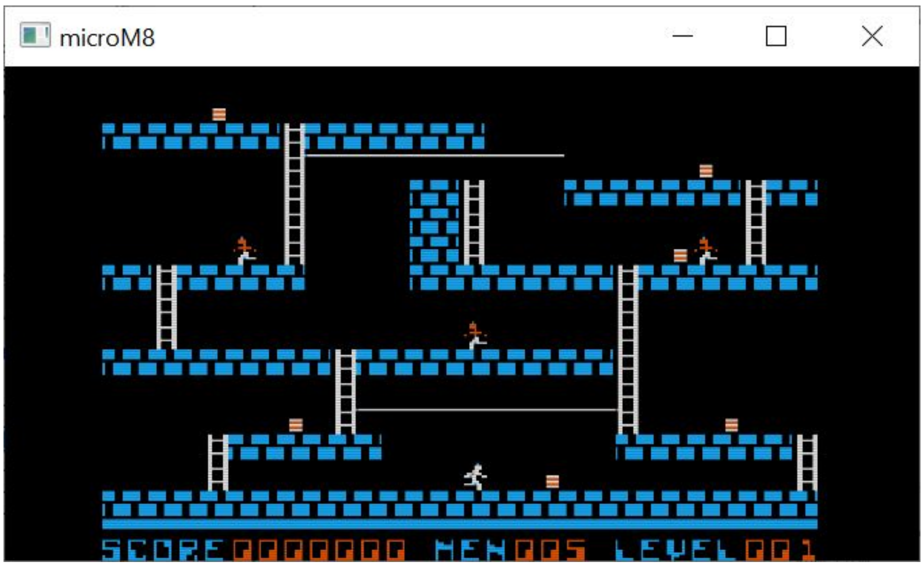
Chapter 1

Lode Runner

Lode Runner was a game originally written in 1982 by Douglas E. Smith (1960–2014) for the Apple II series of computers, and published by Broderbund.

You control the movement of your character, moving left and right along brick and bedrock platforms, climbing ladders, and "monkey-traversing" ropes strung across gaps. The object is to collect all the gold boxes while avoiding being touched by the guards. You can dig holes in brick parts of the floor which can allow you to reach otherwise unreachable caverns, and the holes can also trap the guards for a short while. Holes fill themselves in after a short time period, and if you're in a hole when that happens, you lose a life. However, if a guard is in the hole and the hole fills, the guard disappears and reappears somewhere along the top of the screen.

You get points for collecting boxes and forcing guards to respawn. Once you collect all the boxes, a ladder will appear leading out of the top of the screen. This gets you to the next level, and play continues.



Lode Runner included 150 levels and also a level editor.

Chapter 2

Apple II Graphics

Hi-res graphics on the Apple II is odd. Graphics are memory-mapped, not exactly consecutively, and bits don't always correspond to pixels. Color especially is odd, compared to today's luxurious 32-bit per pixel RGBA.

The Apple II has two hi-res graphics pages, and maps the area from `$2000-$3FFF` to high-res graphics page 1 (HGR1), and `$4000-$5FFF` to page 2 (HGR2).

We have routines to clear these screens.

3 ⟨*defines* 3⟩≡ (71b) 21 ▷

```
      ORG     $0A
TMP_PTR         DS.W    1
```

Defines:
`TMP_PTR`, used in chunks 4 and 24.

4 ⟨*routines 4*⟩≡ (71b) 24▷

```
    ORG     $7A51
CLEAR_HGR1:
    SUBROUTINE

    LDA     #$20                ; Start at $2000
    LDX     #$40                ; End at $4000 (but not including)
    BNE     CLEAR_PAGE          ; Unconditional jump

CLEAR_HGR2:
    SUBROUTINE

    LDA     #$40                ; Start at $4000
    LDX     #$60                ; End at $6000 (but not including)
    ; fallthrough

CLEAR_PAGE:
    STA     TMP_PTR+1           ; Start with the page in A.
    LDA     #$00
    STA     TMP_PTR
    TAY
    LDA     #$80                ; fill byte = 0x80

.loop:
    STA     (TMP_PTR),Y
    INY
    BNE     .loop
    INC     TMP_PTR+1
    CPX     TMP_PTR+1
    BNE     .loop               ; while TMP_PTR != X * 0x100
    RTS
```

Defines:
  CLEAR_HGR1, never used.
  CLEAR_HGR2, never used.
Uses TMP_PTR 3.

2.1 Pixels and their color

First we'll talk about pixels. Nominally, the resolution of the hi-res graphics screen is 280 pixels wide by 192 pixels tall. In the memory map, each row is represented by 40 bytes. The high bit of each byte is not used for pixel data, but is used to control color.

Here are some rules for how these bytes are turned into pixels:

- Pixels are drawn to the screen from byte data least significant bit first. This means that for the first byte bit 0 is column 0, bit 1 is column 1, and so on.
- A pattern of `11` results in two white pixels at the `1` positions.
- A pattern of `010` results at least in a colored pixel at the `1` position.
- A pattern of `101` results at least in a colored pixel at the `0` position.
- So, a pattern of `01010` results in at least three consecutive colored pixels starting from the first `1` to the last `1`. The last `0` bit would also be colored if followed by a `1`.
- Likewise, a pattern of `11011` results in two white pixels, a colored pixel, and then two more white pixels.
- The color of a `010` pixel depends on the column that the `1` falls on, and also whether the high bit of its byte was set or not.
- The color of a `11011` pixel depends on the column that the `0` falls on, and also whether the high bit of its byte was set or not.

	Odd	Even
High bit clear	Green	Violet
High bit set	Orange	Blue

The implication is that you can only select one pair of colors per byte.

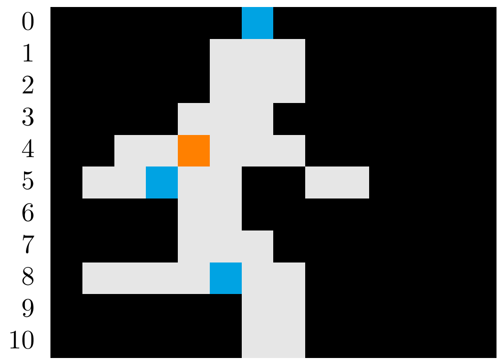
An example would probably be good here. We will take one of the sprites from the game.

Bytes	Bits	Pixel Data
00 00	0000000 0000000	00000000000000
00 00	0000000 0000000	00000000000000
00 00	0000000 0000000	00000000000000
55 00	1010101 0000000	10101010000000
41 00	1000001 0000000	10000010000000
01 00	0000001 0000000	10000000000000
55 00	1010101 0000000	10101010000000
50 00	1010000 0000000	00001010000000
50 00	1010000 0000000	00001010000000
51 00	1010001 0000000	10001010000000
55 00	1010101 0000000	10101010000000

The game automatically sets the high bit of each byte, so we know we're going to see orange and blue. Assuming that the following bits are all zero, and we place the sprite starting at column 0, we should see this:

Here is a more complex sprite:

Bytes	Bits	Pixel Data
40 00	1000000 0000000	00000010000000
60 01	1100000 0000001	00000111000000
60 01	1100000 0000001	00000111000000
70 00	1110000 0000000	00001110000000
6C 01	1101100 0000001	00110111000000
36 06	0110110 0000110	01101100110000
30 00	0110000 0000000	00001100000000
70 00	1110000 0000000	00001110000000
5E 01	1011110 0000001	01111011000000
40 01	1000000 0000001	00000011000000
40 01	1000000 0000001	00000011000000



Take note of the orange and blue pixels. All the patterns noted in the rules above are used.

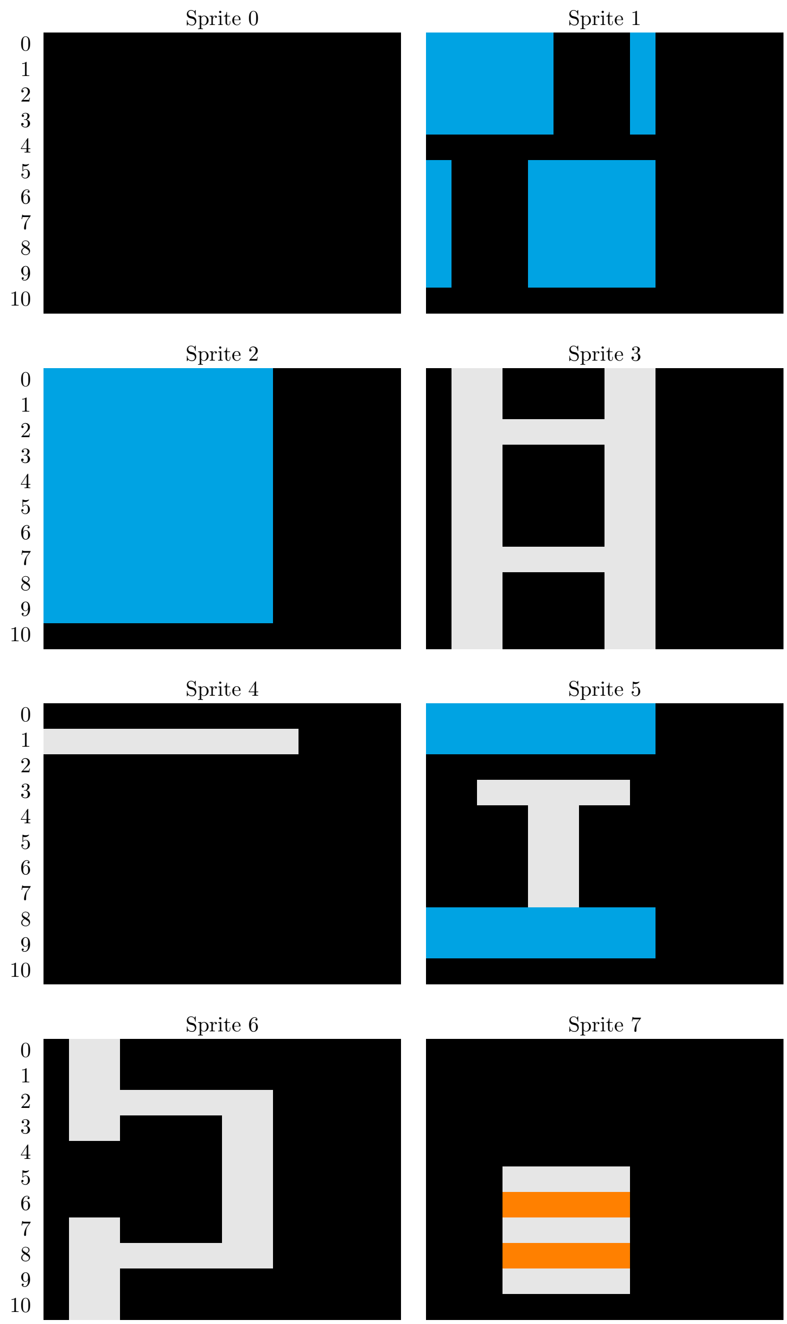
2.2 The sprites

Lode Runner defines 104 sprites, each being 11 rows, with two bytes per row. The first bytes of all 104 sprites are in the table first, then the second bytes, then the third bytes, and so on. Later we will see that only the leftmost 10 pixels out of the 14-pixel description is used.

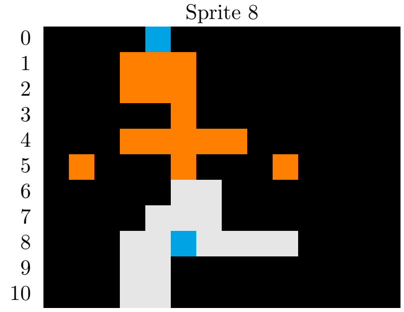
7 ⟨*tables* 7⟩≡ (71b) 22 ▷

```
    ORG     $AD00
SPRITE_DATA:
    INCLUDE "sprite_data.asm"
```

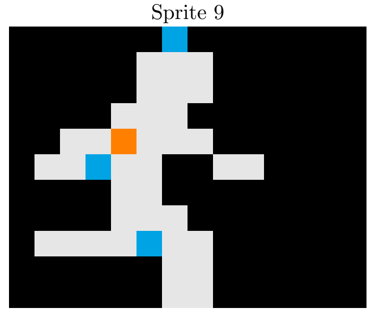
Defines:
SPRITE_DATA, used in chunk 24.


