Resolution Solution

v2002

Contents

Variables

rs.scaleMode

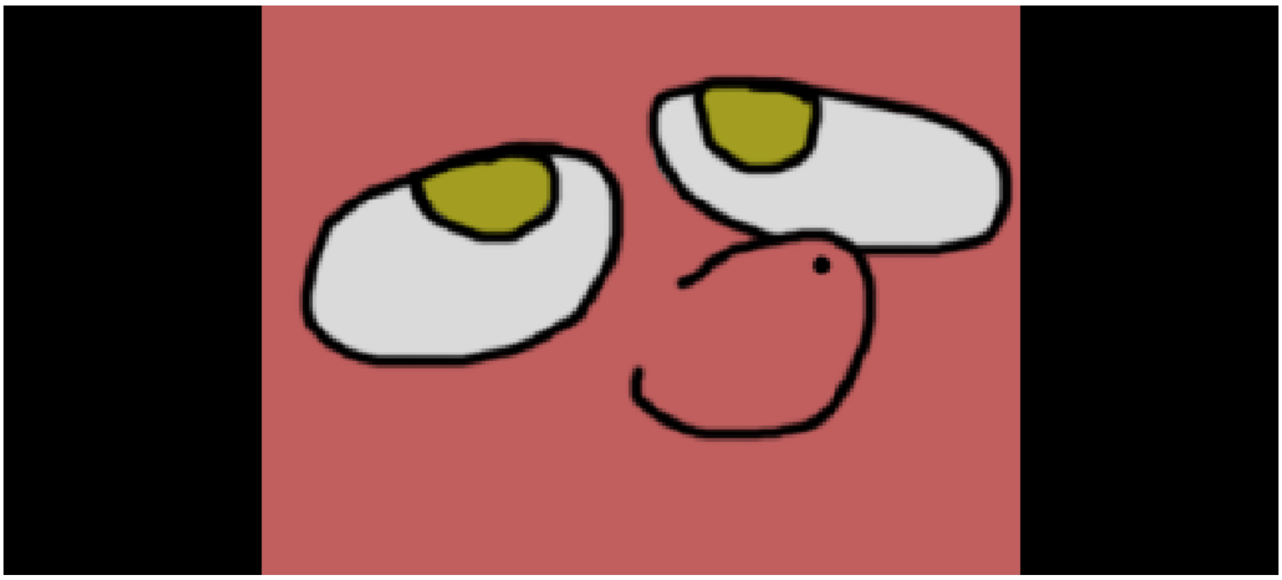
- Integer
- 1 — aspect scaling; 2 — stretch scaling; 3 – pixel perfect scaling;
- Default = 1

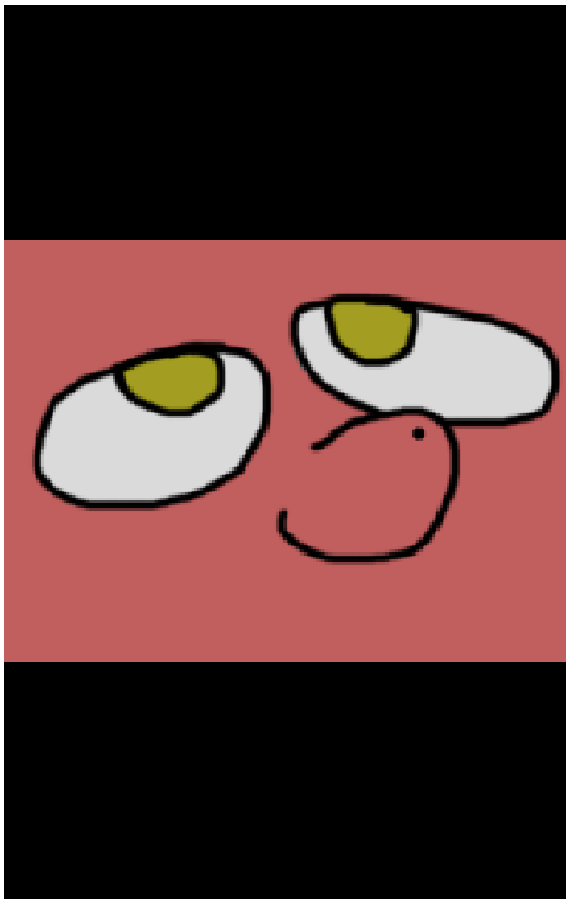
How library will calculate and scale content.

