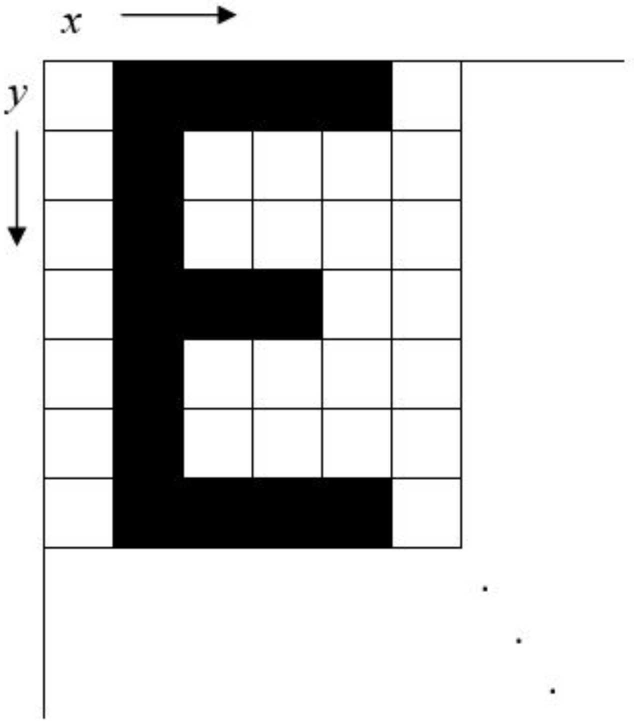
Raster Graphics

Raster Graphics

- raster = a rectangular grid of "pixels" (picture elements)
- **Note:** in raster graphics (0,0) refers to the upper-left
- In monochrome graphics, each pixel is either on or off
  - these notes pertain to monochrome unless stated explicitly

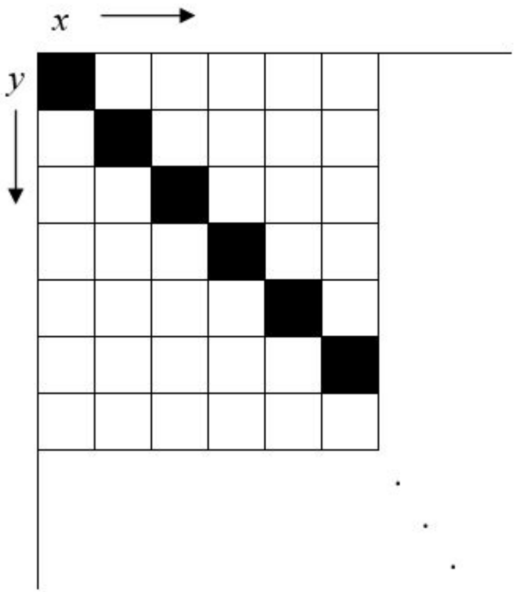


Raster Graphics

- In colour graphics, each pixel is (typically) a mixture of shades of RGB.
- E.g. on the Atari ST (when using a colour monitor), there are 8 shades of each for a total of 512 colours.
- E.g. of raster output devices: typical modern monitor, dot matrix printer, laser printer

Raster Graphics

- How would a diagonal line be drawn?



Vector Graphics

- Is an alternative to raster graphics
- Vector graphics based geometric primitives such as lines
  - instead of pixels
- Vector output devices can draw straight lines and smooth curves.
- E.g. of vector output devices: plotter, vector monitor
- https://www.printcnx.com/resources-and-support/addiational-resources/raster-images-vs-vector-graphics/

Raster vs Vector Graphics

- Lines and curves are just approximations (“staircasing”) con
    - … but increasing the resolution and using techniques like anti-aliasing mitigate this
- Image data size is constant pro or con?
- Image output is constant time pro

Frame Buffers & Bitmaps

- A frame buffer (FB) is region of memory which holds an image to display
  - The FB get displayed to the screen
  - Programmers do **not** write directly to the screen
- A Bitmap is raster image data
- How does a programmer “see” the screen?
- The FB is simply memory. Thus, can only access the screen in the same methods as we access memory.