Sprite 0
Sprite 1
Sprite 2
Sprite 3
Sprite 4
Sprite 5
Sprite 6
Sprite 7

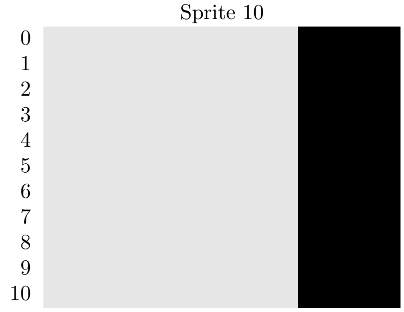
Sprite 8



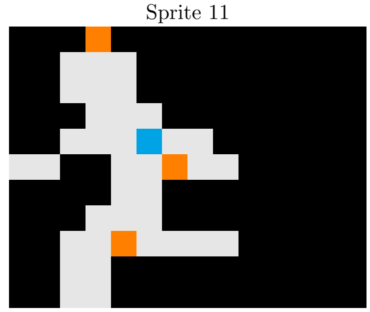
Sprite 9



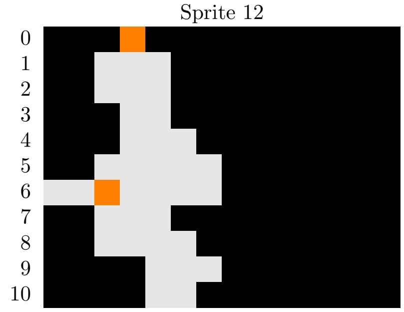
Sprite 10



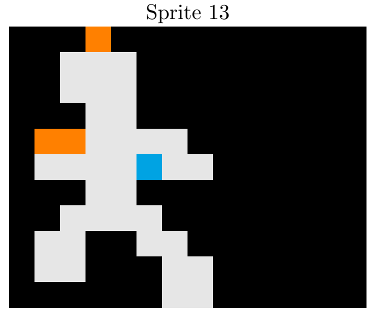
Sprite 11



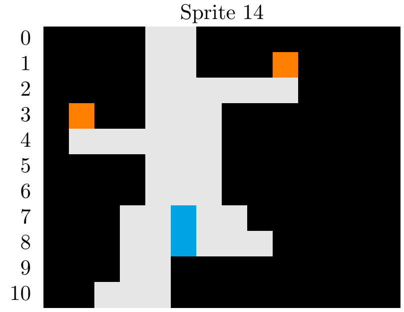
Sprite 12



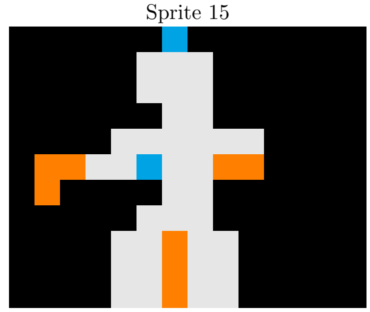
Sprite 13



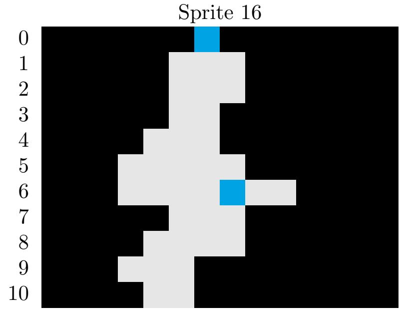
Sprite 14



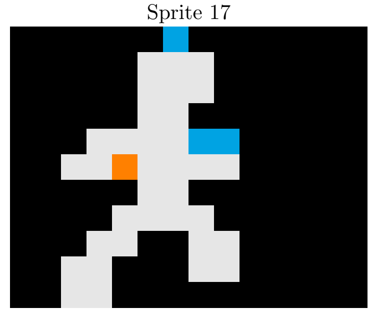
Sprite 15



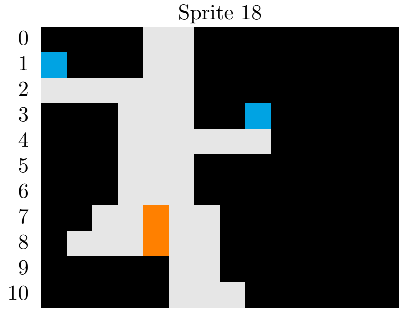
Sprite 16



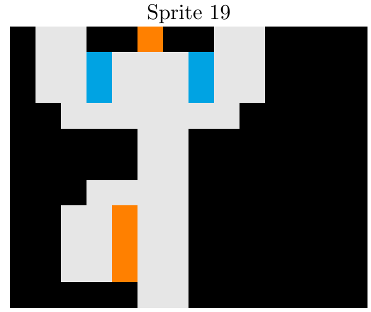
Sprite 17



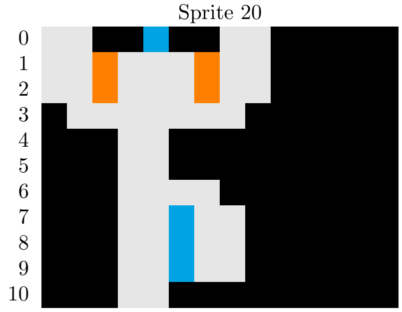
Sprite 18



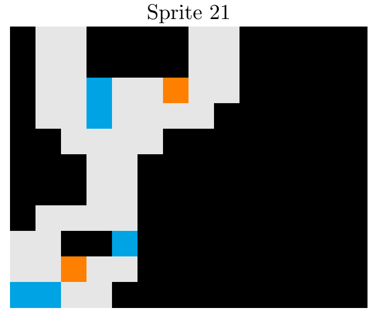
Sprite 19



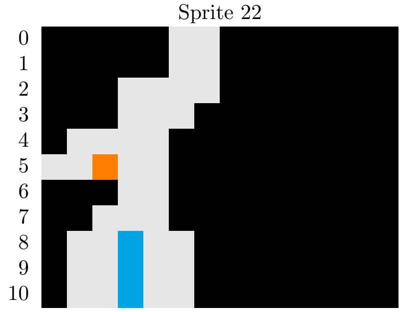
Sprite 20



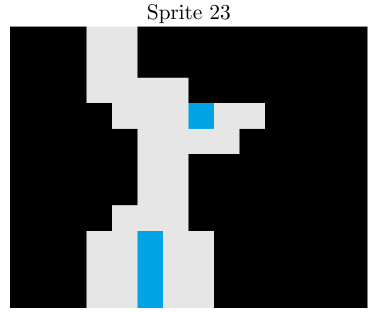
Sprite 21



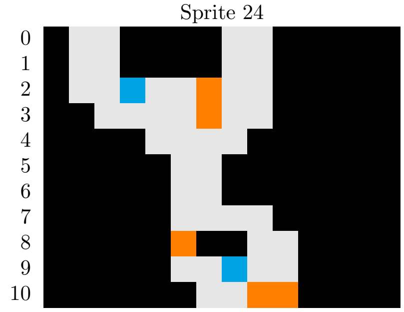
Sprite 22



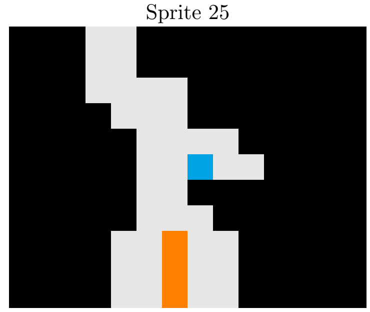
Sprite 23



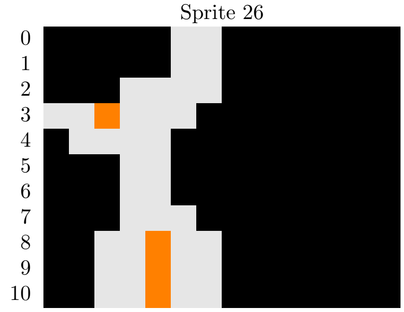
Sprite 24



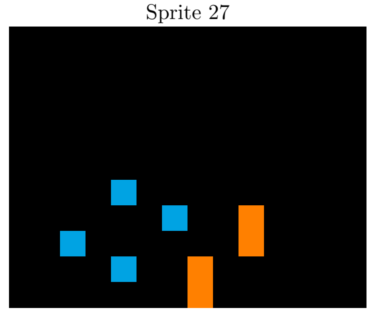
Sprite 25



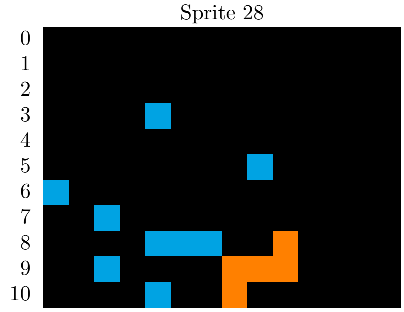
Sprite 26



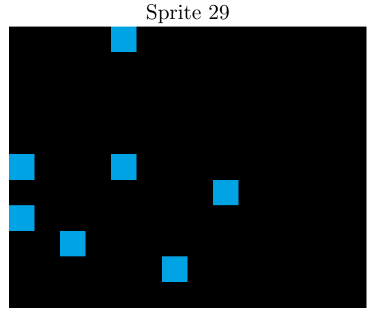
Sprite 27



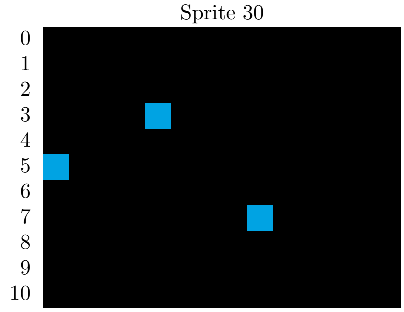
Sprite 28



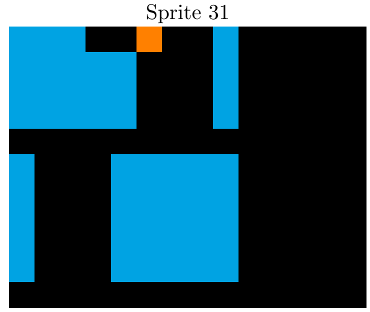
Sprite 29



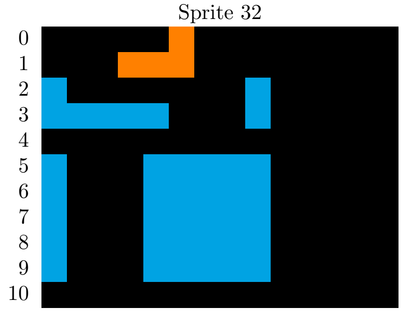
Sprite 30



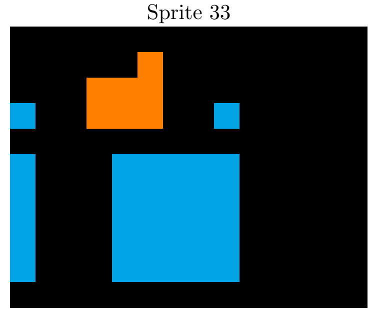
Sprite 31



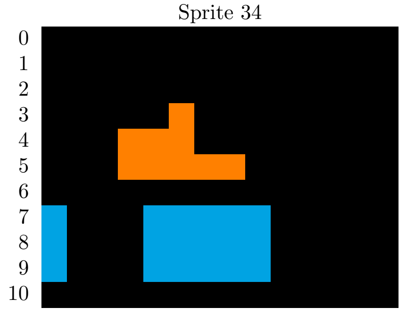
Sprite 32



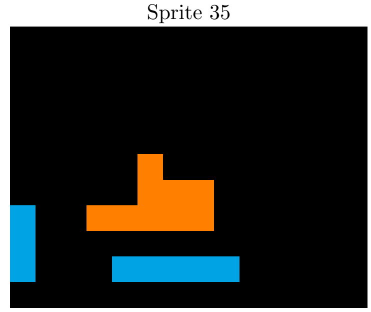
Sprite 33



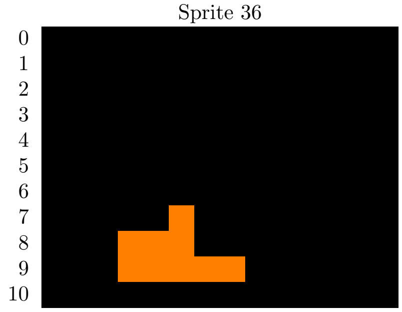
Sprite 34



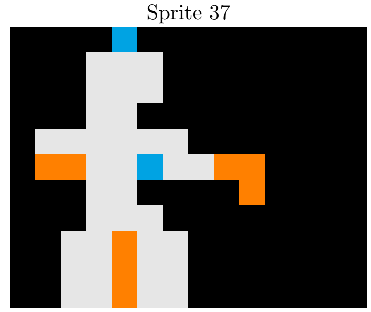
Sprite 35



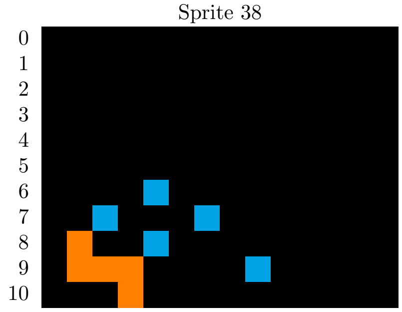
Sprite 36



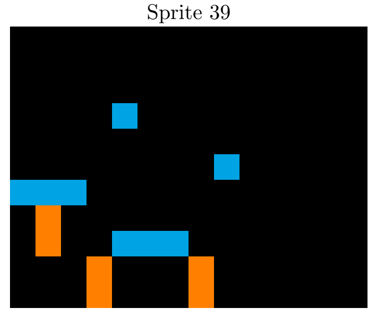
Sprite 37



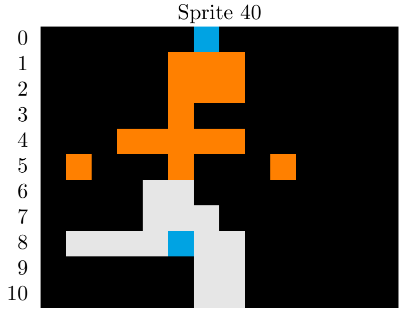
Sprite 38



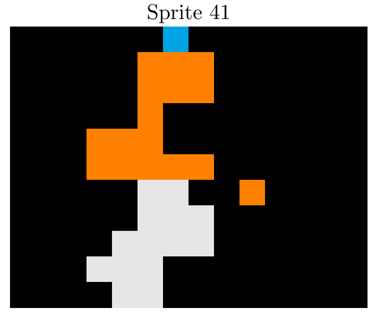
Sprite 39



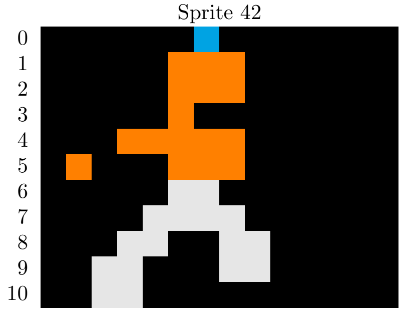
Sprite 40



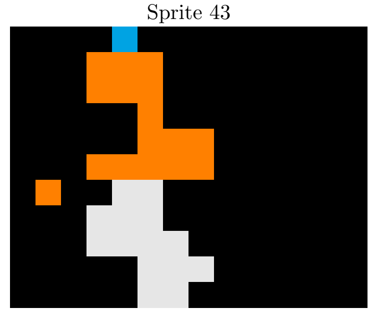
Sprite 41



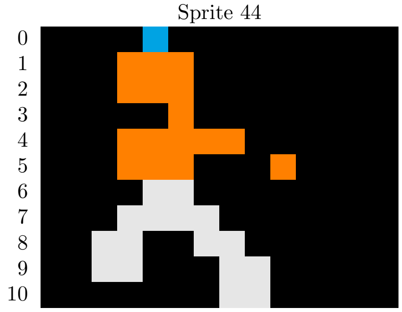
Sprite 42



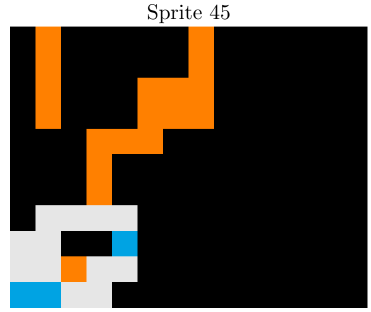
Sprite 43



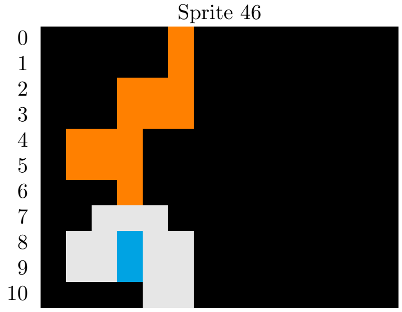
Sprite 44



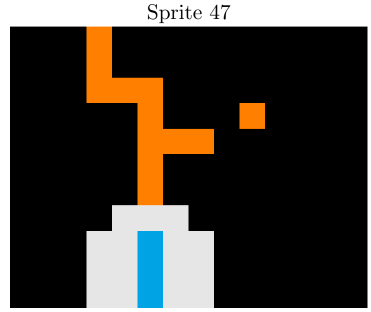
Sprite 45



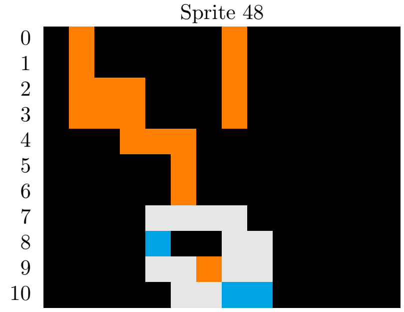
Sprite 46



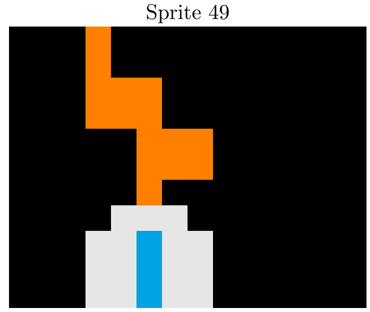
Sprite 47



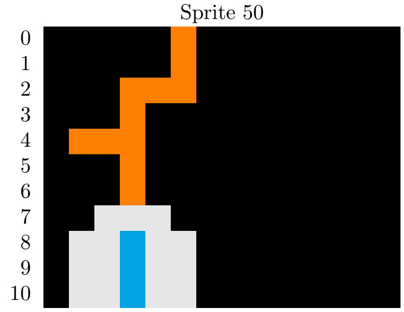
Sprite 48



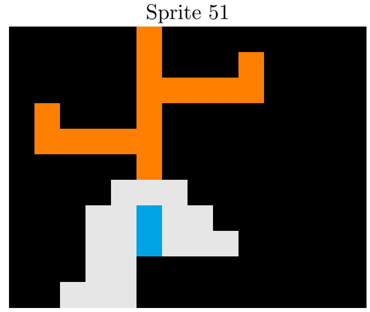
Sprite 49



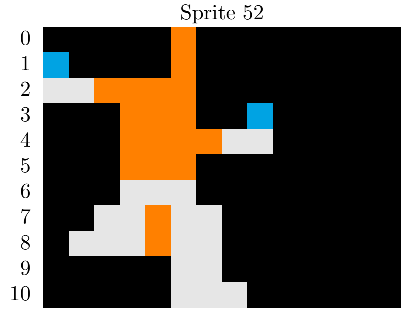
Sprite 50



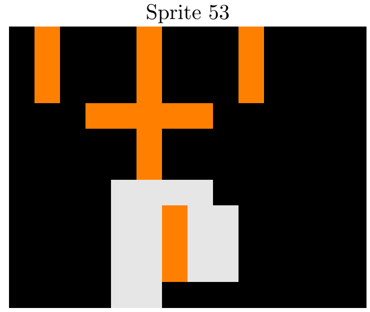
Sprite 51



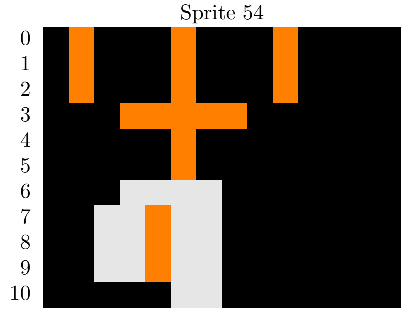
Sprite 52



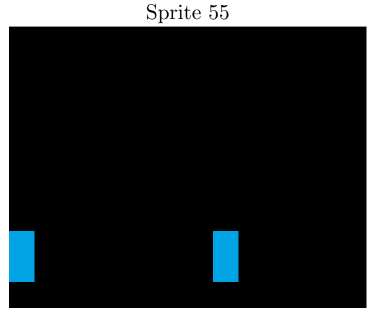
Sprite 53



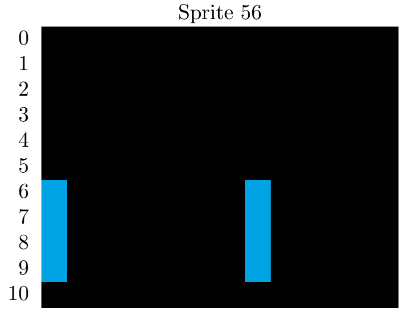
Sprite 54



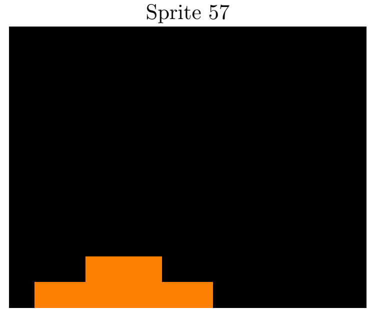
Sprite 55



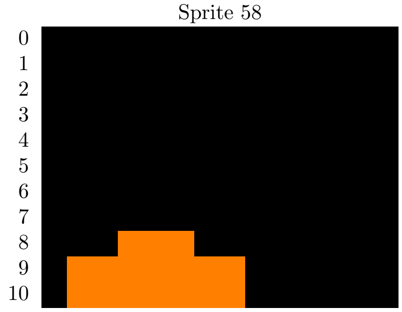
Sprite 56



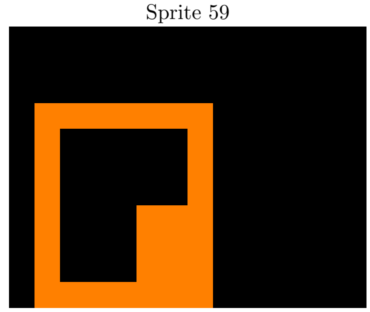
Sprite 57



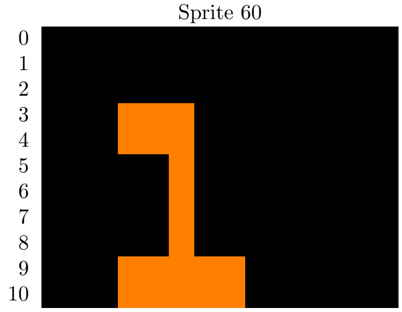
Sprite 58



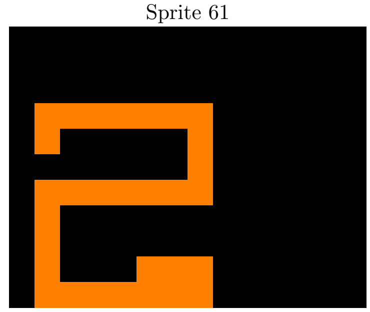
Sprite 59



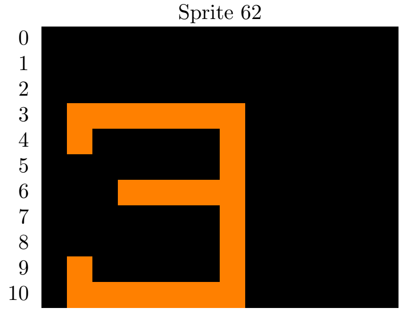
Sprite 60



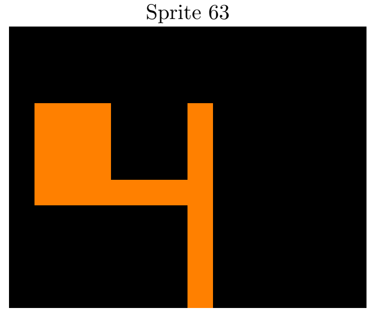
Sprite 61



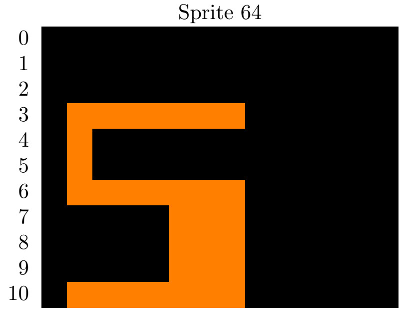
Sprite 62



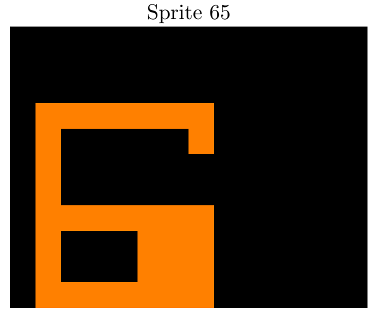
Sprite 63



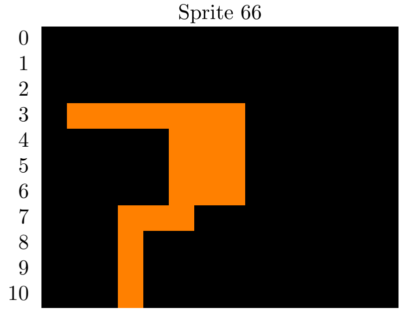
Sprite 64



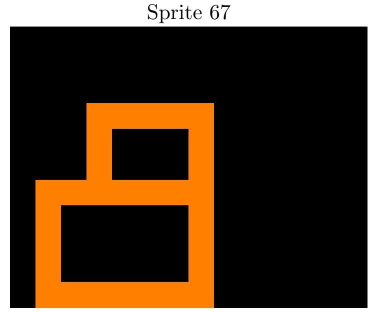
Sprite 65



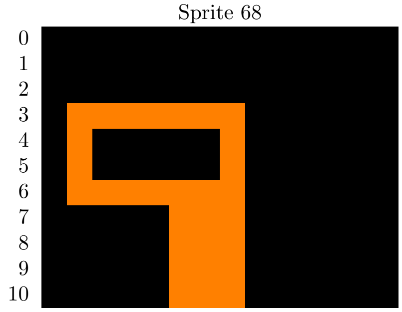
Sprite 66



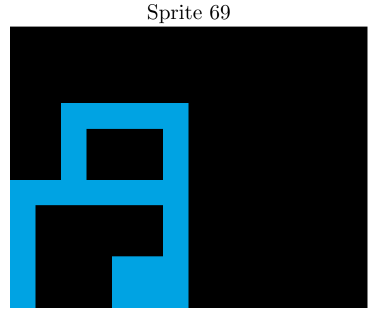
Sprite 67



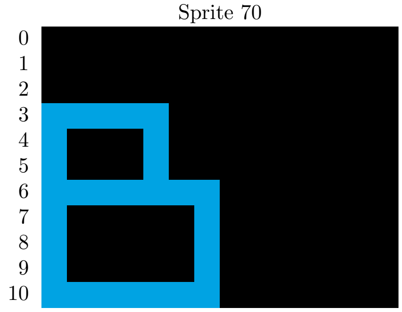
Sprite 68



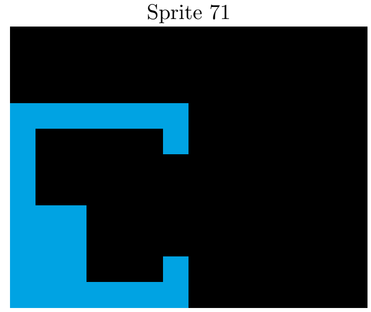
Sprite 69



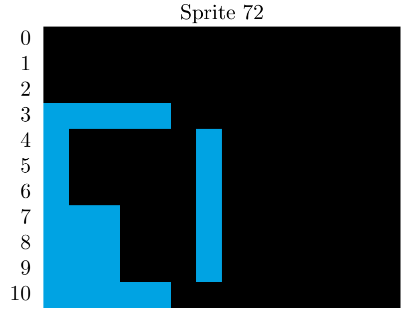
Sprite 70



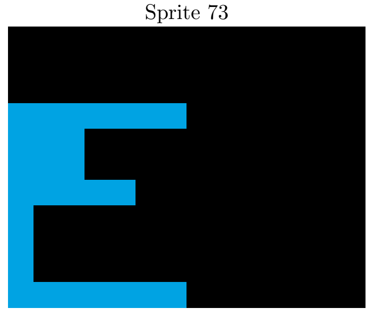
Sprite 71



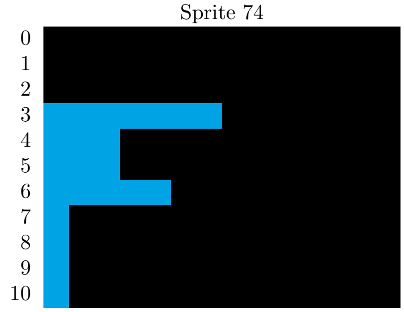
Sprite 72



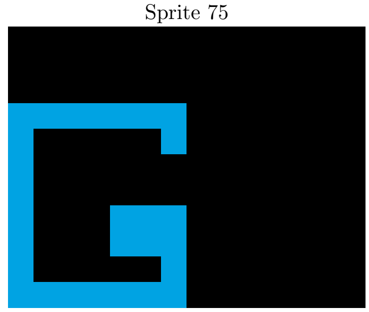
Sprite 73



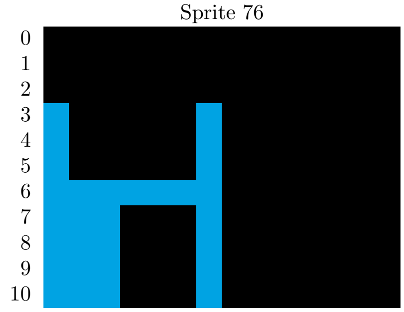
Sprite 74



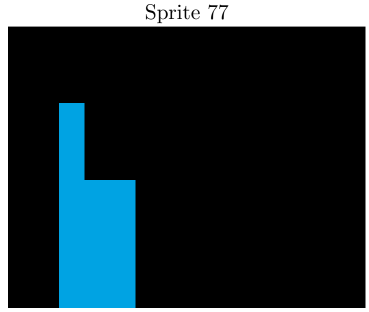
Sprite 75



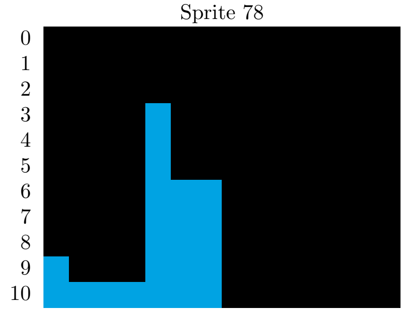
Sprite 76



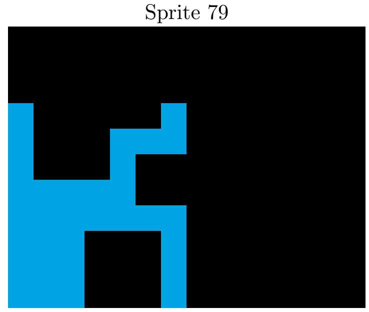
Sprite 77



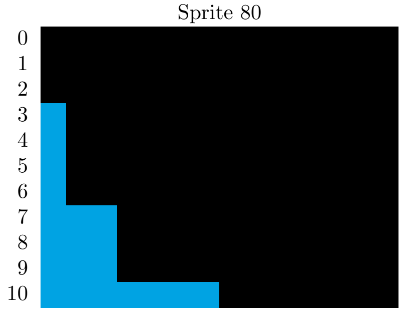
Sprite 78



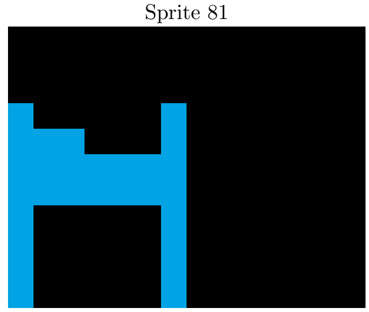
Sprite 79



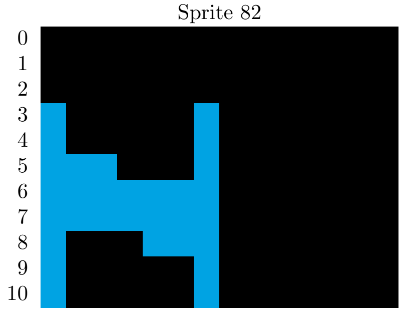
Sprite 80



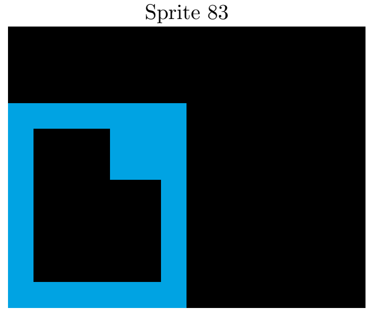
Sprite 81



