master nico / API.md

Go to file ...

ftsf Add TOC to API.md and add more functions ✕ Latest commit 153a364 4 days ago History

1 contributor

676 lines (453 sloc) | 13 KB

Raw Blame

Nico API

Overview

Many functions in Nico take `Pfloat` or `Pint` arguments, these are automatically converted types so you don't have to think too much about types when you don't care.

System

`init(org: string, app: string)` Initialises Nico, must be called before any other Nico operation. `org`: organisation name, used for storing preference data `app`: application name, used for storing preference data

---

`shutdown()` Shuts down Nico and ends the application.

`createWindow(title: string, width: int, height: int, scale: int, fullscreen: bool)` Creates a window with a title of `title` and a canvas of size `width` x `height` and draws it scaled up by `scale` times.

`run(initFunc: proc(), updateFunc: proc(dt: float32), drawFunc: proc())` Runs Nico, first runs `initFunc`. Every frame it calls `updateFunc` passing `dt` as the time since the last call, and then `drawFunc`. Continues to run until `shutdown` is called

---

Input

```
type NicoButton = enum
  pcLeft
  pcRight
  pcUp
  pcDown
  pcA
  pcB
  pcX
  pcY
```

```
    pcL1
    pcL2
    pcL3
    pcR1
    pcR2
    pcR3
    pcStart
    pcBack
```

Common button inputs compatible with most gamepads or keyboard

Buttons

`btn(b: NicoButton): bool` Returns true while the button `b` is held down by any player

---

`btnp(b: NicoButton): bool` Returns true as the button `b` is pressed by any player

`btnup(b: NicoButton): bool` Returns true as the button `b` is released by any player

`btnpr(b: NicoButton, repeat: int = 48): bool` Returns true as the button `b` is pressed and again every `repeat` frames while held down by any player.

Also available are versions which take a player id `btn(b: NicoButton, player: int): bool` Returns true while the button `b` is held down by `player`

`btnp(b: NicoButton, player: int): bool` Returns true as the button `b` is pressed by `player`

`btnup(b: NicoButton, player: int): bool` Returns true as the button `b` is released by `player`

`btnpr(b: NicoButton, player: int, repeat: int = 48): bool` Returns true as the button `b` is pressed and again every `repeat` frames while held down by `player`.

Joysticks

```
type NicoAxis = enum
  pcXAxis
  pcYAxis
  pcXAxis2
  pcYAxis2
  pcLTrigger
  pcRTrigger
```

`jaxis(axis: NicoAxis, player: int): float32` Returns the value of a joystick axis on `player`'s controller

Mouse

`mouse(): (int,int)` returns the current mouse position in canvas units `0,0` being the top left of the window

---

`mouserel(): (float32,float32)` returns the change in mouse position in canvas units but with subpixel precision

---

`mousebtn(b: range[0..2]): bool` returns while the mouse button `b` is down. `0 = left` `1 = middle` `2 = right`

`mousebtnp(b: range[0..2]): bool` returns as the mouse button `b` is pressed.

`mousebtnpr(b: range[0..2], repeat: int = 48): bool` returns as the mouse button `b` is pressed and again every `repeat` frames.

Keyboard

`key(keycode: Keycode): bool` Returns true when key with `keycode` is down

`keyp(keycode: Keycode): bool` Returns true as key with `keycode` is pressed

`keypr(keycode: Keycode, repeat: int = 48): bool` Returns true as key with `keycode` is pressed and again every `repeat` frames

---

Graphics

Colors

`setColor(color: int)` Sets the current drawing color to the palette index `color`

`getColor(): int` Gets the current drawing color

Drawing

`cls()` Sets all pixels to 0, clear the screen.

Pixels

`pset(x,y: int)` Sets pixel to current color, no effect if out of bounds

`pget(x,y: int): int` Gets the pixel color at `x,y`, returns `0` if out of bounds

---

Circles and Ellipses

`circ(cx,cy,r: int)` Draws a circle centered at `cx,cy` with radius `r`

`circfill(cx,cy,r: int)` Draws a filled circle centered at `cx,cy` with radius `r`

`ellipsefill(cx,cy,rx,ry: int)` Draws a filled ellipse centered at `cx,cy` with radius `rx,ry`

Lines

`line(x0,y0,x1,y1: int)` Draws a line between `x0,y0` and `x1,y1`

`hline(x0,y,x1: int)` Draws a horizontal line between `x0` and `x1` on `y`

`vline(x,y0,y1: int)` Draws a vertical line between `y0` and `y1` on `x`

Rectangles

`rect(x0,y0,x1,y1: int)` Draws a rectangle from `x0,y0` to `x1,y1`

`rectfill(x0,y0,x1,y1: int)` Draws a filled rectangle from `x0,y0` to `x1,y1`

`rrect(x0,y0,x1,y1: int, r: int = 1)` Draws a rounded rectangle from `x0,y0` to `x1,y1` with corner radius `r`