Frame Buffers & Bitmaps - Atari ST

- On the Atari ST, the frame buffer is 32,000 bytes. Note:
- By default, it is located at the highest RAM locations ($3F8000 – $3FFCFF on an ST with 4Mb of RAM), just above the stack.
    - The video hardware automatically and periodically scans this buffer to produce a video signal, sent out on the video port to the monitor (details to come)

Frame Buffers & Bitmaps - Atari ST

- In monochrome mode the resolution is 640 × 400 (“high resolution”),
    - i.e. 640 pixels per scan line (one line across the screen) and 400 pixels down the screen (i.e. 400 scan lines). This can be thought of as a 2D array of pixels!
    - Thus, the total number of pixels on the Atari screen is 640 * 400 = 256,000 pixels.
    - 256,000 / 8 = 32,000 bytes

Frame Buffers & Bitmaps - Atari ST

- The frame buffer location can be changed
  - Any 256-byte aligned region of memory can serve as the frame buffer
- We will avoid doing this to start, but “page flipping” will be useful later
- FB size is the same regardless of the mode (mono/colour) which explains why the size of the colour screen is half the mono screen

Frame Buffers & Bitmaps

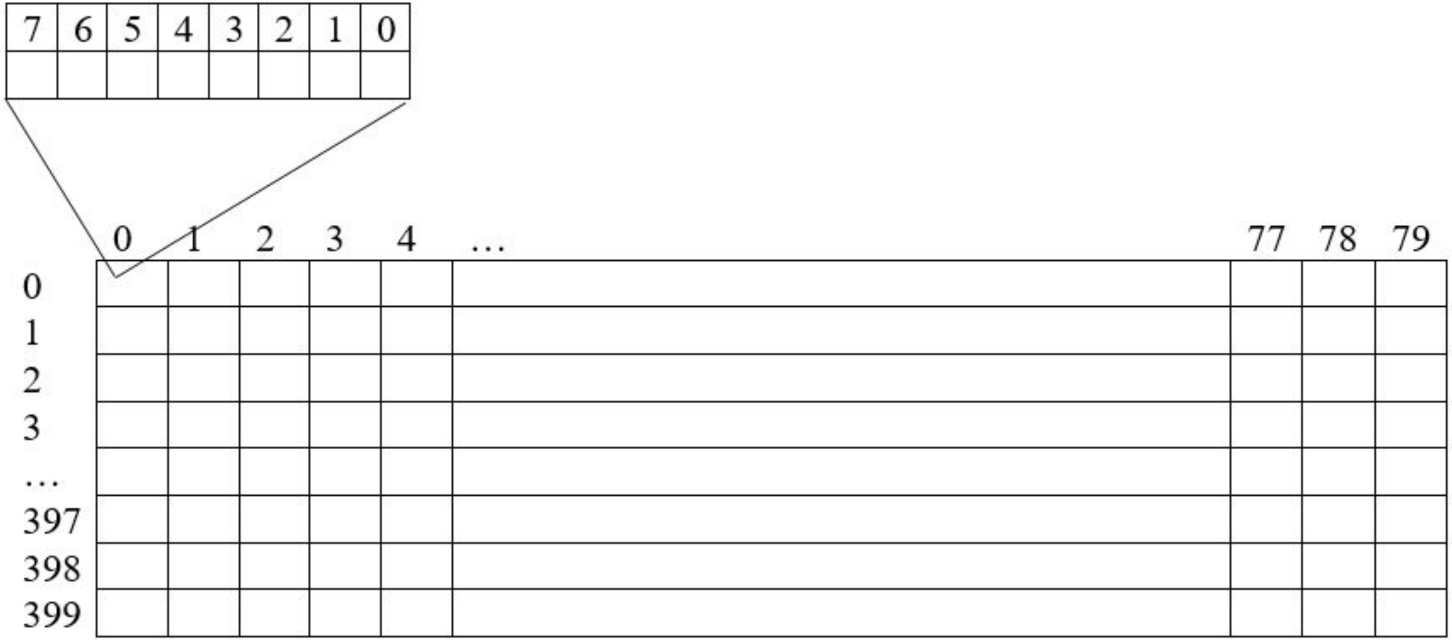
- How does a programmer “see” the screen?
- The FB is simply memory. Thus, can only access the screen in the same methods as we access memory. So how do you access memory?
- Programmers access memory via variables, which are memory locations whose size is either a byte, a word or a longword

Frame Buffers & Bitmaps

- Thus, if accessed as bytes the screen can be viewed as:

Frame Buffers & Bitmaps

- Thus, if accessed as bytes the screen can be viewed as:



Plotting Routines

- Plotting is performed by writing into the frame buffer
- E.g. code which plots to frame buffer on Atari ST:

```
char *base = (char *)Physbase();      /* system call returns FB start */

*base = 0x80;

*base = *(base + 80) = *(base + 160) = 0xFF;
```

Primitive plotting routines

- Primitive graphics routines include the following:
    - plot pixel
    - plot vertical line
    - plot horizontal line
    - plot line (generic)
    - plot “shape”
        - where shape = triangle, rectangle, generic polygon, circle, etc.
    - plot bitmap
    - clear screen/region
    - etc.

