Resolution Solution

v2002

Contents

Explanation of terms

Game size / resolution / game content / virtual size — by this terms, I mean surface with game content that would be scaled to fit on game window.

Window – on most OSs, "window" means surface to which program's content will be showed to. On some platforms, there might not be used term "window" for this surface, such as Android, IOS, game consoles such as PlayStation, but term "window" still be applied in this manual.

Variables

rs.scaleMode

- Type: **number**
  - Possible values:
    1. **Aspect Scaling**
    2. **Stretch Scaling**
    3. **Pixel Perfect Scaling**
- Default value: **1**

How library will calculate and scale content.

To change this value, use **rs.setScaleMode()** function. Manually changing this variable won't update library, until **rs.resize()** is called.

1 - Aspect Scaling

In this mode, **rs.scaleWidth** and **rs.scaleHeight** have same value, so scaled images doesn't look disproportional.

Aspect scaling will add 2 black bars on sides. If window wider then gameWidth, then there will be 2 black bars on left and right sides.

If window taller then game height, then black bars will be on top and bottom.

If game aspect and window aspect same, then there will be no visual black bars.

This scaling mode will fit most games with different art-styles, like vector, hand-draw, 3d renders, etc, due to fact that in this mode game doesn't lost proportions when game size is not same as window size. The only exception might be is pixel art, where pixel bleeding would ruin your game look (but, nobody stops you to use this mode instead). For that, you need scaling mode **3** which is made specifically for pixel-art.

The downsides is - Black bars. Not everyone likes them, especially if game was build for something like 4:3 resolution, while monitor on which user run your game is 16:9. They would get thick black bars on right/top and left/bottom sides.

2 - Stretch Scaling

In this scaling mode, game content will be scaled to fill entire window, no matter what size it has.

When window width bigger then game width. Looks kind of funny, doesn't it?

When window height bigger then game height. Looks kind of funny, doesn't it?

This, as Aspect Scaling, is fine to use with most art-styles, but unlike Aspect scaling, it's more noticeable if game aspect is not same as window aspect. For example, game with 4:3 resolution will look much more distorted on 16:9, then on Aspect scaling.

The upside is that there will be no black bars no matter what.

The downside is game will look distorted if game aspect is very different from window aspect, as you can see on screenshots above.

3 - Pixel Perfect Scaling

As was mentioned before, Pixel Perfect scaling is *perfectly* suited for games with pixel-art.

It will make sure, that calculations result in **number** scaling value (1, 2, 3, …) with minimal value being 1. It will floor down scale value to achieve this. So, if scale value turns out to be 1.7, it will be floored to 1, or floored to 2 if scale is 2.3. It will guaranty, that your game content will stays as crisp as possible, and there will be no pixel bleeding.

The downside is, however, that this mode will produce 4 black bars on 4 sides of window, game and window aspect doesn't result in integer value:
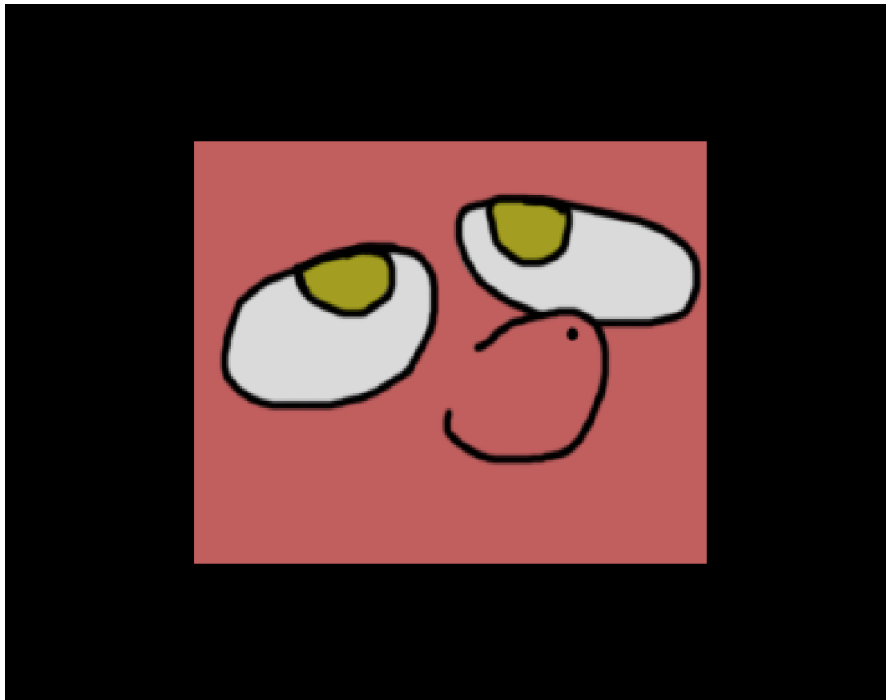
Game size is 640x480, while window size is 1111x792, which resulted in scale value being 1 and 4 black bars. If window was at least 1280x960, scale value would have been 2.

1280x960 window, 640x480 game produced scale 2.

rs.PixelPerfectOffsetsHack [boolean]

- Type: **boolean**

- Default value: **true**

When **rs.scaleMode** == **3**, library renders game in pixel perfect way. But, sometimes, calculations might result in non-integer **xOff** and **yOff**, which is not ideal when you want pixel perfect scaling, because non-integer position would result in non-crispy sprites and sprites wouldn't be perfectly aligned. It's especially noticeable when you actively resizing game window.

When **rs.PixelPerfectOffsetsHack** == **true** and **rs.scaleMode** == **3**, during **rs.resize**, library would check if window width and height if they is even. If not, then it would add **1** to width/height to compensate. This would guaranty that **xOff** and **yOff** always will be even, but, as result, sometimes **xOff** and **yOff** will be **1** pixel off to right side.

I recommend to set **rs.PixelPerfectOffsetsHack** to **true** if pixel bleeding/misalignment is too noticeable when **rs.scaleMode** == **3** during window resizing. In any other case, set it to **false**.

Try to run demo and slowly resize game window with mouse to see if there any noticeable difference to you.

rs.bars [boolean]

- Type: **boolean**

- Default value: **true**

When this variable is **true**, function **rs.drawBars()** will draw bars on top, bottom, left and right sides. If **false**, bars wouldn't be draw, which might result in slight performance improvement.

It make sense to disable them if:

- You don't need them, if you use different method for hiding content outside of virtual game width and height. (For example, you draw content to canvas and then scale it with this library. In this case, you probably wouldn't need this bars at all, because content was already hidden by canvas.)

