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.scale_mode = 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:

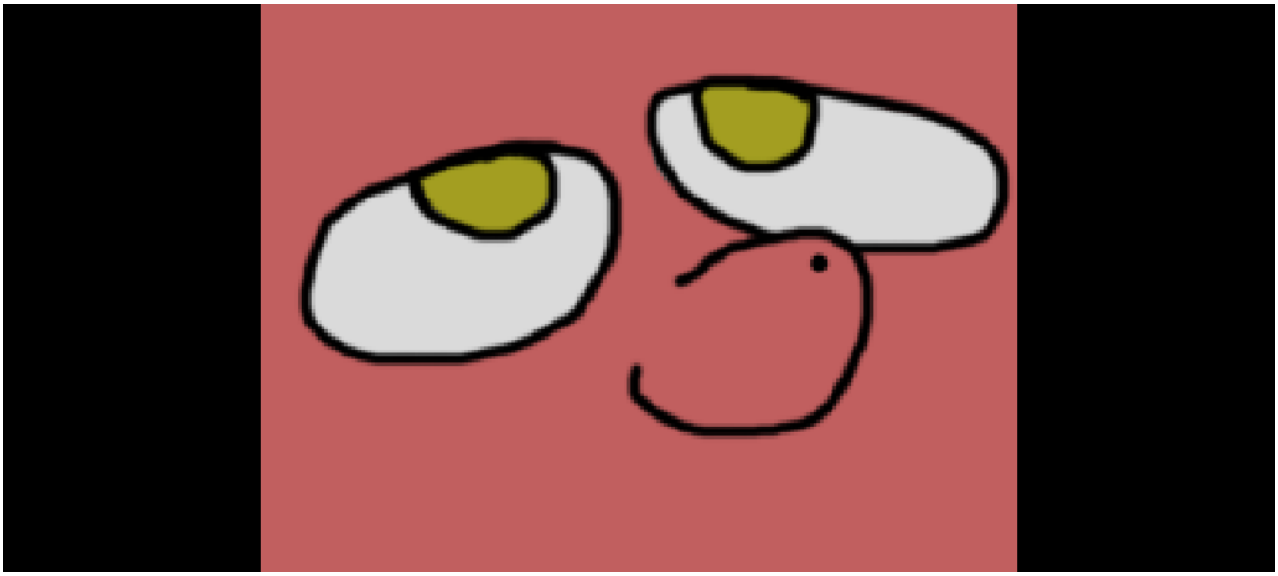
If you manually change this setting (**rs.scale_mode**) directly from the table then you need to call **rs.resize()** immediately after that to update the library.

Example:

