Resolution Solution

v3000

Contents

Explanation of terms

- **Game size** / **resolution** / **game content** / **virtual size** - the size of the game that was intended by the software developer (you).
- **Window** - the game is rendered to a window. The window can be resized by the user to be larger or smaller than the intended size of the game. Some platforms use different terms, but it is the object the game is rendered to.
- **rs table** - the Resolution Solution table that is exposed by this library. Many settings and values can be accessed via **rs** functions but they can also be set directly in the **rs** table. This manual uses "**rs table**" meaning the table provided by this library.
  Example:
  `rs.scaleMode = 2`
- **Aspect** / **aspect ratio** - means the size of the game compared to the size of the window as a ratio. Example: a game of 800 x 600 that is rendered in a window that is 1600 x 1200 has an aspect ratio 2:1.

Variables

rs.scale_mode

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

A setting that tells the library how it should scale the game to the window.

To change this setting, use the **rs.conf()** function.

For advanced users:

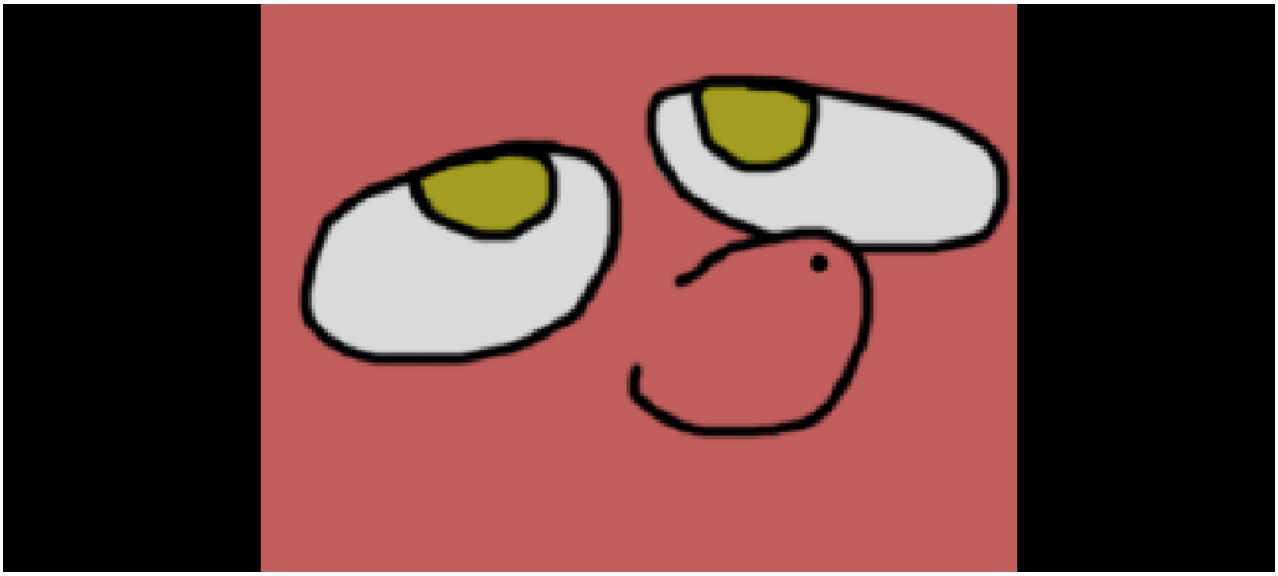
If you manually change this setting (**rs.scale_mode**) directly from the table then you need to call **rs.resize()** immediately after that to update the library.

Example:

```
rs.scale_mode = 2
rs.resize()
```

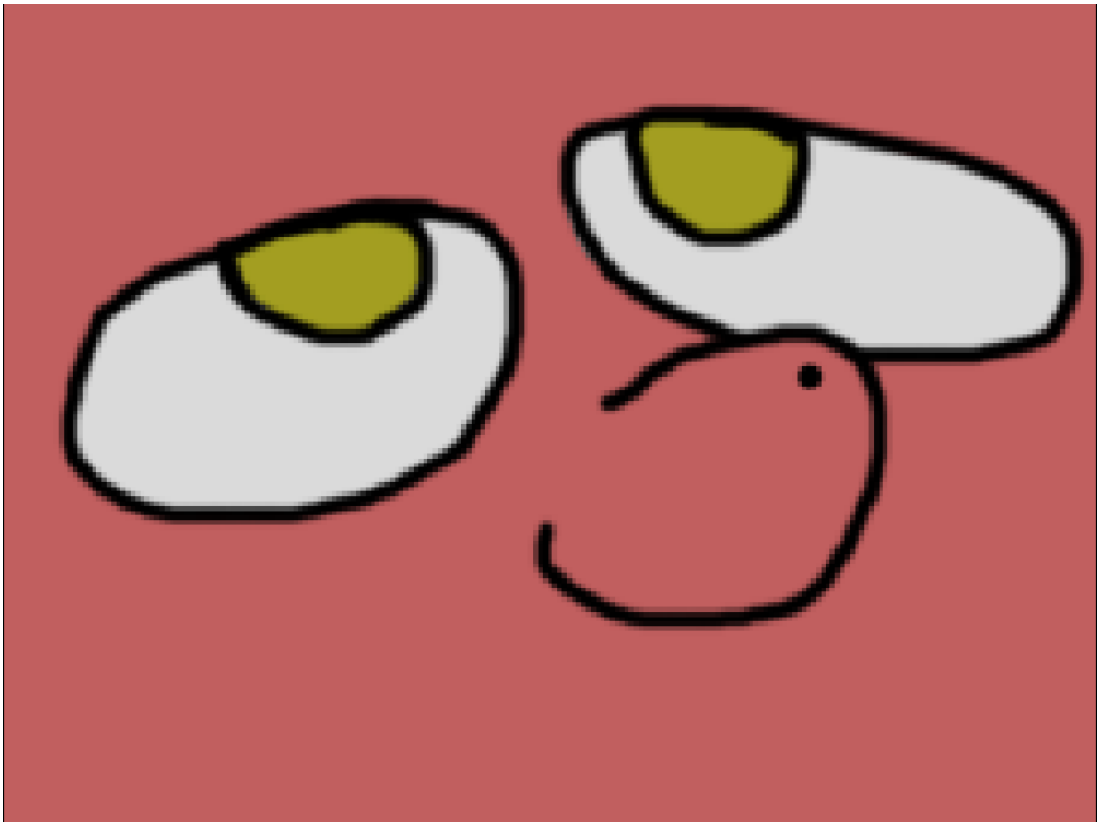
1 – Aspect Scaling

Aspect scaling will ensure **rs.scale_width** and **rs.scale_height** have the same value so images look proportional and not skewed.