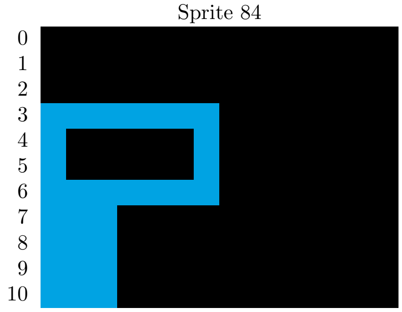
Sprite 82



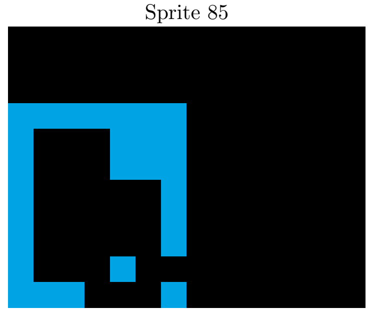
Sprite 83



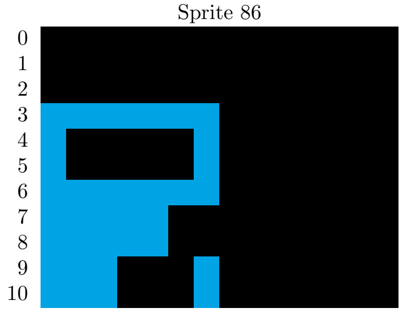
Sprite 84



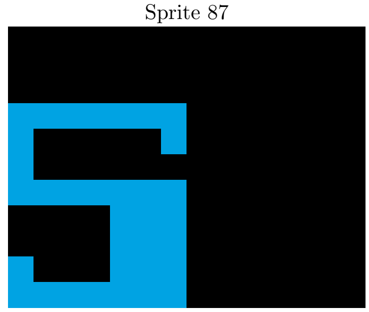
Sprite 85



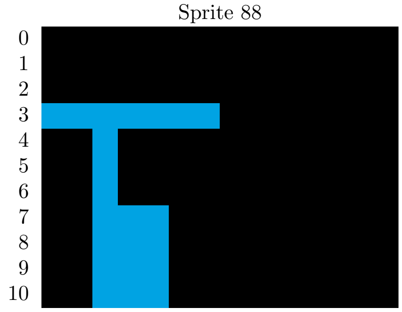
Sprite 86



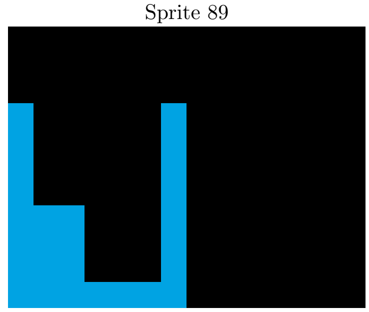
Sprite 87



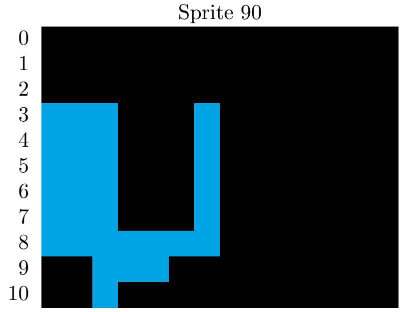
Sprite 88



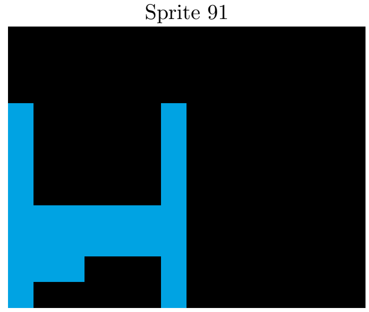
Sprite 89



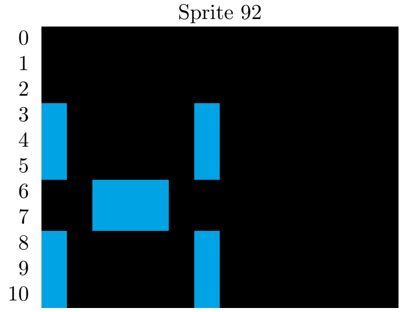
Sprite 90



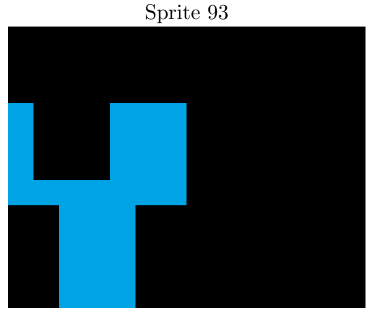
Sprite 91



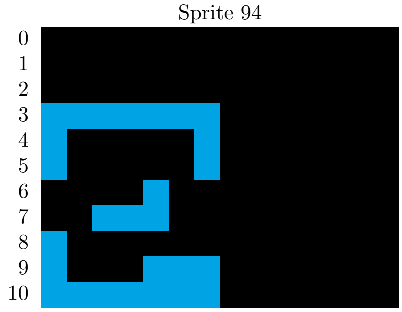
Sprite 92



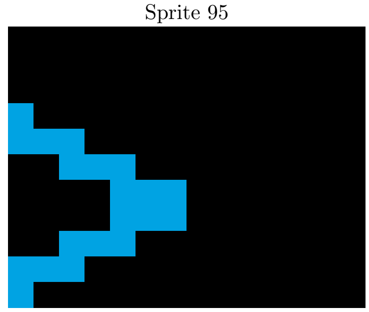
Sprite 93

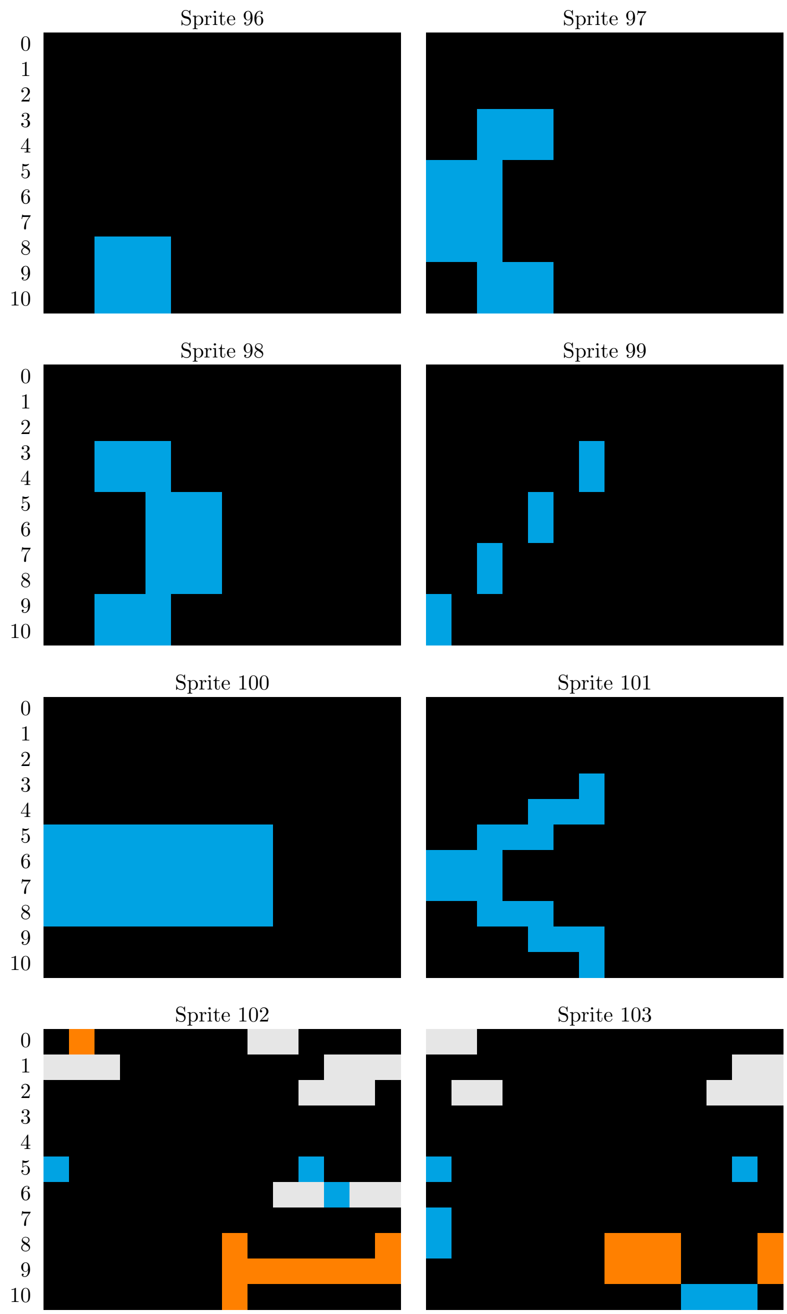


Sprite 94



Sprite 95





2.3 Shifting sprites

This is all very good if we're going to draw sprites exactly on 7-pixel boundaries, but what if we want to draw them starting at other columns? In general, such a shifted sprite would straddle three bytes, and Lode Runner sets aside an area of memory at the end of zero page for 11 rows of three bytes that we'll write to when we want to compute the data for a shifted sprite.

21 ⟨*defines* 3⟩+≡ (71b) ◁3 23c▷

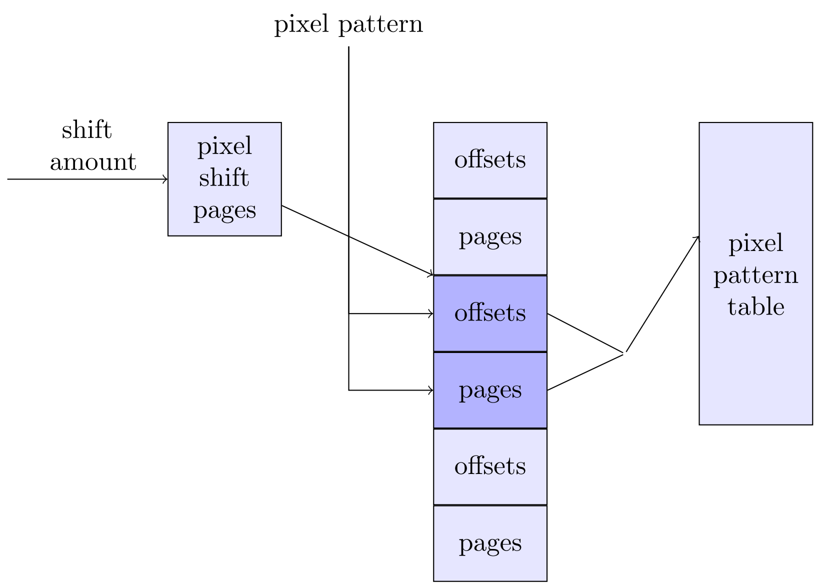
```
        ORG     $DF
BLOCK_DATA      DS      33
```

Defines:
BLOCK_DATA, used in chunks 24 and 30.

Lode Runner also contains tables which show how to shift any arbitrary 7-pixel pattern right by any amount from zero to six pixels.

For example, suppose we start with a pixel pattern of `0110001`, and we want to shift that right by three bits. The 14-bit result would be `0000110 0010000`. However, we have to break that up into bytes, reverse the bits (remember that each byte's bits are output as pixels least significant bit first), and set their high bits, so we end up with `10110000 10000100`.

Now, given a shift amount and a pixel pattern, we should be able to find the two-byte shifted pattern. Lode Runner accomplishes this with table lookups as follows:



The pixel pattern table is a table of every possible pattern of 7 consecutive pixels spread out over two bytes. This table is 512 entries, each entry being two bytes. A naive table would have redundancy. For example the pattern `0000100` starting at column 0 is exactly the same as the pattern `0001000` starting at column 1. This table eliminates that redundancy.

22 ⟨*tables* 7⟩+≡ (71b) ◁7 23a▷

```
      ORG     $A900
PIXEL_PATTERN_TABLE:
      INCLUDE "pixel_pattern_table.asm"
```

Defines:
`PIXEL_PATTERN_TABLE`, never used.

Now we just need tables which index into PIXEL_PATTERN_TABLE for every 7-pixel pattern and shift value. This table works by having the page number for the shifted pixel pattern at index `shift * 0x100 + 0x80 + pattern` and the offset at index `shift * 0x100 + pattern`.

23a ⟨*tables* 7⟩+≡ (71b) ◁22 23b▷

```
      ORG     $A200
  PIXEL_SHIFT_TABLE:
      INCLUDE "pixel_shift_table.asm"
```

Defines:
PIXEL_SHIFT_TABLE, never used.

Rather than multiplying the shift value by `0x100`, we instead define another table which holds the page numbers for the shift tables for each shift value.

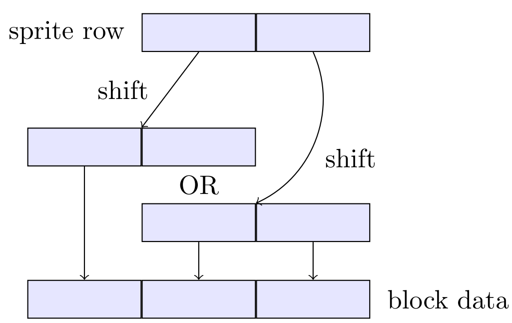
23b ⟨*tables* 7⟩+≡ (71b) ◁23a 26a▷

```
      ORG     $84C1
  PIXEL_SHIFT_PAGES:
      HEX     A2 A3 A4 A5 A6 A7 A8
```

Defines:
PIXEL_SHIFT_PAGES, used in chunk 24.

So we can get shifted pixels by indexing into all these tables.

Now we can define a routine that will take a sprite number and a pixel shift amount, and write the shifted pixel data into the BLOCK_DATA area. The routine first shifts the first byte of the sprite into a two-byte area. Then it shifts the second byte of the sprite, and combines that two-byte result with the first. Thus, we shift two bytes of sprite data into a three-byte result.



Rather than load addresses from the tables and store them, the routine modifies its own instructions with those addresses.

23c ⟨*defines* 3⟩+≡ (71b) ◁21 26b▷

```
      ORG     $1D
  ROW_COUNT       DS      1
  SPRITE_NUM      DS      1
```

Defines:
ROW_COUNT, used in chunks 24 and 30.
SPRITE_NUM, used in chunks 24 and 30.

