin
```

**sa:newRectangle(mode, x, y, width, height, color)**

Params string mode the mode, "fill" or "line"

Returns Rectangle

**sa:newCircle(mode, x, y, r, color)**

Params string mode "fill" or "line"

Returns Circle

**sa:newPolygon(mode, x, y, points, color)**

Params string mode "fill" or "line"

Params table points the table full of points eg {x,y,x2,y2}

Returns Polygon

**sa:newLine(color, x, y)**

Returns Line

## Rectangle

`rectangle:setPointOfCreation(x, y)`  
params numbers x,y the position of creation.

## Circle

`circle:setRadius(r)`  
params number r Radius in radians.

## Polygon

`polygon:setPoints(points)`  
params table points

## Shared

`<item>:unpack()`  
Returns table table of data from the shape that can be saved using Tserial.

`<item>:OnCopy()`

`<item>:setMod(px, py)`  
Params numbers px,py the x and y modifiers

`<item>:getColision()`  
Returns HardonCollider shape

`<item>:removeTexture()`

`<item>:setTexture(sheet, quad)`  
Params Sheet sheet  
Params string quad the name of the quad.

`<item>:update(dt)`

`<item>:draw(debug)`

`<item>:makeColision()`

`<item>:remove()`

`<item>:OnRightClick(menu, ui, editor)`  
same as *OnMenu* just slightly changed syntax

## Line

`line:addPoint(x, y)`

`line:addTempPoint(x, y)`  
Params numbers x,y the x and y position of the temporary point.

**line:update (key, key2)**

Params booleans **key, key2** key is for left click and key2 is for middle click for edit mode.

**line:makeSilent (x)**

Params bool **x** sets silent mode.

**line:remove ()**

**line:makeColision ()**

**line:draw ()**

**line:checkPoints ()**

**line:returnPoints ()**

Returns **table** of points, use in coalition with `polygon:setPoints ()`

## Textures

There are two modules for textures:

### Texture manager

```
local tm = Gamestate.current().textureManager
```

**Note:** functions that start with *pr* are used for once only loads.

**tm:newSheet (name, image)**

Params string **name** of the new sheet.

Params `love2dImage` **image**

Returns **Sheet**

**tm:prNewSheet (name, image)**

Returns **Sheet** Does NOT add it to the active sheets.

**tm:remove ([sheet or sheet\_name])**

Params **Sheet sheet**

Params string **sheet\_name** the name of the sheet

**tm:addSheet (sheet)**

Parameters **sheet** (*Sheet*) –

**tm:seek (sheet)**

Params string **sheet** the name of the sheet

Returns **Sheet**

**tm:save (map, parents)**

Params string **map**

Params table **parents** Use: `getParents ()`

**tm:load (map, parents)**

Params string map

Params table parents Use: `getParents()`

**tm:unpack ()**

Returns table to save using Tserial.

**tm:clear ()**

**tm:prLoad (map)**

Parameters map (string) -

## Sheet

**sheet:addQuad (name, x, y, w, h, delay, frame1, frame2)**

Params numbers x,y,w,h position and dimensions

Params delay,frame1,frame2 the beginning and end frames and the delay between them.

**sheet:getImage ()**

Returns love2DImage

**sheet:seek (s, remove)**

Params string s the name of the quad

Params bool remove if you want to remove the sheet; pass true

**sheet:getQuads ()**

Returns table,image full of quads, then the sheets image.

**sheet:getQuad (name)**

Returns Love2dQuad the individual quad.

**sheet:save ()**

**sheet:remove ()**

**sheet:applyEdit ()**

## Texture mapper

```
local tmap = Gamestate.current().textureMapper
```

**tmap:new (shape, img, quad)**

Params Shape shape shapeAdmin shape.

Params Love2DImage img

Params Quad quad

**tmap:setImage (img, quad)**

Params Love2DImage img

Params Quad quad

**tmap:update(dt)**

**tmap:updateCol()**

Call when the points of your Shape changes.

**tmap:draw(sx, sy)**

**Params numbers sx,sy** scalex and scaley

**tmap:remove()**

## Text admin

to get text admin:

```
local ta = Gamestate.current().textAdmin
```

**ta:addText(x, y, z, font)**

**Params numbers x,y,z** coordinates.

**Params love2DFont font**

**Returns Text**

**ta:load(data)**

**Params** table of saved text.

## Text

**Text:setText(t, font)**

**Params string t** the text

**Params love2DFont** the font

The Text inherits the following functions:

getColision, draw, update, OnMenu, OnMenu, OnCopy, setMod, save, remove

**However it does not include “load“ !**

## layers

To manage z coordinates I decided to produce a layer system, it is supposed to replicate the layer method just like in that of graphics production software including photoshop etc.

Obviously it is still missing a bulk of features such as the ability to add effects and drawing methods, but this is coming soon when I sort out some performance issues.

To get layers:

```
local layers = Gamestate.current().layers
```

**layers:update(dt)**

**Params number dt** delta time.

**layers:clear()**

**layers:unpack(zmap)**

**Params number zmap** the zmap of the target layer.

**layers:save ()**

**Returns table** the table with all the saved data.

**layers:load (a)**

**Params table a** the table retrieved from `layers:save ()`

**layers:getRange ()**

**Returns numbers min,max** the range of z cords in use.

**layers:seek (zmap)**

**Params number zmap** the z coord of the target layer.

**layers:removeLayer ([layer or zmap])**

**Params Layer layer**

**Params number zmap** the z coord of the layer.

**layers:move (l, z)**

**Params Layer l**

**Params number z** the new z coordinate.

**layers:moveLayers (old, new)**

**layers:addLayer (name, zmap, modx, mody, col, sound, transp)**

**Params string name** the identifying name of the layer.

**Params numbers zmap,modx,mody** the z coord and modifier position.

**Params booleans col,sound**

**Params number transp** transparency, out of 255

**Returns Layer**

## Layer

**Layer:getMod ()**

**Returns numbers modx,mody**

**Layer:setParallax (x, y)**

**Params numbers x,y** the modifiers

**Layer:setCollision (col)**

**Params boolean col** if the layer enables collision.

**Layer:setName (name)**

**Params string name** the new name of the layer.

**Layer:setData (name, zmap, modx, mody, col, sound, transp)**

Same params as `layers:addLayer ()`

**Layer:unpack ()**

**Returns** name,zmap,modx,mody,col,sound,transp

## Map admin

The map admin is a table of functions used for drawing and parsing the map.

Be very careful when modifying these functions.

---

**Note:** The self parameter in the functions bellow relates to a State such as Play,Editor or Battle.

---

`mapAdmin.parse` (*active, passive, self*)

**Params table active** HardonCollider active shapes.

**Params table passive** HardonCollider passive shapes.

**Returns table** active shapes.

`mapAdmin.scanModules` (*self, collider, ret*)

**Params HardonCollider collider**

**Params table ret** return table.

**Returns table** of modules.

`mapAdmin.updateSingle` (*shape, dt*)

**Params Instance shape** the instance as found from parse/parseSingle

`mapAdmin.parseHead` (*self*)

parses the head portion, use for Editor only.

`mapAdmin.parseSingle` (*self, shape, dt, rt* [, *cond or filter* ])

**Params HardonCollider\_shape shape**

**Params table rt** return table for priority instances.

**Params boolean cond** if the instance should be updated or not.

**Params function filter** filter that either allows an instance through or not

`mapAdmin.parseTail` (*self, dt*)

**Params number dt** delta time.

Used for Editor only.

`mapAdmin.draw_visible` (*self, zmap, current\_zmap*)

**Params none zmap** used for legacy.

**Params number current\_zmap** the current zmap.

`mapAdmin.mousepressed` (*self, mx, x, y, b*)

**Params mx** use `ui:getmx()`

**Params x,y** position of the click

**Params string b** the keyConstant.

`mapAdmin.updateCol` (*self, mx*)

**Params mx** use `ui:getmx()`

`mapAdmin.shapeCol` (*self, mx, dt, a, b*)

**Params mx** use `ui:getmx()`

**Params HardonCollider\_shapes a,b** the shapes colliding

mapAdmin.**shapeStop** (*self, mxb, dt, a, b*)

**Params mxb** use ui:getmxb()

**Params HardonCollider\_shapes a,b** the shapes colliding

mapAdmin.**clear** (*collider, self*)

**Params HardonCollider collider**

mapAdmin.**save** (*self, name, items*)

**Params string name** the map name.

**Params table items** a table of instances.

mapAdmin.**load** (*self, file*)

**Params string file** the filename of the map.

## Modules

The module system is made to facilitate large constructs for complex systems to be installed in the game.

If your idea is not complex then I'd highly encourage you to use scripts instead.

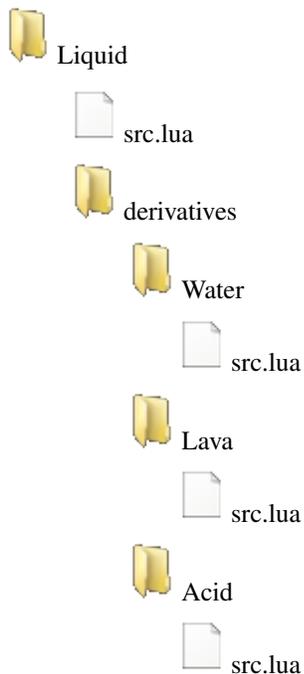
## Creating

To create a module we must first create a file structure.

Our working directory will be a brand new map. We will start in **your\_map/modules/**

I am going to create water so here is how I am going to do it.

We need to create:



I will focus on creating the Module **Liquid** and the derivative **Water**

## Module

The **Water** module is fairly easy to build, **src.lua** is our main component and will be scanned when loading modules.

Here are the functions that we need to define first:

```
liquid = {}

function liquid:new(gamestate, colider, phyWorld, x, y, width, height)
    local self = {
        x = x or 0,
        y = y or 0,
        zmap = gamestate.current_zmap,
        width = width or 32,
        height = height or 32,
        parent = gamestate,
        type = "liquid"
        Collider = colider,
    }

    setmetatable(self, {__index = liquid})
    self:makeColision()

    return self
end

function liquid:makeColision()
    local Collider = self.colider
    local x,y = self.x,self.y
    local w,h = self.width,self.height

    if self.colision then
        Collider:remove(self.colision)
    end
    self.colision = Collider:addRectangle(x,y,w,h)

    function self.colision.getUserData() -- this is required for identification
        return {source = self,part = "main"}
    end

    if self.OnMakeColision then -- this is necessary!
        self.OnMakeColision(self.colision)
    end
end

function liquid:getColision()
    return self.colision
end

function liquid:update(dt)
    -- this part is optional

    local w,h = (self.width)/2, (self.height)/2
    self.topX,self.topY = incrementPos(self.x,self.y,w,h)
    if not self.Free then
        self.dx = self.topX + (self.modx or 0)
        self.dy = self.topY + (self.mody or 0)
    end
end
```

```
        else
            self.dx = nil
            self.dy = nil
        end

        -- end of optional

        self.colision:moveTo(self.dx or self.x,self.dy or self.y)
    end

    function liquid:draw(debug)
        if debug then
            setColor(colors.yellow)
            self.colision:draw("line")
            setColor()
        end
    end

    function liquid:save()
        local t = {
            x = self.x,
            y = self.y,
            z = self.zmap,
            w = self.width,
            h = self.height,
        }
        return t
    end

    function liquid:load(table)
        self.x = table.x
        self.y = table.y
        self.zmap = table.zmap
        self.width = table.width
        self.height = table.height
    end

    function liquid:remove()
        self.Collider:remove(self.colision)
    end
end
```

These are all the mandatory functions, you must define them for each one of your modules.

To make a water effect I decided to split our work in 3 sections.

1. **Get the reflection of the block above.**



---

**Note:** White = new tile Gray = the new position.

---

At *liquid:new*