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

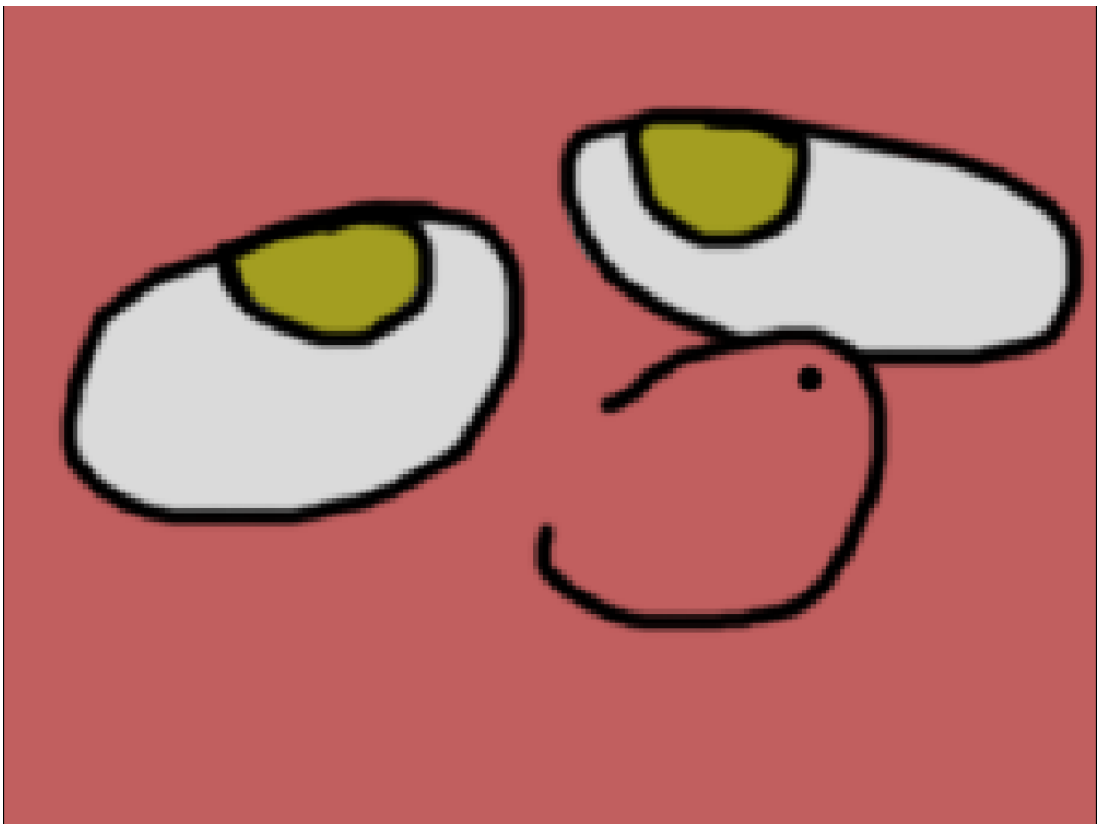
1 – Aspect Scaling

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.



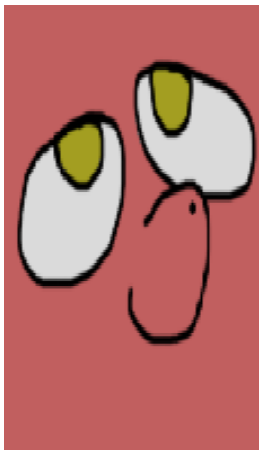
If game aspect and window aspect same, then there will be no visual black bars.

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.



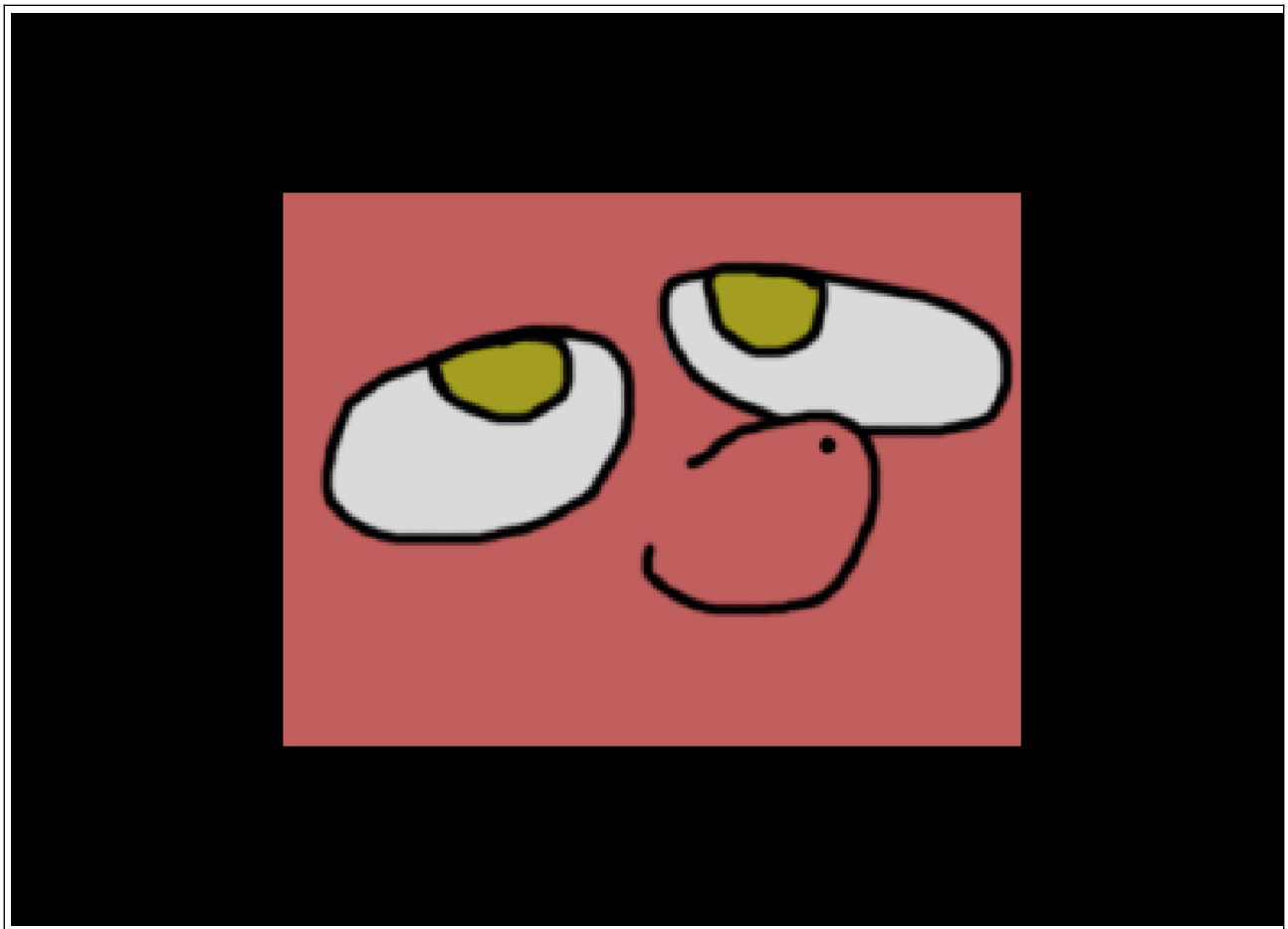
The game might look funny when the window is larger than the game.