- For debug purposes. With default **rs.start()** and **rs.stop()**, content behind black bars is not removed and still there. This black bars is only mask content. So you can disable bars rendering if you need to see what happens behind this bars.

rs.debug [boolean]

- Type: boolean

- Default value: **false**

Related to **rs.debugFunc()**. Activates/deactivate debug window with some info about current library state, such as scale mode, offsets, library version, etc. If this value **false**, then function **rs.debugFunc()** will do nothing if you call it in **love.draw()**. If **true**, then debug window would show up.
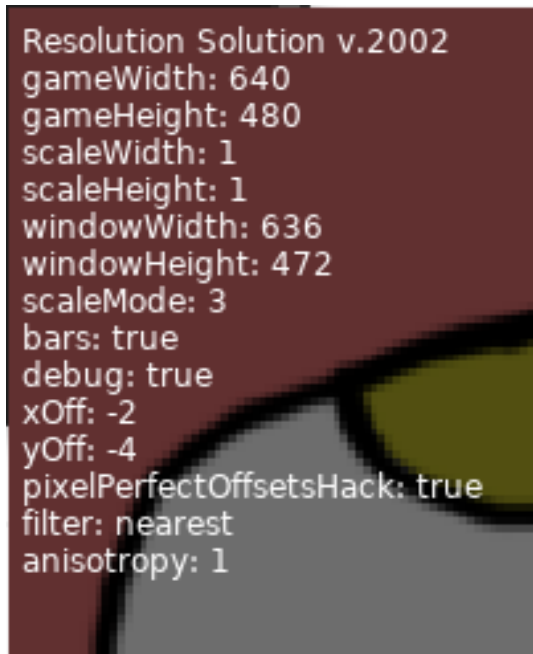
This debug info includes black, transparent background so text will be more visible.

rs.scaleWidth [number], rs.scaleHeight [number]

- Types:

  1. rs.scaleWidth: **number**

  2. rs.scaleHeight: **number**

- Default value: *will be calculated on first rs.resize() call.*

This 2 values is result of calculations between virtual/game size and window size to fit game content on window no matter which size it has (with exception with 3rd scale mode – Pixel Perfect – where if window **smaller** then game width and height then parts of game content will be simple not visible). This values will be calculated differently, depending on scale mode that you use.

In mode 1 and 3, **rs.scaleWidth** and **rs.scaleHeight** have same value, because there scaling same for width and height. In mode 2, width and height scaling is 2 **independent** values.

Changing this value manually will break library calculations, since this value intended to be updated via **rs.resize()** or built-in functions that relies on **rs.resize()** because this is read-only value that you can use in your own calculations if they somehow relies on scaling data.

You can use **rs.getScale()** to get both **rs.scaleWidth** and **rs.scaleHeight** at same time.

rs.gameWidth [number], rs.gameHeight [number]

- Types:
    - rs.gameWidth: **number**
    - rs.gameHeight: **number**
- Default values:
    - rs.gameWidth: **800**
    - rs.gameHeight: **600**

This is width and height values, that library will scale game to. They represent "resolution" of your game – amount of content/information that can be visible on screen in pixels.

You can get both **rs.gameWidth** and **rs.gameHeight** in single function call via **rs.getGame()**

Game with 640x480 resolution/game size. Image has same size, so it 100% fits game resolution.

Now game is 800x600, but image still 640x480. It doesn't fit game screen anymore and now you can see white rectangle that was outside of 640x480 boundaries.

As you can see, while this library takes care about scaling your game, you still need to design and implement yourself different resolutions (If you wants/can implement this, of course). For example, that if someone would play game in 800x600 they should be able to see important parts of level (such as enemies, bosses, items, etc) same way as someone who would play with 1920x1080, or, at least, provide some ways to tell player that on that part of level that player can't see on 640x480 resolutions is something interesting (for example, place NPC that would tell player "hey, did you know that there something interesting lies? Go ahead and check!").

To change this values, use **rs.setGame()** or **rs.init()** functions instead of manually changing them.

rs.windowWidth [number], rs.windowHeight [number]

- Types:

  1. rs.windowWidth: **number**

  2. rs.windowHeight: **number**

- Default values: *will be calculated on first rs.resize() call.*

This is mostly equivalent of **love.graphics.getWidth()** and **love.graphics.getHeight()**. Library utilize this values for calculating scaling values, offsets, etc. You can use this variables instead of **love.graphics.getWidth()** and **love.graphics.getHeight()**, but it's same things.

W – **rs.windowWidth**
H – **rs.windowHeight**

rs.xOff [number], rs.yOff [number]

- Types:

    1. rs.xOff: **number**

    2. rs.yOff: **number**

- Default values: *will be calculated on first rs.resize() call.*

This variables represent width and height for black bars. For example, with mode 1 (Aspect Scaling) if window width **bigger** then game width, you will get 2 black bars on left and right sides. And **rs.xOff** will represent width for black bar, but keep in mind, that there will be **2** black bars, so if you *2 this value, you will get how much horizontal space on screen takes black bars. Same for **rs.yOff**.

In mode 1 (Aspect Scaling) there will be only 2 bars at time. If window width **bigger** then game width, then there will be bars on left and right sides. And if window height **bigger** then game height, then there will be bars on top and bottom. And unless window has same aspect as game window (when you can't see black bars at all, meaning **rs.xOff** and **rs.yOff** equal 0) then either **rs.xOff** or **rs.yOff** will be 0 and another one will be non-0.

In mode 2 (Stretch Scaling) there no bars at all, so there **rs.xOff** and **rs.yOff** always will be **0**.

In mode 3 (Pixel Perfect Scaling) there will be 4 bars, so both rs.xOff and rs.yOff will be non-0 unless game window and game size (including integer scaling factor that this mode produces) same size.