Plotting Routines

- Given a screen location as pixel coordinates: x, y plot this pixel
    - y = row   however, need to consider bytes per line
    - need to use x to determine:
        - which byte is being referenced
        - which bit in this byte is being referenced

plot_pixel routine

```
#define SCREEN_WIDTH  640
#define SCREEN_HEIGHT 400

void plot_pixel(char *base, int row, int col) {
    if (col >= 0 && col < SCREEN_WIDTH && row >= 0 && row < SCREEN_HEIGHT)
        *(base + row * 80 + (col >> 3)) |= 1 << (7 - (col & 7));
/*                             ------                    -----
                               col / 8                     col % 8
*/
}
```

Plotting Routines

- The shifts and bitwise operations are far faster
  - compared to div/mod
  - use the 68000 reference card to see the speed difference
- For a single pixel using div and mod is not bad, but when plotting many pixels speed is critical
- The plotting of horizontal and vertical lines can be implemented using “plot pixel”, but this is inefficient. Why?

Plotting Routines

- Horizontal line re-calculating pixel positions in the same byte is redundant
- Vertical line the pixel position is the same for every line.
- Exercise: sketch the basic ideas of the optimized versions.
- For a vertical line, from x, y1 to x, y2:
  - A vertical line is a single pixel on a series of consecutive scan lines
  - Compute the address of the byte for x, y1
  - Compute the bit position for x, y1 in this byte
  - In a byte with one bit set, shift the pixel to the correct position
  - Loop from y1 to y2 writing this byte to the correct memory location

Plotting Routines

- For a horizontal line, from x1,y to x2,y:
  - Simplifying assumption: that the mod of x1 by 8 is 0 and the mod of x2 by 8 is 7. Both of these conditions mean that both the starting and ending locations require a full byte
  - Compute the start address of the byte for x, y1
  - Compute the end address of the byte for x, y2
  - Loop from the start address to the end address writing 0xFF to each byte

Plotting Routine

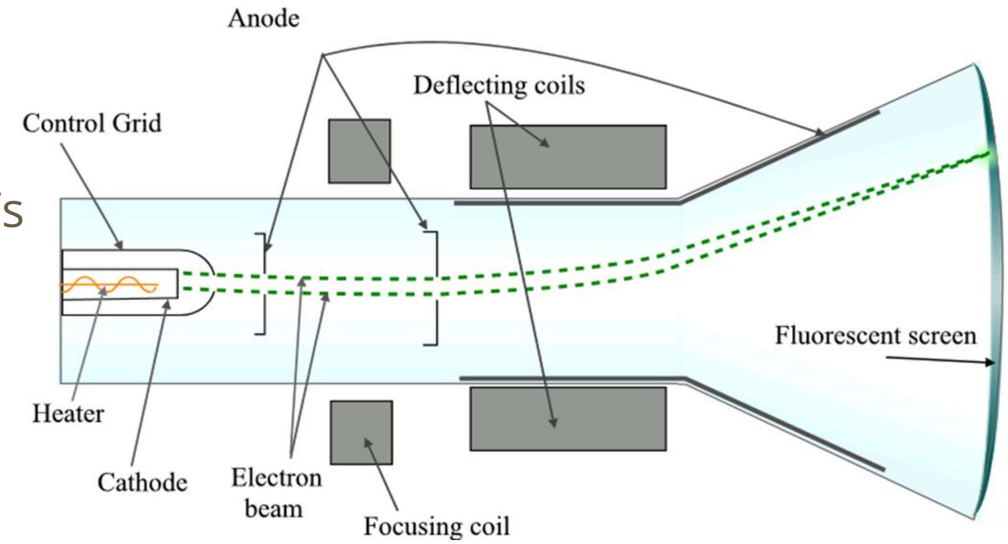
- Removing the simplifying assumption
- Thus, it is possible that only a portion of the starting byte and the ending byte are used
  - Do the same loop from previous slide
    - From *start address+1* to *end address–1*
  - Shift **0xFF** to the right the appropriate number of bits based on x1 and write it to the start address
  - Shift **0xFF** to the left the appropriate number of bits based on x2 and write it to the end address

Plotting Routine

- There is an efficient algorithm for plotting arbitrary straight lines called Bresenham’s algorithm
- The actual implementation of these routines are left as exercises.