Stretch scaling might give strange results.

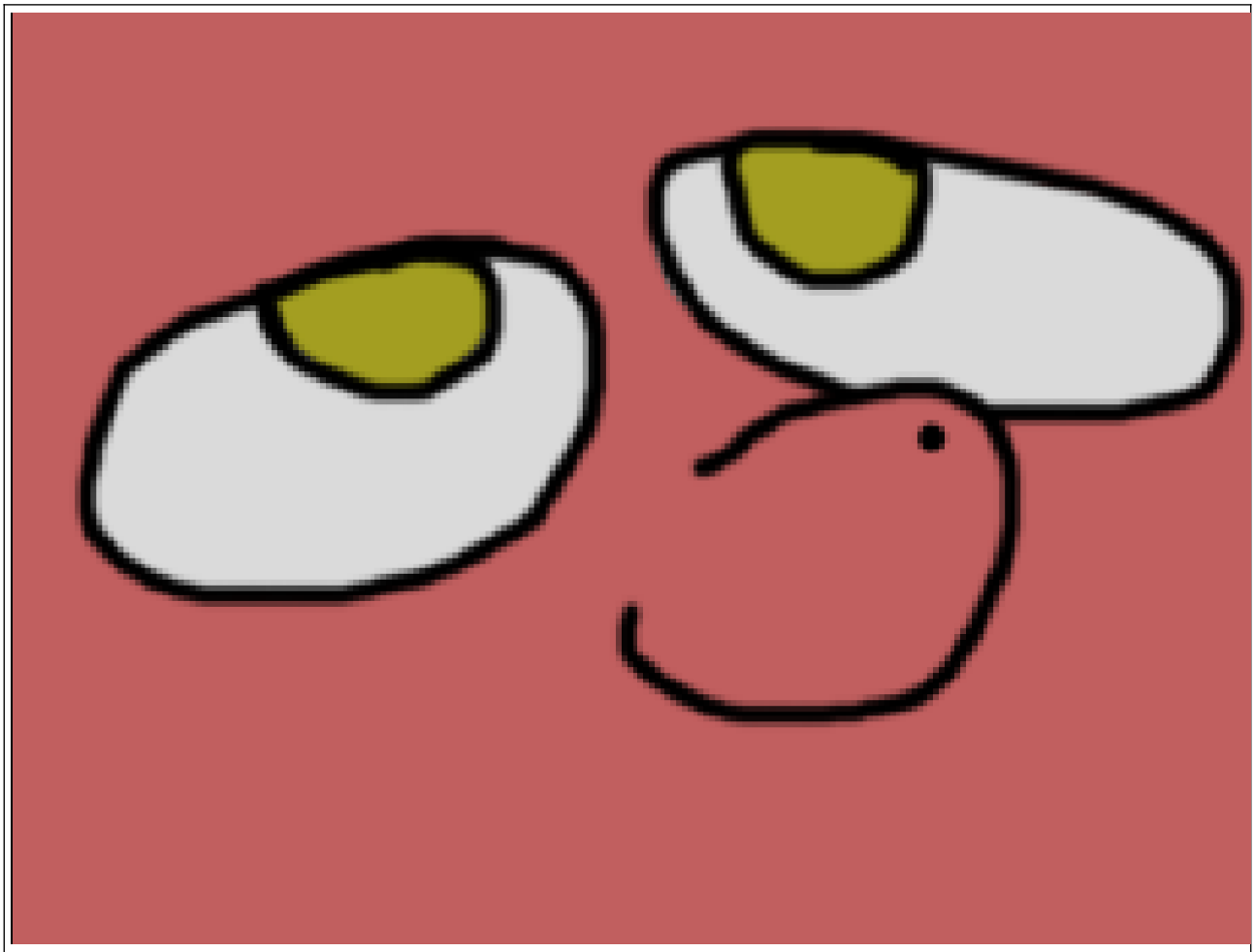
3 – Pixel Perfect Scaling

Pixel Perfect scaling is intended for games with pixel art. It will result in whole-number scaling (1, 2, 3 ... etc) with the lowest value being 1. The scale value will round down to the lowest value (e.g. 1.7 becomes 1.0). Your game content will stay as crisp as possible and there will be no pixel bleeding.

This mode will produce 4 black bars on all sides of the window.



Game size is 640 x 480 while the window is 1111 x 792. Pixel Perfect scaling produces scaling == 1 with 4 black bars. If the window was 1280 x 960 then the scale would be 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 when rs.resize() is first called.*
- This values is read-only, meaning you shouldn’t change them.

