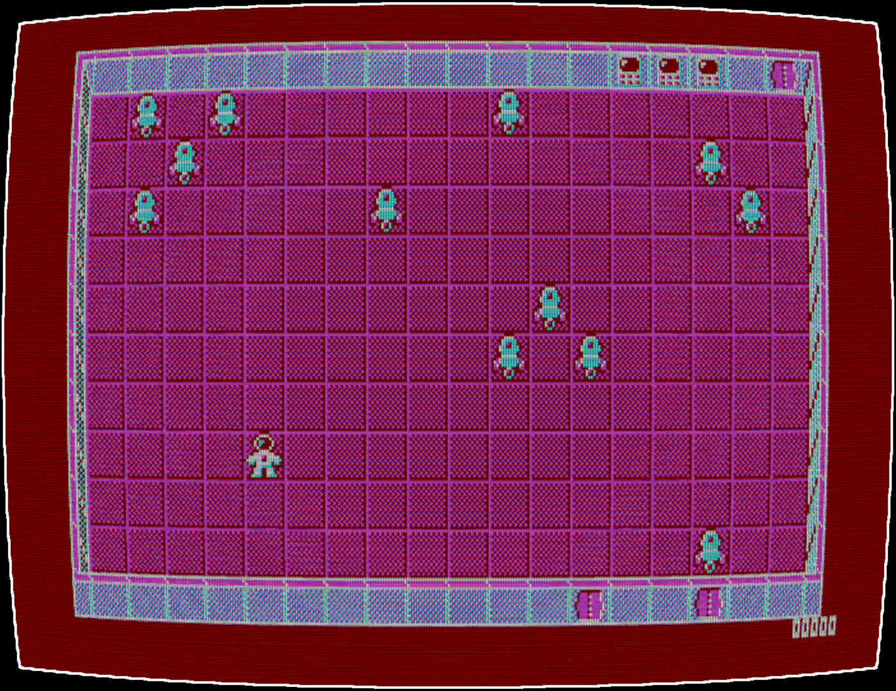
CGALIB 2



CYNINGSTAN

Contents

Chapter 1

Introduction

CGALIB provides a fast graphics library for 320x200 graphics on a CGA card, along with sound and keyboard routines for a typical XT-class PC, for use with OpenWatcom C. Features provided by the library include:

- Full use of all CGA palettes in 320x200 mode.
- Emulation of those palettes on EGA and VGA.
- Emulation of 320x200 mode in 640x200 for monochrome devices.
- Variable sized text with a selection of 4x8 fonts.
- Drawing directly to the screen or to an off-screen bitmap.
- Keyboard reading for an 83/84 key keyboard.
- Sound effects and music for the PC speaker.

CGALIB is intended primarily for turn-based games like adventures, puzzles and strategy games. For speed, it does not offer pixel perfect positioning but divides the screen into a grid of 80 by 200 cells. Bitmaps need to be a multiple of 4 pixels by 1 pixel to fit into this grid. Coordinates are by the pixel, but those coordinates need to be divisible by these values.

Action games are still possible, and can be achieved by sprite rotation, but *CGALIB* might not be the most appropriate library for such projects.

Chapter 2

Licence

This library, its associated programs and utilities, and its documentation have been released into the public domain by its author Damian Gareth Walker.

Chapter 3

Binary Package Contents

The CGALIB binary package for Watcom C contains the following directory structure and files:

- `CGALIB\` is the main directory
  - `DEMO.EXE` is the demonstration program
  - `MAKEBIT.EXE` is the bitmap maker utility
  - `MAKEFONT.EXE` is the font maker utility
  - `MAKETUNE.EXE` is the music maker utility
  - `MAKEFX.EXE` is the sound effect maker utility
  - `CGALIB.LIB` is the large model library
  - `BIT\` contains some supplied bitmaps
    - `DEMO.BIT` are the bitmaps for the demo game
    - `MAKEBIT.BIT` are the bitmaps for the bitmap maker
    - `MAKEFONT.BIT` are the bitmaps for the font maker
    - `MAKEFX.BIT` are the bitmaps for the sound effect maker
  - `FNT\` contains the supplied fonts

    * `PAST.FNT` is a medieval style font
    * `PRESENT.FNT` is a regular font
    * `FUTURE.FNT` is a sci-fi inspired font
    * `MAKETUNE.FNT` is the font used by the music maker utility. It is a modified version of `PRESENT.FNT`, with the dollar sign replaced by the musical flat sign
- `INC\` contains the C headers

Chapter 4

Source Package Contents

The CGALIB source package for Watcom C contains the following directory structure and files:

- `cgalib\` is the destination directory for binaries and data
- `inc\` is the include directory
  - `bitmap.h` is the header file for the bitmap module
  - `cgalib.h` is the main header file
  - `cga_x11.h` is the X11 header file
  - `effect.h` is the sound effect header
  - `font.h` is the header file for the font module
  - `keyboard.h` is the keyboard header file
  - `screen.h` is the header file for the screen module
  - `screen_u.h` is the header for joint X11/DOS screen functions
  - `speaker.h` is the speaker header
  - `tune.h` is the tune header

- `obj\` is the directory for compiled object files
- `src\` is the source code directory
  - `bitmap.c` is the bitmap module source
  - `cga_x11.c` is the X11 development source
  - `demo.c` is the demonstration program source
  - `effect.c` is the sound effects source
  - `effect_x.c` is the dummy X11 sound effect source
  - `font.c` is the font module source
  - `fxdemo.c` is the sound effects demo source
  - `keyboard.c` is the keyboard source for DOS
  - `keybrd_x.c` is the keyboard source for X11
  - `makebit.c` is the bitmap maker utility source
  - `makefont.c` is the font maker utility source
  - `maketune.c` is the tune maker utility source
  - `playback.c` is the tune player utility source
  - `screen.c` is the screen module source
  - `screen_c.c` is the CGA screen hardware source
  - `screen_x.c` is the X11 screen source
  - `speaker.c` is the speaker handler source
  - `tune.c` is the tune module source
  - `tune_x.c` is the dummy X11 tune module source
- `makefile.gcc` is the makefile to build for X11
- `makefile.gcc` is the makefile to build for DOS

Chapter 5

Building a Project with CGALIB

To use *CGALIB*'s functions in your project, you need to do the following two things. Firstly, you need to include the `cgalib.h` header in your own project's source:

```
#include "cgalib.h"
```

This automatically includes the individual headers for the *CGALIB* modules. You can copy those headers into your project's header directory, but a better idea is to add *CGALIB*'s include folder to your include path on compilation, like this:

```
C:\PROJECT\> wcc project.c -I=\cgalib\inc
```

This assumes that *CGALIB* is installed in a directory called `\cgalib`. When you link your object file into an executable, you need to link it with the `cgalib.lib` file:

```
C:\PROJECT\> wcl project.obj \cgalib\cgalib.lib
```

Chapter 6

Rebuilding CGALIB

You might want to rebuild *CGALIB* from its sources, particularly if your project uses a memory model other than the default Large (ml), if you've made a customised version of it, or of its utilities or its demonstration program. A makefile is provided to simplify this process. If you unpacked the source files into the `\cgasrc` directory, then you can build the project like this:

```
C:\CGASRC\> wmake -f makefile.wcc
```

This builds the LIB file for the large memory model, and stores it in the `\cgasrc\cgalib` directory. It also builds the demonstration program and the utilities against the small model library file, and stores their executables in the `\cgasrc\cgalib` directory. It copies the header files into the `\cgasrc\cgalib\inc` directory, and also copies the supplied font and bitmap files into the relevant directories under `\cgasrc\cgalib`.

The result of this is that the `\cgasrc\cgalib` directory contains all of the files you would expect in the `\cgalib` directory of a binary distribution of *CGALIB*.

If your project uses a memory model other than large, then rebuild instead with a command like the following:

```
C:\CGASRC\> wmake -f makefile.wcc MODEL=ml
```

Valid values are `ms` for small, `mc` for compact, `mm` for medium, `ml` for large, and `mh` for huge.

Chapter 7

Modules

CGALIB has the following modules:

- the Screen module,
- the Bitmap module,
- the Font module,
- the Keyboard module,
- the Speaker modules, comprising:
  - the Effect module,
  - the Tune module.

The *Screen* module handles hardware screen issues like setting the video mode and the palette. It also handles drawing directly to the screen, and extracting bitmaps directly from the screen.

The *Bitmap* module handles the manipulation of bitmaps: their creation and destruction, copying them in full or in part, drawing to them, and loading and storing them in files.

The *Font* module handles the use of fonts. It supports creating, copying and destroying them, and defining their character patterns by copying individual characters to and from bitmaps. It also has some manipulation (changing a font's colour) and allows loading and storing them in files.

The *Keyboard* module handles keyboard input. It supports individual tracking of the up or down state on any key on the keyboard via their scan codes, along with simpler ASCII input.

The *Effect* module handles sound effects using the PC speaker. It supports tone changes up, down or at random, repetition, and silences, making it easy to create noises like laser zaps and white noise.

The *Tune* module handles the playing of music using the PC speaker. It can simulate multiple voices using arpeggios, chords being played by a quick succession of short notes.

Chapter 8

Summary of Functions

The summary of functions below gives the literal declarations of the functions as found in the .h file. Some of them are wrapped in a structure, so that their call syntax resembles calls to object-oriented methods. The call syntax is made more obvious in the following reference chapters, which expand upon each of these functions with examples.

