a strategy for flicker-free heterogeneous multi-cell icon animation across a non-blank text-screen

~ an all original algorithm by voidstar ~

(thinking it’s 1979 all over again here in April 2021)

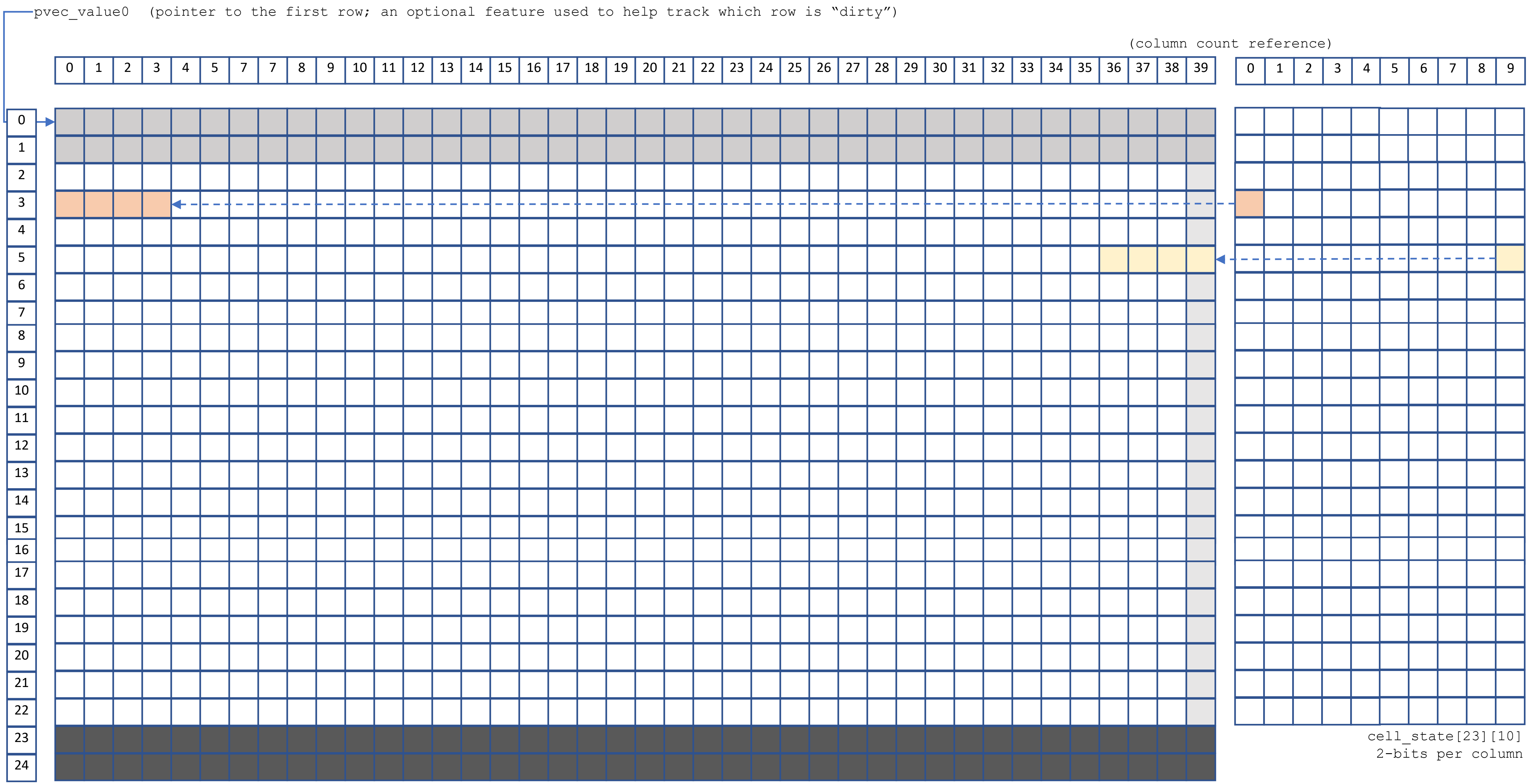
Heterogeneous means the icon might have holes/transparent portions, as opposed to being a solid homogeneous block.

Multi-cell means more than 1 text cell in length and/or width. If the icon is just a single cell, then that is a no-brainer exercise to animate.

