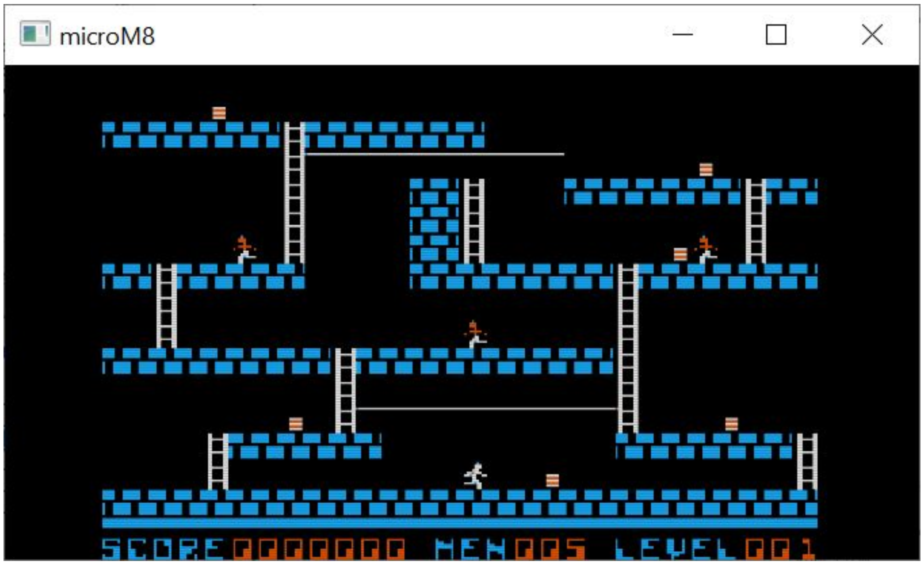
Chapter 1

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⟩≡ (109b) 21 ▷

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

Defines:
`TMP_PTR`, used in chunks 4, 24, and 50b.

4 ⟨*routines* 4⟩≡ (109b) 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, used in chunk 92.
  CLEAR_HGR2, used in chunk 87b.
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.

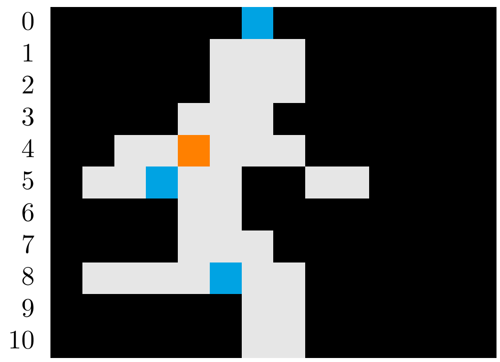
An example would probably be good here. We will take one of the sprites from the game.

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.

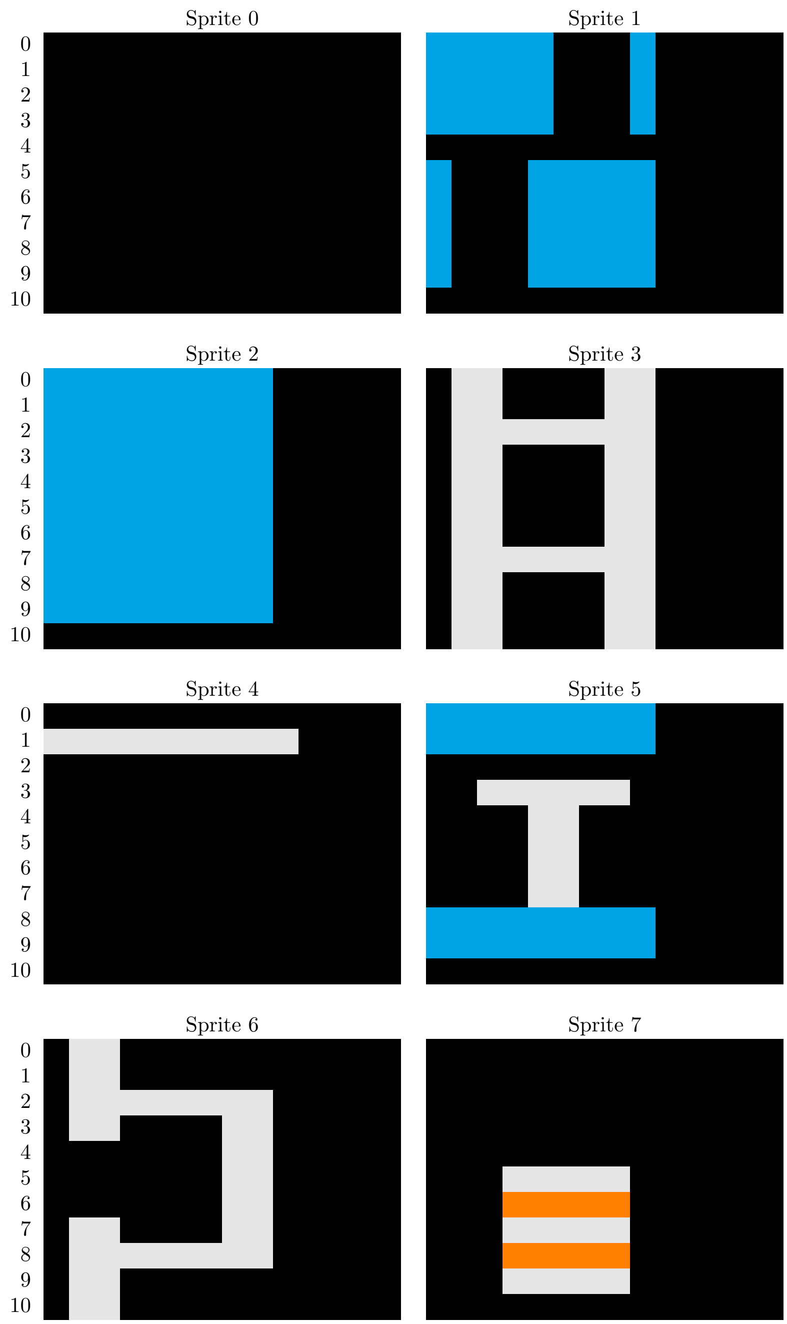
2.2 The sprites

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.

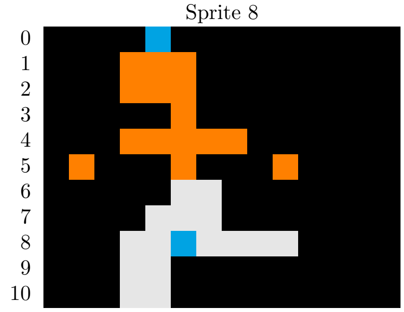
7 ⟨*tables* 7⟩≡ (109b) 22 ▷

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

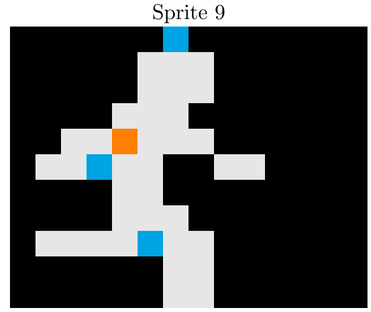
Defines:
`SPRITE_DATA`, used in chunk 24.


Sprite 0
Sprite 1
Sprite 2
Sprite 3
Sprite 4
Sprite 5
Sprite 6
Sprite 7
0
1
2
3
4
5
6
7
8
9
10

