```
--------------
File: gui.nim
--------------

import
  tables,
  nimpy,
  nimraylib_now/raylib,
  nimraylib_now/raygui as rg

pyExportModule("pyMeow")

type
  DropDownBox = object
    rec: Rectangle
    text: string
    active: cint
    editMode: bool

  TextBox = object
    rec: Rectangle
    text: string
    editMode: bool

  ColorPicker = object
    rec: Rectangle
    color: Color

  Spinner = object
    rec: Rectangle
    value: cint
    editMode: bool

var
  dropDownTable: Table[int, DropDownBox]
  textBoxTable: Table[int, TextBox]
  colorPickerTable: Table[int, ColorPicker]
  spinnerTable: Table[int, Spinner]

converter toCint(x: float|int): cint = x.cint

template getRec: Rectangle =
  Rectangle(
    x: posX,
    y: posY,
    width: width,
    height: height
  )

proc fade(alpha: float) {.exportpy: "gui_fade".} =
  rg.fade(alpha)

proc windowBox(posX, posY, width, height: float, title: string): bool {.exportpy: "gui_window_box".} =
  rg.windowBox(getRec, title)
```

```
proc groupBox(posX, posY, width, height: float, text: string) {.exportpy: "gui_group_box".} =
  rg.groupBox(getRec, text)

proc line(posX, posY, width, height: float, text: string) {.exportpy: "gui_line".} =
  rg.line(getRec, text)

proc panel(posX, posY, width, height: float) {.exportpy: "gui_panel".} =
  rg.panel(getRec)

proc label(posX, posY, width, height: float, text: string) {.exportpy: "gui_label".} =
  rg.label(getRec, text)

proc button(posX, posY, width, height: float, text: string): bool {.exportpy: "gui_button"} =
  rg.button(getRec, text)

proc labelButton(posX, posY, width, height: float, text: string): bool {.exportpy: "gui_label_button".} =
  rg.labelButton(getRec, text)

proc checkBox(posX, posY, width, height: float, text: string, checked: bool): bool {.exportpy: "gui_check_box".} =
  rg.checkBox(getRec, text, checked)

proc comboBox(posX, posY, width, height: float, text: string, active: int = 0): int {.exportpy: "gui_combo_box".} =
  rg.comboBox(getRec, text, active)

proc dropdownBox(posX, posY, width, height: float, text: string, id: int, active: int = 0): int {.exportpy: "gui_dropdown_box".} =
  if id notin dropDownTable:
    dropDownTable[id] = DropDownBox(
      rec: getRec,
      text: text,
      active: active,
    )
  if rg.dropdownBox(getRec, dropDownTable[id].text.cstring, dropDownTable[id].active.addr, dropDownTable[id].editMode):
    dropDownTable[id].editMode = not dropDownTable[id].editMode
    if dropDownTable[id].editMode:
      rg.lock()
    else:
      rg.unlock()
  dropDownTable[id].active

proc textBox(posX, posY, width, height: float, text: string, id: int): string {.exportpy: "gui_text_box".} =
  if id notin textBoxTable:
    textBoxTable[id] = TextBox(
      rec: getRec,
      text: text & newString(width.int),
    )
  if rg.textBox(getRec, textBoxTable[id].text.cstring, 250, textBoxTable[id].editMode):
    textBoxTable[id].editMode = not textBoxTable[id].editMode
  $textBoxTable[id].text.cstring

proc progressBar(posX, posY, width, height: float, textLeft, textRight: string, value, minValue, maxValue: float): float {.exportpy: "gui_progress_bar".} =
  rg.progressBar(getRec, textLeft, textRight, value, minValue, maxValue)

proc statusBar(posX, posY, width, height: float, text: string) {.exportpy: "gui_status_bar".} =
  rg.statusBar(getRec, text)

proc messageBox(posX, posY, width, height: float, title, message, buttons: string): int {.exportpy: "gui_message_box".} =
  rg.messageBox(getRec, title, message, buttons)

proc colorPicker(posX, posY, width, height: float, id: int): Color {.exportpy: "gui_color_picker".} =
  if id notin colorPickerTable:
    colorPickerTable[id] = ColorPicker(
      rec: getRec,
      color: Raywhite
    )
  colorPickerTable[id].color = rg.colorPicker(getRec, colorPickerTable[id].color)
  colorPickerTable[id].color

proc colorBarAlpha(posX, posY, width, height: float, alpha: float): float {.exportpy: "gui_color_bar_alpha".} =
  rg.colorBarAlpha(getRec, alpha)

proc colorBarHue(posX, posY, width, height: float, value: float): float {.exportpy: "gui_color_bar_hue".} =
  rg.colorBarHue(getRec, value)

proc scrollBar(posX, posY, width, height: float, value, minValue, maxValue: int): int {.exportpy: "gui_scroll_bar"} =
  rg.scrollBar(getRec, value, minValue, maxValue)

proc spinner(posX, posY, width, height: float, text: string, value, minValue, maxValue, id: int): int {.exportpy: "gui_spinner".} =
  if id notin spinnerTable:
    spinnerTable[id] = Spinner(
      rec: getRec,
      value: value
    )
  spinnerTable[id].editMode = rg.spinner(getRec, text, spinnerTable[id].value.addr, minValue, maxValue, spinnerTable[id].editMode)
  spinnerTable[id].value

proc slider(posX, posY, width, height: float, textLeft, textRight: string, value, minValue, maxValue: float): float {.exportpy: "gui_slider".} =
  rg.slider(getRec, textLeft, textRight, value, minValue, maxValue)

proc sliderBar(posX, posY, width, height: float, textLeft, textRight: string, value, minValue, maxValue: float): float {.exportpy: "gui_slider_bar".} =
  rg.sliderBar(getRec, textLeft, textRight, value, minValue, maxValue)

proc loadStyle(fileName: string) {.exportpy: "gui_load_style".} =
  rg.loadStyle(fileName)

--------------
File: input.nim
--------------
```

```
import
  os, nimpy, nimraylib_now

pyExportModule("pyMeow")

when defined(linux):
  import x11/[x, xlib, xtst]

  var
    display = XOpenDisplay(nil)
    root = XRootWindow(display, 0)

  proc keyPressed*(key: int): bool {.exportpy: "key_pressed".} =
    var keys: array[0..31, char]
    discard XQueryKeymap(display, keys)
    let keycode = XKeysymToKeycode(display, key.culong)
    (ord(keys[keycode.int div 8]) and (1 shl (keycode.int mod 8))) != 0

  proc pressKey*(key: int, hold: bool = false) {.exportpy: "press_key".} =
    let keycode = XKeysymToKeycode(display, key.KeySym)
    discard XTestFakeKeyEvent(display, keycode.cuint, 1, CurrentTime)
    if not hold:
      discard XTestFakeKeyEvent(display, keycode.cuint, 0, CurrentTime)

  proc mouseMove*(x, y: cint, relative: bool = false) {.exportpy: "mouse_move".} =
    if relative:
      discard XTestFakeRelativeMotionEvent(display, x, y, CurrentTime)
    else:
      discard XTestFakeMotionEvent(display, -1, x, y, CurrentTime)
    discard XFlush(display)

  proc mouseClick* {.exportpy: "mouse_click"} =
    discard XTestFakeButtonEvent(display, 1, 1, 0)
    discard XFlush(display)
    sleep(2)
    discard XTestFakeButtonEvent(display, 1, 0, 0)
    discard XFlush(display)

  proc mousePosition*: Vector2 {.exportpy: "mouse_position".} =
    var
      qRoot, qChild: Window
      qRootX, qRootY, qChildX, qChildY: cint
      qMask: cuint

    discard XQueryPointer(display, root, qRoot.addr, qChild.addr, qRootX.addr, qRootY.addr, qChildX.addr
, qChildY.addr, qMask.addr)
    result.x = qRootX.cfloat
    result.y = qRootY.cfloat

elif defined(windows):
  import winim

  proc keyPressed*(vKey: int32): bool {.exportpy: "key_pressed".} =
    GetAsyncKeyState(vKey).bool
  proc pressKey(vKey: int) {.exportpy: "press_key".} =
    var input: INPUT
    input.‘type‘ = INPUT_KEYBOARD
    input.ki.wVk = vKey.uint16
    SendInput(1, input.addr, sizeof(input).int32)

  proc mouseMove(x, y: int32) {.exportpy: "mouse_move".} =
    var input: INPUT
    input.mi = MOUSE_INPUT(
      dwFlags: MOUSEEVENTF_MOVE,
      dx: x.int32,
      dy: y.int32,
    )
    SendInput(1, input.addr, sizeof(input).int32)

  proc mouseClick {.exportpy: "mouse_click".} =
    var
      down: INPUT
      release: INPUT
    down.mi = MOUSE_INPUT(dwFlags: MOUSEEVENTF_LEFTDOWN)
    release.mi = MOUSE_INPUT(dwFlags: MOUSEEVENTF_LEFTUP)
    SendInput(1, down.addr, sizeof(down).int32)
    sleep(3)
    SendInput(1, release.addr, sizeof(release).int32)

  proc mousePosition*: Vector2 {.exportpy: "mouse_position".} =
    var point: POINT
    discard GetCursorPos(point.addr)
    result.x = point.x.cfloat
    result.y = point.y.cfloat

--------------
File: memcore.nim
--------------

import
  os, strformat, sequtils,
  strutils, nimpy

pyExportModule("pyMeow")

when defined(windows):
  import winim
elif defined(linux):
  import
    posix, strscans, tables,
    ptrace

  proc process_vm_readv(
    pid: int,
    localIov: ptr IOVec,
    liovcnt: culong,
    remoteIov: ptr IOVec,
    riovcnt: culong,
    flags: culong
  ): cint {.importc, header: "<sys/uio.h>", discardable.}

  proc process_vm_writev(
      pid: int,
      localIov: ptr IOVec,
      liovcnt: culong,
      remoteIov: ptr IOVec,
      riovcnt: culong,
      flags: culong
  ): cint {.importc, header: "<sys/uio.h>", discardable.}

type
  Process* = object
    name: string
    pid: int
    debug: bool
    when defined(windows):
      handle: HANDLE

  Module = object
    name: string
    base: ByteAddress
    ‘end‘: ByteAddress
    size: int

  Page = object
    start: ByteAddress
    ‘end‘: ByteAddress
    size: int

proc checkRoot =
  when defined(linux):
    if getuid() != 0:
      raise newException(IOError, "Root access required!")