24 ⟨*routines* 4⟩+≡ (71b) ◁4 26c▷

```
    ORG     $8438
COMPUTE_SHIFTED_SPRITE:
    SUBROUTINE
    ; Enter routine with X set to pixel shift amount and
    ; SPRITE_NUM containing the sprite number to read.

.offset_table       EQU $A000               ; Target addresses in read
.page_table         EQU $A080               ; instructions. The only truly
.shift_ptr_byte0    EQU $A000               ; necessary value here is the
.shift_ptr_byte1    EQU $A000               ; 0x80 in .shift_ptr_byte0.

    LDA     #$0B                            ; 11 rows
    STA     ROW_COUNT
    LDA     #<SPRITE_DATA
    STA     TMP_PTR
    LDA     #>SPRITE_DATA
    STA     TMP_PTR+1                       ; TMP_PTR = SPRITE_DATA
    LDA     PIXEL_SHIFT_PAGES,X
    STA     .rd_offset_table + 2
    STA     .rd_page_table + 2
    STA     .rd_offset_table2 + 2
    STA     .rd_page_table2 + 2             ; Fix up pages in lookup instructions
                                            ; based on shift amount (X).

    LDX     #$00                            ; X is the offset into BLOCK_DATA.

.loop:                                      ; === LOOP === (over all 11 rows)
    LDY     SPRITE_NUM
    LDA     (TMP_PTR),Y
    TAY                                     ; Get sprite pixel data.

.rd_offset_table:
    LDA     .offset_table,Y                 ; Load offset for shift amount.
    STA     .rd_shift_ptr_byte0 + 1
    CLC
    ADC     #$01
    STA     .rd_shift_ptr_byte1 + 1         ; Fix up instruction offsets with it.
.rd_page_table:
    LDA     .page_table,Y                   ; Load page for shift amount.
    STA     .rd_shift_ptr_byte0 + 2
    STA     .rd_shift_ptr_byte1 + 2         ; Fix up instruction page with it.

.rd_shift_ptr_byte0:
    LDA     .shift_ptr_byte0                ; Read shifted pixel data byte 0
    STA     BLOCK_DATA,X                    ; and store in block data byte 0.
.rd_shift_ptr_byte1:
    LDA     .shift_ptr_byte1                ; Read shifted pixel data byte 1
    STA     BLOCK_DATA+1,X                  ; and store in block data byte 1.
```

```
    LDA     TMP_PTR
    CLC
    ADC     #$68
    STA     TMP_PTR
    LDA     TMP_PTR+1
    ADC     #$00
    STA     TMP_PTR+1                   ; TMP_PTR++

    ; Now basically do the same thing with the second sprite byte

    LDY     SPRITE_NUM
    LDA     (TMP_PTR),Y
    TAY                                 ; Get sprite pixel data.

.rd_offset_table2:
    LDA     .offset_table,Y             ; Load offset for shift amount.
    STA     .rd_shift_ptr2_byte0 + 1
    CLC
    ADC     #$01
    STA     .rd_shift_ptr2_byte1 + 1    ; Fix up instruction offsets with it.
.rd_page_table2:
    LDA     .page_table,Y               ; Load page for shift amount.
    STA     .rd_shift_ptr2_byte0 + 2
    STA     .rd_shift_ptr2_byte1 + 2    ; Fix up instruction page with it.

.rd_shift_ptr2_byte0:
    LDA     .shift_ptr_byte0            ; Read shifted pixel data byte 0
    ORA     BLOCK_DATA+1,X              ; OR with previous block data byte 1
    STA     BLOCK_DATA+1,X              ; and store in block data byte 1.
.rd_shift_ptr2_byte1:
    LDA     .shift_ptr_byte1            ; Read shifted pixel data byte 1
    STA     BLOCK_DATA+2,X              ; and store in block data byte 2.

    LDA     TMP_PTR
    CLC
    ADC     #$68
    STA     TMP_PTR
    LDA     TMP_PTR+1
    ADC     #$00
    STA     TMP_PTR+1                   ; TMP_PTR++

    INX
    INX
    INX                                 ; X += 3
    DEC     ROW_COUNT                   ; ROW_COUNT--
    BNE     .loop                       ; loop while ROW_COUNT > 0
    RTS
```

Defines:
  COMPUTE_SHIFTED_SPRITE, used in chunk 30.
Uses BLOCK_DATA 21, PIXEL_SHIFT_PAGES 23b, ROW_COUNT 23c, SPRITE_DATA 7, SPRITE_NUM 23c,

and TMP_PTR 3.

2.4 Memory mapped graphics

Within a screen row, consecutive bytes map to consecutive pixels. However, rows themselves are not consecutive in memory.

To make it easy to convert a row number from 0 to 191 to a base address, Lode Runner has a table and a routine to use that table.