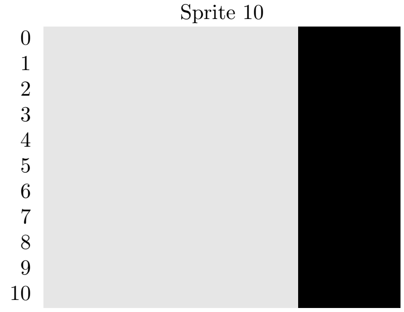
Sprite 8



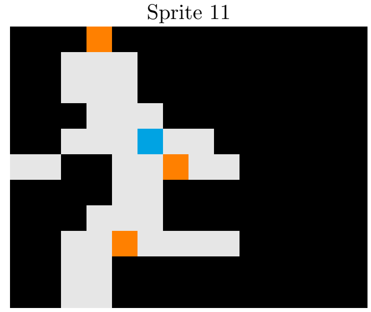
Sprite 9



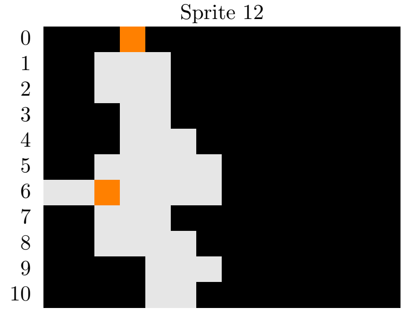
Sprite 10



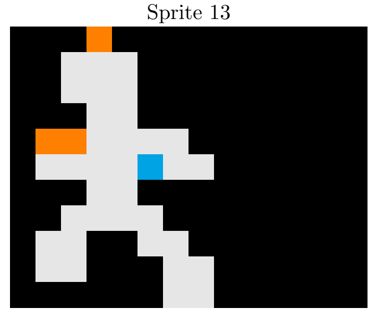
Sprite 11



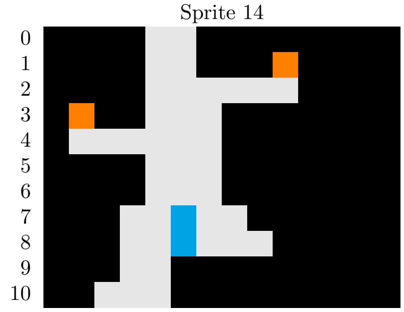
Sprite 12



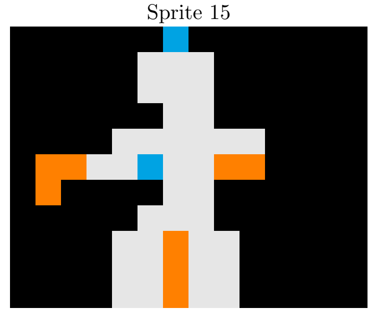
Sprite 13



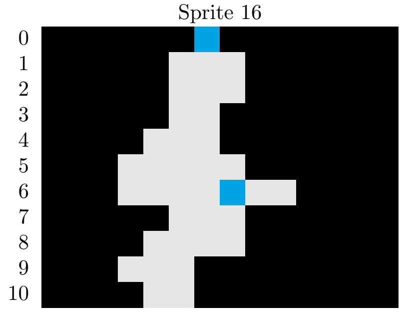
Sprite 14



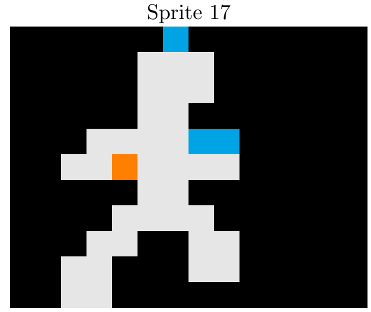
Sprite 15



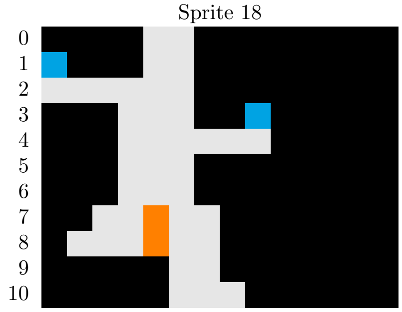
Sprite 16



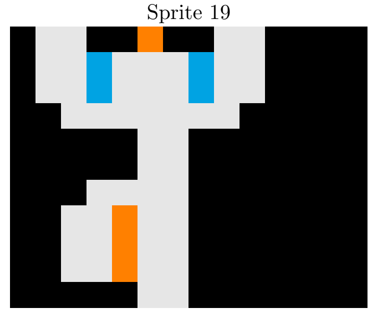
Sprite 17



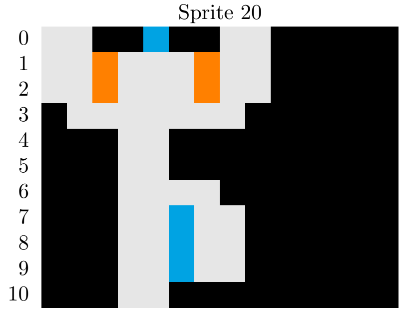
Sprite 18



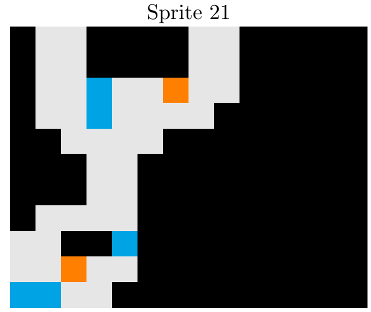
Sprite 19



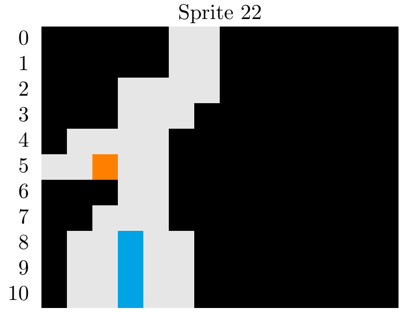
Sprite 20



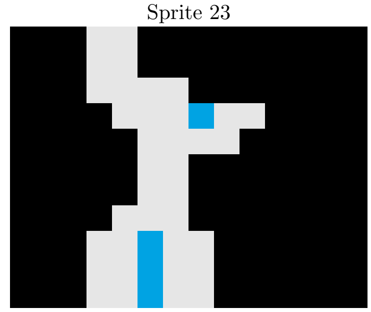
Sprite 21



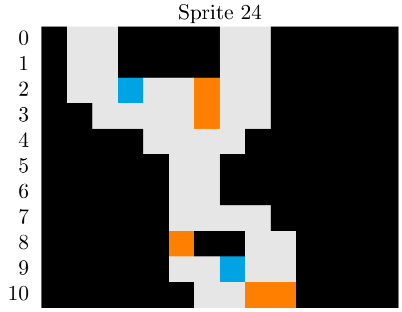
Sprite 22



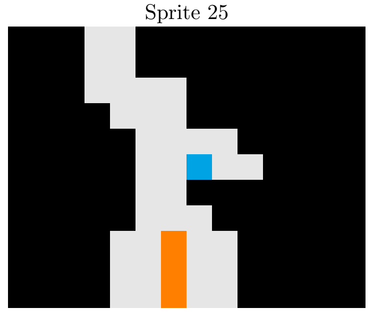
Sprite 23



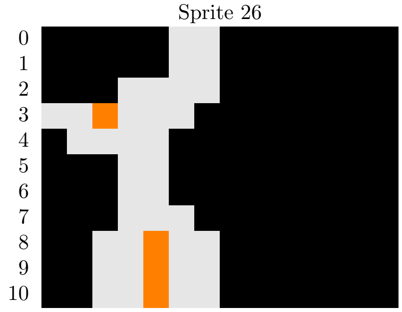
Sprite 24



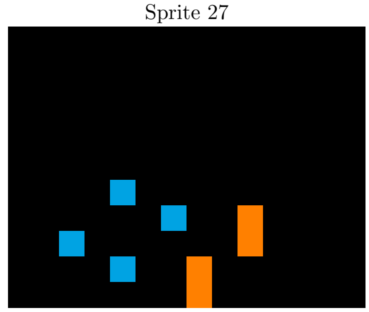
Sprite 25



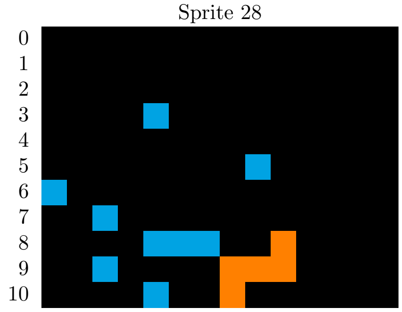
Sprite 26



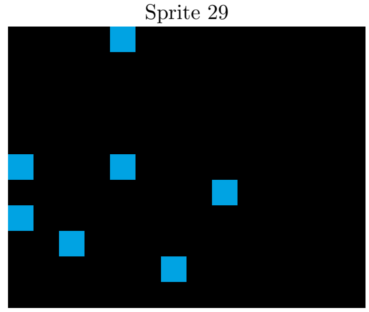
Sprite 27



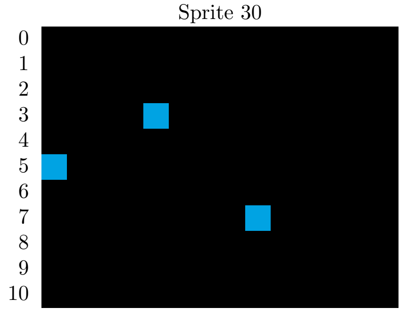
Sprite 28



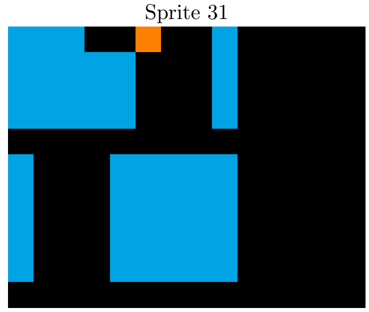
Sprite 29



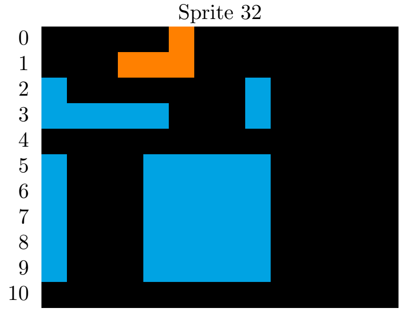
Sprite 30



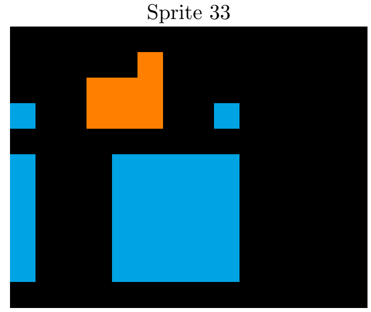
Sprite 31



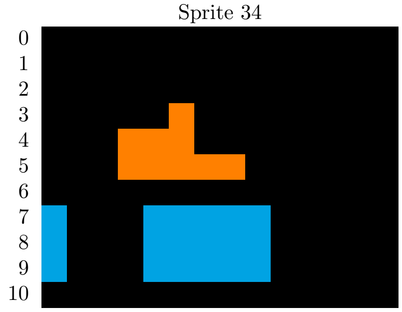
Sprite 32



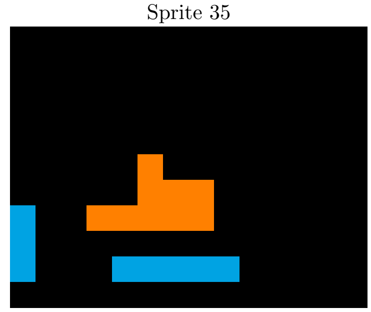
Sprite 33



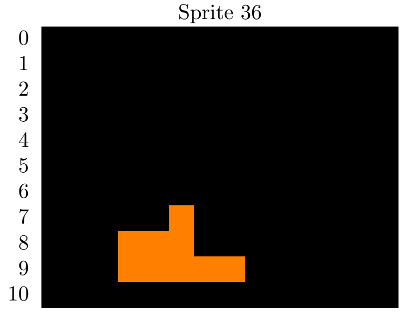
Sprite 34



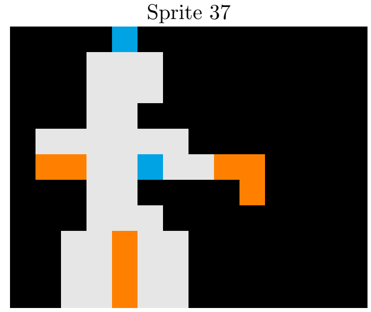
Sprite 35



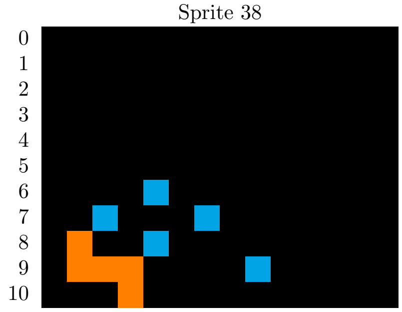
Sprite 36



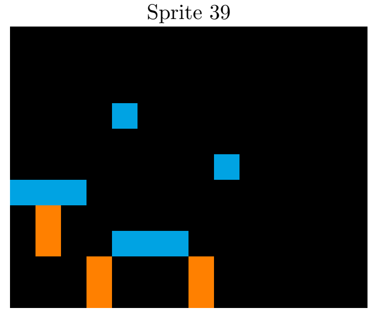
Sprite 37



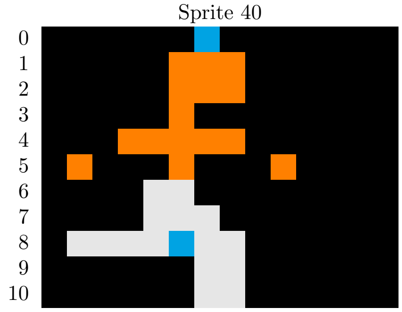
Sprite 38



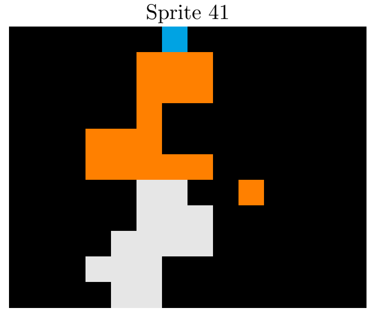
Sprite 39



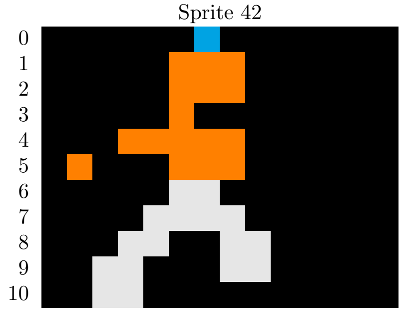
Sprite 40



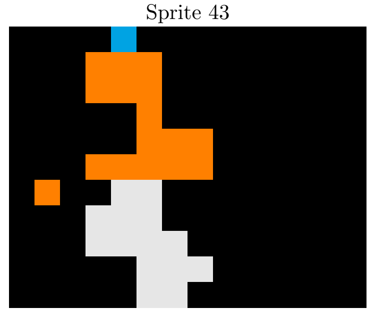
Sprite 41



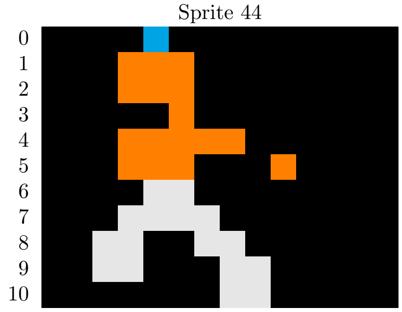
Sprite 42



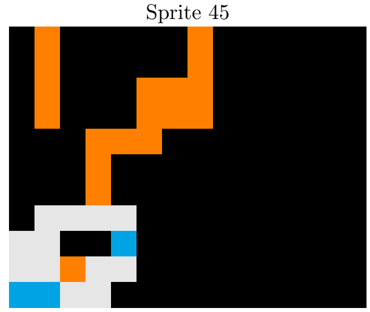
Sprite 43



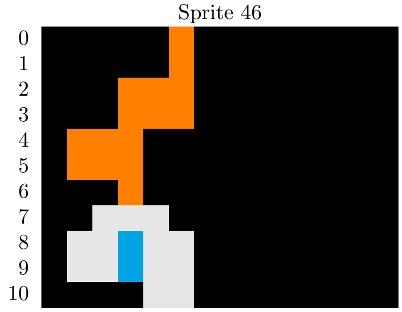
Sprite 44



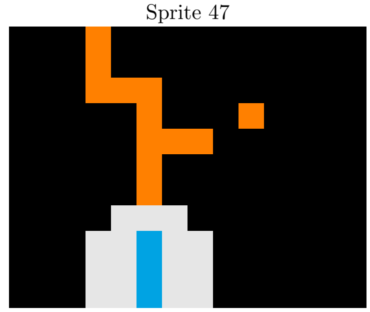
Sprite 45



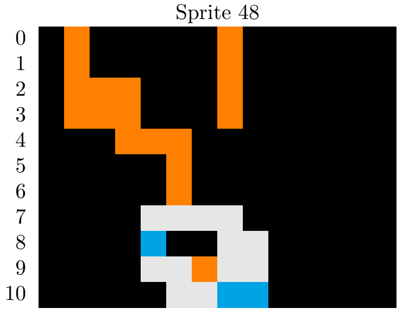
Sprite 46



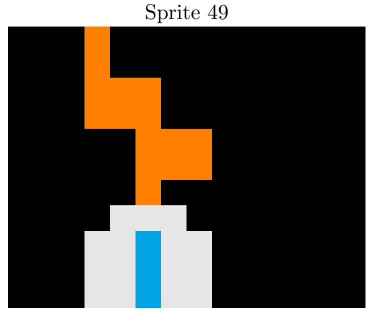
Sprite 47



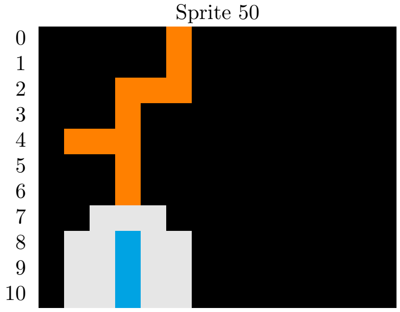
Sprite 48



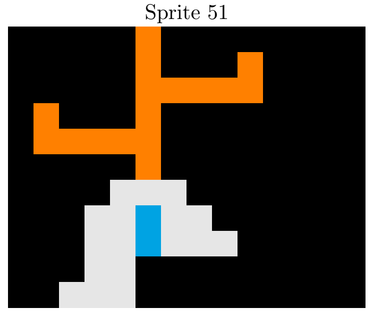
Sprite 49



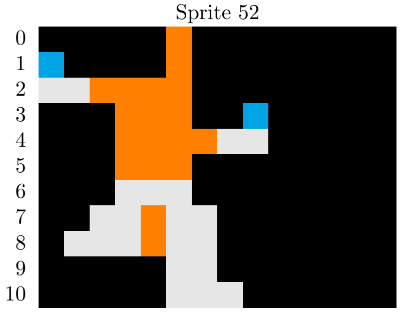
Sprite 50



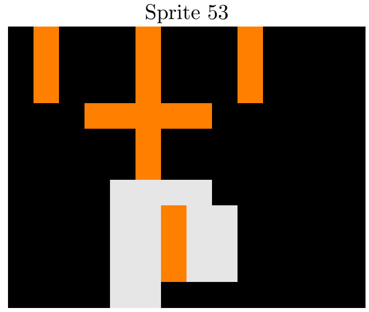
Sprite 51



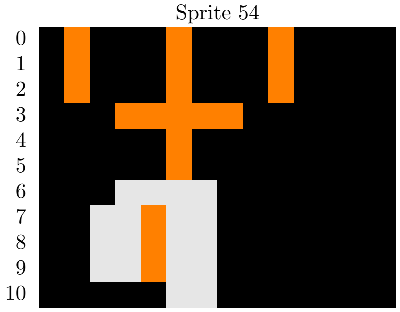
Sprite 52



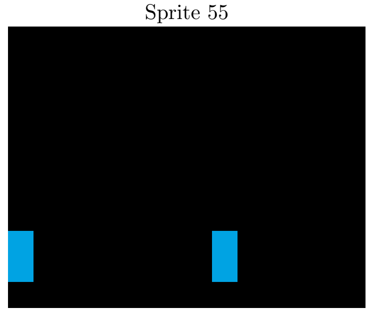
Sprite 53



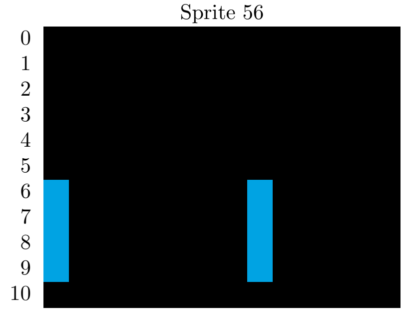
Sprite 54