Visualization:

*Aspect Scaling (1), when window height **bigger** then game height.*
*For both top and bottom bar used same **rs.yOff** value.*

*Aspect Scaling (1), when window width **bigger** then game width.*
*For both left and right bar used same **rs.xOff** value.*
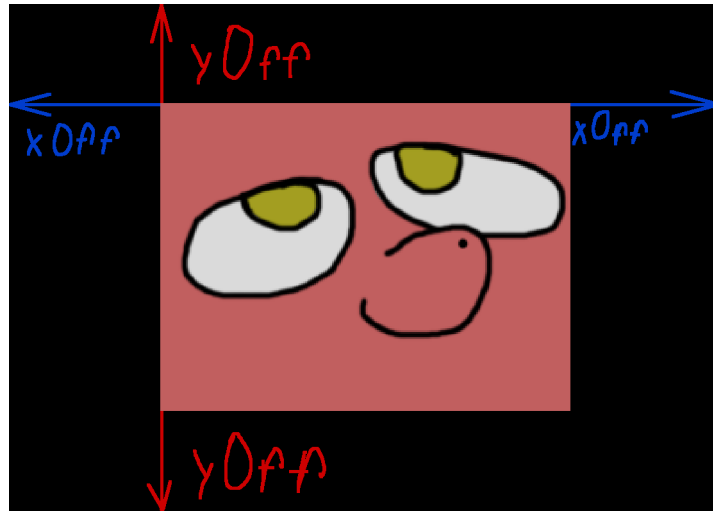
Pixel Perfect Scaling (3).
*For both left and right bar used same **rs.xOff** value and for both top and bottom bar used same **rs.yOff** value.*

Black Bars Coordinates [numbers]

rs.x1, rs.y1, rs.w1, rs.h1 – 1 bar; types: **numbers**.

rs.x2, rs.y2, rs.w2, rs.h2 – 2 bar; types: **numbers.**

rs.x3, rs.y3, rs.w3, rs.h3 – 3 bar; types: **numbers.**

rs.x4, rs.y4, rs.w4, rs.h4 – 4 bar; types: **numbers.**

Every black bar is simple **love.graphics.rectangle(**«fill»**)** with given coordinates, that calculated during **rs.resize()** based on current scaling mode and window size. When scale mode 2, there not bars at all.

Here visualizations:
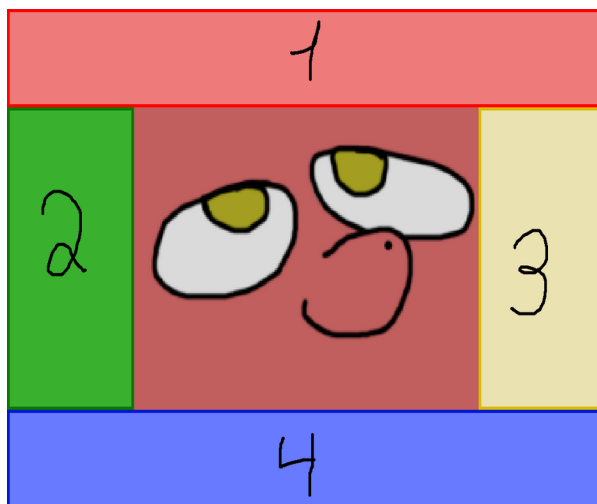
*When **rs.scaleMode** is 3*

When scale mode is 1 and window width bigger then game width.

When scale mode is 1 and window height bigger then game height.

Usually, you don't need this variables, unless you doing some custom rendering for this black bars.

Black Bars Colors [numbers]

rs.r — Red component; type: **number**; from 0 to 1; default **1**;

rs.g — Green component; type **number**; from 0 to 1; default **1**;

rs.b — Blue component; type **number**; from 0 to 1; default **1**;

rs.a — Alpha channel; type **number**; from 0 to 1; default **0**;

You can change color and alpha channel for black bars. They all use same color. To change all color variables, use function **rs.setColor()** or edit each of this variable individually, like this:

```
rs.a = 1
rs.r = 0.1
rs.g = 0.4
rs.b = 0.4
```

Remember, that starting from love 11, color become 0 — 1 as number, while before they was 0 — 255.

By default, black bars, in fact, black and you can't see though them.

rs.gameZone [table]

- Type: table
  - Table contains:
    - x
      - Type: **number**
      - X coordinate of top-left point of game zone.
    - y
      - Type: **number**
      - Y coordinate of top-left point of game zone.
    - w
      - Type: **number**
      - Width of game zone.
    - h
      - Type: **number**
      - Height of game zone.
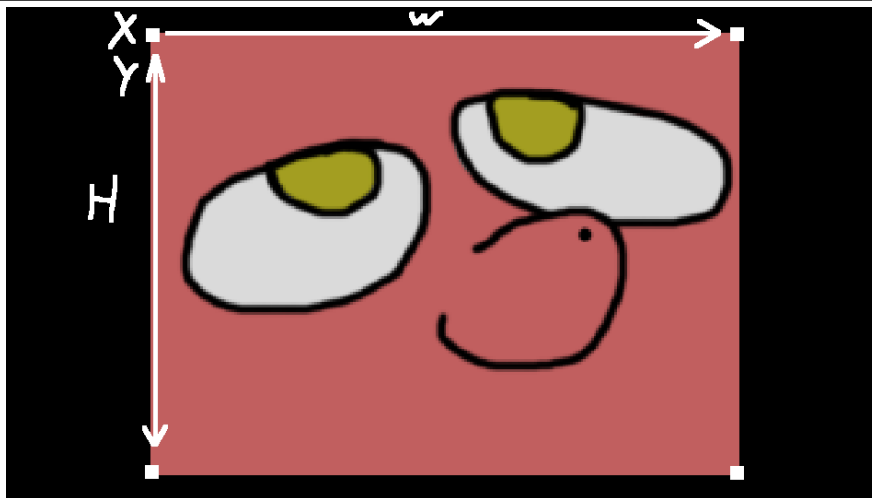
X and **Y** is top-left point
W – width
H – height

This table contains coordinates for game zone – zone inside black bars, what you can see on screen.

This might be useful for some cases:

- For custom UI rendering. With rs.gameZone you can locate screen edges inside scaled zone.

With combination of love.graphics.setScissor() you can disable black bars render and rely on scissors instead (which might be result in slightly better performance).

Like this:

```
love.graphics.setScissor(rs.getGameZone())
```

Functions

rs.init(options: table [optionally])

- Arguments:

  - options

    - Type: **table**

  - Note: If you pass nothing or **nil**, then no option would be changed, but library will recalculate data.

- Returns: nothing

With this function you can quickly change all library options at once. As name suggest, it's better to use once you required library to initialize it.

Example of usage:

```
rs = require("resolution_solution")
rs.init({width = 800, height = 600, mode = 3, a = 0.5})
```

Here, we required library and initialized it with next options:

- Library will scale window content to 800x600. (width, height)
- Library will do so with pixel perfect way. (mode)
- Black bars will be draw with 50% of transparency (a).

There more options available, here complete list:

- **width**:

  - Type: **number**

    - You can set pretty much any number, but setting negative numbers will break everything.