If the window is wider than the game width then the library will draw bars on the left and right side of the window.

If the window is taller than the game height then the black bars will be drawn on the top and bottom.



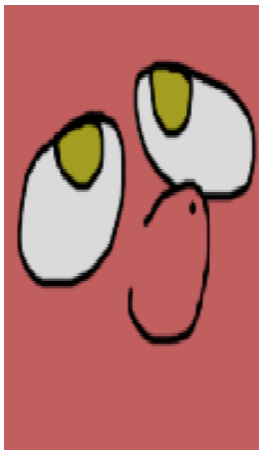
If game aspect and window aspect same, then there will be no visual black bars.

This mode might be best when graphical assets need to remain proportional. Assets that include vector, hand-drawn, 3D etc might look best with this setting. Pixel art might look best with setting **3**.

2 – Stretch Scaling

Game content will be scaled to fill the entire window even if it skews the game content.

The game might look funny when the window is larger than the game.



The game might look funny when the window is larger than the game.

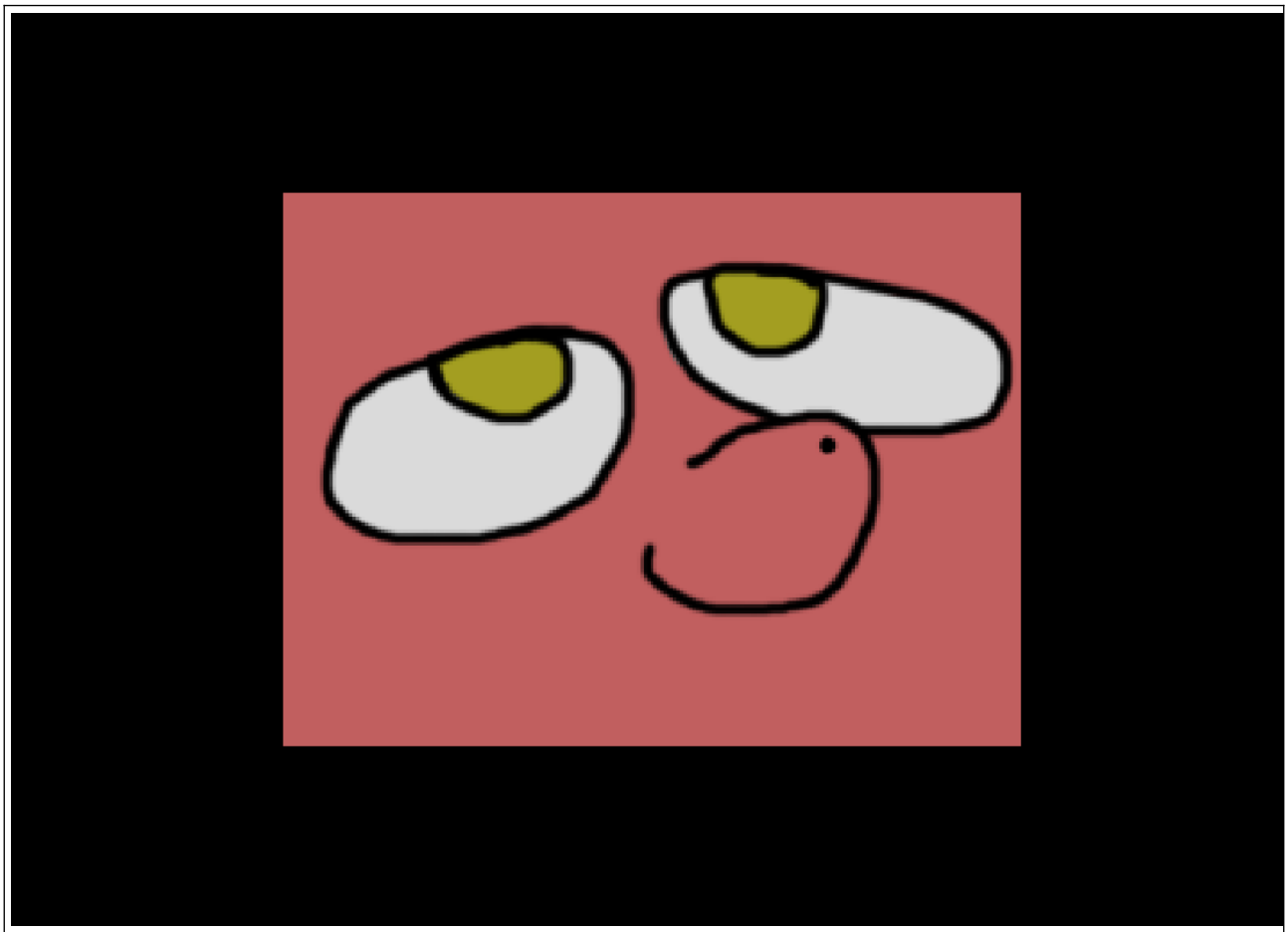
Aspect scaling will never display black bars might give strange results.