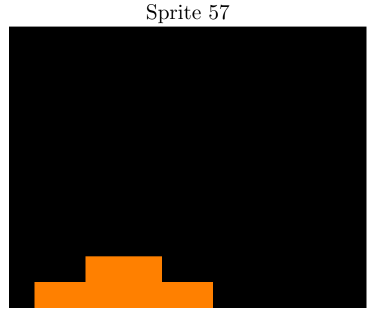
Sprite 55



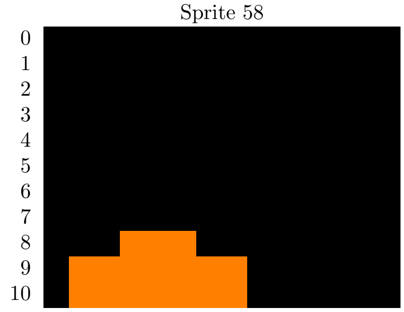
Sprite 56



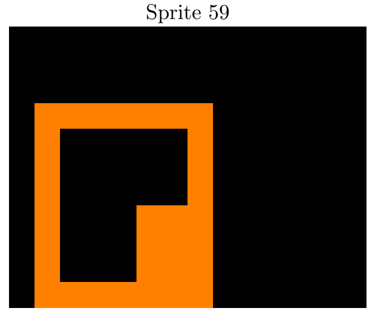
Sprite 57



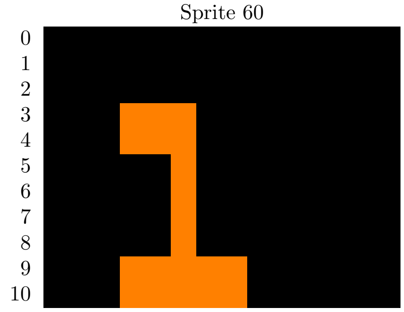
Sprite 58



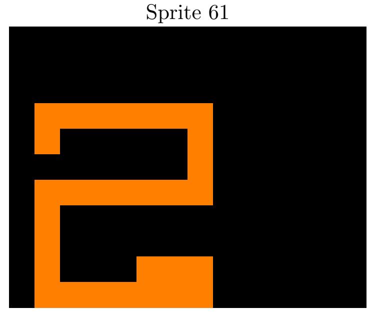
Sprite 59



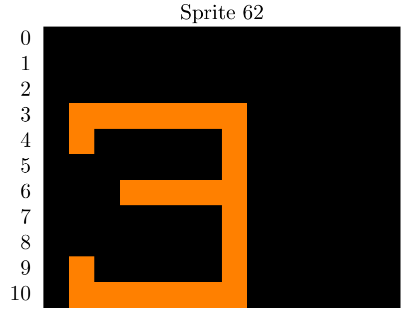
Sprite 60



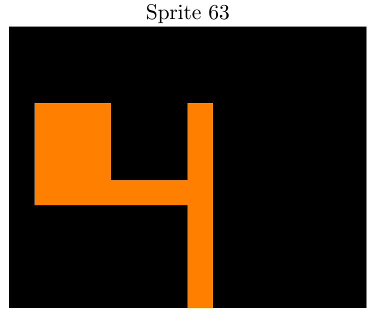
Sprite 61



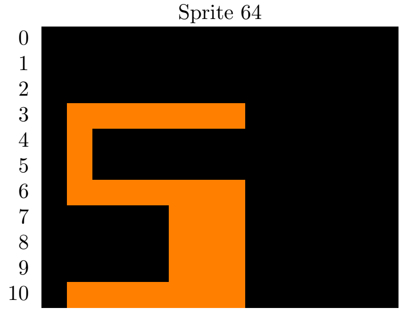
Sprite 62



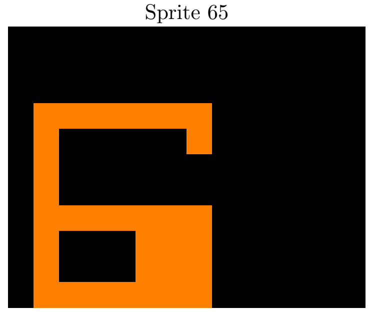
Sprite 63



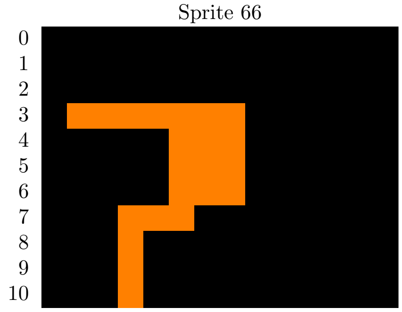
Sprite 64



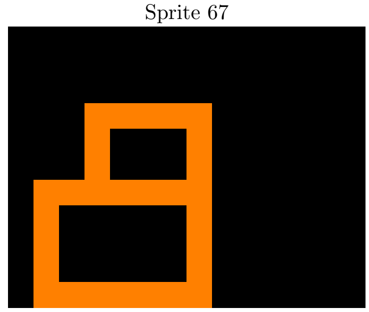
Sprite 65



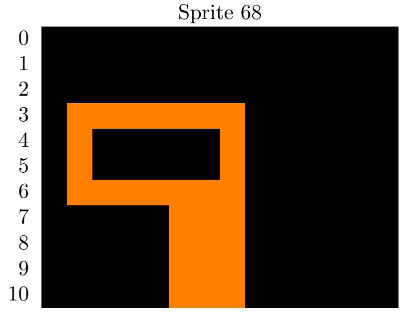
Sprite 66



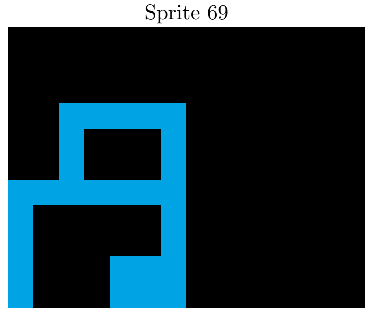
Sprite 67



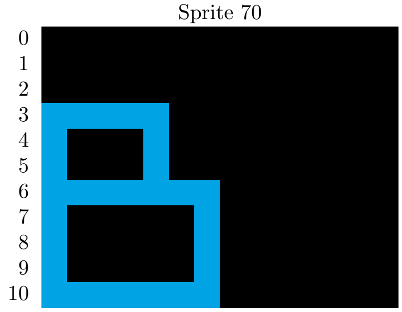
Sprite 68



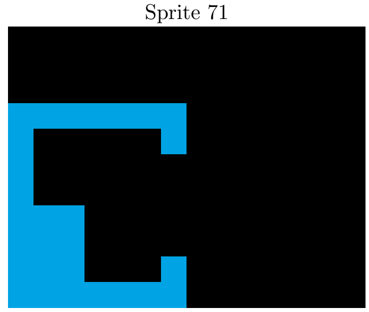
Sprite 69



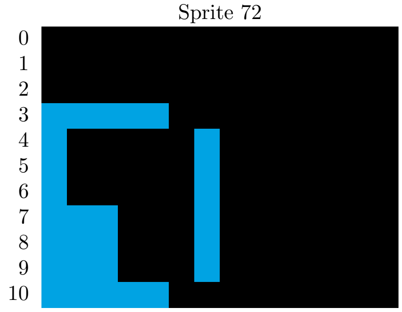
Sprite 70



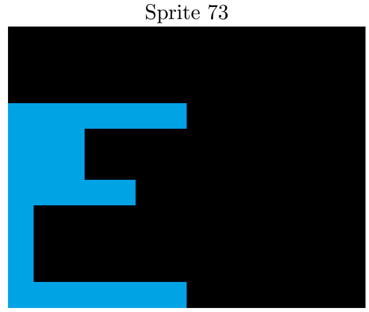
Sprite 71



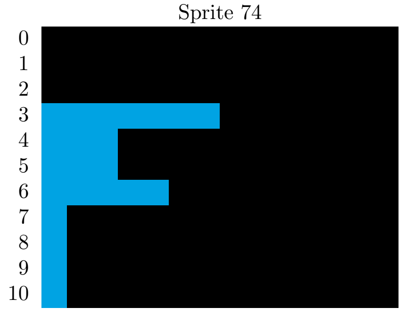
Sprite 72



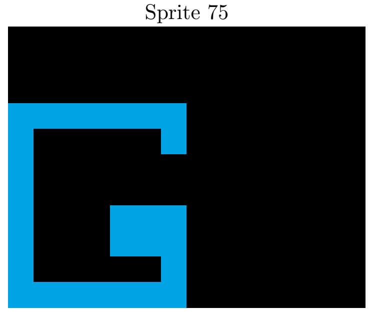
Sprite 73



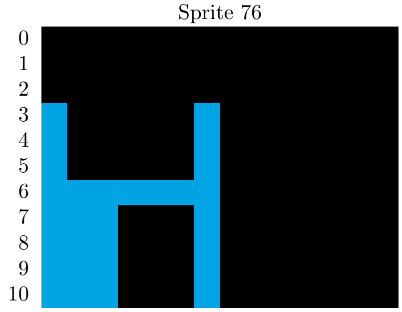
Sprite 74



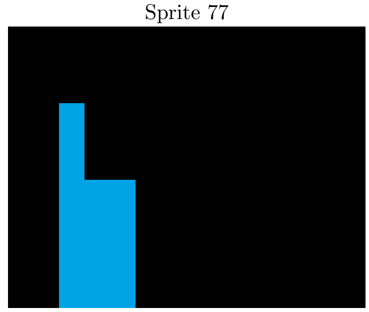
Sprite 75



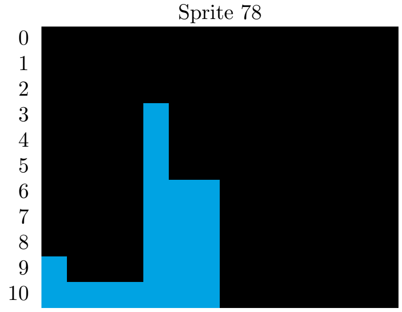
Sprite 76



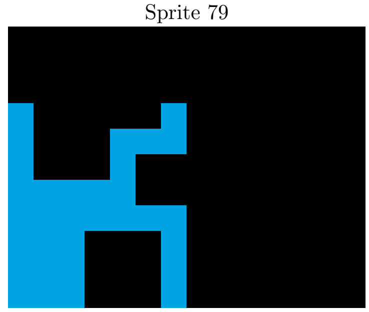
Sprite 77



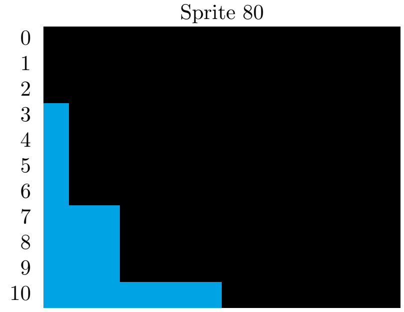
Sprite 78



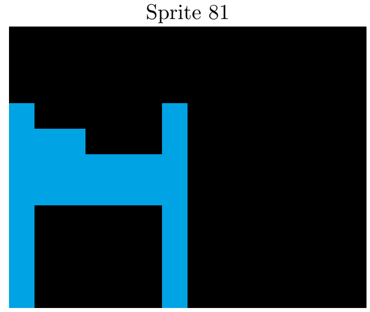
Sprite 79



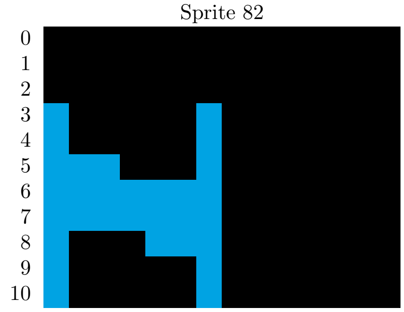
Sprite 80



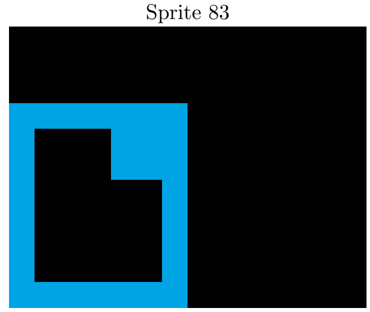
Sprite 81



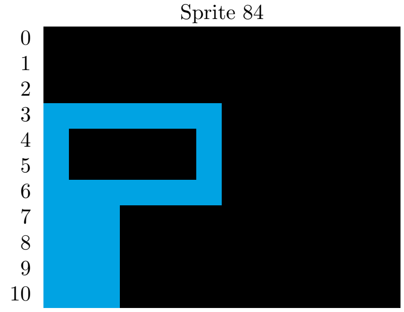
Sprite 82



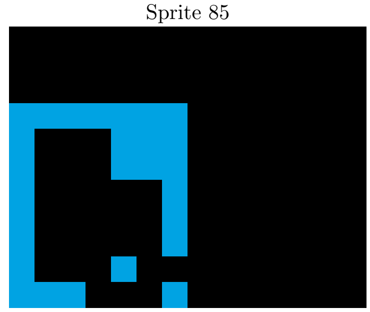
Sprite 83



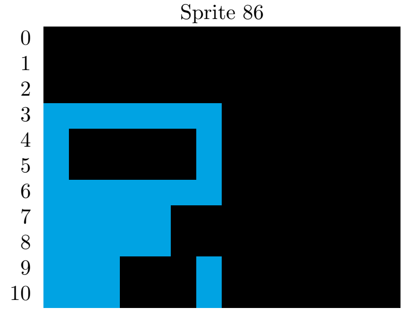
Sprite 84



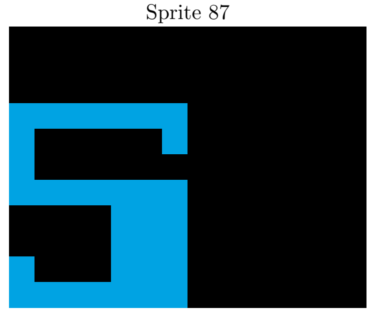
Sprite 85



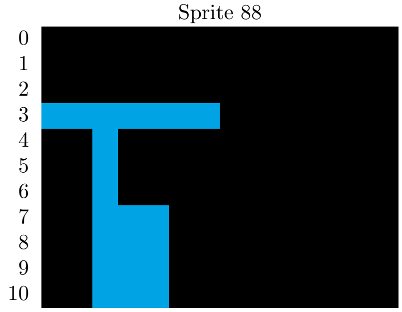
Sprite 86



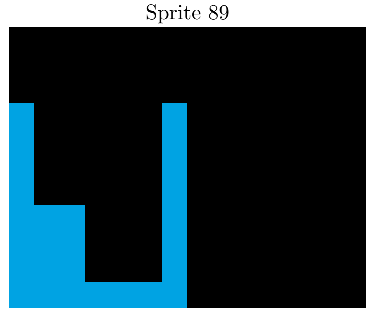
Sprite 87



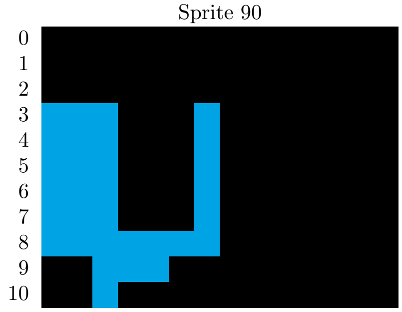
Sprite 88



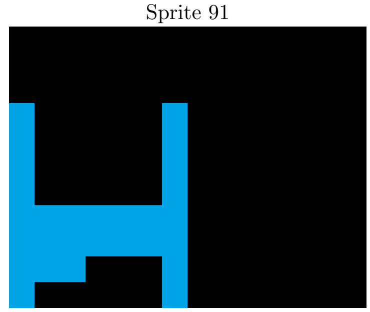
Sprite 89



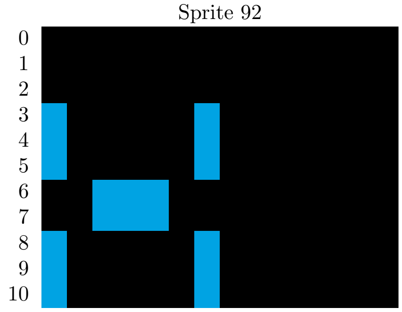
Sprite 90



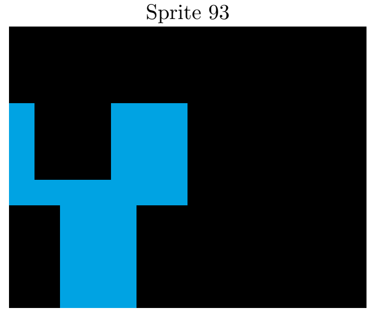
Sprite 91



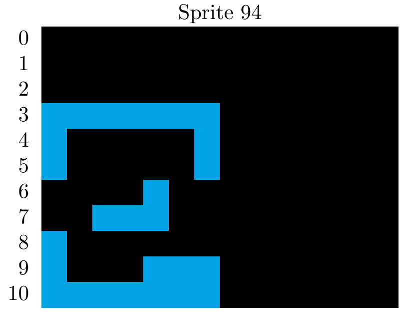
Sprite 92



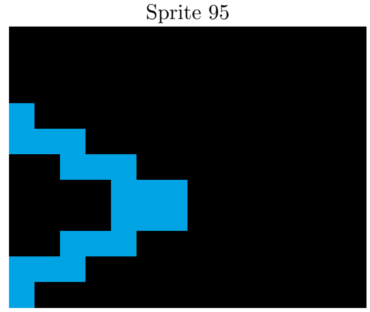
Sprite 93

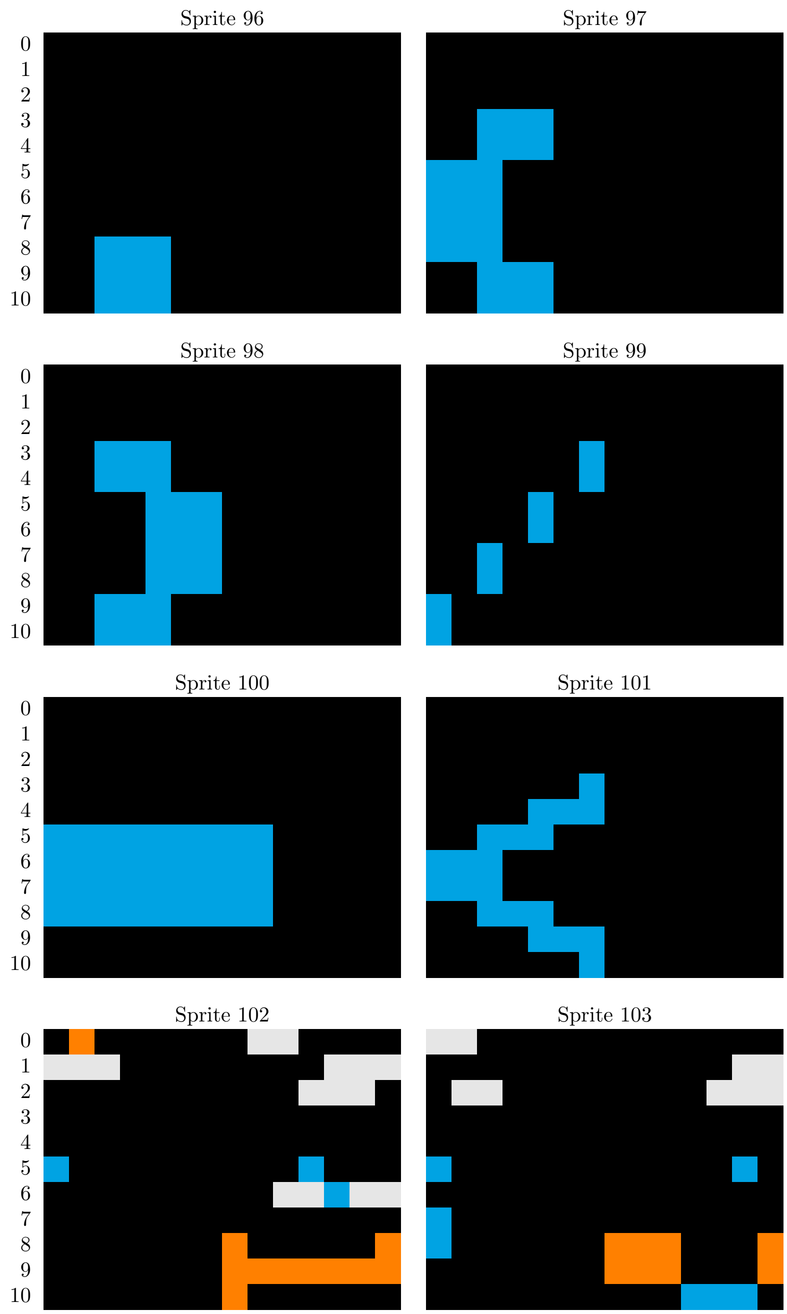


Sprite 94



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⟩+≡ (109b) ◁3 23c▷

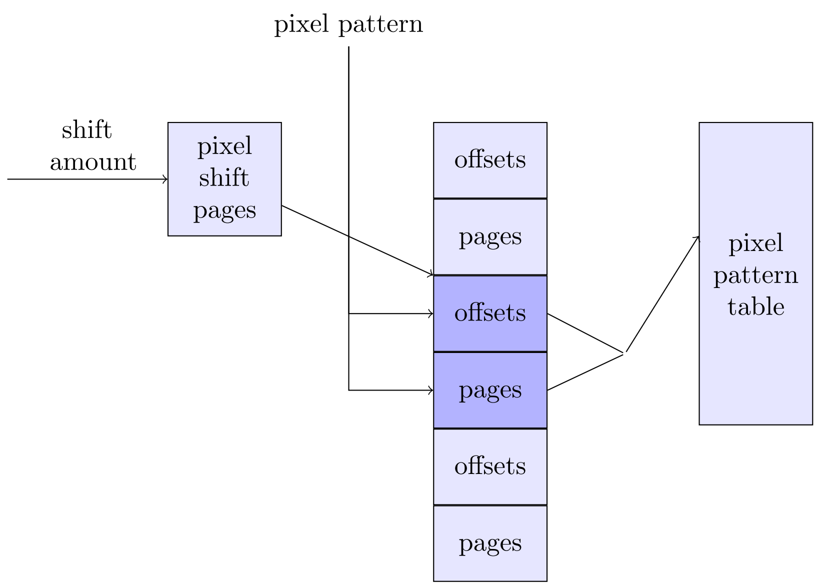
```
        ORG     $DF
BLOCK_DATA      DS      33
```

Defines:
BLOCK_DATA, used in chunks 24, 33, and 36.

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⟩+≡ (109b) ◁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⟩+≡ (109b) ◁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.

