Play64

Play64 is a fantasy console developed for The Tool Jam 2025. It features a D-pad and two action buttons, with a display capable of rendering four shades of violet. Games—referred to as "carts"—are written in Lua 5.1. The console itself is implemented using BlitzMax NG and the Raylib graphics library.

Controls

- The up button is mapped to the UP and W keys
- The down button is mapped to the DOWN and S keys
- The left button is mapped to the LEFT and A keys
- The right button is mapped to the RIGHT and D keys
- The A button is mapped to the Z and N keys
- The B button is mapped to the X and M keys
- Press ESC to close the application window
- Press R to reset the console
- Press F to toggle fullscreen mode
- Press P to pause the game

How to release your game

To prepare your game for release, edit the `config.ini` file: set `Game` to your game's internal name, `Title` to the window title, and `Intro` to `False`. Remove any unused files from the `carts` directory. You may then rename the `play64.exe` executable as desired.

"config.ini" file

```
[Settings]
; This is the configuration file for the Play64 fantasy console
; It contains settings for the console

Fullscreen=False
; Set to True for fullscreen mode
; Set to False for windowed mode

Width=800
; Width of the window in pixels

Height=600
; Height of the window in pixels

Intro=True
; Set to True to show the intro screen
; Set to False to skip the intro screen

Game=
; The name of the game to load
; Leave empty to load the selection screen
```

```
Title=Play64 fantasy console
; This is the title of the window
; It will be displayed in the title bar
; The title of the window can be changed in the code
```

input

The object module offer an interface to the input system

input.down()

Return a string with the key that is pressed

```
print(input.down())
-- print up or down or left or right or a or b
```

input.pressed()

Return a string with the key that has been pressed

```
print(input.pressed())
-- print up or down or left or right or a or b
```

input.released()

Return a string with the key that has been released

```
print(input.released())
-- print up or down or left or right or a or b
```

draw

The object module offer an interface to the drawing commands

colors palette

4 color palette 0 = darkest, 3 = brightest

draw.background(color:Int)

Change the color of background 0 = darkest, 3 = brightest

```
draw.background(0)
```

draw.color(color:Int)

set the color of the draw command 0 = darkest, 3 = brightest

```
draw.color(3)
```

draw.point(x:Int, y:int)

Draw a pixel

```
--draw a point
draw.point(32,18)
```

draw.line(startx:Int, startY:Int, endX:Int, endY:Int)

Draw a line

```
--draw a line
draw.line(0,0,18,36)
```

draw.rect(x:Int, y:Int, Width:Int, height:Int)

Draw a rectangle

```
--draw a rectangle
draw.rect(0,0,20,20)
```

draw.text(text:String, x:Int, y:Int, size:Int)

Print a string of text Minimum size = 10

```
draw.text("a string of text",0,0,15)
```

draw.circle(x:Int, y:Int, Radius:Int)

Draw a circle

```

```

```
--draw a circle
draw.rect(30,30,10)
```

draw.triangle(x1:Int, y1:Int, x2:Int, y2:Int, x3:Int, y3:Int)

Draw a triangle

```
--draw a triangle
draw.triangle(1,1,63,1,32,63)
```

draw.sprite(title:String, x:Int, y:Int)

```
draw.sprite("enemy", 10,10)
-- same as
sprites.draw("enemy",10,10)
```

app

The object module offer an interface for manage all the app settings

app.dt()

Return the delta time in seconds The default game target fps is 30 frame per second.

app.title(title:String)

set the title of the window

app.width(width:Int)

set the width of the window

app.height(heigth:Int)

set the height of the window

app.fullscreen()

toggle the fullscreen mode

app.reset()

reset the cart and go back to selection screen

app.restart()

restart the current cart

app.loadCart(cart:String)

load a specific cart

```
app.loadCart("snake")
```

app.save(key:String, value:String)

save a value of the current game

```
app.save("highscore", "1000")
```

app.load(key:String, value:String)

load a value of the current game

```
app.load("highscore")
```

Sound

The object module offer an interface to sound system

sound.play(sound:String)

play a built-in sound by name

```
--play bump sound effect
sound.play("bump")
```

Possible values: bottle_break, bump, cancel, cat_meow, click, confirm, crunch, digital_alarm, dog_bark, door_slow_open, drink, evil_laugh, explosion, gun, hurt, jump, laser_gun, low_health, menu_in, menu_out, monster_scream, notso_confirm, pause, phone_ring, powerdown, powerup, siren, steps, sword_slash, thunder, trampoline, water_splash,