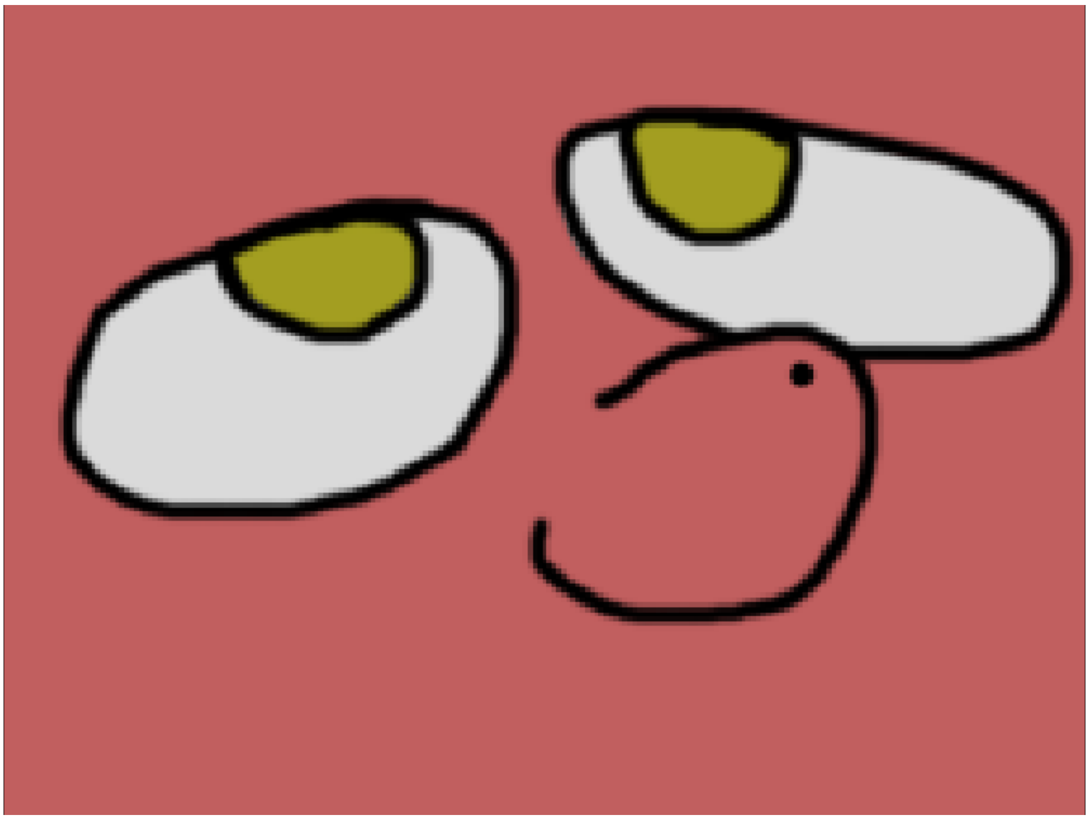
To change this value, use **rs.setScaleMode()** function. Manually changing this variable won’t update library, until **rs.resize()** is called.

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 gameHeight, then black bars will be on top and bottom.

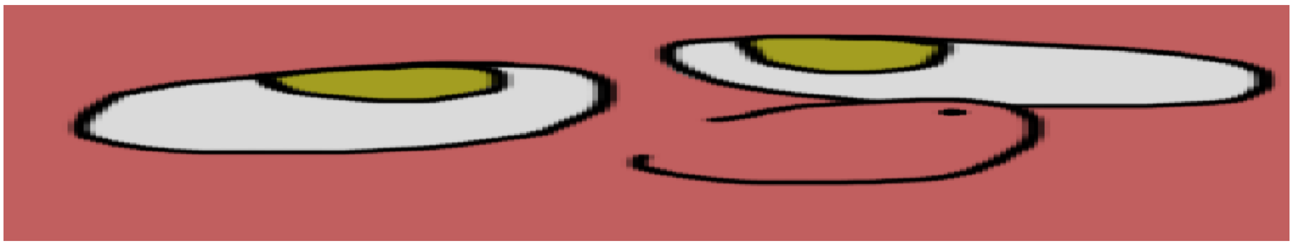

If game aspect and window aspect same, then there will be no visual black bars.