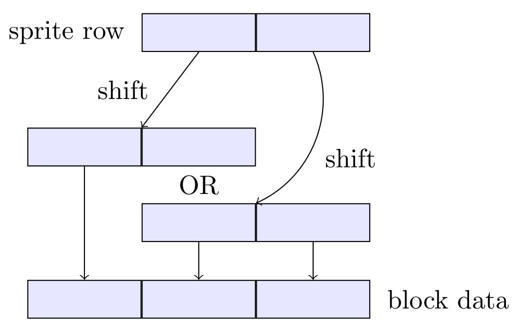
23b ⟨*tables* 7⟩+≡ (109b) ◁23a 26a▷

```
    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⟩+≡ (109b) ◁21 26b▷

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

Defines:
`ROW_COUNT`, used in chunks 24, 33, and 36.
`SPRITE_NUM`, used in chunks 24, 33, 36, and 100a.

24 ⟨*routines 4*⟩+≡ (109b) ◁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 chunks 33 and 36.
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⟩+≡ (109b) ◁23b 28▷

```
    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 27.
ROW_TO_OFFSET_LO, used in chunks 26c and 27.

26b ⟨*defines* 3⟩+≡ (109b) ◁23c 32a▷

```
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, 33, and 92.
ROW_ADDR, used in chunks 26c, 27, 33, 36, 60, 73a, 82, and 93.
ROW_ADDR2, used in chunks 27, 36, 60, and 73a.

26c ⟨*routines* 4⟩+≡ (109b) ◁24 27▷

```
    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 chunks 33 and 93.
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.

27 ⟨*routines* 4⟩+≡ (109b) ◁26c 29a▷

```
    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 36 and 69–72.

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 and vice-versa, we use tables and lookup routines. Each sprite is 10 pixels across by 11 pixels down.

28 ⟨*tables* 7⟩+≡ (109b) ◁26a 30b▷

```
    ORG     $1C35
HALF_SCREEN_COL_TABLE:
    ; 28 cols of 5 double-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
SCREEN_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_BYTE_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
HALF_SCREEN_COL_BYTE_TABLE:
    HEX     00 00 00 00 01 01 01 02 02 02 02 03 03 03 04 04
    HEX     04 04 05 05 05 06 06 06 06 07 07 07 08 08 08 08
    HEX     09 09 09 0A 0A 0A 0A 0B 0B 0B 0C 0C 0C 0C 0D 0D
    HEX     0D 0E 0E 0E 0E 0F 0F 0F 10 10 10 10 11 11 11 12
    HEX     12 12 12 13 13 13 14 14 14 14 15 15 15 16 16 16
    HEX     16 17 17 17 18 18 18 18 19 19 19 1A 1A 1A 1A 1B
    HEX     1B 1B 1C 1C 1C 1C 1D 1D 1D 1E 1E 1E 1E 1F 1F 1F
    HEX     20 20 20 20 21 21 21 22 22 22 22 23 23 23 24 24
    HEX     24 24 25 25 25 26 26 26 26 27 27 27
HALF_SCREEN_COL_SHIFT_TABLE:
    HEX     00 02 04 06 01 03 05 00 02 04 06 01 03 05 00 02
    HEX     04 06 01 03 05 00 02 04 06 01 03 05 00 02 04 06
    HEX     01 03 05 00 02 04 06 01 03 05 00 02 04 06 01 03
    HEX     05 00 02 04 06 01 03 05 00 02 04 06 01 03 05 00
    HEX     02 04 06 01 03 05 00 02 04 06 01 03 05 00 02 04
    HEX     06 01 03 05 00 02 04 06 01 03 05 00 02 04 06 01
    HEX     03 05 00 02 04 06 01 03 05 00 02 04 06 01 03 05
    HEX     00 02 04 06 01 03 05 00 02 04 06 01 03 05 00 02
    HEX     04 06 01 03 05 00 02 04 06 01 03 05
```

Defines:
- COL_BYTE_TABLE, used in chunks 29b and 33.
- COL_SHIFT_TABLE, used in chunks 29b and 33.
- HALF_SCREEN_COL_BYTE_TABLE, used in chunk 30a.
- HALF_SCREEN_COL_SHIFT_TABLE, used in chunk 30a.
- HALF_SCREEN_COL_TABLE, used in chunk 29a.
- SCREEN_ROW_TABLE, used in chunks 29a and 33.

Here is the routine to return the screen coordinates for the given sprite coordinates. The reason that GET_SCREEN_COORDS_FOR returns half the screen column coordinate is that otherwise the screen column coordinate wouldn't fit in a register.

29a ⟨*routines* 4⟩+≡ (109b) ◁27 29b▷

```
    ORG     $885D
GET_SCREEN_COORDS_FOR:
    SUBROUTINE
    ; Enter routine with Y set to sprite row (0-16) and
    ; X set to sprite column (0-27). On return, Y will be set to
    ; screen row, and X is set to half screen column.

    LDA     SCREEN_ROW_TABLE,Y
    PHA
    LDA     HALF_SCREEN_COL_TABLE,X
    TAX                         ; X = HALF_SCREEN_COL_TABLE[X]
    PLA
    TAY                         ; Y = SCREEN_ROW_TABLE[Y]
    RTS
```

Defines:
GET_SCREEN_COORDS_FOR, used in chunks 31, 33, and 101.
Uses HALF_SCREEN_COL_TABLE 28 and SCREEN_ROW_TABLE 28.

This routine takes a sprite column and converts it to the memory-mapped byte offset and right-shift amount.

29b ⟨*routines* 4⟩+≡ (109b) ◁29a 30a▷

```
    ORG     $8868
GET_BYTE_AND_SHIFT_FOR_COL:
    SUBROUTINE
    ; Enter routine with X set to sprite column. On
    ; return, A will be set to screen column byte number
    ; and X will be set to an additional right shift amount.

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

Defines:
GET_BYTE_AND_SHIFT_FOR_COL, used in chunk 33.
Uses COL_BYTE_TABLE 28 and COL_SHIFT_TABLE 28.

This routine takes half the screen column coordinate and converts it to the memory-mapped byte offset and right-shift amount.

30a ⟨*routines* 4⟩+≡ (109b) ◁29b 31a▷

```
    ORG     $8872
GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL:
    SUBROUTINE
    ; Enter routine with X set to half screen column. On
    ; return, A will be set to screen column byte number
    ; and X will be set to an additional right shift amount.

    LDA     HALF_SCREEN_COL_BYTE_TABLE,X
    PHA                         ; A = HALF_SCREEN_COL_BYTE_TABLE[X]
    LDA     HALF_SCREEN_COL_SHIFT_TABLE,X
    TAX                         ; X = HALF_SCREEN_COL_SHIFT_TABLE[X]
    PLA
    RTS
```

Defines:
GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL, used in chunk 36.
Uses HALF_SCREEN_COL_BYTE_TABLE 28 and HALF_SCREEN_COL_SHIFT_TABLE 28.

We also have some utility routines that let us take a sprite row or column and get its screen row or half column, but offset in either row or column by anywhere from -2 to +2.

30b ⟨*tables* 7⟩+≡ (109b) ◁28 31b▷

```
    ORG     $888A
ROW_OFFSET_TABLE:
    HEX     FB FD 00 02 04
```

Defines:
ROW_OFFSET_TABLE, used in chunk 31a.

31a ⟨*routines* 4⟩+≡ (109b) ◁30a 31c▷
```
    ORG     $887C
GET_SCREEN_ROW_OFFSET_IN_X_FOR:
    SUBROUTINE
    ; Enter routine with X set to offset+2 (in double-pixels) and
    ; Y set to sprite row. On return, X will retain its value and
    ; Y will be set to the screen row.

    TXA
    PHA
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    TAX                             ; Restore X
    TYA
    CLC
    ADC     ROW_OFFSET_TABLE,X
    TAY
    RTS
```
Defines:
GET_SCREEN_ROW_OFFSET_IN_X_FOR, used in chunk 100a.
Uses GET_SCREEN_COORDS_FOR 29a and ROW_OFFSET_TABLE 30b.

31b ⟨*tables* 7⟩+≡ (109b) ◁30b 32b▷
```
    ORG     $889D
COL_OFFSET_TABLE:
    HEX     FE FF 00 01 02
```
Defines:
COL_OFFSET_TABLE, used in chunk 31c.

31c ⟨*routines* 4⟩+≡ (109b) ◁31a 33▷
```
    ORG     $888F
GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR:
    SUBROUTINE
    ; Enter routine with Y set to offset+2 (in double-pixels) and
    ; X set to sprite column. On return, Y will retain its value and
    ; X will be set to the half screen column.

    TYA
    PHA
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    TAY                             ; Restore Y
    TXA
    CLC
    ADC     COL_OFFSET_TABLE,Y
    TAX
    RTS
```
Defines:
GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR, used in chunk 100a.
Uses COL_OFFSET_TABLE 31b and GET_SCREEN_COORDS_FOR 29a.

Now we can finally write the routines that draw a sprite on the screen. We have one routine that draws a sprite at a given game row and game column. There are two entry points, one to draw on HGR1, and one for HGR2.

32a ⟨*defines* 3⟩+≡ (109b) ◁26b 39▷

```
ROWNUM         EQU    $1B
COLNUM         EQU    $1C
MASK0          EQU    $50
MASK1          EQU    $51
COL_SHIFT_AMT  EQU    $71
GAME_COLNUM    EQU    $85
GAME_ROWNUM    EQU    $86
```

Defines:
- `COL_SHIFT_AMT`, used in chunks 33 and 36.
- `COLNUM`, used in chunks 33 and 36.
- `GAME_COLNUM`, used in chunks 33, 40a, 42a, 45, 47, 54d, 59a, 61, 85a, 87b, 101, and 107.
- `GAME_ROWNUM`, used in chunks 33, 40a, 45, 47, 53, 56–59, 61, 83, 85b, 87b, 92, 93, 95c, 96c, 99a, 101, and 107.
- `MASK0`, used in chunk 33.
- `MASK1`, used in chunk 33.
- `ROWNUM`, used in chunks 33 and 36.

32b ⟨*tables* 7⟩+≡ (109b) ◁31b 54a▷

```
    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 33.
- `PIXEL_MASK1`, used in chunk 33.

33 ⟨*routines* 4⟩+≡ (109b) ◁31c 38▷

```
    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_SCREEN_COORDS_FOR
    STY     ROWNUM              ; ROWNUM = SCREEN_ROW_TABLE[GAME_ROWNUM]

    LDX     GAME_COLNUM
    JSR     GET_BYTE_AND_SHIFT_FOR_COL
    STA     COLNUM              ; COLNUM = COL_BYTE_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 40a and 42a.
DRAW_SPRITE_PAGE2, used in chunks 40a, 42a, 59a, 61, and 101.

Uses BLOCK_DATA 21, COL_BYTE_TABLE 28, COL_SHIFT_AMT 32a, COL_SHIFT_TABLE 28, COLNUM 32a, COMPUTE_SHIFTED_SPRITE 24, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_BYTE_AND_SHIFT_FOR_COL 29b, GET_SCREEN_COORDS_FOR 29a, HGR_PAGE 26b, MASK0 32a, MASK1 32a, PIXEL_MASK0 32b, PIXEL_MASK1 32b, ROW_ADDR 26b, ROW_COUNT 23c, ROW_TO_ADDR 26c, ROWNUM 32a, SCREEN_ROW_TABLE 28, and SPRITE_NUM 23c.

There is a different routine which draws a sprite at a given screen coordinate. Upon entry, the Y register needs to be set to the screen row coordinate (0-191). However, the X register needs to be set to half the screen column coordinate (0-139) because otherwise the maximum coordinate (279) wouldn't fit in a register.

36 ⟨*draw sprite at screen coordinate* 36⟩≡

```
    ORG     $8336
DRAW_SPRITE_AT_PIXEL_COORDS:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; Y set to the screen row to draw it at, and X
    ; set to *half* the screen column to draw it at.

    STY     ROWNUM
    STA     SPRITE_NUM
    JSR     GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL
    STA     COLNUM
    STX     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_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y
    INX
    INY
    LDA     BLOCK_DATA+1,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y
    INX
    INX
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .loop1
    RTS
```

```
.need_3_bytes:
    LDY     ROWNUM
    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     BLOCK_DATA,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y
    INX
    INY
    LDA     BLOCK_DATA+1,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y
    INX
    INY
    LDA     BLOCK_DATA+2,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y
    INX
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .need_3_bytes
    RTS
```

Defines:
  DRAW_SPRITE_AT_PIXEL_COORDS, used in chunks 101 and 103.

Uses BLOCK_DATA 21, COL_SHIFT_AMT 32a, COLNUM 32a, COMPUTE_SHIFTED_SPRITE 24, GET_BYTE_AND_SHIFT_FOR_HALF_SCREEN_COL 30a, ROW_ADDR 26b, ROW_ADDR2 26b, ROW_COUNT 23c, ROW_TO_ADDR_FOR_BOTH_PAGES 27, ROWNUM 32a, 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.

38 ⟨*routines* 4⟩+≡ (109b) ◁33 40a▷

```
    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 40a.

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

39 ⟨*defines* 3⟩+≡ (109b) ◁32a 40b▷

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

Defines:
DRAW_PAGE, used in chunks 40a, 42a, 87b, 91, and 92.

40a ⟨*routines* 4⟩+≡ (109b) ◁38 41▷

```
      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`, used in chunk 90b.
  `PUT_CHAR`, used in chunks 41, 88c, and 89b.
Uses `CHAR_TO_SPRITE_NUM` 38, `DRAW_PAGE` 39, `DRAW_SPRITE_PAGE1` 33, `DRAW_SPRITE_PAGE2` 33, `GAME_COLNUM` 32a, and `GAME_ROWNUM` 32a.

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.

40b ⟨*defines* 3⟩+≡ (109b) ◁39 42b▷

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

Defines:
  `SAVED_RET_ADDR`, used in chunks 41 and 49.

41 ⟨*routines* 4⟩+≡ (109b) ◁40a 42a▷

```
      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, used in chunks 88 and 89.
Uses PUT_CHAR 40a and SAVED_RET_ADDR 40b.

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.

42a ⟨*routines* 4⟩+≡ (109b) ◁41 43▷

```
      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 45, 47, and 88–90.
Uses DRAW_PAGE 39, DRAW_SPRITE_PAGE1 33, DRAW_SPRITE_PAGE2 33, and GAME_COLNUM 32a.

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.

42b ⟨*defines* 3⟩+≡ (109b) ◁40b 44b▷

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

Defines:
HUNDREDS, used in chunks 43, 47, and 89c.
TENS, used in chunks 43–45, 47, 89c, and 90a.
UNITS, used in chunks 43–45, 47, 89c, and 90a.

43 ⟨*routines* 4⟩+≡ (109b) ◁42a 44a▷

```
    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 chunks 47 and 89c.
Uses HUNDREDS 42b, TENS 42b, and UNITS 42b.

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

44a ⟨*routines* 4⟩+≡ (109b) ◁43 45▷

```
      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 chunks 45 and 90a.
Uses TENS 42b and UNITS 42b.

2.7 Score and status

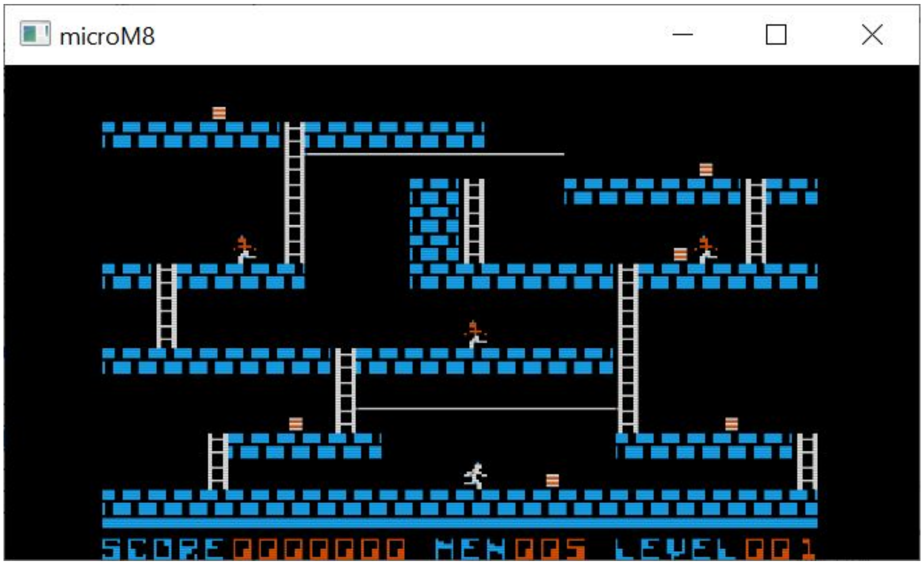
Lode Runner stores your score as an 8-digit BCD number.

44b ⟨*defines* 3⟩+≡ (109b) ◁42b 46▷

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

Defines:
SCORE, used in chunks 45, 88a, 101, and 106.

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.