Non-blank means the screen may have a content on the background (a game map, for instance), i.e. content other than it’s default clrscr() state (including perhaps a stack of other heterogeneous multi-cell icons).

contact.steve.usa@gmail.com

REV.3

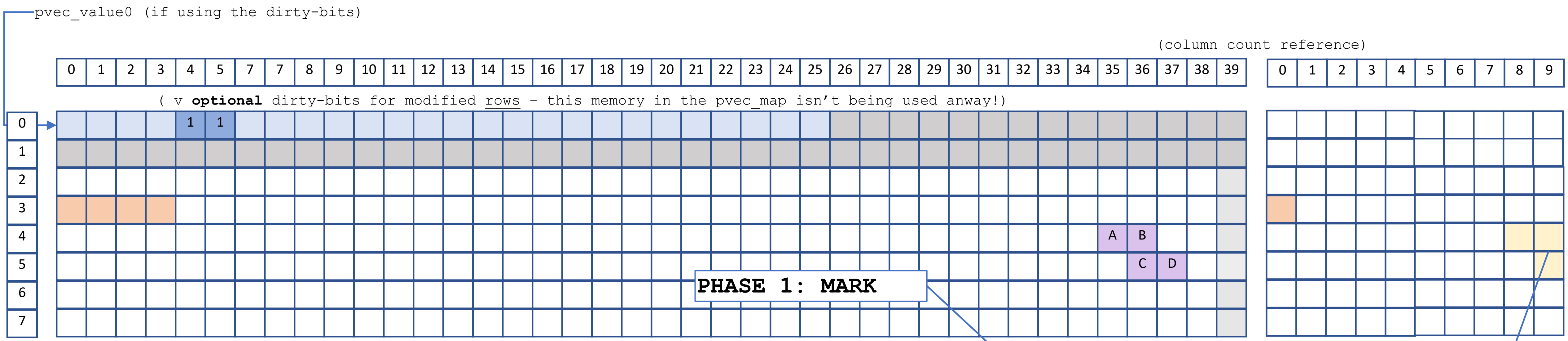


pvec_value0 (pointer to the first row; an optional feature used to help track which row is "dirty")

(column count reference)

cell_state[23][10]
2-bits per column

pvec_map[23][38]
full copy of the intended original background
(not using **row 23 and 24** and not using **col 39** for the map)



pvec_value0 (if using the dirty-bits)

(column count reference)

(v **optional** dirty-bits for modified rows – this memory in the pvec_map isn't being used anway!)

PHASE 1: MARK

pvec_map

cell_state

Loop across the size of the icon,
do some work to figure out the relative x/y offset..
e.g. want to place 3x2 size icon at **35,4**

Game logic happens here.. Initial location of object is set to be at 35,4

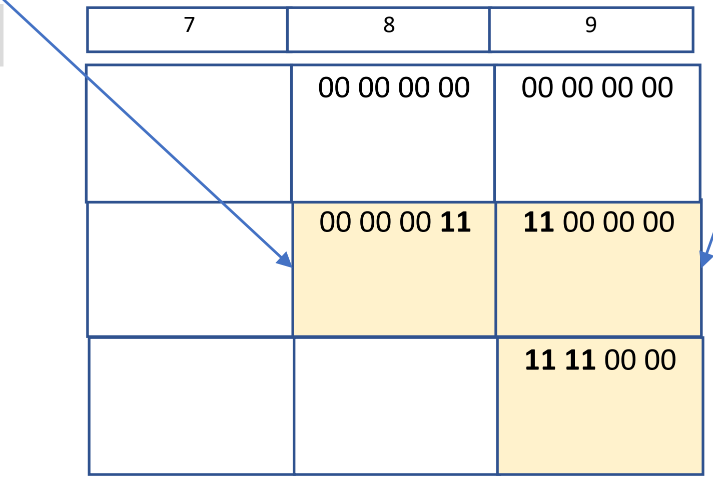
X	Y		
35	4	→	A
36	4	→	B
36	5	→	C
37	5	→	D

$8000+(40*y)+x
32768+(40*y)+x

32963	A
32964	B
33004	C
33004	D

locations_to_draw

```
static unsigned char or_equal_modifier[] = {0xC0,0x30,0x0C,0x03};
//  0xC0 == 1100 0000    (value "3" at first "half nibble" *)
//  0x30 == 0011 0000    (value "3" at second "half nibble" *)
//  0x0C == 0000 1100    (value "3" at third "half nibble" *)
//  0x03 == 0000 0011    (value "3" at fourth "half nibble" *)
…
div4_table is (0 to 40) / 4 stored sequentially in an array
mod4_table is (0 to 40) % 4 stored sequentially in an array
…
#define MARK_LOCATION_AS_DRAWN(data_x,data_y) \
    ptr_pvec_value0[data_y] = TRUE; \
    cell_state[data_y][div4_table[data_x]] |=
      or_equal_modifier[mod4_table[data_x]];
```