  - Game width to which library will scale to. For more info look for: **rs.gameWidth**

- **height**

  - Type: **number**

    - You can set pretty much any number, but setting negative numbers will break everything.

  - Game height to which library will scale to. For more info look for: **rs.gameHeight**

- **bars**

- Type: **boolean**
- Should library draw black bars or not. For more info look for: **rs.bars**

- **debug**
  - Type: **boolean**
  - Should game show debug info with library data or not. For more info look for: **rs.debug** and **rs.debugFunc()**

- **mode**
  - Type: **number**
    - Can be 1, 2 and 3.
  - How library should scale your game. For more info look for **rs.scaleMode**

- **r**
  - Type: **number**
    - From 0 to 1 in float, like 0.1
  - Red component for black bars. For more info look for **Black Bars Colors**

- **g**
  - Type: **number**
    - From 0 to 1 in float, like 0.1
  - Green component for black bars. For more info look for **Black Bars Colors**

- **b**
  - Type: **number**
    - From 0 to 1 in float, like 0.1
  - Blue component for black bars. For more info look for **Black Bars Colors**

- **a**
  - Type: **number**
    - From 0 to 1 in float, like 0.1
  - Alpha channel for black bars. For more info look for **Black Bars Colors**

- **hack**
  - Type: **boolean**
  - Should game use pixel perfect hack or not. For more info look for **rs.PixelPerfectOffsetsHack**

rs.setScaleMode(mode: number)

- Arguments:
  - mode: **number**
    - Acceptable values: 1, 2, 3
- Returns: nothing

Refer to **rs.scaleMode** for more info about scaling modes.

Reads value that you passed as argument, do sanity check, set this argument to **rs.scaleMode** and call **rs.resize()** to update library. You can pass only **1**, **2**, **3** as argument. Anything other then that will raise error.

Don't manually change **rs.scaleMode** because library won't update itself once you change this variable, unless you call **rs.resize()** afterward.

Example of usage:

```
rs = require("resolution_solution")
rs.setScaleMode(2)
```

Here we required library and then switched it scaling mode to 2, meaning stretched that will fill entire window.

rs.switchScaleMode(sideToSwitch: number [optional])

- Arguments:

  - sideToSwitch: **number**

    - Acceptable values: **1**, **-1**

  - If argument is **nil**, then it would be same as **1**.

- Returns: nothing

Refer to **rs.scaleMode** for more info about scaling modes.

This function needed to switch scale modes by 1 or -1. Calling this function with 1 will result in current scale mode + 1. If scale mode become more then **3**, then it will become **1**, so you could infinitely scroll between scale modes them. Calling with -1 will result in current scale mode – 1, and if scaling mode become **less** then 1, then scale mode will switch to **3**.

(I'm not sure about that one.)

Example of usage:

```
rs = require("resolution_solution")
rs.setScaleMode(1) -- Current scale mode is 1
rs.switchScaleMode(1) -- now mode is 2
rs.switchScaleMode(1) -- now mode is 3
rs.switchScaleMode(1) -- and... again 1.
rs.switchScaleMode(-1) – back to 3.
```

rs.switchPixelHack()

- Arguments: none

- Returns: nothing

rs.switchBars()

- Arguments: none

- Returns: nothing

rs.debugFunc(x: number or string; y: number or string)

w -- library will silently place this window on screen even if you would input something crazy like -999 or 9999

rs.switchDebug()

- Arguments: none

- Returns: nothing

rs.nearestFilter(filter: boolean [optional], anisotropy: number [optional])

- Arguments:
  - filter
    - Type: boolean
      - **true**, filtering would be set to "**nearest**" (for both **min** and **max** values).
      - **false**, filtering would be set to "**linear**" (for both **min** and **max** values).
    - Note: you can pass nothing or **nil** (which is same thing) which would result in same result as passing **true**.
  - anisotropy
    - Type: number
      - Set this number (if you need to) according to this: https://love2d.org/wiki/love.graphics.setDefaultFilter
    - Note: you can pass nothing or **nil** (which is same thing), which means library will take anisotropy value from **love.graphics.getDefaultFilter()** and pass it.
- Returns: nothing.

This function used to quickly set default filtering to **"nearest"**, which is go-to for Pixel Art. Simple call **rs.nearestFilter()** after you initialized library to set default filtering to **"nearest"**.

This library is mostly wrapper for https://love2d.org/wiki/love.graphics.setDefaultFilter, so learn about it before touching this function. This library exists simple because I (author of this library) associates filtering method with scaling (because this filtering all about scaling, duh!) so I prefer to set it in same place as this library in games code.

Example:

```
rs = require("resolution_solution")
rs.nearestFilter()
rs.init({w = 800, h = 600})
```

rs.resize(w: int [optional], h: int [optinal])

rs.start()

rs.stop()

rs.unscaleStart()

rs.unscaleStop()

rs.setColor(r: number, g: number, b: number, a: number)

- Arguments:

    1. **r**

        - type: **number**

        - Can be in-between 0 – 1. Passing more then 1 and less then 0 will result in error raising.

    2. **g**

        - type: **number**

        - Can be in-between 0 – 1. Passing more then 1 and less then 0 will result in error raising.

    3. **b**

        - type: **number**

        - Can be in-between 0 – 1. Passing more then 1 and less then 0 will result in error raising.

    4. **a**

        - type: **number**

        - Can be in-between 0 – 1. Passing more then 1 and less then 0 will result in error raising.

- Returns: nothing

This function allow you quickly change all colors of *black* bars via 1 function call. If you need to change individual color component, the just change it via table fields: **rs.r, rs.g, rs.b, rs.a**. Remember, that in that case there wouldn't be any checks so you can input something more then 1 and less then 0.

Example:

```
rs.setColor(0.4, 0.9, 1, 0.5)
```

rs.getColor()

- Arguments: none
- Returns:
    1. **r**
        - Type: **number**
        - Red component for color of black bars.
    2. **g**
        - Type: **number**
        - Green component for color of black bars.
    3. **b**
        - Type: **number**
        - Blue component for color of black bars.
    4. **a**
        - Type**: number**
        - Alpha channels for color of black bars

Shortcut function, that would return all colors for *black* bars. If you need to get individual color, get them directly from table fields: **rs.r**, **rs.g**, **rs.b**, **rs.a**

Example:

```
local r, g, b, a = rs.getColor()
```

rs.getGameZone()

- Arguments: none
- Returns:
  1. **x**
     - Type: **number**
     - X coordinate of top-left point of game zone.
  2. **y**
     - Type: **number**
     - Y coordinate of top-left point of game zone.
  3. **width**
     - Type: **number**
     - Width of game zone.
  4. **height**
     - Type**: number**
     - Height of game zone.