45 ⟨*routines* 4⟩+≡ (109b) ◁44a 47▷

```
      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`, used in chunk 101.
Uses `BCD_TO_DECIMAL2` 44a, `GAME_COLNUM` 32a, `GAME_ROWNUM` 32a, `PUT_DIGIT` 42a, `SCORE` 44b, `TENS` 42b, and `UNITS` 42b.

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

46 ⟨*defines* 3⟩+≡ (109b) ◁44b 48▷

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

Defines:
  `LEVELNUM`, used in chunks 47, 82a, 97a, and 98a.
  `LIVES`, used in chunks 47 and 106.

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.

47 ⟨*routines* 4⟩+≡ (109b) ◁45 65▷

```
    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 63.
PUT_STATUS_LIVES, used in chunk 63.
Uses GAME_COLNUM 32a, GAME_ROWNUM 32a, HUNDREDS 42b, LEVELNUM 46, LIVES 46, PUT_DIGIT 42a, TENS 42b, TO_DECIMAL3 43, and UNITS 42b.

Chapter 3

Sound

3.1 Sound "strings"

A sound "string" describes a sound to play in terms of pitch and duration, ending in a 00. Just like in the PUT_STRING routine, rather than take an address pointing to a sound 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.

Because NOTE_INDEX is not zeroed out, this actually appends to the sound data buffer.

The format of a sound string is duration, followed by pitch, although the pitch is lower for higher numbers.

One example of a sound string is 07 45 06 55 05 44 04 54 03 43 02 53, found in CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER.

48 ⟨*defines* 3⟩+≡ (109b) ◁46 50a▷

```
NOTE_INDEX      EQU     $54
SOUND_DURATION  EQU     $0E00       ; 128 bytes
SOUND_PITCH     EQU     $0E80       ; 128 bytes
```

Defines:
- NOTE_INDEX, used in chunks 49, 52, and 107.
- SOUND_DURATION, used in chunks 49 and 52.
- SOUND_PITCH, used in chunks 49 and 52.

49 ⟨*load sound data* 49⟩≡ (109a)

```
      ORG     $87E1
  LOAD_SOUND_DATA:
      SUBROUTINE

      PLA
      STA     SAVED_RET_ADDR
      PLA
      STA     SAVED_RET_ADDR+1
      BNE     .next

  .loop:
      LDY     #$00
      LDA     (SAVED_RET_ADDR),Y
      BEQ     .end
      INC     NOTE_INDEX
      LDX     NOTE_INDEX
      STA     SOUND_DURATION,X
      INY
      LDA     (SAVED_RET_ADDR),Y
      STA     SOUND_PITCH,X

      INC     SAVED_RET_ADDR
      BNE     .next
      INC     SAVED_RET_ADDR+1

  .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:
  LOAD_SOUND_DATA, used in chunk 101.
Uses NOTE_INDEX 48, SAVED_RET_ADDR 40b, SOUND_DURATION 48, and SOUND_PITCH 48.

3.2 Playing notes

The PLAY_NOTE routines plays a note through the built-in speaker. The time the note is played is based on X and Y forming a 16-bit counter (X being the most significant byte), but A controls the pitch, which is how often the speaker is clicked. The higher A, the lower the pitch.

The ENABLE_SOUND location can also disable playing the note, but the routine still takes as long as it would have.

50a ⟨*defines* 3⟩+≡ (109b) ◁48 51b▷

```
ENABLE_SOUND    EQU     $99     ; If 0, do not click speaker.
SPKR            EQU     $C030   ; Access clicks the speaker.
```

Defines:
ENABLE_SOUND, used in chunk 50b.
SPKR, used in chunk 50b.

50b ⟨*play note* 50b⟩≡ (109a)

```
    ORG     $87BA
PLAY_NOTE:
    SUBROUTINE

    STA     TMP_PTR
    STX     TMP_PTR+1

.loop:
    LDA     ENABLE_SOUND
    BEQ     .decrement_counter
    LDA     SPKR

.decrement_counter:
    DEY
    BNE     .counter_decremented
    DEC     TMP_PTR+1
    BEQ     .end

.counter_decremented:
    DEX
    BNE     .decrement_counter
    LDX     TMP_PTR
    JMP     .loop

.end:
    RTS
```

Defines:
PLAY_NOTE, used in chunks 52 and 103.
Uses ENABLE_SOUND 50a, SPKR 50a, and TMP_PTR 3.

3.3 Playing a sound

The SOUND_DELAY routine delays an amount of time based on the X register. The total number of cycles is about 905 per each X. Since the Apple //e clock cycle was 980 nsec (on an NTSC system), this routine would delay approximately 887 microseconds times X. PAL systems were very slightly slower (by 0.47%), which corresponds to 883 microseconds times X.

51a ⟨*sound delay* 51a⟩≡ (109a)

```
    ORG     $86B5
SOUND_DELAY:
    SUBROUTINE

    LDY     #$B4        ; 180
.loop:
    DEY                 ; 2 cycles
    BNE     .loop       ; 3 cycles
    DEX                 ; 2 cycles
    BNE     .loop       ; 3 cycles
    RTS
```

Defines:
SOUND_DELAY, used in chunk 52.

Finally, the PLAY_SOUND routine plays one section of the sound string stored in the SOUND_PITCH and SOUND_DURATION buffers. We have to break up the playing of the sound so that gameplay doesn't pause while playing the sound, although game play does pause while playing the note.

Alternatively, if there is no sound string, we can play the note stored in location \$A4 as long as location \$9B is zero. The duration is 2 + SOUND_PERIOD.

The routine is designed to delay approximately the same amount regardless of sound duration. The delay is controlled by SOUND_PERIOD. This value is hardcoded to 6.

51b ⟨*defines* 3⟩+≡ (109b) ◁50a 54b▷

```
    ORG     $8C
SOUND_PERIOD:
    HEX     06
```

Defines:
SOUND_PERIOD, used in chunk 52.

52 ⟨*play sound 52*⟩≡ (109a)

```
    ORG     $8811
PLAY_SOUND:
    SUBROUTINE

    LDY     NOTE_INDEX
    BEQ     .no_more_notes
    LDA     SOUND_PITCH,Y
    LDX     SOUND_DURATION,Y
    JSR     PLAY_NOTE

    LDY     NOTE_INDEX              ; Y = NOTE_INDEX
    DEC     NOTE_INDEX              ; NOTE_INDEX--
    LDA     SOUND_PERIOD
    SEC
    SBC     SOUND_DURATION,Y        ; A = SOUND_PERIOD - SOUND_DURATION[Y]
    BEQ     .done
    BCC     .done                   ; If A <= 0, done.
    TAX
    JSR     SOUND_DELAY

.done:
    SEC
    RTS

.no_more_notes:
    LDA     $9B
    BNE     .end
    LDA     $A4
    LSR                     ; pitch = $A4 >> 1
    INC     $A4             ; $A4++
    LDX     SOUND_PERIOD
    INX
    INX                     ; duration = SOUND_PERIOD + 2
    JSR     PLAY_NOTE

    CLC
    RTS

.end:
    LDX     SOUND_PERIOD
    JSR     SOUND_DELAY

    CLC
    RTS
```

Defines:
`PLAY_SOUND`, never used.

Uses `NOTE_INDEX` 48, `PLAY_NOTE` 50b, `SOUND_DELAY` 51a, `SOUND_DURATION` 48, `SOUND_PERIOD` 51b, and `SOUND_PITCH` 48.

Chapter 4

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.

4.1 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:

53 ⟨*level draw routine* 53⟩≡ (109a) 54c ▷

```
      ORG     $63B3
  DRAW_LEVEL_PAGE2:
      SUBROUTINE
      ; Returns carry set if there was no player sprite in the level,
      ; or carry clear if there was.

      LDY     15
      STY     GAME_ROWNUM

  .row_loop:
```

Defines:
DRAW_LEVEL_PAGE2, used in chunk 84a.
Uses GAME_ROWNUM 32a.

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.

54a ⟨*tables* 7⟩+≡ (109b) ◁32b 55d▷

```
    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 54c, 61, 84b, 101, and 103.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES, used in chunks 54c, 61, 84b, and 103.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2, used in chunks 54c, 84b, 101, and 103.

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

54b ⟨*defines* 3⟩+≡ (109b) ◁51b 55a▷

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

Defines:
PTR1, used in chunks 54, 56b, 61, 84b, 85a, and 103.
PTR2, used in chunks 54c, 56–58, 84b, 85a, 101, and 103.

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.

54c ⟨*level draw routine* 53⟩+≡ (109a) ◁53 54d▷

```
    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 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 54a, PTR1 54b, and PTR2 54b.

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

54d ⟨*level draw routine* 53⟩+≡ (109a) ◁54c 54e▷

```
    LDY     27
    STY     GAME_COLNUM

.col_loop:
```

Uses GAME_COLNUM 32a.

We load the sprite from the level data.

54e ⟨*level draw routine* 53⟩+≡ (109a) ◁54d 55c▷

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

Uses PTR1 54b.

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.

55a ⟨*defines* 3⟩+≡ (109b) ◁54b 55b▷

```
    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 chunks 56c, 83, and 101.
GUARD_COUNT, used in chunks 57b, 83, and 107.
LADDER_COUNT, used in chunks 56a and 83.
PLAYER_COL, used in chunks 58c, 59b, 83, 100a, 101, 103, and 107.
PLAYER_ROW, used in chunks 58c, 100a, 101, 103, and 107.

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.

55b ⟨*defines* 3⟩+≡ (109b) ◁55a 58b▷

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

Defines:
VERBATIM, used in chunks 55c, 59b, and 82c.

55c ⟨*level draw routine* 53⟩+≡ (109a) ◁54e 56a▷

```
    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 55b.

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.

55d ⟨*tables* 7⟩+≡ (109b) ◁54a 57a▷

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

Defines:
LADDER_LOCS_COL, used in chunk 56a.
LADDER_LOCS_ROW, used in chunk 56a.

56a ⟨*level draw routine* 53⟩+≡ (109a) ◁55c 56b▷

```
    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 32a, LADDER_COUNT 55a, LADDER_LOCS_COL 55d, and LADDER_LOCS_ROW 55d.

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

56b ⟨*level draw routine* 53⟩+≡ (109a) ◁56a 56c▷

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

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

Uses PTR1 54b and PTR2 54b.

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

56c ⟨*level draw routine* 53⟩+≡ (109a) ◁56b 57b▷

```
.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 55a.

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.

57a ⟨*tables* 7⟩+≡ (109b) ◁55d 73b▷

```
    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 57b.
GUARD_FLAGS_0C78, used in chunk 57b.
GUARD_FLAGS_0C80, used in chunk 57b.
GUARD_FLAGS_0C88, used in chunk 57b.
GUARD_LOCS_COL, used in chunk 57b.
GUARD_LOCS_ROW, used in chunk 57b.

57b ⟨*level draw routine* 53⟩+≡ (109a) ◁56c 58a▷

```
.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 32a, GUARD_COUNT 55a, GUARD_FLAGS_0C70 57a, GUARD_FLAGS_0C78 57a, GUARD_FLAGS_0C80 57a, GUARD_FLAGS_0C88 57a, GUARD_LOCS_COL 57a, GUARD_LOCS_ROW 57a, and PTR2 54b.

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

58a ⟨*level draw routine* 53⟩+≡ (109a) ◁57b 58c▷
```
.next_row:
    BPL     .row_loop
.next_col:
    BPL     .col_loop
```

Next we check for sprite 9, the player.

58b ⟨*defines* 3⟩+≡ (109b) ◁55b 62▷
```
PLAYER_X_ADJ                EQU     $02     ; [0-4] minus 2 (so 2 = right on the sprite location)
PLAYER_Y_ADJ                EQU     $03     ; [0-4] minus 2 (so 2 = right on the sprite location)
PLAYER_ANIM_STATE           EQU     $04     ; Index into SPRITE_ANIM_SEQS
PLAYER_FACING_DIRECTION     EQU     $05     ; Hi bit set: facing left, otherwise facing right
```
Defines:
PLAYER_ANIM_STATE, used in chunks 58c, 100, and 103.
PLAYER_X_ADJ, used in chunks 58c, 100a, and 101.
PLAYER_Y_ADJ, used in chunks 58c, 100a, 101, and 103.
Uses SPRITE_ANIM_SEQS 99b.

58c ⟨*level draw routine* 53⟩+≡ (109a) ◁58a 58d▷
```
.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_X_ADJ
    STX     PLAYER_Y_ADJ            ; Set Player X and Y movement to 0.
    LDX     #$08
    STX     PLAYER_ANIM_STATE       ; Corresponds to sprite 9 (see SPRITE_ANIM_SEQS)

    LDA     #$00
    STA     (PTR2),Y
    LDA     #$09
    BNE     .draw_sprite            ; Unconditional jump.
```
Uses GAME_ROWNUM 32a, PLAYER_ANIM_STATE 58b, PLAYER_COL 55a, PLAYER_ROW 55a, PLAYER_X_ADJ 58b, PLAYER_Y_ADJ 58b, PTR2 54b, and SPRITE_ANIM_SEQS 99b.

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.

58d ⟨*level draw routine* 53⟩+≡ (109a) ◁58c 59a▷
```
.check_for_5:
    CMP     #$05
    BNE     .draw_sprite
    LDA     #$01                    ; Brick sprite
```

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

59a ⟨*level draw routine* 53⟩+≡ (109a) ◁58d 59b▷
```
  .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 33, GAME_COLNUM 32a, and GAME_ROWNUM 32a.

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!

59b ⟨*level draw routine* 53⟩+≡ (109a) ◁59a 60▷
```
      LDA     VERBATIM
      BEQ     .copy_page2_to_page1

      LDA     PLAYER_COL
      BPL     .reveal_screen

      SEC                             ; Oops, no player! Return error.
      RTS
```
Uses PLAYER_COL 55a and VERBATIM 55b.

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

60 ⟨*level draw routine* 53⟩+≡ (109a) ◁59b 61▷

```
.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!

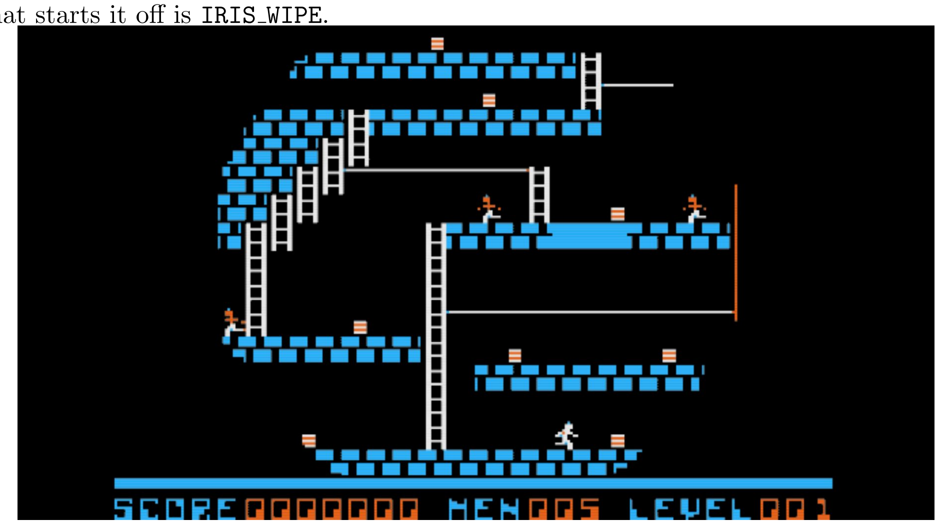
61 ⟨*level draw routine* 53⟩+≡ (109a) ◁60

```
.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 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 54a, DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, IRIS_WIPE 63, and PTR1 54b.

4.2 Iris Wipe

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



62 ⟨*defines* 3⟩+≡ (109b) ◁58b 64▷

```
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 63 and 74–76.
WIPE_MODE, used in chunks 63 and 106.

63 ⟨*iris wipe* 63⟩≡ (109a)

```
    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 61.

Uses IRIS_WIPE_STEP 67, PUT_STATUS_LEVEL 47, PUT_STATUS_LIVES 47, WIPE_COUNTER 62, and WIPE_MODE 62.

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.

64 ⟨*defines* 3⟩+≡ (109b) ◁62 66▷

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

Defines:
MATH_TMPH, used in chunks 65, 77, and 78a.
MATH_TMPL, used in chunks 65, 77, and 78a.

