Frameskipping, interpolation and re-entrant IRQ code by Cadaver

(This rant is mirrored from Cadaver's site)

This is a theoretical rant about what you can possibly do when it seems you run out of rastertime and your program starts to slow down. Of course, the obvious solution is to optimize code or leave routines out entirely, but this is not about that…

0. Running out of time

Traditionally games & demos use frame-based movement; for example 50 times a second the screen & sprites move a little bit to create the illusion of motion. But for the C64, trying to do all the movement/logic code, as well as the actual graphics code (scrolling the screen, raster IRQs etc.) on each frame can simply be too much.

If a program has been designed well, going “over” that “limit” doesn't involve ugly graphical effects like flickering/jerking of screen, but it's just visible as an overall slowdown. Practically, as I wrote in a previous rant, this design involves making sure that no matter what happens, raster interrupts have enough time to do their screen-update task; in fact their task should be just the immediate setting of VIC registers (like screen-splits and sprite multiplexing) and playing music/sound.

All the time-consuming things like movement, AI & scrolling are done in the main program; if it is too “slow” the screen will nevertheless show correctly. However, this rant challenges that tried-and-true approach with a new, more complex but more powerful approach.

The basic question is, what can we do once we see that the C64 can't simply handle the load? Of course, there's no magic in this; no approach can magically give you more clock cycles, but there are ways to sort of “cheat”, while keeping the internal logic of the program uncompromised.

1. Movement/logic vs. graphics update

To know what we can do about slowing down we have to see if two distinct areas can be identified from the code:

Movement/logic

- For advancing the internal state of the program. This means moving the characters, processing AI, doing collision detection, and executing virtual machine bytecode :)

Graphics update

- For rendering the internal state onscreen. On C64, this means things like scrolling (shifting the screen memory if no VIC tricks are used), sorting the sprites for multiplexing, and actually showing the multiplexed sprites.

Usually in games these are easy to separate, while in demo effects this is not always the case. If they can't be separated in your project, the rest of this rant will not be of much use to you.

2. Frameskipping

Let's first look at an approach many know from PC games/demos: frameskipping. PCs are usually powerful enough to handle the AI & movement on each frame but not necessarily to render complex graphics scenes.

The idea is to keep track of the time a frame was last rendered onscreen; if for example the previous rendering was 3 frames away, call the movement/logic code 3 times before the next rendering call. Now the motion becomes more jerky, as not each frame is drawn, but the apparent speed remains the same.

Frameskipping can be very useful in 3D games or isometric 2D bitmap games even on C64. Of course, if also the movement/logic code takes a lot of time, it will not help much.

Also in games like Last Ninja, where the sprites have to be masked against the background (very time-consuming!), frameskipping can help immensely. Now I'm not sure if Last Ninjas actually frameskip. I remember seeing the characters move with bigger steps in some CPU-intense scenes, like the maintenance areas in LN2:Office level, so likely at least LN2 frameskipped.

3. Interpolation

But what if we want to still keep running at 50Hz, scrolling almost the whole screen with lots of sprites flying around? And what if we have some really complex AI routines that take a lot of time? Perhaps we want to depack sprites in realtime, too?

This all is (within some limits) possible on a stock C64. The key is to not run your full movement/logic every frame. Run it for example each 2nd or each 4th frame, and you'll have quite a bit of time left over. In the in-between frames, interpolate the movement of sprites with a line equation (linear interpolation).

I'm not sure if anyone has used this in a C64 game before? The realization came to me in the end of 2001, and after that I started developing this idea. I know that some games like Gauntlet 3 or Myth scroll at 50Hz, while updating the movement of sprites at a lower rate; however they don't show the inbetween- frames for sprites.

There are different ways to do this. The easier but not-so-powerful approach is this: The main program handles everything, raster interrupts just show the screen & sprites in the way the main program wishes. The update loop could then be something like:

1st frame:

- Do full movement/logic code. Before movement, store all the “old” positions of sprites.