Camera

The object module offer an interface to the camera system

camera.target(x:Int, y:Int)

```

```

```
--set position of the camera at 10, 10
camera.target(10,10)
```

camera.offset(x:Int, y:Int)

```
--set offset of the camera at 10, 10
camera.offset(10,10)
```

camera.rotation(deg:Float)

```
--set rotation of the camera at 45
camera.target(45.0)
```

camera.zoom(zoom:Float)

```
--set zoom of the camera at 1.2
camera.target(1.2)
```

camera.reset()

```
--reset the camera
camera.reset()
```

Sprites

sprites.add(title:String, data:String)

Return title:String

```
    local enemySprite = [[
        0 0 0 0 0 0 0 0 0 0
        0 1 1 1 1 1 1 1 1 0
        0 1 0 0 0 0 0 0 1 0
        0 1 0 1 1 1 1 0 1 0
        0 1 0 1 0 0 1 0 1 0
        0 1 0 1 1 1 1 0 1 0
        ]]
    sprite = sprites.add("enemy", enemySprite) -- sprite == "enemy"
```

sprites.draw(title:String, x:Int, y:Int)

```
sprites.draw("enemy",10,10)
-- same as
draw.sprite("enemy", 10,10)
```

sprites.collision(sprite1:String, x1:Int, y1:Int, sprite2:String, x2:Int, y2:Int)

Return 1 in case of collision

```
if sprites.collision("player",player.x,player.y,"enemy",enemy.x,enemy.Y) == 1 then
  --Collision! Do something
end
```

Timer

The object module offer an interface for manage timers

timer.delay(fn, delay)

Calls the function `fn` after the given `delay` time has passed. Returns the associated event.

```
-- Prints "Hello world!" after 2 seconds
timer.delay(function() print("Hello world!") end, 2)
```

timer.recur(fn, delay)

Calls the function `fn` at an interval of `delay`. Returns the associated event.

```
-- Prints  timer!" every half-second
 timer.recur(function() print( timer!") end, .5)
```

Chaining events

To avoid having to deeply nest several functions when creating chains of events, the `:after()` method can be called on an event returned by `timer.delay()`. You can keep using the `:after()` method to create complex timed sequences.

```
-- Prints "cat", "dog", then "owl", waiting 1 second between each print
timer.delay(function() print("cat") end, 1)
  :after(function() print("dog") end, 1)
  :after(function() print("owl") end, 1)
```

Stopping events

An event can be stopped and removed at any point by calling its `:stop()` method. To do this the event must be assigned to a variable when it is created.

```
-- Create a new event
local t = timer.delay(function() print( timer!") end, 10)
-- Remove the event before it has a chance to run
t:stop()
```

Groups timer provides the ability to create event groups; these are objects which can

have events added to them, and which are in charge of updating and handling their contained events. A group is created by calling the timer.group()` function.

```
local group = timer.group()
```

Once a group is created it acts independently of the timer` object, and must be updated each frame using its own update method.

```
group:update()
```

To add a events to a group, the group's `:delay()` or `:recur()` methods should be used.

```
group:delay(function() print("hello world") end, 4)
```

A good example of where groups are useful is for games where you may have a set of events which effect objects in the game world and which you want to pause when the game is paused. A group's events can be paused by simply neglecting to call its `update()` method; when a group is destroyed its events are also destroyed.

object

The object module offer an interface for OOP programming

Creating a new class

```
Point = object:extend()

function Point:new(x, y)
  self.x = x or 0
```

```
  self.y = y or 0
end
```

Creating a new object

```
local p = Point(10, 20)
```

Extending an existing class

```
Rect = Point:extend()

function Rect:new(x, y, width, height)
  Rect.super.new(self, x, y)
  self.width = width or 0
  self.height = height or 0
end
```

Checking an object's type

```
local p = Point(10, 20)
print(p:is(Object)) -- true
print(p:is(Point)) -- true
print(p:is(Rect)) -- false
```

Using mixins

```
PairPrinter = object:extend()

function PairPrinter:printPairs()
  for k, v in pairs(self) do
    print(k, v)
  end
end


Point = object:extend()
Point:implement(PairPrinter)

function Point:new(x, y)
  self.x = x or 0
  self.y = y or 0
end
```

```
local p = Point()
p:printPairs()
```

Using static variables

```
Point = object:extend()
Point.scale = 2

function Point:new(x, y)
  self.x = x or 0
  self.y = y or 0
end

function Point:getScaled()
  return self.x * Point.scale, self.y * Point.scale
end
```

Creating a metamethod

```
function Point:__tostring()
  return self.x .. ", " .. self.y
end
```

Tween

The object module offer an interface for tweening

Tween creation

```
local t = tween.new(duration, subject, target, [easing])
```

Creates a new tween.