65 ⟨*routines* 4⟩+≡ (109b) ◁47 109a▷

```
      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 75 and 76.
Uses MATH_TMPH 64 and MATH_TMPL 64.

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

66 ⟨*defines* 3⟩+≡ (109b) ◁64 80d▷

```
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 74 and 78.
- `WIPE1`, used in chunks 74 and 77–79.
- `WIPE10D`, used in chunks 71, 72, 76b, and 79b.
- `WIPE10R`, used in chunks 71, 72, 76b, and 79b.
- `WIPE2`, used in chunks 68, 74d, 75a, 77, and 78a.
- `WIPE3H`, used in chunks 70, 75b, and 79a.
- `WIPE3L`, used in chunks 70, 75b, and 79a.
- `WIPE4H`, used in chunks 72, 75c, and 80a.
- `WIPE4L`, used in chunks 72, 75c, and 80a.
- `WIPE5H`, used in chunks 71, 75c, and 80b.
- `WIPE5L`, used in chunks 71, 75c, and 80b.
- `WIPE6H`, used in chunks 69b, 75d, and 79d.
- `WIPE6L`, used in chunks 69b, 75d, and 79d.
- `WIPE7D`, used in chunks 71, 72, 75e, and 79c.
- `WIPE7R`, used in chunks 71, 72, 75e, and 79c.
- `WIPE8D`, used in chunks 69b, 70, 76a, and 80c.
- `WIPE8R`, used in chunks 76a and 80c.
- `WIPE9D`, used in chunks 69b, 70, 76a, and 79f.
- `WIPE9R`, used in chunks 69b, 70, 76a, and 79f.

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

67 ⟨*iris wipe step* 67⟩≡ (109a) 68▷

```
    ORG     $88D7
IRIS_WIPE_STEP:
    SUBROUTINE

⟨WIPE0 = WIPE_COUNTER 74b⟩
⟨WIPE1 = 0 74c⟩
⟨WIPE2 = 2 * WIPE0 74d⟩
⟨WIPE2 = 3 - WIPE2 75a⟩

; 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 75b⟩
⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 75c⟩
⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 75d⟩

; 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 75e⟩
⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 76a⟩
⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 76b⟩
```

Defines:
IRIS_WIPE_STEP, used in chunk 63.

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.

68 ⟨*iris wipe step* 67⟩+≡ (109a) ◁67

```
.loop:

⟨iris wipe loop check 74a⟩

    JSR     DRAW_WIPE_STEP

    LDA     WIPE2+1
    BPL     .89a7

⟨WIPE2 += 4 * WIPE1 + 6 77⟩
    JMP     .8a14

.89a7:

⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 78a⟩
⟨Decrement WIPE0 78b⟩
⟨Increment WIPE3 79a⟩
⟨Decrement WIPE10 modulo 7 79b⟩
⟨Increment WIPE7 modulo 7 79c⟩
⟨Decrement WIPE6 79d⟩

.8a14:

⟨Increment WIPE1 79e⟩
⟨Increment WIPE9 modulo 7 79f⟩
⟨Decrement WIPE4 80a⟩
⟨Increment WIPE5 80b⟩
⟨Decrement WIPE8 modulo 7 80c⟩
    JMP     .loop
```

Uses `DRAW_WIPE_STEP` 69a and `WIPE2` 66.

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.

69a ⟨*draw wipe step* 69a⟩≡ (109a)

```
      ORG     $8A69
  DRAW_WIPE_STEP:
      SUBROUTINE

  ⟨Draw wipe for south part 69b⟩
  ⟨Draw wipe for north part 70⟩
  ⟨Draw wipe for north2 part 71⟩
  ⟨Draw wipe for south2 part 72⟩
```

Defines:
DRAW_WIPE_STEP, used in chunks 68 and 74a.

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

69b ⟨*Draw wipe for south part* 69b⟩≡ (69a)

```
      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 73a, ROW_TO_ADDR_FOR_BOTH_PAGES 27, WIPE6H 66, WIPE6L 66, WIPE8D 66, WIPE9D 66, and WIPE9R 66.

70 ⟨*Draw wipe for north part* 70⟩≡ (69a)

```
.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` 73a, `ROW_TO_ADDR_FOR_BOTH_PAGES` 27, `WIPE3H` 66, `WIPE3L` 66, `WIPE8D` 66, `WIPE9D` 66, and `WIPE9R` 66.

71 ⟨*Draw wipe for north2 part 71*⟩≡ (69a)

```
.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` 73a, `ROW_TO_ADDR_FOR_BOTH_PAGES` 27, `WIPE10D` 66, `WIPE10R` 66, `WIPE5H` 66, `WIPE5L` 66, `WIPE7D` 66, and `WIPE7R` 66.

72 ⟨*Draw wipe for south2 part* 72⟩≡ (69a)

```
.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 73a, ROW_TO_ADDR_FOR_BOTH_PAGES 27, WIPE10D 66, WIPE10R 66, WIPE4H 66, WIPE4L 66, WIPE7D 66, and WIPE7R 66.

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.

73a ⟨*draw wipe block* 73a⟩≡ (109a)

```
    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 69–72.
Uses ROW_ADDR 26b, ROW_ADDR2 26b, WIPE_BLOCK_CLOSE_MASK 73b, and WIPE_BLOCK_OPEN_MASK 73b.

73b ⟨*tables* 7⟩+≡ (109b) ◁57a 82e▷

```
    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 73a.
WIPE_BLOCK_OPEN_MASK, used in chunk 73a.

74a ⟨*iris wipe loop check* 74a⟩≡ (68)

```
      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` 69a, `WIPE0` 66, and `WIPE1` 66.

4.2.1 Initialization

74b ⟨`WIPE0 = WIPE_COUNTER` 74b⟩≡ (67)

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

Uses `WIPE0` 66 and `WIPE_COUNTER` 62.

74c ⟨`WIPE1 = 0` 74c⟩≡ (67)

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

Uses `WIPE1` 66.

74d ⟨`WIPE2 = 2 * WIPE0` 74d⟩≡ (67)

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

Uses `WIPE0` 66 and `WIPE2` 66.

75a ⟨WIPE2 = 3 - WIPE2 75a⟩≡ (67)
```
      LDA     #$03
      SEC
      SBC     WIPE2
      STA     WIPE2
      LDA     #$00
      SBC     WIPE2+1
      STA     WIPE2+1          ; WIPE2 = 3 - WIPE2
```
Uses WIPE2 66.

75b ⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 75b⟩≡ (67)
```
      LDA     WIPE_CENTER_Y
      SEC
      SBC     WIPE_COUNTER
      STA     WIPE3L
      LDA     #$00
      SBC     #$00
      STA     WIPE3H           ; WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER
```
Uses WIPE3H 66, WIPE3L 66, and WIPE_COUNTER 62.

75c ⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 75c⟩≡ (67)
```
      LDA     WIPE_CENTER_Y
      STA     WIPE4L
      STA     WIPE5L
      LDA     #$00
      STA     WIPE4H
      STA     WIPE5H           ; WIPE4 = WIPE5 = WIPE_CENTER_Y
```
Uses WIPE4H 66, WIPE4L 66, WIPE5H 66, and WIPE5L 66.

75d ⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 75d⟩≡ (67)
```
      LDA     WIPE_CENTER_Y
      CLC
      ADC     WIPE_COUNTER
      STA     WIPE6L
      LDA     #$00
      ADC     #$00
      STA     WIPE6H           ; WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER
```
Uses WIPE6H 66, WIPE6L 66, and WIPE_COUNTER 62.

75e ⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 75e⟩≡ (67)
```
      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 65, WIPE7D 66, WIPE7R 66, and WIPE_COUNTER 62.

76a ⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 76a⟩≡ (67)

```
      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 65, WIPE8D 66, WIPE8R 66, WIPE9D 66, and WIPE9R 66.

76b ⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 76b⟩≡ (67)

```
      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 65, WIPE10D 66, WIPE10R 66, and WIPE_COUNTER 62.

4.2.2 All that math stuff

77 ⟨WIPE2 += 4 * WIPE1 + 6 77⟩≡ (68)

```
      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 64, MATH_TMPL 64, WIPE1 66, and WIPE2 66.

78a ⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 78a⟩≡ (68)

```
      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 64, MATH_TMPL 64, WIPE0 66, WIPE1 66, and WIPE2 66.

78b ⟨*Decrement* WIPE0 78b⟩≡ (68)

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

Uses WIPE0 66.

79a ⟨*Increment* `WIPE3` 79a⟩≡ (68)
```
      INC     WIPE3L
      BNE     .89f2
      INC     WIPE3H          ; WIPE3++
  .89f2
```
Uses `WIPE3H` 66 and `WIPE3L` 66.

79b ⟨*Decrement* `WIPE10` *modulo 7* 79b⟩≡ (68)
```
      DEC     WIPE10R
      BPL     .89fc
      LDA     #$06
      STA     WIPE10R
      DEC     WIPE10D
  .89fc
```
Uses `WIPE10D` 66 and `WIPE10R` 66.

79c ⟨*Increment* `WIPE7` *modulo 7* 79c⟩≡ (68)
```
      INC     WIPE7R
      LDA     WIPE7R
      CMP     #$07
      BNE     .8a0a
      LDA     #$00
      STA     WIPE7R
      INC     WIPE7D
  .8a0a
```
Uses `WIPE7D` 66 and `WIPE7R` 66.

79d ⟨*Decrement* `WIPE6` 79d⟩≡ (68)
```
      DEC     WIPE6L
      LDA     WIPE6L
      CMP     #$FF
      BNE     .8a14
      DEC     WIPE6H
```
Uses `WIPE6H` 66 and `WIPE6L` 66.

79e ⟨*Increment* `WIPE1` 79e⟩≡ (68)
```
      INC     WIPE1
      BNE     .8a1a
      INC     WIPE1+1          ; WIPE1++
  .8a1a
```
Uses `WIPE1` 66.

79f ⟨*Increment* `WIPE9` *modulo 7* 79f⟩≡ (68)
```
      INC     WIPE9R
      LDA     WIPE9R
      CMP     #$07
      BNE     .8a28
      LDA     #$00
      STA     WIPE9R
      INC     WIPE9D
  .8a28
```
Uses `WIPE9D` 66 and `WIPE9R` 66.

80a ⟨*Decrement* WIPE4 80a⟩≡ (68)
```
      DEC     WIPE4L
      LDA     WIPE4L
      CMP     #$FF
      BNE     .8a32
      DEC     WIPE4H
  .8a32
```
Uses WIPE4H 66 and WIPE4L 66.

80b ⟨*Increment* WIPE5 80b⟩≡ (68)
```
      INC     WIPE5L
      BNE     .8a38
      INC     WIPE5H            ; WIPE5++
  .8a38
```
Uses WIPE5H 66 and WIPE5L 66.

80c ⟨*Decrement* WIPE8 *modulo 7* 80c⟩≡ (68)
```
      DEC     WIPE8R
      BPL     .8a42
      LDA     #$06
      STA     WIPE8R
      DEC     WIPE8D
  .8a42
```
Uses WIPE8D 66 and WIPE8R 66.

4.3 Level data

Now that we have the ability to draw a level from level data, we need a routine to get that level data. Recall that level data needs to be stored in pointers specified in the CURR_LEVEL_ROW_SPRITES_PTR_ tables.

4.3.1 Getting the compressed level data

The level data is stored in the game in compressed form, so we first grab the data for the level and put it into the COMPRESSED_LEVEL_DATA buffer.

There's one switch here, PREGAME_MODE, which dictates whether we're going to display the high-score screen, attract-mode game play, or an actual level for playing.

80d ⟨*defines* 3⟩+≡ (109b) ◁66 82d▷
```
  PREGAME_MODE    EQU     $A7
```
Defines:
PREGAME_MODE, used in chunks 81a, 92, 97, 98, 106, and 107.

81a ⟨*load compressed level data* 81a⟩≡

```
      ORG     $630E
  LOAD_COMPRESSED_LEVEL_DATA:
      SUBROUTINE
      ; Enter routine with A set to 1.

      STA     $bf74
      LDA     PREGAME_MODE
      LSR
      BEQ     .copy_level_data        ; If PREGAME_MODE was 0 or 1, copy level data

      LDA     $96
      LSR
      LSR
      LSR
      LSR
      CLC
      ADC     3
      STA     $b7ec       ; = 3 + 16 * $96
      LDA     $96
      AND     #$0F
      STA     $b7ed
      LDA     #$00
      STA     $b7f0
      LDA     #$0D
      STA     $b7f1
      LDA     #$00
      STA     $b7eb
      LDY     #$E8
      LDA     #$B7        ; AY = B7E8

      JSR     $23         ; JMP ($24)
      BCC     .end
      JMP     $6008

  .end:
      RTS

  .copy_level_data:
  ⟨Copy level data 81b⟩
```

Uses PREGAME_MODE 80d.

We're not really using ROW_ADDR here as a row address, just as a convenient place to store a pointer. Also, we can see that level data is stored in 256-byte pages at 9F00, A000, and so on. Level numbers start from 1, so 9E00 doesn't actually contain level data.

81b ⟨*Copy level data* 81b⟩≡ (81a)

```
      ⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 82a⟩
      ⟨Copy data from ROW_ADDR into COMPRESSED_LEVEL_DATA 82b⟩
```

82a ⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 82a⟩≡ (81b)

```
    LDA     LEVELNUM        ; 1-based
    CLC
    ADC     #$9E
    STA     ROW_ADDR+1
    LDY     #$00
    STY     ROW_ADDR        ; ROW_ADDR <- 9E00 + LEVELNUM * 0x100
```

Uses LEVELNUM 46 and ROW_ADDR 26b.

82b ⟨*Copy data from* ROW_ADDR *into* COMPRESSED_LEVEL_DATA 82b⟩≡ (81b)

```
.copyloop:
    LDA     (ROW_ADDR),Y
    STA     COMPRESSED_LEVEL_DATA,Y
    INY
    BNE     .copyloop
    RTS
```

Uses ROW_ADDR 26b.

4.3.2 Uncompressing and displaying the level

82c ⟨*load level* 82c⟩≡

```
    ORG     $6238
LOAD_LEVEL:
    SUBROUTINE
    ; Enter routine with X set to whether the level should be
    ; loaded verbatim or not.

    STX     VERBATIM

    ⟨Initialize level counts 83⟩

    LDA     1
    STA     ALIVE
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    ⟨uncompress level data 84a⟩
```

Defines:
LOAD_LEVEL, used in chunks 86 and 107.
Uses VERBATIM 55b.

82d ⟨*defines* 3⟩+≡ (109b) ◁80d 87a▷

```
TMP                     EQU     $1A
LEVEL_DATA_INDEX        EQU     $92
```

82e ⟨*tables* 7⟩+≡ (109b) ◁73b 89a▷

```
    ORG     $0C98
TABLE_0C98      DS      6
    ORG     $0CE0
TABLE_0CE0      DS      31
```

Here we are initializing variables in preparation for loading the level data. Since drawing the level will keep track of ladder, gold, and guard count, we need to zero them out. There are also some areas of memory whose purpose is not yet known, and these are zeroed out also.

83 ⟨*Initialize level counts* 83⟩≡ (82c)

```
    LDX     #$FF
    STX     PLAYER_COL
    INX
    STX     LADDER_COUNT
    STX     GOLD_COUNT
    STX     GUARD_COUNT
    STX     $19
    STX     $A0
    STX     LEVEL_DATA_INDEX
    STX     TMP
    STX     GAME_ROWNUM
    TXA

    LDX     30
.loop1
    STA     TABLE_0CE0,X
    DEX
    BPL     .loop1

    LDX     5
.loop2
    STA     TABLE_0C98,X
    DEX
    BPL     .loop2
```

Uses GAME_ROWNUM 32a, GOLD_COUNT 55a, GUARD_COUNT 55a, LADDER_COUNT 55a, and PLAYER_COL 55a.

The level data is stored in "compressed" form, just 4 bits per sprite since we don't use any higher ones to define a level. For each of the 16 game rows, we load up the compressed row data and break it apart, one 4-bit sprite per column.

Once we've done that, we draw the level using DRAW_LEVEL_PAGE2. That routine returns an error if there was no player sprite in the level. If there was no error, we simply return. Otherwise we have to handle the error condition, since there's no point in playing without a player!

84a ⟨*uncompress level data* 84a⟩≡ (82c)

```
.row_loop:
⟨get row destination pointer for uncompressing level data 84b⟩
⟨uncompress row data 85a⟩
⟨next compressed row for row_loop 85b⟩

    JSR     DRAW_LEVEL_PAGE2
    BCC     .end                ; No error

⟨handle no player sprite in level 86⟩

.end:
    RTS

.62c4:
    JMP     $6008       ; play? complain? fall over?
```

Uses DRAW_LEVEL_PAGE2 53.

Each row will have their sprite data stored at locations specified by the CURR_LEVEL_ROW_SPRITES_PTR_ tables.