Screen Module

```
Screen *new_Screen (int mode, int show);
struct screen {
  void (*destroy) (Screen *screen);
  int (*write) (Screen *screen, int format,
    FILE *output);
  int (*read) (Screen *screen, int format,
    FILE *input);
  void (*show) (Screen *screen);
  void (*hide) (Screen *screen);
  int (*shown) (Screen *screen);
  void (*update) (Screen *screen);
  void (*palette) (Screen *screen, int foreground,
    int background);
  Bitmap *(*get) (Screen *screen, int x, int y,
    int w, int h);
  void (*put) (Screen *screen, Bitmap *source,
    int x, int y, int mode);
  void (*transfer) (Screen *screen, Bitmap *bitmap,
    int xd, int yd, int xs, int ys, int w, int h,
    int draw);
  void (*box) (Screen *screen, int x, int y, int width,
    int height, unsigned int pattern);
  void (*print) (Screen *screen, int x, int y,
    char *message);
}
```

Bitmap Module

```
Bitmap *new_Bitmap (int width, int height);
Bitmap *read_Bitmap (FILE *input);
struct bitmap {
  void (*destroy) (Bitmap *bitmap);
  Bitmap *(*clone) (Bitmap *bitmap);
  int (*write) (Bitmap *bitmap, FILE *output);
  Bitmap *(*get) (Bitmap *bitmap, int x, int y,
    int w, int h);
  void (*put) (Bitmap *bitmap, Bitmap *source,
    int x, int y, int mode);
  void (*transfer) (Bitmap *bitmap, Bitmap *source,
    int xd, int yd, int xs, int ys, int w, int h,
    int draw);
  void (*box) (Bitmap *bitmap, int x, int y,
    int width, int height, unsigned int pattern);
  void (*print) (Bitmap *bitmap, int x, int y,
    char *message);
};
```

Font Module

```
Font *new_Font (int first, int last, int width, int height);
Font *read_Font (FILE *input, int version);
struct font {
  void (*destroy) (Font *font);
  Font *(*clone) (Font *font);
  int (*write) (Font *font, FILE *output);
  void (*put) (Font *font, Bitmap *bitmap,
    int code);
  Bitmap *(*get) (Font *font, int code);
  void (*recolour) (Font *font, int ink,
    int paper);
};
```

Keyboard Module

```
KeyHandler *new_KeyHandler (void);
struct keyhandler {
  void (*destroy) (void);
  int (*key) (int scancode);
  int (*anykey) (void);
  int (*ascii) (void);
  int (*scancode) (void);
  void (*wait) (void);
};
```

Speaker Module

```
Speaker *get_Speaker (void);
```

```
struct speaker {
  void (*destroy) (void);
  int (*writeint) (int *value, FILE *output);
  int (*readint) (int *value, FILE *input);
};
```

Effect Module

```
Effect *new_Effect (void);
struct effect {
  void (*destroy) (Effect *effect);
  int (*read) (Effect *effect, FILE *input);
  int (*write) (Effect *effect, FILE *output);
  void (*play) (Effect *effect);
};
```

Tune Module

```
Tune *new_Tune (void);
Note *new_Note (int pitch, int duration);
struct tune {
  void (*destroy) (Tune *tune);
  void (*add) (Tune *tune, Note *note);
  int (*read) (Tune *tune, FILE *input);
  int (*write) (Tune *tune, FILE *output);
  void (*play) (Tune *tune, KeyHandler *keys);
};
```

Chapter 9

The Screen Module

Most of the screen functions are accessed through a function pointer in the Screen structure. The exception is the function that creates the structure: `new_Screen`.

X coordinates and widths in the screen functions *must* be divisible by 4. This is for speed: with 2-bit CGA graphics the byte boundaries are on every fourth pixel. Allowing arbitrary X coordinates would necessitate bit rotation and slow the library down.

9.1 new_Screen

Declaration:

```
Screen *new_Screen (int mode, int show);
```

Example:

```
/* initialise a screen in mode 4 (320x200 colour) */
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
/* ... do things with the screen ... */
screen->destroy (screen);
```

This sets the video mode and initialises various internal screen variables. The `mode` parameter sets the initial screen mode. The screen modes are as follows:

- `CGALIB_MEDIUM`: this is the standard 320x200 4-colour mode. Upon initialisation, the palette will be set to light cyan, magenta and white with a black background.
- `CGALIB_MEDIUM_MONO`: this is the 320x200 "monochrome" mode. On a monochrome or composite monitor this will provide four grey scales. On an RGB colour monitor it will have a cyan, red and white palette, with both intensities available. Upon initialisation, the light palette is chosen with a black background.
- `CGALIB_HIGH`: this is the 640x200 high resolution mode. CGALIB will treat this as a 320x200 screen, using 320x200 coordinates and rendering graphics in dithered monochrome. It is useful for portable devices with murky monochrome screens that may not show the more interesting colour choices clearly. Palette changes are ignored in this mode.

The `show` parameter determines whether the screen is shown on creation. It is possible to have multiple screens in existence at once. When using this facility, it might be desirable to create a screen but not immediately show it. The values for `show` are:

- `CGALIB_HIDDEN`: the screen is hidden. If another CGA screen is already shown, that screen remains visible. Otherwise, the screen remains in its default text mode.
- `CGALIB_SHOWN`: the screen is shown immediately. The graphics mode is set to whatever mode was requested (but see Hercules notes below) and the screen is cleared.

If a Hercules graphics card is detected, the `mode` parameter is ignored, and the screen is set to Hercules graphics mode. This emulates the `CGALIB_HIGH` mode, with 640x200 monochrome graphics scaled and centred on the 720x348 Hercules graphics screen.

9.2 screen->destroy

Declaration:

```
struct screen {
  void (*destroy) (Screen *screen);
}
```

Example:

```
/* Simple screen lifecycle */
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
/* ... do things with the screen ... */
screen->destroy (screen);
```

Destroys a screen when it is no longer needed. If the screen was shown at the time of destruction, the screen will be cleared and returned to 80 column text mode. The *screen* parameter is a pointer to the screen structure.

9.3 screen->write

Declaration:

```
struct screen {
  int (*write) (Screen *screen, int format,
    FILE *output);
}
```

Example:

```
/* convert a BSAVE file to a bitmap */
FILE *input, *output;
Screen *screen;
input = fopen ("screen.pic", "rb");
output = fopen ("screen.bit", "wb");
```

```
screen = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
screen->read (screen, CGALIB_SCREEN_FORMAT, input);
screen->write (screen, CGALIB_BITMAP_FORMAT, output);
fclose (output);
fclose (input);
```

Writes the data from a screen to a file, either in BSAVE format, or in the form of a CGALIB bitmap. The *screen* parameter is a pointer to the screen structure. The *format* is one of the following:

- `CGALIB_BITMAP_FILE`: the data was saved in the form of a 320x200 bitmap.
- `CGALIB_SCREEN_FILE`: the data was saved in BSAVE format, consisting of a 7-byte header followed by a byte-for-byte copy of the contents of screen menory. These files can be created and read by packages like PC Paint.

The *output* parameter is the output file handle. The function leaves the opening and closing of files to you, in order that you can combine screens, bitmaps and other data into a single asset file if desired.

9.4 screen->read

Declaration:

```
struct screen {
  int (*read) (Screen *screen, int format,
    FILE *input);
}
```

Example:

```
/* load a BSAVE pic file from PC Paint */
FILE *input;
Screen *screen;
```

```
screen = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
input = fopen("screen.pic", "rb");
screen->read (screen, CGALIB_SCREEN_FILE, input);
fclose (input);
/* ... do things with the screen ... */
screen->destroy (screen);
```

This reads screen data from an already open input file. The *format* is one of the following:

- `CGALIB_BITMAP_FILE`: the data was saved in the form of a 320x200 bitmap.
- `CGALIB_SCREEN_FILE`: the data was saved in BSAVE format, consisting of a 7-byte header followed by a byte-for-byte copy of the contents of screen menory. These files can be created and read by packages like PC Paint.

The *input* parameter is the file handle of a previously opened input file. Using a file handle rather than a filename allows your screen data to be saved on its own or as part of a larger asset file.

9.5 screen->show

Declaration:

```
struct screen {
  void (*show) (Screen *screen);
}
```

Example:

```
Screen *map, *status;
map = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
status = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
/* ... build up both screens ... */
```

```
map->show (map);
/* ... do things on the map interface ... */
map->show (status);
/* ... do things on the status interface ... */
screen->destroy (map);
screen->destroy (status);
```

This makes the specified screen visible. CGALIB allows for multiple screens to be held in memory at once. Note that this can be memory intensive, and is recommended only when multiple user interfaces have to be kept up to date, e.g. a strategic map and a tactical map or unit list in a strategy game.

9.6 screen->hide

Declaration:

```
struct screen {
  void (*hide) (Screen *screen);
}
```

Example:

```
Screen *game;
game = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
/* ... */
game->hide (game);
/* ... */
game->destroy (game);
```

If the specified screen is currently visible, it will be hidden and the text mode screen will be shown instead. If the specified screen is not currently visible, this call will have no effect. This method and screen->show() above might be useful if your program wants to switch between graphics and text modes.

9.7 screen->shown

Declaration:

```
struct screen {
  int (*shown) (Screen *screen);
}
```

Example:

```
Screen *screen;
/* ... */
if (screen->shown (screen)) {
  /* some animation on screen */
}
```

Returns TRUE (or CGALIB_SHOWN) if the specified screen is currently visible. Returns FALSE (or CGALIB_HIDDEN) if the specified screen is currently hidden. This is useful for cases like the example given: maybe you want show a particular animation but not if the screen is not currently shown.