Shortcut function, that would return data of game zone. If you need to get individual value, get them directly from table fields: **rs.gameZone.x**, **rs.gameZone.y, rs.gameZone.w, rs.gameZone.h**

For more info look for **rs.gameZone**

Example:

```
local x, y, w, h = rs.getGameZone()
```

rs.defaultColor()

- Arguments: none

- Returns: nothing

Changes **rs.r**, **rs.g**, **rs.b**, **rs.a** to default black color. If you changed colors of *black* bars via manually editing **rs.r**, **rs.g**, **rs.b**, **rs.a** or **rs.setColor()** or **rs.init()**, and need to return back, then just call **rs.defaultColor()** and *black* bars become black.

Example:

```
rs.setColor(1, 0.3, 0.6, 1) -- we changed *black* bars to be
pink bars.
```
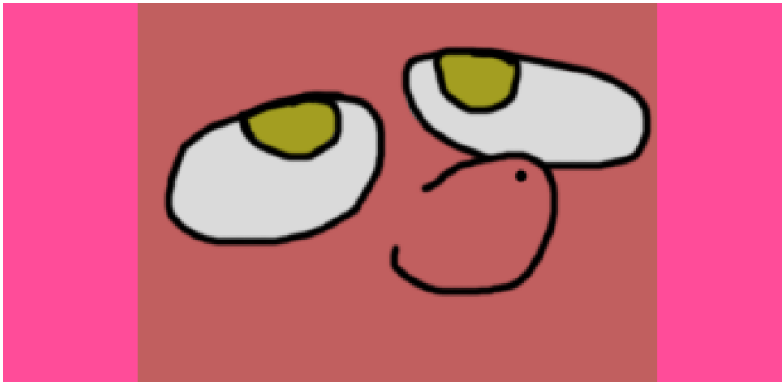

We made pink bars!

We don't like them, so let's revert back.

```
rs.defaultColor()
```


Back to black!

rs.drawBars()

- Arguments: none

- Returns: nothing

   This function draw black bars. Library internally uses this function during **rs.stop()** to draw black bars. You can call this function during **love.draw()** to draw black bars. This might be useful, if you use custom scaling functionality and you need to draw bars independently from **rs.start()** and **rs.stop()**.

   If **rs.bars == false**, then bars wouldn't be rendered at all, and if **true** then they will be rendered. To change this, you can manually change **rs.bars**, via **rs.init()**, via **rs.switchBars()** which would switch from **true** to **false** and vice versa (think about this function as shortcut for `rs.bars = not rs.bars`).

   Example:

   `love.graphics.setBackgroundColor(0, 0.4, 0.6, 1) -- Lets set blue background`

   `rs.defaultColor() -- And set *black* bars to be black.`

   `rs.bars = true -- And activate rendering of black bars (which is true by default, but this is example)`
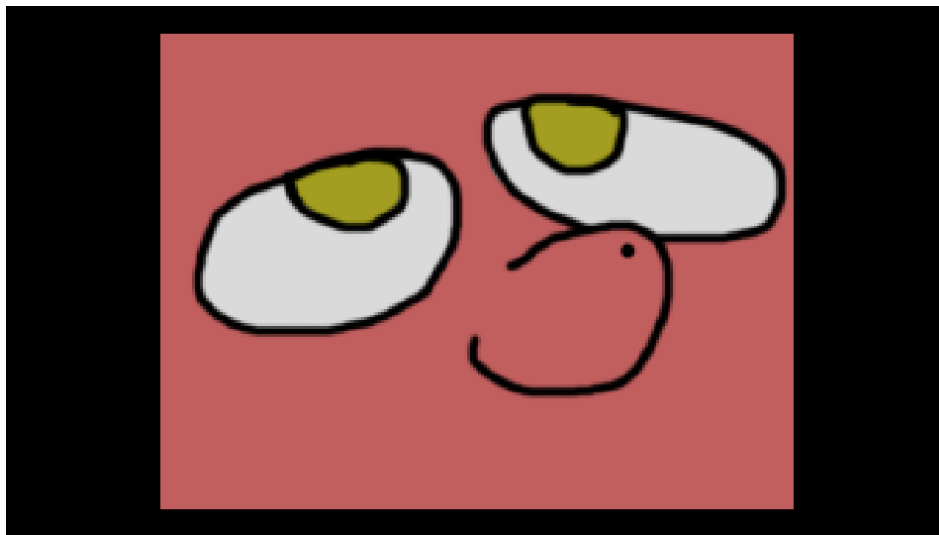
You would see blue background and black bars. (Here we have red pony instead, but imagine that this is blue background.)

`rs.bars = false` – Now lets disable rendering for black bars.
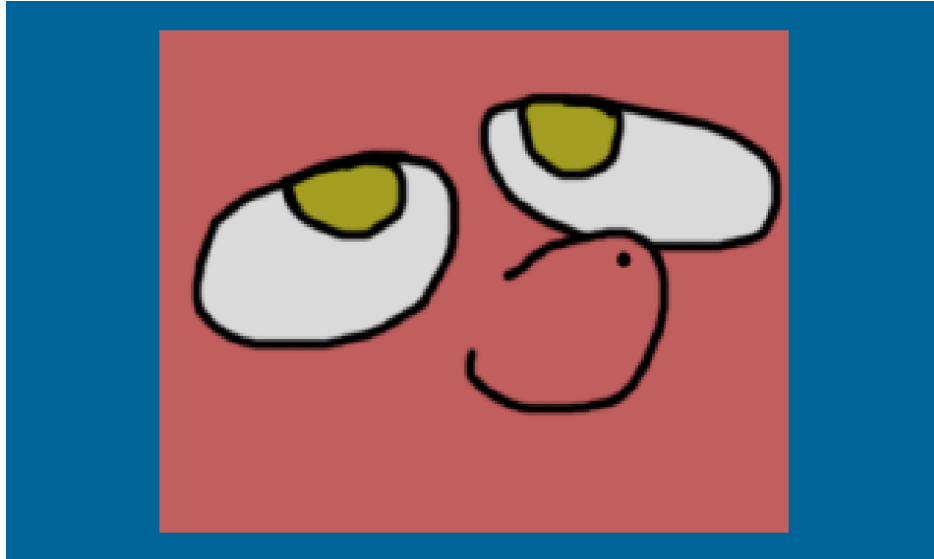
And since we disabled rendering for black bars, we can see that background, in fact, blue!

rs.getScale()

- Arguments: none
- Returns:
    1. x scale
        - Type: number
    2. y scale
        - Type: number

Shortcut function, that would return both **rs.scaleWidth** and **rs.scaleHeight**.