`rrectfill(x0,y0,x1,y1: int, r: int = 1)` Draws a filled rounded rectangle from `x0,y0` to `x1,y1` with corner radius `r`

`box(x,y,w,h: int)` Draws a rectangle with top left corner `x,y` of width and height `w,h`

`boxfill(x,y,w,h: int)` Draws a filled rectangle with top left corner `x,y` of width and height `w,h`

`boxfill(x,y,w,h: int)` Draws a filled rectangle with top left corner `x,y` of width and height `w,h`

`rectCorner(x0,y0,x1,y1: int)` Draws only the corners of a rectangle

`rrectCorner(x0,y0,x1,y1: int, r: int = 1)` Draws only the corners of a rounded rectangle

Triangles

`trifill(ax,ay,bx,by,cx,cy: int)` Draws a filled triangle between points `(ax,ay),(bx,by),(cx,cy)`

Quads

`quadfill(ax,ay,bx,by,cx,cy,dx,dy: int)` Draws a filled quad between points `(ax,ay),(bx,by),(cx,cy),(dx,dy)`

Sprites

`loadSpritesheet(index: int, filename: string, sw, sh: int = 8)` Loads the file at `filename` (must be a PNG file) into spritesheet slot `index`. Each sprite will be of size `sw,sh`

`setSpritesheet(index: int)` Sets the current spritesheet to `index`

`spr(spr: int, x,y: int)` Draws sprites `spr` from the current spritesheet at `x,y`.

`spr(spr: int, x,y: int, w,h: int = 1, hflip, vflip: bool = false)` Draws `w,h` sprites starting from `spr` from the current spritesheet at `x,y`, optionally flipped.

`sprs(spr: int, x,y: int, w,h: int = 1, dw,dh: int = 1, hflip, vflip: bool = false)` Draws `w,h` tiles starting from `spr` from the current spritesheet at `x,y`, optionally flipped and scaled to `dw,dh` tiles.

Text

`loadFont(index: int, filename: string)` Loads font at `filename` into font index `index`. `filename` must be a PNG file with a specific format see example in `examples/assets/font.png`. `filename.dat` should also exist and contain a list of characters included in the font, see example in `examples/assets/font.png.dat`.

`setFont(index: int)` sets the current font to the font loaded into index `index`

`glyph(c: Rune, x,y: int)` Draws a unicode character `c` at `x,y`

`print(text: string, x,y: int)` Draws `text` at `x,y` in current color

`printc(text: string, x,y: int)` Draws `text` centered at `x,y` in current color

`printr(text: string, x,y: int)` Draws `text` right aligned at `x,y` in current color

`glyphWidth(c: Rune): int` returns the width of a unicode character `c`

`textWidth(text: string): int` returns the width of `text`

Tilemap

`newMap(index: int, w,h: int, tw,th: int = 8)` create a new tilemap in index `index` with size `w,h` and each tile of size `tw,th`

`loadMap(index: int, filename: string)` loads tilemap at `filename` into index `index` `filename` should be in Tiled's json format.

`saveMap(index: int, filename: string)` saves the tilemap in slot `index` to `filename` in Tiled's json format.

`setMap(index: int)` use the map at index `index` for future map calls

`mapWidth(): int` returns the current map's width in tiles

`mapHeight(): int` returns the current map's height in tiles

`mapDraw(tx,ty,tw,th: int, dx,dy: int, dw,dh: int = -1, loop: bool = false, ox,oy: int = 0)` draws current tilemap to the canvas at `dx,dy` starting from tile `tx,ty` and drawing `tw,th` tiles. `dw,dh` can be used for scaling the tilemap. `loop` will repeat the tilemap `ox,oy` specifies a pixel offset for tiles

---

Palettes

`loadPaletteFromGPL(filename: string): Palette` Returns a loaded palette from the given filename in Gimp Palette Format.

`loadPaletteCGA(): Palette` Returns a 4 color CGA Palette (Black, Cyan, Magenta, White)

`loadPalettePico8(): Palette` Returns the 16 color "Pico-8" palette

`loadPalettePico8Extra(): Palette` Returns the 16 color "Pico-8" palette + the 16 color secret "Pico-8" palette

`loadPaletteGrayscale(): Palette` Returns at 256 level grayscale palette

`setPalette(palette: Palette)` Sets the current palette

`pal(a,b: int)` Maps color `a` to color `b` for subsequent drawing operations

`pal()` Resets palette mapping

`palt(color: int, transparent: bool)` Makes `color` transparent or not for subsequent sprite drawing operations By default color 0 is transparent.

`palt()` Resets transparent colors such that only color 0 is transparent.

