Article development led by [acmqueue](https://queue.acm.org)
queue.acm.org

Emulating a video system shows how even a simple interface can be more complex—and capable—than it appears.

BY GEORGE PHILLIPS

Simplicity Betrayed

AN EMULATOR IS a program that runs programs built for different computer architectures from the host platform that supports the emulator. Approaches differ, but most emulators simulate the original hardware in some way. At a minimum the emulator interprets the original CPU instructions and provides simulated hardware-level devices for input and output. For example, keyboard input is taken from the host platform and translated into the original hardware format, resulting in the emulated program “seeing” the same sequence of keystrokes. Conversely, the emulator will translate the original hardware screen format into an equivalent form on the host machine.

The emulator is similar to a program that implements the Java Virtual Machine (JVM). The difference is merely one of degree. JVM is designed to enable efficient and tractable implementations, whereas an emulator’s machine is defined by real hardware that generally imposes undesirable constraints on the emulator. Most significantly, the original hardware may be fully described only in terms

of how existing software uses it. JVM tends to be forward-looking with the expectation that new code will be written and run under JVM to increase its portability. Emulators tend to be backward-looking, expecting only to make old code more portable.

In either case the emulator and JVM give the programs that run under them a suite of services needed to interact with the host system. JVM presents those services as a series of API calls; the emulator presents them as simulated hardware. Nonetheless, the simulated hardware is an API—just not in the form most programmers expect.

TRS-80 Example

As an example hardware API, consider the TRS-80 video system, which displays text and graphics on a modified television set. It has 16 lines of characters with 64 columns each. It supports the normal ASCII character set, with an additional 64 graphics characters allowing every possible combination of a 2-pixel by 3-pixel sub-block. Judicious use of the graphics characters provide an effective 128-pixel by 48-pixel resolution, albeit with pixels the size of watermelon seeds. A TRS-80 program displays a character by writing the character value to the memory location associated with the desired position. In effect, the API has only one call:

```
void ChangeCharacter  
  (location /* 0 - 1024 */,  
   character)
```

Emulating such a simple graphics format is trivial. It can be rendered quite adequately on a 512-by-192 image allotting each character an 8-by-12 rectangle. Each graphics pixel is a 4-by-4 rectangle, while the characters themselves occupy the upper two-thirds, or an 8-by-8 area. While you could get away with any old font for the characters, a little more work will get something that looks dot-for-dot identical to the original. (To get the

same aspect ratio as the original, the image should be doubled in height. We'll keep the numbers as is to simplify exposition.)

Figure 1 shows how an emulator converts the hardware-level character values into an actual image representation. While the figure serves as a blueprint for the emulator, it also shows what a TRS-80 program must do in order to display something. One detail is missing: changes to screen memory are not displayed instantaneously. The hardware redraws the display every $1/60^{\text{th}}$ of a second. This means the emulator must assemble the static image as described and dis-

play it every $1/60^{\text{th}}$ of a second.

The resulting emulation will look very much like the original. The majority of programs that run under the emulator will appear exactly the same as when run on original hardware, but if you pay very close attention you will see some differences. Watch as the screen is cleared from all black to all white. In both the original and the emulation you get a tear, because filling the screen takes long enough that it goes over more than one video frame. Just before the fill starts, the frame will be black. The fill is partially done on the next frame, and the display shows white at the top and black

at the bottom. That happens for just an instant; by the next frame the filling is done and the frame is filled with white.

Even though this is at the edge of perception, the tear exhibited by the two is quite different. You will need a video camera to see it in the original, but the emulator can either dump frames for offline analysis or be told to step one frame at a time. Figure 2 shows the difference between the tear on the original hardware and that seen in the emulator.