proc getErrorStr: string =
  when defined(linux):
    let
      errCode = errno
      errMsg = strerror(errCode)
  elif defined(windows):
    var
      errCode = osLastError()
      errMsg = osErrorMsg(errCode)
    stripLineEnd(errMsg)
  result = fmt"[Error: {errCode} - {errMsg}]"

proc memoryErr(m: string, address: ByteAddress) {.inline.} =
  raise newException(
    AccessViolationDefect,
    fmt"{m} failed [Address: 0x{address.toHex()}] {getErrorStr()}"
  )

proc is64bit(process: Process): bool {.exportpy: "is_64_bit".} =
  when defined(linux):
    var buffer = newSeq[byte](5)
    let exe = open(fmt"/proc/{process.pid}/exe", fmRead)
    discard exe.readBytes(buffer, 0, 5)
    result = buffer[4] == 2
    close(exe)
  elif defined(windows):
    var wow64: BOOL
    discard IsWow64Process(process.handle, wow64.addr)
    result = wow64 != TRUE

iterator enumProcesses: Process {.exportpy: "enum_processes".} =
  var p: Process
  when defined(linux):
    checkRoot()
    let allFiles = toSeq(walkDir("/proc", relative = true))
    for pid in mapIt(filterIt(allFiles, isDigit(it.path[0])), parseInt(it.path)):
        p.pid = pid
        p.name = readFile(fmt"/proc/{pid}/comm").strip()
        yield p
  elif defined(windows):
    var
      pe: PROCESSENTRY32
      hResult: WINBOOL
    let hSnapShot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0)
    defer: CloseHandle(hSnapShot)
    pe.dwSize = sizeof(PROCESSENTRY32).DWORD
    hResult = Process32First(hSnapShot, pe.addr)
    while hResult:
      p.name = nullTerminated($$pe.szExeFile)
      p.pid = pe.th32ProcessID
      yield p
      hResult = Process32Next(hSnapShot, pe.addr)

proc pidExists(pid: int): bool {.exportpy: "pid_exists".} =
  pid in mapIt(toSeq(enumProcesses()), it.pid)

proc processExists(processName: string): bool {.exportpy: "process_exists".} =
  processName in mapIt(toSeq(enumProcesses()), it.name)

proc processRunning(process: Process): bool {.exportpy: "process_running".} =
  when defined(linux):
    return kill(process.pid.cint, 0) == 0
  elif defined(windows):
    var exitCode: DWORD
    GetExitCodeProcess(process.handle, exitCode.addr)
    return exitCode == STILL_ACTIVE

proc getProcessId(processName: string): int {.exportpy: "get_process_id".} =
  checkRoot()
  for process in enumProcesses():
    if process.name in processName:
      return process.pid
  raise newException(Exception, fmt"Process ’{processName}’ not found")

proc getProcessName(pid: int): string {.exportpy: "get_process_name".} =
  checkRoot()
  for process in enumProcesses():
    if process.pid == pid:
      return process.name
  raise newException(Exception, fmt"Process ’{pid}’ not found")

proc openProcess(process: PyObject, debug: bool = false): Process {.exportpy: "open_process".} =
  let
    pyMod = pyBuiltinsModule()
    pyInt = pyMod.getAttr("int")
    pyStr = pyMod.str
    objT = pyMod.type(process)

  var sPid: int
  if objT == pyInt:
    sPid = process.to(int)
    if not pidExists(sPid):
      raise newException(Exception, fmt"Process ID ’{sPid} does not exist")
  elif objT == pyStr:
    let processName = process.to(string)
    for p in enumProcesses():
      if processName in p.name:
        sPid = p.pid
        break
    if sPid == 0:
      raise newException(Exception, fmt"Process ’{processName}’ not found")
  else:
    raise newException(Exception, "Process ID or Process Name required")

  checkRoot()
  result.debug = debug
  result.pid = sPid
  result.name = getProcessName(sPid)

  when defined(windows):
    result.handle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, sPid.DWORD)
    if result.handle == FALSE:
      raise newException(Exception, fmt"Unable to open Process [Pid: {sPid}] {getErrorStr()}")

proc closeProcess(process: Process) {.exportpy: "close_process".} =
  when defined(windows):
    CloseHandle(process.handle)

iterator enumModules(process: Process): Module {.exportpy: "enum_modules"} =
  when defined(linux):
    checkRoot()
    var modTable: Table[string, Module]
    for l in lines(fmt"/proc/{process.pid}/maps"):
      let s = l.split("/")
      if s.len > 1:
        var
          pageStart, pageEnd: ByteAddress
          modName = s[^1]
        discard scanf(l, "$h-$h", pageStart, pageEnd)
        if modName notin modTable:
          modTable[modName] = Module(name: modName)
          modTable[modName].base = pageStart
          modTable[modName].size = pageEnd - modTable[modName].base
        else:
          modTable[modName].size = pageEnd - modTable[modName].base

    for name, module in modTable:
      modTable[name].‘end‘ = module.base + module.size
      yield modTable[name]

  elif defined(windows):
    var
      module: Module
      modules = newSeq[HMODULE](1024)
      moduleInfo: MODULEINFO

    EnumProcessModulesEx(process.handle, modules[0].addr, modules.len.DWORD, nil, LIST_MODULE
S_ALL)
    for m in modules:
      GetModuleInformation(process.handle, m, moduleInfo.addr, sizeof(moduleInfo).DWORD)
      if not moduleInfo.lpBaseOfDll.isNil():
        var modName: array[1024, char]
        GetModuleBaseNameA(process.handle, m, modName[0].addr, sizeof(modName).DWORD)
        module.name = nullTerminated($$modName)
        module.base = cast[ByteAddress](moduleInfo.lpBaseOfDll)
        module.size = moduleInfo.SizeOfImage
        module.‘end‘ = module.base + module.size
        yield module

proc getModule(process: Process, moduleName: string): Module {.exportpy: "get_module".} =
  checkRoot()
  for module in enumModules(process):
    if moduleName == module.name:
      return module
  raise newException(Exception, fmt"Module ’{moduleName}’ not found")

iterator enumMemoryRegions(process: Process, module: Module): Page {.exportpy: "enum_memory_regi
ons".} =
  var result: Page
  when defined(linux):
    checkRoot()
    var pageStart, pageEnd: ByteAddress
    for l in lines(fmt"/proc/{process.pid}/maps"):
      if module.name in l and scanf(l, "$h-$h", result.start, result.‘end‘):
        result.size = pageEnd - pageStart
        yield result
  elif defined(windows):
    var
      mbi = MEMORY_BASIC_INFORMATION()
      curAddr = module.base
    while VirtualQueryEx(process.handle, cast[LPCVOID](curAddr), mbi.addr, sizeof(mbi).SIZE_T) != 0 and
 curAddr != module.‘end‘:
      result.start = curAddr
      result.‘end‘ = result.start + mbi.RegionSize.int
      result.size = mbi.RegionSize.int
      curAddr += result.size
      yield result

proc read*(process: Process, address: ByteAddress, t: typedesc): t =
  when defined(linux):
    var
      ioSrc, ioDst: IOVec
      size = sizeof(t).uint

    ioDst.iov_base = result.addr
    ioDst.iov_len = size
    ioSrc.iov_base = cast[pointer](address)
    ioSrc.iov_len = size
    if process_vm_readv(process.pid, ioDst.addr, 1, ioSrc.addr, 1, 0) == -1:
      memoryErr("Read", address)
  elif defined(windows):
    if ReadProcessMemory(
      process.handle, cast[pointer](address), result.addr, sizeof(t), nil
    ) == FALSE:
      memoryErr("Read", address)

  if process.debug:
    echo "[R] [", type(result), "] 0x", address.toHex(), " -> ", result

proc write*(process: Process, address: ByteAddress, data: auto) =
  when defined(linux):
    var
      ioSrc, ioDst: IOVec
      size = sizeof(data).uint
      d = data

    ioSrc.iov_base = d.addr
    ioSrc.iov_len = size
    ioDst.iov_base = cast[pointer](address)
    ioDst.iov_len = size
    if process_vm_writev(process.pid, ioSrc.addr, 1, ioDst.addr, 1, 0) == -1:
      memoryErr("Write", address)
  elif defined(windows):
    if WriteProcessMemory(
      process.handle, cast[pointer](address), data.unsafeAddr, sizeof(data), nil
    ) == FALSE:
      memoryErr("Write", address)

  if process.debug:
    echo "[W] [", type(data), "] 0x", address.toHex(), " -> ", data

proc writeArray*[T](process: Process, address: ByteAddress, data: openArray[T]): int {.discardable.} =
  when defined(linux):
    var
      ioSrc, ioDst: IOVec
      size = (sizeof(T) * data.len).uint

    ioSrc.iov_base = data.unsafeAddr
    ioSrc.iov_len = size
    ioDst.iov_base = cast[pointer](address)
    ioDst.iov_len = size
    if process_vm_writev(process.pid, ioSrc.addr, 1, ioDst.addr, 1, 0) == -1:
      memoryErr("WriteArray", address)
  elif defined(windows):
    if WriteProcessMemory(
      process.handle, cast[pointer](address), data.unsafeAddr, sizeof(T) * data.len, nil
    ) == FALSE:
      memoryErr("WriteArray", address)

  if process.debug:
    echo "[W] [", type(data), "] 0x", address.toHex(), " -> ", data

proc readSeq*(process: Process, address: ByteAddress, size: int, t: typedesc = byte): seq[t] =
  result = newSeq[t](size)
  when defined(linux):
    var
      ioSrc, ioDst: IOVec
      bsize = (size * sizeof(t)).uint

    ioDst.iov_base = result[0].addr
    ioDst.iov_len = bsize
    ioSrc.iov_base = cast[pointer](address)
    ioSrc.iov_len = bsize
    if process_vm_readv(process.pid, ioDst.addr, 1, ioSrc.addr, 1, 0) == -1:
      memoryErr("readSeq", address)
  elif defined(windows):
    if ReadProcessMemory(
      process.handle, cast[pointer](address), result[0].addr, size * sizeof(t), nil
    ) == FALSE:
      memoryErr("readSeq", address)

  if process.debug:
    echo "[R] [", type(result), "] 0x", address.toHex(), " -> ", result

proc aob1(pattern: string, byteBuffer: seq[byte], single: bool): seq[ByteAddress] =
  # Credits to Iago Beuller
  const
    wildCard = ’?’
    doubleWildCard = "??"