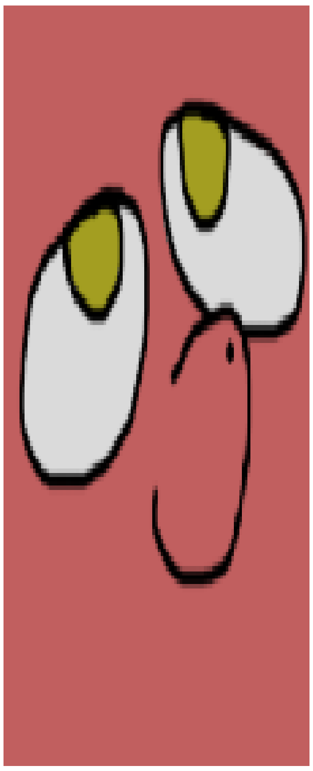
This scaling mode will fit most games with different art-styles, like vector, hand-draw, 3d renders, etc, due to fact that in this mode game doesn’t lost proportions when game size is not same as window size. The only exception might be is pixel art, where pixel bleeding would ruin your game look (but, 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 scalled to fill entire window, no matter what size it has.





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.

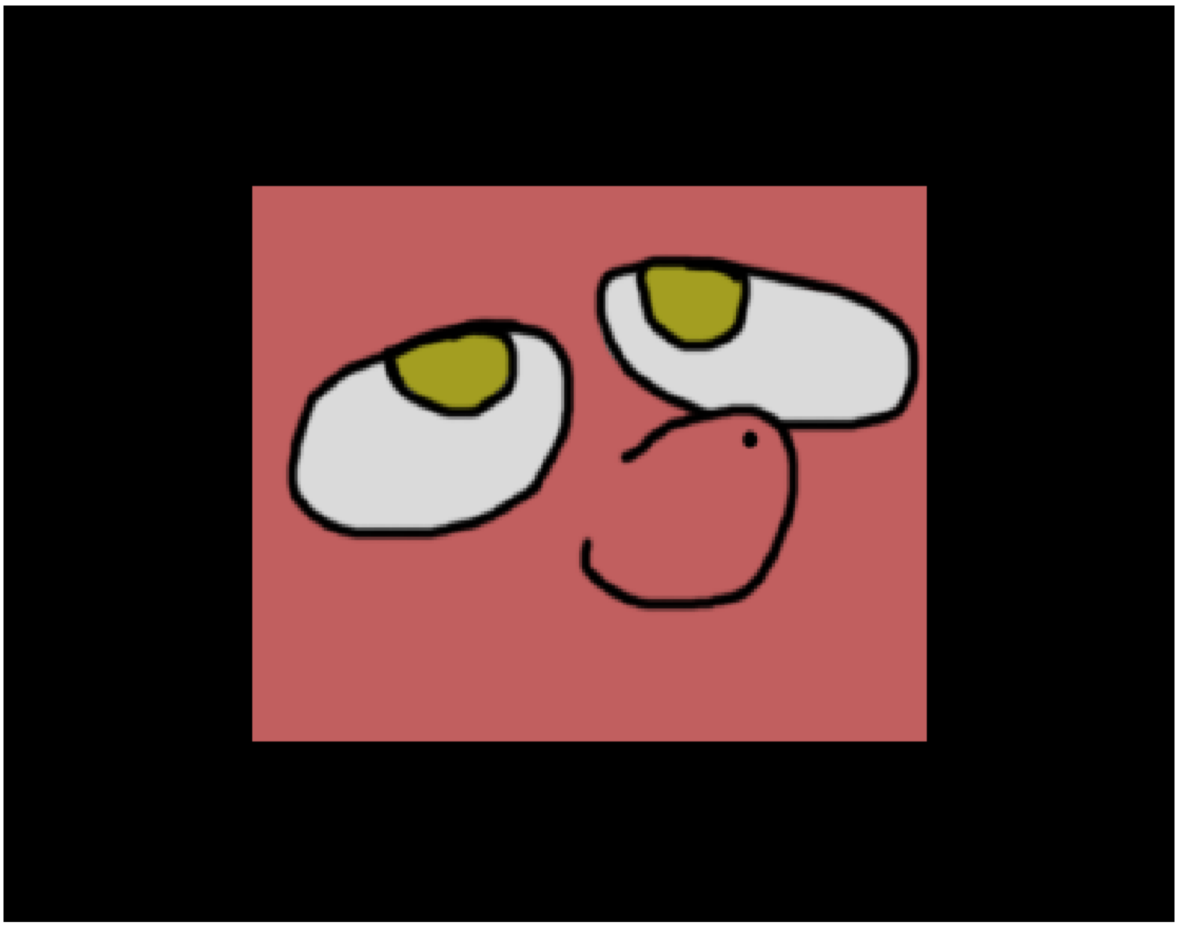
The downside is game will look distorted if game aspect is very different from window aspect, as you can see on screenshots above.

