Using `SDL_bgi`

If you're familiar with BGI, the Borland Graphics Interface (`GRAPHICS.H`) provided by Turbo C or Borland C++, then using `SDL_bgi` along with `gcc` or `clang` will be straighforward. If you don't even know what BGI was, don't worry: you will find `SDL_bgi` an easy and fun way to do graphics programming in C.

Although `SDL_bgi` is almost perfectly compatible with the original `GRAPHICS.H` by Borland, a few minor differences have been introduced. The original BGI library mainly targeted the VGA video display controller, which was quite limited and provided a maximum of 16 colours. `SDL_bgi` uses modern graphics capabilities provided by `SDL2`, while retaining backwards compatibility as much as possible.

Compiling Programs

To compile a C or C++ program on GNU/Linux, macOS or Raspios you can use the `gcc`,`clang`, or `tcc` compilers. With `gcc` or `clang`:

```
$ gcc -o program program.c -lSDL_bgi -lSDL2
```

With `tcc`:

```
$ tcc -o program program.c -w -D SDL_DISABLE_IMMINTRIN_H \
    -I /usr/include/SDL2 -lSDL_bgi -lSDL2
```

You may get compilation errors affecting `libpulsecommon`; they can be safely ignored.

`tcc` can also be invoked from scripts. You just need to add the following line (it can't be split with \) at the start of your C source (GNU/Linux):

```
#!/usr/bin/tcc -run -w -D SDL_DISABLE_IMMINTRIN_H -I /usr/include/SDL2 -lSDL_bgi -lSDL2
```

but for better compatibility, please have a look at the `test/tccrun` script.

To compile a program in MSYS2 + mingw-w64:

```
$ gcc -o program.exe program.c -lmingw32 -L/mingw64/bin \
    -lSDL_bgi -lSDL2main -lSDL2 # -mwindows
```

The `-mwindows` switch creates a window-only program, i.e. a terminal is not started. *Beware:* functions provided by `stdio.h` will not work if you don't start a terminal; your program will have to rely on mouse input only.

Code::Blocks users should read the file `howto_CodeBlocks.md`.

Dev-C++ users should read the file `howto_Dev-Cpp.md`.

To compile a program to WebAsssembly using `emcc`:

```
$ emcc -o program.html program.c \
    -std=gnu99 -O2 -Wall -lSDL_bgi -lm \
    -s USE_SDL=2             `# uses SDL2 module` \
    -s ALLOW_MEMORY_GROWTH=1 `# needed for the argb palette` \
    -s ASYNCIFY              `# implement loops` \
    -s SINGLE_FILE           `# standalone html files`
```

where `SDL_bgi.bc` is the precompiled `SDL_bgi` wasm module. The resulting `program.html` can be loaded and run in web browsers, without the need of starting a local web server:

```
$ firefox program.html
```

Compilation details

Windows users *must* declare the `main()` function as:

```
int main (int argc, char *argv[])
```

even if `argc` and `argv` are not going to be used. Your program will not compile if you use a different `main()` definition (i.e. `int main (void)`), because of conflict with the `WinMain()` definition. It's an SDL2 feature; please consult https://wiki.libsdl.org/FAQWindows for details.

Most old programs that use the original BGI library should compile unmodified. For instance,

```
int gd = DETECT, gm;
initgraph (&gd, &gm, "");
```

opens an 800x600 window, mimicking SVGA graphics. If the environment variable `SDL_BGI_RES` is set to `VGA`, window resolution will be 640x480.

Minimal `dos.h` and `conio.h` are provided in the `test/` directory; they're good enough to compile the original `bgidemo.c`.