3 – Pixel Perfect Scaling

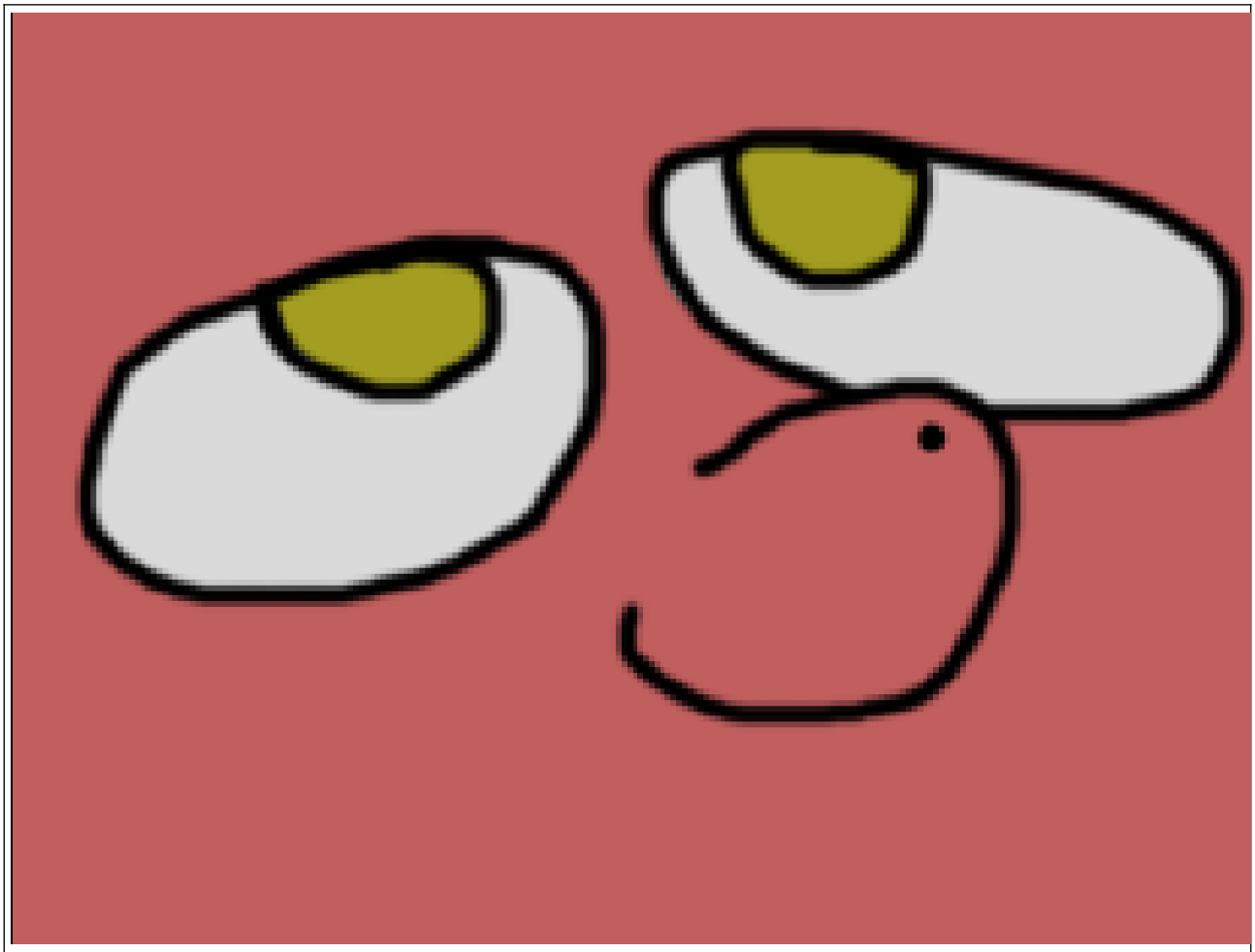
As was mentioned before, Pixel Perfect scaling is *perfectly* suited for games with pixel-art.

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.scale_width [number], rs.scale_height [number]

- **rs.scale_width**
  - Type: **number**.
- **rs.scale_height**
  - Type: **number**.
- Default values: *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.scale_width** and **rs.scale_height** 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.

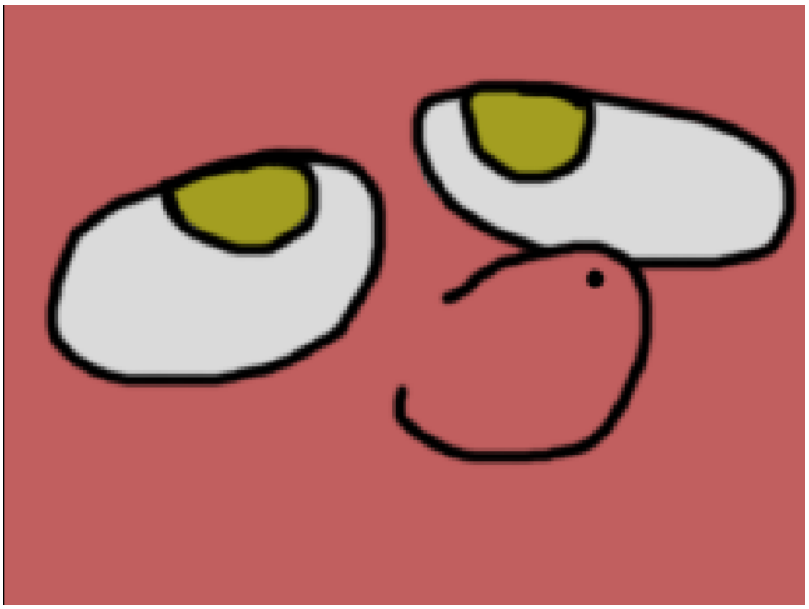
You can use **rs.get_both_scales()** to get both **rs.scale_width** and **rs.scale_height** at same time.

rs.game_width [number], rs.game_height [number]