Example:

```
local xScale, yScale = rs.getScale()
```

rs.setGame(width: number, height: number)

rs.getGame()

- Arguments: none
- Returns:
  1. game width
     - Type: **number**
  2. game height
     - Type: **number**

Shortcut function, that would return both **rs.gameWidth** and **rs.gameHeight**. For more info look for **rs.gameWidth** and **rs.gameHeight**.

Example:

```
local gameWidth, gameHeight = rs.getGame()
```

rs.getWindow()

- Arguments: none

- Returns:

    1. window width

        - Type: **number**

    2. window height

        - Type: **number**

Shortcut function, that would return both **rs.windowWidth** and **rs.windowHeight**. For more info look for **rs.windowWidth** and **rs.windowHeight**.

You can also use **love.graphics.getDimensions()** instead or **love.graphics.getWidth()** and **love.graphics.getHeight()**.

Example:

```
local windowWidth, windowHeight = rs.getWindow()
```

rs.isMouseInside()

rs.getWindow()

- Arguments: none

- Returns:

  1. Is mouse inside game zone?

     - Type: boolean

Shortcut function, that would return both **rs.windowWidth** and **rs.windowHeight**. For more info look for **rs.windowWidth** and **rs.windowHeight**.

You can also use **love.graphics.getDimensions()** instead or **love.graphics.getWidth()** and **love.graphics.getHeight()**.

Example:

```
local windowWidth, windowHeight = rs.getWindow()
```

rs.toGame()

rs.toGameX()

rs.toGameY()

rs.toScreen()

rs.toScreenX()

rs.toScreenY()

rs.setMode(width: number, height: number, flags: table)

Tips and Tricks

How to scissor game content to hide something outside?

love.graphics.setScissor(rs.getGameZone())

How to get resolution of screen on which window is placed currently?

```
width, height = love.window.getDesktopDimensions(select(3,
love.window.getMode()).display)
```

Demo

```lua
rs = require("resolution_solution")
-- Refer ro source code of library, for rs.init() to get full list of avaliable options or their explanation.
-- but in most cases you only need to specify game width/height and default scale mode.
rs.init({width = 640, height = 480, mode = 3})
-- This function allow you to change color of bars that you will appear in aspect and pixel perfect modes.
-- By default, they will have black color, but you can change it and even make transparent.
-- To change individual color, use rs.r, rs.g, rs.b, rs.a for red, green, blue and alpha.
-- Also rs.getColor() will return 4 arguments with currect colors and rs.defaultColor() will return default black
color.
rs.setColor(0.1, 0.5, 0.2, 0.5)

-- Filter, works best for pixeleted raphics.
-- can be use without arguments, which same as "true".
-- It's simple wrapper for love.graphics.setDefaultFilter().
-- Refer source code for more info.
rs.nearestFilter(true)

-- Make window resizeable. I strongly suggest you to always make window resiable, via this love function or
conf.lua
-- After all, this library was designed for this.
rs.setMode(800, 600, {resizable = true})
-- Show library name and version in title.
love.window.setTitle(tostring(rs._NAME .. " v." .. rs._VERSION))

-- Example rectangle, that demonstrate how you can implement, for example, mouse collision detection,
-- and other translate functions.
local rectangle1 = {
  x = 100,
  y = 100,
  w = 100,
  h = 100,
  click = 0
}

-- Show/hide rectangle around scaled area.
local showGameZone = true

-- library was designed to update at love.resize() (it possible to update at love.update(), but it's not something
that you want to do),
-- so place it there. Also, side not: never forget to use rs.init() (even if you don't need to change any
settings) at least 1 at start of game/scene.
-- This is required since until 1st window resize, library will be not updated, so no scale, no offset, nothing
will be calculated.
love.resize = function(w, h)
  rs.resize(w, h)
end

-- Change options with keyborad
love.keypressed = function(key, scancode, isrepeat)
  if key == "f1" then
    rs.switchScaleMode()
  elseif key == "f2" then
      rs.switchBars()
  elseif key == "f3" then
```

```lua
        rs.switchDebug()
  elseif key == "f4" then
        showGameZone = not showGameZone
  elseif key == "f5" then
        rs.switchPixelHack()
  end
end


-- Example of how you can implement mouse collision detection function.
local mouseFunc = function(x, y, w, h)

  -- Translate mouse to ingame coordinates
  local mx, my = rs.toGame(love.mouse.getPosition())
  if mx   >= x                 and -- left
     my     >= y                   and -- top
     mx     <= x        +    w  and -- right
     my     <= y        +    h  then -- bottom
      return true
    end

    return false
end


love.mousepressed = function(x, y, button, istouch, presses)

    -- Example of usage for mouse collision.
    -- Add 1 to counter if clicked.
    if rs.isMouseInside() and mouseFunc(rectangle1.x, rectangle1.y, rectangle1.w, rectangle1.h) and button == 1
then
        rectangle1.click = rectangle1.click + 1
    end

    -- Example of how to use and transka scaled coordinates to screen coordinates.
    -- Set mouse cursor to rectangle.
    if button == 2 then
      love.mouse.setPosition(rs.toScreenX(rectangle1.x), rs.toScreenY(rectangle1.y))
    end

    -- Another translation example.
    -- Move rectangle to cursor.
    if button == 3 then
      rectangle1.x, rectangle1.y = rs.toGame(love.mouse.getPosition())
    end
end

love.draw = function()
  -- Start scaling
  rs.start()
  -- Background color.
    love.graphics.setBackgroundColor(0, 0.4, 0.6, 1)

    -- Change rectangle color if we touch it.
    if rs.isMouseInside() and mouseFunc(rectangle1.x, rectangle1.y, rectangle1.w, rectangle1.h) then
      love.graphics.setColor(1, 0.5, 0.5, 1)
    else
        love.graphics.setColor(0.5, 0.5, 0.5, 1)
    end
```

```lua
    -- Draw rectangle.
    love.graphics.rectangle("fill", rectangle1.x, rectangle1.y, rectangle1.w, rectangle1.h)

    -- Show counter and explanation.
    love.graphics.setColor(0, 0, 0, 1)
    love.graphics.print("Click on me!" .. tostring(rectangle1.click) ..  "\nYou can't click on me, if i behind\
nbars, because library\ncan take care of it!", rectangle1.x, rectangle1.y)

    -- Scaled text.
    love.graphics.print("I'm scaled text!", 200, 50)

    -- Example of how you can implement UI that should be scaled separately/differently from game.
    rs.unscaleStart()
      love.graphics.setColor(1, 1, 1, 1)
      love.graphics.print("I'm unscaled, despite being in-between rs.start() and rs.stop()!\nAlso bars draws ontop
of me!", 180, 50)
    rs.unscaleStop()

  -- Stop scaling.
  rs.stop()

  -- Bars text.
  love.graphics.setColor(1, 1, 1, 1)
  love.graphics.print("Bars can be any color you want! Not only black!", rs.windowWidth - 200, 0)

  love.graphics.setColor(0, 0, 0, 1)
    -- Example of how you can use rs.gameZone.
    -- Draw rectangle.
  if showGameZone then
    -- love.graphics.rectangle("line", rs.gameZone.x, rs.gameZone.y, rs.gameZone.w, rs.gameZone.h)
  end

  -- Call debug function.
  rs.debugFunc()

  -- Instructions.
  love.graphics.setColor(1, 1, 1, 1)
  love.graphics.print("Press F1 to change scaleMode. F2 to enable/disable bars. F3 to enable/disable debug info.
Press f4 to show/hide game zone borders.\nPress F5 to activate/deactivate pixel perfect hack (make sure to be in
scaling mode 3). When active, try resize window and see if you notice difference. \nTry to change window size and
click on rectangle. Press right mouse button to move cursor to rectangle. Press middle mouse to move rectangle
under cursor.", 0, rs.windowHeight - 100)
end
```

