Multidirectional scrolling and the "game world" by Cadaver

(This rant is mirrored from Cadaver's site)

This is a short and theoretical (not going into the technical specifics much) rant about making a block-based “game world” that is scrolled in many directions and populating it with objects.

0. Background

Some of the knowledge in this rant originally comes from the book “C-64 Pelintekijän Opas” (“C-64 Game Maker's Guide”) by Jukka Tapanimäki, author of Netherworld and others. Also credits shall go to my big brother who did initial experiments with blockmap-scrolling in the late '80s, and I soon followed.

This rant mainly expresses my own methods and experiences, they're not necessarily the best or anything but have worked nicely, anyway.

1. The map-data and the block-data

It would probably take too much memory to store each screen of a game world as only the character codes, without any encoding. However, Chris Butler's Commando and Ghosts'n Goblins do just fine by doing things this way, so it's not entirely unusable.

The popular method of graphics encoding seen in for example SEUCK is the map- data & block-data system (sometimes the blocks are also called tiles.) The rest of this rant assumes you're familiar with this system.

In the terms of this rant, map data consists of 8-bit block numbers, so there can be 256 different blocks maximum. I've never used multiple levels of blocks (blocks constructed of even smaller blocks like seen in Faery Tale Adventure) so I'm not going into such things, however they would allow scrolling *huge* maps around.

The way I organize map-data in memory is that each horizontal row is stored in the memory sequentially, starting from the top of map and ending to the bottom. To make any sense of the map-data, the length of a row in blocks is very significant to know, so I store it in a “map header” before the map data itself. To avoid any multiplication operations, I precalculate the starting address of each map row into an array, from which it can be easily accessed. Similarly, I also precalculate the starting address of each block (maptbllo/hi and blktbllo/hi tables.)

The data for each block is stored similarly, each horizontal row sequentially. For example the following-looking block:

```
ABCD
```

```
EFGH
IJKL
MNOP
```

would simply be stored in the memory as ABCDEFGHIJKLMNOP

I prefer 4×4 blocks: the block size is always a tradeoff between memory use and the complexity of graphics that can be achieved. But for easy calculations, the block size should be a power of two. The block data consists usually of 8-bit values too, that are the characters (screen codes) used to display the block on screen.

2. Scrolling

Scrolling on the C64 is not so pleasant because after scrolling one character- length with the hardware registers, screen memory data (and color memory data too, possibly) need to be shifted in the direction of the scroll. Shifting 1000 bytes (the whole screen) around takes more than half of the rastertime of a frame.

(I won't go into any VIC-II trick methods of scrolling, they're usually too limiting for games, at least for scrolling in all directions)

Anyway, it's not so bad as the screen shifting operation can be split on multiple frames, based on maximum scrolling speed. (If you scroll with 4 pixels/frame, you can split the shifting on two frames)

Naturally, there must be a way to hide the “unfinished” screen shifting, because it would look very ugly. Therefore doublebuffering (using 2 screens and switching between them as needed) has to be used.

2.1 Splitting leads to restrictions: 8-directional scrolling

This “splitting” is a good thing but it imposes some restrictions on at which speed you can scroll, and to what directions. Without splitting, you know that it's time to shift the screen data when the hardware scroll register(s) wrap. But with splitting?

Have you taken a look at games such as Turrican or Navy Seals? You'll notice that they only scroll one char-length at a time, never less. They're likely using the “scroll registers centered when idle”-ideology. This means: when not scrolling, the hardware scroll register is either set to 4 or 3, centered on a char. Now, if scrolling happens from right to left at speed of 2 pixels/ frame, the scrolling register will get the following values on successive frames:

```
4 (initial state, first half of screen-memory shifting)
2 (second half of screen shifting, possibly drawing new data)
0 (shifted screen has to appear next frame, so shift color-memory now and
swap
   the doublebuffer screens)
6 (scrolling one char finished, this frame we don't have to do anything)
4 (if scrolling for more than one char, the loop starts again. Otherwise,
```

```
stop
   here)
```

What if we're scrolling from left to right? If we'd be starting from 4 again we'd hit the “wrap point” one frame earlier and we'd not be ready yet. Therefore, we must first reset the scroll register to 3 (this is barely noticeable, so it isn't a bad thing.) The scroll register values on successive frames will now be:

```
3 (initial state, first half of screen shifting)
5 (second half of screen shifting, possibly drawing new data)
7 (color-screen shifting, swap screens)
1 (don't have to do anything)
3 (loop starts again if necessary)
```

This method gives the possibility to 8-directional scrolling, but not freedirectional.

2.2 Freedirectional scrolling

This is a method I found out myself and don't know if it has been actually used in any games. Possibly Chuck Rock or X-Out, they seem to allow quite free direction in scrolling.

Here the scrolling doesn't need to be “centered” when idle and any speeds up to 4 pixels/frame can be used. As a consequence, the screen & color memory shifting can be split on two frames only, so it'll be more CPU-intensive.