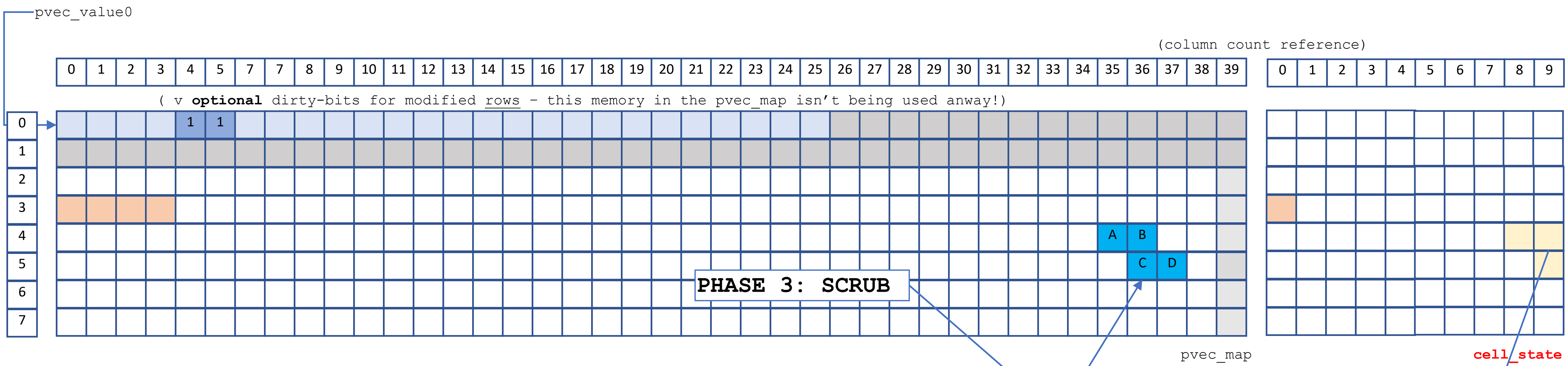
7	8	9
	00 00 00 00	00 00 00 00
	00 00 00 **11**	**11** 00 00 00
		11 11 00 00

We don't draw these immediately. We first apply the process described in the next slide. If you draw them now, you end up with a slight "shadow-sliding" effect (which could be useful for ghost-type icons).

* Could use "10"(2) instead of "11" (3). Either way, one value is "wasted." But don't use "01" or "00", those mean something in the next step.

PHASE 2: QUEUE

```
#define BUFFER_LOCATION_TO_DRAW(data_x, data_y, target_symbol) \
  locations_to_draw[num_locations_to_draw].symbol = target_symbol;  \
  locations_to_draw[num_locations_to_draw].offset = 0x8000+(40*data_y)+data_x;  \
  ++num_locations_to_draw;
```



(v **optional** dirty-bits for modified rows – this memory in the pvec_map isn't being used anway!)