    wildCardIntL = 256
    wildCardIntR = 257
    doubleWildCardInt = 258

  proc splitPattern(pattern: string): seq[string] =
    var patt = pattern.replace(" ", "")
    try:
      for i in countup(0, patt.len-1, 2):
        result.add(patt[i..i+1])
    except CatchableError:
      raise newException(Exception, "Invalid pattern")

  proc patternToInts(pattern: seq[string]): seq[int] =
    for hex in pattern:
      if wildCard in hex:
        if hex == doubleWildCard:
```

```
        result.add(doubleWildCardInt)
      elif hex[0] == wildCard:
        result.add(wildCardIntL)
      else:
        result.add(wildCardIntR)
    else:
        result.add(parseHexInt(hex))

proc getIndexMatchOrder(pattern: seq[string]): seq[int] =
  let middleIndex = (pattern.len div 2) - 1
  var midHexByteIndex, lastHexByteIndex: int

  for i, hb in pattern:
    if hb != doubleWildCard:
      if not (wildCard in hb):
        if i <= middleIndex or midHexByteIndex == 0:
          midHexByteIndex = i
        lastHexByteIndex = i
      result.add(i)

  discard result.pop()
  result.delete(result.find(midHexByteIndex))
  result.insert(midHexByteIndex, 1)
  result.insert(lastHexByteIndex, 0)

let
  hexPattern = splitPattern(pattern)
  intsPattern = patternToInts(hexPattern)
  pIndexMatchOrder = getIndexMatchOrder(hexPattern)

if pIndexMatchOrder.len == 0:
  return

var
  found: bool
  b, p: int

for i in 0..byteBuffer.len-hexPattern.len:
  found = true
  for pId in pIndexMatchOrder:
    b = byteBuffer[i+pId].int
    p = intsPattern[pId]

    if p != b:
      found = false

      if p == wildCardIntL:
        if b.toHex(1)[0] == hexPattern[pId][1]:
          found = true
        else:
          break
      elif p == wildCardIntR and b.toHex(2)[0] == hexPattern[pId][0]:
        found = true
      else:
        break
    if found:
      result.add(i)
      if single:
        return

proc aob2(pattern: string, byteBuffer: seq[byte], single: bool): seq[ByteAddress] =
  const
    wildCard = "??"
    wildCardByte = 200.byte # Not safe

  proc patternToBytes(pattern: string): seq[byte] =
    var patt = pattern.replace(" ", "")
    try:
      for i in countup(0, patt.len-1, 2):
        let hex = patt[i..i+1]
        if hex == wildCard:
          result.add(wildCardByte)
        else:
          result.add(parseHexInt(hex).byte)
    except CatchableError:
      raise newException(Exception, "Invalid pattern")

  let bytePattern = patternToBytes(pattern)
  for curIndex, _ in byteBuffer:
    for sigIndex, s in bytePattern:
      if byteBuffer[curIndex + sigIndex] != s and s != wildCardByte:
        break
      elif sigIndex == bytePattern.len-1:
        result.add(curIndex)
        if single:
          return
        break

proc aobScanModule(process: Process, moduleName, pattern: string, relative: bool = false, single: bool =
 true, algorithm: int = 0): seq[ByteAddress] {.exportpy: "aob_scan_module".} =
  let
    module = getModule(process, moduleName)
    # TODO: Reading a whole module is a bad idea. Read pages instead.
    byteBuffer = process.readSeq(module.base, module.size)

  result = if algorithm == 0: aob1(pattern, byteBuffer, single) else: aob2(pattern, byteBuffer, single)
  if result.len != 0:
    if not relative:
      for i, a in result:
        result[i] += module.base

proc aobScanRange(process: Process, pattern: string, rangeStart, rangeEnd: ByteAddress, relative: bool
= false, single: bool = true, algorithm: int = 0): seq[ByteAddress] {.exportpy: "aob_scan_range".} =
  if rangeStart >= rangeEnd:
    raise newException(Exception, "Invalid range (rangeStart > rangeEnd)")

  let byteBuffer = process.readSeq(rangeStart, rangeEnd - rangeStart)

  result = if algorithm == 0: aob1(pattern, byteBuffer, single) else: aob2(pattern, byteBuffer, single)
  if result.len != 0:
    if not relative:
      for i, a in result:
        result[i] += rangeStart

proc pageProtection(process: Process, address: ByteAddress, newProtection: int32): int32 {.exportpy: "page_protection".} =
  when defined(linux):
    ptrace.pageProtection(process.pid, address, newProtection)
  elif defined(windows):
    var mbi = MEMORY_BASIC_INFORMATION()
    discard VirtualQueryEx(process.handle, cast[LPCVOID](address), mbi.addr, sizeof(mbi).SIZE_T)
    discard VirtualProtectEx(process.handle, cast[LPCVOID](address), mbi.RegionSize, newProtection, result.addr)

proc allocateMemory(process: Process, size: int, protection: int32 = 0): ByteAddress {.exportpy: "allocate_memory".} =
  when defined(linux):
    var prot = PROT_READ or PROT_WRITE or PROT_EXEC
    if protection != 0:
      prot = protection
    ptrace.allocateMemory(process.pid, size, prot)
  elif defined(windows):
    var prot = PAGE_EXECUTE_READWRITE
    if protection != 0:
      prot = protection
    cast[ByteAddress](VirtualAllocEx(process.handle, nil, size, MEM_COMMIT or MEM_RESERVE, prot.int32))

--------------
File: memop.nim
--------------

import
  encodings, strutils,
  nimraylib_now/raylib,
  nimpy, memcore

pyExportModule("pyMeow")

proc pointerChain32(process: Process, base: ByteAddress, offsets: openArray[int]): int32 {.exportpy: "pointer_chain_32".} =
  result = process.read(base, int32)
  for offset in offsets[0..^2]:
    result = process.read(result + offset, int32)
  result = result + offsets[^1].int32

proc pointerChain64(process: Process, base: ByteAddress, offsets: openArray[int]): int64 {.exportpy: "pointer_chain_64".} =
  result = process.read(base, int64)
  for offset in offsets[0..^2]:
    result = process.read((result + offset).ByteAddress, int64)
  result = result + offsets[^1]

proc pointerChain(process: Process, baseAddr: ByteAddress, offsets: openArray[int], size: int = 8): ByteA
```

```
ddress {.exportpy: "pointer_chain".} =
  result = if size == 8: process.read(baseAddr, ByteAddress) else: process.read(baseAddr, int32)
  for o in offsets[0..^2]:
    result = if size == 8: process.read(result + o, ByteAddress) else: process.read(result + o, int32)
  result = result + offsets[^1]

proc readString(process: Process, address: ByteAddress, size: int = 30): string {.exportpy: "r_string".} =
  let s = process.readSeq(address, size, char)
  $cast[cstring](s[0].unsafeAddr)

proc bytesToString(process: Process, address: ByteAddress, size: int): string {.exportpy: "bytes_to_string"
.} =
  let s = process.readSeq(address, size, char)
  s.join("").convert("utf-8", "utf-16").strip()

proc readInt(process: Process, address: ByteAddress): int32 {.exportpy: "r_int".} =
  process.read(address, int32)

proc readInts(process: Process, address: ByteAddress, size: int32): seq[int32] {.exportpy: "r_ints".} =
  process.readSeq(address, size, int32)

proc readInt16(process: Process, address: ByteAddress): int16 {.exportpy: "r_int16".} =
  process.read(address, int16)

proc readInts16(process: Process, address: ByteAddress, size: int32): seq[int16] {.exportpy: "r_ints16".} =

  process.readSeq(address, size, int16)

proc readInt64(process: Process, address: ByteAddress): int64 {.exportpy: "r_int64".} =
  process.read(address, int64)

proc readInts64(process: Process, address: ByteAddress, size: int32): seq[int64] {.exportpy: "r_ints64".} =

  process.readSeq(address, size, int64)

proc readUInt(process: Process, address: ByteAddress): uint32 {.exportpy: "r_uint".} =
  process.read(address, uint32)

proc readUInts(process: Process, address: ByteAddress, size: int32): seq[uint32] {.exportpy: "r_uints".} =
  process.readSeq(address, size, uint32)

proc readUInt64(process: Process, address: ByteAddress): uint64 {.exportpy: "r_uint64".} =
  process.read(address, uint64)

proc readUInts64(process: Process, address: ByteAddress, size: int32): seq[uint64] {.exportpy: "r_uints64
".} =
  process.readSeq(address, size, uint64)

proc readFloat(process: Process, address: ByteAddress): float32 {.exportpy: "r_float".} =
  process.read(address, float32)

proc readFloats(process: Process, address: ByteAddress, size: int32): seq[float32] {.exportpy: "r_floats".}
=
  process.readSeq(address, size, float32)
```

```
proc readFloat64(process: Process, address: ByteAddress): float64 {.exportpy: "r_float64".} =
  process.read(address, float64)

proc readFloats64(process: Process, address: ByteAddress, size: int32): seq[float64] {.exportpy: "r_floats
64".} =
  process.readSeq(address, size, float64)

proc readByte(process: Process, address: ByteAddress): byte {.exportpy: "r_byte".} =
  process.read(address, byte)

proc readBytes(process: Process, address: ByteAddress, size: int32): seq[byte] {.exportpy: "r_bytes".} =
  process.readSeq(address, size, byte)