9.8 screen->update

Declaration:

```
struct screen {
  void (*update) (Screen *screen);
}
```

Example:

```
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
screen->updates = 1;
/* ... draw onto screen ... */
screen->update (screen);
```

When graphical operations are applied to a screen, such as put, transfer, box or print, the results do not appear immediately, even if the screen is currently shown. Instead they get written to a back buffer, and only transferred to video memory when the update() method is called. This allows a number of drawing operations to be performed, and the result transferred to the screen all at once, increasing the perceived speed of drawing.

There is an attribute called `screen->updates`, which can be set to the following values:

- `screen->updates = 0;`: update the screen immediately on every drawing operation; no calls to screen->update are necessary.
- `screen->updates = 1;`: screen->update will copy a single potentially large rectangle from the back buffer to video memory, encompassing all drawing operations that have been performed since the last screen->update on this screen.
- `screen->updates = n /* 2 or more */;`: up to *n* drawing operations are remembered individually, and copied one by one to video memory on a call to screen->update. If more than *n* drawing operations are performed, the surplus operations are combined and copied as one potentially large rectangle.

The default when a new screen is created is 0. 1 is recommended if you might have a large number of drawing operations in one area of the screen, for example, building up a map from tiles. 2 is recommended if your screen updates tend to be small and scattered, like a status display in a text based game.

9.9 screen->palette

Declaration:

```
struct screen {
  void (*palette) (Screen *screen,
    int foreground, int background);
}
```

Example:

```
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
screen->palette (screen,
  CGALIB_GREEN_RED_BROWN,
  CGALIB_LIGHT_CYAN);
/* ... */
```

Sets the CGA palette for the screen. The six foreground palettes can be referred to by the number 0..5, or by one of these six enumerated constants:

0. `CGALIB_GREEN_RED_BROWN`
1. `CGALIB_CYAN_MAGENTA_WHITE`
2. `CGALIB_CYAN_RED_WHITE`
3. `CGALIB_LIGHT_GREEN_RED_YELLOW`
4. `CGALIB_LIGHT_CYAN_MAGENTA_WHITE`
5. `CGALIB_LIGHT_CYAN_RED_WHITE`

The background colour can be specified by a number 0..15 from the standard CGA palette, or by one of the following enumerated constants:

0. `CGALIB_BLACK`
1. `CGALIB_BLUE`
2. `CGALIB_GREEN`
3. `CGALIB_CYAN`

4. `CGALIB_RED`
5. `CGALIB_MAGENTA`
6. `CGALIB_BROWN`
7. `CGALIB_WHITE`
8. `CGALIB_GREY`
9. `CGALIB_LIGHT_BLUE`
10. `CGALIB_LIGHT_GREEN`
11. `CGALIB_LIGHT_CYAN`
12. `CGALIB_LIGHT_RED`
13. `CGALIB_LIGHT_MAGENTA`
14. `CGALIB_YELLOW`
15. `CGALIB_WHITE_HIGH`

CGALIB is EGA-aware, and after setting the CGA palette registers it will use EGA BIOS calls to ensure that EGA and later video hardware sets the correct palette. If the screen was initialised to CGALIB_HIGH or Hercules monochrome graphics mode, palette changes will be ignored.

9.10 screen->get

Declaration:

```
struct screen {
  Bitmap *(*get) (Screen *screen, int x, int y,
    int w, int h);
}
```

Example:

```
Screen *screen;
Bitmap *bitmap;
/* ... initialise screen ... */
bitmap = screen->get (screen, 152, 92, 16, 16);
/* ... */
```

Extracts an area of pixels from the specified screen and returns them as a new bitmap. The area extracted will have its top left at x, y, having a width of w pixels and a height of h. The bitmap will be created by `screen->get` and should not be initialised by `new_Bitmap` beforehand.

9.11 screen->put

Declaration:

```
struct screen {
  void (*put) (Screen *screen, Bitmap *source,
    int x, int y, int mode);
}
```

Example:

```
Screen *screen;
Bitmap *bitmap;
int x;
/* ... initialise and draw screen ... */
bitmap = screen->get (screen, 0, 0, 16, 16);
for (x = 0; x < 320; x += 16)
  screen->put (screen, bitmap, x, 0, CGALIB_SET);
/* ... */
```

Draws a bitmap onto the specified screen. The bitmap will be drawn at location x, y, and the drawing mode will be one of the following:

- `CGALIB_SET`: Every pixel in the bitmap, from edge to edge, will be written to the screen, obliterating what was there before. This is useful for filling an area with background tiles.

- `CGALIB_RESET`: The inverse colour of every pixel in the bitmap, from edge to edge, will be written to the screen, obliterating what was there before.
- `CGALIB_AND`: The pixels of the bitmap will be ANDed with what is already on the screen. This is useful to apply a sprite mask to whatever is on the screen, before drawing the sprite.
- `CGALIB_OR`: The pixels of the bitmap will be ORed with what is already on the screen. This is useful to draw a sprite against a background, especially after a sprite mask has already been applied.
- `CGALIB_XOR`: The pixels of the bitmap will be exclusive-ORed with what is already on the screen. This is a cheap way of moving sprites around a background: one XOR operation to draw the sprite, and another XOR at the same location to remove it.

Since the X coordinate must be a multiple of 4, smooth sprite movement requires pre-drawn rotated sprites for the intervening sprite positions. CGALIB is probably not fast enough for action games on a typical CGA PC with an 8088 processor, but sprite-style operations are still useful for things like characters in an RPG location, chess pieces on an ornate board, or units on a strategic map.

9.12 screen->transfer

Declaration:

```
struct screen {
  void (*transfer) (Screen *screen,
    Bitmap *bitmap, int xd, int yd, int xs,
    int ys, int w, int h, int draw);
}
```

Example:

```
Screen *screen;
Bitmap *sprites, *masks;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
/* ... initialise screen background ... */
sprites = new_Bitmap (160, 16);
masks = new_Bitmap (160, 16);
/* ... initialise sprite and mask sheets ... */
screen->transfer (screen, masks, 152, 92, 32, 0,
  16, 16, CGALIB_AND);
screen->transfer (screen, sprites, 152, 92,
  32, 0, 16, 16, CGALIB_OR);
/* ... */
```

Transfers part of a bitmap onto the screen. This is useful if you keep your sprites on a single bitmap that forms a sprite sheet. The example above draws a sprite from such a sprite sheet. screen->transfer is also useful for transferring part of a large play area into a smaller window on the screen, to make a scrolling map. The parameters are as follows:

- `screen` is the destination screen.
- `bitmap` is the source bitmap.
- `xd, yd` are the destination coordinates on the screen to draw to.
- `xs, ys` are the coordinates in the source bitmap to copy from.
- `w, h` are the width and height of the area to copy.
- `draw` is the draw mode: see `screen->put` for a summary.

9.13 screen->box

Declaration:

```
struct screen {
  void (*box) (Screen *screen, int x, int y,
    int width, int height, unsigned int pattern);
}
```

Example:

```
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
screen->box (0, 0, 320, 200, CGALIB_BOX_DITHERED);
```

Draws a box on the specified screen, with the top left at the specified coordinates, and with the given width and height. The pattern can be any 16-bit number that defines a bit pattern, every 2 adjacent bits forming one pixel. There are some defined constants for commonly used patterns:

- `CGALIB_BOX_BLANK`: a solid box in colour 0, usually the background colour.
- `CGALIB_BOX_FILLED`: a solid box in colour 3, the main foreground colour.
- `CGALIB_BOX_DITHERED`: a checkerboard pattern of colours 0 and 3.
- `CGALIB_BOX_VSTRIPED`: vertical stripes in colours 0 and 3.
- `CGALIB_BOX_HSTRIPED`: horizontal stripes in colours 0 and 3.

A number of other patterns can be defined, including multicolour patterns. The 16-bit number defines a 4x2 pixel block that is repeated over the whole surface of the box. For instance, the pattern value 0x1be4 will create a kaleidoscopic box that contains all four colours.

9.14 screen->print

Declaration:

```
struct screen {
  void (*print) (Screen *screen, int x, int y,
    char *message);
}
```

Example:

```
Screen *screen;
screen = new_Screen (CGALIB_MEDIUM, CGALIB_SHOWN);
screen->font = read_Font ("present.fnt");
screen->print (screen, 144, 96, "Welcome!");
```

Prints a message on the current screen, in the currently selected font, at the specified location. The font must be initialised (usually loaded from disk), and assigned to the screen before printing; there is no default font.

9.15 Attributes

Some attributes are intended to be modified directly by the developer. The list of attributes in a screen object are as follows:

- `screen->mode`: the video mode. This should not be modified. Some screen parameters are initialised when the screen is created, and they will not be updated when this attribute is changed, causing corrupt screen output or worse.

- `screen->foreground`: the foreground palette as set by the `screen->palette` method. Changing this while the screen is visible will not change the visible palette; use the `screen->palette` method instead.

