COMP 2659 Course Project – Stage 2: Raster Graphics Library

Released: Thursday, January 25, 2018
Target Completion Date: Friday, February 2, 2018 (at the latest)

Objectives:

- To learn about raster ("bitmapped") graphics and the frame buffer.
- To learn about common 2D plotting operations, and to gain experience implementing them efficiently in software.

Overview

In the previous stage you developed a detailed specification for a simple 2D, monochrome video game for the Atari ST. **It is critical that the specification is as thorough, complete and clear as possible before proceeding.**

In this stage you will design and implement a raster graphics library to support the required graphics capabilities of your game.

The library is not responsible for animation. Its routines are for plotting static images like pixels, lines, shapes and/or bitmaps to the frame buffer. The graphics will become animated in stage 5.

Requirement: Provision of Low-Level Plotting Routines

Depending on your game, you must to develop some or all of the following low-level routines, and perhaps others:

- clear screen and/or clear region
- plot pixel
- plot horizontal line
- plot vertical line
- plot line (generic)
- various plot "shape" routines (where shape is something like square, rectangle, polygon, etc.)
- plot bitmap (perhaps multiple variants, e.g. for different bitmap dimensions)

These routines are "low level" from a design perspective because they will play a support role in later stages: other modules will invoke them to request fundamental screen drawing operations. They are also "low level" in the sense that they must not in turn rely on support from existing software modules. No operating system support is allowed in the implementation of your library, nor may it call third-party or standard C library functions. All plotting must be performed by your code, directly to the frame buffer.

Each function must take a "base" pointer as input, as well as function-specific input. The base pointer is the start address of the frame buffer into which the plotting must be performed.

Remember that your drawing routines may need to cope with screen boundary conditions. Depending on your game, this may include wrap-around or clipping. Forgetting about screen edges is a major cause of bugs.

Requirement: Game-Independence

Your raster graphics library must be independent of your game, in the sense that the library's design will not be coupled to game rules, object types, physics, events, etc.

For example, if the game was Pong, the raster graphics library would not "know" about paddles, balls, their properties or behaviours, ball movement or rebound physics, scoring rules, etc. If Pong's paddles and ball were each 4 bits wide, however, the library could include a *generic* routine for plotting *arbitrary* 4-pixel-wide bitmaps. This routine could later be called variously by other code to accomplish the plotting of paddles, the ball, and other 4-pixel wide images used in the game.

In theory, it should be possible to reuse the library for other games.

Despite the above, for this project, you are only required to implement the graphics routines that your specific game will need. For example, if your game's graphics are based around 16×16 bitmaps, then you will certainly need to provide some variant of a `plot_bitmap_16` routine. But, you might not need to implement routines for plotting line segments or individual pixels.

A Note on Text

Games often require the printing of text. This is easily accomplished using "bitmap" fonts: a bitmap (called a "glyph") is provided for each character that might be plotted. Therefore, as long as bitmaps can be plotted, it will later be possible to print text in a straightforward way.

It is possible to design your own bitmap fonts (or maybe to find one that is free to use). Alternatively, it is possible to use the Atari ST's own bitmap font table. Your instructor will discuss how to accomplish this later. As well, I:\Labs\CompSci\Resources\2659\project\fonts folder contains a number of 8×16 font tables that you can use. For future reference, the font table is an array of 256 8×16 glyphs, indexed by ASCII value. If you plan to use these, then consider including a plotting routine for 8-bit-wide bitmaps as part of stage 2.

A Note on Smart Graphical Layout for Convenience and Efficiency

As was described in an earlier document, it is easiest (for architectural reasons) to base the game's graphics as much as possible on bitmaps which have widths of 8, 16 or 32 pixels (or multiples thereof) – ideally a suitable set of bitmaps have already been developed as part of your stage 1 game specification.

If possible, it is convenient to arrange the frame buffer area so that the plotting of horizontal lines, bitmaps, etc. is aligned as much as possible on byte or word boundaries. When possible, this can lead to faster plotting and sometimes to simpler plotting algorithms.

The above are not possible in every game. For example, if some objects move smoothly and horizontally across the screen, then plotting (at least of those objects) will not always be byte-aligned.

Requirement: Test Driver

Your raster graphics library must be accompanied by a "test driver" program which invokes and thoroughly exercises each of its plotting routines.

Add tests for each function you write as you go. If you avoid prompt testing, it will be impossible to build robust code.

A Note on Efficiency

At first, focus on the correctness of the plotting routines. Implement them in C, and avoid tricky optimizations.

Once a routine is working correctly, it may then be possible to speed it up (sometimes by a large factor). Having a thorough set of tests will allow you to verify changes quickly and repeatedly.

Once a routine is working as quickly as possible in C, one final consideration might be to rewrite it in assembly language so that the code can be further hand-optimized. However, this should only be attempted as extra time permits. The benefits may only justify the time and effort for the most heavily-called routines.

Skeleton Code

Your graphics library must be implemented as a C module named "raster" consisting of the following three files:

- `raster.h` ← header file describing the module's public interface, as needed by clients
- `raster.c` ← contains definitions for routines implemented in C (i.e. most of them)
- `rast_asm.s` ← contains definitions for routines implemented in assembly (if any)

To assist you, your instructor has placed a skeleton project in:

I:\Labs\CompSci\Resources\2659\project\stage 2\skeleton

Also included in this folder are a make file for the project and a test driver skeleton.

A Note on the Proper Use of Header Files

A header file is intended to contain declarations relevant to a corresponding project module. Therefore, header files should contain information needed by client modules, including public function prototypes and public type definitions. Header files must *not* contain variable declarations or function definitions which result in space being reserved or in object code generation! Among other problems, this can lead to link errors (what if the same "`.h`" file is included in more than one "`.c`" module?). For some reason, a common student mistake is to define arrays for bitmaps in header files. Avoid this! Types, constants, functions, etc. which are purely for internal use in the module should similarly not be mentioned in the header.

Other Requirements

Your code must be highly readable and self-documenting, including proper indentation and spacing, descriptive variable and function names, etc. No one function should be longer than approximately 25-30 lines of code – decompose as necessary. Code which is unnecessarily complex or hard to read will be severely penalized.

For each function you develop, write a short header block comment which specifies:

1. its purpose, from the caller's perspective (if not perfectly clear from the name);
2. the purpose of each input parameter (if not perfectly clear from the name);
3. the purpose of each output parameter and return value (if not perfectly clear from the name);
4. any assumptions, limitations or known bugs.

At the top of each source file, write a short block comment which summarizes the common purpose of the data structures and functions in the file.

You should add inline comments if you need to explain an algorithm or clarify a particularly tricky block of code, but keep this to a minimum.