proc readVec2(process: Process, address: ByteAddress): Vector2 {.exportpy: "r_vec2".} =
  process.read(address, Vector2)

proc readVec3(process: Process, address: ByteAddress): Vector3 {.exportpy: "r_vec3".} =
  process.read(address, Vector3)

proc readBool(process: Process, address: ByteAddress): bool {.exportpy: "r_bool".} =
  process.read(address, byte).bool

proc writeString(process: Process, address: ByteAddress, data: string) {.exportpy: "w_string".} =
  process.writeArray(address, data.cstring.toOpenArrayByte(0, data.high))

template writeData =
  process.write(address, data)

template writeDatas =
  process.writeArray(address, data)

proc writeInt(process: Process, address: ByteAddress, data: int32) {.exportpy: "w_int".} =
  writeData

proc writeInts(process: Process, address: ByteAddress, data: openArray[int32]) {.exportpy: "w_ints".} =
  writeDatas

proc writeInt16(process: Process, address: ByteAddress, data: int16) {.exportpy: "w_int16".} =
  writeData

proc writeInts16(process: Process, address: ByteAddress, data: openArray[int16]) {.exportpy: "w_ints16".}
 =
  writeDatas

proc writeInt64(process: Process, address: ByteAddress, data: int64) {.exportpy: "w_int64".} =
  writeData

proc writeInts64(process: Process, address: ByteAddress, data: openArray[int64]) {.exportpy: "w_ints64".}
 =
  writeDatas

proc writeUInt(process: Process, address: ByteAddress, data: uint32) {.exportpy: "w_uint".} =
  writeData

proc writeUInts(process: Process, address: ByteAddress, data: openArray[uint32]) {.exportpy: "w_uints".}
```

```
=
  writeDatas

proc writeUInt64(process: Process, address: ByteAddress, data: uint64) {.exportpy: "w_uint64".} =
  writeData

proc writeUInts64(process: Process, address: ByteAddress, data: openArray[uint64]) {.exportpy: "w_uints
64".} =
  writeDatas

proc writeFloat(process: Process, address: ByteAddress, data: float32) {.exportpy: "w_float".} =
  writeData

proc writeFloats(process: Process, address: ByteAddress, data: openArray[float32]) {.exportpy: "w_floats"
.} =
  writeDatas

proc writeFloat64(process: Process, address: ByteAddress, data: float64) {.exportpy: "w_float64".} =
  writeData

proc writeFloats64(process: Process, address: ByteAddress, data: openArray[float64]) {.exportpy: "w_floa
ts64".} =
  writeDatas

proc writeByte(process: Process, address: ByteAddress, data: byte) {.exportpy: "w_byte".} =
  writeData

proc writeBytes(process: Process, address: ByteAddress, data: openArray[byte]) {.exportpy: "w_bytes".} =

  writeDatas

proc writeVec2(process: Process, address: ByteAddress, data: Vector2) {.exportpy: "w_vec2".} =
  writeData

proc writeVec3(process: Process, address: ByteAddress, data: Vector3) {.exportpy: "w_vec3".} =
  writeData

proc writeBool(process: Process, address: ByteAddress, data: bool) {.exportpy: "w_bool".} =
  process.write(address, data.byte)

--------------
File: overlay.nim
--------------

import
  strformat, nimpy,
  nimraylib_now as rl,
  input

from utils import getDisplayResolution

pyExportModule("pyMeow")

when defined(linux):
  import strutils, osproc
```

```
elif defined(windows):
  import winim
  SetProcessDPIAware()

type OverlayOptions = object
  exitKey: int
  target: string
  trackTarget: bool
  targetX, targetY: int
  targetWidth, targetHeight: int

var overlayOpts: OverlayOptions

proc getWindowInfo(name: string): tuple[x, y, width, height: int] {.exportpy: "get_window_info".} =
  when defined(linux):
    let
      p = startProcess("xwininfo", "", ["-name", name], options={poUsePath, poStdErrToStdOut})
      (lines, exitCode) = p.readLines()

    template parseI: int32 = parseInt(i.split()[^1])

    if exitCode != 1:
      for i in lines:
        if "error" in i:
          raise newException(Exception, fmt"Window ({name}) not found")
        if "te upper-left X:" in i:
          result.x = parseI
        elif "te upper-left Y:" in i:
          result.y = parseI
        elif "Width:" in i:
          result.width = parseI
        elif "Height:" in i:
          result.height = parseI
    else:
      raise newException(Exception, "XWinInfo failed (installed ’xwininfo’?)")
  elif defined(windows):
    var
      rect: RECT
      winInfo: WINDOWINFO
    let hwnd = FindWindowA(nil, name)
    if hwnd == 0:
      raise newException(Exception, fmt"Window ({name}) not found")
    discard GetClientRect(hwnd, rect.addr)
    discard GetWindowInfo(hwnd, winInfo.addr)
    result.x = winInfo.rcClient.left
    result.y = winInfo.rcClient.top
    result.width = rect.right
    result.height = rect.bottom

proc overlayInit(target: string = "Full", fps: int = 0, title: string = "PyMeow", logLevel: int = 5, exitKey: int = -1, trackTarget: bool = false) {.exportpy: "overlay_init"} =
  let res = getDisplayResolution()
  setTraceLogLevel(logLevel)
  setTargetFPS(fps.cint)
  setConfigFlags(WINDOW_UNDECORATED)
  setConfigFlags(WINDOW_MOUSE_PASSTHROUGH)
  setConfigFlags(WINDOW_TRANSPARENT)
  setConfigFlags(WINDOW_TOPMOST)
  when defined(windows):
    # Multisample seems to void the transparent framebuffer on most linux distro’s.
    # Needs more tests
    setConfigFlags(MSAA_4X_HINT)
  initWindow(res[0] - 1, res[1] - 1, title)

  if target != "Full":
    let winInfo = getWindowInfo(target)
    overlayOpts.targetX = winInfo.x
    overlayOpts.targetY = winInfo.y
    overlayOpts.targetWidth = winInfo.width
    overlayOpts.targetHeight = winInfo.height
    setWindowSize(winInfo.width, winInfo.height)
    setWindowPosition(winInfo.x, winInfo.y)
  overlayOpts.target = target
  overlayOpts.trackTarget = trackTarget

  if exitKey != -1:
    overlayOpts.exitKey = exitKey
  else:
    when defined(windows):
      overlayOpts.exitKey = 0x23
    elif defined(linux):
      overlayOpts.exitKey = 0xFF57
  setExitKey(KeyboardKey.NULL)

proc overlayLoop: bool {.exportpy: "overlay_loop".} =
  clearBackground(Blank)
  if keyPressed(overlayOpts.exitKey):
    rl.closeWindow()
  if overlayOpts.trackTarget:
    let winInfo = getWindowInfo(overlayOpts.target)
    if winInfo.x != overlayOpts.targetX or winInfo.y != overlayOpts.targetY:
      overlayOpts.targetX = winInfo.x
      overlayOpts.targetY = winInfo.y
      rl.setWindowPosition(winInfo.x, winInfo.y)
    if winInfo.width != overlayOpts.targetWidth or winInfo.height != overlayOpts.targetHeight:
      overlayOpts.targetWidth = winInfo.width
      overlayOpts.targetHeight = winInfo.height
      rl.setWindowSize(winInfo.width, winInfo.height)
  not windowShouldClose()

proc beginDrawing {.exportpy: "begin_drawing".} =
  rl.beginDrawing()

proc endDrawing {.exportpy: "end_drawing".} =
  rl.endDrawing()

proc getFPS: int {.exportpy: "get_fps".} =
  rl.getFPS()

proc getScreenHeight: int {.exportpy: "get_screen_height".} =
  rl.getScreenHeight()

proc getScreenWidth: int {.exportpy: "get_screen_width".} =
  rl.getScreenWidth()

proc getWindowPosition: Vector2 {.exportpy: "get_window_position".} =
  rl.getWindowPosition()

proc getWindowHandle(): int {.exportpy: "get_window_handle".} =
  cast[int](rl.getWindowHandle())

proc setFPS(fps: int) {.exportpy: "set_fps".} =
  rl.setTargetFPS(fps)

proc setWindowPosition(x, y: int) {.exportpy: "set_window_position".} =
  rl.setWindowPosition(x, y)

proc setWindowIcon(filePath: string) {.exportpy: "set_window_icon".} =
  rl.setWindowIcon(rl.loadImage(filePath))

proc setWindowFlag(flag: int) {.exportpy: "set_window_flag".} =
  let f = flag.cuint
  if rl.isWindowState(f):
    rl.clearWindowState(f)
    return
  rl.setWindowState(f)

proc setWindowSize(width, height: int) {.exportpy: "set_window_size".} =
  rl.setWindowSize(width, height)

proc setWindowTitle(title: string) {.exportpy: "set_window_title".} =
  rl.setWindowTitle(title)

proc takeScreenshot(fileName: string) {.exportpy: "take_screenshot".} =
  rl.takeScreenshot(fileName)

proc overlayClose {.exportpy: "overlay_close".} =
  rl.closeWindow()

proc toggleMouse {.exportpy: "toggle_mouse".} =
  if isWindowState(WINDOW_MOUSE_PASSTHROUGH):
    clearWindowState(WINDOW_MOUSE_PASSTHROUGH)
    return
  setWindowState(WINDOW_MOUSE_PASSTHROUGH)

--------------
File: pixel.nim
--------------

import
  sequtils, nimpy,
  nimraylib_now as rl

from utils import getDisplayResolution, compareColorPCT
from input import mousePosition
```

```
pyExportModule("pyMeow")

type Pixel = object
  x, y: int
  color: rl.Color

when defined(linux):
  import x11/[x, xlib, xutil]
  var disp: PDisplay
elif defined(windows):
  import winim

iterator pixelEnumRegion(x, y, width, height: float): Pixel {.exportpy: "pixel_enum_region".} =
  when defined(linux):
    if disp.isNil:
      disp = XOpenDisplay(nil)