- `screen->background`: the background colour as set by the `screen->palette` method. Changing this while the screen is visible will not change the visible background; use the `screen->palette` method instead.
- `screen->ink`: the colour for printing text with `screen->print` or for drawing boxes with `screen->box`. This is intended to be set directly by the developer before printing or drawing boxes. Because of the limitations of drawing box patterns at speed, this should only be changed for box drawing when `screen->paper` is 0, The default for `screen->ink` is 3.
- `screen->paper`: the background colour for a text box created with `screen->print`, or for "off" pixels with a box drawn with `screen->box`. Because of the limitations of drawing box patterns at speed, this should only be changed for box drawing when `screen->ink` is 3, otherwise unpredictable results will occur. The default for `screen->paper` is 0.
- `screen->font`: the font to be used for printing text on this screen. This defaults to NULL, so it *must* be set to a valid font before attempting to use `screen->print`.

Chapter 10

The Bitmap Module

Most of the bitmap functions are accessed through a function pointer in the Bitmap structure. The exceptions are the functions that create the structure: `new_Bitmap` and `read_Bitmap`.

X coordinates and widths in the bitmap functions *must* be divisible by 4. This is for speed: with 2-bit CGA graphics the byte boundaries are on every fourth pixel. Allowing arbitrary X coordinates would necessitate bit rotation and slow the library down.

10.1 new_Bitmap

Declaration:

```
Bitmap *new_Bitmap (int width, int height);
```

Example:

```
Bitmap *bitmap;
bitmap = new_Bitmap (16, 16);
```

This creates a bitmap of the specified size and returns a pointer to it. It should not be used if the bitmap is to be read from a file; in that case use `read_Bitmap` to create the bitmap instead.

10.2 read_Bitmap

Declaration:

```
Bitmap *read_Bitmap (FILE *input);
```

Example:

```
Bitmap *bitmaps[16];
FILE *fh;
int c;
fh = fopen ("tiles.bit", "rb");
for (c = 0; c < 16; ++c)
  bitmaps[c] = read_Bitmap (fh);
fclose (fh);
```

Creates a bitmap, reads its size and contents from an already open file, and returns a pointer to that new bitmap. The example above assumes sixteen bitmaps are stored in the file `files.bit`, and loads all of them into an array of bitmaps.

10.3 bitmap->destroy

Declaration:

```
struct bitmap {
  void (*destroy) (Bitmap *bitmap);
}
```

Example:

```
Bitmap *bitmap;
bitmap = new_Bitmap (16, 16);
/* ... use bitmap ... */
bitmap->destroy (bitmap);
```

Destroys a bitmap, freeing the memory that the bitmap took up. Bitmaps should always be destroyed when no longer needed, to prevent memory leaks.

10.4 bitmap->clone

Declaration:

```
struct bitmap {
  Bitmap *(*clone) (Bitmap *bitmap);
}
```

Example:

```
Bitmap *bitmap, *copy;
bitmap = new_Bitmap (16, 16);
/* ... make bitmap ... */
copy = bitmap->clone (bitmap);
/* ... use bitmaps ... */
copy->destroy (copy);
bitmap->destroy (bitmap);
```

Creates a clone of the specified bitmap, and returns a pointer to it. The clone is an independent copy of the bitmap; modifications made to the clone do not affect the original bitmap.

10.5 bitmap->write

Declaration:

```
struct bitmap {
  int (*write) (Bitmap *bitmap, FILE *output);
}
```

Example:

```
Bitmap *bitmaps[16];
FILE *fh;
int c;
for (c = 0; c < 16; ++c)
  bitmaps[c] = new_Bitmap (16, 16);
```

```
/* ... draw bitmaps ... */
fh = fopen ("tiles.bit", "wb");
for (c = 0; c < 16; ++c)
  bitmaps[c]->write (bitmaps[c], fh);
fclose (fh);
for (c = 0; c < 16; ++c)
  bitmaps[c]->destroy (bitmaps[c]);
```

Writes a bitmap to an already open file. This is more useful than a function that writes a bitmap to a named individual file, as most projects will have multiple bitmaps and will want to combine them into a single asset file for efficiency of storage.

10.6 bitmap->get

Declaration:

```
struct bitmap {
  Bitmap *(*get) (Bitmap *bitmap, int x, int y, int w, int h);
}
```

Example:

```
Bitmap *large, *small;
FILE *fh;
fh = fopen ("large.bit", "rb");
large = read_Bitmap (fh);
fclose (fh);
small = large->get (large, 100, 100, 16, 16);
/* ... use bitmaps ... */
small->destroy (small);
large->destroy (large);
```

Creates a small bitmap by extracting part of a larger bitmap, of w*h pixels, from an area whose top left point is x,y. The small bitmap becomes a separate independent bitmap, which must be destroyed when no longer needed.

10.7 bitmap->put

Declaration:

```
struct bitmap {
  void (*put) (Bitmap *bitmap, Bitmap *source,
    int x, int y, int mode);
}
```

Example:

```
Bitmap *large, *small;
large = new_Bitmap (144, 144);
small = new_Bitmap (16, 16);
/* ... draw both bitmaps ... */
large->put (large, small, 64, 64, CGALIB_SET);
```

Draws a small bitmap onto the specified larger bitmap. The small bitmap will be drawn at location x, y of the larger bitmap, and the drawing mode will be one of the following:

- `CGALIB_SET`: Every pixel in the smaller bitmap, from edge to edge, will be written to the larger bitmap, obliterating what was there before. This is useful for filling an area with background tiles.
- `CGALIB_RESET`: The inverse colour of every pixel in the smaller bitmap, from edge to edge, will be written to the larger bitmap, obliterating what was there before.
- `CGALIB_AND`: The pixels of the smaller bitmap will be ANDed with what is already on the larger bitmap. This is useful to apply a sprite mask to whatever is on the screen, before drawing the sprite.
- `CGALIB_OR`: The pixels of the smaller bitmap will be ORed with what is already on the larger bitmap. This is useful to draw a sprite against a background, especially after a sprite mask has already been applied.

- CGALIB_XOR: The pixels of the smaller bitmap will be exclusive-ORed with what is already on the larger bitmap.

It is no longer necessary, as it was in the original CGALIB, to use a bitmap to implement a back buffer, as this version's Screen module implements a back buffer itself. The functionality is still useful, though. One might use it to create a large map, a portion of which can be scrolled around the screen in a map window, as seen in the games *Star Cadre: Combat Class* and *The Chambers Beneath*.

10.8 bitmap->transfer

Declaration:

```
struct bitmap {
  void (*transfer) (Bitmap *bitmap,
    Bitmap *source, int xd, int yd, int xs,
    int ys, int w, int h, int draw);
}
```

Example:

```
Bitmap *dest, *source;
dest = new_Bitmap (144, 144);
source = new_Bitmap (160, 128);
/* ... prepare both bitmaps ... */
dest->transfer (dest, source,
  112, 112, /* ...where on dest bitmap */
  128, 96, /* ...where on source bitmap */
  16, 16, /* size of area to transfer */
  CGALIB_SET);
```

Transfers part of one bitmap onto another bitmap. The functionality was added in CGALIB1 where the developer had to implement a backbuffer on their own using bitmaps, but there are still uses for it now. The parameters are as follows:

- `bitmap` is the destination bitmap.
- `bitmap` is the source bitmap.
- `xd, yd` are the coordinates on the destination bitmap to draw to.
- `xs, ys` are the coordinates in the source bitmap to copy from.
- `w, h` are the width and height of the area to copy.
- `draw` is the draw mode: see `bitmap->put` for a summary.

10.9 bitmap->box

Declaration:

```
struct bitmap {
  void (*box) (Bitmap *bitmap, int x, int y,
    int width, int height, unsigned int pattern);
}
```

Example:

```
Bitmap *bitmap;
bitmap = new_Bitmap (160, 160);
bitmap->box (0, 0, 160, 160, CGALIB_BOX_BLANK);
```

Draws a box on the specified bitmap, with the top left at the specified coordinates, and with the given width and height. The pattern can be any 16-bit number that defines a bit pattern, every 2 adjacent bits forming one pixel. There are some defined constants for commonly used patterns:

- `CGALIB_BOX_BLANK`: a solid box in colour 0, usually the background colour.
- `CGALIB_BOX_FILLED`: a solid box in colour 3, the main foreground colour.

- `CGALIB_BOX_DITHERED`: a checkerboard pattern of colours 0 and 3.
- `CGALIB_BOX_VSTRIPED`: vertical stripes in colours 0 and 3.
- `CGALIB_BOX_HSTRIPED`: horizontal stripes in colours 0 and 3.

A number of other patterns can be defined, including multicolour patterns. The 16-bit number defines a 4x2 pixel block that is repeated over the whole surface of the box. For instance, the pattern value 0x1be4 will create a kaleidoscopic box that contains all four colours.

10.10 bitmap->print

Declaration:

```
struct bitmap {
  void (*print) (Bitmap *bitmap, int x, int y,
    char *message);
}
```

Example:

```
Bitmap *bitmap;
bitmap = new_Bitmap (144, 144);
bitmap->font = read_Font ("present.fnt");
bitmap->print (bitmap, 56, 68, "Welcome!");
```

Prints a message on the specified bitmap, in the currently selected font, at the specified location. The font must be initialised (usually loaded from disk), and assigned to the bitmap before printing; there is no default font.