Changelog

v1000, 7 January 2021

Initial release! Yay!

v1001, 6 February 2022

New:

- Added comments for "Simple demo"

- Added more comments for functions and variables in library.

Changed / Fixed:

- Now, scaling.stop() will remember color that was set before it and return it back after.

- Fixed typo in "Simple demo"

v1002, 8 February 2022

Changed:

- Fixed (probably) edge cases in isMouseInside() function, so now it should corectly deal with non integer offsets provided by scaling.xOff/yOff

- Now **rs.isMouseInside()** return always true if scale mode is == 2, since there is no black bars in that scaling method so no need to wast CPU time on that.

- Updated **rs.isMouseInside()** comments.

- Rewritten "Simple demo", now it uses modified demo from github page.

- Fixed typos, rewritten/rephrased comments.

- Added note in **rs.toGame/rs.toScreen** about rounding/mismatching.

- Added note about **rs.isMouseInside()**.

v1003, 12 February 2022

New:

- Added library license text in **rs._LICENSE_TEXT**

- Added auto-completion API for Zerobrane Studio!

Changed / Fixed:

- Updated comments.

v1004, 19 May 2022

Renamed:

- **rs.widthScale -> rs.scaleWidth**

- **rs.heightScale -> rs.scaleHeight**

v1005, 19 May 2022

New:

- **rs.gameZone table**, which contains x, y, w, h coordinates of scaled area.

  You might need it when you want to draw UI, which shouldn't be scaled by library regardless of current scaling mode (stretching or with black bars), because to draw UI you need to know where starts/ends scaled area on window. And it might help for camera libraries, which uses love.graphics.setScissors

v1006, 20 May 2022

New:

- **rs.drawBlackBars()** added.
  So now you can call it to draw black bar outside of **rs.start()** and **rs.stop()**. Some libraries, especially that use love's scissors functionality might broke back bars rendering;

  Or camera (or any translating related) libraries might mess with coordinate translating. Which might end up in broken graphics and frustration.

  Also, this function uses same rules as **rs.stop()**, meaning **rs.drawBars** = false will result in **rs.drawBlackBars()** will be not rendered.

- Now **rs.stop()** will draw black bars via **rs.drawBlackBars()** function.

v2000, 27 December 2022

Big rewrite! Check source file for all detailed changes. (Some functionality in this version is not compatible with old versions.). Source file, at almost top, now include some tips and "tricks", check them out.

New:

- Pixel Perfect scaling! **rs.setScaleMode(3)** to check it out!

- **rs.init(options)** - before, to change some options in library, you could update value directly from rs.* table or use provided built-in functions. Now, considering new "insides" of library, changing options directly as **rs.scaleMode** == 1 will do nothing, because library will be updated only on **rs.resize()** or via newly (and old one) provided functions, including **rs.init()**.

  You should call **rs.init()** at least once, even if you don't need to update anything in options, otherwise until first **rs.resize()**, you will see black screen.

  You can pass argument as table with options, or pass nothing to just update.

- **rs.setScaleMode()** - allow you to change scale mode via number. Pass 1, 2, 3 to change.

- **rs.debug** - boolean, which controls if **rs.debugFunc()** will be rendered or not.

- **rs.debugFunc()** - function that will show some data that useful for debug. Call it somewhere in love.draw() as **rs.debugFunc()**.

- **rs.switchDebug()** - switch **rs.debug**, from true to false and vice-versa.

Removed:

- **rs.windowChanged()** - because now there no need in this callback.

- **rs.gameChanged()** - also not really useful anymore.

- **rs.gameAspect** - it was not really useful anyway.

- **rs.windowAspect** - also not useful.

- **rs.update()** - explained below.

Changed / Fixed:

- Before, there was only 2 bars: Left/right or top/bottom and they was avaliable only at scaleMode 1. With introduced 3rd scale method, Pixel Perfect, that has bars at top/bottom/left/right at same time, their functionality changed. You can still access them as: rs.x1, rs.y1, rs.w1, rs.h1 (from 1 to 4, rs.x1, rs.x2, rs.x3...), but order changed:

  1. top bar

  2. left bar

  3. for right bar

  4. for bottom bar

- Apparently, rs.gameZone table was never updated, because i forgot to do so in rs.update... Welp, that sucks. Now it updates properly.

- Now all functions, that expects arguments, have error messages to point out if you passed something wrong. Yay!

- **rs.resize** - now library update loop was designed around love.resize() function, instead of love.update(), like other scaling libs do. So less wasted frame time, yay! Don't forget to pass w and h from love.resize(w, h) to library as rs.resize(w, h). For comparability sake, it should be possible to put rs.resize at love.update and just pass rs.resize(love.graphics.getWidth(), love.graphics.getHeight()). It was not tested properly, but i believe there shouldn't be any problem with it, except maybe performance.