```
self = {
  ...
  ...
  canvas = love.graphics.newCanvas(32,32)
}
```

At *liquid:makeColision*

```
canvas = love.graphics.newCanvas(w,h)
```

At *liquid:update*

```
local game = Gamestate.current()
local mainCanvas = game.mainCanvas
local lg = love.graphics

local w,h = self.width,self.height
local upx,upy = game.cameraBox:bbox() -- the left top position of the camera.

local cx,cy = self.x or self.dx,self.x or self.dy -- our position

-- we need to find the new coordinates

cy = cy + h
-- we plan to reflect the top of our tile so we add height parameter to our y_
↪position.

-- now we need to find the difference between the two and translate the new tile to_
↪the top.
local tx = cx - upx
local ty = cy - upy

-- I didn't use the findDistance as our numbers may be negative in some cases,_
↪findDistance gives you only absolutes.
-- now the new tile will be drawn at [0,0] of our new canvas.
lg.setCanvas(self.canvas)
  lg.push()
  lg.translate(tx,ty)
  lg.draw(mainCanvas)
  lg.pop()
lg.setCanvas()
```



2. Flip the reflection and place it at our block.

At *liquid:draw*

```
local x,y = self.colision:bbox() -- bbox gives our top cordinates
-- we need to make a reflection so we need to pass -1 as our y-scale.
-- we want to scale it around the center hence we pass width and height halved as our
↳offsets.
love.graphics.draw(self.canvas,x,y,0,1,-1,self.width/2,self.height/2)
```

### 3. Add a water effect from shader.

At *liquid:update*

```
...
...
lg.setCanvas(self.canvas)

...

if self.shader then
    lg.setShader(self.shader)
end

lg.draw(mainCanvas)

...
```

If we coded our shader properly it should now look like this:



Here's how our final render should look like when applied across a larger field.



### Expanding the module

We may need to expand the module to allow for more features, for example we may want to copy a tile at a different position so we could assign a factor.

This is our expected result:



At this point it is obvious that we need a few more functions:

```
function liquid:setFactor(factor)
    self.factor = factor
end
-- and
function liquid:setShader(shader)
    self.shader = shader
end
-- also
function liquid:setSize(w,h)
    self.width = w
    self.height = h
    self:makeColision()
end
```

We will also need to make a small incision

At *liquid:update*

```
...
cy = cy + h *(self.factor or 1)
...
```

## Derivative

Now we have a module running we need to make a derivative, in this case we will make Water. to do this we will create a



Water

at */my\_map/modules/liquids/*

in this folder we will place:



water.glsl



src.lua

In *src.lua* we will place the following code:

```
local derivative = {}
local shader

function derivative:new(module, dir, gamestate, colider, phyWorld, x, y)
    if not shader then shader = love.graphics.newShader(dir.."/water.glsl")

    local instance = module:new(gamestate, colider, phyWorld, x, y)

    instance:setShader(shader)

    return instance
end

return derivative
```

I will let you code *water.glsl* on your own as I am not proficient enough at GLSL at the moment.

If you are kind enough then please get me [The Orange Book](#), on my amazon wish list (when I have it up).

This will surely get me up to speed with GLSL.

## Using

We finally managed to reach the point at which we start to use our module.

To place the module on the map use the  and find it.

---

**Note:** If you cannot find the module then you failed the file structure test.

---

When you have placed your module  it and a menu will pop up with the following:

*Delete*

*Add Trigger*

*Remove Trigger*

*Set Layer*

*Properties*

To add more to this menu use the `<module>OnMenu()`

## Dependencies

Some derivatives might need dependencies, for example the *Elevator's helevator* derivative needs the *Door* module

To do this you need to define a few functions in *Elevator/helevator/src.lua*