The idea is to have two kinds of frames:

- The first frame we'll add the scrolling speed to the hardware scroll registers. If they'd wrap, just limit them to the end of the 0-7 range. Now we also precalculate the hardware scroll values for the next frame, using the same scrolling speed. This time we allow wrapping. If wrapping does not happen, we don't need to shift screen data and never get to the “second” kind of frame. If wrapping happens, we shift the screen memory in the hidden screen and draw new data to the sides.
- The second frame we'll shift the color memory and finally swap the doublebuffer screens & put the precalculated hardware scroll values into use.

In fact, a much simpler approach than the 8-directional when you understand it. For an example, see the Freedirectional Scrolling test program.

2.3 Shifting the screen-memory data

In fact doublebuffering also makes it easier to code the screen shifting. If one is working only on one screen one must be careful in what direction the shifting loops should go, to not wipe out the entire screen data. Not so with doublebuffering, because the screen data is always being copied from the currently visible screen to the other, hidden screen. If one uses the X register index for the source and Y for the target, all 8 directions can be achieved by simply adjusting the initial index register values. (Naturally one needs two actual shifting loops, one for screen1→screen2 and one for screen2→screen1)

2.4 Drawing new data on the sides

In addition to shifting the screen, new data must be drawn on the side(s) of the screen. Practically, one must keep track of the screen top-left edge's position in block-coordinates (amount of blocks measured from the map's top-left edge) and also the position within a block.

Here's an example of what the screen will look like when scrolled to the left: In this example there's three kinds of blocks: consisting of 1,2 & 3 characters (quite unimaginative)

```
Before:
11112222333311112222333311112222333 3
11112222333311112222333311112222333 3
11112222333311112222333311112222333 3
11112222333311112222333311112222333 3
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
33331111222233331111222233331111222 2
33331111222233331111222233331111222 2
33331111222233331111222233331111222 2
33331111222233331111222233331111222 2
```

```
After:
11122223333111122223333111122223333 1
11122223333111122223333111122223333 1
11122223333111122223333111122223333 1
11122223333111122223333111122223333 1
22233331111222233331111222233331111 2
22233331111222233331111222233331111 2
22233331111222233331111222233331111 2
22233331111222233331111222233331111 2
33311112222333311112222333311112222 3
33311112222333311112222333311112222 3
33311112222333311112222333311112222 3
33311112222333311112222333311112222 3
```

Basically, one must just think what data should appear to the side, it requires some brain gymnastics but there's no short cut to it. When you got one direction done the rest are quite easy. In multidirectional scrolling the difficulty is that you must take into account both the X & Y-position within the block, the previous example could for example look like this as well:

```
Before:
11112222333311112222333311112222333 3
11112222333311112222333311112222333 3
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
22223333111122223333111122223333111 1
33331111222233331111222233331111222 2
```

```
3333111122223333111122223333111122222
3333111122223333111122223333111122222
3333111122223333111122223333111122222
1111222233331111222233331111222233333
1111222233331111222233331111222233333
```

```
After:
1112222333311112222333311112222333331
1112222333311112222333311112222333331
2223333111122223333111122223333111112
2223333111122223333111122223333111112
2223333111122223333111122223333111112
2223333111122223333111122223333111112
3331111222233331111222233331111222223
3331111222233331111222233331111222223
3331111222233331111222233331111222223
3331111222233331111222233331111222223
1112222333311112222333311112222333331
1112222333311112222333311112222333331
```

2.5 Shifting the color-memory data

The color memory is quite ugly to work with because it can't be doublebuffered. So, updating it must be done when the portion being updated isn't being displayed. If you don't care about NTSC compatibility, you'll have enough time to shift about 20-21 lines around during the time it's not being displayed. But having NTSC compatibility included isn't so much harder: you simply have to split the color screen update in two:

- Already when the lower half of the screen is being displayed, you can start shifting the upper half.
- Having done that, see that the game screen displaying has ended (probably it has, but if it's a SuperCPU equipped machine you can't be sure)
- Then shift the lower half of the color memory.

Care must be taken when shifting the colors from top to bottom - the row at the split point must be buffered into a separate memory location at first (how to do it can be seen in Metal Warrior 2 & 3 source code.)

There's also an abominable method of updating the color memory that wastes a lot of time (used in Nobby The Aardwark, Darkman and Cool World at least) <ocde>

```
      ldy screen,x
      lda charcolortable,y
      sta colormemory,x
      ldy screen+40,x
      lda charcolortable,y
      sta colormemory+40,x
      </code>
```

It is wasting 4 cycles each byte, compared to plain shifting! I strongly recommend not to use this method, although it's very easy to use. In fact it's so slow that you can forget about NTSC compatibility when using it.