Please note that non-BGI functions are *not* implemented. If you need `conio.h` for GNU/Linux, please have a look at one of these:

- https://github.com/nowres/conio-for-linux
- https://gitlab.com/marcodiego/conio

To specify the window size, you can use the new SDL driver:

```
gd = SDL;
gm = <mode>;
```

where `<mode>` can be one of the following:

`CGA`	`320x200`
`SDL_320x200`	`320x200`
`EGA`	`640x350`
`SDL_640x480`	`640x350`
`VGA`	`640x480`
`SDL_640x480`	`640x480`

`SVGA`	`800x600`
`SDL_800x600`	`800x600`
`SDL_1024x768`	`1024x768`
`SDL_1152x900`	`1152x900`
`SDL_1280x1024`	`1280x1024`
`SDL_1366x768`	`1366x768`
`SDL_FULLSCREEN`	`fullscreen`

More less common resolutions are listed in `SDL_bgi.h`. You may want to use `initwindow(int width, int height)` instead.

`SDL_bgi.h` defines the `_SDL_BGI_H` constant. You can check for its presence and write programs that employ `SDL_bgi` extensions; please have a look at the test program `fern.c`.

Screen Refresh

The only real difference between the original BGI and `SDL_bgi` is the way the screen is refreshed. In BGI, every graphics element drawn on screen was immediately displayed. This was a terribly inefficient way of drawing stuff: the screen should be refreshed only when the drawing is done. For example, in SDL2 this action is performed by `SDL_RenderPresent()`.

You can choose whether to open the graphics system using `initgraph()`, which toggles BGI compatibility on and forces a screen refresh after every graphics command, or using `initwindow()` that leaves you in charge of refreshing the screen when needed, using the new function `refresh()`.

The first method is fully compatible with the original BGI, but it also painfully slow. An experimental feature is 'auto mode': if the environment variable `SDL_BGI_RATE` is set to `auto`, screen refresh is automatically performed; this is *much* faster than the default. This variable may also contain a refresh rate; e.g. 60. Unfortunately, auto mode may not work on some NVIDIA graphic cards.

As a tradeoff between performance and speed, a screen refresh is also performed by `getch()`, `kbhit()`, and `delay()`. Functions `sdlbgifast(void)`, `sdlbgislow(void)`, and `sdlbgiauto(void)` are also available. They trigger fast, slow, and auto mode, respectively.

Documentation and sample BGI programs are available at this address:

https://winbgim.codecutter.org/V6_0/doc/

Nearly all programs can be compiled with `SDL_bgi`.

The original Borland Turbo C 2.0 manual is also available here:

https://archive.org/details/bitsavers_borlandturReferenceGuide1988_19310204.

Avoid Slow Programs

This is possibly the slowest `SDL_bgi` code one could write:

```
while (! event ()) {
  putpixel (random(x), random(y), random(col));
  refresh ();
}
```

This code, which plots pixels until an event occurs (mouse click or key press), is extremely inefficient. First of all, calling `event()` is relatively expensive; secondly, refreshing the screen after plotting a single pixel is insane. You should write code like this:

```
counter = 0;
stop = 0;
while (! stop) {
  putpixel (random(x), random(y), random(col));
  if (1000 == ++counter) {
    if (event())
      stop = 1;
    refresh ();
    counter = 0;
  }
}
```

In general, you should use `kbhit()`, `mouseclick()` and `event()` sparingly, because they're slow.

Differences

Please see the accompanying document `compatibility`.

Colours

`SDL_bgi` has two colour palettes: one for compatibility with old BGI, the other for ARGB colours. Colour depth is always 32 bit; `SDL_bgi` has not been tested on lesser colour depths.

The default BGI palette includes 16 named colours (`BLACK`...`WHITE`); standard BGI functions, like `setcolor()` or `setbkcolor()`, use this palette. By default, colours in the default palette don't have the same RGB values as the original BGI colours; the palette is brighter and (hopefully) better looking. The original RGB values will be used if the environment variable `SDL_BGI_PALETTE` is set to `BGI`.

An extended ARGB palette of `PALETTE_SIZE` additional colours can be used by functions like `setrgbcolor()` or `setbkrgbcolor()` described below; please note the `rgb` in the function names. `PALETTE_SIZE` is initialised to 4096, but it can be increased using `resizepalette()`.

Please see the example programs in the `test/` directory.