Although apparently minor, this difference is puzzling. The emulator implemented the specification exactly as written, and no bugs were found in the code. Moreover, the specification was obviously correct and complete. Except for the evidence at hand, the situation is impossible. Of course, the problem lies in the specification, which only appears complete. The assumption that a particular character is either there or not is incorrect. The hardware does not draw a character at a time; it draws one line of a character at a time. That character has the opportunity to change after the drawing has already started. The tearing on the original results from a character being blank on the initial passes and subsequently filled in. Put another way, the character is not atomic but made up of 12 pieces stacked on top of one another; each piece is 8-by-1. Incidentally, those 8-by-1 pieces are atomic—they are displayed entirely or not at all. The graphics hardware ends up reading each displayed character 12 times.

Refining the emulator to correct this difference is straightforward. Instead of waiting an entire $1/60^{\text{th}}$ of a second before drawing the screen, it will be drawn a line at a time. With 192 lines the emulation loop looks something like this:

```
for (i = 0; i < 192; i++) {
    emulate CPU for 86.8 micro-seconds
    draw line i of the video display
}
```

Now the tearing on the emulator is the same as the hardware. You may be tempted to declare the specification and the emulator complete be-

Figure 1. Translating TRS-80 screen memory into a displayed image.

Figure 2. Difference in tears between the emulation and the original hardware.

cause of the major increase in output fidelity. As a conscientious developer, however, your reaction must be exactly the opposite. A rather small test case required a considerable change in the program. Now is the time to investigate further and look for additional bugs. In all likelihood the specification needs more refinement. At the very least, a better test case for the new functionality is needed. After a bit of thought it becomes clear that displaying a one-line-high pixel (1-by-4) would make such a test case.

This can be done in three simple steps.

1. Write an ordinary 4-by-4 pixel on screen.
2. Wait until the first line has been drawn by the graphics hardware.
3. Quickly erase the pixel.

All that will be visible on screen is the 1-by-4 part of the pixel that was drawn before you pulled the rug out from under the 4-by-4 pixel. Many pixels can be combined to create something seemingly impossible on a stock TRS-80: a high-resolution diagonal line.

The only thing missing is some way of knowing which line the hardware is drawing at any one point. Fortunately, the graphics hardware generates an interrupt when it draws a frame. When that interrupt happens you know exactly where the graphics hardware is. A few difficulties of construction remain, but they come down to trivial matters such as putting in delays between the memory accesses to ensure you turn pixels on and off in step with each line being drawn.

Here the emulator is a boon. Making such a carefully timed procedure work on real hardware is very difficult. Any mistake in timing will result in either no display because a pixel was erased too quickly or a blocky line caused by erasing pixels too slowly. Not only does the debugger not care about time, it eschews it entirely. Single-stepping through the code is useless. To be fair, the debugger cannot single-step the graphics hardware. Even if it did, the phosphor would fade from sight before you could see what was happening.

The emulator can single-step the

The majority of programs that run under the emulator will appear exactly the same as when run on original hardware, but if you pay very close attention you will see some differences.

processor and the graphics. It can show exactly what is being drawn and point out when screen memory writes happen at the incorrect times. In no time at all a demonstration program is written that shows a blocky line in a simple emulator and a diagonal line in a more accurate emulator (see Figure 3).

The Real Machine

The program is impressive as it must read/write to the display with micro-second-level timing. The real excitement is running the program on the original machine. After all, the output of the emulator on a PC is theoretically compelling but it is actually producing graphics that pale in comparison to anything else on the platform. On the real machine it will produce something never before seen.

Sadly, the program utterly fails to work on the real machine. Most of the time the display is blank. It occasionally flashes the ordinary block line for a frame, and very rarely one of the small pixels shows up as if by fluke.

Once again, the accurate emulation is not so accurate. The original tearing effect proves that the fundamental approach is valid. What must be wrong is the timing itself. For those strong in software a number of experimental programs can tease out the discrepancies. Hardware types will go straight to the schematic diagrams that document the graphics hardware in detail. Either way, several characteristics will become evident:

- ▶ Each line takes 64 microseconds, not 86.8.
- ▶ There are 264 lines per frame; 192 visible and 72 hidden.
- ▶ A frame is 16,896 microseconds or 59.185 frames per second, not 60.