    let
      root = XRootWindow(disp, 0)
      shot = XGetImage(
        disp, root,
        x.int, y.int,
        width.cuint, height.cuint,
        AllPlanes, ZPixmap
      )
    defer: discard XDestroyImage(shot)

    var p: Pixel
    p.color.a = 255
    for x in 0..<width.int:
      for y in 0..<height.int:
        var xp = XGetPixel(shot, x, y)
        p.x = x
        p.y = y
        p.color.r = ((xp and shot.red_mask) shr 16).uint8
        p.color.g = ((xp and shot.green_mask) shr 8).uint8
        p.color.b = (xp and shot.blue_mask).uint8
        yield p

  elif defined(windows):
    var
      hdc = GetDC(0)
      hDest = CreateCompatibleDC(hdc)

    var hbDesktop = CreateCompatibleBitmap(hdc, width.int, height.int)
    SelectObject(hDest, hbDesktop)
    BitBlt(hDest, 0, 0, width.int, height.int, hdc, x.int, y.int, SRCCOPY)

    var
      size = (width * height * 4).int
      pBits = newSeq[uint8](size)

    GetBitmapBits(hbDesktop, size, cast[LPVOID](pBits[0].addr))
    DeleteObject(hbDesktop)
    DeleteDC(hDest)
    ReleaseDC(0, hdc)
    for y in 0..<height.int:
      for x in 0..<width.int:
        var i = (y * width.int + x) * 4
        yield Pixel(
            x: x,
            y: y,
            color: rl.Color(
              a: 255,
              b: pBits[i],
              g: pBits[i + 1],
              r: pBits[i + 2],
            )
          )

iterator pixelEnumScreen: Pixel {.exportpy: "pixel_enum_screen".} =
  let res = getDisplayResolution()
  for p in pixelEnumRegion(0, 0, res[0].float, res[1].float):
    yield p

proc pixelAtMouse: Pixel {.exportpy: "pixel_at_mouse".} =
  let pos = mousePosition()
  result = pixelEnumRegion(pos.x, pos.y, 1, 1).toSeq()[0]

proc pixelSaveToFile(x, y, width, height: float, fileName: string): bool {.exportpy: "pixel_save_to_file".} =
  rl.setTraceLogLevel(5)
  var img = genImageColor(width.int, height.int, Blank)
  for p in pixelEnumRegion(x, y, width, height):
    imageDrawPixel(
      img.addr,
      p.x, p.y,
      rl.Color(r: p.color.r, g: p.color.g, b: p.color.b, a: 255)
    )
  exportImage(img, (fileName & ".png").cstring)

iterator pixelSearchColors(x, y, width, height: float, colors: openArray[rl.Color], similarity: float): Pixel {.exportpy: "pixel_search_colors".} =
  for p in pixelEnumRegion(x, y, width, height):
    for color in colors:
      if compareColorPCT(p.color, color) >= similarity:
        yield p

--------------
File: ptrace.nim
--------------

#[
  Credits to:
    - https://github.com/ba0f3/ptrace.nim
    - https://guidedhacking.com/members/obdr.128625/

  Currently just x64 processes are supported
]#
```

```
import
  posix, strformat,
  strscans, strutils

{.pragma: sys, importc, header: "sys/syscall.h".}

proc ptrace[T](request: cint, pid: int, a: pointer, data: T): pointer {.cdecl, importc, header: "sys/ptrace.h", discardable.}

const
  PTRACE_TRACEME* = 0
  PTRACE_PEEKTEXT* = 1
  PTRACE_PEEKDATA* = 2
  PTRACE_PEEKUSER* = 3
  PTRACE_POKETEXT* = 4
  PTRACE_POKEDATA* = 5
  PTRACE_POKEUSER* = 6
  PTRACE_CONT* = 7
  PTRACE_KILL* = 8
  PTRACE_SINGLESTEP* = 9
  PTRACE_GETREGS* = 12
  PTRACE_SETREGS* = 13
  PTRACE_ATTACH* = 16
  PTRACE_DETACH* = 17
  PTRACE_SYSCALL* = 24
  PTRACE_SETOPTIONS* = 0x4200
  PTRACE_GETEVENTMSG* = 0x4201
  PTRACE_GETSIGINFO* = 0x4202
  PTRACE_SETSIGINFO* = 0x4203
  PTRACE_SEIZE* = 0x4206
  PTRACE_INTERRUPT* = 0x4207
  PTRACE_LISTEN* = 0x4208

let
  SYS_llseek* {.sys, importc: "SYS__llseek".}: cint
  SYS_newselect* {.sys, importc: "SYS__newselect".}: cint
  SYS_sysctl* {.sys, importc: "SYS_sysctl".}: cint
  SYS_access* {.sys.}: cint
  SYS_acct* {.sys.}: cint
  SYS_add_key* {.sys.}: cint
  SYS_adjtimex* {.sys.}: cint
  SYS_afs_syscall* {.sys.}: cint
  SYS_alarm* {.sys.}: cint
  SYS_bdflush* {.sys.}: cint
  SYS_bpf* {.sys.}: cint
  SYS_break* {.sys.}: cint
  SYS_brk* {.sys.}: cint
  SYS_capget* {.sys.}: cint
  SYS_capset* {.sys.}: cint
  SYS_chdir* {.sys.}: cint
  SYS_chmod* {.sys.}: cint
  SYS_chown* {.sys.}: cint
  SYS_chown32* {.sys.}: cint
  SYS_chroot* {.sys.}: cint
  SYS_clock_adjtime* {.sys.}: cint
```

```
SYS_clock_getres* {.sys.}: cint
SYS_clock_gettime* {.sys.}: cint
SYS_clock_nanosleep* {.sys.}: cint
SYS_clock_settime* {.sys.}: cint
SYS_clone* {.sys.}: cint
SYS_close* {.sys.}: cint
SYS_creat* {.sys.}: cint
SYS_create_module* {.sys.}: cint
SYS_delete_module* {.sys.}: cint
SYS_dup* {.sys.}: cint
SYS_dup2* {.sys.}: cint
SYS_dup3* {.sys.}: cint
SYS_epoll_create* {.sys.}: cint
SYS_epoll_create1* {.sys.}: cint
SYS_epoll_ctl* {.sys.}: cint
SYS_epoll_pwait* {.sys.}: cint
SYS_epoll_wait* {.sys.}: cint
SYS_eventfd* {.sys.}: cint
SYS_eventfd2* {.sys.}: cint
SYS_execve* {.sys.}: cint
SYS_execveat* {.sys.}: cint
SYS_exit* {.sys.}: cint
SYS_exit_group* {.sys.}: cint
SYS_faccessat* {.sys.}: cint
SYS_fadvise64* {.sys.}: cint
SYS_fadvise64_64* {.sys.}: cint
SYS_fallocate* {.sys.}: cint
SYS_fanotify_init* {.sys.}: cint
SYS_fanotify_mark* {.sys.}: cint
SYS_fchdir* {.sys.}: cint
SYS_fchmod* {.sys.}: cint
SYS_fchmodat* {.sys.}: cint
SYS_fchown* {.sys.}: cint
SYS_fchown32* {.sys.}: cint
SYS_fchownat* {.sys.}: cint
SYS_fcntl* {.sys.}: cint
SYS_fcntl64* {.sys.}: cint
SYS_fdatasync* {.sys.}: cint
SYS_fgetxattr* {.sys.}: cint
SYS_finit_module* {.sys.}: cint
SYS_flistxattr* {.sys.}: cint
SYS_flock* {.sys.}: cint
SYS_fork* {.sys.}: cint
SYS_fremovexattr* {.sys.}: cint
SYS_fsetxattr* {.sys.}: cint
SYS_fstat* {.sys.}: cint
SYS_fstat64* {.sys.}: cint
SYS_fstatat64* {.sys.}: cint
SYS_fstatfs* {.sys.}: cint
SYS_fstatfs64* {.sys.}: cint
SYS_fsync* {.sys.}: cint
SYS_ftime* {.sys.}: cint
SYS_ftruncate* {.sys.}: cint
SYS_ftruncate64* {.sys.}: cint
SYS_futex* {.sys.}: cint
```

```
SYS_futimesat* {.sys.}: cint
SYS_get_kernel_syms* {.sys.}: cint
SYS_get_mempolicy* {.sys.}: cint
SYS_get_robust_list* {.sys.}: cint
SYS_get_thread_area* {.sys.}: cint
SYS_getcpu* {.sys.}: cint
SYS_getcwd* {.sys.}: cint
SYS_getdents* {.sys.}: cint
SYS_getdents64* {.sys.}: cint
SYS_getegid* {.sys.}: cint
SYS_getegid32* {.sys.}: cint
SYS_geteuid* {.sys.}: cint
SYS_geteuid32* {.sys.}: cint
SYS_getgid* {.sys.}: cint
SYS_getgid32* {.sys.}: cint
SYS_getgroups* {.sys.}: cint
SYS_getgroups32* {.sys.}: cint
SYS_getitimer* {.sys.}: cint
SYS_getpgid* {.sys.}: cint
SYS_getpgrp* {.sys.}: cint
SYS_getpid* {.sys.}: cint
SYS_getpmsg* {.sys.}: cint
SYS_getppid* {.sys.}: cint
SYS_getpriority* {.sys.}: cint
SYS_getrandom* {.sys.}: cint
SYS_getresgid* {.sys.}: cint
SYS_getresgid32* {.sys.}: cint
SYS_getresuid* {.sys.}: cint
SYS_getresuid32* {.sys.}: cint
SYS_getrlimit* {.sys.}: cint
SYS_getrusage* {.sys.}: cint
SYS_getsid* {.sys.}: cint
SYS_gettid* {.sys.}: cint
SYS_gettimeofday* {.sys.}: cint
SYS_getuid* {.sys.}: cint
SYS_getuid32* {.sys.}: cint
SYS_getxattr* {.sys.}: cint
SYS_gtty* {.sys.}: cint
SYS_idle* {.sys.}: cint
SYS_init_module* {.sys.}: cint
SYS_inotify_add_watch* {.sys.}: cint
SYS_inotify_init* {.sys.}: cint
SYS_inotify_init1* {.sys.}: cint
SYS_inotify_rm_watch* {.sys.}: cint
SYS_io_cancel* {.sys.}: cint
SYS_io_destroy* {.sys.}: cint
SYS_io_getevents* {.sys.}: cint
SYS_io_setup* {.sys.}: cint
SYS_io_submit* {.sys.}: cint
SYS_ioctl* {.sys.}: cint
SYS_ioperm* {.sys.}: cint
SYS_iopl* {.sys.}: cint
SYS_ioprio_get* {.sys.}: cint
SYS_ioprio_set* {.sys.}: cint
SYS_ipc* {.sys.}: cint
```

SYS_kcmp* {.sys.}: cint
SYS_kexec_load* {.sys.}: cint
SYS_keyctl* {.sys.}: cint
SYS_kill* {.sys.}: cint
SYS_lchown* {.sys.}: cint
SYS_lchown32* {.sys.}: cint
SYS_lgetxattr* {.sys.}: cint
SYS_link* {.sys.}: cint
SYS_linkat* {.sys.}: cint
SYS_listxattr* {.sys.}: cint
SYS_llistxattr* {.sys.}: cint
SYS_lock* {.sys.}: cint
SYS_lookup_dcookie* {.sys.}: cint
SYS_lremovexattr* {.sys.}: cint
SYS_lseek* {.sys.}: cint
SYS_lsetxattr* {.sys.}: cint
SYS_lstat* {.sys.}: cint
SYS_lstat64* {.sys.}: cint
SYS_madvise* {.sys.}: cint
SYS_mbind* {.sys.}: cint
SYS_memfd_create* {.sys.}: cint
SYS_migrate_pages* {.sys.}: cint
SYS_mincore* {.sys.}: cint
SYS_mkdir* {.sys.}: cint
SYS_mkdirat* {.sys.}: cint
SYS_mknod* {.sys.}: cint
SYS_mknodat* {.sys.}: cint
SYS_mlock* {.sys.}: cint
SYS_mlockall* {.sys.}: cint
SYS_mmap* {.sys.}: cint
SYS_mmap2* {.sys.}: cint
SYS_modify_ldt* {.sys.}: cint
SYS_mount* {.sys.}: cint
SYS_move_pages* {.sys.}: cint
SYS_mprotect* {.sys.}: cint
SYS_mpx* {.sys.}: cint
SYS_mq_getsetattr* {.sys.}: cint
SYS_mq_notify* {.sys.}: cint
SYS_mq_open* {.sys.}: cint
SYS_mq_timedreceive* {.sys.}: cint
SYS_mq_timedsend* {.sys.}: cint
SYS_mq_unlink* {.sys.}: cint
SYS_mremap* {.sys.}: cint
SYS_msync* {.sys.}: cint
SYS_munlock* {.sys.}: cint
SYS_munlockall* {.sys.}: cint
SYS_munmap* {.sys.}: cint
SYS_name_to_handle_at* {.sys.}: cint
