Using `SDL_bgi`

Although `SDL_bgi` is almost perfectly compatible with the original BGI library, a few minor differences were introduced to take advantage of modern SDL graphics. You don't want a slow library!

Compiling programs

To compile a program (GNU/Linux, OS X):

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

To compile a program (MSYS2 + mingw-w64):

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

The `-mwindows` 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`.

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

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

even if `argc` and `argv` will not 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. 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, "");
```

will open an 800x600 window, mimicking SVGA graphics. Very basic `dos.h` and `conio.h` are provided in the `test/` directory; they're good enough to compile the original `bgidemo.c` (not provided: it's not FOSS) unmodified. Please note that non-BGI functions such a `gotoxy()` are *not* implemented.

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`

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 update

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 second method is *much* faster and is preferable.

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

Documentation and sample BGI programs are available at this address: http://www.cs.colorado.edu/~main/cs1300/doc/bgi/ Nearly all programs can be compiled with `SDL_bgi`.

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 something 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

- The following functions may be called but do nothing:

```
_graphfreemem       - unneeded
_graphgetmem        - unneeded
installuserdriver   - it makes no sense in SDL
installuserfont     - should I implement it for SDL_ttf?
registerbgidriver   - it makes no sense in SDL
registerbgifont     - it makes no sense in SDL
setgraphbufsize     - unneeded
```

- `setpalette()` only affects future drawing. That is, you can't get a "rotating palette animation" as in Turbo C.
- an 8x8 bitmap font is included, and it's the only one font. Changes to other BGI fonts (e.g. `TRIPLEX_FONT`, and others) have no effect: consider using `SDL_ttf`!

Colours

The default BGI palette includes 16 named colours (`BLACK`...`WHITE`); standard BGI functions use this palette.

An extended ARGB palette of `PALETTE_SIZE` additional colours can be created and accessed using functions described below. Please see the example programs in the `test/` directory.

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.

- `void initwindow(int width, int height)` lets you open a window specifying its size.
- `void detectgraph(int *gd, int *gm)` returns `SDL`, `SDL_FULLSCREEN`.
- `void setrgbpalette(int color, int r, int g, int b)` sets an additional palette containing RGB colours (up to `MAXRGBCOLORS` + 1). See example in `test/mandelbrot.c`.
- `void setrgbcolor(int col)` and `void setbkrgbcolor(int col)` are the RGB equivalent of `setcolor(int col)` and `setbkcolor(int col)`. `col` is an allocated colour entry in the RGB 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.
- `IS_BGI_COLOR(int c)` and `IS_RGB_COLOR(int c)` return 1 if the current colour is standard BGI or RGB, respectively. The argument is actually redundant.
- `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 RGB colour in the extended palette.

- `setalpha(int col, Uint8 alpha)` sets the alpha component of colour 'col'.
- `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 mouseclick(void)` returns the code of the mouse button that was clicked, or 0 if none was clicked. 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 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()`.
- `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 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 setwinoptions(char *title, int x, int y, Uint32 flags)` lets you specify the window title (default is `SDL_bgi`), window position, and SDL2 window flags OR'ed together.
- `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 xkbhit(void)` returns 1 when any key is pressed, including Shift, Alt, etc.

Multiple Windows

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

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

- Optionally, use `setwinoptions(char *title, int x, int y, Uint32 flags)` as explained above;
- `int getcurrentwindow()` returns the current window identifier;
- `void setcurrentwindow(int id)` sets the current window. `id` is an integer identifier, as returned by `getcurrentwindow()`;
- `void closewindow(int id)` closes a window of given id.

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()` declaration.

Bugs & Issues

Drawing in BGI compatibility (slow) mode is much slower than it should, since `SDL_UpdateTexture()` doesn't work as expected: instead of refreshing an `SDL_Rect` correctly, it only works on entire textures. It looks like it's an SDL2 bug.

Console routines such as `getch()` may hang in Mingw. As far as I can tell, it's a bug in Mingw console handling.

Colours don't have the same RGB values as the original BGI colours. But they look better (IMHO).

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