What's most remarkable is how the emulator appeared to be very accurate in simulating a tear when it was, in fact, quite wrong. So much has been written about the brittleness of computer systems that it is easy to forget how flexible and forgiving they can be at times. The numbers bring some relief to the emulator code itself. What appeared to be floating-point values for timing are in fact just multiples of the system clock. Simple, integer relationships exist between the speed of the CPU and the graphics hardware.

A Z-80 Cycle Waster

The Z-80 code is in the comments alongside equivalent C code. The C program is self-contained and runs an exhaustive test verifying that waitHL() always uses $H * 256 + L + 100$ cycles. Observe that the JR conditional branch instructions take extra time if the branch is taken. Those time differences along with looping are used to expand the subroutine's running time in proportion to the requested number of cycles.

```
// Z-80 registers are akin to C global variables.
```

```
unsigned char A, H, L;
unsigned char C, Z;
```

```
int cycles;
```

```
void t(int n)
{
    cycles += n;
}
```

```
void waitA();
```

```
/*   Cycle Cost   C code           Z-80 code */
```

```
void wait256()
{
    wHL256:
        t(4);    H--; Z = H != 0;    // DEC  H
        t(7);    A = 127;            // LD   A,127
        t(17);   waitA();            // CALL wA
}
```

```
void waitHL()
{
    wHL:
        t(4);    H++; Z = H == 0;    // INC  H
        t(4);    H--; Z = H == 0;    // DEC  H
        t(7);    if (!Z) {            // JR   NZ,wHL256
            t(5);    wait256();
                       goto wHL;
        }
        t(4);    A = L;              // LD   A,L
        t(0);    waitA();
}
```

```
void waitA()
{
    wA:
        t(4);    C = A & 1;          // RRCA
        A = (A << 7) | (A >> 1);
        t(7);    if (C) {              // JR   C,have1
            t(5);    goto have1;
        }

        t(4);                                // NOP

    have1:
        t(4);    C = A & 1;          // RRCA
        A = (A << 7) | (A >> 1);
        t(7);    if (!C) {              // JR   NC,no2
            t(5);    goto no2;
        }

        t(7);    if (!C) {              // JR   NC,no2
            t(5);    goto no2;
        }
}
```

```
no2:
    t(4);    C = A & 1;          // RRCA
    A = (A << 7) | (A >> 1);
    t(7);    if (!C) {              // JR   NC,no4
        t(5);    goto no4;
    }

    t(5);    if (!C) {              // RET  NC
        t(6);    return;
    }

    t(4);                                // NOP

    no4:
        t(4);    C = A & 1;          // RRCA
        A = (A << 7) | (A >> 1);
        t(7);    if (!C) {              // JR   NC,no8
            t(5);    goto no8;
        }

        t(13);   /* *0 = A; */        // LD   (0),A

    no8:
        t(7);    A &= 15;            // AND  A,15
        Z = A == 0;
        t(5);    if (Z) {              // RET  Z
            t(6);    return;
        }

    wait16:
        t(4);    A--; Z = A == 0;    // DEC  A
        t(7);    if (!Z) {              // JR   NZ,wait16
            t(5);    goto wait16;
        }

        t(5);    if (Z) {              // RET  Z
            t(6);    return;
        }
}
```

```
//-----
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
    for (int hl = 0; hl < 65536; hl++)
    {
        H = hl / 256;
        L = hl & 255;
        cycles = 0;
        waitHL();

        if (cycles != hl + 100)
        {
            printf("Blew it on %d (got %d instead of %d)\n", hl, cycles, hl + 100);
        }
    }

    return 0;
}
```

We can restate timings from the CPU's perspective:

- ▶ Each line takes 128 cycles.
- ▶ The hidden 72 lines go for 9,216 cycles.
- ▶ Each frame is 33,792 cycles ($264 * 128$).