10.11 Attributes

Some attributes are intended to be modified directly by the developer. The list of attributes in a bitmap object are as follows:

- `bitmap->width`: the width of the bitmap. This should not be changed after the bitmap is created, or unpredictable results might occur.
- `bitmap->height`: the height of the bitmap. This should not be changed after the bitmap is created, or unpredictable results might occur.
- `bitmap->ink`: the colour for printing text with `bitmap->print` or for drawing boxes with `bitmap->box`. This is intended to be set directly by the developer before printing or drawing boxes. Because of the limitations of drawing box patterns at speed, this should only be changed for box drawing when `bitmap->paper` is 0, The default for `bitmap->ink` is 3.
- `bitmap->paper`: the background colour for a text box created with `bitmap->print`, or for "off" pixels with a box drawn with `bitmap->box`. Because of the limitations of drawing box patterns at speed, this should only be changed for box drawing when `bitmap->ink` is 3, otherwise unpredictable results will occur. The default for `bitmap->paper` is 0.
- `bitmap->font`: the font to be used for printing text on this bitmap. This defaults to NULL, so it *must* be set to a valid font before attempting to use `bitmap->print`.

Chapter 11

The Font Module

Most of the font functions are accessed through a function pointer in the Font structure. The exceptions are the functions that create the structure: `new_Font` and `read_Font`. As with other CGALIB graphical modules, all widths must be multiples of 4 pixels.

11.1 new_Font

Declaration:

```
Font *new_Font (int first, int last, int width,
  int height);
```

Example:

```
Font *font;
font = new_Font (32, 126, 4, 8);
```

This creates a new font and returns a pointer to it. The `first` and `last` parameters specify the first and last ASCII codes that the font includes. It is possible to define fonts that only include, for instance, upper case alphabetic characters, or digits. The `width` and `height` parameters define the size of the characters in pixels. If you want to read a font from disk, there is no need to use `new_Font`; use `read_Font` instead.

11.2 read_Font

Declaration:

```
Font *read_Font (FILE *input, int version);
```

Example:

```
Font *font;
FILE *fh;
fh = fopen ("present.fnt", "rb");
font = read_Font (fh, 2);
fclose (fh);
```

Creates a font, reads its contents from an already open file, and returns a pointer to that new font. The `input` parameter is the input file handle. The `version` parameter is the version of CGALIB under which the font was created. It should be 2, unless you are importing a font created under CGALIB1, in which case it should be 1. CGALIB1 fonts are fixed at 4x8 pixels, and therefore the saved fonts lack character size information.

11.3 font->destroy

Declaration:

```
struct font {
  void (*destroy) (Font *font);
}
```

Example:

```
Font *font;
font = new_Font (32, 126, 4, 8);
/* ... use font ... */
font->destroy (font);
```

Destroys a font, freeing the memory that the font took up. Fonts should always be destroyed when no longer needed, to prevent memory leaks.

11.4 font->clone

Declaration:

```
struct font {
  Font *(*clone) (Font *font);
}
```

Example:

```
Font *font, *copy;
font = new_Font (32, 127, 4, 8);
/* ... make font ... */
copy = font->clone (font);
/* ... use fonts ... */
copy->destroy (copy);
font->destroy (font);
```

Creates a clone of the specified font, and returns a pointer to it. The clone is an independent copy of the font; modifications made to the clone do not affect the original font. This function is useful in order to create or cache variants of fonts, such as those in a different colour, or those with underlines or strikethrough.

11.5 font->write

Declaration:

```
struct font {
  int (*write) (Font *font, FILE *output);
}
```

Example:

```
Font *fonts[3];
FILE *fh;
int c;
for (c = 0; c < 3; ++c)
  fonts[c] = new_Font (32, 126, 4, 8);
/* ... draw fonts ... */
fh = fopen ("fonts.dat", "wb");
for (c = 0; c < 3; ++c)
  fonts[c]->write (fonts[c], fh);
fclose (fh);
for (c = 0; c < 3; ++c)
  fonts[c]->destroy (fonts[c]);
```

Writes a font to an already open file. The example above creates three fonts, and saves them to the same file. Fonts will be saved with a version number of 2.

11.6 font->put

Declaration:

```
struct bitmap {
  void (*put) (Font *font, Bitmap *bitmap, int code);
}
```

Example:

```
Screen *screen;
Bitmap *bitmap;
Font *font;
```

```
FILE *fh;
int c, x, y;

screen = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
fh = fopen ("fntsheet.bsv", "rb");
screen->read (screen, CGALIB_SCREEN_FILE, fh);
fclose (fh);

font = new_Font (32, 126, 4, 8);
bitmap = new_Bitmap (4, 8);
for (c = 32; c < 127; ++c) {
  x = c % 16;
  y = (c - 32) / 16;
  screen->get (screen, bitmap, x, y, 4, 8);
  font->put (font, bitmap, c);
}
bitmap->destroy (bitmap);

fh = fopen ("cstmfont.fnt", "wb");
font->write (font, fh);
fclose (fh);
font->destroy (font);
screen->destroy (screen);
```

Transfers a bitmap into a font character. The example above is almost a complete program: it loads a screen in BSAVE format, which is assumed to have a block of 16x6 characters in the upper left, each character of size 4x8. As each character is extracted into a bitmap, that bitmap is transferred into a character using `font->put`. This is the usual way to create a custom font, and would usually form part of an asset generation program for a project.

11.7 font->get

Declaration:

```
struct bitmap {
  Bitmap *(*get) (Font *font, int code);
}
```

Example:

```
Font *font, *underline;
Bitmap *bitmap;
FILE *fh;
int c;

fh = fopen ("font.fnt", "rb");
font = read_Font (fh, 2);
fclose (fh);

underline = new_Font (font->first, font->last,
  font->width, font->height);
for (c = font->first; c <= font->last; ++c) {
  bitmap = font->get (font, c);
  bitmap->box (bitmap,
    0, font->height - 1,
    font->width, 1,
    CGALIB_SET);
  underline->put (underline, bitmap, c);
  bitmap->destroy (bitmap);
}

underline->destroy (underline);
font->destroy (font);
```

The extracts a character from the specified font and creates a new bitmap out of it. It is the reverse of `font->put`. It is less often used than `font->put`, but can be useful as a means of manipulating font characters. The example above loads a font, applies an underline to every character, transferring these modified characters into a new underline font.

11.8 font->recolour

Declaration:

```
struct bitmap {
  void (*recolour) (Font *font,
    int ink, int paper);
}
```

Example:

```
Font *white, *magenta;

fh = fopen ("font.fnt", "rb");
white = read_Font (fh, 2);
fclose (fh);

magenta = white->clone (white);
magenta->recolour (magenta, 2, 3);
```

Takes a font assumed to be in colour 3 on a background of colour 0, and applies an alternative colour set to it. The source font could be printed in any colour by setting ink and paper attributes of the destination screen or bitmap before printing, but this is slow, as each character is recoloured as it is printed. So instead, font->recolour allows an entire font to be recoloured in advance. If the paper and ink colours of the print destination are left at 3 and 0 respectively, the recoloured font will be shown in its new colours.

The above example assumes a palette of black, cyan, magenta, white. It loads a font assumed to be white-on-black, creates a clone of it, and changes the colours of the cloned font to magenta on white.

11.9 Attributes

The attributes of a font object are intended to be read-only. Here is a list.

- `first`: the first character code in the font. For a full ASCII set, this would be 32, the space.
- `last`: the last character code in the font. For a full ASCII set, this would be 126, the tilde.
- `width`: the width of the font, which is always a multiple of 4.
- `height`: the height of the font.
- `pixels`: the pixel data. The format of this data is the same as the pixel data of a number of bitmaps concatenated. It is technically possible to manipulate fonts by setting these bytes directly, but it is much easier to extract each character into a bitmap.

Chapter 12

The Keyboard Module

The keyboard module supports the 83-key keyboard layout. The extended keys of a 101-key keyboard are mapped on to their equivalents on the 83-key keyboard. For example, on pressing the left cursor key on a 101-key keyboard, the keyboard module will register this as the 4 on the numeric keypad, which is where the 83-key keyboards left cursor function resides.

Most of the keyboard functions are accessed through a function pointer in the KeyHandler structure. The exception is the function that creates the structure: `new_KeyHandler`.

12.1 new_KeyHandler

Declaration:

```
KeyHandler *new_KeyHandler (void);
```

Example:

```
KeyHandler *keys;
keys = new_KeyHandler ();
```

```
/* ... use key handler ... */
keys->destroy ();
```

Creates a new KeyHandler object and returns a pointer to it. If a KeyHandler object already exists, a pointer to the existing KeyHandler is returned instead; there can only be one KeyHandler in existence. The KeyHandler completely takes over control of the keyboard from the BIOS or any TSR program that might be currently handling the keyboard, so the keys that they look for (e.g. Ctrl-Alt-Del) will not be automatically handled.

12.2 keyhandler->destroy

Declaration:

```
struct keyhandler {
  void (*destroy) (void);
};
```

Example:

```
KeyHandler *keys;
keys = new_KeyHandler ();
/* ... use key handler ... */
keys->destroy ();
```

Destroys the KeyHandler when no longer needed, usually at the end of a program. Control of the keyboard is handed back to whatever was handling it before, e.g. the BIOS. Failure to destroy the KeyHandler before closing the program will almost certainly freeze the computer, as the keyboard interrupt will still be trying to use your keyboard handler.

12.3 keyhandler->key

Declaration:

```
struct keyhandler {
  int (*key) (int scancode);
};
```

Example:

```
KeyHandler *keys;
keys = new_KeyHandler ();
printf ("Press ENTER:");
while (! keys->key (KEY_ENTER));
printf ("\n");
while (keys->key (KEY_ENTER));
keys->destroy ();
```

Checks for the state of a particular key, returning 1 if the key is pressed, or 0 if the key is not pressed. The parameter is the scancode for the key. A full list of defined constants for the original XT keyboard is available to save you looking up scancodes; they are listed in `keyboard.h`.