These two values depend on the game size and window size and are influenced by the scale mode specified.

In mode 1 and 3, **rs.scale_width** and **rs.scale_height** will have the same value because that scaling uses the same width and height. Mode 2 might result in a different width and height.

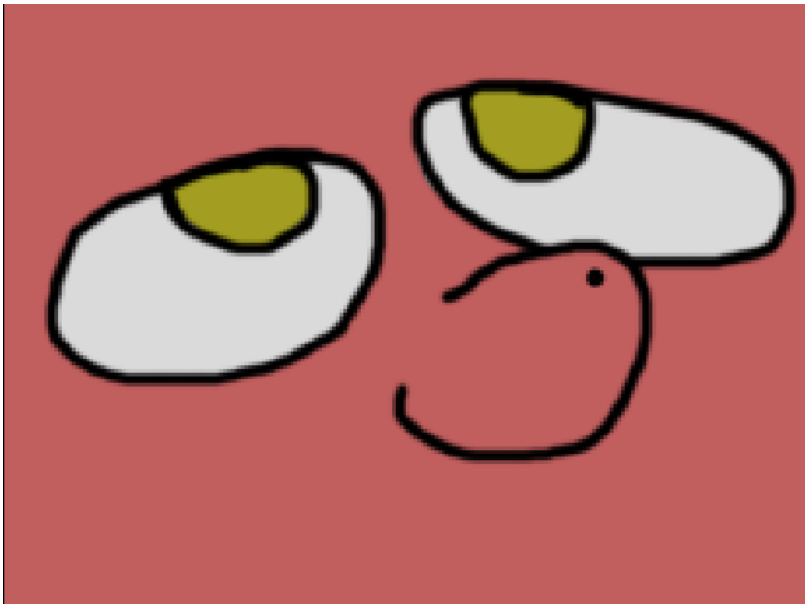
Changing this value manually will break the automated calculations provided by the library during **rs.resize()**.

rs.get_both_scales() will provided the width and height scales in a single call.

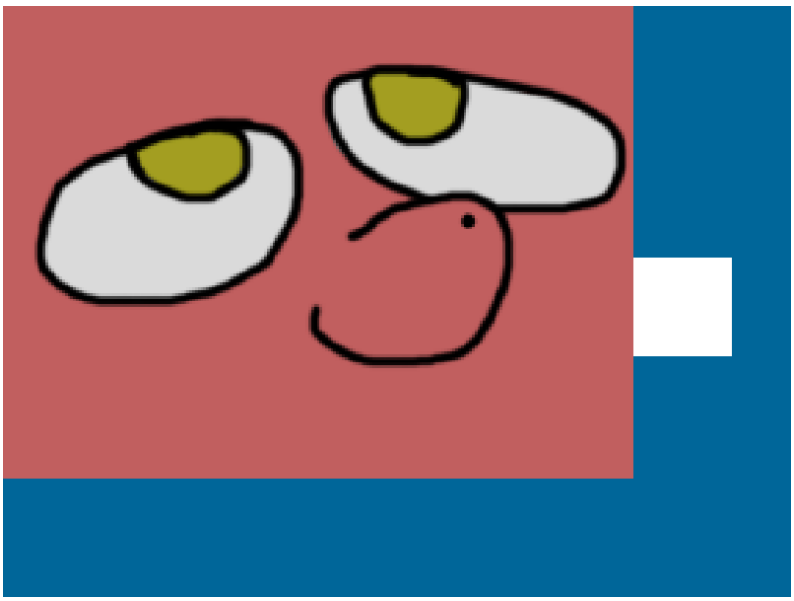
rs.game_width [number], rs.game_height [number]

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

Specifies the width and height values your game is designed for. **rs.get_game_size()** will provided the width and height in a single call.