$8000+(40*y)+x
32768+(40*y)+x

32963	A
32964	B
33004	C
33004	D

locations_to_draw

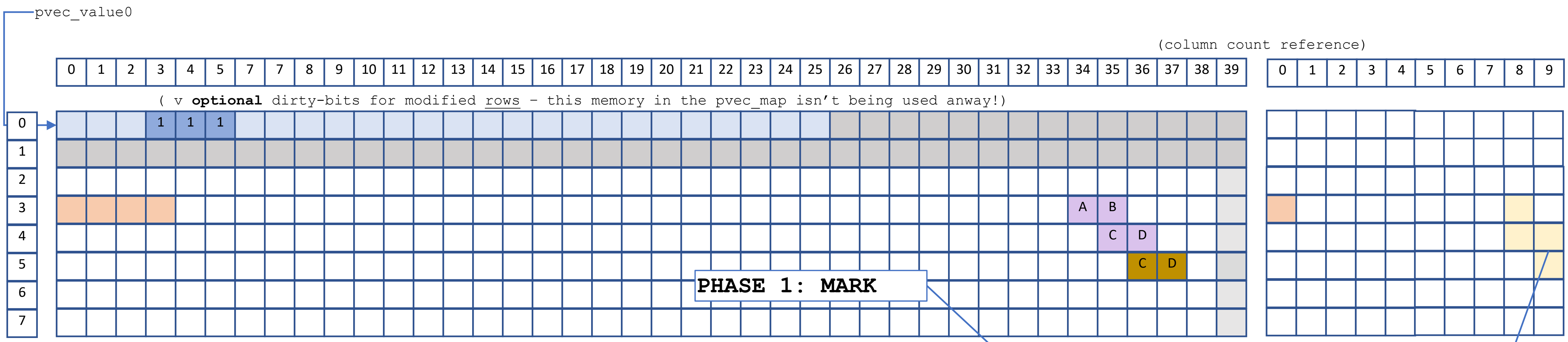
```
for (temp_y = 2; temp_y < MAX_MAP_ROWS; ++temp_y) {
  if (ptr_pvec_value0[temp_y] == TRUE) {  // something was drawn on this row
    ptr_value = VEC_GET(pvec_map, temp_y);  // ptr to this row, be ready to restore
    for (temp_x = 0; temp_x < NUM_CELL_STATES; ++temp_x) {  // NUM_CELL_STATES=10
      i = cell_state[temp_y][temp_x];
      if (i == 0) {
        // nothing to do, no animation impacts this frame
      } else {
        if ((i & 0xC0) == 0xC0) { i &= 0x3F; i |= 0x40; }  // “11” -> “01”
        else if ((i & 0XC0) == 0x40) {  // restore and do 1→0 for this half-nibble
          g_i = (4*temp_x) + 0;  // +0 is offset (this is “cell+0”)
          ch = ptr_value[ g_i ];
          TRANSLATE_MAP_SYMBOL(ch, &symbol);
          POKE(BASE_SCREEN_ADDRESS + (MAX_BOARD_WIDTH*temp_y)+g_i, symbol);
          i &= 0x3F;  //<  reduce cell_1_state from 1 to 0 (important) 0011 1111
        }
        // REPEAT FOR CELL+1    0x30      &= 0xCF     |= 0x10
        // REPEAT FOR CELL+2    0x0C      &= 0xF3     |= 0x04
        // REPEAT FOR CELL+3    0x03      &= 0xFC     |= 0x01
        cell_state[temp_y][temp_x] = i;
      }
    }
  }
}
```

PHASE 4: COMMIT

NOW draw all the buffered locations_to_draw…
POKE them all onto the screen. And set the location N count back to 0.

7	8	9
	00 00 00 00	00 00 00 00
	00 00 00 **01**	**01** 00 00 00
		01 **01** 00 00

During the next iteration:
IF the icon stays idle, these same bits go back from “01” to “11”
IF the icon moves, only the overlapping portions go from “01” to “11”
New locations entirely will also go to “11”



PHASE 1: MARK

Loop across the size of the icon,
do some work to figure out the relative x/y offset..
e.g. want to place 3x2 size icon at **34,3**

Game logic happens here.. Player indicated to go UP and LEFT… MARK_LOCATION_AS_DRAWN is used on all the non-empty icon cells.

$8000+(40*y)+x
32768+(40*y)+x

X	Y		
34	3	→	A
35	3	→	B
35	4	→	C
36	4	→	D

32922	A
32923	B
32963	C
32964	D

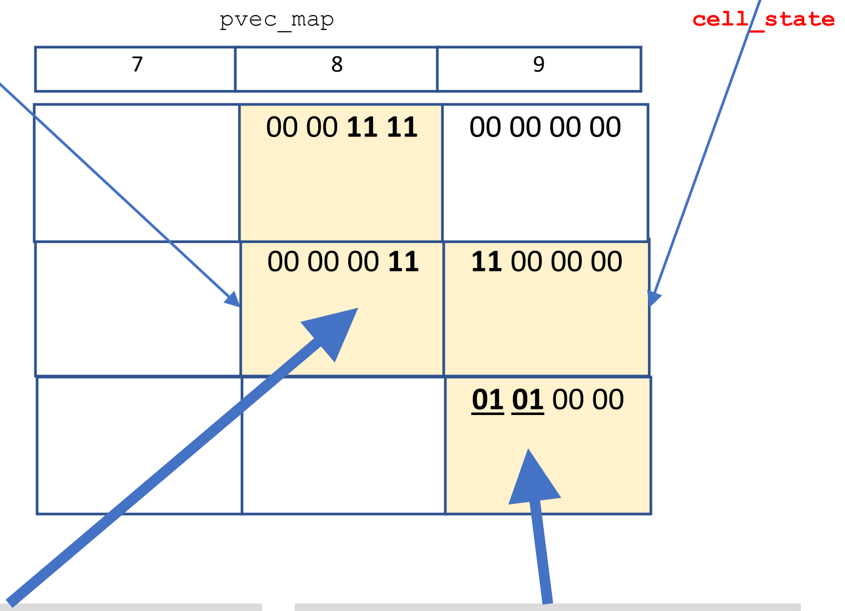
locations_to_draw

```
static unsigned char or_equal_modifier[] = {0xC0,0x30,0x0C,0x03};
//  0xC0 == 1100 0000   (value "3" at first "half nibble")
//  0x30 == 0011 0000   (value "3" at second "half nibble")
//  0x0C == 0000 1100   (value "3" at third "half nibble")
//  0x03 == 0000 0011   (value "3" at fourth "half nibble")
…
div4_table is (0 to 40) / 4 stored sequentially in an array
mod4_table is (0 to 40) % 4 stored sequentially in an array
…
#define MARK_LOCATION_AS_DRAWN(data_x,data_y) \
    encode_x = div4_table[data_x]; \
    ptr_pvec_value0[data_y] = TRUE; \
    cell_state[data_y][encode_x] |= or_equal_modifier[mod4_table[data_x]];
```