12.4 keyhandler->anykey

Declaration:

```
struct keyhandler {
  int (*anykey) (void);
};
```

Example:

```
KeyHandler *keys;
keys = new_KeyHandler ();
printf ("Press any key:");
while (! keys->anykey ());
printf ("\n");
keys->destroy ();
```

Checks to see if any key has been pressed. If a key was pressed, 1 is returned. If no key was pressed, 0 is returned.

12.5 keyhandler->ascii

Declaration:

```
struct keyhandler {
  int (*ascii) (void);
};
```

Example:

```
KeyHandler *keys;
int ch;
keys = new_KeyHandler ();
printf ("Enter your name: ");
do {
  keys->wait (0);
  ch = keys->ascii ();
  if (ch >= " " && ch <= "~")
    printf ("%c", ch);
} while (ch != 13);
printf ("\n");
keys->destroy ();
```

Returns the ASCII value of the last key pressed, or 0 if no key was pressed. After calling this method, the value for the last key pressed is reset to 0; this will also affect the `anykey()` and `scancode()` methods. The value returned will take note of the Shift key and return upper case characters or symbols as appropriate. It makes the assumption that Caps Lock and Num Lock are off. There are ASCII approximations for a few keys:

- ←: 8 (same as Backspace)
- →: 9 (same as Tab)
- ↓: 10
- ↑: 11

- Del: 127

This allows for implementation of the most rudimentary editing, but it is probably better to use the `scancode` method if you need a more precise reading of the keyboard.

12.6 keyhandler->scancode

Declaration:

```
struct keyhandler {
  int (*scancode) (void);
};
```

Example:

```
KeyHandler *keys;
int sc;
keys = new_KeyHandler ();
printf ("Press some keys (ESC ends):\n");
do {
  keys->wait (0);
  sc = keys->scancode ();
  printf ("0x%02x ", sc);
} while (sc != KEY_ESC);
printf ("\n");
keys->destroy ();
```

Returns the scan code value of the last key pressed, or 0 if no key was pressed. After calling this method, the value for the last key pressed is reset to 0; this will also affect the `anykey()` and `ascii()` methods.

12.7 keyhandler->wait

Declaration:

```
struct keyhandler {
  void (*wait) (void);
};
```

Example:

```
KeyHandler *keys;
keys = new_KeyHandler ();
printf ("Press any key:");
keys->wait ();
printf ("\n");
keys->destroy ();
```

Waits until a key is pressed. You would then use the `ascii` or `scancode` method to see what the key was.

Chapter 13

The Speaker Module

The Speaker module binds together two sub-modules, the Tune module for music, and the Effect module for sound effects. It provides common services for both of them.

13.1 get_Speaker

Declaration:

```
Speaker *get_Speaker (void);
```

Example:

```
Speaker *speaker;
speaker = get_Speaker ();
/* ... make noises ... */
speaker->destroy ();
```

If no Speaker object exists, this method creates a new one. A pointer to the Speaker object is then returned. This is done internally by the Effect and Tune modules, but you will need to call this method in your own program in order to get a reference to the Speaker object in order to destroy it, as neither the Effect nor Tune modules will destroy the Speaker object for you; they do not know when you are finished making noises.

13.2 speaker->destroy

Declaration:

```
struct speaker {
  void (*destroy) (void);
}
```

Example:

```
Speaker *speaker;
speaker = get_Speaker ();
/* ... make noises ... */
speaker->destroy ();
```

This destroys the speaker object when you are done making noises, usually just before your program finishes. This frees up the memory used by the Speaker library.

13.3 speaker->writeint

Declaration:

```
struct speaker {
  int (*writeint) (int *value, FILE *output);
}
```

Example:

```
Speaker *speaker;
FILE *fh;
```

```
int value;
speaker = get_Speaker ();
value = 24;
fh = fopen ("pitch", "wb");
speaker->writeint (&value, fh);
fclose (fh);
speaker->destroy ();
```

Writes a single-byte integer to an already open file. This is used internally by both the Effect and Tune modules, when writing sound effects and tunes to files.

13.4 speaker->readint

Declaration:

```
struct speaker {
  int (*readint) (int *value, FILE *output);
}
```

Example:

```
Speaker *speaker;
FILE *fh;
int value;
speaker = get_Speaker ();
fh = fopen ("pitch", "rb");
speaker->readint (&value, fh);
fclose (fh);
/* ... do something with value ... */
speaker->destroy ();
```

Reads a single-byte integer from an already open file. This is used internally by both the Effect and Tune modules, when reading sound effects and tunes from files.

13.5 Attributes

A single attribute is stored in the Speaker object: `frequencies`. This is a pointer to an array of frequencies for individual notes, and is used by both the Effect and Tune modules. It is not intended that a developer should change these, and it is rare that a developer would want to even refer to them. There are 108 values in this array.

Chapter 14

The Effect Module

The Effect module handles sound effects through the PC speaker. It is a part of the Speaker module. If no Speaker object exists when the Effect module is first used, a Speaker object is created. The developer must remember, if they did not create the Speaker object themselves, to obtain a pointer to it with `get_Speaker` and destroy it when all speaker output is done.

14.1 new_Effect

Declaration:

```
Effect *new_Effect (void);
```

Example:

```
Effect *effect;
effect = new_Effect ();
/* ... define and use sound effect ... */
effect->destroy (effect);
```

Creates a sound effect object. If no Speaker object exists, it will be created here. The effect can be defined by loading it from a file, or by setting the effects attributes directly (see later).

14.2 effect->destroy

Declaration:

```
struct effect {
  void (*destroy) (Effect *effect);
}
```

Example:

```
Effect *effect;
effect = new_Effect ();
/* ... define and use sound effect ... */
effect->destroy (effect);
```

Destroys a sound effect when it is no longer needed. This does not destroy the Speaker object; that must be done separately.

14.3 effect->read

Declaration:

```
struct effect {
  int (*read) (Effect *effect, FILE *input);
}
```

Example:

```
Effect *effect;
FILE *fh;
effect = new_Effect ();
```

```
fh = fopen ("pewpew.eff", "rb");
effect->read (effect, fh);
fclose (fh);
/* ... use sound effect ... */
effect->destroy (effect);
```

Reads the sound effect from an already open file. This is how a program would load sound effects if they had been bundled in an asset file.

14.4 effect->write

Declaration:

```
struct effect {
  int (*write) (Effect *effect, FILE *output);
}
```

Example:

```
Effect *effect;
FILE *fh;
effect = new_Effect ();
/* ... define sound effect ... */
fh = fopen ("pewpew.eff", "wb");
effect->write (effect, fh);
fclose (fh);
effect->destroy (effect);
```

Writes the sound effect to an already open file. This is how sound effects would be written to an asset file, ready to be loaded by the main program in a project.

14.5 effect->play

Declaration:

```
struct effect {
  void (*play) (Effect *effect);
}
```

Example:

```
Effect *effect;
FILE *fh;
effect = new_Effect ();
fh = fopen ("pewpew.eff", "rb");
effect->read (effect, fh);
fclose (fh);
effect->play (effect);
effect->destroy (effect);
```

Plays the sound effect. This is done synchronously; control returns from play() when the sound effect is finished.

14.6 Attributes

The sound effect is defined by directly modifying its attributes. All values are integers in the range 0..255, and are as follows:

- `pattern`: the basic sound pattern used for the sound effect. There are defined constants representing each of the valid values:
  - `EFFECT_NOISE`: noise made from random pitches
  - `EFFECT_FALL`: a falling tone
  - `EFFECT_RISE`: a rising tone
  - `EFFECT_STEP_DOWN`: a single tone, falling every repetition
  - `EFFECT_STEP_UP`: a single tone, rising every repetition
- `repetitions`: the number of repetitions of the sound effect.

- `low`: the lowest tone used in the sound effect. The scale is in semitones, with 0 starting at the bottom C of an organ keyboard, middle C being 24.
- `high`: the highest tone used in the sound effect.
- `duration`: the duration of a single repetition in clock ticks. A clock tick is $\frac{1}{18}$ of a second.
- `pause`: the duration of a pause between repetitions, in clock ticks.

As an example, the code below will define and play an effect with a "pewpew" noise, as commonly used for laser fire in science fiction settings:

```
Effect *pewpew;
pewpew = new_Effect ();
pewpew->pattern = EFFECT_FALL;
pewpew->repetitions = 2;
pewpew->low = 0;
pewpew->high = 60;
pewpew->duration = 3;
pewpew->pause = 0;
pewpew->play (pewpew);
pewpew->destroy (pewpew);
```

Chapter 15

The Tune Module

The Tune module works by keeping the notes short (about $\frac{1}{18}$ second) and achieves the impression of harmony by using arpeggios of these brief notes. Each of these notes is held in a `Note` object, which are brought together within a `Tune` object.

15.1 new_Tune

Declaration:

```
Note *new_Note (void);
```

Example:

```
Tune *tune;
tune = new_Tune ();
/* ... do things with tune ... */
tune->destroy (tune);
```

Creates a new tune object, whose notes can be added by the program, or loaded from previously saved data in a file. Creating the tune object will create a speaker object if one does not exist; destroying the tune object will not destroy the speaker object.