```
local der = {}
function der:new(...)
...

```

```

local door_der = self.dep[1].derivative
local door_mod = self.dep[1].module
local door_dir = self.dep[1].der_dir

local door = door_der:new(door_mod,door_dir,mode,colider,phyWorld)

...

end
end

function der:getDependencies()
  local d = {}
  local dep = {module = "door",derivative = "elevator_door"} -- search for Module:
↪door, derivative: elevator_door
  table.insert(d,dep)
  return d
end

function der:setDependencies(d)
  self.dep = d
end

```

So this derivative needs the *Door* module which must have *elevator\_door* derivative.

!! Beware of [Circular](#) dependencies !!

## Available functions

### Module

#### Necessary

**<module>:new(gamestate, colider, phyWorld, x, y)**

#### Parameters

- **gamestate** – the gamestate the derivative was created in.
- **coluder** – active HardonColider instance.
- **phyWorld** – physics world (Box2D)
- **x, y** (*numbers*) – positions.

**Returns** module

**<module>:makeColision()**

**<module>:update(dt)**

**<module>:draw(debug)**

**Parameters** **debug** (*bool*) – debug mode.

**<module>:save()**

**Returns** table of data.

**<module>:load(table)**

**Parameters** **table** – the table of data returned from `<module>:save()`  
`<module>:remove()`

## Optional

`<module>:OnMenu(menu, ui)`

### Parameters

- **menu** (*ui\_item*) – the Menu.
- **ui** (*ui\_package*) – The UI package currently active.

`<module>:OnRightClick(ui, menu)`

### Parameters

- **menu** (*ui\_item*) – the Menu.
- **ui** (*ui\_package*) – The UI package currently active.

`<module>:OnLoseFocus()`

## Derivative

### Necessary

`<derivative>:new(module, dir, gamestate, colider, phyWorld, x, y)`

### Parameters

- **module** – the module instance
- **dir** – the source directory
- **gamestate** – the gamestate the derivative was created in.
- **coluder** – active HardonColider instance.
- **phyWorld** – physics world (Box2D)
- **x, y** (*numbers*) – positions.

**Returns** instance

## Optional

`<derivative>:getDependencies()`

**Returns** **table** {module = string, derivative = string}

`<derivative>:setDependencies(d)`

**Parameters** **d** (*table*) –

## Stock

There are a number of stock modules but I will not document them here, they are fully open source so you can read them yourself.

They are located under *resources/modules/*

## Assets

This portion of the guide is specifically designed to tell you all about where and how you should keep your assets and how to minimise the space your assets take.

## Maps

Maps are independent objects, meaning that there are no dynamic links outside of them.

This in turn means that any object outside of the map cannot be used to enhance or add to the map, however there are cases where some outside elements are required. One of them is other maps, if you don't want to include a map as a child you need to include it in **either maps/** or **inbuilt\_maps/** folders.

The maps are neatly divided into their file structure as described here *File structure*.

It is important to notice that resources are close to the same syntax:



The folder spritesheet and image will look **identical**, however the reason behind this is that the spritesheets are named the same as their parent images.

So if you have an image called *I-am-an-image.png* then the spritesheet will be called: *I-am-an-image.png*, the extension wont change.

This is done deliberately to allow for appropriate naming and prevent name clashes.

---

**Note:** All saved files are actually made up of readable text so if you want to you can just *write* the map.

This means that if you want to merge a spritesheet you can just copy ones content into another, but beware that there may be some internal name clashes.

---

### Inheritance

the inheritance law applies to all maps and states the following:

**All children have the same resources as their parent.**

Therefore if you have a child of a map it will be able to use all the resources found in its parent.