- Scroll screen if necessary
- Sort sprites
- Instruct raster IRQs to update screen on next frame

2nd frame:

- Interpolate sprite positions between old & new (much faster than performing the full movement)
- Scroll screen if necessary
- Sort sprites
- Instruct raster IRQs to update screen on next frame

3rd frame:

- Start the cycle again from the beginning

The key to get good performance is to always calculate as much as you can, don't stop to wait! If you have a double-buffered screen, you don't have to care of the raster beam position when you scroll the screen-RAM (except if you use character-sprites like Turrican etc.) Of course, you *do* have to care of it when you scroll the color-RAM :)

However, the bottleneck is still the full movement/logic code. By doing only the interpolation on each second frame we win a little time but still, if the movement/logic takes too much time, the frame update won't be ready on time and the whole action slows down.

4. Advanced interpolation & re-entrant IRQs

This is a bit like multitasking. The concept of this was quite complex & disgusting to me at first but then I realized it can be quite a nice way to do things. Now the movement/logic can continue on its task whenever the CPU has free time and frame updates will still come in time.

We'll have 3 areas of code:

- Movement/logic, handled by main program. Execution of this is triggered each 2 or 4 frames. This code doesn't touch any VIC registers or handle scrolling on its own!

- Code for handling the next frame update and interpolating sprite movement. Scrolling goes here also. This will be executed from IRQs, however, whenever needed, it will be interrupted by

- The actual low-level raster IRQ code. Screen splits, setting multiplexed sprites onscreen and playing music/sound.

The idea is that once the movement/logic part has completed processing, it waits that a frame counter maintained by the frame update IRQ has reached its end value (2 or 4). Now it will reset this counter, and the frame update IRQ will interpolate the next 2 or 4 frames. After this, it stops. The catch is that if the movement/logic part is too slow or has too little CPU time left over from the other areas, a visible “pause” will occur at this point.

Another thing to consider (especially on NTSC machines) is that if a big portion of screen is scrolled, with color-RAM, and with lots of sprites to be interpolated & sorted, the frame update IRQ might also run too slow. In that case it should just wait for the next frame.

Now the big question is: how do we invoke the frame update IRQ while also letting the low-level IRQs run? I don't want to use multiple interrupt sources, so I did the following:

For starters, all IRQs must save CPU registers on the stack. Using fixed zeropage addresses instead would corrupt the regs in case of nested IRQs (that will occur with this approach!)

I usually have one raster interrupt at the top of the screen, to set up the gamescreen display, and to start firing up the sprite multiplexing interrupts. And then another in the bottom of the screen for scorepanel display & playing music.

Now, whenever either the top or bottom interrupt has completed processing, we do, before exiting it:

```
        dec $d019                         ;Acknowledge raster interrupt
        cli                               ;Allow further interrupts
        jmp frameupdate                   ;Jump to the frame update code
```

The frame update IRQ code itself must not be re-entered, so the first thing it has to do is to maintain an execution counter like:

```
frameupdate:
        inc exec_count
        lda exec_count                    ;If already executing, skip the
update
        cmp #$02                          ;code
        bcs skip
        <actual frame update code here>
skip:   dec exec_count
        pla                               ;Exit the interrupt
        tay
        pla
        tax
        pla
        rti
```

We see that the frame update code can be invoked from either the top or the bottom interrupt. So it must maintain some kind of internal state about what it is going to do next. And whenever it can't do anything useful for the time being, it exits. For example a color-RAM update is only sensible to do after the bottom interrupt (must not be visible!) This complicates things a bit…

A problem with this “multitasking” approach is that each of the code areas must maintain their own set of sprite information, so that they don't step on each other's toes, causing wrong sprite movement/animation to be shown. You might already be familiar with this if you've done doublebuffered sprite- multiplexing; this is just extending that idea one more layer.

Doing movement/logic only on each 4 frames is already quite heavy interpolation. The actual motion happens in big steps. I do this in MW4's second preview that runs at 50Hz, unlike the first preview. I was personally worried that control might feel too lagged with this low update rate so I got myself some guinea pigs (thanks CreaMD and Pixman) but after they confirmed that it didn't feel too lagged I carried on with my plans.

5. Conclusion

I hope these ideas were interesting. Any example code would be as big as an entire game engine itself so I'll leave that out :), but you can study the source code of the second MW4 preview to see how it was implemented in practice.

Note that MW4 full game uses only the simpler interpolation method described in chapter 4 of this rant, as it didn't need anything more, and the memory use of the chapter 4-style interpolation would have been too heavy. Also, 4-frame interpolation feels indeed quite lagged, just compare the preview to the full game :)

Lasse Öörni - loorni@student.oulu.fi