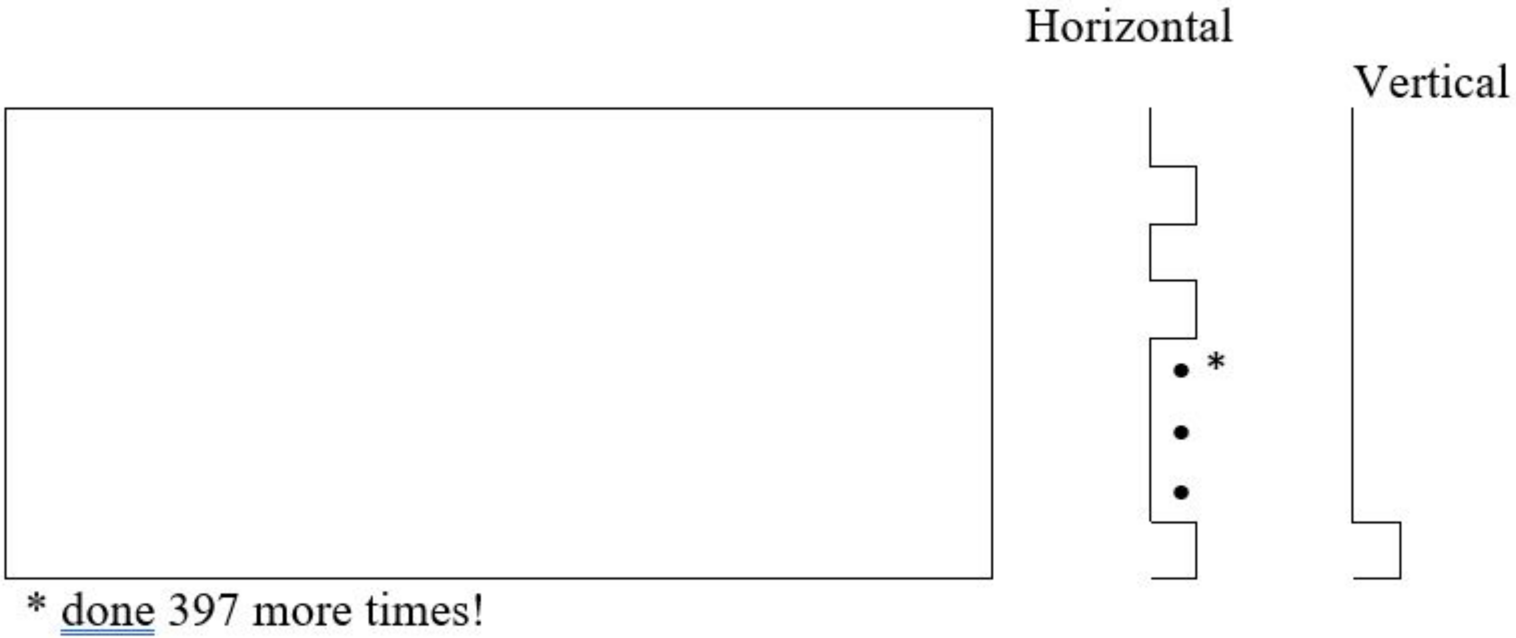
Video Output Hardware

- How does a monitor work? We will consider traditional CRT monitors as the ST uses this technology
- LCD displays have mostly replaced CRT monitors
  - there are pros and cons …
- Electromagnets at right angles to each other control the beam's vertical and horizontal position ("deflection")



Video Digital Hardware

- Diagrams: horizontal and vertical sync signals and the resulting scan lines



Video Digital Hardware

- As it sweeps along, the beam's intensity is modulated by a video signal. The signal is blanked during horizontal and vertical retraces
- Monitors have a refresh rate
  - The number of times the entire screen is redrawn in one second
- Why don't we notice a flicker?
- Because the refresh rate is faster than our eyes can perceive

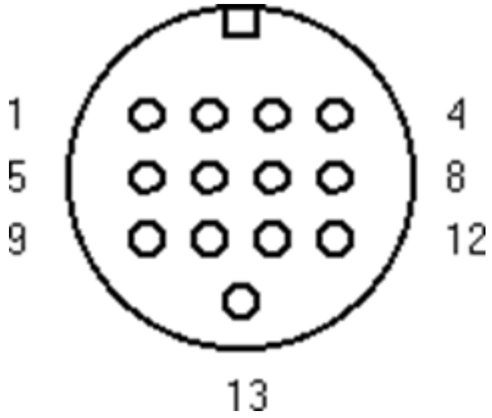
Video Digital Hardware - Transmission

- The video signal is sent out to the monitor through the computer's video port
- Example:
  - an analog signal through a VGA port, or some other port standard
  - a digital signal through a DVI, HDMI, or DisplayPort port