This is moderated by you, meaning that if a resource is copied you will have to delete it to save space, it is important that you do this to ensure small file size.

---

**Note:** Lucidity will **NEVER** delete files that you make.

If you notice that Lucidity is in any way removing files other than *.troll* files, make sure to check the modules you are using.

---

### Save data

#### How does it work?

Save data works by scanning through the registry of the modules and saving them, when loading the registry matches that of the saved modules.

The data is recorded in either a save file or *one\_use.troll*, which is used for once only loads.

#### One use

The *one\_use.troll* is created before you switch a state, in most cases a transition from Play to Battle.

However the save file is removed on load, this is done by checking its extension, if it has a *.troll* it will be removed.

#### I found a 'one\_use.troll' in my saves, why is that?

The *one\_use.troll* was created to allow for loading just in case you crash when switching states, this allows for an easy recovery and prevents loss of data.

### Modules

When you pick a module from the *Resources* pool, it will always copy in full, in other words it will copy the module and all the files in its folder but ignore any derivatives.

#### Derivatives

When picking a derivative for the module it will only copy the single derivative and its dependencies.

As afro mentioned in modules, everything will be copied in the derivative's own folder.

**C**

camPos() (built-in function), 18  
checkPlayerCol() (built-in function), 3  
copyFiles() (built-in function), 3

**D**

deepCopy() (built-in function), 5  
deepCopy2() (built-in function), 5  
drawRelease() (built-in function), 4

**F**

findDimensions() (built-in function), 4  
findDistance() (built-in function), 4  
findItem() (built-in function), 5  
findMap() (built-in function), 4

**G**

gamepadpress() (built-in function), 4  
getEnv() (built-in function), 3  
getFactors() (built-in function), 4  
getPackageAlt() (built-in function), 3  
getParents() (built-in function), 5  
getSaves() (built-in function), 5  
getTween() (built-in function), 18

**H**

hsvToRgb() (built-in function), 4

**I**

imagestencil() (built-in function), 5  
incrementPos() (built-in function), 4  
inRange() (built-in function), 4  
inRangeHigher() (built-in function), 4  
inRangeLower() (built-in function), 4

**K**

keyboardpress() (built-in function), 4

**L**

loadGame() (built-in function), 5  
loadGameShort() (built-in function), 5  
loopRange() (built-in function), 3

**M**

mapAdmin.clear() (built-in function), 50  
mapAdmin.draw\_visible() (built-in function), 49  
mapAdmin.load() (built-in function), 50  
mapAdmin.mousepressed() (built-in function), 49  
mapAdmin.parse() (built-in function), 49  
mapAdmin.parseHead() (built-in function), 49  
mapAdmin.parseSingle() (built-in function), 49  
mapAdmin.parseTail() (built-in function), 49  
mapAdmin.save() (built-in function), 50  
mapAdmin.scanModules() (built-in function), 49  
mapAdmin.shapeCol() (built-in function), 49  
mapAdmin.shapeStop() (built-in function), 50  
mapAdmin.updateCol() (built-in function), 49  
mapAdmin.updateSingle() (built-in function), 49  
mergeTable() (built-in function), 4  
MinMaxFromTable() (built-in function), 4  
mousepress() (built-in function), 4

**N**

nokey() (built-in function), 4

**P**

parse() (built-in function), 19  
parseColor() (built-in function), 3  
pralaxPos() (built-in function), 22  
precentOf() (built-in function), 5  
printStats() (built-in function), 3

**Q**

queryKey() (built-in function), 5

**R**

RCtoWH() (built-in function), 3

removeFromTable() (built-in function), 4  
render() (built-in function), 18  
reverseTable() (built-in function), 5

## S

saveGame() (built-in function), 4  
saveGameShort() (built-in function), 5  
saveLog() (built-in function), 3  
showFPS() (built-in function), 4

## T

tableLenght() (built-in function), 3

## W

worldPos() (built-in function), 18