SYS_nanosleep* {.sys.}: cint
SYS_nfsservctl* {.sys.}: cint
SYS_nice* {.sys.}: cint
SYS_oldfstat* {.sys.}: cint
SYS_oldlstat* {.sys.}: cint
SYS_oldolduname* {.sys.}: cint
SYS_oldstat* {.sys.}: cint

```
SYS_olduname* {.sys.}: cint
SYS_open* {.sys.}: cint
SYS_open_by_handle_at* {.sys.}: cint
SYS_openat* {.sys.}: cint
SYS_pause* {.sys.}: cint
SYS_perf_event_open* {.sys.}: cint
SYS_personality* {.sys.}: cint
SYS_pipe* {.sys.}: cint
SYS_pipe2* {.sys.}: cint
SYS_pivot_root* {.sys.}: cint
SYS_poll* {.sys.}: cint
SYS_ppoll* {.sys.}: cint
SYS_prctl* {.sys.}: cint
SYS_pread64* {.sys.}: cint
SYS_preadv* {.sys.}: cint
SYS_prlimit64* {.sys.}: cint
SYS_process_vm_readv* {.sys.}: cint
SYS_process_vm_writev* {.sys.}: cint
SYS_prof* {.sys.}: cint
SYS_profil* {.sys.}: cint
SYS_pselect6* {.sys.}: cint
SYS_ptrace* {.sys.}: cint
SYS_putpmsg* {.sys.}: cint
SYS_pwrite64* {.sys.}: cint
SYS_pwritev* {.sys.}: cint
SYS_query_module* {.sys.}: cint
SYS_quotactl* {.sys.}: cint
SYS_read* {.sys.}: cint
SYS_readahead* {.sys.}: cint
SYS_readdir* {.sys.}: cint
SYS_readlink* {.sys.}: cint
SYS_readlinkat* {.sys.}: cint
SYS_readv* {.sys.}: cint
SYS_reboot* {.sys.}: cint
SYS_recvmmsg* {.sys.}: cint
SYS_remap_file_pages* {.sys.}: cint
SYS_removexattr* {.sys.}: cint
SYS_rename* {.sys.}: cint
SYS_renameat* {.sys.}: cint
SYS_renameat2* {.sys.}: cint
SYS_request_key* {.sys.}: cint
SYS_restart_syscall* {.sys.}: cint
SYS_rmdir* {.sys.}: cint
SYS_rt_sigaction* {.sys.}: cint
SYS_rt_sigpending* {.sys.}: cint
SYS_rt_sigprocmask* {.sys.}: cint
SYS_rt_sigqueueinfo* {.sys.}: cint
SYS_rt_sigreturn* {.sys.}: cint
SYS_rt_sigsuspend* {.sys.}: cint
SYS_rt_sigtimedwait* {.sys.}: cint
SYS_rt_tgsigqueueinfo* {.sys.}: cint
SYS_sched_get_priority_max* {.sys.}: cint
SYS_sched_get_priority_min* {.sys.}: cint
SYS_sched_getaffinity* {.sys.}: cint
SYS_sched_getattr* {.sys.}: cint
```

```
SYS_sched_getparam* {.sys.}: cint
SYS_sched_getscheduler* {.sys.}: cint
SYS_sched_rr_get_interval* {.sys.}: cint
SYS_sched_setaffinity* {.sys.}: cint
SYS_sched_setattr* {.sys.}: cint
SYS_sched_setparam* {.sys.}: cint
SYS_sched_setscheduler* {.sys.}: cint
SYS_sched_yield* {.sys.}: cint
SYS_seccomp* {.sys.}: cint
SYS_select* {.sys.}: cint
SYS_sendfile* {.sys.}: cint
SYS_sendfile64* {.sys.}: cint
SYS_sendmmsg* {.sys.}: cint
SYS_set_mempolicy* {.sys.}: cint
SYS_set_robust_list* {.sys.}: cint
SYS_set_thread_area* {.sys.}: cint
SYS_set_tid_address* {.sys.}: cint
SYS_setdomainname* {.sys.}: cint
SYS_setfsgid* {.sys.}: cint
SYS_setfsgid32* {.sys.}: cint
SYS_setfsuid* {.sys.}: cint
SYS_setfsuid32* {.sys.}: cint
SYS_setgid* {.sys.}: cint
SYS_setgid32* {.sys.}: cint
SYS_setgroups* {.sys.}: cint
SYS_setgroups32* {.sys.}: cint
SYS_sethostname* {.sys.}: cint
SYS_setitimer* {.sys.}: cint
SYS_setns* {.sys.}: cint
SYS_setpgid* {.sys.}: cint
SYS_setpriority* {.sys.}: cint
SYS_setregid* {.sys.}: cint
SYS_setregid32* {.sys.}: cint
SYS_setresgid* {.sys.}: cint
SYS_setresgid32* {.sys.}: cint
SYS_setresuid* {.sys.}: cint
SYS_setresuid32* {.sys.}: cint
SYS_setreuid* {.sys.}: cint
SYS_setreuid32* {.sys.}: cint
SYS_setrlimit* {.sys.}: cint
SYS_setsid* {.sys.}: cint
SYS_settimeofday* {.sys.}: cint
SYS_setuid* {.sys.}: cint
SYS_setuid32* {.sys.}: cint
SYS_setxattr* {.sys.}: cint
SYS_sgetmask* {.sys.}: cint
SYS_sigaction* {.sys.}: cint
SYS_sigaltstack* {.sys.}: cint
SYS_signal* {.sys.}: cint
SYS_signalfd* {.sys.}: cint
SYS_signalfd4* {.sys.}: cint
SYS_sigpending* {.sys.}: cint
SYS_sigprocmask* {.sys.}: cint
SYS_sigreturn* {.sys.}: cint
SYS_sigsuspend* {.sys.}: cint
```

```
SYS_socketcall* {.sys.}: cint
SYS_splice* {.sys.}: cint
SYS_ssetmask* {.sys.}: cint
SYS_stat* {.sys.}: cint
SYS_stat64* {.sys.}: cint
SYS_statfs* {.sys.}: cint
SYS_statfs64* {.sys.}: cint
SYS_stime* {.sys.}: cint
SYS_stty* {.sys.}: cint
SYS_swapoff* {.sys.}: cint
SYS_swapon* {.sys.}: cint
SYS_symlink* {.sys.}: cint
SYS_symlinkat* {.sys.}: cint
SYS_sync* {.sys.}: cint
SYS_sync_file_range* {.sys.}: cint
SYS_syncfs* {.sys.}: cint
SYS_sysfs* {.sys.}: cint
SYS_sysinfo* {.sys.}: cint
SYS_syslog* {.sys.}: cint
SYS_tee* {.sys.}: cint
SYS_tgkill* {.sys.}: cint
SYS_time* {.sys.}: cint
SYS_timer_create* {.sys.}: cint
SYS_timer_delete* {.sys.}: cint
SYS_timer_getoverrun* {.sys.}: cint
SYS_timer_gettime* {.sys.}: cint
SYS_timer_settime* {.sys.}: cint
SYS_timerfd_create* {.sys.}: cint
SYS_timerfd_gettime* {.sys.}: cint
SYS_timerfd_settime* {.sys.}: cint
SYS_times* {.sys.}: cint
SYS_tkill* {.sys.}: cint
SYS_truncate* {.sys.}: cint
SYS_truncate64* {.sys.}: cint
SYS_ugetrlimit* {.sys.}: cint
SYS_ulimit* {.sys.}: cint
SYS_umask* {.sys.}: cint
SYS_umount* {.sys.}: cint
SYS_umount2* {.sys.}: cint
SYS_uname* {.sys.}: cint
SYS_unlink* {.sys.}: cint
SYS_unlinkat* {.sys.}: cint
SYS_unshare* {.sys.}: cint
SYS_uselib* {.sys.}: cint
SYS_ustat* {.sys.}: cint
SYS_utime* {.sys.}: cint
SYS_utimensat* {.sys.}: cint
SYS_utimes* {.sys.}: cint
SYS_vfork* {.sys.}: cint
SYS_vhangup* {.sys.}: cint
SYS_vm86* {.sys.}: cint
SYS_vm86old* {.sys.}: cint
SYS_vmsplice* {.sys.}: cint
SYS_vserver* {.sys.}: cint
SYS_wait4* {.sys.}: cint
```

```
  SYS_waitid* {.sys.}: cint
  SYS_waitpid* {.sys.}: cint
  SYS_write* {.sys.}: cint
  SYS_writev* {.sys.}: cint

type
  Registers = object
    r15: pointer
    r14: pointer
    r13: pointer
    r12: pointer
    rbp: pointer
    rbx: pointer
    r11: pointer
    r10: pointer
    r9: pointer
    r8: pointer
    rax: pointer
    rcx: pointer
    rdx: pointer
    rsi: pointer
    rdi: pointer
    origRax: pointer
    rip: pointer
    cs: pointer
    eflags: pointer
    rsp: pointer
    ss: pointer
    fsBase: pointer
    gsBase: pointer
    ds: pointer
    es: pointer
    fs: pointer
    gs: pointer

proc injectSyscall(pid: int, syscall: int, arg0, arg1, arg2, arg3, arg4, arg5: pointer): pointer {.discardable.} =
  var
    status: cint
    regs, oldRegs: Registers
    injectionAddr: pointer

  let injectionBuf = [0x0f.byte, 0x05, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90]

  ptrace(PTRACE_ATTACH, pid, nil, nil)
  wait(status.addr)
  ptrace(PTRACE_GETREGS, pid, nil, oldRegs.addr)
  regs = oldRegs
  regs.rax = cast[pointer](syscall)
  regs.rdi = arg0
  regs.rsi = arg1
  regs.rdx = arg2
  regs.r10 = arg3
  regs.r8 = arg4
  regs.r9 = arg5
  injectionAddr = cast[pointer](regs.rip)
  var oldData = ptrace(PTRACE_PEEKDATA, pid, injectionAddr, 0)
  ptrace(PTRACE_POKEDATA, pid, injectionAddr, cast[pointer](injectionBuf))
  ptrace(PTRACE_SETREGS, pid, nil, regs.addr)
  ptrace(PTRACE_SINGLESTEP, pid, nil, nil)
  discard waitpid(pid.cint, status, WSTOPPED)
  ptrace(PTRACE_GETREGS, pid, nil, regs.addr)
  result = cast[pointer](regs.rax)
  ptrace(PTRACE_POKEDATA, pid, injectionAddr, oldData)
  ptrace(PTRACE_SETREGS, pid, nil, oldRegs.addr)
  ptrace(PTRACE_DETACH, pid, nil, nil)

proc pageProtection*(pid, src, protection: int) =
  var pageStart, pageEnd: ByteAddress
  for l in lines(fmt"/proc/{pid}/maps"):
    discard scanf(l, "$h-$h", pageStart, pageEnd)
    if src > pageStart and src < pageEnd:
      break
  injectSyscall(pid, SYS_mprotect, cast[pointer](pageStart), cast[pointer](pageEnd), cast[pointer](protectio
n), nil, nil, nil)

proc allocateMemory*(pid, size, protection: int): ByteAddress =
  let ret = injectSyscall(pid, SYS_mmap, nil, cast[pointer](size), cast[pointer](protection), cast[pointer](MAP
_ANONYMOUS or MAP_PRIVATE), cast[pointer](-1), nil)
  cast[ByteAddress](ret)

--------------
File: render.nim
--------------

import
  nimraylib_now as rl,
  nimpy, tables

pyExportModule("pyMeow")

type
  FontObj = object
    id: int
    font: Font

var
  fontTable: Table[int, FontObj]

converter toCint(x: float|int): cint = x.cint

proc drawFPS(posX, posY: float) {.exportpy: "draw_fps".} =
  rl.drawFPS(posX, posY)

proc drawText(text: string, posX, posY, fontSize: float, color: Color) {.exportpy: "draw_text".} =
  rl.drawText(text, posX, posY, fontSize, color)

proc drawPixel(posX, posY: float, color: Color) {.exportpy: "draw_pixel".} =
  rl.drawPixel(posX, posY, color)

proc drawLine(startPosX, startPosY, endPosX, endPosY: float, color: Color, thick: float = 1.0) {.exportpy: "
draw_line".} =
  rl.drawLineEx(Vector2(x: startPosX, y: startPosY), Vector2(x: endPosX, y: endPosY), thick, color)

proc drawCircle(centerX, centerY, radius: float, color: Color) {.exportpy: "draw_circle".} =
  rl.drawCircle(centerX, centerY, radius, color)

proc drawCircleLines(centerX, centerY, radius: float, color: Color) {.exportpy: "draw_circle_lines".} =
  rl.drawCircleLines(centerX, centerY, radius, color)

proc drawCircleSector(centerX, centerY, radius, startAngle, endAngle: float, segments: int, color: Color) {.