Video Digital Hardware - Transmission

- E.g. the Atari ST video port
  - source: “The Atari ST Internals” by Jim Boulton - modified:

```
01  Audio out
02  (unused)
03  General purpose output
04  Monochrome detect
05  Audio in
06  Green
07  Red
08  Ground
09  Horizontal sync
10  Blue
11  Monochrome          monochrome video signal
12  Vertical sync
13  Ground
```



Video Digital Hardware

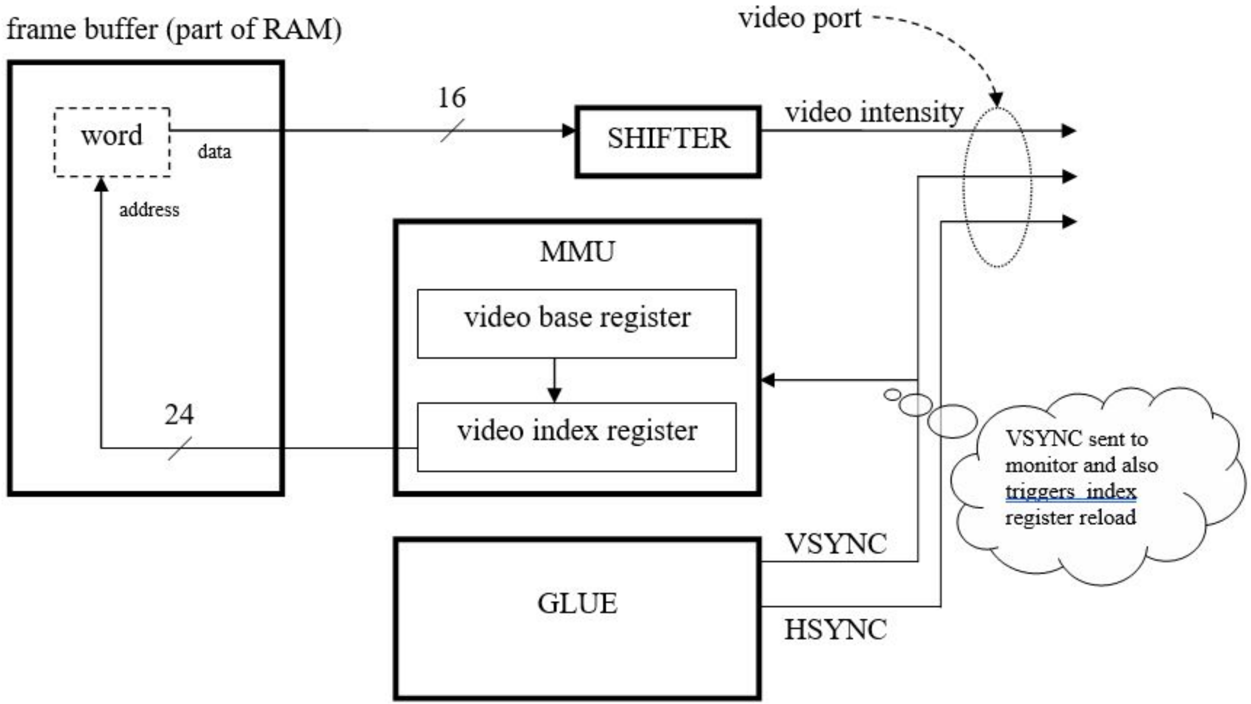
- How is the video signal generated?
  - The contents of the frame buffer must be repeatedly scanned and "serialized"
- E.g. On the Atari ST:
  - The monochrome monitor has a refresh rate of 70 Hz.
  - The frame buffer must therefore be rescanned every 1/70th of a second.