84b ⟨*get row destination pointer for uncompressing level data* 84b⟩≡ (84a)

```
    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 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 54a, PTR1 54b, and PTR2 54b.

To uncompress the data for a row, we use the counter in TMP as an odd/even switch so that we know which 4-bit chunk (nibble) in a byte we want. Even numbers are for the low nibble while odd numbers are for the high nibble.

In addition, if we encounter any sprite number 10 or above then we replace it with sprite 0 (all black).

85a ⟨*uncompress row data* 85a⟩≡ (84a)

```
    LDA     0
    STA     GAME_COLNUM

.col_loop:
    LDA     TMP                         ; odd/even counter
    LSR
    LDY     LEVEL_DATA_INDEX
    LDA     COMPRESSED_LEVEL_DATA,Y
    BCS     .628c                       ; odd?
    AND     #$0F
    BPL     .6292                       ; unconditional jump
.628c

    LSR
    LSR
    LSR
    LSR
    INC     LEVEL_DATA_INDEX

.6292
    INC     TMP

    LDY     GAME_COLNUM
    CMP     10
    BCC     .629c
    LDA     0                           ; sprite >= 10 -> sprite 0
.629c:

    STA     (PTR1),Y
    STA     (PTR2),Y

    INC     GAME_COLNUM
    LDA     GAME_COLNUM
    CMP     28
    BCC     .col_loop                   ; loop while GAME_COLNUM < 28
```

Uses GAME_COLNUM 32a, PTR1 54b, and PTR2 54b.

85b ⟨*next compressed row for* row_loop 85b⟩≡ (84a)

```
    INC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    CPY     16
    BCC     .row_loop                   ; loop while GAME_ROWNUM < 16
```

Uses GAME_ROWNUM 32a.

When there's no player sprite in the level, a few things can happen. Firstly, if `\$96` is zero, we're going to jump to `\$6008`. Otherwise, we set `\$96` to zero, increment `\$97`, set X to `0xFF`, and retry `LOAD_LEVEL` from the very beginning.

86 ⟨*handle no player sprite in level* 86⟩≡ (84a)

```
    LDA     $96
    BEQ     .62c4

    LDX     0
    STX     $96
    INC     $97
    DEX
    JMP     LOAD_LEVEL
```

Uses `LOAD_LEVEL` 82c.

Chapter 5

High scores

For this routine, we have two indexes. The first is stored in `\$55` and is the high score number, from 1 to 10. The second is stored in `\$56` and keeps our place in the actual high score data table stored at `\$1F00`.

There are ten slots in the high score table, each with eight bytes. The first three bytes are for the player initials, the fourth byte is the level – or zero if the row should be empty – and the last four bytes are the BCD-encoded score, most significant byte first.

87a ⟨*defines* 3⟩+≡ (109b) ◁82d 90c▷

```
HI_SCORE_DATA    EQU     $1F00
```

Defines:
`HI_SCORE_DATA`, used in chunks 89 and 90a.

87b ⟨*construct and display high score screen* 87b⟩≡ (109a)

```
        ORG     $786B

HI_SCORE_SCREEN:
        SUBROUTINE

        JSR     CLEAR_HGR2
        LDA     #$40
        STA     DRAW_PAGE
        LDA     #$00
        STA     GAME_COLNUM
        STA     GAME_ROWNUM

        ⟨draw high score table header 88a⟩
        ⟨draw high score rows 88b⟩
        ⟨show high score page 91⟩
```

Defines:
`HI_SCORE_SCREEN`, used in chunk 98d.
Uses `CLEAR_HGR2` 4, `DRAW_PAGE` 39, `GAME_COLNUM` 32a, and `GAME_ROWNUM` 32a.

88a ⟨*draw high score table header* 88a⟩≡ (87b)

```
      ; "    LODE RUNNER HIGH SCORES\r"
      ; "\r"
      ; "\r"
      ; "    INITIALS LEVEL  SCORE\r"
      ; "    -------- ----- --------\r"
      JSR     PUT_STRING
      HEX     A0 A0 A0 A0 CC CF C4 C5 A0 D2 D5 CE CE C5 D2 A0
      HEX     C8 C9 C7 C8 A0 D3 C3 CF D2 C5 D3 8D 8D 8D A0 A0
      HEX     A0 A0 C9 CE C9 D4 C9 C1 CC D3 A0 CC C5 D6 C5 CC
      HEX     A0 A0 D3 C3 CF D2 C5 8D A0 A0 A0 A0 AD AD AD AD
      HEX     AD AD AD AD A0 AD AD AD AD AD A0 AD AD AD AD AD
      HEX     AD AD AD 8D 00
```

Uses PUT_STRING 41 and SCORE 44b.

88b ⟨*draw high score rows* 88b⟩≡ (87b)

```
      LDA     #$01
      STA     $55             ; Used for row number
  .loop:
      ⟨draw high score row number 88c⟩
      ⟨draw high score initials 89b⟩
      ⟨draw high score level 89c⟩
      ⟨draw high score 90a⟩
      ⟨next high score row 90b⟩
```

88c ⟨*draw high score row number* 88c⟩≡ (88b)

```
      CMP     #$0A
      BNE     .display_0_to_9
      LDA     #1
      JSR     PUT_DIGIT
      LDA     #0
      JSR     PUT_DIGIT
      JMP     .rest_of_row_number

  .display_0_to_9:
      LDA     #$A0
      JSR     PUT_CHAR        ; space
      LDA     $55
      JSR     PUT_DIGIT

  .rest_of_row_number:
      ; ".    "
      JSR     PUT_STRING
      HEX     AE A0 A0 A0 A0 00
```

Uses PUT_CHAR 40a, PUT_DIGIT 42a, and PUT_STRING 41.

89a ⟨*tables* 7⟩+≡ (109b) ◁82e 99b▷

```
      ORG     $79A2
  HI_SCORE_TABLE_OFFSETS:
      HEX     00 08 10 18 20 28 30 38 40 48
```

Defines:
HI_SCORE_TABLE_OFFSETS, used in chunk 89b.

89b ⟨*draw high score initials* 89b⟩≡ (88b)

```
      LDX     $55
      LDY     HI_SCORE_TABLE_OFFSETS,X
      STY     $56
      LDA     HI_SCORE_DATA+3,Y
      BNE     .draw_initials
      JMP     .next_high_score_row
  .draw_initials:
      LDY     $56
      LDA     HI_SCORE_DATA,Y
      JSR     PUT_CHAR
      LDY     $56
      LDA     HI_SCORE_DATA+1,Y
      JSR     PUT_CHAR
      LDY     $56
      LDA     HI_SCORE_DATA+2,Y
      JSR     PUT_CHAR

      ; "    "
      JSR     PUT_STRING
      HEX     A0 A0 A0 A0 00
```

Uses HI_SCORE_DATA 87a, HI_SCORE_TABLE_OFFSETS 89a, PUT_CHAR 40a, and PUT_STRING 41.

89c ⟨*draw high score level* 89c⟩≡ (88b)

```
      LDY     $56
      LDA     HI_SCORE_DATA+3,Y
      JSR     TO_DECIMAL3
      LDA     HUNDREDS
      JSR     PUT_DIGIT
      LDA     TENS
      JSR     PUT_DIGIT
      LDA     UNITS
      JSR     PUT_DIGIT

      ; "  "
      JSR     PUT_STRING
      HEX     A0 A0 00
```

Uses HI_SCORE_DATA 87a, HUNDREDS 42b, PUT_DIGIT 42a, PUT_STRING 41, TENS 42b, TO_DECIMAL3 43, and UNITS 42b.

90a ⟨*draw high score* 90a⟩≡ (88b)
```
      LDY     $56
      LDA     HI_SCORE_DATA+4,Y
      JSR     BCD_TO_DECIMAL2
      LDA     TENS
      JSR     PUT_DIGIT
      LDA     UNITS
      JSR     PUT_DIGIT

      LDY     $56
      LDA     HI_SCORE_DATA+5,Y
      JSR     BCD_TO_DECIMAL2
      LDA     TENS
      JSR     PUT_DIGIT
      LDA     UNITS
      JSR     PUT_DIGIT

      LDY     $56
      LDA     HI_SCORE_DATA+6,Y
      JSR     BCD_TO_DECIMAL2
      LDA     TENS
      JSR     PUT_DIGIT
      LDA     UNITS
      JSR     PUT_DIGIT

      LDY     $56
      LDA     HI_SCORE_DATA+7,Y
      JSR     BCD_TO_DECIMAL2
      LDA     TENS
      JSR     PUT_DIGIT
      LDA     UNITS
      JSR     PUT_DIGIT
```
Uses `BCD_TO_DECIMAL2` 44a, `HI_SCORE_DATA` 87a, `PUT_DIGIT` 42a, `TENS` 42b, and `UNITS` 42b.

90b ⟨*next high score row* 90b⟩≡ (88b)
```
  .next_high_score_row:
      JSR     NEWLINE
      INC     $55
      LDA     $55
      CMP     #11
      BCS     .end
      JMP     .loop
```
Uses `NEWLINE` 40a.

90c ⟨*defines* 3⟩+≡ (109b) ◁87a 94b▷
```
  TXTPAGE2                EQU     $C055
```
Defines:
`TXTPAGE2`, used in chunk 91.

91 ⟨*show high score page* 91⟩≡ (87b)

```
  .end:
      STA     TXTPAGE2        ; Flip to page 2
      LDA     #$20
      STA     DRAW_PAGE       ; Set draw page to 1
      RTS
```

Uses `DRAW_PAGE` 39 and `TXTPAGE2` 90c.

Chapter 6

Game play

6.1 Splash screen

92 ⟨*splash screen* 92⟩≡ (109a)

```
    ORG     $6008
.main:
    JSR     CLEAR_HGR1

    LDA     #$FF
    STA     .rd_table+1
    LDA     #$0E
    STA     .rd_table+2     ; RD_TABLE = 0x0EFF
    LDY     0
    STY     GAME_ROWNUM
    STY     PREGAME_MODE
    STY     $96             ; GAME_ROWNUM = $96 = PREGAME_MODE = 0
    LDA     #$20
    STA     HGR_PAGE
    STA     DRAW_PAGE       ; HGR_PAGE = DRAW_PAGE = 0x20

    ⟨splash screen loop 93⟩

    STA     TXTPAGE1
    STA     HIRES
    STA     MIXCLR
    STA     TXTCLR
    JMP     .618E
```

Uses `CLEAR_HGR1` 4, `DRAW_PAGE` 39, `GAME_ROWNUM` 32a, `HGR_PAGE` 26b, `HIRES` 94b, `MIXCLR` 94b, `PREGAME_MODE` 80d, `TXTCLR` 94b, and `TXTPAGE1` 94b.

This loop writes a screen of graphics by reading from the table starting at `\$0F00`. The table is in pairs of bytes, where the first byte is the byte offset from the beginning of the row, and the second byte is the byte to write. However, if the first byte is `0x00` then we end that row.

As in other cases, the pointer into the table is stored in the `LDA` instruction that reads from the table.

The code takes advantage of the fact that all bytes written to the page have their high bit set, while offsets from the beginning of the row are always less than `0x80`. Thus, if we read a byte and it is `0x00`, we end the loop. Otherwise, if the byte is less than `0x80` we set that as the offset. Otherwise, the byte has its high bit set, and we write that byte to the graphics page.

93 ⟨*splash screen loop* 93⟩≡ (92)

```
.draw_splash_screen_row:
    JSR     ROW_TO_ADDR     ; ROW_ADDR = ROW_TO_ADDR(Y)
    LDY     #0

.loop:
    INC     .rd_table+1
    BNE     .rd_table
    INC     .rd_table+2     ; RD_TABLE++

.rd_table:
    LDA     $1A84           ; A <- *RD_TABLE ($1A84 is just a dummy value)
    BEQ     .end_of_row     ; if A == 0: break
    BPL     .is_row_offset  ; if A > 0: A -> Y, .loop
    STA     (ROW_ADDR),Y    ; *(ROW_ADDR+Y) = A

    INY                     ; Y++
    BPL     .loop           ; While Y < 0x80 (really while not 00)

.is_row_offset:
    TAY
    BPL     .loop           ; Unconditional jump

.end_of_row:
    INC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    CPY     #192
    BCC     .draw_splash_screen_row
```

Uses `GAME_ROWNUM` 32a, `ROW_ADDR` 26b, and `ROW_TO_ADDR` 26c.

6.2 Startup code

The startup code is run immediately after relocating memory blocks.

94a ⟨*startup code* 94a⟩≡ (109a)
  ⟨*set startup softswitches* 94c⟩
  ⟨*set stack size* 94d⟩
  ⟨*maybe set carry but not really* 95a⟩
  ⟨*ready yourself* 95c⟩

The first address, `ROMIN_RDROM_WRRAM2` is a bank-select switch. By reading it twice, we set up the memory area from `\$D000-\$DFFF` to read from the ROM, but write to RAM bank 2.

The next four softswiches set up the display for full-screen hi-res graphics, page 1.

94b ⟨*defines* 3⟩+≡ (109b) ◁90c 95b▷

```
ROMIN_RDROM_WRRAM2      EQU     $C081
TXTCLR                  EQU     $C050
MIXCLR                  EQU     $C052
TXTPAGE1                EQU     $C054
HIRES                   EQU     $C057
```

Defines:
  `HIRES`, used in chunks 92 and 94c.
  `MIXCLR`, used in chunks 92 and 94c.
  `ROMIN_RDROM_WRRAM2`, used in chunk 94c.
  `TXTCLR`, used in chunks 92 and 94c.
  `TXTPAGE1`, used in chunks 92, 94c, and 106.

94c ⟨*set startup softswitches* 94c⟩≡ (94a)

```
    ORG     $5F7D

    LDA     ROMIN_RDROM_WRRAM2
    LDA     ROMIN_RDROM_WRRAM2
    LDA     TXTCLR
    LDA     MIXCLR
    LDA     TXTPAGE1
    LDA     HIRES
```

Uses `HIRES` 94b, `MIXCLR` 94b, `ROMIN_RDROM_WRRAM2` 94b, `TXTCLR` 94b, and `TXTPAGE1` 94b.

The 6502 stack, at maximum, runs from `\$0100-\$01FF`. The stack starts at `\$0100` plus the stack index (the S register), and grows towards `\$0100`. Here we are setting the S register to `0x07` which makes for a very small stack – 8 bytes.

94d ⟨*set stack size* 94d⟩≡ (94a)

```
    LDX     #$07
    TXS
```

This next part seems to set the carry only if certain bits in location \$5F94 are set. I can find no writes to this location, so the effect is that the carry is cleared. It's entirely possible that this was altered by the cracker.

95a ⟨*maybe set carry but not really* 95a⟩≡ (94a)

```
    CLC
    LDA     #$01
    AND     #$A4
    BEQ     .short_delay_mode
    SEC
    ; fall through to short delay mode
```

This next part sets the delay for this game mode, and also reads the keyboard strobe softswtich. That just clears the keyboard strobe in readiness to see if a key is pressed. Then we get dumped into the main loop.

95b ⟨*defines* 3⟩+≡ (109b) ◁94b 95d▷

```
KBDSTRB     EQU     $C010
```

Defines:
KBDSTRB, used in chunks 95c and 97b.

95c ⟨*ready yourself* 95c⟩≡ (94a)

```
    ORG     $5F9A

.short_delay_mode:
    LDX     #$22            ; Number of times to check for keyboard press (34).
    LDY     #$02            ; Number of times to do X checks (2).
                            ; GAME_ROWNUM was initialized to 1, so we do 34*2*1 checks.
    LDA     KBDSTRB
    LDA     #$CA            ; Fake keypress 0x4A (J)
    JMP     .check_for_button_down
```

Uses GAME_ROWNUM 32a and KBDSTRB 95b.

Checking for a joystick button (or equivalently the open apple and solid apple keys) to be pressed involves checking the high bit after reading the corresponding button softswitch. Here we're checking if any of the buttons are pressed.

95d ⟨*defines* 3⟩+≡ (109b) ◁95b 96b▷

```
BUTN0       EQU     $C061       ; Or open apple
BUTN1       EQU     $C062       ; Or solid apple
STORED_KEY  EQU     $95
    ORG     $95
    HEX     CA
```

Defines:
BUTN0, used in chunk 96a.
BUTN1, used in chunk 96a.
STORED_KEY, used in chunk 96a.

96a ⟨*check for button down* 96a⟩≡

```
      ORG     $6199

  .check_stored_key:
      LDA     STORED_KEY

  .check_for_button_down:
      CMP     #$CB                ; Key pressed is 0x4B (K)?
      BEQ     .no_button_pressed  ; Skip check button presses.
      LDA     BUTN1
      BMI     .button_pressed
      LDA     BUTN0
      BMI     .button_pressed

      ; fall through to .no_button_pressed
```

Uses BUTN0 95d, BUTN1 95d, and STORED_KEY 95d.

Here we read the keyboard, which involves checking the high bit of the KBD softswitch. This also loads the ASCII code for the key. We check for a keypress in a loop based on the X and Y registers, and on GAME_ROWNUM! So we check for X x Y x GAME_ROWNUM iterations. This controls alternation between "attract-mode" gameplay and the high score screen.

96b ⟨*defines* 3⟩+≡ (109b) ◁95d

```
  KBD         EQU     $C000
```

Defines:
KBD, used in chunk 96c.

96c ⟨*no button pressed* 96c⟩≡

```
      ORG     $61A9

  .no_button_pressed:
      LDA     KBD
      BMI     .key_pressed
      DEX
      BNE     .check_stored_key
      DEY
      BNE     .check_stored_key
      DEC     GAME_ROWNUM
      BNE     .check_stored_key

      ; fall through to .no_button_or_key_timeout
```

Uses GAME_ROWNUM 32a and KBD 96b.

If one of the joystick buttons was pressed:

97a ⟨*button pressed at startup* 97a⟩≡
```
      ORG     $6201

  .button_pressed:
      LDX     #$00
      STX     $96
      INX
      STX     LEVELNUM
      STX     $9D
      LDA     #$02
      STX     PREGAME_MODE
      JMP     .play_game
```
Uses `LEVELNUM` 46 and `PREGAME_MODE` 80d.

And if one of the keys was pressed:

97b ⟨*key pressed at startup* 97b⟩≡
```
      ORG     $61F6

  .key_pressed:
      STA     KBDSTRB     ; Clear keyboard strobe
      CMP     #$85        ; if ctrl-E:
      BEQ     .ctrl_e_pressed
      CMP     #$8D        ; if return key:
      BEQ     .return_pressed

      ; fall through to .button_pressed
```
Uses `KBDSTRB` 95b.

Two keys are special, ctrl-E, which opens the level editor, and return, which starts a new game (?).

97c ⟨*ctrl-e pressed* 97c⟩≡
```
      ORG     $6211

  .ctrl_e_pressed:
      JMP     .start_level_editor
```

97d ⟨*return pressed* 97d⟩≡
```
      ORG     $61E4

  .return_pressed:
      LDA     #$01
      JSR     $6359
```

Finally, if no key or button was pressed and we've reached the maximum number of polls through the loop:

98a ⟨*timed out waiting for button or keypress* 98a⟩≡

```
    ORG     $61B8

.no_button_or_key_timeout:
    LDA     PREGAME_MODE
    BNE     .check_game_mode    ; If PREGAME_MODE != 0, .check_game_mode.