The number of frames per second is still a floating-point number, but the emulator core can return to integers as you might expect for a digital system.

With the new timings in place, the emulator exhibits the same problems as the real hardware. With a bit of (tedious) fiddling with the timing, the program almost works on the real hardware.

There's just one problem left. Remember that interrupt that gave a synchronization point between the CPU and the graphics hardware? Turns out

it only happens every *second* frame. The program works but flashes between a perfect diagonal line and a chunky one. There's no hardware facility to help out here, but there is an obvious, if distasteful, software solution.

Once the diagonal line has been drawn, you know exactly when it must be drawn again: 33,792 cycles from when you started drawing it the first time. If it takes T cycles to draw the line, then you just write a cycle-wasting loop that runs for $33,792 - T$ cycles and jump back to the line-drawing routine. Since that jump takes 10 cycles, however, you better make that $33,792 - T - 10$ cycles. This seems like a fine nit to pick, but even being a single cycle off in the count will lose synchronization. In two seconds the sync is off by almost an entire line. Losing sync

has an effect similar to the vertical roll that afflicted old televisions.

An ad hoc solution will work just fine. The proof-of-concept demonstration program will be complete. The possibilities for even more impressive graphics are clear. Hand-timing everything, however, is tedious, slow, and error prone. You're stuck with writing in assembly, but the timing effort takes you back to the days when code was hand-assembled. Much of the burden can be lifted by taking the instruction timing table from the emulator and putting it into the assembler. Assemblers have always been able to measure the size of their output, generally to fill in buffer sizes and the like. Here's that facility in use to define *length* as the number of bytes in a message, which will vary if the message is changed:

Figure 3. A diagonal line in a simple emulator and a more accurate emulator.

Incorrect

Correct

```
message: ascii      'Hello,
world.'
length  byte *-message
```

This works because the special `**` variable keeps track of the memory location into which data and code are assembled. To automate timing simply add a `time()` function that says how many cycles are used by the program up to that point. It can't account for loops and branches but will give accurate results for straight-line code. At a high level the diagonal slash demo will be:

```
start: ...some code to draw
        ...the diagonal line
waste equ      33792      -
(time(*) - time(start)) - 10
        ...code to use up
        ..."waste" cycles
goto start
```

Straightforward, but what about the code to waste the cycles? The assembler could be extended to supply that code automatically. Instead, keeping with the principle of minimal design, the task can be left to an ordinary subroutine. Writing a subroutine that runs for a given number of cycles is a different requirement from what you are accustomed to, but it is possible. (See the accompanying sidebar for one such cycle-wasting subroutine.)

Figure 4. The letter "A" displayed with square pixels and on the original hardware.

As programmers we can see the potential of the diagonal-line demonstration program. Although it has only one pixel per line, there is a clear path to more complex and compelling images, to say nothing of animations and other effects. One final bump in the road awaits. Every time the CPU accesses screen memory, it denies access to the graphics hardware. This results in a blank line that is two- or three-characters wide. The more pixels you change on a per-line basis, the more blanked-out portions there will be. Once again you will find that although the graphics may look fine on the emulator, they will be riddled with “holes” on the real machine because of the blanking side effect.

Moreover, as you try to do more work per line, the exact positions of the blank spots will matter a great deal. Their exact positions will be a measure of emulator accuracy and can be used to maximize the graphics displayed per line. Several discoveries await and will be the result of a feedback loop of emulator refinement, test program development, measurement of the original system leading to further emulator refinement, and so on. Along the way you will discover the following:

- ▶ The visible portion of a line takes 102.4 cycles; the rest of the time (25.6 cycles) is used for setting up drawing the next line.
- ▶ Blank spots do not cover the entire time an instruction takes but only the sub-portion of the instruction that accesses video memory.
- ▶ The emulator must be extended to report exactly when memory is accessed on a sub-instructional basis.
- ▶ Our method of synchronization is crude and can be depended upon to be accurate only to within a few characters.
- ▶ Finer synchronization can be accomplished, but the emulator must be upgraded so programs using the technique can still be tested.
- ▶ Video blanking can be put to good use sculpting graphics that cannot be constructed in other ways.