26a ⟨*tables* 7⟩+≡ (71b) ◁23b 27b▷

```
    ORG     $1A85
ROW_TO_OFFSET_LO:
    INCLUDE "row_to_offset_lo_table.asm"
ROW_TO_OFFSET_HI:
    INCLUDE "row_to_offset_hi_table.asm"
```

Defines:
ROW_TO_OFFSET_HI, used in chunks 26c and 27a.
ROW_TO_OFFSET_LO, used in chunks 26c and 27a.

26b ⟨*defines* 3⟩+≡ (71b) ◁23c 29a▷

```
ROW_ADDR        EQU     $0C     ; 2 bytes
ROW_ADDR2       EQU     $0E     ; 2 bytes
HGR_PAGE        EQU     $1F     ; 0x20 for HGR1, 0x40 for HGR2
```

Defines:
HGR_PAGE, used in chunks 26c and 30.
ROW_ADDR, used in chunks 26c, 27a, 30, 50, and 63a.
ROW_ADDR2, used in chunks 27a, 50, and 63a.

26c ⟨*routines* 4⟩+≡ (71b) ◁24 27a▷

```
    ORG     $7A31
ROW_TO_ADDR:
    SUBROUTINE
    ; Enter routine with Y set to row. Base address
    ; (for column 0) will be placed in ROW_ADDR.

    LDA     ROW_TO_OFFSET_LO,Y
    STA     ROW_ADDR
    LDA     ROW_TO_OFFSET_HI,Y
    ORA     HGR_PAGE
    STA     ROW_ADDR+1
    RTS
```

Defines:
ROW_TO_ADDR, used in chunk 30.
Uses HGR_PAGE 26b, ROW_ADDR 26b, ROW_TO_OFFSET_HI 26a, and ROW_TO_OFFSET_LO 26a.

There's also a routine to load the address for both page 1 and page 2.

27a ⟨*routines 4*⟩+≡ (71b) ◁26c 28▷

```
    ORG     $7A3E
ROW_TO_ADDR_FOR_BOTH_PAGES:
    SUBROUTINE
    ; Enter routine with Y set to row. Base address
    ; (for column 0) will be placed in ROW_ADDR (for page 1)
    ; and ROW_ADDR2 (for page 2).

    LDA     ROW_TO_OFFSET_LO,Y
    STA     ROW_ADDR
    STA     ROW_ADDR2
    LDA     ROW_TO_OFFSET_HI,Y
    ORA     #$20
    STA     ROW_ADDR+1
    EOR     #$60
    STA     ROW_ADDR2+1
    RTS
```

Defines:
ROW_TO_ADDR_FOR_BOTH_PAGES, used in chunks 59–62.
Uses ROW_ADDR 26b, ROW_ADDR2 26b, ROW_TO_OFFSET_HI 26a, and ROW_TO_OFFSET_LO 26a.

Lode Runner's screens are organized into 28 sprites across by 17 sprites down. To convert between sprite coordinates and screen coordinates, we use tables and lookup routines. Each sprite is 10 pixels across by 11 pixels down.

27b ⟨*tables 7*⟩+≡ (71b) ◁26a 29b▷

```
    ORG     $1C35
ROW_TABLE2:
    ; 28 rows of 5 pixels each
    HEX     00 05 0a 0f 14 19 1e 23 28 2d 32 37 3c 41 46 4b
    HEX     50 55 5a 5f 64 69 6e 73 78 7d 82 87
ROW_TABLE:
    ; 17 rows of 11 pixels each
    HEX     00 0B 16 21 2C 37 42 4D 58 63 6E 79 84 8F 9A A5
    HEX     B5
COL_TABLE:
    ; Byte number
    HEX     00 01 02 04 05 07 08 0A 0B 0C 0E 0F 11 12 14 15
    HEX     16 18 19 1B 1C 1E 1F 20 22 23 25 26
COL_SHIFT_TABLE:
    ; Right shift amount
    HEX     00 03 06 02 05 01 04 00 03 06 02 05 01 04 00 03
    HEX     06 02 05 01 04 00 03 06 02 05 01 04
```

Defines:
COL_SHIFT_TABLE, used in chunks 28 and 30.
COL_TABLE, used in chunks 28 and 30.
ROW_TABLE, used in chunks 28 and 30.
ROW_TABLE2, used in chunk 28.

28 ⟨*routines* 4⟩+≡ (71b) ◁27a 30▷

```
      ORG     $885D
  GET_ROWNUM_FOR:
      SUBROUTINE
      ; Enter routine with Y set to sprite row. On
      ; return,Y  will be set to screen row.
      ; We can also set X to something, and on return
      ; X is set to something based on ROW_TABLE2, but
      ; so far I'm not sure what it's used for.

      LDA     ROW_TABLE,Y
      PHA
      LDA     ROW_TABLE2,X
      TAX                         ; X = ROW_TABLE2[X]
      PLA
      TAY                         ; Y = ROW_TABLE[Y]
      RTS

  GET_COLNUM_FOR:
      SUBROUTINE
      ; Enter routine with X set to sprite number. On
      ; return, A will be set to screen column byte number
      ; and X will be set to an additional right shift amount.

      LDA     COL_TABLE,X
      PHA                         ; A = COL_TABLE2[X]
      LDA     COL_SHIFT_TABLE,X
      TAX                         ; X = COL_SHIFT_TABLE[X]
      PLA
      RTS
```

Defines:
 GET_COLNUM_FOR, used in chunk 30.
 GET_ROWNUM_FOR, used in chunk 30.
Uses COL_SHIFT_TABLE 27b, COL_TABLE 27b, ROW_TABLE 27b, and ROW_TABLE2 27b.

Now we can finally write the routines that draw a sprite on the screen. There are two entry points, one to draw on HGR1, and one for HGR2.

29a ⟨*defines* 3⟩+≡ (71b) ◁26b 34▷

```
    ORG     $1B
ROWNUM          DS      1
COLNUM          DS      1
    ORG     $50
MASK0           DS      1
MASK1           DS      1
    ORG     $71
COL_SHIFT_AMT   DS      1
    ORG     $85
GAME_COLNUM     DS      1
GAME_ROWNUM     DS      1
```

Defines:
- COL_SHIFT_AMT, used in chunk 30.
- COLNUM, used in chunk 30.
- GAME_COLNUM, used in chunks 30, 35a, 37a, 40, 42, 44d, 49a, and 51a.
- GAME_ROWNUM, used in chunks 30, 35a, 40, 42, 43, 46–49, and 51a.
- ROWNUM, used in chunk 30.

29b ⟨*tables* 7⟩+≡ (71b) ◁27b 44a▷

```
    ORG     $8328
PIXEL_MASK0:
    BYTE    %00000000
    BYTE    %00000001
    BYTE    %00000011
    BYTE    %00000111
    BYTE    %00001111
    BYTE    %00011111
    BYTE    %00111111
PIXEL_MASK1:
    BYTE    %11111000
    BYTE    %11110000
    BYTE    %11100000
    BYTE    %11000000
    BYTE    %10000000
    BYTE    %11111110
    BYTE    %11111100
```

Defines:
- PIXEL_MASK0, used in chunk 30.
- PIXEL_MASK1, used in chunk 30.

30 ⟨*routines 4*⟩+≡ (71b) ◁28 33▷

```
    ORG     $82AA
DRAW_SPRITE_PAGE1:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
    ; set to the column to draw it at.

    STA     SPRITE_NUM
    LDA     #$20                ; Page number for HGR1
    BNE     DRAW_SPRITE         ; Actually unconditional jump

DRAW_SPRITE_PAGE2:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; GAME_ROWNUM set to the row to draw it at, and GAME_COLNUM
    ; set to the column to draw it at.

    STA     SPRITE_NUM
    LDA     #$40                ; Page number for HGR2
    ; fallthrough

DRAW_SPRITE:
    STA     HGR_PAGE
    LDY     GAME_ROWNUM
    JSR     GET_ROWNUM_FOR
    STY     ROWNUM              ; ROWNUM = ROW_TABLE[GAME_ROWNUM]

    LDX     GAME_COLNUM
    JSR     GET_COLNUM_FOR
    STA     COLNUM              ; COLNUM = COL_TABLE[GAME_COLNUM]
    STX     COL_SHIFT_AMT       ; COL_SHIFT_AMT = COL_SHIFT_TABLE[GAME_COLNUM]

    LDA     PIXEL_MASK0,X
    STA     MASK0               ; MASK0 = PIXEL_MASK0[COL_SHIFT_AMT]
    LDA     PIXEL_MASK1,X
    STA     MASK1               ; MASK1 = PIXEL_MASK1[COL_SHIFT_AMT]

    JSR     COMPUTE_SHIFTED_SPRITE

    LDA     #$0B
    STA     ROW_COUNT
    LDX     #$00
    LDA     COL_SHIFT_AMT
    CMP     #$05
    BCS     .need_3_bytes       ; If COL_SHIFT_AMT >= 5, we need to alter three screen bytes,
                                ; otherwise just two bytes.

.loop1:
    LDY     ROWNUM
```

```
        JSR     ROW_TO_ADDR
        LDY     COLNUM
        LDA     (ROW_ADDR),Y
        AND     MASK0
        ORA     BLOCK_DATA,X
        STA     (ROW_ADDR),Y        ; screen[COLNUM] = screen[COLNUM] & MASK0 | BLOCK_DATA[i]

        INX                         ; X++
        INY                         ; Y++
        LDA     (ROW_ADDR),Y
        AND     MASK1
        ORA     BLOCK_DATA,X
        STA     (ROW_ADDR),Y        ; screen[COLNUM+1] = screen[COLNUM+1] & MASK1 | BLOCK_DATA[i+1]

        INX
        INX                         ; X += 2
        INC     ROWNUM              ; ROWNUM++
        DEC     ROW_COUNT           ; ROW_COUNT--
        BNE     .loop1              ; loop while ROW_COUNT > 0
        RTS

    .need_3_bytes
        LDY     ROWNUM
        JSR     ROW_TO_ADDR
        LDY     COLNUM
        LDA     (ROW_ADDR),Y
        AND     MASK0
        ORA     BLOCK_DATA,X
        STA     (ROW_ADDR),Y        ; screen[COLNUM] = screen[COLNUM] & MASK0 | BLOCK_DATA[i]

        INX                         ; X++
        INY                         ; Y++
        LDA     BLOCK_DATA,X
        STA     (ROW_ADDR),Y        ; screen[COLNUM+1] = BLOCK_DATA[i+1]

        INX                         ; X++
        INY                         ; Y++
        LDA     (ROW_ADDR),Y
        AND     MASK1
        ORA     BLOCK_DATA,X
        STA     (ROW_ADDR),Y        ; screen[COLNUM+2] = screen[COLNUM+2] & MASK1 | BLOCK_DATA[i+2]

        INX                         ; X++
        INC     ROWNUM              ; ROWNUM++
        DEC     ROW_COUNT           ; ROW_COUNT--
        BNE     .need_3_bytes       ; loop while ROW_COUNT > 0
        RTS
```

Defines:
`DRAW_SPRITE_PAGE1`, used in chunks 35a and 37a.
`DRAW_SPRITE_PAGE2`, used in chunks 35a, 37a, 49a, and 51a.

Uses BLOCK_DATA 21, COL_SHIFT_AMT 29a, COL_SHIFT_TABLE 27b, COL_TABLE 27b, COLNUM 29a, COMPUTE_SHIFTED_SPRITE 24, GAME_COLNUM 29a, GAME_ROWNUM 29a, GET_COLNUM_FOR 28, GET_ROWNUM_FOR 28, HGR_PAGE 26b, PIXEL_MASK0 29b, PIXEL_MASK1 29b, ROW_ADDR 26b, ROW_COUNT 23c, ROW_TABLE 27b, ROW_TO_ADDR 26c, ROWNUM 29a, and SPRITE_NUM 23c.

2.5 Printing strings

Now that we can put sprites onto the screen at any game coordinate, we can also have some routines that print strings. We saw above that we have letter and number sprites, plus some punctuation. Letters and punctuation are always blue, while numbers are always orange.

There is a basic routine to put a character at the current `GAME_COLNUM` and `GAME_ROWNUM`, incrementing this "cursor", and putting it at the beginning of the next line if we "print" a newline character.

We first define a routine to convert the ASCII code of a character to its sprite number. Lode Runner sets the high bit of the code to make it be treated as ASCII.

33 ⟨*routines* 4⟩+≡ (71b) ◁30 35a▷

```
    ORG     $7b2a
CHAR_TO_SPRITE_NUM:
    SUBROUTINE
    ; Enter routine with A set to the ASCII code of the
    ; character to convert to sprite number, with the high bit set.
    ; The sprite number is returned in A.

    CMP     #$C1                    ; 'A' -> sprite 69
    BCC     .not_letter
    CMP     #$DB                    ; 'Z' -> sprite 94
    BCC     .letter

.not_letter:
    ; On return, we will subtract 0x7C from X to
    ; get the actual sprite. This is to make A-Z
    ; easier to handle.
    LDX     #$7C
    CMP     #$A0                    ; ' ' -> sprite 0
    BEQ     .end
    LDX     #$DB
    CMP     #$BE                    ; '>' -> sprite 95
    BEQ     .end
    INX
    CMP     #$AE                    ; '.' -> sprite 96
    BEQ     .end
    INX
    CMP     #$A8                    ; '(' -> sprite 97
    BEQ     .end
    INX
    CMP     #$A9                    ; ')' -> sprite 98
    BEQ     .end
    INX
    CMP     #$AF                    ; '/' -> sprite 99
    BEQ     .end
    INX
    CMP     #$AD                    ; '-' -> sprite 100
```

```
    BEQ     .end
    INX
    CMP     #$BC                    ; '<' -> sprite 101
    BEQ     .end
    LDA     #$10                    ; sprite 16: just one of the man sprites
    RTS

.end:
    TXA

.letter:
    SEC
    SBC     #$7C
    RTS
```

Defines:
CHAR_TO_SPRITE_NUM, used in chunk 35a.

Now we can define the routine to put a character on the screen at the current position.

34 ⟨*defines* 3⟩+≡ (71b) ◁29a 35b▷

```
DRAW_PAGE   EQU     $87     ; 0x20 for page 1, 0x40 for page 2
```

Defines:
DRAW_PAGE, used in chunks 35a and 37a.

35a ⟨*routines* 4⟩+≡ (71b) ◁33 36▷

```
      ORG     $7b64
  PUT_CHAR:
      SUBROUTINE
      ; Enter routine with A set to the ASCII code of the
      ; character to put on the screen, with the high bit set.

      CMP     #$8D
      BEQ     NEWLINE                 ; If newline, do NEWLINE instead.
      JSR     CHAR_TO_SPRITE_NUM
      LDX     DRAW_PAGE
      CPX     #$40
      BEQ     .draw_to_page2

      JSR     DRAW_SPRITE_PAGE1
      INC     GAME_COLNUM
      RTS

  .draw_to_page2
      JSR     DRAW_SPRITE_PAGE2
      INC     GAME_COLNUM
      RTS

  NEWLINE:
      SUBROUTINE
      INC     GAME_ROWNUM
      LDA     #$00
      STA     GAME_COLNUM
      RTS
```

Defines:
NEWLINE, never used.
PUT_CHAR, used in chunk 36.
Uses CHAR_TO_SPRITE_NUM 33, DRAW_PAGE 34, DRAW_SPRITE_PAGE1 30, DRAW_SPRITE_PAGE2 30, GAME_COLNUM 29a, and GAME_ROWNUM 29a.

The PUT_STRING routine uses PUT_CHAR to put a string on the screen. Rather than take an address pointing to a string, instead it uses the return address as the source for data. It then has to fix up the actual return address at the end to be just after the zero-terminating byte of the string.

35b ⟨*defines* 3⟩+≡ (71b) ◁34 37b▷

```
      ORG     $10
  SAVED_RET_ADDR      DS.W    1
```

Defines:
SAVED_RET_ADDR, used in chunk 36.