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.

You need to know if your game world will exist outside the window and make that clear to the player. If something exists outside the window, you need to think about how the player will know that.

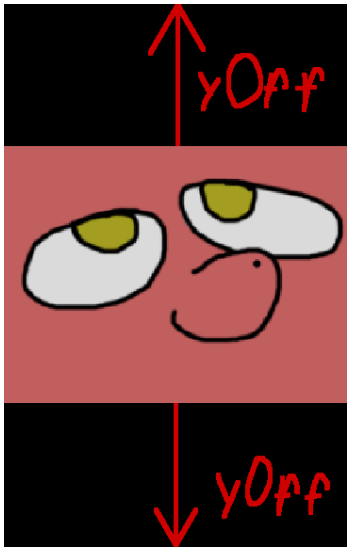
rs.x_offset [number], rs.y_offset [number]

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

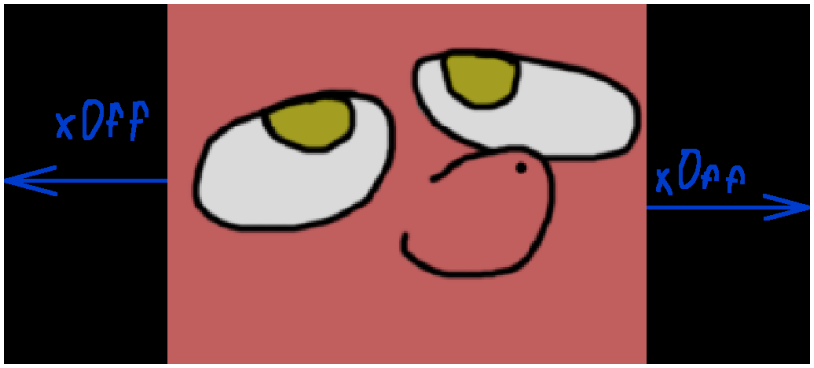
- Default values:*will be calculated when rs.resize() is first called.*

These variables represent offset values that the library generates to position the game inside the window in accordance with scaling mode. Scale mode 2 will create no offset because the game fills the whole window.

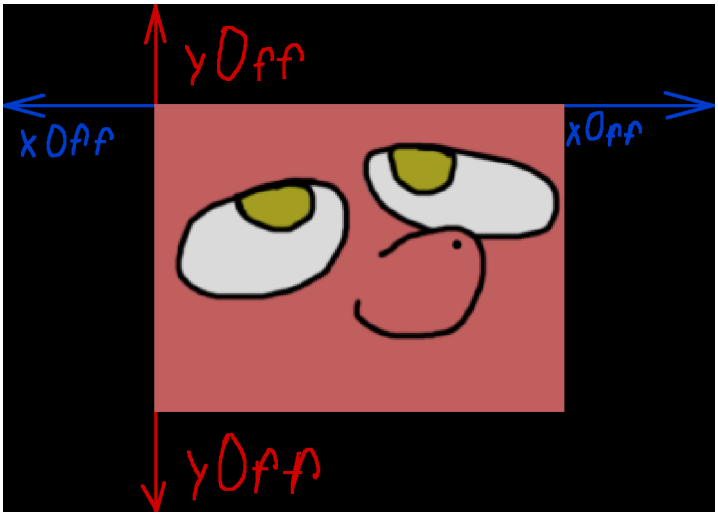
Visualization:



When window taller then game.



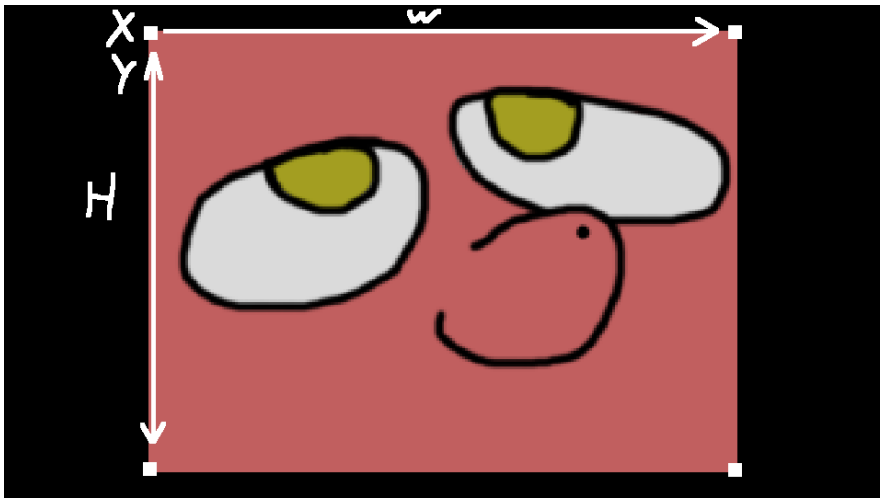
When window wider then game.