We don't draw these immediately. We first apply the process described in the next slide.

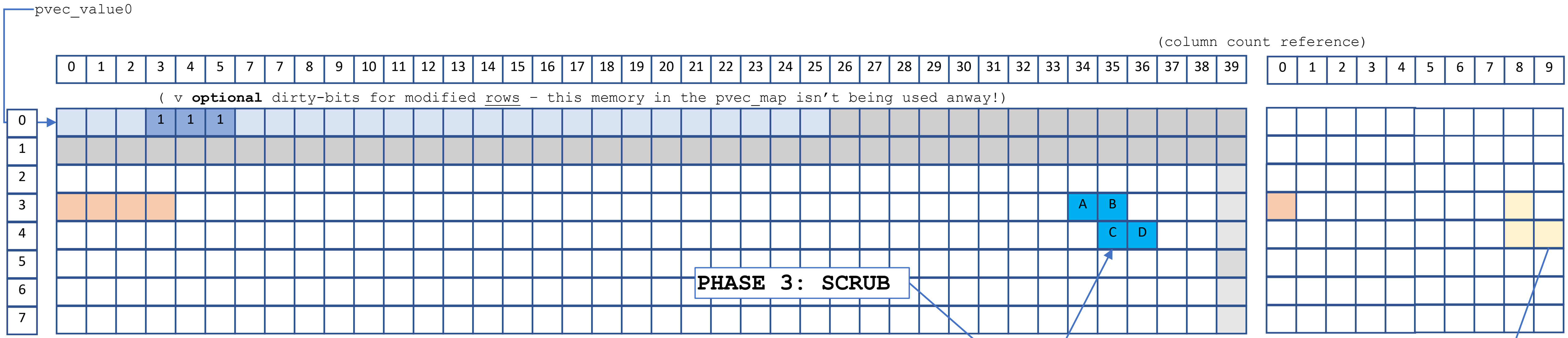
PHASE 2: QUEUE

```
#define BUFFER_LOCATION_TO_DRAW(data_x, data_y, target_symbol) \
  locations_to_draw[num_locations_to_draw].symbol = target_symbol;  \
  locations_to_draw[num_locations_to_draw].offset = 0x8000+(40*data_y)+data_x;  \
  ++num_locations_to_draw;
```



These stayed at "11", indicating to DON'T refresh back to the map background piece, since something else is about to be drawn here (something that's queued into the locations_to_draw!).

These stayed at "01" ! "01" indicates this cell had been drawn on in the past, but it wasn't marked as being refreshed. So this cells needs to get repainted with the original background cell.



(v **optional** dirty-bits for modified rows – this memory in the pvec_map isn't being used anway!)