15.2 new_Note

Declaration:

```
Note *new_Note (int pitch, int duration);
```

Example:

```
Note *note;
note = new_Note (24, 12);
/* ... do things with note ... */
note->destroy (note);
```

Creates a new note object. The example creates a note of middle C (pitch 24) that lasts for about $\frac{2}{3}$ of a second (duration 12). Notes are played back staccato, so this note would sound for about $\frac{1}{18}$ of a second, with an $\frac{11}{18}$ pause afterwards. You can also build up arpeggios in this time; see `tune->add` for more details.

15.3 note->destroy

Declaration:

```
struct note {
  void (*destroy) (Note *note);
}
```

Example:

```
Note *note;
note = new_Note (24, 12);
/* ... do *not* add note to a tune ... */
note->destroy (note);
```

Destroys a note. It is important not to use this method after the note has been added to a Tune object. It is rare that you would want to create a note at all if you were not intending to add it to a tune, but `note->destroy` is needed internally to allow `tune->destroy` to destroy the notes that were added to a tune.

15.4 tune->destroy

Declaration:

```
struct tune {
  void (*destroy) (Tune *tune);
}
```

Example:

```
Tune *tune;
tune = new_Tune ();
/* ... do things with tune ... */
tune->destroy (tune);
```

Destroys a tune and all the notes that it contains. The developer shouldn't try to destroy individual notes used in a tune, as `tune->destroy` will have already destroyed them.

15.5 tune->add

Declaration:

```
struct tune {
  void (*add) (Tune *tune, Note *note);
}
```

Example:

```
int frere[32][2] = {
  {24, 12}, {26, 12}, {28, 12}, {24, 12},
  {24, 12}, {26, 12}, {28, 12}, {24, 12},
  {28, 12}, {29, 12}, {31, 24},
  {28, 12}, {29, 12}, {31, 24},
  {31, 6}, {33, 6}, {31, 6}, {29, 6} {28, 12}, {24, 12},
  {31, 6}, {33, 6}, {31, 6}, {29, 6} {28, 12}, {24, 12},
  {24, 12}, {19, 12}, {24, 24},
  {24, 12}, {19, 12}, {24, 24}
};
Tune *tune;
Note *note;
int n;
tune = new_Tune ();
for (n = 0; n < 32; ++n)
  tune->add (tune, new_Note (frere[n][0], frere[n][1]));
tune->play (tune);
tune->destroy (tune);
```

Adds a single note to a tune. The example above prepares and plays a monophonic version of the traditional tune Frère Jacques. Polyphonic music is emulated by using arpeggios. To play multiple notes at the same time, give the first note its proper duration, but give subsequent notes in the chord a duration of 0. These will sound as soon after the first note as possible. Assuming that so many notes can fit into the desired duration, the current chord will last for the duration before the next group of notes is sounded. The following example gives part of Pachelbel's famous Canon in D:

```
int canon[24][2] = {
  {26, 12}, {21, 12}, {23, 12}, {18, 12},
  {19, 12}, {14, 12}, {19, 12}, {21, 12},
  {26, 12}, {42, 0}, {21, 12}, {40, 0},
  {23, 12}, {38, 0}, {18, 12}, {37, 0},
  {19, 12}, {35, 0}, {14, 12}, {33, 0},
  {19, 12}, {35, 0}, {21, 12}, {37, 0}
};
Tune *tune;
Note *note;
int n;
tune = new_Tune ();
for (n = 0; n < 24; ++n)
  tune->add (tune, new_Note (canon[n][0], canon[n][1]));
tune->play (tune);
```

```
tune->destroy (tune);
```

Pay attention to the notes with 0 duration in the array - these are sounded along with the previous note.

15.6 tune->read

Declaration:

```
struct tune {
  int (*read) (Tune *tune, FILE *input);
}
```

Example:

```
Tune *tune;
FILE *fh;
tune = new_Tune ();
fh = fopen ("music.tun", "rb");
tune->read (tune, fh);
fclose (fh);
tune->play (tune);
tune->destroy (tune);
```

Reads a tune from an already open file. This allows music for a project to be packed into an asset file along with the graphics, fonts and sound effects.

15.7 tune->write

Declaration:

```
struct tune {
  int (*write) (Tune *tune, FILE *output);
}
```

Example:

```
int frere[32][2] = {
  {24, 12}, {26, 12}, {28, 12}, {24, 12},
  {24, 12}, {26, 12}, {28, 12}, {24, 12},
  {28, 12}, {29, 12}, {31, 24},
  {28, 12}, {29, 12}, {31, 24},
  {31, 6}, {33, 6}, {31, 6}, {29, 6} {28, 12}, {24, 12},
  {31, 6}, {33, 6}, {31, 6}, {29, 6} {28, 12}, {24, 12},
  {24, 12}, {19, 12}, {24, 24},
  {24, 12}, {19, 12}, {24, 24}
};
Tune *tune;
Note *note;
int n;
FILE *fh;
tune = new_Tune ();
for (n = 0; n < 32; ++n)
  tune->add (tune, new_Note (frere[n][0], frere[n][1]));
fh = fopen ("frere.tun", "wb");
tune->write (tune, fh);
fclose (fh);
tune->destroy (tune);
```

Writes a tune to an already open file. The above example builds up the Frère Jacques tune from an array and then saves it.

15.8 tune->play

Declaration:

```
struct tune {
  int (*play) (Tune *tune);
}
```

Example:

```
Tune *tune;
FILE *fh;
tune = new_Tune ();
fh = fopen ("frere.tun", "rb");
tune->read (tune, fh);
fclose (fh);
tune->play (tune);
tune->destroy (tune);
```

Plays a tune. Initially the tune is played from the beginning. If the user presses a key, the tune will stop and control will return. If `tune->play` is called again, the tune will resume from where it left off. If `tune->play` is called after the tune has finished, playback will start again from the beginning.

15.9 Attributes

The Note object contains the following attributes:

- `pitch`: the pitch of the note. This is in the range of 0..255, with 0 being the bottom C of an organ keyboard. It is possible to modify this value before and after the note has been added to a tune.
- `duration`: the duration of the note, in clock ticks, which are about $\frac{1}{18}$ of a second. The value can range from 0 to 255. A sensible value for a quarter note or crotchet would be 12.
- `next`: a pointer to the next note in a tune. Once the note has been added to a tune, and further notes added afterwards, this pointer is used to link the list of notes together.

The Tune object contains the following attributes:

- notes: a pointer to the first note in the tune. The note's `next` attribute points to the next note, and each note's `next` attribute from there forward points to the following note, right up till the end of the tune. The final note's `next` attribute will be NULL. A developer familiar with linked lists can cut, copy and splice tunes together by inserting and removing sections of this linked list.

- note: a pointer to the current note being played. If NULL, the playback has not started or has finished. Any other value is a pointer to the last note played. This pointer can be changed to point to any of the notes in the `notes` linked list, or alternatively it can be set to NULL so that the next playback will start from the beginning of the tune.

Chapter 16

The Demonstration Game

The demonstration game, just called DEMO.EXE, is an implementation of the classic Droids game. Your aim in this game is to survive for as long as possible, while being chased by a host of droids in an arena. If a droid catches you, then you die and the game is over. A score is kept based on the number of droids destroyed.

The key tool for defeating the droids is their own stupidity. They head straight for the player without looking where they are going. If two droids collide, they are both destroyed, leaving a pile of debris behind. If other droids crash into the debris, they are also destroyed.

Sometimes the player is cornered and there is no escape. Well, there is: the teleport feature. You can teleport at any time to a random part of the arena. But this comes with a risk and a cost. The risk is that you teleport right into the path of a droid which will then kill you. The cost is a score penalty, equal to half the number of droids that the level started with, every time you teleport. So use the teleport feature as sparingly as possible!

The game uses eight way movement. The directional controls are the keypad with or without Num Lock. You can also use the keys Q, W, E, A, D, Z, X, C for movement if you prefer, with S being an extra "down" key. Space teleports. Any other key lets the droids move while the player stands still - useful if the player is shielded by debris and wants the droids to collide with it and cause their own destruction.

If you are playing on a portable monochrome screen, or you just dislike the game's colours, you can launch it in monochrome mode using the -m parameter at the command line. This will use dithering in 640x200 monochrome mode to give a black and white display.

Chapter 17

The Asset Generation Utilities

There are four asset generation utilities supplied with CGALIB that are suitable for small projects. *Makebit* is the bitmap editor, *Makefont* is the font editor, *Maketune* is the music editor, and *Makefx* is the sound-effect generator.

17.1 Makebit

Makebit, the bitmap editor, supports up to 24 bitmaps in a file. Bitmap sizes can be from 4x2 pixels to 24x24. The bitmaps are edited by moving a cursor around an expanded pixel grid. The keys to operate the editor are as follows:

- Cursor keys: move the cursor in the editing grid.
- Page up/down: go to the previous or next bitmap.
- 0..3: paint the pixel under the cursor in the desired colour.

- SPACE: paint the cursor pixel in the last used colour.
- V: flip the bitmap vertically.
- H: flip the bitmap horizontally.
- R: rotate the bitmap clockwise (square bitmaps only).
- SHIFT+R: rotate the bitmap anticlockwise (ditto).
- C: copy the current bitmap to the clipboard.
- P: paste the copied bitmap over the current one.
- X: clear the current bitmap to the background colour.
- F: fill the bitmap with the last used colour.
- INS: add a new bitmap at the current position.
- DEL: delete the current bitmap.
- [and]: change the foreground palette.
- { and }: change the background colour.
- ESC: save and quit.