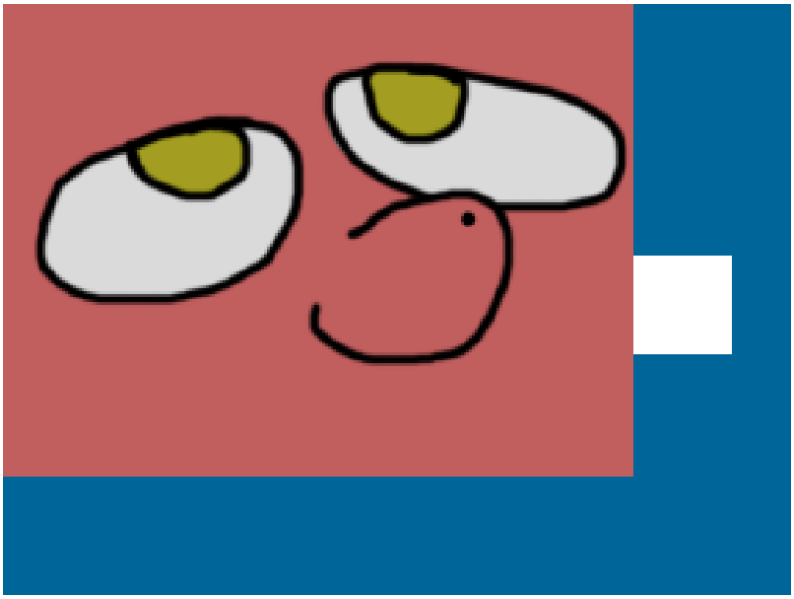
- **rs.game_width**
  - Type: **number**.
  - Default: **800**
- **rs.game_height**
  - Type: **number**.
  - Default: **600**

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.

You can get both **rs.game_width** and **rs.game_height** in single function call via **rs.get_game_size()**



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!*").

rs.x_offset [number], rs.y_offset [number]

- **rs.x_offset**
  - Type: **number**.
- **rs.y_offset**
  - Type: **number**.

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

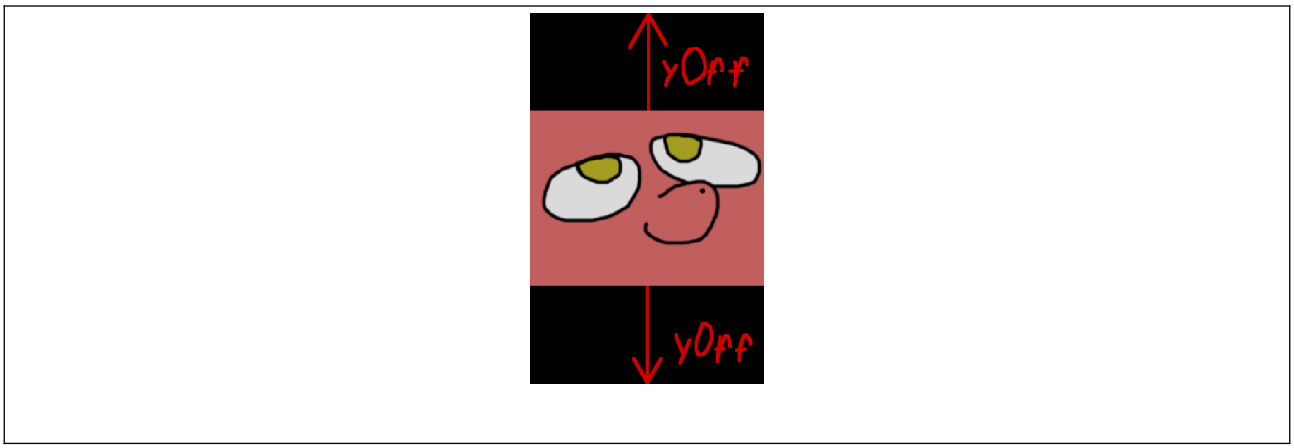
This variables represent width and height for black bars. For example, with mode 1 (Aspect Scaling) if window width **bigger** then game width, you will get 2 black bars on left and right sides. And **rs.xOff** will represent width for black bar, but keep in mind, that there will be **2** black bars,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

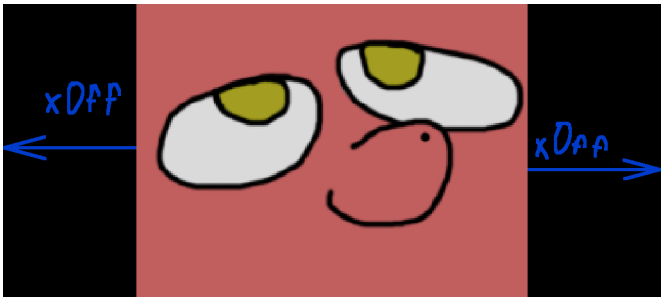
In mode **1** (**Aspect Scaling**) there will be only 2 bars at time. If window width **bigger** then game width, then there will be bars on left and right sides. And if window height **bigger** then game height, then there will be bars on top and bottom. And unless window has same aspect as game window (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.

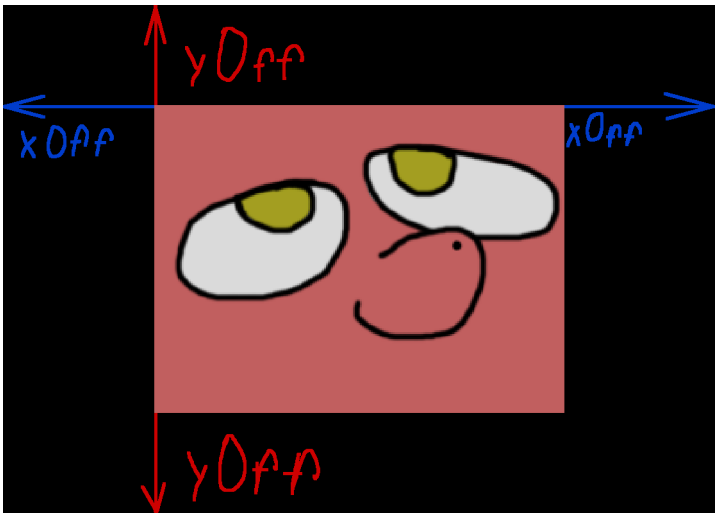
In mode **2** (**Stretch Scaling**) there no bars at all, so there **rs.xOff** and **rs.yOff** always will be **0**.