3 Pixel Perfect Scaling

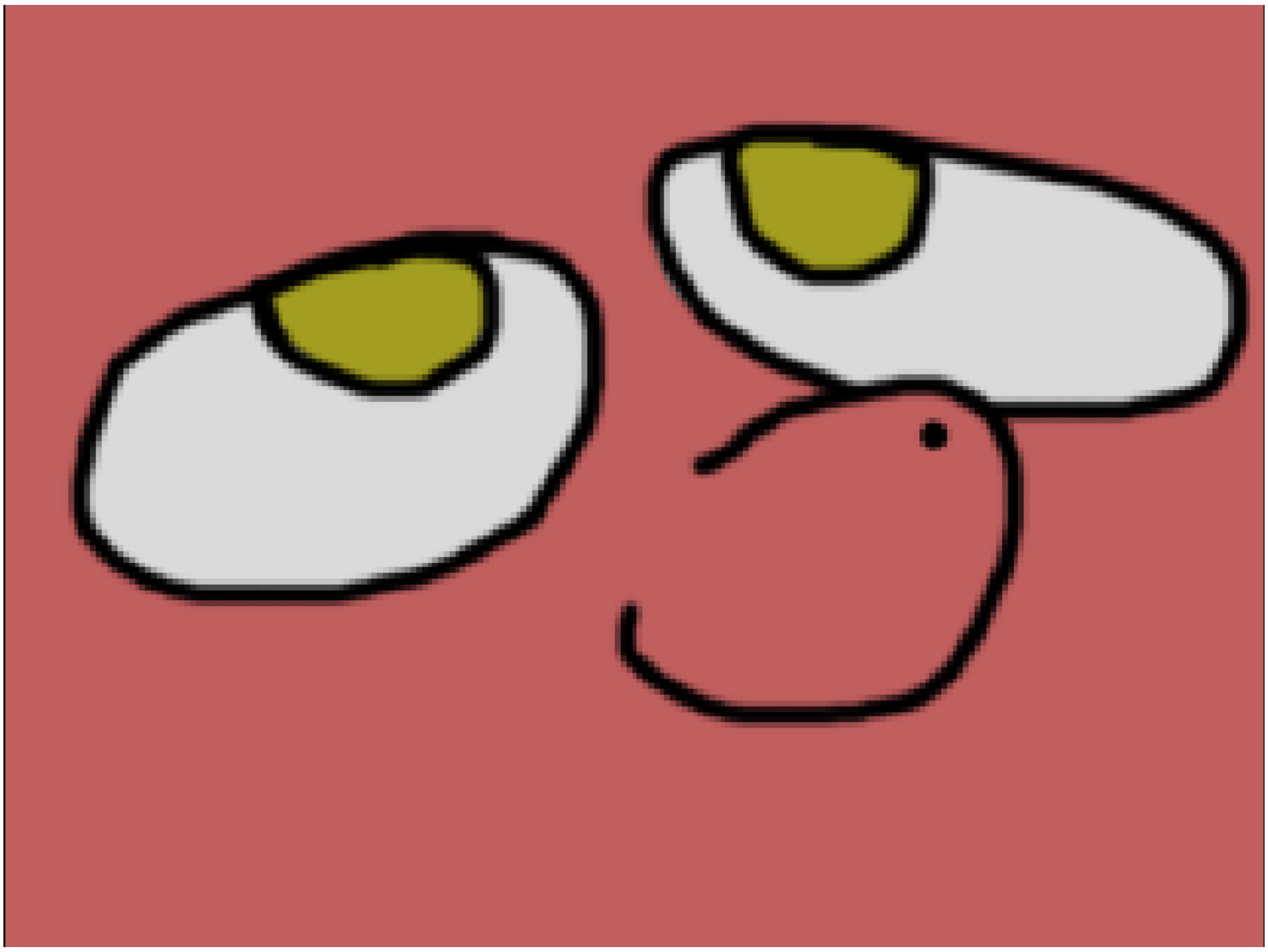
As was mentioned before, Pixel Perfect scaling is *perfectly* suited for games with pixel-art.