Dithering

`ditherPattern(pattern: uint16 = 0b1111_1111_1111_1111)` Sets the current dither pattern for subsequent draw calls, default pattern is no dithering. Each bit specifies a pixel in the 4x4 dithering pattern.

```
0 1 2 3
4 5 6 7
8 9 A B
C D E F
```

Camera

`setCamera(x,y: int = 0)` Sets the camera offset for drawing

`getCamera(): (int,int)` Gets the current camera offset

`clip(x,y,w,h: int)` Sets the clipping area, only pixels within the clipping area will be written do

---

`clip()` Resets the clipping area to the full canvas

`getClip(): (int,int,int,int)` Gets the current clipping area

Misc Graphics

`copy(sx,sy,dx,dy,w,h: int)` Copy a region of the canvas from source `sx,sy` to dest `dx,dy` of size `w,h`

Audio

There are 16 audio channels each channel can either be silent, play a sound sample (sfx), play a generated tone (synth), or play streaming music (music) All audio commands other than loading and volume commands take the channel ID as the first argument

`masterVol(0..255)` Sets the master volume level.

`masterVol(): int` Returns the master volume level.

SFX

`loadSfx(index: 0..63, filename: string)` Load audio file into sfx slot `index`. The entire file will be decoded and loaded into RAM.

`sfx(channel: 0..15, index: 0..63)` Play sfx in `index` on `channel`.

`sfxVol(newVol: 0..255)` Sets the volume for all sfx and synths.

`sfxVol(): int` Gets the volume for all sfx and synths.

Synth

`synth(channel: 0..15, shape: SynthShape, freq: Pfloat, init: 0..15, env: -7..7, length: 0..255)` Plays a synthesised tone on `channel` at pitch of `freq`. `init` is the initial volume of the sound. `env` is the change in volume over time. Positive numbers decay over time, Negative numbers increase in amplitude over time. `length` is the clocks before the note is cut off.

```
type SynthShape = enum
  synSame # no change
  synSin # Sine wave
  synSqr # Square wave
  synP12 # 12.5% Pulse wave
  synP25 # 25.0% Pulse wave
  synSaw # Sawtooth wave
  synTri # Triangle wave
  synNoise # Noise
  synNoise2 # Metallic Noise
  synWav # Use custom waveform
```

Music

`loadMusic(index: 0..63, filename: string)` Load audio file into music slot `index`. The file will be decoded and streamed on demand.

Math

`wrap(x,t: int): int` Wraps an integer `x` by `t` similar to `mod` but more practically handling negative `x`.

```
wrap(0,4) == 0
wrap(1,4) == 1
wrap(2,4) == 2
wrap(3,4) == 3
```

```
wrap(4,4) == 0
wrap(-1,4) == 3
wrap(-2,4) == 2
wrap(-3,4) == 1
wrap(-4,4) == 0
```

`clamp[T](x: T): T` or `clamp01[T](x: T): T` clamps a number to between 0 and 1

--

`mid[T](a,b,c: T): T` returns the middle of 3 numbers.

eg.

```
mid(1,2,3) == 2
mid(3,2,1) == 2
mid(2,3,1) == 2
```

`flr(x: Pfloat): Pfloat` returns `x` rounded down

`ceil(x: Pfloat): Pfloat returns` x` rounded up

`lerp[T](a,b: T, t: Pfloat): T` linearly interpolates between `a` and `b` where `t == 0` will return `a` and `t == 1` will return `b`.

eg.

```
lerp(50.0f, 100.0f, 0.5f) == 75.0f
lerp(50.0f, 100.0f, 0.0f) == 50.0f
lerp(50.0f, 100.0f, 0.0f) == 100.0f
```

`invLerp(a,b,v: Pfloat): Pfloat` returns where `v` is in the range `a..b`.

```
invLerp(50f,100f,75f) == 0.5f
```

`rnd[T: Natural](x: T): T` returns a random integer in range `0..<x`. Will never return `x` but will return `0`.

`rnd(a: openArray[T]): T` returns a random item from input

`rnd(x: Pfloat): Pfloat` returns a random float between `0..<x`

`shuffle[T](a: var seq[T])` shuffles `a` inplace.

`sgn(x: Pint): Pint` returns the sign of `x`.

```
sgn(-10) == -1
sgn(100) == 1
sgn(0) == 0
```

`deg2rad(x: Pfloat): Pfloat` converts degrees to radians

`rad2deg(x: Pfloat): Pfloat` converts radians to degrees

`angleDiff(a,b: Pfloat): Pfloat` returns the difference between two angles in radians

```
angleDiff(deg2rad(-10), deg2rad(10)) == rad2deg(-20)
angleDiff(deg2rad(-180), deg2rad(180)) == rad2deg(0)
```