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:

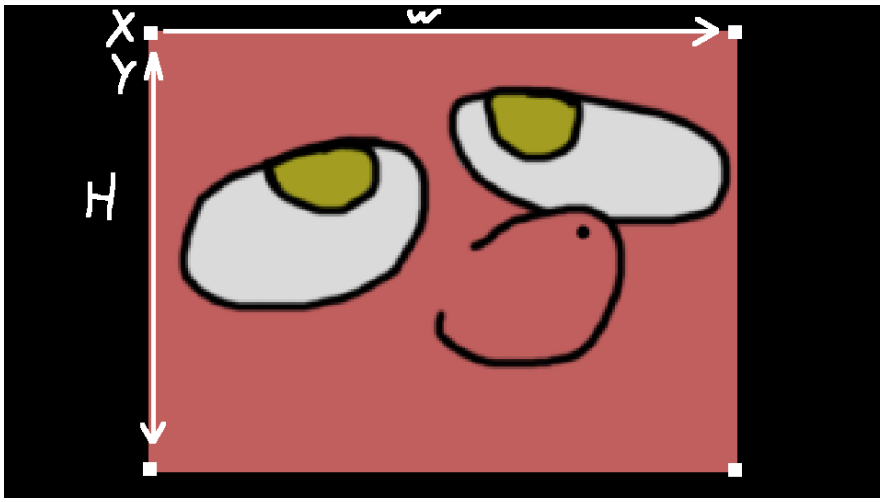



xOff
xOff


yOff
xOff
xOff
yOff

rs.game_zone [table]

- Type: **table**.
  - Table contains:
    - **x** — X coordinate of top-left corner of game zone.
      - Type: **number**.
    - **y** — Y coordinate of top-left corner of game zone.
      - Type: **number**.
    - **w** — width of game zone.
      - Type: **number**.
    - **h** — height of game zone.
      - Type: **number**.



***X** and **Y** is top-left corner.*
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.game_zone** 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.get_game_zone())
```

Functions

rs.get_game_zone()

- 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 values, get them directly from: **rs.game_zone.x**, **rs.game_zone.y, rs.game_zone.w, rs.game_zone.h**.

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

Example:

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

rs.get_game_size()

- Arguments: none.
- Returns:
  1. **rs.game_width**
     - Type: **number**.
  2. **rs.game_height**
     - Type: **number**.

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

Example:

```
-- Basic usage.
local game_width, game_height = rs.get_game_size()
```

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

- Arguments:
  - **width**
    - Type: **number**.
  - **height**
    - Type: **number**.
  - **flags**
    - Type: **table**.
- Returns: nothing.

This function is wrapper for **love.window.setMode()** that you should use with this library.

Wiki page for for **love.window.setMode()**: https://love2d.org/wiki/love.window.setMode

For advanced users:

```
-- This wrapper function equivalent of doing this:
love.window.setMode(width, height, flags)
rs.resize()
```

Example:

```
-- Basic usage.
rs = require("resolution_solution")
rs.conf({game_width = 800, game_height = 600, scale_mode = 3}) -- We want to
create game with 800x600 resolution
-- But we also want to change window size to be same as game.
-- To achieve this, we can do...
rs.setMode(rs.get_game_size(), select(3, love.window.getMode()))
-- Done, now we initialized game with 800x600 game size and resized window to be
same as game width and height.
```

rs.debug_info(x: number [optional], y: number[optional])

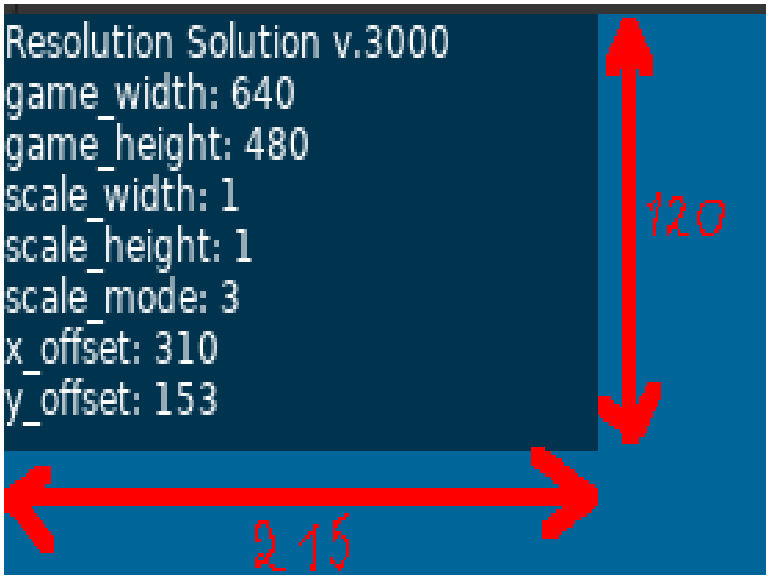
- Arguments:
  1. **x**
     - type: **number**.
     - If you pass nothing or **nil**, then it will same as if you pass **0**.
  2. **y**
     - type: **number**.
     - If you pass nothing or **nil**, then it will same as if you pass **0**.
- Returns: nothing.

Basis debug function, that would show info about some states of library. Can be placed somewhere on screen. Calling **rs.debug_info()** without arguments is equivalent of calling **rs.debug_info(**0, 0**)**, which would place debug window in top-left corner of window.

If you need more advanced debugging, you better to write own solution or use another debugging library / tool.

List of information that function shows:

- Library name
- Library version
- rs.game_width
- rs.game_height
- rs.scale_width
- rs.scale_height
- rs.scale_mode
- rs.x_offset
- rs.y_offset



Currently, size of debug window is: width 215, height 120.

Examples:

```
-- Example drawing this debug window on top-right corner in-between rs.push()
and rs.pop()
rs = require("resolution_solution")
love.graphics.setBackgroundColor(1, 0.3, 0.6, 1)
rs.conf({game_width = 640, h = 480})
rs.setMode(640, 480, {resizable = true})

love.resize = function(w, h)
  rs.resize()
end

love.draw = function()
  rs.push()
    rs.debug_info(rs.game_width - 215)
  rs.pop()
end
```