The following function will load the bitmaps into an array. You need to make sure that the bitmaps array you pass to it contains at least as many elements as the file has bitmaps. For simplicity, this function doesn't check the existence of the file.

```
/* quick and dirty bitmap load function. */
int load_bitmaps (Bitmap **bitmaps, char *filename)
{
    FILE *fp; /* file pointer */
    char header[8]; /* bitmap file header */
    int c = 0; /* counter */
    Bitmap *bit; /* pointer to temporary bitmap */
    /* open the file, read past the header. */
    fp = fopen (filename, "rb");
    fread (header, 8, 1, fp);

    /* read as many bitmaps as the file contains. */
    while ((bit = read_Bitmap (fp)))
        bitmaps[c++] = bit;

    /* return */
    fclose (fp);
    return 1;
}

/* example usage */
Bitmap *bitmaps[24];
load_bitmaps (&bitmaps, "filename.bit");
```

If you need more than 24 bitmaps, or bitmaps larger than the 24x24 pixels that the utility supports, then the recommended course is to put a collage of all the bitmaps onto one or more 320x200 images, saved in BSAVE format. Then write a small utility that loads this image straight into screen memory, extracts the required bitmaps (using `screen->get`), and saves those bitmaps to an asset data file for your project to load. You could also include any fonts, music, sound effects or other data in that asset file. An example of how this works is in the game Team Droid, source code for which is available on the Cyningstan web site.

17.2 Makefont

Makefont, the font editor. This supports fonts of size 4x8 pixels, with up to 255 characters. Each character is edited by moving a cursor around an expanded pixel grid. The keys to operate makefont are:

- Cursor keys: move the cursor in the editing grid.
- Page up/down: go to the previous or next character.
- 0..3: paint the pixel under the cursor in the desired colour.
- SPACE: paint the cursor pixel in the last used colour.
- C: copy the current character to the clipboard.
- P: paste the copied character over the current one.
- X: clear the current character to the background colour.
- F: fill the bitmap with the last used colour.
- [and]: change the foreground palette.
- { and }: change the background colour.
- ESC: save and quit.

The following function will load the saved font ready for use in your own program:

```
/* quick and dirty font load function. */
Font *load_font (char *filename)
{
    FILE *fp; /* file pointer */
    char header[8]; /* bitmap file header */
    int c = 0; /* counter */
    Font *font; /* pointer to loaded font */

    /* open the file, read past the header. */
    fp = fopen (filename, "rb");
    fread (header, 8, 1, fp);

    /* read the font from the input file */
    font = read_Font (fp, header[5] - '0');
    /* return */
    fclose (fp);
    return font;
}

/* example usage */
Font *font;
font = load_font ("example.fnt");
```

If you need to save multiple fonts in a file, or you want to create fonts that are sizes other than 4x8 pixels, then you will need to use an alternative method: draw the characters onto an image file with a program like PCPaint, save the image in BSAVE format, and have your asset generation routine create the font by `get`ting them into bitmaps and `put`ting them into the font. The following function loads an 8x8 font from an image file, where the character codes are assumed to range from 32 to 127 and are arranged in six rows of sixteen characters:

```
Font *makefont (void)
{
    FILE *fp; /* input file pointer */
    Screen *scr; /* screen to load image into */
    Font *fnt; /* the new font file */
    Bitmap *bit; /* temporary bitmap for character */
    int c; /* character counter */

    /* make the screen object and load it in */
    scr = new_Screen (CGALIB_MEDIUM, CGALIB_HIDDEN);
    fp = fopen ("image.bsv", "rb");
    scr->read (scr, CGALIB_SCREEN_FILE, fp);
    fclose (fp);

    /* make the font and copy the characters */
    fnt = new_Font (32, 127, 8, 8);
    for (c = 32; c <= 127; ++c) {
        bit = scr->get (scr,
            8 * (c % 16), 8 * (c / 16) - 16,
            8, 8);
        fnt->put (fnt, bit, c);
        bit->destroy (bit);
    }

    /* destroy the screen and return the font */
    scr->destroy (scr);
    return fnt;
}
```

17.3 Maketune

Maketune is the music editor. It allows you to create beeper music and save it in a file. The tune is displayed as rows of notes to be simultaneously played, starting with the duration of the whole chord in clock ticks. While editing the music, you can play it back from the current cursor position to the end of the file. Moving the cursor to the start of the tune allows playback of the whole tune. The keys to operate the program are:

- Up/Down: move the cursor highlight bar up and down the tune. The cursor highlights the current note row in white on magenta. When editing a line, these instead increase or decrease the current note by a semitone.
- PgUp/PgDn: view the previous or next page of notes.
- Home/End: go to the start or end of the tune.
- INS: Insert a set of simultaneous notes at the cursor position.
- DEL: Delete the set of simultaneous notes at the cursor position.

- ENTER: Alter the values of the notes at the cursor position.
- SPACE: Play the tune, starting at the cursor position.
- M: Set a copy marker for notes to be pasted. When not highlighted by the cursor, these notes will be highlighted in black on cyan.
- C: copy the notes from the copy marker to the current cursor positon. The copy marker and the cursor will both move down, allowing long sections of music to be copied by repeatedly pressing C.
- [: show accidentals as flats.
-]: show accidentals as sharps.
- ESC: save and quit. When editing a line, this finishes editing; if in the process of adding a note to the line, that note will be discarded.

The tune can be loaded into your program with the following routine:

```
/* quick and dirty tune loader */
static Tune *load_tune (char *filename)
{
    Tune *tune; /* the new tune */
    tune = new_Tune ();
    fp = fopen (filename, "rb");
    fread (header, 8, 1, fp);
    tune->read (tune, fp);
    fclose (fp);
    return tune;
}

/* example usage */
Tune *tune;
tune = load_tune ("example.tun");
```

17.4 Makefx

Makefx is the sound effect editor. It allows you to create up to twelve sound effects, and save them in a file. The program allows you to play the sound effects to see if they are to your liking. The keys to operate the program are:

- Up/Down: move the cursor highlight bar up and down the effect list.
- INS: Insert a sound effect at the highlighted (empty) slot.
- DEL: Delete the sound effect at the highlighted slot.
- ENTER: Alter the values of the sound effect at the highlighted slot.
- SPACE: Play the sound effect at the highlighted slot.
- C: copy the current sound effect to the clipboard.
- P: paste the copied sound effect over the current one.
- X: clear the current sound effect to the default values.
- ESC: save and quit.

These sound effects can be loaded into your program with a function like the following:

```
/* quick and dirty sound effect loader */
static void load_effects (Effect **effects, char *filename)
{
    FILE *fp; /* file pointer */
    char header[8]; /* effect file header */
    int c; /* counter */

    /* attempt to open the file, and read and verify the header */
    fp = fopen (filename, "rb");
    fread (header, 8, 1, fp);

    /* read the effects */
    c = 0;
    while (c < 12 && (effects[c]->read (effects[c], fp)))
        ++c;
    while (c < 12)
        effects[c++] = NULL;

    /* return */
    fclose (fp);
}

/* example usage */
Effect *effects[12];
load_effects (effects, "example.eff");
```

Chapter 18

Linux Compilation

There is limited support for compiling CGALIB into a Linux library archive. This is not intended to create production quality linux versions of CGALIB projects, but instead allows the projects to be compiled under Linux in order to use the superior debugging facilities that Linux offers, e.g. gnudb and valgrind. It was created after the author spent too much time debugging crashes and memory leaks in a DOS project, a task that becomes almost trivial with the Linux debug tools.

The Linux version of CGALIB emulates a CGA screen in a 640x400 window. It supports full emulation of CGA mode 4 and 5, giving access to all the colour palettes. It does not support 640x200 monochrome mode. A number of limitations prevent the Linux version of CGALIB from being suitable for production-quality Linux projects:

1. There is no support for maximising or minimising the CGA window.

2. The window is not redrawn if it becomes obscured, or if the user switches to another workspace and back again.

3. Sound is not supported at all.

The intention of Linux compilation is purely as a debug tool, to ensure that memory leaks and memory access errors that DOS debug tools would miss can be easily found. Once a project is free of these bugs, it should be compiled under DOS. If the developer wishes to release native Linux (or Windows or Mac) versions of their DOS CGALIB projects, then they should take advantage of DOSBox's licence which allows DOSBox to be bundled with DOS projects for easy installation under modern operating systems.

Chapter 19

Possible Future Developments

CGALIB is distributed in a complete state. But there are some possibilities for future development. Some of the ideas that might be taken up in future developments are:

- Clipping on `scr->put` and `bit->put` operations. This will allow source bitmaps to overlap the edge of the destination bitmap, so that sprites can be shown leaving the visible play area more cleanly.
- Support in the font editor for font sizes other than 4x8.
- Proper support for software sprites. This would include automatic application of masks, frames of animation, and sprite rotation. This would enable CGALIB to be used for action games, or at least give some animation to liven up turn-based games.
- Conversion to assembly. Some of the functions are still slow on older systems such as a 4.77 MHz 8088-based PC. Although the off-screen graphics manipulation features help to mitigate this, it

would be nice to speed everything up by converting some or all of the functions to assembly language.

The following ideas are not possibilities for future development, as they go beyond the use case of CGALIB or nullify any speed advantage over using a compiler's own graphics library:

- Full EGA/VGA graphics mode support. CGALIB obtains part of its speed advantage by being able to make assumptions about the screen memory layout.

- Support for 16-colour pseudo graphics at 160x100 resolution. It would be attractive to have support for this mode, but for the same reason as the idea above, that would be better implemented as a separate library.

CYNINGSTAN