If you don't have a separate char color for each char but rather for each block only (like SEUCK), the color screen update can be tremendously optimized. Now consider 1,2 and 3 as different colors and look what only really changes when we scroll:

```
1111222233331111222233331111222233331    1112222333311112222333311112222333311
1111222233331111222233331111222233331    1112222333311112222333311112222333311
1111222233331111222233331111222233331    1112222333311112222333311112222333311
1111222233331111222233331111222233331    1112222333311112222333311112222333311
2222333311112222333311112222333311111    2223333111122223333111122223333111112
2222333311112222333311112222333311111 -> 2223333111122223333111122223333111112
2222333311112222333311112222333311111    2223333111122223333111122223333111112
2222333311112222333311112222333311111    2223333111122223333111122223333111112
3333111122223333111122223333111122222    3331111222233331111222233331111222223
3333111122223333111122223333111122222    3331111222233331111222233331111222223
3333111122223333111122223333111122222    3331111222233331111222233331111222223
3333111122223333111122223333111122222    3331111222233331111222233331111222223
```

```
Only every 4th column changes:
   2   3   1   2   3   1   2   3   1
   2   3   1   2   3   1   2   3   1
   2   3   1   2   3   1   2   3   1
   2   3   1   2   3   1   2   3   1
   3   1   2   3   1   2   3   1   2
   3   1   2   3   1   2   3   1   2
   3   1   2   3   1   2   3   1   2
   3   1   2   3   1   2   3   1   2
   1   2   3   1   2   3   1   2   3
   1   2   3   1   2   3   1   2   3
   1   2   3   1   2   3   1   2   3
   1   2   3   1   2   3   1   2   3
```

The knowledge of this can cut color memory accesses to 1/4th portion. See Metal Warrior 1 source code for an example, or reverse-engineer some game where you see colors changing rarely - it's likely to be doing this.

3. Coordinate systems

How do you represent the location of the objects onscreen? If you really care only about things on the screen, you could use the screen-coordinate system that the sprites use. This method is obviously best for speed, but has its drawbacks.

Look at Turrican, when the “walker” enemies move near the boundaries of the visible screen. They sometimes behave very erratically. Turrican is a master- piece otherwise so this can be forgiven, but I wouldn't tolerate this in my own programs…

Why are they behaving so? Because they're only using the screen coordinate system, and checking only the characters on screen for background collisions. In some cases checking outside the screen would happen and the collision routine probably has a hardcoded response of either “obstacle” or “free space” in that case. This is fast, but honestly, incorrect.

Therefore, for all serious multidirectional scrolling projects, I recommend using exclusively a world-coordinate system for all objects in the game world and checking background collisions the “hard way”: by examining the map & block data (that could probably be optimized to check on the screen when possible, but I haven't done that...)

How does one represent world coordinates? Metal Warrior 1 & 2 simply measured pixels from the top-left edge of the game map (as 16bit values). This worked great, to get the position of sprites on screen a value was subtracted based on the scrolling position. But honestly, the background collisions were slow and complicated and there was a complete lack of subpixel accuracy, making all accelerating movement look rough.

Metal Warrior 3 and BOFH use another way to represent world coordinates: the highbyte of the coordinate represents position as blocks and the lowbyte of the coordinate represents position within the block. A block has 4 chars = 32 pixels, so as a bonus we get 3 bits of subpixel accuracy. Now background collision checking becomes very fast & easy to do, because the map position is directly the coordinate highbyte.

The downside is that the sprite displaying and sprite-sprite collision checks need now some bit rotates to get rid of the subpixel accuracy, so they become a bit slower.

4. How to handle the huge amount of objects in a "game world"?

Naturally, a C64 game cannot process tens or hundreds of objects each frame, instead it must concentrate on the things on the visible part or near them. So, there must be a way to distinguish between these inactive (far) and active (near) objects.

I call the active objects “actors” and the inactive objects are part of what I call “leveldata”, that basically stores their positions and actor types, possibly hitpoints, but not any other information.

Each frame I check a part of the leveldata (for example 16 objects). If they fit to the rectangle of the visible screen (or actually a bit larger rectangle) they are put on the screen as new “actors” and removed from the leveldata. I also check that if the active objects go outside that rectangle, they are removed from the screen (from the “actor” table) and put back to the leveldata. In some special cases objects are never removed (the player actor!)

Another advantage of the latter world-coordinate system I described is that now putting objects onscreen becomes very easy. In the leveldata I usually store the objects' locations only at block accuracy. So, checking which objects to put onscreen and which to remove becomes just a matter of checking the coordinate highbytes.

5. Conclusion

This was a quite short and not very in-depth rant. I hope, however, that it shed light on some things about multidirectional scrolling and making a “game world”. These things are among the most complicated things in C64 games, only 3D worlds come to mind as more complex. I pretty much worship C64 games containing such elements (provided that they're also playable otherwise) and therefore I've dedicated most of my C64-related energy into making such games myself.

Lasse Öörni
loorni@student.oulu.fi