Fonts

Fonts that are almost pixel-perfect compatible with the original Borland Turbo C++ 3.0 `.CHR` fonts are built in. Characters in the ASCII range 32 - 254 are available. Loading `.CHR` fonts from disk is also possible.

`.CHR` fonts support was added by Marco Diego Aurélio Mesquita.

Additions

Some functions and macros have been added to add functionality and provide compatibility with other BGI implementations (namely, Xbgi and WinBGIm).

Further, the following variables (declared in `SDL_bgi.h`) are accessible to the programmer:

```
SDL_Window   *bgi_window;
SDL_Renderer *bgi_renderer;
SDL_Texture  *bgi_texture;
```

and can be used by native SDL2 functions; see example in `test/sdlbgidemo.c`.

Screen and Windows Functions

- `void initwindow(int width, int height)` lets you open a window specifying its size. If either `width` or `height` is 0, then `SDL_FULLSCREEN` will be used.
- `void detectgraph(int *gd, int *gm)` returns `SDL`, `SDL_FULLSCREEN`.
- `void setwinoptions(char *title, int x, int y, Uint32 flags)` lets you specify the window title (default is `SDL_bgi`), window position, and some SDL2 window flags OR'ed together. In particular, you can get non-native fullscreen resolution with:

```
setwinoptions ("", -1, -1, SDL_WINDOW_FULLSCREEN);
initwindow (800, 600);
```

- `getscreensize(int *x, int *y)` reports the screen width and height in `x` and `y`. You can also use related functions `getmaxheight()` and `getmaxwidth()`.
- `void sdlbgifast(void)` triggers "fast mode" even if the graphics system was opened with `initgraph()`. Calling `refresh()` is needed to display graphics.

- `void sdlbgislow(void)` triggers “slow mode” even if the graphics system was opened with `initwindow()`. Calling `refresh()` is not needed.
- `void sdlbgiauto(void)` triggers automatic screen refresh. **Note**: it may not work on some graphics cards.

Multiple Windows Functions

Subsequent calls to `initgraph()` or `initwindow()` make it possible to open several windows; only one of them is active (i.e. being drawn on) at any given time, regardless of mouse focus.

Functions `setvisualpage()` and `setactivepage()` only work properly in single window mode.

- `int getcurrentwindow()` returns the active window identifier.
- `void setcurrentwindow(int id)` sets the current active window. `id` is an integer identifier, as returned by `getcurrentwindow()`.
- `void closewindow(int id)` closes a window of given id.

Colour Functions

- `void setrgbpalette(int color, int r, int g, int b)` sets colours in an additional palette containing ARGB colours (up to `PALETTE_SIZE`). See example in `test/mandelbrot.c`.
- `void setrgbcolor(int col)` and `void setbkrgbcolor(int col)` are the ARGB equivalent of `setcolor(int col)` and `setbkcolor(int col)`. `col` is an allocated colour entry in the ARGB palette.
- `COLOR(int r, int g, int b)` can be used as an argument whenever a colour value is expected (e.g. `setcolor()` and other functions). It’s an alternative to `setrgbcolor(int col)` and `setbkrgbcolor(int col)`. Allocating colours with `setrgbpalette()` and using `setrgbcolor()` is much faster, though.
- `COLOR32(Uint32 color)` works like `COLOR()`, but accepts a colour argument as an ARGB Uint32.
- `RGBPALETTE(int n)` works like `COLOR()`, but it sets the n-th colour in the ARGB palette.
- `colorRGB(int r, int g, int b)` can be used to compose a 32 bit colour. This macro is typically used to set values in memory buffers (see below).
- `IS_BGI_COLOR(int c)` and `IS_RGB_COLOR(int c)` return 1 if the current colour is standard BGI or ARGB, respectively. The argument is actually redundant; in fact, a colour entry in the range 0-15 may belong to both palettes. Whether the standard palette or the ARGB palette is the current one is set by standard BGI or ARGB functions.