PHASE 3: SCRUB

pvec_map

cell_state

7	8	9
	00 00 01 01	00 00 00 00
	00 00 00 01	01 00 00 00
		00 00 00 00

$8000+(40*y)+x
32768+(40*y)+x

32922	A
32923	B
32963	C
32964	D

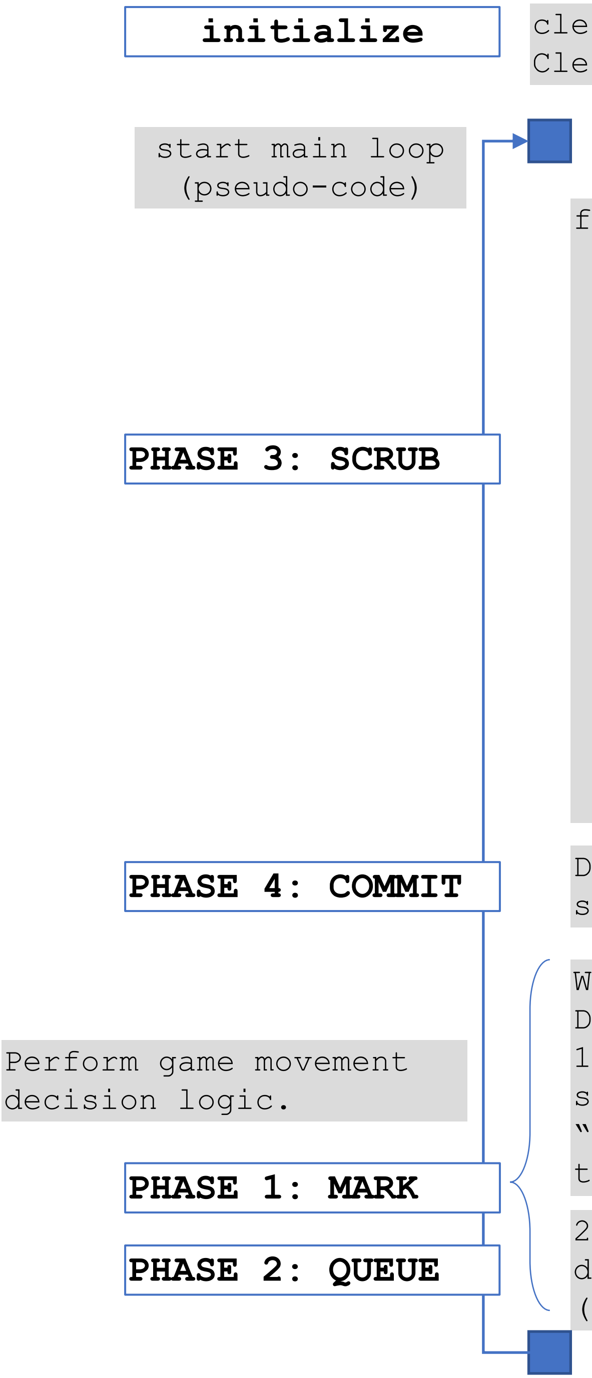
locations_to_draw

```
for (temp_y = 2; temp_y < MAX_MAP_ROWS; ++temp_y) {
  if (ptr_pvec_value0[temp_y] == TRUE) {  // something was drawn on this row
    ptr_value = VEC_GET(pvec_map, temp_y); // ptr to this row, be ready to restore
    for (temp_x = 0; temp_x < NUM_CELL_STATES; ++temp_x) {  // NUM_CELL_STATES=10
      i = cell_state[temp_y][temp_x];
      if (i == 0) {
        // nothing to do, no animation impacts this frame
      } else {
        if ((i & 0xC0) == 0xC0) { i &= 0x3F; i |= 0x40; }  // “11” -> “01”
        else if ((i & 0XC0) == 0x40) {  // restore and do 1→0 for this half-nibble
          g_i = (4*temp_x) + 0;  // +0 is offset (this is “cell+0”)
          ch = ptr_value[ g_i ];
          TRANSLATE_MAP_SYMBOL(ch, &symbol);
          POKE(BASE_SCREEN_ADDRESS + (MAX_BOARD_WIDTH*temp_y)+g_i, symbol);
          i &= 0x3F;  //<  reduce cell_1_state from 1 to 0 (important) 0011 1111
        }
        // REPEAT FOR CELL+1   0x30      &= 0xCF      |= 0x10
        // REPEAT FOR CELL+2   0x0C      &= 0xF3      |= 0x04
        // REPEAT FOR CELL+3   0x03      &= 0xFC      |= 0x01
        cell_state[temp_y][temp_x] = i;
      }
    }
  }
}
```

PHASE 4: COMMIT

During the next iteration:
IF the icon stays idle, these same bits go back from “01” to “11”
IF the icon moves, only the overlapping portions go to “01” to “11”
New locations entirely will also go to “11”

NOW draw all the buffered locations_to_draw…
POKE them all onto the screen. And set the location N count back to 0.