In other words, we're a long way from where we started. Instead of drawing an entire screen at once or even a line at a time, the emulator is down to drawing 1/12th of a character

at a time and interweaving the CPU and the graphics hardware at the level of CPU cycles. The graphics emulation has become extremely accurate. Not only will side effects such as a tear be seen, but they will be exactly the same as they manifest on the original hardware. The results are not purely academic, either. Test programs demonstrate the fidelity of the emulator while still achieving the same output on the original hardware. The result is not tiny differences only of interest to experts but extremely visible differences in program behavior between precise and sloppy emulators.

Can there be anything else?

Having tripped over so many emulator shortcomings, can the answer be anything but yes? In fact, there is a double-wide mode where the characters are doubled in size for a 32-by-16 display. Based on what we've seen up to this point, it's not surprising to learn that it brings in many more complications than might be expected. Even leaving that morass aside, there's one more obvious limitation of the emulator. The original display was a CRT. Each pixel on it looks entirely different from what is seen on a modern LCD flat panel. The pixels there are unrelentingly square, whereas the CRT produced soft-edged ovals of phosphorescence. Figure 4 compares two close-ups of the letter A.

Hard-edged pixels result in an image that is functionally identical to the original but has a completely different feel. The difference between the two is unmistakable. Observe also that the real pixels are neither distinct nor independent. Pixels in adjacent rows overlap. Pixels in adjacent columns not only overlap but also display differently if there is a single one versus several in a row. The first pixel in a row of lit pixels is larger. All these subtle differences combine to create a substantially different picture.

The problem itself is much simpler than the functional issues because there is no feedback to the rest of the implementation. There is no need to change the CPU timing or how the CPU interacts with the graphics system. It is merely a matter of drawing each dot as an alpha-blended patch rather than a hard-edged off/on setting of one or two pixels. What is

troublesome is the increased effort required by the host CPU to pull this off. The work involved is many times greater than before. Only through the aid of a graphics coprocessor or moderately optimized rendering code can the screen be drawn in this fashion in real time. It is difficult to believe that drawing a 30-year-old computer's display takes up so much of a modern system. This is one reason why accurate emulation takes so long to perfect. We can decide to make a better display, but today's platforms may not have the horsepower to accomplish it.

That realistic fuzzy pixels can overlap does lead to noticeable visual artifacts. Two pixels alternating between on and off sitting side by side will appear to be three pixels: two flashing pixels on each end and a single always-on pixel in the middle where the two overlap. I'll leave it to your imagination what useful effect this artifact may have.

Conclusion

A system's complexity is easy to underestimate. Even the simple video system of the TRS-80 has greater depth than anticipated. What lurks beneath the surface is far greater than the high-level description. Take it as a sideways reinforcement of the KISS principle. Yet do not despair. You must also consider the power of tools. Each emulator improvement has led to discoveries that could be exploited for good use once the necessary support tools were built. Above all, however, beware of perfection. No system is perfect, and the cost of pursuing perfection can be much greater than mere time and money invested.

Related articles on queue.acm.org

No Source Code? No Problem!

Peter Phillips and George Phillips
<http://queue.acm.org/detail.cfm?id=945155>

Enhanced Debugging with Traces

Peter Phillips
<http://queue.acm.org/detail.cfm?id=1753170>

George Phillips (gp2000@shaw.ca) works on video games both emulated and otherwise. In a simpler time he worked on the crawling side of an early Web search engine that was chastised for selling advertisements.

© 2010 ACM 0001-0782/10/0600 \$10.00