In scale mode 3 you would get both horizontal and vertical offsets.

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 the coordinates for the game that is inside the black bars. In other words, the coordinates for what you can see on the screen.

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.

This provides an alternate way to return game zone data. Individual values can be retrieved using **rs.game_zone.x**, **rs.game_zone.y**, **rs.game_zone.w**, **rs.game_zone.h**

Also see 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**.

A shortcut function that will return **rs.game_width** and **rs.game_height**. For more information, see **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 a wrapper for love.window.setMode() that should be used with this library. See the Love2d wiki page for information about 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})
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.

-- Another one:
```

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

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

This function is a wrapper for **love.window.updateMode()** that should be used with this library. See the Love2d wiki page for information about **love.window.updateMode()**: https://love2d.org/wiki/love.window.updateMode

For advanced users:

```
-- This wrapper function equivalent of doing this:
love.window.updateMode(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.updateMode(rs.get_game_size())
-- Done, now we initialized game with 800x600 game size and resized window to be
same as game width and height.
```

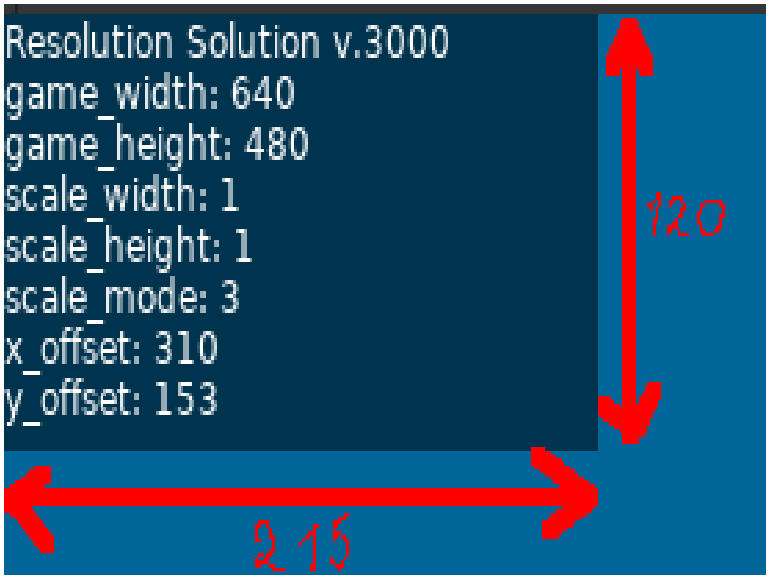
rs.debug_info(x: number [optional], y: number[optional])

- 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.

This is a function to help you debug. It shows information about the library states. You can display these states on the screen to help you troubleshoot. Calling rs.debug_info() without arguments is the same as calling rs.debug_info(0,0) which draws the debug window in the top left corner of the window.

Information displayed:

- 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()
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())

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

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

  rs.push()
    love.graphics.draw(game_canvas)
    rs.debug_info(rs.game_width - 215)
```

```
  rs.pop()
end
```

rs.resize()

- Arguments: none.
- Returns: nothing.

A function that forces the library to update. This is normally called inside love.resize() function.

Example:

```
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())

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

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

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

rs.resize_callback()

- Arguments: none.
- Returns: nothing.

A function that is invoked every time rs.resize() is called. This function is useful if you need to hook into the library. For example, to rescale the UI. To do so, you can write your own rs.resize_callback() and insert your own code inside the function.

Note: rs.resize_callback() is called after rs.resize() is finished updating the library.

Example:

```
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())

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

rs.resize_callback = function()
  print("Library was resized!")
end

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

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

rs.push()

- Arguments: none.
- Returns: nothing.

A function that scales your game. Also see **rs.pop()**.

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())

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

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

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

rs.pop()

- Arguments: none.
- Returns: nothing.