initialize

```
clear cell_states
Clear locations_to_draw
```

A little improvement gained if you also maintain a dirty_flag for each modified row. Optional, since there are pros and cons on doing so.

start main loop (pseudo-code)

PHASE 3: SCRUB

```
for each row y                [2 to 22, inclusive]
  ptr = pvec_map[curr_y]
  for each cell_state col x  [0..9, inclusive]
    mask = cell_state[y]x]
    if mask == 0
       skip the 4 columns represented in this cell_state, no change flagged
    else {
       for each ofsX { [ofs0, ofs1, ofs2, ofs3 …]   (4 sets of 2-bits within this byte)
         if mask @ ofsX == “11” then set mask @ ofsX == “01”
         else if mask @ ofsX == “01” then {
           temp_x = (4*x)+ofsX
           get background map symbol at y,temp_x
           poke screen+width*y+temp_x = symbol
           set mask @ ofsX == “00”
       }
       cell_state[y][x] = mask
```

(the key here is to use bitmask – no need to shift bits at runtime)

PHASE 4: COMMIT

```
Draw all the locations_to_draws
set locations_to_draw count to 0
```

Perform game movement decision logic.

When logic decides to WANT to draw something….
Do two things:
1) Use **MARK_LOCATION_AS_DRAWN(data_x,data_y)** to set the appropriate 2-bit pair to the value of “11” indicating that it is to be drawn during this frame.

PHASE 1: MARK

set modified cell_states to 11’s
(the key here is to use the div4, mod4, and or_equals tables!!!)
(unless the host DIV and MOD operators are very efficient…)

PHASE 2: QUEUE

2) Queue up the location and symbol to be drawn, to use in the next main loop iteration (locations_to_draw)

add to locations_to_draw
(actually only necessary if the icon has actually moved from its previous position – but detecting that change may equal the cost of just adding it to the list… so, it depends)

Main Process Detail Descriptions 1/2

- MARK
  - Use a value of 3 (or 2) at the appropriate 2-bit offsets corresponds to a cell that has been modified. i.e. mark that cell as being drawn on during this frame of the animation. The reason the value 3 or 2 can be used is because that's what fits within 2-bits, where the other two values 1 and 0 are reserved for other use.
    - DIV and MOD operator resulted in many instructions using CC65/6502 processor (i.e. slow to execute). You might be able to hand assemble a better option, or maybe it's more efficient in other compilers or processors. But in this case for the 6502, I found it far more efficient to sacrifice 80 bytes of memory to statically reserve two 40-byte "MOD" and "DIV" tables (plus a 4-byte "or_equal_modifier").
    - You could use these "unused bit values" to add a shadow or "ghosting" effect to your icons, that would let the shape linger on the display an additional frame or so (add an extra bit for even longer ghosting effect). Described further in the SCRUB step.
- QUEUE
  - Queue the location and content of the cell that is being drawn on (i.e. in a vector or array in memory). I use an array of three bytes: store a direct pointer to the screen memory (2-byte address) and the character code that is to be drawn (1 byte). This DOES occupy some memory – if not using the full extent of your screen map (pvec_map in my case), you could stuff it in there (e.g. if not using the dirty bits at pvec_value0, I could queue this data in the first two rows of the map – but this limits the queue to about 13 cells being updated within a frame).
    - However the queue is implemented, keep this statically reserved, no time to calloc/free this every frame.