It will make sure, that calculations result in **int** 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
- true; false
- Default = **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
- true; false
- Default = **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.

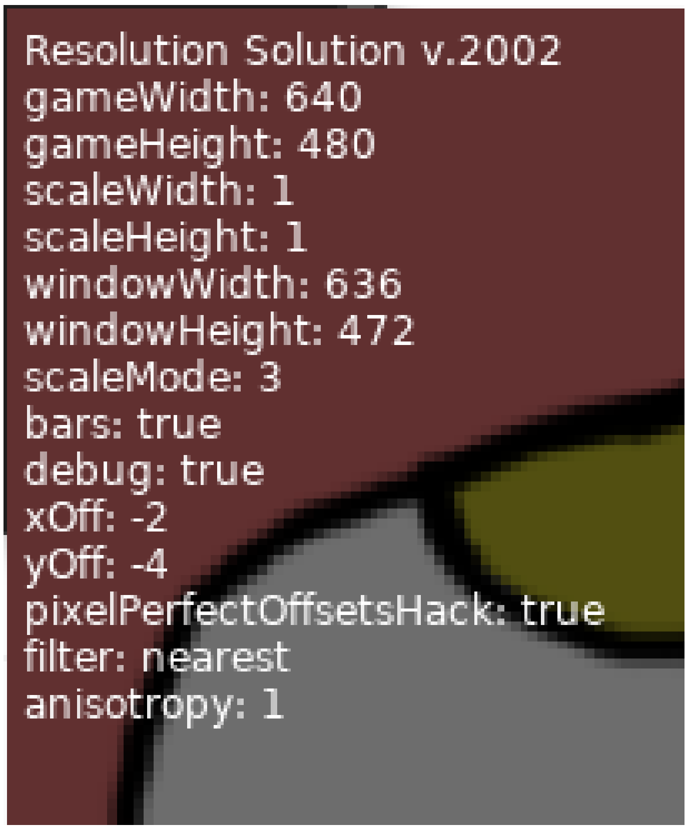
It make sense to disable them if:

- 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
- true; false
- Default = **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.



```
Resolution Solution v.2002
gameWidth: 640
gameHeight: 480
scaleWidth: 1
scaleHeight: 1
windowWidth: 636
windowHeight: 472
scaleMode: 3
bars: true
debug: true
xOff: -2
yOff: -4
pixelPerfectOffsetsHack: true
filter: nearest
anisotropy: 1
```

This debug info includes black, transparent background so text will be more visible.

rs.scaleWidth

- float

Represent scaling

rs.scaleHeight

rs.gameWidth

rs.gameHeight

rs.windowWidth

rs.windowHeight

rs.xOff

rs.yOff

Black Bars coordinates

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.

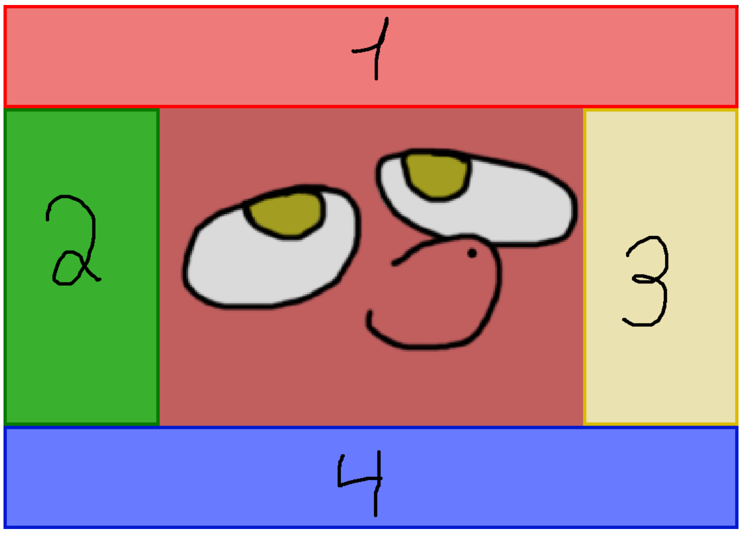
rs.x1, rs.y1, rs.w1, rs.h1 - 1st bar.