A function that closes rs.push(). Also see **rs.push()**

Example:

```
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())

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

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

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

rs.get_both_scales()

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

A shortcut function that returns **rs.scale_width** and **rs.scale_height**.

Example:

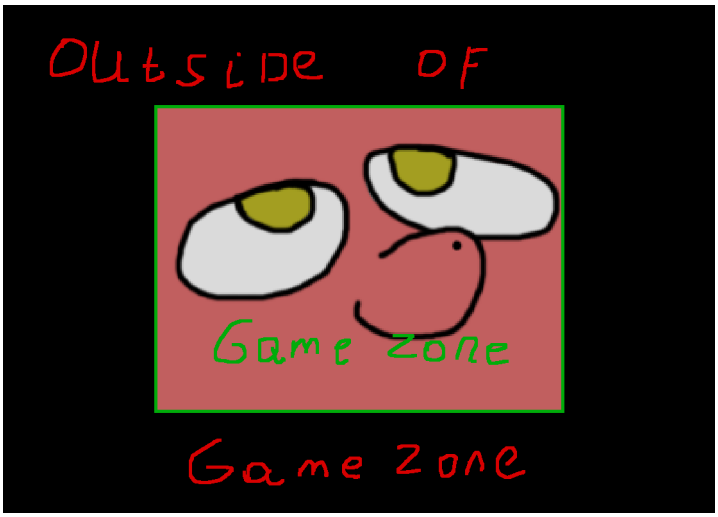
```
-- Basic usage.
local x_scale, y_scale = rs.get_both_scales()
```

rs.is_it_inside()

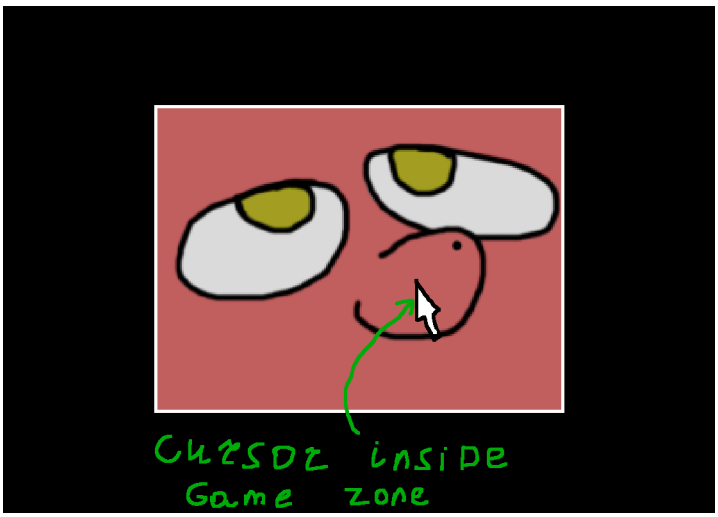
- Arguments: none.
- Returns:
    1. is mouse inside game zone
        - Type: **boolean**.
        - Returns **true** if cursor **inside** game zone.
        - Returns **false** if cursor **outside** of game zone.

A function to determine if given coordinates are inside the game zone. This is useful if you develop games with mouse or touchscreen support that has controls on the screen and outside the game zone.

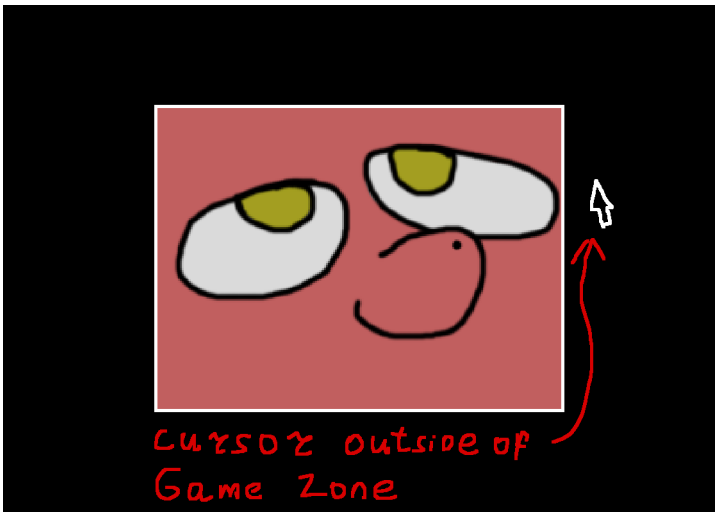
Here quick visualization:



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 is especially useful if you develop games with scrolling worlds or camera movements. Click the black bars should manipulate controls and not objects in the game world. When scale mode == 2, the function will always return true because there won't be any black bars.

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**.

A function to translate coordinates from the window to the game zone. This can be used to translate cursor coordinates so you can check collisions with objects inside the game zone.

If you pass only one argument e.g. **rs.to_game(x)** then the function will return the scaled **x** value.