exportpy: "draw_circle_sector".} =
  rl.drawCircleSector(Vector2(x: centerX, y: centerY), radius, startAngle, endAngle, segments, color)

proc drawCircleSectorLines(centerX, centerY, radius, startAngle, endAngle: float, segments: int, color: Co
lor) {.exportpy: "draw_circle_sector_lines".} =
  rl.drawCircleSectorLines(Vector2(x: centerX, y: centerY), radius, startAngle, endAngle, segments, color)

proc drawRing(centerX, centerY, segments, innerRadius, outerRadius, startAngle, endAngle: float, color:
Color) {.exportpy: "draw_ring".} =
  rl.drawRing(Vector2(x: centerX, y: centerY), innerRadius, outerRadius, startAngle, endAngle, segments,
color)

proc drawRingLines(centerX, centerY, segments, innerRadius, outerRadius, startAngle, endAngle: float, c
olor: Color) {.exportpy: "draw_ring_lines".} =
  rl.drawRingLines(Vector2(x: centerX, y: centerY), innerRadius, outerRadius, startAngle, endAngle, segm
ents, color)

proc drawEllipse(centerX, centerY, radiusH, radiusV: float, color: Color) {.exportpy: "draw_ellipse".} =
  rl.drawEllipse(centerX, centerY, radiusH, radiusV, color)

proc drawEllipseLines(centerX, centerY, radiusH, radiusV: float, color: Color) {.exportpy: "draw_ellipse_lin
es".} =
  rl.drawEllipseLines(centerX, centerY, radiusH, radiusV, color)

proc drawRectangle(posX, posY, width, height: float, color: Color): Rectangle {.exportpy: "draw_rectangle
".} =
  result.x = posX
  result.y = posY
  result.width = width
  result.height = height
  rl.drawRectangle(posX, posY, width, height, color)

proc drawRectangleLines(posX, posY, width, height: float, color: Color, lineThick: float = 1.0): Rectangle {.
exportpy: "draw_rectangle_lines".} =
  result.x = posX
  result.y = posY
  result.width = width
  result.height = height
  rl.drawRectangleLinesEx(result, lineThick, color)

proc drawRectangleRounded(posX, posY, width, height, roundness: float, segments: int, color: Color): Re
ctangle {.exportpy: "draw_rectangle_rounded".} =
  result.x = posX
  result.y = posY
  result.width = width
  result.height = height
  rl.drawRectangleRounded(result, roundness, segments, color)

proc drawRectangleRoundedLines(posX, posY, width, height, roundness: float, segments: int, color: Color, lineThick: float = 1.0): Rectangle {.exportpy: "draw_rectangle_rounded_lines".} =
  result.x = posX
  result.y = posY
  result.width = width
  result.height = height
  rl.drawRectangleRoundedLines(result, roundness, segments, lineThick, color)

proc drawTriangle(pos1X, pos1Y, pos2X, pos2Y, pos3X, pos3Y: float, color: Color) {.exportpy: "draw_triangle".} =
  rl.drawTriangle(
    Vector2(x: pos1X, y: pos1Y),
    Vector2(x: pos2X, y: pos2Y),
    Vector2(x: pos3X, y: pos3Y),
    color
  )

proc drawTriangleLines(pos1X, pos1Y, pos2X, pos2Y, pos3X, pos3Y: float, color: Color) {.exportpy: "draw_triangle_lines".} =
  rl.drawTriangleLines(
    Vector2(x: pos1X, y: pos1Y),
    Vector2(x: pos2X, y: pos2Y),
    Vector2(x: pos3X, y: pos3Y),
    color
  )

proc drawPoly(posX, posY: float, sides: int, radius, rotation: float, color: Color) {.exportpy: "draw_poly"} =
  rl.drawPoly(Vector2(x: posX, y: posY), sides, radius, rotation, color)

proc drawPolyLines(posX, posY: float, sides: int, radius, rotation, lineThick: float, color: Color) {.exportpy: "draw_poly_lines".} =
  rl.drawPolyLinesEx(Vector2(x: posX, y: posY), sides, radius, rotation, lineThick, color)

proc loadTexture(fileName: string): Texture2D {.exportpy: "load_texture".} =
  rl.loadTexture(fileName)

proc loadTextureBytes(fileType: string, data: openArray[uint8]): Texture2D {.exportpy: "load_texture_bytes".} =
  rl.loadTextureFromImage(rl.loadImageFromMemory(fileType, data[0].unsafeAddr, data.len))

proc drawTexture(texture: Texture2D, posX, posY: float, tint: Color, rotation, scale: float) {.exportpy: "draw_texture".} =
  rl.drawTextureEx(texture, Vector2(x: posX, y: posY), rotation, scale, tint)

proc unloadTexture(texture: Texture2D) {.exportpy: "unload_texture".} =
  rl.unloadTexture(texture)

proc loadFont(fileName: string, fontId: int) {.exportpy: "load_font".} =
  fontTable[fontId] = FontObj(
    id: fontId,
    font: rl.loadFont(fileName)
  )
```

```
proc drawFont(fontId: int, text: string, posX, posY, fontSize, spacing: float, tint: Color) {.exportpy: "draw_font".} =
  if fontId notin fontTable:
    raise newException(Exception, "Unknown Font ID")
  rl.drawTextEx(fontTable[fontId].font, text, Vector2(x: posX, y: posY), fontSize, spacing, tint)

proc measureFont(fontId: int, text: string, fontSize, spacing: float): Vector2 {.exportpy: "measure_font".} =
  if fontId notin fontTable:
    raise newException(Exception, "Unknown Font ID")
  rl.measureTextEx(fontTable[fontId].font, text, fontSize, spacing)

--------------
File: sound.nim
--------------

import
  nimpy, tables,
  nimraylib_now as rl

pyExportModule("pyMeow")

type
  SoundObj = object
    id: int
    sound: Sound

var
  curSoundId = -1
  soundTable: Table[int, SoundObj]

proc loadSound(fileName: string): int {.exportpy: "load_sound".} =
  inc curSoundId
  soundTable[curSoundId] = SoundObj(id: curSoundId, sound: rl.loadSound(fileName))
  result = curSoundId

proc unloadSound(soundId: int) {.exportpy: "unload_sound".} =
  rl.unloadSound(soundTable[soundId].sound)
  soundTable.del(soundId)

proc playSound(soundId: int) {.exportpy: "play_sound".} =
  rl.playSound(soundTable[soundId].sound)

proc pauseSound(soundId: int) {.exportpy: "pause_sound".} =
  rl.pauseSound(soundTable[soundId].sound)

proc resumeSound(soundId: int) {.exportpy: "resume_sound".} =
  rl.resumeSound(soundTable[soundId].sound)

proc stopSound(soundId: int) {.exportpy: "stop_sound".} =
  rl.stopSound(soundTable[soundId].sound)

proc playMultiSound(soundId: int) {.exportpy: "play_multisound".} =
  rl.playSoundMulti(soundTable[soundId].sound)
```

```
proc stopMultiSound() {.exportpy: "stop_multisound".} =
  rl.stopSoundMulti()

proc setSoundVolume(soundId: int, volume: int): bool {.exportpy: "set_sound_volume".} =
  rl.setSoundVolume(soundTable[soundId].sound, volume.float / 100.0)

proc isSoundPlaying(soundId: int): bool {.exportpy: "is_sound_playing".} =
  rl.isSoundPlaying(soundTable[soundId].sound)

--------------
File: utils.nim
--------------

import
  colors, nimpy,
  nimraylib_now/raylib as rl

pyExportModule("pyMeow")

when defined(linux):
  import
    osproc, strscans, x11/xlib
elif defined(windows):
  import winim

proc newColor(r, g, b, a: uint8): rl.Color {.exportpy: "new_color".} =
  rl.Color(r: r, g: g, b: b, a: a)

proc newColorHex(hexValue: uint): rl.Color {.exportpy: "new_color_hex".} =
  rl.getColor(hexValue.cuint)

proc getColor(colorName: string): rl.Color {.exportpy: "get_color".} =
  try:
    let c = parseColor(colorName).extractRGB()
    rl.Color(
      r: c.r.uint8,
      g: c.g.uint8,
      b: c.b.uint8,
      a: 255,
    )
  except ValueError:
    rl.Color(
      r: 0,
      g: 0,
      b: 0,
      a: 255,
    )

proc fadeColor(color: rl.Color, alpha: float): rl.Color {.exportpy: "fade_color".} =
  rl.fade(color, alpha)

proc measureText(text: string, fontSize: cint): int {.exportpy: "measure_text".} =
  rl.measureText(text, fontSize)