Atari ST Video Refresh

- On the Atari ST, a chip called the SHIFTER repeatedly indexes through the frame buffer (actually, it is assisted by the MMU chip).
    - For each word it loads, it shifts the value out through the video port
    - This happens independently of the CPU

Atari ST Video Refresh

- Diagram: overview of frame buffer, SHIFTER, video port and monitor



Video Digital Hardware

- What problem can arise if the frame buffer is being written to while it is being simultaneously read (drawn)? What is the solution?
  - There can be tearing of the image being displayed!
- Realize that the CPU is processing at 8MHz vs the Shifter at 70 Hz
- Therefore, it is possible for the FB to be changed before it is written.
- Solution: do NOT write to the buffer while it is being used.
- How do you know it is NOT being used?
  - VSYNC

Text and the Console Driver

- Text must also be plotted.
- The simplest technique is to use a “bitmap font”, where each symbol is stored as a bitmap
  - The font is then an array of bitmaps, indexed by a character encoding such as ASCII or Unicode
- E.g. TOS includes a small number of fixed width fonts. The regular font is based on 8 × 16 bitmaps
  - This divides the screen into 80 text columns and 25 text rows
- Bitmap fonts don’t scale well

Text and the Console Driver

- Other techniques are more flexible, such as “outline fonts” which describe a symbol in terms of vectors.
- When invoking a system call to write a character to the screen, the O/S delegates to a “console driver”. Its responsibilities include:
    - Indexing the font array and plotting to the frame buffer
    - Managing the text cursor position and scrolling
    - Interpreting special characters (e.g. newline, control characters, etc.)
    - Perhaps keeping track of the screen’s text contents
    - Perhaps managing output buffering (not under TOS, though)

Text and the Console Driver

- E.g. TOS supports VT52 “terminal emulation” codes. For example, writing the string ESC E causes the screen to be cleared and the text cursor to be reset to (0,0).
- On the Atari in C Esc must be supplied as an octal constant, “\033E”, numbers starting with a 0 in C are treated as octal.

Colour

- Colour information can be encoded in the frame buffer by allocating more than one bit per pixel
- On the Atari ST:
    - The frame buffer size is constant (32,000 bytes), but three resolutions are supported:
        - High resolution = 640×400 (1 bpp 2 colours)
        - Medium resolution = 640×200 (2 bpp 4 colours)
        - Low resolution = 320×200 (4 bpp 16 colours)
- Note: the colour monitor supports only the lowest two resolutions, while the monochrome monitor supports only high resolution

Colour

- ... But wait! The ST supports 8 shades each of RGB, giving 512 colours!
- The SHIFTER maintains palette registers. The palette can be changed, but only a max. of 16 colours can be loaded at a time. (Similar to GIF images which has a 256 colour palette, i.e 8 bits)
- Systems with large amounts of memory can afford larger frame buffers, and therefore much richer colour without using palettes

Line-A (TOS-specific)

- Line-A is the low-level graphics part of TOS
    - It is also the interface by which TOS can be queried about mouse and font information
- Here, we only study enough of Line-A to be able to poll mouse state
- Line-A is actually a collection of system calls and global system variables
- To access any of these, Line-A call #0 must be invoked first (but only once)

Line-A (TOS-specific)

- E.g. from C – track and display mouse x coordinate until key press:

```
/* File:  TRK_MSE.C */
#include <stdio.h>
#include <osbind.h>
#include <linea.h>                          /* necessary header file */
                                            /* for MOUSE_BT          */

int main()
{
     short x;

     linea0();                              /* init. Line-A */
     x = GCURX;                             /* poll mouse x coord. */
     printf("%3d\r", x);
     fflush(stdout);

     while (!Cconis())                      /* repeat 'til key press: */
          if (x != GCURX)                   /* display x if changed */
          {
               x = GCURX;
               printf("%3d\r", x);
               fflush(stdout);
          }

     Cnecin();                              /* consume key press */
     return 0;
}
```

GPUs

- In this course, all graphics operations are to be implemented in software
- However, modern computers often have heavy graphics processing loads
  - Doing everything in software is too CPU-intensive.
- A GPU (Graphics Processing Unit), such as one found in a PC graphics card, removes much of the burden from the CPU
  - It implements 2D and 3D graphics “primitive” operations
    - such as plotting triangles in hardware

GPUs

- Idea:
  - CPU sends drawing commands to GPU
  - GPU plots (“rasterizes”) into frame buffer using optimized algorithms in H/W
  - GPU (or associated H/W) is responsible for generating video signal from frame buffer contents

GPUs

- A software library such as OpenGL or DirectX acts as an abstraction layer, so programs can control the GPU effectively.
- Note that optimized frame buffer RAM typically resides on a PC's graphics card as well, although it is "mapped" into the main address space and is CPU-accessible.
- E.g. later versions of the Atari ST shipped with a "blitter" chip – a chip for copying and plotting bitmaps (the "bit blit" operation). This can be considered the ancestor of modern GPUs