```
scaled_X = rs.to_game(x)
```

if you pass the 2nd argument e.g. **rs.to_game(nil, y)** then the function will return the scaled **y** value.

```
scaled_y = rs.to_game(nil, y)
```

rs.to_game(x, y) will return both values.

```
scaled_x, scaled_y = rs.to_game(x, y)
```

Example:

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

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

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

  -- Translate it to game.
  mx, my = rs.to_game(mx, my)
  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
        is_touching = true
      else
        is_touching = false
      end
end

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

  if is_touching 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)

  love.graphics.setCanvas()

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

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

Arguments:

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

Returns:

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

A function similar to **rs.to_game** but reversed. It translates coordinates from the game world to the window. For example, when you want to move the cursor to an object inside the game zone.

If you pass only one argument e.g. rs.to_window(x) then the function will return the scaled **x** value.

```
x = rs.to_window(scaled_x)
```

if you pass the 2nd argument e.g. **rs.to_window(nil, y)** then the function will return the scaled **y** value.

```
y = rs.to_window(nil, scaled_y)
```

rs.to_window(x, y) will return both values.

```
x, y = rs.to_window(scaled_x, scaled_y)
```

Example:

```
-- Basic example.
-- Press left button of mouse to teleport cursor to center or rectangle
-- that you can see in center of window.
-- Note: might not work if you use Linux with Wayland.
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())

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

love.mousepressed = function(x, y, button)
   if button == 1 then
      love.mouse.setPosition(
        rs.toScreen(
          (rs.game_width / 2), -- Translate X.
          (rs.game_height / 2) -- Translate y.
          )
        )
   end
end

love.draw = function()
  love.graphics.setCanvas(game_canvas)
  love.graphics.clear(0, 0, 0, 1)
  love.graphics.setColor(1, 1, 1, 1)
  -- Place 100x100 rectangle in center of game zone.
  love.graphics.rectangle("line", (rs.game_width / 2) - 50, (rs.game_height / 2)
- 50, 100, 100)
  love.graphics.rectangle("line", (rs.game_width / 2) - 4, (rs.game_height / 2)
- 4, 4, 4)
  love.graphics.setCanvas()

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

Tips and Tricks

This is a collection of code snippets and examples of useful things that you can do with this library. For more complex examples, go to Resolution Solution repository.

1. How 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.

---

2. When you going to work with Pixel Perfect scaling, you might notice that when game window has non even sizes (like 801x600) you would have pixel bleeding, because due to calculations, rs.x_offset and rs.y_offset would become floats rather then ints. The best course of actions here would be:
   - Ask user to play in fullscreen (because all monitors have even resolutions) for best experience.
   - Don't allow window resizing by user and present to them list of supported window sizes that they can use instead.
   - Just leave it as it.
   - You could try this:

```
force_even_window = function()
--  local window_width, window_height = love.graphics.getWidth(),
love.graphics.getHeight()
--  local is_width_even = (window_width % 2 == 0) -- false == width is not even
--  local is_height_even = (window_height % 2 == 0) -- false == height is not
even

--  if not is_width_even then
--    window_width = window_width + 1
--  end
```

```
--  if not is_height_even then
--    window_height = window_height + 1
--  end

--  if not is_width_even or not is_height_even then
--    love.window.updateMode(window_width, window_height)
--  end

--  rs.resize()
--end
```

This function will round window width and height to nearest biggest even numbers. For example, if width is 801 then function will change it to 802.

The question here will be is when to run this function. Calling it during love.resize() or love.update() is out of window, since only way to change window size in love is love.window.set/updateMode() and this functions will re-create window every time when you call them, which would stop user from resizing window.

Because of that, I strongly don't recommend this solution. Use it only if you REALLY needs game to be scaled as pixel-perfect as possible.

---

3. For pixel-art it is also **love.graphics.setDefaultFilter("nearest", "nearest")** is must-have. For Stretch Scaling and Aspect Scaling **love.graphics.setDefaultFilter("linear", "linear")** suits better, but it's up to you to decide.
4. If you use this library, all calculations should be based around **rs.game_width** and **rs.game_height** for everything that should be subject of scaling, rather then window size.

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

Back to basics.