- `duration` means how much the change will take until it's finished. It must be a positive number.
- `subject` must be a table with at least one key-value. Its values will be gradually changed by the tween until they look like `target`. All the values must be numbers, or tables with numbers.
- `target` must be a table with at least the same keys as `subject`. Other keys will be ignored.
- `easing` can be either a function or a function name (see the easing section below). It's default value is `'linear'`
- `t` is the object that must be used to perform the changes - see the "Tween methods" section below.

This function only creates and returns the tween. It must be captured in a variable and updated via `t:update(dt)` in order for the changes to take place.

Tween methods

```
local complete = t:update()
```

Gradually changes the contents of `subject` to that it looks more like `target` as time passes.

- `t` is a tween returned by `tween.new`
- `complete` is `true` if the tween has reached its limit (its *internal clock* is `>= duration`). It is false otherwise.

When the tween is complete, the values in `subject` will be equal to `target`'s. The way they change over time will depend on the chosen easing function.

```
local complete = t:set(clock)
```

Moves a tween's internal clock to a particular moment.

- `t` is a tween returned by `tween.new`
- `clock` is a positive number or 0. It's the new value of the tween's internal clock.
- `complete` works like in `t:update`; it's `true` if the tween has reached its end, and `false` otherwise.

If clock is greater than `t.duration`, then the values in `t.subject` will be equal to `t.target`, and `t.clock` will be equal to `t.duration`.

```
t:reset()
```

Resets the internal clock of the tween back to 0, resetting `subject`.

- `t` is a tween returned by `tween.new`

This method is equivalent to `t:set(0)`.

Easing functions

Easing functions are functions that express how slow/fast the interpolation happens in tween.

The easing functions can be found in the table `tween.easing`.

They can be divided into several families:

- `linear` is the default interpolation. It's the simplest easing function.
- `quad`, `cubic`, `quart`, `quint`, `expo`, `sine` and `circle` are all "smooth" curves that will make transitions look natural.
- The `back` family starts by moving the interpolation slightly "backwards" before moving it forward.
- The `bounce` family simulates the motion of an object bouncing.
- The `elastic` family simulates inertia in the easing, like an elastic gum.

Each family (except `linear`) has 4 variants:

- `in` starts slow, and accelerates at the end
- `out` starts fast, and decelerates at the end
- `inOut` starts and ends slow, but it's fast in the middle
- `outIn` starts and ends fast, but it's slow in the middle

family	in	out	inOut	outIn
Linear	linear	linear	linear	linear
Quad	inQuad	outQuad	inOutQuad	outInQuad
Cubic	inCubic	outCubic	inOutCubic	outInCubic
Quart	inQuart	outQuart	inOutQuart	outInQuart
Quint	inQuint	outQuint	inOutQuint	outInQuint
Expo	inExpo	outExpo	inOutExpo	outInExpo
Sine	inSine	outSine	inOutSine	outInSine
Circ	inCirc	outCirc	inOutCirc	outInCirc
Back	inBack	outBack	inOutBack	outInBack
Bounce	inBounce	outBounce	inOutBounce	outInBounce
Elastic	inElastic	outElastic	inOutElastic	outInElastic

When you specify an easing function, you can either give the function name as a string. The following two are equivalent:

```
local t1 = tween.new(10, subject, {x=10}, tween.easing.linear)
local t2 = tween.new(10, subject, {x=10}, 'linear')
```

But since `'linear'` is the default, you can also do this:

```
local t3 = tween.new(10, subject, {x=10})
```

Custom easing functions

You are not limited to tween's easing functions; if you pass a function parameter in the easing, it will be used.

The passed function will need to take 4 parameters:

- `t` (time): starts in 0 and usually moves towards duration
- `b` (begin): initial value of the of the property being eased.
- `c` (change): ending value of the property - starting value of the property
- `d` (duration): total duration of the tween

And must return the new value after the interpolation occurs.

```
local cubicbezier = function (x1, y1, x2, y2)
  local curve = love.math.newBezierCurve(0, 0, x1, y1, x2, y2, 1, 1)
  return function (t, b, c, d) return c * curve:evaluate(t/d) + b end
end

local label = { x=200, y=0, text = "hello" }
local labelTween = tween.new(4, label, {y=300}, cubicbezier(.35, .97, .58, .61))
```

Credits

- Engine written in BlitzMax NG
- Carts written in Lua 5.1
- The graphic library is Raylib
- Classic by rxi for lua OOP
- Tween by kikito
- Tick by rxi
- Sound Effects by Coffee 'Valen' Bat