How to use `SDL_bgi` in Python programs

Please make sure that you installed the `SDL_bgi` binaries for your platform, as described in `INSTALL_Python.md`, before proceeding.

Implementation Details

`SDL_bgi` is written in C, and its Python bindings are implemented via the ctypes library. In general:

- functions in Python have the same name as their C counterparts;
- constants and enums are implemented as variables;
- standard C types (`int`, `float`, `char *`, etc.) are mapped to Python types;
- structs are available as `ctypes` classes that have the same name and field names. For example:

```
C struct                      Python class
--------                      ------------
struct arccoordstype {        class arccoordstype (Structure):
  int x;                          _fields_ = [ ("x", c_int),
  int y;                                       ("y", c_int),
  int xstart;                                  ("xstart", c_int)
  int ystart;                                  ("ystart", c_int)
  int xend;                                    ("xend", c_int),
  int yend;                                    ("yend", c_int) ]
};
```

There are minor differences explained below.

Syntax differences

`ctypes` implements new types that are mapped to equivalent Python types; for example, `c_int` is equivalent to `int`. Please refer to `ctypes`' Reference.

2D arrays can be implemented via Numpy; please see for example `demo/life.py` or `demo/buffers_numpy.py`. Strictly speaking, Numpy is not required; but working with arrays without it is a pain.

Memory buffers, used for example by `getimage()` or `getbuffer()`, are implemented using function `create_string_buffer()`.

The `byref()` function can be used to pass variables by reference, as in the following functions:

```
# void detectgraph (int *graphdriver, int *graphmode);
graphdriver, graphmode = c_int (), c_int ()
detectgraph (byref (graphdriver), byref (graphmode))
print ("graphdriver, graphmode: ", graphdriver.value, graphmode.value)
```

```
# void getarccoords (struct arccoordstype *arccoords);
ac = arccoordstype ()
getarccoords (byref (ac))
print ("x, y, xstart, ystart, xend, yend: ", ac.x, ac.y,
        ac.xstart, ac.ystart, ac.xend, ac.yend)

# void getaspectratio (int *xasp, int *yasp);
xasp, yasp = c_int (), c_int ()
getaspectratio (byref (xasp), byref (yasp))
print ("xasp, yasp: ", xasp.value, yasp.value)

# void getfillsettings (struct fillsettingstype *fillinfo);
fillinfo = fillsettingstype ()
getfillsettings (byref (fillinfo))
print ("pattern, color: ", fillinfo.pattern, fillinfo.color)

# void getimage ()
isize = imagesize (0, 0, len, 16)
image = create_string_buffer (isize)
getimage (0, 0, len, 16, image)

# void getlinesettings (struct linesettingstype *lineinfo);
lineinfo = linesettingstype ()
getlinesettings (byref (lineinfo))
print ("linestyle, thickness: ", ls.linestyle, ls.thickness)

# void getmoderange (int graphdriver, int *lomode, int *himode);
lomode, himode = c_int (), c_int ()
getmoderange (0, byref (lomode), byref (himode))
print ("lomode, himode: ", lomode.value, lomode.value)

# void getmouseclick (int btn, int *x, int *y);
kind, x, y = c_int (), c_int (), c_int ()
getmouseclick (kind, byref (x), byref (y))
print ("mouse x, mouse y: ", x.value, y.value)

# void getscreensize (int x, int y);
x, y = c_int (), c_int ()
getscreensize (byref (x), byref (y))
print ("size x, size y: ", x, y)
```

Pythonic Syntax

The following functions also provide a more Pytonic syntax that only uses standard Python types:

```
# void detectgraph (int *graphdriver, int *graphmode);
graphdriver, graphmode = detectgraph ()
print ("graphdriver, graphmode: ", graphdriver, graphmode);
```

```
# void getarccoords (struct arccoordstype *arccoords);
ac = arccoordstype ()
ac = getarccoords ()
print ("x, y, xstart, ystart, xend, yend: ", ac.x, ac.y,
        ac.xstart, ac.ystart, ac.xend, ac.yend)

# void getaspectratio (int *xasp, int *yasp);
xasp, yasp = getaspectratio ()
print ("xasp, yasp: ", xasp, yasp)

# void getfillsettings (struct fillsettingstype *fillinfo);
fs = fillsettingstype ()
fs = getfillsettings ()
print ("pattern, color: ", fs.pattern, fs.color)

# void getlinesettings (struct linesettingstype *lineinfo);
ls = linesettingstype ()
ls = getlinesettings ()
print ("linestyle, thickness: ", ls.linestyle, ls.thickness)

# void getmoderange (int graphdriver, int *lomode, int *himode);
lomode, himode = getmoderange ()
print ("lomode, himode: ", lomode, lomode)

# void getmouseclick (int btn, int *x, int *y);
x, y = getmouseclick (WM_LBUTTONDOWN)
print ("mouse x, mouse y: ", x, y)

# void getscreensize (int x, int y);
x, y = getscreensize ()
print ("size x, size y: ", x, y)

# void initgraph (int *graphdriver, int *graphmode, char *pathtodriver)
initgraph ()
```

Helper Functions

The following functions can be useful:

`list2vec (list)`: converts a Python list of integers to a vector; used for example by `drawpoly()`

`vec2buf (vector)`: returns a string buffer that contains the values stored in `vector`. This is a 1-dimensional array that can be obtained from a Numpy 2D array 'matrix' with `reshape (matrix, -1)`.

`sizeofint ()`: equivalent to C `sizeof (int)`. Please note that this is not the same as `sys.getsizeof()`!

Missing Features

SDL2-based variables `bgi_window`, `bgi_renderer`, `bgi_texture`, and function `copysurface()` are not available.

Speeding Things Up

Python is an interpreted language, and its performance is quite poor if compared to compiled code. The PyPy interpreter should make Python code run faster, but `SDL_bgi` programs run much slower with PyPy than with CPython. Another Python implementation, Pyston, actually runs `SDL_bgi` programs definitely faster than CPython.

To give your programs a real boost, I strongly suggest that module Numba be used. Numba is a high performance Python JIT compiler that can translate a large subset of Python and NumPy code into fast machine code. It uses simple function decorators; please have a look at `demo/mandelbrot.py` to see how it works.

Making Standalone Binaries

To deploy a Python program as a standalone executable file, you may use PyInstaller or Nuitka.

Pyinstaller

Run it as in the following example:

```
test$ pyinstaller -F fern.py
121 INFO: PyInstaller: 5.4.1
121 INFO: Python: 3.10.4
...
7373 INFO: Building EXE from EXE-00.toc completed successfully.
test$ _
```

The resulting executable will be created in directory `dist/`.

Nuitka

TODO: module 'zstandard', depends22_x64

Run it as in the following example:

```
test$ nuitka3 --onefile --remove-output fern.py
Nuitka-Options:INFO: Used command line options: --onefile \
  --remove-output fern.py
Nuitka:INFO: Starting Python compilation with Nuitka '1.1.3' \
  on Python '3.10' commercial grade 'not installed'.
...
Nuitka:INFO: Successfully created 'fern.bin'.
test$ _
```

When run on Windows, you get **`fern.exe`** and **`fern.cmd`**, which is a batch file that sets up the proper runtime environment for the executable. Run **`fern.cmd`** to start the program; on MSYS2, use:

```
test$ start fern.cmd
```

On my GNU/Linux Mint 20.2 box, Nuitka creates a much smaller executable than Pyinstaller does.