Main Process Detail Descriptions 2/2

- SCRUB
  - The term “scrub” here is to scrub every non-zero value in your cell_states array down by 1 point. But it’s a little bit more then that…
  - Quickly scan the cell_states and anything that is currently >1, “scrub” the value back down by one increment of 1 (using bit mask or subtraction, whichever is less opcodes). **Else** if the cell_state value is already 1 (from a scrub during the last frame), this indicates it wasn’t marked as drawn on during this frame, so then do two things: “scrub” the value down from 1 to 0, AND restore the cell content back to the underlining background (i.e. that portion of the icon can be scrubbed off the display).
  - By “quickly scan” that means a few things:
    - if the entire byte is already 0, you can skip these 4 text cells and move to the next group of 4 (the processor may be more efficient at quickly examining if the entire byte is zero – then again, it might not be, and we have to examine each bit anyway).
    - If the “dirty bit” representing the entire row is 0, nothing on this row has been drawn on, and you can skip the entire row (if using the dirty-bit option, which does take some memory – it also means the approach slows down as more rows start to be drawn on, which can result in an inconsistent performance; I use the dirty-bit in Destiny Hunter, and preferred the effect of the monsters being faster at the very beginning of the stage, it makes their initial appearance more “shocking”)
    - As mentioned in the MARK phase, you can mark things with values >1 and let portions of the icon linger slightly longer on the display. This could be used for something like a “snail” or “ghost” type icon, or anything where you might want to leave a slight trail. With only 2-bits, you just have 1 extra frame; for a longer effect, you’d need 3-bits, etc.
  - Restoring any portion of the underlining background typically does mean having that full background resident in memory, ready to go. This also means you need to convert the “half-nibble” being scrubbed back into a normal x-y screen coordinate. To find “real” column x by: multiply the cell_state index being iterated by 4 (or bit-shift << 2 if that is faster), PLUS add the half-nibble offset.
- COMMIT
  - Now that we’ve erased (restored to background) *only* the portions that were no longer marked (during the scrub), we’re now ready to actually draw the mark cells that were queued. We commit to drawing them (iterate the queue and POKE them onto the screen).
    - If memory is extremely tight and you can’t really afford a queue/array of locations_to_draw… You can technically ditch that and just draw the marked locations right away. Again, the scrub process is only going to scrub/restore what wasn’t marked eitherway. But the resulting effect, in my opinion, is more smooth if you erase-first, then draw the new content. i.e. if you draw first and then erase, the icon sort of “hops” to the new location – and maybe there is a situation where that ends up being useful.
    - However, more importantly, the policy used here also impacts when you having multiple icons at what happens when they start to overlap – which one ends up “on top”. e.g. A cell_sate that is already marked (value >1), its location doesn’t really need to be queued again (let first icon “win”).
- Experiment and see what works for you, but I found this workflow MARK, QUEUE, SCRUB, COMMIT was more consistent and visually pleasing.

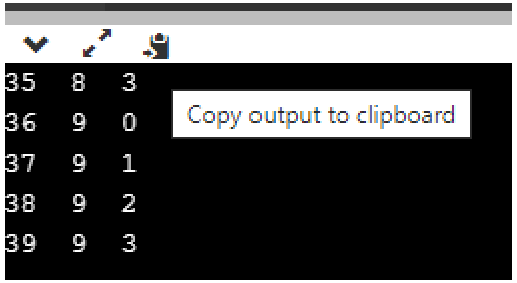
Example Application

- Destiny Hunter
    - Search “Destiny Hunter: original modern game for the vintage Commodore PET (aka the Mini-Adventure of Edwad”
    - https://www.youtube.com/watch?v=Tk7uaLM_mCU
    - See 2:20 minute offset
- Evolution of approach…
    - Smeared (no erase whatsoever)
        - https://www.youtube.com/watch?v=48Qq_ziKjMQ
    - Flickered (no flicker-free approach)
        - https://www.youtube.com/watch?v=RCrBt0mpSb8

Producing the MOD/DIV tables

- Online C compiler
  - https://www.onlinegdb.com/online_c_compiler

    -
      ```
      #include <stdio.h>
      int main() {
        unsigned int i;
        for (i = 0; i < 40; ++i) {
          printf("%2u %2u %2u\n", i, i / 4, i % 4);
        }
        return 0;
      }
      ```
    - "40" in this case is the number of columns of the text screen
      - algorithm can scale to 80 or N columns, as needed



```
35  8  3
36  9  0
37  9  1
38  9  2
39  9  3
```

- Places values into Excel, make your static arrays. Examples:
  - `static unsigned char mod4_table[] = {0,1,2,3,0,1,…, 3,0,1,2,3};`
    - len = sizeof(mod4_table)/sizeof(unsigned char);
  - `static unsigned char div4_table[] = {0,0,0,…, 8,9,9,9,9};`
    - The "…" is notional, not actual C syntax (just to save space for this presentation)

A fun line of code! Wield it carefully.

-
  ```
  #define MARK_LOCATION_AS_DRAWN(data_x,data_y) \
      cell_state[data_y][div4_table[data_x]] |= or_equal_modifier[mod4_table[data_x]];
  ```
- (don’t need encode_x, just access div4 directory)
- (benefit of dirty-bit is somewhat marginal, makes inconsistent performance)
- Could be issues in “how far away” the address of these static tables end up being (relative to the section of code invoking this macro)
  - This gets nit-picky but could make a difference…
  - e.g. if +/- 256 bytes, a single opcode can access using a RELATIVE address instead of needing a 2-byte ABSOLUTE address
  - Compiler may have .near directives to help?
- Obviously no asserts or error checking, so be careful.