- **rs.switchScaleMode()** - before until 3rd scaling method, there was only 2 methods and this function acted more like "boolean". Now, you can pass 1 or -1 to choose how you want to switch methods: 1 -> 2 -> 3 -> 1... or 3 -> 2 -> 1 -> 1. If you pass nothing, function will act as you passed 1.

- Demo was rewritten.

- From now on, i will include minified version of library, with removed comments and minified code, that will make filesize lesser. https://mothereff.in/lua-minifier.

Renamed:

- **rs.drawBars** -> **rs.bars**

- **rs.drawBlackBars** -> **rs.drawBars**

- **rs.switchDrawBars** -> **rs.switchBars**

v2001, 31 December 2022

Small update, that add some QoL features, and hack for Pixel Perfect Scaling. Check source code (specifically **rs.pixelPerfectOffsetsHack** and **rs.resize()**) and this update log for more info.

Happy new year!

New:

- **rs.nearestFilter(filter, anisotropy)** - this function is easier to use wrapper for love.graphics.setDefaultFilter(). It expects 2 optional arguments:

  1. true/false or nil. true/nil results in nearest filtering, while false is linear.

  2. Anisotropy. It can be number or nil. If number, function will simply use it to slap into love.graphics.setDefaultFilter(), but if nil, then library will simply get anisotropy value from love.graphics.getDefaultFilter() and will use it instead. Current love uses 1 as default.

     If this function was never run, library will never touch or edit love.graphics.setDefaultFilter()

- **rs.pixelPerfectOffsetsHack** = false/true - very experimental feature, that aim to fix pixel bleeding in perfect scaling mode when window size is not even. It result in always "clean" pixels, but comes with side effects such as:

  1. **rs.windowWidth** and **rs.windowHeight** will be wrong by 1 if window is non even. The workaround is to use love.graphics.getWidth/Height instead.

  2. On non even window size, offset from left and top will place game content on 1 pixel left/upper. But if you never ever draw anything inside **rs.unscaleStart()** and **rs.unscaleStop()** and outside of **rs.start()** and **rs.stop()**, then you should probably fine using this hack.

- **rs.switchPixelHack()** - turn on/off mentioned hack.

- **rs.setMode()** - wrapper around love.window.setMode(). You should use this functions instead, since using love.window.setMode() might don't trigger love.resize() and therefore **rs.resize()** and as result scaling calculations will be wrong.

Changed / Fixed:

- Updated demo to include all new functions and values.

- Updated **rs.debugFunc()** to include all new values and functions.

v2002, 25 August 2023

In this update, I finally provided somewhat «decent» documentation using LibreOffice, so check it out and give your opinion about it.

There also small changes to how some functions works, so make sure to check them out.

New:

- Documentation as PDF file (+ source file for PDF. Made with LibreOffice.) with images and examples.

- **rs.resizeCallback()** - callback that will be called every time when **rs.resize()** was triggered.

  Might be useful, if you need update something related to resize function. For example, UI.

Changed / Fixed:

- **rs.pixelPerfectOffsetsHack** – now will be **true** be default.

  After some testing, I come up to decision, that it doesn't produce any problems. It will retain same name for compatibility.

- **rs.nearestFilter()** - optimized slightly and code cleaned.

- **rs.nearestFilter()** - added pcall check for input arguments.

- **rs.getGameZone()** - changed behavior. Now returns 4 values with **x**, **y**, **w**, **h** of game zone.

  Before, it was returning table like this:

  ```
  return {x, y, w, h}
  ```

  and to get values you would need do something like this:

  ```
  local gameZone = rs.getGameZone()
  ```

  ```
  love.graphics.rectangle("line", gameZone.x, gameZone.y,
  gameZone.w, gameZone.h)
  ```

  But now, it is simple as:

  ```
  love.graphics.rectangle("line", rs.getGameZone())
  ```

  ```
  love.graphics.setScissor(rs.getGameZone())
  ```

  Of course, if you need to access only specific value, you still can do:

  ```
  local x = rs.gameZone.x
  ```

- **rs.debugFunc()** - added 2 arguments that you can pass to it to place on-screen.

  For 1st argument, you can pass **"left"** or **"right"** strings or number. It's where this function's debug "window" will be drawn. **"left"** means on left side of screen and **"right"** on right side of screen. You can also input any number if you need to place it somewhere else.

  And for 2nd, same thing. **"top"** for top of window, **"bottom"** for bottom of window. And any number value if you want to place it somewhere else.

- **rs.debugFunc()** - now internally uses **love.graphics.printf()** which might result in slightly better performance (not that it important here much).

- **rs.debugFunc()** - now should return font back that was used before calling this function.

- **rs.init()** - added additional sanity checks for **r, g, b, a** values that you can pass to change color of black bars. Starting from love 11, colors values become $0 - 1$, before was $0 - 255$. So passing something more then 1 and less then 0 result in error raising.

- **rs.setColor()** - added additional sanity checks for **r, g, b, a** values that you can pass to change color of black bars. Starting from love 11, colors values become $0 - 1$, before was $0 - 255$. So passing something more then 1 and less then 0 result in error raising.

- New description for library: "Yet another scaling library." (Because this description reflects library better.)

- **rs.resize()** - library internally uses this function to re-calculate data, and also you, as developer, should place it in **love.resize()** and pass w and h to it. Now, if you don't pass arguments to it, library will take it manually using **love.graphics.getWidth()** and **love.graphics.getHeight()**. That means, that now you can call this function without any arguments and it will still works just fine, but it's still good idea to pass **w** and **h** arguments from **love.resize(w, h)** to **rs.resize()**.

- **rs.resize()** - I accidentally placed 2 times:
  `rs.windowWidth, rs.windowHeight = windowWidth, windowHeight`
  Which resulted in library saving window width and height sizes twice, instead of once. Now this was fixed.

- **rs.setMode()** - added input sanity check.

- All functions that do some sanity check now should report properly where error happened instead of "error happened somewhere in resolution_solution.lua"

Removed:

- I will no longer provide minified version of library. (I'm not sure why I did provided minified version on 1st place anyway.)

- ZeroBrane API was removed from repository. (Since this API doesn't work very good and I don't use ZeroBrane much.)

- Removed history for demo from repository. (It was not that useful anyway.)

- Removed "releases" from repository. (Because I don't want to maintain it.)

- Removed "documentation" from codebase. (Since now all documentation moved to newly added PDF documentation).