rs.x2, rs.y2, rs.w2, rs.h2 -2nd bar.

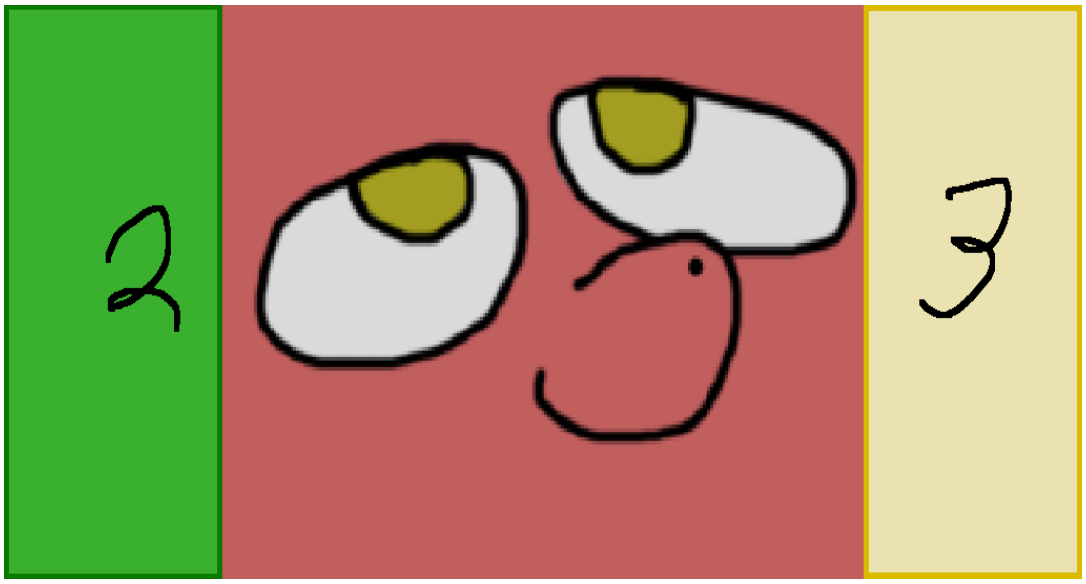
rs.x3, rs.y3, rs.w3, rs.h3 - 3rd bar.

rs.x4, rs.y4, rs.w4, rs.h4 4th bar.