```
-- Example drawing this debug window on bottom-right corner outside rs.push()
and rs.pop()
rs = require("resolution_solution")
love.graphics.setBackgroundColor(1, 0.3, 0.6, 1)
rs.conf({w = 640, h = 480})
rs.setMode(640, 480, {resizable = true})

love.resize = function(w, h)
  rs.resize()
end
```

```
love.draw = function()
  rs.push()
  rs.pop()

  rs.debugFunc(rs.windowWidth - 230, rs.windowHeight - 230)
end
```

rs.resize()

- Arguments: none.
- Returns: nothing.

Function that update library. In normal circumstances, you need to place it inside **love.resize()** w, h**)** function, pass it’s arguments and never touch again.

Example:

```
-- Basic usage.
rs = require("resolution_solution")

love.resize = function(w, h)
  rs.resize()
end
```

rs.resize_callback()

- Arguments: none.
- Returns: nothing.

Simple function, that will be called every time, when **rs.resize()** is called. Note, that **rs.resize_callback()** will be called ***after*** **rs.resize()** would finish updating library.

This function can be useful, if you need to hook into library and, for example, update UI. To do so, you need to replace **rs.resize_callback()** and fill it with something that you need.

Note: once you setup this function, you might want to immediately call itself to calculate data that depended on it.

Example:

```
-- Basic usage.
rs = require("resolution_solution")
love.resize = function(w, h)
  rs.resize()
end

rs.resize_callback = function()
  print("Resolution Solution was updated!")
  -- Update some data.
end
rs.resize_callback()
```

rs.push()

- Arguments: none.
- Returns: nothing.

Function that would actually start scaling your game. Don’t forget to check **rs.pop()**.

Example:

```
-- Basic usage.
rs = require("resolution_solution")

love.resize = function(w, h)
  rs.resize()
end

love.draw = function()
  rs.start()
    -- draw here something that you want to be affected by scaling
  rs.stop()
end
```

*Important note: **rs.push()** and **rs.pop()** relies on **love.graphics.push()** and l**ove.graphics.pop()**. This might cause conflict with other libraries, that also relies on **push** and **pop** functions, due to fact, that they will cancel each other. Most notable example is camera libraries. Lets dive deep:*

```
rs = require("resolution_solution")
camera_library = ("generic_camera_library")

-- most camera libraries initializes like this:
local camera = camera_library.new(0, 0, 2000, 2000) -- usually, here you specify
boundaries that camera cannot pass through

love.update = function(dt)
  camera:update() -- usually cameras updated like that.
end

love.resize = function(w, h)
  rs.resize()
end

-- And here important point
love.draw = function()
  rs.push()
    camera:push()
      -- draw something that supposed to be subject of camera.
    camera:pop()
  rs.pop()
end
```

And… this will fail. What **rs.push()** do is:

```
love.graphics.pop()
love.graphics.origin()
-- other translations that irrelevant to example.
```

And camera libraries to pretty much same thing with **camera:push()**:

```
love.graphics.pop()
love.graphics.origin()
-- other transformations that irrelevant here.
```

If you remember what **love.graphics.push()-pop()** combo do... https://love2d.org/wiki/love.graphics.pop and https://love2d.org/wiki/love.graphics.push

...They will restore transformations to default inside push-pop combo. So all scale and offset transformations done by **rs.push()** will be gone once **camera:push()** will be called.

There exist way to workaround this. Canvases! (https://love2d.org/wiki/Canvas)

Idea here is to draw everything that camera wants to canvas and then scale this canvas in **rs.push()** and **rs.pop()**. There tons ways to achieve this, but let's make simple example:

```
rs = require("resolution_solution")
camera_library = ("generic_camera_library")

local camera = camera_library.new(0, 0, 2000, 2000)

love.update = function(dt)
  camera:update() -- usually cameras updated like that.
end

love.resize = function(w, h)
  rs.resize()
end

local test_canvas = love.graphics.newCanvas(rs.game_width, rs.game_height)

love.draw = function()
  camera:push()
  -- Select our canvas that will be affected by camera.
    love.graphics.setCanvas(test_canvas)
  -- Clear canvas from what was there in last frame
    love.graphics.clear(0, 0, 0, 0)
  -- Draw something that we want to be affected by camera.

  -- Once we done, deselect canvas.
  love.graphics.setCanvas()
  -- And detach camera, since we don't need it anymore.
  cam:pop()
```

```
  -- Start scaling
  rs.push()
 -- And then simple draw that canvas!
  love.graphics.draw(test_canvas)

  -- Draw everything else that should be affected by scaling library, but not by
camera!
  rs.pop()
end
```

rs.pop()

- Arguments: none.
- Returns: nothing.

Function that closes **rs.push()**. For more info look for **rs.push()**.

Example:

```
-- Basic usage.
rs = require("resolution_solution")