- `ALPHA_VALUE(int c)`, `RED_VALUE(int c)`, `GREEN_VALUE(int c)`, and `BLUE_VALUE(int c)` return the A, R, G, B component of an ARGB colour in the extended palette.
- `getrgbpalette(struct rgbpalettetype* palette)` and `setallrgbpalette (struct rgbpalettetype *palette)` work like their BGI counterpart, but use the ARGB palette. The `colors` member of `struct rgbpalettetype` variables must be initialised; please see `test/rgbpalette.c`.
- `setalpha(int col, Uint8 alpha)` sets the alpha component of colour 'col'.
- `setblendmode(int blendmode)` sets the blending mode for screen refresh (`SDL_BLENDMODE_NONE` or `SDL_BLENDMODE_BLEND`).

Buffer Functions

- `getbuffer (Uint32 *buffer)` and `putbuffer (Uint32 *buffer)` copy the current window contents to a memory buffer, and the reverse. Using `getbuffer()` and `putbuffer()` is faster than direct pixel manipulation, as shown by `test/psychedelia.c`
- `getlinebuffer (int y, Uint32 *linebuffer)` and `putlinebuffer (int y, Uint32 *linebuffer)` work like `getbuffer()` and `putbuffer()`, but on a single line of pixels at `y` coordinate.

Mouse Functions

- `int mouseclick(void)` returns the code of the mouse button that is being clicked, 0 otherwise. Mouse buttons and movement constants are defined in `SDL_bgi.h`:

```
WM_LBUTTONDOWN
WM_MBUTTONDOWN
WM_RBUTTONDOWN
WM_WHEEL
WM_WHEELUP
WM_WHEELDOWN
WM_MOUSEMOVE
```

- `int isdoubleclick(void)` returns 1 if the last mouse click was a double click.
- `int mousex(void)` and `int mousey(void)` return the mouse coordinates of the last click.
- `int ismouseclick(int btn)` returns 1 if the `btn` mouse button was clicked.

- `void getmouseclick(int kind, int *x, int *y)` sets the x, y coordinates of the last button click expected by `ismouseclick()`.
- `void getleftclick(void)`, `void getmiddleclick(void)`, and `void getrightclick(void)` wait for the left, middle, and right mouse button to be clicked and released.
- `int getclick(void)` waits for a mouse click and returns the button that was clicked.

Miscellaneous Functions

- `showerrorbox(const char *message)` and `showinfobox(const char *message)` open a window message box with the specified message.
- `void _putpixel(int x, int y)` is equivalent to `putpixel(int x, int y, int col)`, but uses the current drawing colour and the pixel is not refreshed in slow mode.
- `random(range)` is defined as macro: `rand()%range`
- `int getch()` waits for a key and returns its ASCII code. Special keys and the `SDL_QUIT` event are also reported; please see `SDL_bgi.h`.
- `void delay(msec)` waits for `msec` milliseconds.
- `int getevent(void)` waits for a keypress or mouse click, and returns the code of the key or mouse button. It also catches and returns `SDL_QUIT` events.
- `int event(void)` is a non-blocking version of `getevent()`.
- `int eventtype(void)` returns the type of the last event.
- `void readimagefile(char *filename, int x1, int y1, int x2, int y2)` reads a `.bmp` file and displays it immediately (i.e. no refresh needed).
- `void writeimagefile(char *filename, int left, int top, int right, int bottom)` writes a `.bmp` file from the screen rectangle defined by (left,top–right,bottom).
- `void kbhit(void)` returns 1 when a key is pressed, excluding Shift, Alt, etc.
- `int lastkey(void)` returns the last key that was detected by `kbhit()`.
- `void xkbhit(void)` returns 1 when any key is pressed, including Shift, Alt, etc.

The Real Thing

You may want to try the online Borland Turbo C 2.01 emulator at the Internet Archive:

https://archive.org/details/msdos_borland_turbo_c_2.01.

The `bgidemo.c` program demonstrates the capabilities of the BGI library. You can download it and compile it using `SDL_bgi`; in Windows, you will have to change its `main()` definition.

Bugs & Issues

Please see the accompanying document `BUGS`.

Probably, this documentation is not 100% accurate. Your feedback is more than welcome.