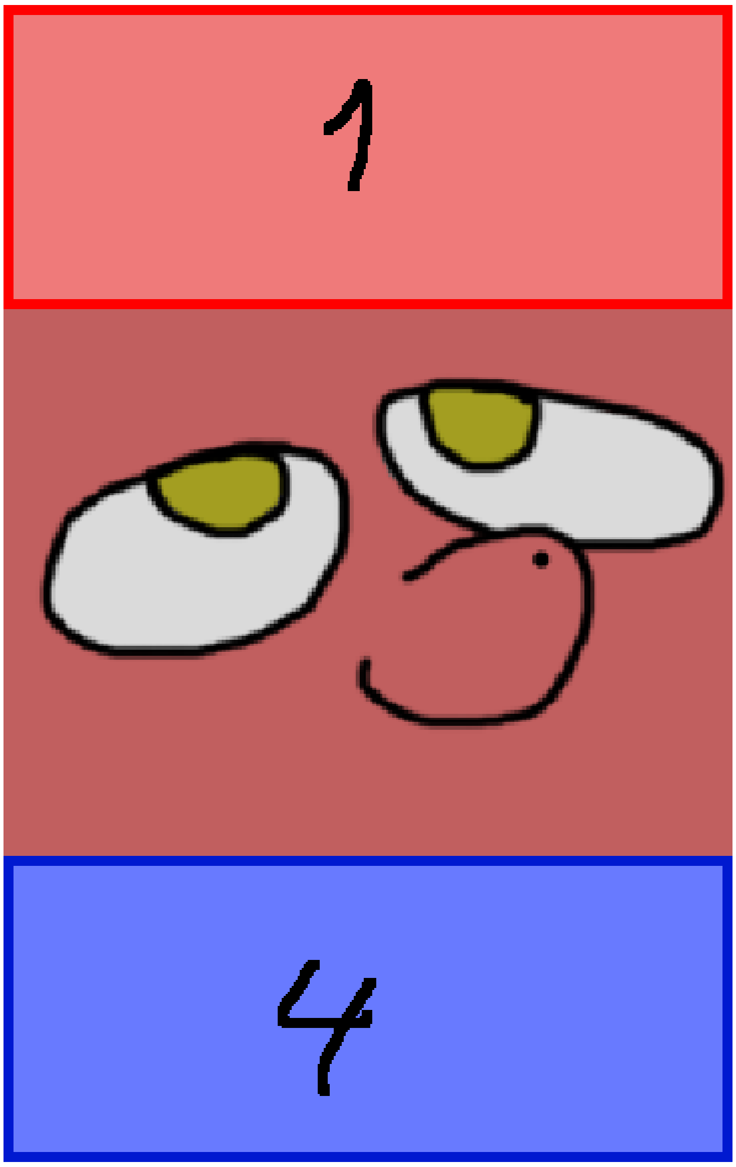
Here visualisations:



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

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
```

rs.r — Red component; float; from 0 to 1; default 1;

rs.g — Green component; float; from 0 to 1; default 1;

rs.b — Blue component; float; from 0 to 1; default 1;

rs.a — Alpha channel; float; from 0 to 1; default 0;

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

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

rs.gameZone

Functions

rs.init(options: table or nil)

- Arguments: 1 – **table** or **nil**
- 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 initialise it. You can run this function without any arguments to update library, it will use variables it already has.

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: number, int
- height: number, int
- bars: boolean — true, false
- debug: boolean — true, false
- mode: number, int — can be 1, 2 and 3.

  If you pass something other then 1, 2 or 3, library will raise error.
- r: number, float — from 0 to 1
- g: number, float — from 0 to 1
- b: number, float — from 0 to 1
- a: number, float — from 0 to 1
- hack: boolean — true, false

rs.setScaleMode(mode: int)

- Arguments: 1 – **int**, can be 1, 2 and 3
- Returns: nothing

Refer to «rs.scaleMode» for more info about scaling modes.

Get 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. Use this function instead.

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(side: int or nil)

- Arguments: 1 **– int**, can be **1** or **-1**; Can be also **nil**
- Returns: nothing

Get 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. Use this function instead.

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.switchPixelHack()

rs.switchBars

rs.debugFunc

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

rs.switchDebug

rs.nearestFilter

rs.resize

rs.start

rs.stop

rs.unscaleStart

rs.unscaleStop

rs.setColor

rs.getColor

rs.getGameZone

rs.defaultColor

rs.drawBars

rs.getScale

rs.setGame

rs.getGame

rs.getWindow

rs.isMouseInside

rs.toGame

rs.toGameX

rs.toGameY

rs.toScreen

rs.toScreenX

rs.toScreenY

Tips and Tricks

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

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
      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

    -- 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

Version 1000, 7 january 2021

Initial release

Version 1001, 6 february 2022

Changed:

- Now, scaling.stop() will remember color that was set before it and return it back after.
- Fixed typo in "Simple demo"

New:

- Added comments for "Simple demo"
- Added more comments for functions and variables in library.

Version 1002, 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()**.

Version 1003, 12 february 2022

New:

- Added library license text in **rs._LICENSE_TEXT**
- Added auto-completion API for Zerobrane Studio!

Changed:

- Updated comments.

Version 1004, 19 may 2022

Renamed:

- **rs.widthScale -> rs.scaleWidth**
- **rs.heightScale -> rs.scaleHeight**

Version 1005, 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

Version 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.

Version 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:

- 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

Version v2001 31 december 2022

Small update, that add some QoL features, and hack for Pixel Pefrect 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 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()**.

Updated:

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

Version v2002 24 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:

- **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.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.)

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.)