proc runTime: float64 {.exportpy: "run_time".} =
  rl.getTime()

proc worldToScreen(matrix: array[0..15, float], pos: Vector3, algo: int = 0): Vector2 {.exportpy: "world_to_screen".} =
  var
    clip: Vector3
    ndc: Vector2

  if algo == 0:
    clip.z = pos.x * matrix[3] + pos.y * matrix[7] + pos.z * matrix[11] + matrix[15]
    clip.x = pos.x * matrix[0] + pos.y * matrix[4] + pos.z * matrix[8] + matrix[12]
    clip.y = pos.x * matrix[1] + pos.y * matrix[5] + pos.z * matrix[9] + matrix[13]
  elif algo == 1:
    clip.z = pos.x * matrix[12] + pos.y * matrix[13] + pos.z * matrix[14] + matrix[15]
    clip.x = pos.x * matrix[0] + pos.y * matrix[1] + pos.z * matrix[2] + matrix[3]
    clip.y = pos.x * matrix[4] + pos.y * matrix[5] + pos.z * matrix[6] + matrix[7]

  if clip.z < 0.2:
    raise newException(Exception, "2D Position out of bounds")

  ndc.x = clip.x / clip.z
  ndc.y = clip.y / clip.z
  result.x = (getScreenWidth() / 2 * ndc.x) + (ndc.x + getScreenWidth() / 2)
  result.y = -(getScreenHeight() / 2 * ndc.y) + (ndc.y + getScreenHeight() / 2)

proc checkCollisionPointRec(pointX, pointY: float, rec: rl.Rectangle): bool {.exportpy: "check_collision_point_rec".} =
  rl.checkCollisionPointRec(Vector2(x: pointX, y: pointY), rec)

proc checkCollisionRecs(rec1, rec2: rl.Rectangle): bool {.exportpy: "check_collision_recs".} =
  rl.checkCollisionRecs(rec1, rec2)

proc checkCollisionCircleRec(posX, posY, radius: float, rec: rl.Rectangle): bool {.exportpy: "check_collision_circle_rec".} =
  rl.checkCollisionCircleRec(Vector2(x: posX, y: posY), radius, rec)

proc checkCollisionLines(startPos1X, endPos1X, startPos1Y, endPos1Y, startPos2X, startPos2Y, endPos2X, endPos2Y: float): Vector2 {.exportpy: "check_collision_lines".} =
  discard rl.checkCollisionLines(
    Vector2(x: startPos1X, y: startPos1Y),
    Vector2(x: endPos1X, y: endPos1Y),
    Vector2(x: startPos2X, y: startPos2Y),
    Vector2(x: endPos2X, y: endPos2Y),
    result.addr
  )

proc checkCollisionCircles(pos1X, pos1Y, radius1, pos2X, pos2Y, radius2: float): bool {.exportpy: "check_collision_circles".} =
  rl.checkCollisionCircles(
    Vector2(x: pos1X, y: pos1Y),
    radius1,
    Vector2(x: pos2X, y: pos2Y),
    radius2
  )
```

```
proc getDisplayResolution*: (int, int) {.exportpy: "get_display_resolution".} =
  when defined(linux):
    let
      disp = XOpenDisplay(nil)
      scrn = DefaultScreenOfDisplay(disp)
    defer: discard XCloseDisplay(disp)
    (scrn.width.int, scrn.height.int)
  elif defined(windows):
    (GetSystemMetrics(SM_CXSCREEN).int, GetSystemMetrics(SM_CYSCREEN).int)

proc compareColorPCT*(color1, color2: rl.Color): float {.exportpy: "compare_color_pct".} =
  let
    r = abs(color1.r.int - color2.r.int).float / 255
    g = abs(color1.g.int - color2.g.int).float / 255
    b = abs(color1.b.int - color2.b.int).float / 255
  result = 100 - ((r + g + b) / 3 * 100)

proc getMonitorCount: int {.exportpy: "get_monitor_count".} =
  rl.getMonitorCount()

proc getMonitorName(monitor: cint = 0): string {.exportpy: "get_monitor_name".} =
  $rl.getMonitorName(monitor)

proc getMonitorRefreshRate(monitor: cint = 0): int {.exportpy: "get_monitor_refresh_rate".} =
  rl.getMonitorRefreshRate(monitor)

proc getWindowTitle(processId: int): string {.exportpy: "get_window_title".} =
  when defined(windows):
    var winHandle = GetWindow(GetTopWindow(0), GW_HWNDNEXT)
    while winHandle != FALSE:
      if IsWindowVisible(winHandle):
        var winProcessId: DWORD
        GetWindowThreadProcessId(winHandle, winProcessId.addr)
        if winProcessId == processId:
          let winLength = GetWindowTextLength(winHandle)
          var winTitle = newWString(winLength)
          GetWindowText(winHandle, winTitle, winLength + 1)
          return nullTerminated($$winTitle)
      winHandle = GetNextWindow(winHandle, GW_HWNDNEXT)
  elif defined(linux):
    let
      p = startProcess("wmctrl", "", ["-l", "-p"], options={poUsePath, poStdErrToStdOut})
      (lines, exitCode) = p.readLines()

    if exitCode == 0:
      for l in lines:
        let (r, _, _, pid, _, title) = l.scanTuple("$h $s$i $s$i $s$+ $s$+")
        if r and pid != 0:
          if pid == processId:
            return title
      raise newException(Exception, "No Window found. PID: " & $processId)
    else:
      raise newException(Exception, "wmctrl failed (installed ’wmctrl’?)")
```

```
File: vec.nim
--------------

import
  nimpy, nimraylib_now, nimraylib_now/raymath as rm

pyExportModule("pyMeow")

proc vec2(x, y: float = 0): Vector2 {.exportpy: "vec2".} =
  Vector2(x: x, y: y)
proc vec3(x, y, z: float = 0): Vector3 {.exportpy: "vec3".} =
  Vector3(x: x, y: y, z: z)

proc vec2Add(v1, v2: Vector2): Vector2 {.exportpy: "vec2_add".} =
  rm.add(v1, v2)
proc vec2AddValue(v: Vector2, value: float): Vector2 {.exportpy: "vec2_add_value".} =
  rm.addValue(v, value)
proc vec3Add(v1, v2: Vector3): Vector3 {.exportpy: "vec3_add".} =
  rm.add(v1, v2)
proc vec3AddValue(v: Vector3, value: float): Vector3 {.exportpy: "vec3_add_value".} =
  rm.addValue(v, value)

proc vec2Subtract(v1, v2: Vector2): Vector2 {.exportpy: "vec2_subtract".} =
  rm.subtract(v1, v2)
proc vec2SubtractValue(v: Vector2, value: float): Vector2 {.exportpy: "vec2_subtract_value".} =
  rm.subtractValue(v, value)
proc vec3Subtract(v1, v2: Vector3): Vector3 {.exportpy: "vec3_subtract".} =
  rm.subtract(v1, v2)
proc vec3SubtractValue(v: Vector3, value: float): Vector3 {.exportpy: "vec3_subtract_value".} =
  rm.subtractValue(v, value)

proc vec2Multiply(v1, v2: Vector2): Vector2 {.exportpy: "vec2_multiply".} =
  rm.multiply(v1, v2)
proc vec2MultiplyValue(v: Vector2, value: float): Vector2 {.exportpy: "vec2_multiply_value".} =
  rm.scale(v, value)
proc vec3Multiply(v1, v2: Vector3): Vector3 {.exportpy: "vec3_multiply".} =
  rm.multiply(v1, v2)
proc vec3MultiplyValue(v: Vector3, value: float): Vector3 {.exportpy: "vec3_multiply_value".} =
  rm.scale(v, value)

proc vec2Divide(v1, v2: Vector2): Vector2 {.exportpy: "vec2_divide".} =
  rm.divide(v1, v2)
proc vec3Divide(v1, v2: Vector3): Vector3 {.exportpy: "vec3_divide".} =
  rm.divide(v1, v2)

proc vec2Length(v: Vector2): float {.exportpy: "vec2_length".} =
  rm.length(v)
proc vec3Length(v: Vector3): float {.exportpy: "vec3_length".} =
  rm.length(v)

proc vec2LengthSqr(v: Vector2): float {.exportpy: "vec2_length_sqr".} =
  rm.lengthSqr(v)
proc vec3LengthSqr(v: Vector3): float {.exportpy: "vec3_length_sqr".} =
  rm.lengthSqr(v)
```

```nim
proc vec2Distance(v1, v2: Vector2): float {.exportpy: "vec2_distance".} =
  rm.distance(v1, v2)
proc vec3Distance(v1, v2: Vector3): float {.exportpy: "vec3_distance".} =
  rm.distance(v1, v2)

proc vec2Closest(v: Vector2, vectorList: varargs[Vector2]): Vector2 {.exportpy: "vec2_closest".} =
  var closestValue = float32.high
  for vec in vectorList:
    let dist = v.vec2Distance(vec)
    if dist < closestValue:
      result = v
      closestValue = dist
proc vec3Closest(v: Vector3, vectorList: varargs[Vector3]): Vector3 {.exportpy: "vec3_closest".} =
  var closestValue = float32.high
  for vec in vectorList:
    let dist = v.vec3Distance(vec)
    if dist < closestValue:
      result = v
      closestValue = dist
```