36 ⟨*routines* 4⟩+≡ (71b) ◁35a 37a▷

```
    ORG     $86E0
PUT_STRING:
    SUBROUTINE

    PLA
    STA     SAVED_RET_ADDR
    PLA
    STA     SAVED_RET_ADDR+1
    BNE     .next

.loop:
    LDY     #$00
    LDA     (SAVED_RET_ADDR),Y
    BEQ     .end
    JSR     PUT_CHAR

.next:
    INC     SAVED_RET_ADDR
    BNE     .loop
    INC     SAVED_RET_ADDR+1
    BNE     .loop

.end:
    LDA     SAVED_RET_ADDR+1
    PHA
    LDA     SAVED_RET_ADDR
    PHA
    RTS
```

Defines:
  `PUT_STRING`, never used.
Uses `PUT_CHAR` 35a and `SAVED_RET_ADDR` 35b.

Like PUT_CHAR, we also have PUT_DIGIT which draws the sprite corresponding to digits 0 to 9 at the current position, incrementing the cursor.

37a ⟨*routines* 4⟩+≡ (71b) ◁36 38▷

```
    ORG     $7B15
PUT_DIGIT:
    SUBROUTINE
    ; Enter routine with A set to the digit to put on the screen.

    CLC
    ADC     #$3B                    ; '0' -> sprite 59, '9' -> sprite 68.
    LDX     DRAW_PAGE
    CPX     #$40
    BEQ     .draw_to_page2
    JSR     DRAW_SPRITE_PAGE1
    INC     GAME_COLNUM
    RTS

.draw_to_page2:
    JSR     DRAW_SPRITE_PAGE2
    INC     GAME_COLNUM
    RTS
```

Defines:
PUT_DIGIT, used in chunks 40 and 42.
Uses DRAW_PAGE 34, DRAW_SPRITE_PAGE1 30, DRAW_SPRITE_PAGE2 30, and GAME_COLNUM 29a.

2.6 Numbers

We also need a way to put numbers on the screen.

First, a routine to convert a one-byte decimal number into hundreds, tens, and units.

37b ⟨*defines* 3⟩+≡ (71b) ◁35b 39b▷

```
    ORG     $C0
HUNDREDS        DS      1
TENS            DS      1
UNITS           DS      1
```

Defines:
HUNDREDS, used in chunks 38 and 42.
TENS, used in chunks 38–40 and 42.
UNITS, used in chunks 38–40 and 42.

38 ⟨*routines* 4⟩+≡ (71b) ◁37a 39a▷

```
      ORG     $7AF8
  TO_DECIMAL3:
      SUBROUTINE
      ; Enter routine with A set to the number to convert.

      LDX     #$00
      STX     TENS
      STX     HUNDREDS

  .loop1:
      CMP     100
      BCC     .loop2
      INC     HUNDREDS
      SBC     100
      BNE     .loop1

  .loop2:
      CMP     10
      BCC     .end
      INC     TENS
      SBC     10
      BNE     .loop2

  .end:
      STA     UNITS
      RTS
```

Defines:
`TO_DECIMAL3`, used in chunk 42.
Uses `HUNDREDS` 37b, `TENS` 37b, and `UNITS` 37b.

There's also a routine to convert a BCD byte to tens and units.

39a ⟨*routines* 4⟩+≡ (71b) ◁38 40▷

```
    ORG     $7AE9
BCD_TO_DECIMAL2:
    SUBROUTINE
    ; Enter routine with A set to the BCD number to convert.

    STA     TENS
    AND     #$0F
    STA     UNITS
    LDA     TENS
    LSR
    LSR
    LSR
    LSR
    STA     TENS
    RTS
```

Defines:
BCD_TO_DECIMAL2, used in chunk 40.
Uses TENS 37b and UNITS 37b.

2.7 Score and status

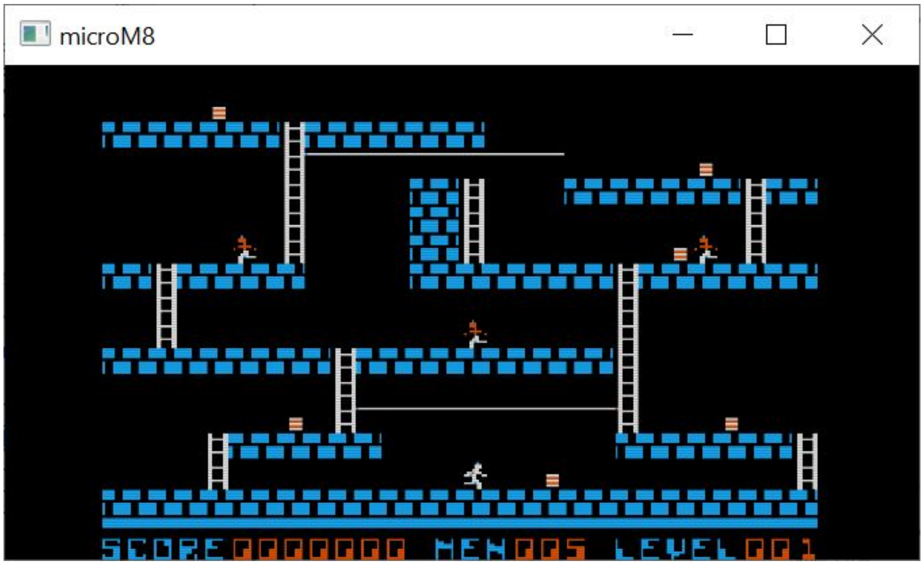
Lode Runner stores your score as an 8-digit BCD number.

39b ⟨*defines* 3⟩+≡ (71b) ◁37b 41▷

```
    ORG     $8D
SCORE       DS      4       ; BCD format, tens/units in first byte.
```

Defines:
SCORE, used in chunk 40.

The score is always put on the screen at row 16 column 5, but only the last 7 digits. Row 16 is the status line, as can be seen at the bottom of this screenshot.



There's a routine to add a 4-digit BCD number to the score and then update it on the screen.

40 ⟨*routines* 4⟩+≡ (71b) ◁39a 42▷

```
    ORG     $7A92
ADD_AND_UPDATE_SCORE:
    SUBROUTINE
    ; Enter routine with A set to BCD tens/units and
    ; Y set to BCD thousands/hundreds.

    CLC
    SED                         ; Turn on BCD addition mode.
    ADC     SCORE
    STA     SCORE
    TYA
    ADC     SCORE+1
    STA     SCORE+1
    LDA     #$00
    ADC     SCORE+2
    STA     SCORE+2
    LDA     #$00
    ADC     SCORE+3
    STA     SCORE+3             ; SCORE += param
    CLD                         ; Turn off BCD addition mode.

    LDA     5
    STA     GAME_COLNUM
```

```
        LDA     16
        STA     GAME_ROWNUM

        LDA     SCORE+3
        JSR     BCD_TO_DECIMAL2
        LDA     UNITS               ; Note we skipped TENS.
        JSR     PUT_DIGIT

        LDA     SCORE+2
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDA     SCORE+1
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JSR     PUT_DIGIT

        LDA     SCORE
        JSR     BCD_TO_DECIMAL2
        LDA     TENS
        JSR     PUT_DIGIT
        LDA     UNITS
        JMP     PUT_DIGIT           ; tail call
```

Defines:
`ADD_AND_UPDATE_SCORE`, never used.
Uses `BCD_TO_DECIMAL2` 39a, `GAME_COLNUM` 29a, `GAME_ROWNUM` 29a, `PUT_DIGIT` 37a, `SCORE` 39b, `TENS` 37b, and `UNITS` 37b.

The other elements in the status line are the number of men (i.e. lives) and the current level.

41 ⟨*defines* 3⟩+≡ (71b) ◁39b 44b▷

```
    ORG     $A6
LEVELNUM    DS      1
    ORG     $C8
LIVES       DS      1
```

Defines:
`LEVELNUM`, used in chunk 42.
`LIVES`, used in chunk 42.

Here are the routines to put the lives and level number on the status line. Lives starts at column 16, and level number starts at column 25.

42 ⟨*routines* 4⟩+≡ (71b) ◁40 51b▷

```
    ORG     $7a70
PUT_STATUS_LIVES:
    SUBROUTINE

    LDA     LIVES
    LDX     16
    ; fallthrough

PUT_STATUS_BYTE:
    SUBROUTINE
    ; Puts the number in A as a three-digit decimal on the screen
    ; at row 16, column X.

    STX     GAME_COLNUM
    JSR     TO_DECIMAL3
    LDA     16
    STA     GAME_ROWNUM
    LDA     HUNDREDS
    JSR     PUT_DIGIT
    LDA     TENS
    JSR     PUT_DIGIT
    LDA     UNITS
    JMP     PUT_DIGIT           ; tail call

PUT_STATUS_LEVEL:
    SUBROUTINE

    LDA     LEVELNUM
    LDX     25
    BNE     PUT_STATUS_BYTE     ; Unconditional jump
```

Defines:
PUT_STATUS_LEVEL, used in chunk 53.
PUT_STATUS_LIVES, used in chunk 53.

Uses GAME_COLNUM 29a, GAME_ROWNUM 29a, HUNDREDS 37b, LEVELNUM 41, LIVES 41, PUT_DIGIT 37a, TENS 37b, TO_DECIMAL3 38, and UNITS 37b.

Chapter 3

Levels

One of the appealing things about Lode Runner are its levels. 150 levels are stored in the game, and there is even a level editor included.

3.1 Drawing a level

Let's see how Lode Runner draws a level. We start with the routine `DRAW_LEVEL_PAGE2`, which draws a level on HGR2. Note that HGR1 would be displayed, so the player doesn't see the draw happening.

We start by looping backwards over rows 15 through 0:

43 ⟨*level draw routine* 43⟩≡ (51b) 44c▷

```
    ORG     $63B3
DRAW_LEVEL_PAGE2:
    SUBROUTINE

    LDY     15
    STY     GAME_ROWNUM

.row_loop:
```

Defines:
`DRAW_LEVEL_PAGE2`, never used.
Uses `GAME_ROWNUM` 29a.

We'll assume the level data is stored in a table which contains 16 pointers, one for each row. As usual in Lode Runner, the pages and offsets for those pointers are stored in separate tables. these are CURR_LEVEL_ROW_SPRITES_PTR_PAGES and CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS.

44a ⟨*tables* 7⟩+≡ (71b) ◁29b 45d▷

```
    ORG     $1C05
CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS:
    HEX     00 1C 38 54 70 8C A8 C4 E0 FC 18 34 50 6C 88 A4
CURR_LEVEL_ROW_SPRITES_PTR_PAGES:
    HEX     08 08 08 08 08 08 08 08 08 08 09 09 09 09 09 09
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2:
    HEX     0A 0A 0A 0A 0A 0A 0A 0A 0A 0A 0B 0B 0B 0B 0B 0B
```

Defines:
CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS, used in chunks 44c and 51a.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES, used in chunks 44c and 51a.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2, used in chunk 44c.

At the beginning of this loop, we create two pointers which we'll simply call PTR1 and PTR2.

44b ⟨*defines* 3⟩+≡ (71b) ◁41 45a▷

```
PTR1        EQU     $06     ; 2 bytes
PTR2        EQU     $08     ; 2 bytes
```

Defines:
PTR1, used in chunks 44, 46b, and 51a.
PTR2, used in chunks 44c and 46–48.

We set PTR1 to the pointer corresponding to the current row, and PTR2 to the other page, though I don't know what it's for yet.

44c ⟨*level draw routine* 43⟩+≡ (51b) ◁43 44d▷

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR1
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
    STA     PTR1+1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1
```

Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 44a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 44a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 44a, PTR1 44b, and PTR2 44b.

Next, we loop over the columns backwards from 27 to 0.

44d ⟨*level draw routine* 43⟩+≡ (51b) ◁44c 44e▷

```
    LDY     27
    STY     GAME_COLNUM

.col_loop:
```

Uses GAME_COLNUM 29a.

We load the sprite from the level data.

44e ⟨*level draw routine* 43⟩+≡ (51b) ◁44d 45c▷

```
    LDA     (PTR1),Y
```

Uses PTR1 44b.

Now, as we place each sprite, we count the number of each piece we've used so far. Remember that anyone can create a level, but there are some limitations. Specifically, we are limited to 45 ladders, one player, and 5 guards. We store the counts as we go.

We'll assume that these values are zeroed before the DRAW_LEVEL_PAGE2 routine is called.

45a ⟨*defines* 3⟩+≡ (71b) ◁44b 45b▷

```
    ORG     $00
PLAYER_COL      DS      1       ; The column number of the player.
PLAYER_ROW      DS      1       ; The row number of the player.
    ORG     $8D
GUARD_COUNT     DS      1
    ORG     $93
GOLD_COUNT      DS      1
    ORG     $A3
LADDER_COUNT    DS      1
```

Defines:
GOLD_COUNT, used in chunk 46c.
GUARD_COUNT, used in chunk 47b.
LADDER_COUNT, used in chunk 46a.
PLAYER_COL, used in chunks 48c and 49b.
PLAYER_ROW, used in chunk 48c.

However, there's a flag called VERBATIM that tells us whether we want to ignore these counts and just draw the level as specified. Possibly when we're using the level editor.

45b ⟨*defines* 3⟩+≡ (71b) ◁45a 48b▷

```
    ORG     $A2
VERBATIM        DS      1
```

Defines:
VERBATIM, used in chunks 45c and 49b.

45c ⟨*level draw routine* 43⟩+≡ (51b) ◁44e 46a▷

```
    LDX     VERBATIM
    BEQ     .draw_sprite1       ; This will then unconditionally jump to
                                ; .draw_sprite2. We have to do that because of
                                ; relative jump amount limitations.
```

Uses VERBATIM 45b.

Next we handle sprite 6, which is a symbol used to denote ladder placement. If we've already got the maximum number of ladders, we just put in a space instead. For each ladder placed, we write the LADDER_LOCS table with its coordinates.

45d ⟨*tables* 7⟩+≡ (71b) ◁44a 47a▷

```
    ORG     $0C00
LADDER_LOCS_COL     DS      48
LADDER_LOCS_ROW     DS      48
```

Defines:
LADDER_LOCS_COL, used in chunk 46a.
LADDER_LOCS_ROW, used in chunk 46a.

46a ⟨*level draw routine* 43⟩+≡ (51b) ◁45c 46b▷

```
    CMP     #$06
    BNE     .check_for_box

    LDX     LADDER_COUNT
    CPX     45
    BCS     .remove_sprite

    INC     LADDER_COUNT
    INX
    LDA     GAME_ROWNUM
    STA     LADDER_LOCS_ROW,X
    TYA
    STA     LADDER_LOCS_COL,X
```

Uses GAME_ROWNUM 29a, LADDER_COUNT 45a, LADDER_LOCS_COL 45d, and LADDER_LOCS_ROW 45d.

In any case, we remove the sprite from the current level data.

46b ⟨*level draw routine* 43⟩+≡ (51b) ◁46a 46c▷

```
.remove_sprite:
    LDA     0
    STA     (PTR1),Y
    STA     (PTR2),Y

.draw_sprite1
    BEQ     .draw_sprite        ; Unconditional jump.
```

Uses PTR1 44b and PTR2 44b.

Next, we check for sprite 7, the gold box.

46c ⟨*level draw routine* 43⟩+≡ (51b) ◁46b 47b▷

```
.check_for_box:
    CMP     #$07
    BNE     .check_for_8

    INC     GOLD_COUNT
    BNE     .draw_sprite        ; This leads to a situation where if we wrap
                                ; GOLD_COUNT around back to 0 (so 256 boxes)
                                ; we end up falling through, which eventually
                                ; just draws the sprite anyway. So this is kind
                                ; of unconditional.
```

Uses GOLD_COUNT 45a.

Next, we check for sprite 8, a guard. If we've already got the maximum number of guards, we just put in a space instead. For each guard placed, we write the GUARD_LOCS table with its coordinates. We also write some other guard-related tables.

47a ⟨*tables* 7⟩+≡ (71b) ◁45d 63b▷

```
      ORG     $0C60
  GUARD_LOCS_COL      DS      8
  GUARD_LOCS_ROW      DS      8
  GUARD_FLAGS_0C70    DS      8
  GUARD_FLAGS_0C78    DS      8
  GUARD_FLAGS_0C80    DS      8
  GUARD_FLAGS_0C88    DS      8
```

Defines:
GUARD_FLAGS_0C70, used in chunk 47b.
GUARD_FLAGS_0C78, used in chunk 47b.
GUARD_FLAGS_0C80, used in chunk 47b.
GUARD_FLAGS_0C88, used in chunk 47b.
GUARD_LOCS_COL, used in chunk 47b.
GUARD_LOCS_ROW, used in chunk 47b.

47b ⟨*level draw routine* 43⟩+≡ (51b) ◁46c 48a▷

```
  .check_for_8:
      CMP     #$08
      BNE     .check_for_9

      LDX     GUARD_COUNT
      CPX     5
      BCS     .remove_sprite          ; If GUARD_COUNT > 5, remove sprite.