love.resize = function(w, h)
  rs.resize()
end

love.draw = function()
  rs.push()
    -- draw here something that you want to be affected by scaling
  rs.pop()
end
```

rs.get_both_scales()

- Arguments: none.
- Returns:
  1. **rs.scale_width**
     - Type: **number**.
  2. **rs.scale_height**
     - Type: **number**.

Shortcut function, that would return both **rs.scale_width** and **rs.scale_height**.

Example:

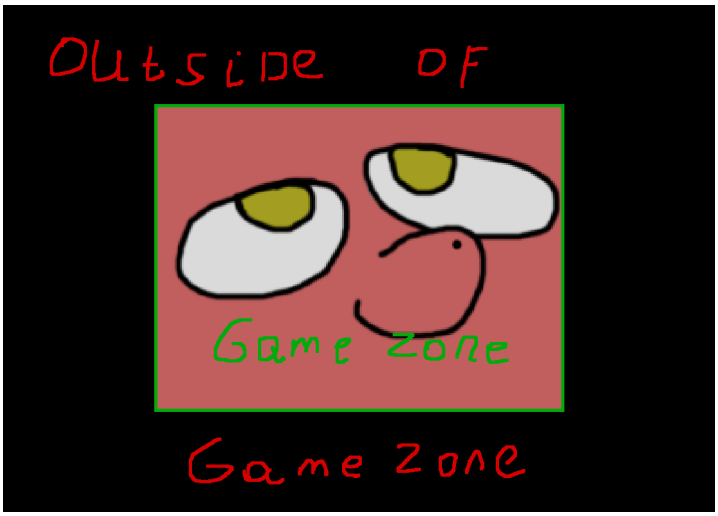
```
-- Basic usage.
local x_scale, y_scale = rs.get_both_scales()
```

rs.is_it_inside()

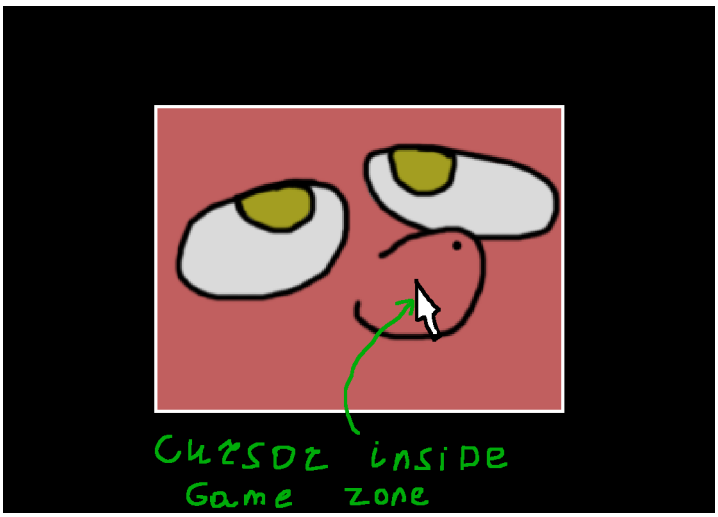
- Arguments: none.
- Returns:
  1. is mouse inside game zone
     - Type: **boolean**.
     - Returns **true** if cursor **inside** game zone.
     - Returns **false** if cursor **outside** of game zone.

Function to determine if given coordinates inside game zone (for more info about look for **rs.game_zone**). You might need it, if you develop game with support of mouse/touchscreen, because you, usually, don't want to trigger buttons (or any other similar element that you can interact with via cursor) if they behind black bars.

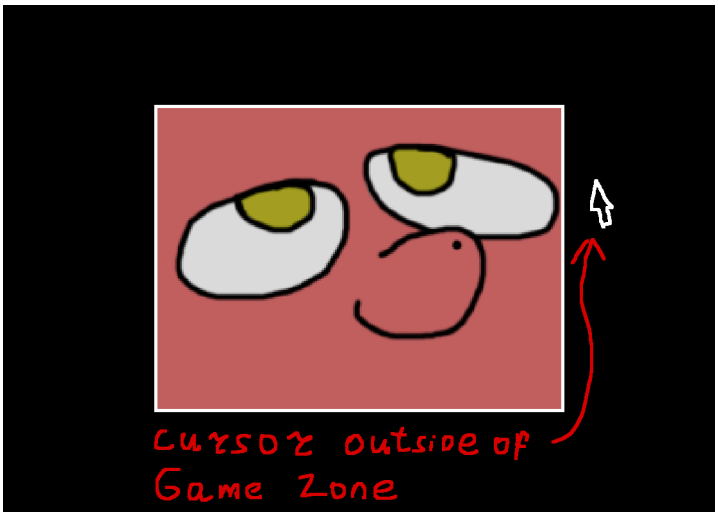
Here quick visualization:



Quick reminder what considered "game zone" and what "outside of game zone".



*Cursor inside game zone, so **rs.is_it_inside(love.mouse.getPosition())** will return **true**.*



*Cursor outside of game zone, so **rs.is_it_inside(love.mouse.getPosition())** will return **false**.*

This especially useful, if you develop game with scrolling/camera, like Real-Time Strategies, because you might move camera in that way, that hit-box of units will be behind where black bars and you don't want game to pick this unit, since user cannot see them!

When scale mode == 2 (Stretch Scaling), then this function will always return **true**, because there no black bars at all.

Example:

```
-- Basic usage.
local rs = require("resolution_solution")
rs.conf({
```

```
    game_width = 640,
    game_height = 480,
    scale_mode = 1
  })
love.graphics.setBackgroundColor(0.3, 0.5, 1)
rs.setMode(rs.game_width, rs.game_height, {resizable = true})

local game_canvas = love.graphics.newCanvas(rs.get_game_size())

local is_inside = false

love.load = function()
  image = love.graphics.newImage("image.png")
end

love.resize = function()
  rs.resize()
end

love.update = function()
  is_inside = rs.is_it_inside(love.mouse.getPosition())
end

love.draw = function()
  love.graphics.setCanvas(game_canvas)
  love.graphics.clear(0, 0, 0, 1)

  if is_inside then
    love.graphics.setColor(0, 1, 0, 1)
    love.graphics.print("Cursor inside game zone.", rs.game_width / 2,
rs.game_height / 2)
  else
    love.graphics.setColor(1, 0, 0, 1)
    love.graphics.print("Cursor outside game zone.", rs.game_width / 2,
rs.game_height / 2)
  end

  love.graphics.setColor(1, 1, 1, 1)
  love.graphics.setCanvas()

  rs.push()
    love.graphics.draw(game_canvas)
  rs.pop()
end
```

rs.to_game(x: number, y: number)

Arguments:

1. **x**
   - Type: **number**.
2. **y**
   - Type: **number**.

Returns:

1. scaled to game **X**
   - Type: **number**.
2. Scaled to game **Y**
   - Type: **number**.

Function to translate coordinates from window to game game zone. This can be used to translate cursor coordinates so you can check collision of object inside game zone.

If you gonna implement cursor support in your game, then you also might want to look for **rs.isMouseInside()** to check if mouse even inside game zone.

Example:

```
-- Basic example.
-- Change size of window and try to touch red rectangle with your cursor!
-- It will become green if it detects you touching it.
local rs = require("resolution_solution")
rs.init({width = 640, height = 480, mode = 1})
rs.setMode(rs.gameWidth, rs.gameHeight, {resizable = true})
love.graphics.setBackgroundColor(0, 0.4, 0.6, 1)

local isTouching = false
local rectangle = {x = 200, y = 200, w = 100, h = 100}

love.resize = function(w, h)
  rs.resize(w, h)
end

love.update = function(dt)
  -- Get cursor position.
  local mx, my = love.mouse.getPosition()
```

```
  -- Translate it to game.
  mx, my = rs.toGame(mx, my)

  -- Check if cursor inside game zone and touches rectangle.
    if rs.isMouseInside() then
      if mx    >= rectangle.x                  and -- left
         my    >= rectangle.y                  and -- top
         mx    <= rectangle.x         +    rectangle.w  and -- right
         my    <= rectangle.y         +    rectangle.h  then -- bottom
        isTouching = true
      else
        isTouching = false
      end
    end

end

love.draw = function()
  rs.start()
    if isTouching then
      -- Green color, means that we touch rectangle.
      love.graphics.setColor(0, 1, 0, 1)
    else
      -- Red color, means that we don't touch rectangle.
      love.graphics.setColor(1, 0, 0, 1)
    end

    love.graphics.rectangle("line", rectangle.x, rectangle.y, rectangle.w,
rectangle.h)
  rs.stop()
end
```

rs.to_window(x: number, y: number)

Arguments:

1. **x**
    - Type: **number**.
2. **y**
    - Type: **number**.

Returns:

1. scaled from game **X**
    - Type: **number**.
2. Scaled from game **Y**
    - Type: **number**.

Function similar to **rs.toGame()** but reversed: it translated coordinates **from** game zone **to** window. For example, if you want to teleport cursor on object that inside game zone.

If you gonna implement cursor support in your game, then you also might want to look for **rs.isMouseInside()** to check if mouse even inside game zone.

Example:

```
-- Basic example.
-- Press left button of mouse to teleport cursor to center or rectangle
-- that you can see on window.
-- Note: might not work if you use Linux with Wayland.
local rs = require("resolution_solution")
rs.init({width = 640, height = 480, mode = 1})
rs.setMode(rs.gameWidth, rs.gameHeight, {resizable = true})
love.graphics.setBackgroundColor(0, 0.4, 0.6, 1)

love.resize = function(w, h)
  rs.resize(w, h)
end

love.mousepressed = function(x, y, button)
   if button == 1 then
      love.mouse.setPosition(
        rs.toScreen(
          (rs.gameWidth / 2), -- Translate X.
          (rs.gameHeight / 2) -- Translate y.
          )
```

```
        )
   end
end


love.draw = function()
  rs.start()
    -- Place 100x100 rectangle in center of game zone.
    love.graphics.rectangle("line", (rs.gameWidth / 2) - 50, (rs.gameHeight / 2)
- 50, 100, 100)
    love.graphics.rectangle("line", (rs.gameWidth / 2) - 4, (rs.gameHeight / 2)
- 4, 4, 4)
  rs.stop()
end
```

Tips and Tricks

This is collection of code snippets and example of useful thing that you could do using this library.

1. Example how you can ditch black bars rendering that library provides and use **love.graphics.setScissor()** instead:

```
local rs = require("resolution_solution")
rs.init({width = 640, height = 480, mode = 1})
rs.setMode(rs.gameWidth, rs.gameHeight, {resizable = true})

-- Disable bars rendering.
rs.bars = false

love.resize = function(w, h)
  rs.resize(w, h)
end


love.draw = function()
  rs.start()
    -- Get scissor that was before.
    local oldScissorX, oldScissorY, oldScissorW, oldScissorH =
love.graphics.getScissor()

    -- Set scissor using coordinates of game zone that library provides.
    love.graphics.setScissor(rs.getGameZone())

    -- Set blue color for rectangle, so we will be able to distinguish between
background color and game zone color.
    love.graphics.setColor(0, 0.4, 0.6, 1)
    -- Draw said rectangle.
    love.graphics.rectangle("fill", 0, 0, rs.gameWidth, rs.gameHeight)

    -- If you wouldn't return old scissor, love might start generate graphical
garbage.
    love.graphics.setScissor(oldScissorX, oldScissorY, oldScissorW, oldScissorH)
  rs.stop()
end
```

---

2. Proper way to get resolution of screen on which window with game placed. Just keep in mind, that there people who uses their system with several monitors.

So for example, if user have 2 monitors: 1 is 1366x768; 2 is 1920x1080. If you would try this:

```
width, height = love.window.getDesktopDimensions(1)
```

You would make annoying (to user) mistake, because user might place game window on 2nd monitor, so game would get 1366x768 instead of 1920x1080 that user might expected.

So, instead do this:

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

This function would get index of screen on which game window and placed and pass it, so now game will correctly get 1920x1080 if it placed on 2nd monitor and 1366x768 if on 1st.

---

3. Good example of formulas for scaling UI outside of rs.start() and rs.stop():

```
-- Example of scaling rectangles.
rectangle = {
               x = rs.gameZone.x,
               y = rs.gameZone.y,
               w = math.min(rs.gameZone.w * 0.04, rs.gameZone.h * 0.04),
               h = math.min(rs.gameZone.w * 0.05, rs.gameZone.h * 0.05)
             }
-- Rectangle that placed on top-left and will take 4% of game zone by width and
5% by height

-- Font
font = love.graphics.newFont(math.min(rs.gameZone.w * 0.04, rs.gameZone.h *
0.04))
-- But be careful with fonts scaling. Very big size for fonts might be very
-- resource consuming (memory usage, CPU usage for resizing fonts, GPU resources
to render font on screen, etc).
```

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 typos in "Simple demo".

v1002, 8 February 2022

Changed / Fixed:

- Fixed (probably) edge cases in **rs.isMouseInside()** function, so now it should correctly deal with non integer offsets provided by **scaling.xOff** and **scaling.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 black bars rendering;

  Or camera (or any other library that relies graphics transformations) 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 available only at **rs.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... Whops!
- 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()** and **love.graphics.getheight()** 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.

v3000, 30 August 2023

Upgrades, people, upgrades!