    ; When PREGAME_MODE = 0:
    LDX     #$01
    STX     PREGAME_MODE        ; Set PREGAME_MODE = 1
    STX     LEVELNUM
    STX     $AC
    STX     $9D                 ; LEVELNUM = $AC = $9D = 1
    LDX     $99
    STX     .restore_99+1       ; Save previous value of $99
    STA     $99                 ; $99 = 0
    JMP     .init_game_data
```

Uses `LEVELNUM` 46 and `PREGAME_MODE` 80d.

98b ⟨*check game mode* 98b⟩≡

```
    ORG     $61DE

.check_game_mode:
    CMP     #$01
    BNE     .game_mode_not_1
    BEQ     .display_high_score_screen        ; Unconditional jump
```

98c ⟨*game mode not 1* 98c⟩≡

```
    ORG     $61F3

.game_mode_not_1:
    JMP     $6008
```

98d ⟨*display high score screen* 98d⟩≡

```
    ORG     $61E9

.display_high_score_screen:
    JSR     HI_SCORE_SCREEN
    LDA     #$02
    STA     PREGAME_MODE            ; PREGAME_MODE = 2
    JMP     .long_delay_attact_mode
```

Uses `HI_SCORE_SCREEN` 87b and `PREGAME_MODE` 80d.

When we change over to attract mode, we set the delay to the next mode very large: 195075 times around the loop.

99a ⟨*long delay attract mode* 99a⟩≡
```
    ORG     $618E

.long_delay_attact_mode:
    JSR     $869f
    LDX     #$FF
    LDY     #$FF
    LDA     #$03
    STA     GAME_ROWNUM

    ; fall through to .check_stored_key
```
Uses GAME_ROWNUM 32a.

6.3 Moving the player

The player's sprite position is stored in PLAYER_COL and PLAYER_ROW, while the offset from the exact sprive location is stored in PLAYER_X_ADJ and PLAYER_Y_ADJ. These adjustments are offset by 2, so that 2 means zero offset. The player also has a PLAYER_ANIM_STATE which is an index into the SPRITE_ANIM_SEQS table. The GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER gets the sprite corresponding to the player's animation state and the player's adjusted screen coordinate.

99b ⟨*tables* 7⟩+≡ (109b) ◃89a 108b▹
```
    ORG     $6968
SPRITE_ANIM_SEQS:
    HEX     0B 0C 0D        ; player running left
    HEX     18 19 1A        ; player monkey swinging left
    HEX     0F              ; player digging left
    HEX     13              ; player falling, facing left
    HEX     09 10 11        ; player running right
    HEX     15 16 17        ; player monkey swinging right
    HEX     25              ; player digging right
    HEX     14              ; player falling, facing right
    HEX     0E 12           ; player climbing on ladder
```
Defines:
SPRITE_ANIM_SEQS, used in chunks 58 and 100a.

100a ⟨*get player sprite data* 100a⟩≡

```
      ORG     $6B85
  GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER:
      SUBROUTINE
      ; Using PLAYER_COL/ROW, PLAYER_X/Y_ADJ, and PLAYER_ANIM_STATE,
      ; return the player sprite in A, and the screen coords in X and Y.

      LDX     PLAYER_COL
      LDY     PLAYER_X_ADJ
      JSR     GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR
      STX     SPRITE_NUM          ; Used only as a temporary to save X
      LDY     PLAYER_ROW
      LDX     PLAYER_Y_ADJ
      JSR     GET_SCREEN_ROW_OFFSET_IN_X_FOR
      LDX     PLAYER_ANIM_STATE
      LDA     SPRITE_ANIM_SEQS,X
      LDX     SPRITE_NUM
      RTS
```

Defines:
GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER, used in chunk 103.
Uses GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR 31c, GET_SCREEN_ROW_OFFSET_IN_X_FOR 31a, PLAYER_ANIM_STATE 58b, PLAYER_COL 55a, PLAYER_ROW 55a, PLAYER_X_ADJ 58b, PLAYER_Y_ADJ 58b, SPRITE_ANIM_SEQS 99b, and SPRITE_NUM 23c.

Since PLAYER_ANIM_STATE needs to play a sequence over and over, there is a routine to increment the animation state and wrap if necessary. It works by loading A with the lower bound, and X with the upper bound.

100b ⟨*increment player animation state* 100b⟩≡

```
      ORG     $6BF4
  INC_ANIM_STATE:
      SUBROUTINE

      INC     PLAYER_ANIM_STATE
      CMP     PLAYER_ANIM_STATE
      BCC     .check_upper_bound      ; lower bound < PLAYER_ANIM_STATE?
      ; otherwise PLAYER_ANIM_STATE <= lower bound:

  .write_lower_bound:
      STA     PLAYER_ANIM_STATE       ; PLAYER_ANIM_STATE = lower bound
      RTS

  .check_upper_bound:
      CPX     PLAYER_ANIM_STATE
      BCC     .write_lower_bound       ; PLAYER_ANIM_STATE > upper bound?
      ; otherwise PLAYER_ANIM_STATE <= upper bound:
      RTS
```

Defines:
INC_ANIM_STATE, never used.
Uses PLAYER_ANIM_STATE 58b.

This routine checks whether the player picks up gold. First we check to see if the player's location is exactly on a sprite coordinate, and return if not. Otherwise, we check the phantom sprite data to see if there's gold at the player's location, and return if not. So if there is gold, we decrement the gold count, put a blank sprite in the phantom sprite data, increment the score by 250, place a gold sprite on the screen at the player location, and then load up data into the sound area.

101 ⟨*check for gold picked up by player* 101⟩≡

```
    ORG     $6B9D
CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER:
    SUBROUTINE

    LDA     PLAYER_X_ADJ
    CMP     #$02
    BNE     .end
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BNE     .end

    LDY     PLAYER_ROW
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1                                  ; PTR2 <- CURR_LEVEL_ROW_SPRITES_PTR2_ + PLAYER_

    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$07                ; Gold
    BNE     .end

    LSR     $94
    DEC     GOLD_COUNT          ; GOLD_COUNT--

    LDY     PLAYER_ROW
    STY     GAME_ROWNUM
    LDY     PLAYER_COL
    STY     GAME_COLNUM
    LDA     #$00
    STA     (PTR2),Y
    JSR     DRAW_SPRITE_PAGE2   ; Register and draw blank at player loc in PTR2

    LDY     PLAYER_ROW
    LDX     PLAYER_COL
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #$07                            ; Gold
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS     ; Draw gold at player loc

    LDY     #$02
    LDA     #$50
```

```
    JSR     ADD_AND_UPDATE_SCORE            ; SCORE += 250
    JSR     LOAD_SOUND_DATA
    HEX     07 45 06 55 05 44 04 54 03 43 02 53 00

.end:
    RTS
```

Uses ADD_AND_UPDATE_SCORE 45, CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 54a, DRAW_SPRITE_AT_PIXEL_COORDS 36, DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_SCREEN_COORDS_FOR 29a, GOLD_COUNT 55a, LOAD_SOUND_DATA 49, PLAYER_COL 55a, PLAYER_ROW 55a, PLAYER_X_ADJ 58b, PLAYER_Y_ADJ 58b, PTR2 54b, and SCORE 44b.

103 ⟨*move player* 103⟩≡

```
    ORG     $64BD
MOVE_PLAYER:
    SUBROUTINE

    LDA     #$01
    STA     $94                     ; $94 = 1
    LDA     $9C
    BEQ     .data_9C_zero           ; If $9C == 0
    BPL     .data_9C_positive_      ; If $9C < 0x80
    JMP     $67E7                   ; Otherwise (if $9C >= 0x80)

.data_9C_positive_:
    JMP     data_9C_positive

.data_9C_zero:
    LDY     PLAYER_ROW
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1                                  ; PTR2 <- CURR_LEVEL_ROW_SPRITES_PTR2_ + PLAYER_

    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$03
    BEQ     .sprite_is_ladder_                      ; ladder at phantom location?
    CMP     #$04
    BEQ     .sprite_is_pole                         ; pole at phantom location?
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BEQ     .sprite_is_ladder_                      ; player at exact sprite row?

    ; player is not on exact sprite row, fallthrough.

.sprite_is_pole:
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BCC     .player_moving_up                       ; player to the left of sprite row?
    LDY     PLAYER_ROW
    CPY     #$0F
    BEQ     .sprite_is_ladder_                      ; player exactly sprite row 15?

    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR1
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
    STA     PTR1+1                                  ; PTR1 = CURR_LEVEL_ROW_SPRITES_PTR_ + Y
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
    STA     PTR2+1                                  ; PTR2 = CURR_LEVEL_ROW_SPRITES_PTR2_ + Y
```

```
    LDY     PLAYER_COL
    LDA     (PTR1),Y
    CMP     #$00                    ; Empty
    BEQ     .player_moving_up
    CMP     #$08                    ; Guard
    BEQ     .sprite_is_ladder_
    LDA     (PTR2),Y
    CMP     #$01                    ; Brick
    BEQ     .sprite_is_ladder_
    CMP     #$02                    ; Stone
    BEQ     .sprite_is_ladder_
    CMP     #$03                    ; Ladder
    BNE     .player_moving_up

.sprite_is_ladder_:
    JMP     .sprite_is_ladder

.player_moving_up:
    LDA     #$00
    STA     $9B                     ; $9B = 0
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    ; A = sprite number
    ; X = half screen col
    ; Y = screen row
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS

    LDA     #$07                    ; Next anim state: player falling, facing left
    LDX     PLAYER_FACING_DIRECTION
    BMI     .player_facing_left
    LDA     #$0F                    ; Next anim state: player falling, facing right
.player_facing_left:
    STA     PLAYER_ANIM_STATE

    JSR     $6C13

    INC     PLAYER_Y_ADJ            ; Go down faster
    LDA     PLAYER_Y_ADJ
    CMP     #$05
    BCS     .down_too_fast          ; PLAYER_Y_ADJ >= 5
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER
    JMP     $6C02           ; tailcall

.down_too_fast:
    LDA     #$00
    STA     PLAYER_Y_ADJ            ; Wrap around to move up???

    LDY     PLAYER_ROW
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR1
    STA     PTR2
```

```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
    STA     PTR1+1                              ; PTR1 = CURR_LEVEL_ROW_SPRITES_PTR_ + PLAYER_RO
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
    STA     PTR2+1                              ; PTR2 = CURR_LEVEL_ROW_SPRITES_PTR2_ + PLAYER_R

    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01            ; Brick
    BNE     .move_down
    LDA     #$00            ; Store empty sprite

.move_down:
    STA     (PTR1),Y
    INC     PLAYER_ROW      ; Move down

    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
    STA     PTR1+1                              ; PTR1 = CURR_LEVEL_ROW_SPRITES_PTR_ + PLAYER_RO
    LDY     PLAYER_COL
    LDA     #$09        ; player facing right
    STA     (PTR1),Y
    JMP     $6C02       ; tailcall

.sprite_is_ladder:
    LDA     $9B
    BNE     .658f
    LDA     #$64
    LDX     #$08
    JSR     PLAY_NOTE

.658f:
    LDA     #$20
    STA     $A4
    STA     $9B
    JSR     $6A12
    LDA     $9E
    CMP     #$C9
    BNE     .65a4
    JSR     $66BD
    BCS     .65c2
    RTS
```

Defines:
MOVE_PLAYER, used in chunk 108a.
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 54a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 54a, DRAW_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 100a, PLAY_NOTE 50b, PLAYER_ANIM_STATE 58b, PLAYER_COL 55a, PLAYER_ROW 55a, PLAYER_Y_ADJ 58b, PTR1 54b, and PTR2 54b.

6.4 Initialization

106 ⟨*Initialize game data* 106⟩≡

```
    ORG     $6056

.init_game_data:
    LDA     0
    STA     SCORE
    STA     SCORE+1
    STA     SCORE+2
    STA     SCORE+3
    STA     $97
    STA     WIPE_MODE       ; WIPE_MODE = SCORE = $97 = 0
    STA     $53
    STA     $AB
    STA     $A8             ; $53 = $AB = $A8 = 0
    LDA     #$9b    ; 155
    STA     $A9             ; $A9 = 155
    LDA     5
    STA     LIVES           ; LIVES = 5
    LDA     PREGAME_MODE
    LSR
    ; if PREGAME_MODE was 0 or 1 (i.e. not displaying high score screen),
    ; play the game.
    BEQ     .put_status_and_start_game

    ; We were displaying the high score screen
    LDA     1
    JSR     $6359
    CMP     #$00
    BNE     .6086
    JSR     $8106
    JMP     $6008

.6086:
    LDA     $1FFF
    BNE     .6091
    LDA     $36
    LDX     $37
    BNE     .6095

.6091:
    LDA     $38
    LDX     $39

.6095:
    STA     JMP_ADDR
    STX     JMP_ADDR+1

.put_status_and_start_game:
```

```
        JSR     PUT_STATUS
        STA     TXTPAGE1
```

Uses LIVES 46, PREGAME_MODE 80d, SCORE 44b, TXTPAGE1 94b, and WIPE_MODE 62.

107 ⟨*start game* 107⟩≡

```
        ORG     $609F

    .start_game:
        LDX     #$01
        JSR     LOAD_LEVEL
        LDA     #$00
        STA     $9E
        STA     $9F
        LDA     PREGAME_MODE
        LSR
        ; if PREGAME_MODE was 0 or 1 (i.e. not displaying high score screen),
        ; play the game.
        BEQ     .play_game

        ; When PREGAME_MODE is 2:
        JSR     $869F
        LDA     PLAYER_COL
        STA     GAME_COLNUM
        LDA     PLAYER_ROW
        STA     GAME_ROWNUM
        LDA     #$09
        JSR     $8700

    .play_game:
        LDX     #$00
        STX     $9C
        STX     NOTE_INDEX

        LDA     $97
        CLC
        ADC     GUARD_COUNT         ; GUARD_COUNT + $97 can't be greater than 8.
        TAY
        LDX     TIMES_3_TABLE,Y     ; X = 3 * Y
        LDA     $6CA7,X
        STA     $60
        LDA     $6CA8,X
        STA     $61
        LDA     $6CA9,X
        STA     $62

        LDY     $97
        LDA     $621D,Y
        STA     $5F
```

Uses GAME_COLNUM 32a, GAME_ROWNUM 32a, GUARD_COUNT 55a, LOAD_LEVEL 82c, NOTE_INDEX 48, PLAYER_COL 55a, PLAYER_ROW 55a, PREGAME_MODE 80d, and TIMES_3_TABLE 108b.

108a ⟨*game loop* 108a⟩≡

```
      JSR     MOVE_PLAYER
```

Uses MOVE_PLAYER 103.

108b ⟨*tables* 7⟩+≡ (109b) ◁99b

```
      ORG     $6214
  TIMES_3_TABLE:
      HEX     00 03 06 09 0C 0F 12 15 18
```

Defines:
  TIMES_3_TABLE, used in chunk 107.

Chapter 7

The whole thing

We then put together the entire assembly file:

109a ⟨*routines* 4⟩+≡ (109b) ◁65

```
; Ideally these are in the order they were placed in the original file.
; However, since each section should start with ORG, it should not be
; necessary.
⟨startup code 94a⟩

; Graphics routines
⟨level draw routine 53⟩
⟨splash screen 92⟩
⟨construct and display high score screen 87b⟩
⟨iris wipe 63⟩
⟨iris wipe step 67⟩
⟨draw wipe step 69a⟩
⟨draw wipe block 73a⟩

; Sound routines
⟨load sound data 49⟩
⟨sound delay 51a⟩
⟨play note 50b⟩
⟨play sound 52⟩
```

109b ⟨* 109b⟩≡

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

Chapter 8

Defined Chunks

Chapter 9

Index