      INC     GUARD_COUNT
      INX
      TYA
      STA     GUARD_LOCS_COL,X
      LDA     GAME_ROWNUM
      STA     GUARD_LOCS_ROW,X
      LDA     #$00
      STA     GUARD_FLAGS_0C70,X
      STA     GUARD_FLAGS_0C88,X
      LDA     #$02
      STA     GUARD_FLAGS_0C78,X
      STA     GUARD_FLAGS_0C80,X

      LDA     #$00
      STA     (PTR2),Y
      LDA     #$08
      BNE     .draw_sprite            ; Unconditional jump.
```

Uses GAME_ROWNUM 29a, GUARD_COUNT 45a, GUARD_FLAGS_0C70 47a, GUARD_FLAGS_0C78 47a, GUARD_FLAGS_0C80 47a, GUARD_FLAGS_0C88 47a, GUARD_LOCS_COL 47a, GUARD_LOCS_ROW 47a, and PTR2 44b.

Here we insert a few unconditional branches because of relative jump limitations.

48a ⟨*level draw routine* 43⟩+≡ (51b) ◁47b 48c▷

```
.next_row:
    BPL     .row_loop
.next_col:
    BPL     .col_loop
```

Next we check for sprite 9, the player.

48b ⟨*defines* 3⟩+≡ (71b) ◁45b 52▷

```
PLAYER_FLAGS_0002   EQU     $02
PLAYER_FLAGS_0003   EQU     $03
PLAYER_FLAGS_0004   EQU     $04
```

Defines:
PLAYER_FLAGS_0002, used in chunk 48c.
PLAYER_FLAGS_0003, used in chunk 48c.
PLAYER_FLAGS_0004, used in chunk 48c.

48c ⟨*level draw routine* 43⟩+≡ (51b) ◁48a 48d▷

```
.check_for_9:
    CMP     #$09
    BNE     .check_for_5

    LDX     PLAYER_COL
    BPL     .remove_sprite          ; If PLAYER_COL > 0, remove sprite.

    STY     PLAYER_COL
    LDX     GAME_ROWNUM
    STX     PLAYER_ROW
    LDX     #$02
    STX     PLAYER_FLAGS_0002
    STX     PLAYER_FLAGS_0003
    LDX     #$08
    STX     PLAYER_FLAGS_0004

    LDA     #$00
    STA     (PTR2),Y
    LDA     #$09
    BNE     .draw_sprite            ; Unconditional jump.
```

Uses GAME_ROWNUM 29a, PLAYER_COL 45a, PLAYER_FLAGS_0002 48b, PLAYER_FLAGS_0003 48b, PLAYER_FLAGS_0004 48b, PLAYER_ROW 45a, and PTR2 44b.

Finally, we check for sprite 5, the symbol for a brick, and replace it with a brick. If the sprite is anything else, we just draw it.

48d ⟨*level draw routine* 43⟩+≡ (51b) ◁48c 49a▷

```
.check_for_5:
    CMP     #$05
    BNE     .draw_sprite
    LDA     #$01                    ; Brick sprite
```

We finally draw the sprite, on page 2, and advance the loop.

49a ⟨*level draw routine* 43⟩+≡ (51b) ◁48d 49b▷
```
  .draw_sprite:
      JSR     DRAW_SPRITE_PAGE2

      DEC     GAME_COLNUM
      LDY     GAME_COLNUM
      BPL     .next_col                ; Jumps to .col_loop

      DEC     GAME_ROWNUM
      LDY     GAME_ROWNUM
      BPL     .next_row                ; Jumps to .row_loop
```
Uses DRAW_SPRITE_PAGE2 30, GAME_COLNUM 29a, and GAME_ROWNUM 29a.

After the loop, in verbatim mode, we copy the entire page 2 into page 1 and return. Otherwise, if we did place a player sprite, reveal the screen. If we didn't place a player sprite, that's an error!

49b ⟨*level draw routine* 43⟩+≡ (51b) ◁49a 50▷
```
      LDA     VERBATIM
      BEQ     .copy_page2_to_page1

      LDA     PLAYER_COL
      BPL     .reveal_screen

      SEC                              ; Oops, no player! Return error.
      RTS
```
Uses PLAYER_COL 45a and VERBATIM 45b.

To copy the page, we'll need that second ROW_ADDR2 pointer.

50 ⟨*level draw routine* 43⟩+≡ (51b) ◁49b 51a▷

```
.copy_page2_to_page1:
    LDA     #$20
    STA     ROW_ADDR2+1
    LDA     #$40
    STA     ROW_ADDR+1
    LDA     #$00
    STA     ROW_ADDR2
    STA     ROW_ADDR
    TAY

.copy_loop:
    LDA     (ROW_ADDR),Y
    STA     (ROW_ADDR2),Y
    INY
    BNE     .copy_loop

    INC     ROW_ADDR2+1
    INC     ROW_ADDR+1
    LDX     ROW_ADDR+1
    CPX     #$60
    BCC     .copy_loop

    CLC
    RTS
```

Uses ROW_ADDR 26b and ROW_ADDR2 26b.

Revealing the screen, using an iris wipe. Then, we remove the guard and player sprites!

51a ⟨*level draw routine* 43⟩+≡ (51b) ◁50

```
.reveal_screen
    JSR     IRIS_WIPE

    LDY     15
    STY     GAME_ROWNUM

.row_loop2:
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
    STA     PTR1+1
    LDY     27
    STY     GAME_COLNUM

.col_loop2:
    LDA     (PTR1),Y
    CMP     #$09
    BEQ     .remove
    CMP     #$08
    BNE     .next

.remove:
    LDA     #$00
    JSR     DRAW_SPRITE_PAGE2

.next:
    DEC     GAME_COLNUM
    LDY     GAME_COLNUM
    BPL     .col_loop2

    DEC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    BPL     .row_loop2

    CLC
    RTS
```

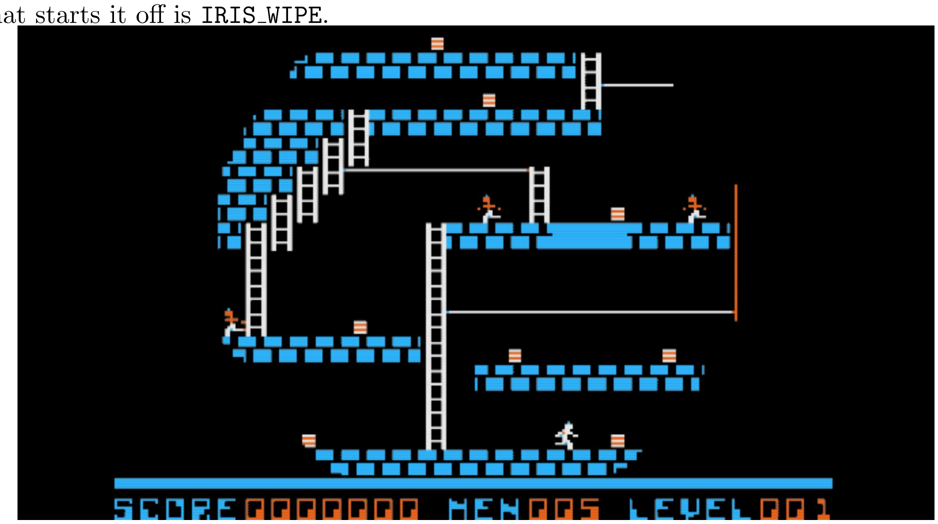
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 44a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 44a, DRAW_SPRITE_PAGE2 30, GAME_COLNUM 29a, GAME_ROWNUM 29a, IRIS_WIPE 53, and PTR1 44b.

51b ⟨*routines* 4⟩+≡ (71b) ◁42 55▷

⟨*level draw routine* 43⟩

3.2 Iris Wipe

Whenever a level is finished or starts, there's an iris wipe transition. The routine that starts it off is `IRIS_WIPE`.



52 ⟨*defines* 3⟩+≡ (71b) ◁48b 54▷

```
WIPE_COUNTER        EQU     $6D
WIPE_MODE           EQU     $A5     ; 0 for open, 1 for close.
WIPE_DIR            EQU     $72     ; 0 for close, 1 for open.
WIPE_CENTER_X       EQU     $77
WIPE_CENTER_Y       EQU     $73
```

Defines:
`WIPE_COUNTER`, used in chunks 53 and 64–66.
`WIPE_MODE`, used in chunk 53.

53 ⟨*iris wipe* 53⟩≡ (71a)

```
    ORG     $88A2
IRIS_WIPE:
    SUBROUTINE

    LDA     88
    STA     WIPE_CENTER_Y
    LDA     140
    STA     WIPE_CENTER_X

    LDA     WIPE_MODE
    BEQ     .iris_open

    LDX     #$AA
    STX     WIPE_COUNTER
    LDX     #$00
    STX     WIPE_DIR             ; Close

.loop_close:
    JSR     IRIS_WIPE_STEP
    DEC     WIPE_COUNTER
    BNE     .loop_close

.iris_open:
    LDA     #$01
    STA     WIPE_COUNTER
    STA     WIPE_MODE           ; So next time we will close.
    STA     WIPE_DIR            ; Open
    JSR     PUT_STATUS_LIVES
    JSR     PUT_STATUS_LEVEL

.loop_open:
    JSR     IRIS_WIPE_STEP
    INC     WIPE_COUNTER
    LDA     WIPE_COUNTER
    CMP     #$AA
    BNE     .loop_open
    RTS
```

Defines:
IRIS_WIPE, used in chunk 51a.

Uses IRIS_WIPE_STEP 57, PUT_STATUS_LEVEL 42, PUT_STATUS_LIVES 42, WIPE_COUNTER 52, and WIPE_MODE 52.

The routine IRIS_WIPE_STEP does a lot of math to compute the circular iris, all parameterized on WIPE_COUNTER.

Here is a routine that divides a 16-bit value in A and X (X being LSB) by 7, storing the result in Y, with remainder in A. The routine effectively does long division. It also uses two temporaries.

54 ⟨*defines* 3⟩+≡ (71b) ◁52 56▷

```
MATH_TMPL      EQU     $6F
MATH_TMPH      EQU     $70
```

Defines:
MATH_TMPH, used in chunks 55, 67, and 68a.
MATH_TMPL, used in chunks 55, 67, and 68a.

55 ⟨*routines* 4⟩+≡ (71b) ◁51b 71a▷

```
      ORG     $8A45
  DIV_BY_7:
      SUBROUTINE
      ; Enter routine with AX set to (unsigned) numerator.
      ; On exit, Y will contain the integer portion of AX/7,
      ; and A contains the remainder.

      STX     MATH_TMPL
      LDY     8
      SEC
      SBC     7

  .loop:
      PHP
      ROL     MATH_TMPH
      ASL     MATH_TMPL
      ROL
      PLP
      BCC     .adjust_up
      SBC     7
      JMP     .next

  .adjust_up
      ADC     7

  .next
      DEY
      BNE     .loop

      BCS     .no_adjust
      ADC     7
      CLC

  .no_adjust
      ROL     MATH_TMPH
      LDY     MATH_TMPH
      RTS
```

Defines:
DIV_BY_7, used in chunks 65 and 66.
Uses MATH_TMPH 54 and MATH_TMPL 54.

Now, for one iris wipe step, we will need lots and lots of temporaries.

56 ⟨*defines* 3⟩+≡ (71b) ◁54

```
WIPE0       EQU     $69     ; 16-bit value
WIPE1       EQU     $67     ; 16-bit value
WIPE2       EQU     $6B     ; 16-bit value
WIPE3L      EQU     $75
WIPE4L      EQU     $76
WIPE5L      EQU     $77
WIPE6L      EQU     $78
WIPE3H      EQU     $79
WIPE4H      EQU     $7A
WIPE5H      EQU     $7B
WIPE6H      EQU     $7C
WIPE7D      EQU     $7D     ; Dividends
WIPE8D      EQU     $7E
WIPE9D      EQU     $7F
WIPE10D     EQU     $80
WIPE7R      EQU     $81     ; Remainders
WIPE8R      EQU     $82
WIPE9R      EQU     $83
WIPE10R     EQU     $84
```

Defines:
`WIPE0`, used in chunks 64 and 68.
`WIPE1`, used in chunks 64 and 67–69.
`WIPE10D`, used in chunks 61, 62, 66b, and 69b.
`WIPE10R`, used in chunks 61, 62, 66b, and 69b.
`WIPE2`, used in chunks 58, 64d, 65a, 67, and 68a.
`WIPE3H`, used in chunks 60, 65b, and 69a.
`WIPE3L`, used in chunks 60, 65b, and 69a.
`WIPE4H`, used in chunks 62, 65c, and 70a.
`WIPE4L`, used in chunks 62, 65c, and 70a.
`WIPE5H`, used in chunks 61, 65c, and 70b.
`WIPE5L`, used in chunks 61, 65c, and 70b.
`WIPE6H`, used in chunks 59b, 65d, and 69d.
`WIPE6L`, used in chunks 59b, 65d, and 69d.
`WIPE7D`, used in chunks 61, 62, 65e, and 69c.
`WIPE7R`, used in chunks 61, 62, 65e, and 69c.
`WIPE8D`, used in chunks 59b, 60, 66a, and 70c.
`WIPE8R`, used in chunks 66a and 70c.
`WIPE9D`, used in chunks 59b, 60, 66a, and 69f.
`WIPE9R`, used in chunks 59b, 60, 66a, and 69f.

The first thing we do for a single step is initialize all those variables!

57 ⟨*iris wipe step* 57⟩≡ (71a) 58▹

```
    ORG     $88D7
IRIS_WIPE_STEP:
    SUBROUTINE

⟨WIPE0 = WIPE_COUNTER 64b⟩
⟨WIPE1 = 0 64c⟩
⟨WIPE2 = 2 * WIPE0 64d⟩
⟨WIPE2 = 3 - WIPE2 65a⟩

; WIPE3, WIPE4, WIPE5, and WIPE6 correspond to
; row numbers. WIPE3 is above the center, WIPE6
; is below the center, while WIPE4 and WIPE5 are on
; the center.

⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 65b⟩
⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 65c⟩
⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 65d⟩

; WIPE7, WIPE8, WIPE9, and WIPE10 correspond to
; column byte numbers. Note the division by 7 pixels!
; WIPE7 is left of center, WIPE10 is right of center,
; while WIPE8 and WIPE9 are on the center.

⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 65e⟩
⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 66a⟩
⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 66b⟩
```

Defines:
IRIS_WIPE_STEP, used in chunk 53.

Now we loop. This involves checking `WIPE1` against `WIPE0`:

- If `WIPE1` < `WIPE0`, return.
- If `WIPE1` == `WIPE0`, go to `DRAW_WIPE_STEP` then return.
- Otherwise, call `DRAW_WIPE_STEP` and go round the loop.

Going around the loop involves calling `DRAW_WIPE_STEP`, then adjusting the numbers.

58 ⟨*iris wipe step* 57⟩+≡ (71a) ◁57

```
.loop:

⟨iris wipe loop check 64a⟩

    JSR     DRAW_WIPE_STEP

    LDA     WIPE2+1
    BPL     .89a7

⟨WIPE2 += 4 * WIPE1 + 6 67⟩
    JMP     .8a14

.89a7:

⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 68a⟩
⟨Decrement WIPE0 68b⟩
⟨Increment WIPE3 69a⟩
⟨Decrement WIPE10 modulo 7 69b⟩
⟨Increment WIPE7 modulo 7 69c⟩
⟨Decrement WIPE6 69d⟩

.8a14:

⟨Increment WIPE1 69e⟩
⟨Increment WIPE9 modulo 7 69f⟩
⟨Decrement WIPE4 70a⟩
⟨Increment WIPE5 70b⟩
⟨Decrement WIPE8 modulo 7 70c⟩
    JMP     .loop
```

Uses `DRAW_WIPE_STEP` 59a and `WIPE2` 56.

Drawing a wipe step draws all four parts. There are two rows which move north and two rows that move south. There are also two left and right offsets, one short and one long. This makes eight combinations.

59a ⟨*draw wipe step* 59a⟩≡ (71a)
```
      ORG     $8A69
  DRAW_WIPE_STEP:
      SUBROUTINE

  ⟨Draw wipe for south part 59b⟩
  ⟨Draw wipe for north part 60⟩
  ⟨Draw wipe for north2 part 61⟩
  ⟨Draw wipe for south2 part 62⟩
```
Defines:
DRAW_WIPE_STEP, used in chunks 58 and 64a.

Each part consists of two halves, right and left (or east and west).

59b ⟨*Draw wipe for south part* 59b⟩≡ (59a)
```
      LDY     WIPE6H
      BNE     .draw_north
      LDY     WIPE6L
      CPY     176
      BCS     .draw_north        ; Skip if WIPE6 >= 176

      JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

      ; East side
      LDY     WIPE9D
      CPY     40
      BCS     .draw_south_west
      LDX     WIPE9R
      JSR     DRAW_WIPE_BLOCK

  .draw_south_west
      ; West side
      LDY     WIPE8D
      CPY     40
      BCS     .draw_north
      LDX     WIPE9R
      JSR     DRAW_WIPE_BLOCK
```
Uses DRAW_WIPE_BLOCK 63a, ROW_TO_ADDR_FOR_BOTH_PAGES 27a, WIPE6H 56, WIPE6L 56, WIPE8D 56, WIPE9D 56, and WIPE9R 56.

60 ⟨*Draw wipe for north part* 60⟩≡ (59a)

```
.draw_north:
    LDY     WIPE3H
    BNE     .draw_north2
    LDY     WIPE3L
    CPY     176
    BCS     .draw_north2        ; Skip if WIPE3 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE9D
    CPY     40
    BCS     .draw_north_west
    LDX     WIPE9R
    JSR     DRAW_WIPE_BLOCK

.draw_north_west
    ; West side
    LDY     WIPE8D
    CPY     40
    BCS     .draw_north2
    LDX     WIPE9R
    JSR     DRAW_WIPE_BLOCK
```

Uses `DRAW_WIPE_BLOCK` 63a, `ROW_TO_ADDR_FOR_BOTH_PAGES` 27a, `WIPE3H` 56, `WIPE3L` 56, `WIPE8D` 56, `WIPE9D` 56, and `WIPE9R` 56.

61 ⟨*Draw wipe for north2 part* 61⟩≡ (59a)

```
.draw_north2:
    LDY     WIPE5H
    BNE     .draw_south2
    LDY     WIPE5L
    CPY     176
    BCS     .draw_south2        ; Skip if WIPE5 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE10D
    CPY     40
    BCS     .draw_north2_west
    LDX     WIPE10R
    JSR     DRAW_WIPE_BLOCK

.draw_north2_west
    ; West side
    LDY     WIPE7D
    CPY     40
    BCS     .draw_south2
    LDX     WIPE7R
    JSR     DRAW_WIPE_BLOCK
```

Uses DRAW_WIPE_BLOCK 63a, ROW_TO_ADDR_FOR_BOTH_PAGES 27a, WIPE10D 56, WIPE10R 56, WIPE5H 56, WIPE5L 56, WIPE7D 56, and WIPE7R 56.

62 ⟨*Draw wipe for south2 part 62*⟩≡ (59a)

```
.draw_south2:
    LDY     WIPE4H
    BNE     .end
    LDY     WIPE4L
    CPY     176
    BCS     .end        ; Skip if WIPE4 >= 176

    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES

    ; East side
    LDY     WIPE10D
    CPY     40
    BCS     .draw_south2_west
    LDX     WIPE10R
    JSR     DRAW_WIPE_BLOCK

.draw_south2_west
    ; West side
    LDY     WIPE7D
    CPY     40
    BCS     .draw_south2
    LDX     WIPE7R
    JMP     DRAW_WIPE_BLOCK           ; tail call

.end:
    RTS
```

Uses DRAW_WIPE_BLOCK 63a, ROW_TO_ADDR_FOR_BOTH_PAGES 27a, WIPE10D 56, WIPE10R 56, WIPE4H 56, WIPE4L 56, WIPE7D 56, and WIPE7R 56.

Drawing a wipe block depends on whether we're opening or closing on the level. Closing on the level just blacks out pixels on page 1. Opening on the level copies some pixels from page 2 into page 1.

63a ⟨*draw wipe block* 63a⟩≡ (71a)

```
    ORG     $8AF6
DRAW_WIPE_BLOCK:
    SUBROUTINE
    ; Enter routine with X set to the column byte and Y set to
    ; the pixel number within that byte (0-6). ROW_ADDR and
    ; ROW_ADDR2 must contain the base row address for page 1
    ; and page 2, respectively.

    LDA     WIPE_DIR
    BNE     .open
    LDA     (ROW_ADDR),Y
    AND     WIPE_BLOCK_CLOSE_MASK,X
    STA     (ROW_ADDR),Y

.open:
    LDA     (ROW_ADDR2),Y
    AND     WIPE_BLOCK_OPEN_MASK,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y
    RTS
```

Defines:
DRAW_WIPE_BLOCK, used in chunks 59–62.
Uses ROW_ADDR 26b, ROW_ADDR2 26b, WIPE_BLOCK_CLOSE_MASK 63b, and WIPE_BLOCK_OPEN_MASK 63b.

63b ⟨*tables* 7⟩+≡ (71b) ◁47a

```
    ORG     $8B0C
WIPE_BLOCK_CLOSE_MASK:
    BYTE    %11110000
    BYTE    %11110000
    BYTE    %11110000
    BYTE    %11110000
    BYTE    %10001111
    BYTE    %10001111
    BYTE    %10001111
WIPE_BLOCK_OPEN_MASK:
    BYTE    %10001111
    BYTE    %10001111
    BYTE    %10001111
    BYTE    %10001111
    BYTE    %11110000
    BYTE    %11110000
    BYTE    %11110000
```

Defines:
WIPE_BLOCK_CLOSE_MASK, used in chunk 63a.
WIPE_BLOCK_OPEN_MASK, used in chunk 63a.

64a ⟨*iris wipe loop check* 64a⟩≡ (58)

```
      LDA     WIPE1+1
      CMP     WIPE0+1
      BCC     .draw_wipe_step ; Effectively, if WIPE1 > WIPE0, jump to .draw_wipe_step.
      BEQ     .8969           ; Otherwise jump to .loop1, which...

  .loop1:
      LDA     WIPE1
      CMP     WIPE0
      BNE     .end
      LDA     WIPE1+1
      CMP     WIPE0+1
      BNE     .end            ; If WIPE0 != WIPE1, return.
      JMP     DRAW_WIPE_STEP

  .end:
      RTS

  .8969:
      LDA     WIPE1
      CMP     WIPE0
      BCS     .loop1          ; The other half of the comparison from .loop.

  .draw_wipe_step:
```

Uses DRAW_WIPE_STEP 59a, WIPE0 56, and WIPE1 56.

3.2.1 Initialization

64b ⟨WIPE0 = WIPE_COUNTER 64b⟩≡ (57)

```
      LDA     WIPE_COUNTER
      STA     WIPE0
      LDA     #$00
      STA     WIPE0+1         ; WIPE0 = WIPE_COUNTER
```

Uses WIPE0 56 and WIPE_COUNTER 52.

64c ⟨WIPE1 = 0 64c⟩≡ (57)

```
      ; fallthrough with A = 0
      STA     WIPE1
      STA     WIPE1+1         ; WIPE1 = 0
```

Uses WIPE1 56.

64d ⟨WIPE2 = 2 * WIPE0 64d⟩≡ (57)

```
      LDA     WIPE0
      ASL
      STA     WIPE2
      LDA     WIPE0+1
      ROL
      STA     WIPE2+1         ; WIPE2 = 2 * WIPE0
```

Uses WIPE0 56 and WIPE2 56.

65a ⟨WIPE2 = 3 - WIPE2 65a⟩≡ (57)
```
      LDA     #$03
      SEC
      SBC     WIPE2
      STA     WIPE2
      LDA     #$00
      SBC     WIPE2+1
      STA     WIPE2+1         ; WIPE2 = 3 - WIPE2
```
Uses WIPE2 56.

65b ⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 65b⟩≡ (57)
```
      LDA     WIPE_CENTER_Y
      SEC
      SBC     WIPE_COUNTER
      STA     WIPE3L
      LDA     #$00
      SBC     #$00
      STA     WIPE3H          ; WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER
```
Uses WIPE3H 56, WIPE3L 56, and WIPE_COUNTER 52.

65c ⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 65c⟩≡ (57)
```
      LDA     WIPE_CENTER_Y
      STA     WIPE4L
      STA     WIPE5L
      LDA     #$00
      STA     WIPE4H
      STA     WIPE5H          ; WIPE4 = WIPE5 = WIPE_CENTER_Y
```
Uses WIPE4H 56, WIPE4L 56, WIPE5H 56, and WIPE5L 56.

65d ⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 65d⟩≡ (57)
```
      LDA     WIPE_CENTER_Y
      CLC
      ADC     WIPE_COUNTER
      STA     WIPE6L
      LDA     #$00
      ADC     #$00
      STA     WIPE6H          ; WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER
```
Uses WIPE6H 56, WIPE6L 56, and WIPE_COUNTER 52.

65e ⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 65e⟩≡ (57)
```
      LDA     WIPE_CENTER_X
      SEC
      SBC     WIPE_COUNTER
      TAX
      LDA     #$00
      SBC     #$00
      JSR     DIV_BY_7
      STY     WIPE7D
      STA     WIPE7R          ; WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7
```
Uses DIV_BY_7 55, WIPE7D 56, WIPE7R 56, and WIPE_COUNTER 52.

66a ⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 66a⟩≡ (57)

```
      LDX     WIPE_CENTER_X
      LDA     #$00
      JSR     DIV_BY_7
      STY     WIPE8D
      STY     WIPE9D
      STA     WIPE8R
      STA     WIPE9R          ; WIPE8 = WIPE9 = WIPE_CENTER_X / 7
```

Uses DIV_BY_7 55, WIPE8D 56, WIPE8R 56, WIPE9D 56, and WIPE9R 56.

66b ⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 66b⟩≡ (57)

```
      LDA     WIPE_CENTER_X
      CLC
      ADC     WIPE_COUNTER
      TAX
      LDA     #$00
      ADC     #$00
      JSR     DIV_BY_7
      STY     WIPE10D
      STA     WIPE10R         ; WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7
```

Uses DIV_BY_7 55, WIPE10D 56, WIPE10R 56, and WIPE_COUNTER 52.

3.2.2 All that math stuff

67 ⟨WIPE2 += 4 * WIPE1 + 6 67⟩≡ (58)

```
      LDA     WIPE1
      ASL
      STA     MATH_TMPL
      LDA     WIPE1+1
      ROL
      STA     MATH_TMPH       ; MATH_TMP = WIPE1 * 2

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     WIPE2
      CLC
      ADC     MATH_TMPL
      STA     MATH_TMPL
      LDA     WIPE2+1
      ADC     MATH_TMPH
      STA     MATH_TMPH       ; MATH_TMP += WIPE2

      LDA     #$06
      CLC
      ADC     MATH_TMPL
      STA     WIPE2
      LDA     #$00
      ADC     MATH_TMPH
      STA     WIPE2+1        ; WIPE2 = MATH_TMP + 6
```

Uses MATH_TMPH 54, MATH_TMPL 54, WIPE1 56, and WIPE2 56.

68a ⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 68a⟩≡ (58)

```
      LDA     WIPE1
      SEC
      SBC     WIPE0
      STA     MATH_TMPL
      LDA     WIPE1+1
      SBC     WIPE0+1
      STA     MATH_TMPH       ; MATH_TMP = WIPE1 - WIPE0

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     MATH_TMPL
      ASL
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ROL
      STA     MATH_TMPH       ; MATH_TMP *= 2

      LDA     MATH_TMPL
      CLC
      ADC     #$10
      STA     MATH_TMPL
      LDA     MATH_TMPH
      ADC     #$00
      STA     MATH_TMPH       ; MATH_TMP += 16

      LDA     MATH_TMPL
      CLC
      ADC     WIPE2
      STA     WIPE2
      LDA     MATH_TMPH
      ADC     WIPE2+1
      STA     WIPE2+1        ; WIPE2 += MATH_TMP
```

Uses MATH_TMPH 54, MATH_TMPL 54, WIPE0 56, WIPE1 56, and WIPE2 56.

68b ⟨*Decrement* WIPE0 68b⟩≡ (58)

```
      LDA     WIPE0
      PHP
      DEC     WIPE0
      PLP
      BNE     .b9ec
      DEC     WIPE0+1         ; WIPE0--
  .b9ec
```

Uses WIPE0 56.

69a ⟨*Increment* WIPE3 69a⟩≡ (58)

```
      INC     WIPE3L
      BNE     .89f2
      INC     WIPE3H          ; WIPE3++
  .89f2
```

Uses WIPE3H 56 and WIPE3L 56.

69b ⟨*Decrement* WIPE10 *modulo 7* 69b⟩≡ (58)

```
      DEC     WIPE10R
      BPL     .89fc
      LDA     #$06
      STA     WIPE10R
      DEC     WIPE10D
  .89fc
```

Uses WIPE10D 56 and WIPE10R 56.

69c ⟨*Increment* WIPE7 *modulo 7* 69c⟩≡ (58)

```
      INC     WIPE7R
      LDA     WIPE7R
      CMP     #$07
      BNE     .8a0a
      LDA     #$00
      STA     WIPE7R
      INC     WIPE7D
  .8a0a
```

Uses WIPE7D 56 and WIPE7R 56.

69d ⟨*Decrement* WIPE6 69d⟩≡ (58)

```
      DEC     WIPE6L
      LDA     WIPE6L
      CMP     #$FF
      BNE     .8a14
      DEC     WIPE6H
```

Uses WIPE6H 56 and WIPE6L 56.

69e ⟨*Increment* WIPE1 69e⟩≡ (58)

```
      INC     WIPE1
      BNE     .8a1a
      INC     WIPE1+1         ; WIPE1++
  .8a1a
```

Uses WIPE1 56.

69f ⟨*Increment* WIPE9 *modulo 7* 69f⟩≡ (58)

```
      INC     WIPE9R
      LDA     WIPE9R
      CMP     #$07
      BNE     .8a28
      LDA     #$00
      STA     WIPE9R
      INC     WIPE9D
  .8a28
```

Uses WIPE9D 56 and WIPE9R 56.

70a ⟨*Decrement* `WIPE4` 70a⟩≡ (58)

```
      DEC     WIPE4L
      LDA     WIPE4L
      CMP     #$FF
      BNE     .8a32
      DEC     WIPE4H
  .8a32
```

Uses `WIPE4H` 56 and `WIPE4L` 56.

70b ⟨*Increment* `WIPE5` 70b⟩≡ (58)

```
      INC     WIPE5L
      BNE     .8a38
      INC     WIPE5H          ; WIPE5++
  .8a38
```

Uses `WIPE5H` 56 and `WIPE5L` 56.

70c ⟨*Decrement* `WIPE8` *modulo 7* 70c⟩≡ (58)

```
      DEC     WIPE8R
      BPL     .8a42
      LDA     #$06
      STA     WIPE8R
      DEC     WIPE8D
  .8a42
```

Uses `WIPE8D` 56 and `WIPE8R` 56.

Chapter 4

The whole thing

We then put together the entire assembly file:

71a ⟨*routines* 4⟩+≡ (71b) ◁55

```
; These are in the order they were placed in the original file.
⟨iris wipe 53⟩
⟨iris wipe step 57⟩
⟨draw wipe step 59a⟩
⟨draw wipe block 63a⟩
```

71b ⟨* 71b⟩≡

```
      PROCESSOR 6502
⟨defines 3⟩
⟨tables 7⟩
⟨routines 4⟩
```

Chapter 5

Defined Chunks

Chapter 6

Index