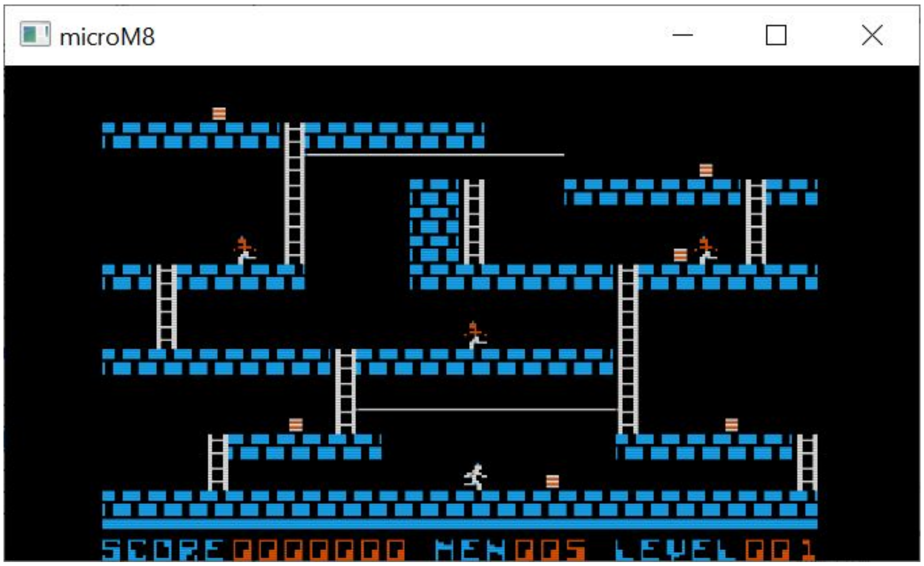
Chapter 1

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⟩≡ (215) 21 ▷

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

Defines:
  `TMP_PTR`, used in chunks 4, 24, 58, and 205.

4 ⟨*routines* 4⟩≡ (215) 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 chunks 51, 117, and 208.
  CLEAR_HGR2, used in chunks 51, 112b, 135, and 189.
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.

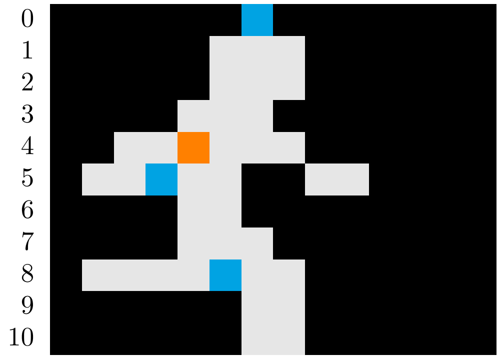
An example would probably be good here. We will take one of the sprites from the game.

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

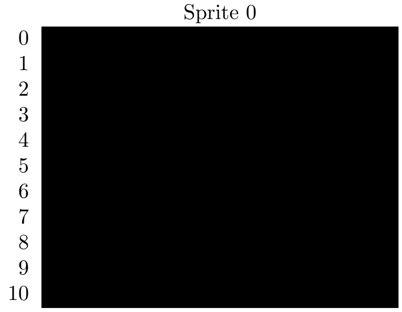
Lode Runner defines 104 sprites, each being 11 rows, with two bytes per row. The first bytes of all 104 sprites are in the table first, then the second bytes, then the third bytes, and so on. Later we will see that only the leftmost 10 pixels out of the 14-pixel description is used.

7 ⟨*tables* 7⟩≡ (215) 22▷

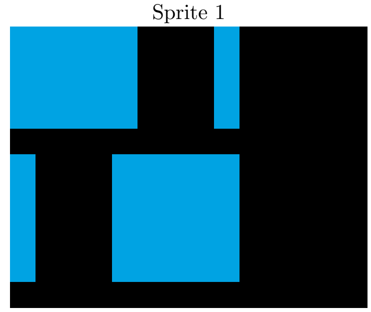
```
    ORG     $AD00
SPRITE_DATA:
    INCLUDE "sprite_data.asm"
```

Defines:
`SPRITE_DATA`, used in chunk 24.

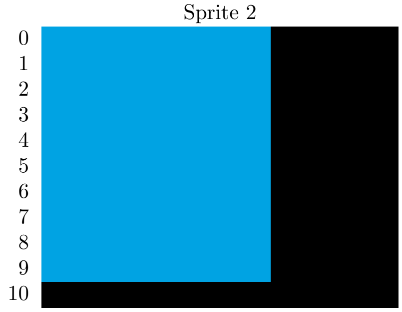
Sprite 0



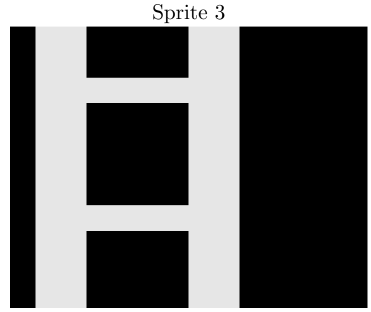
Sprite 1



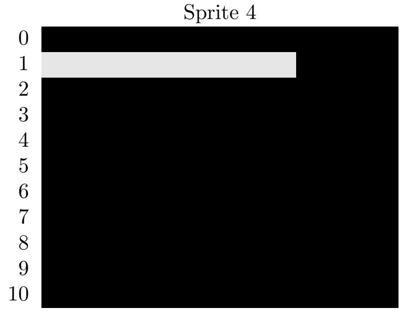
Sprite 2



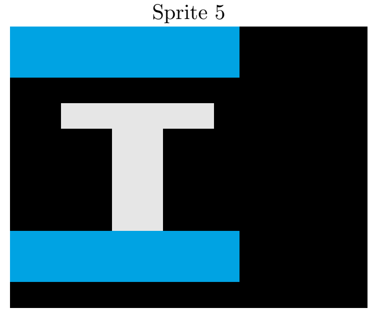
Sprite 3



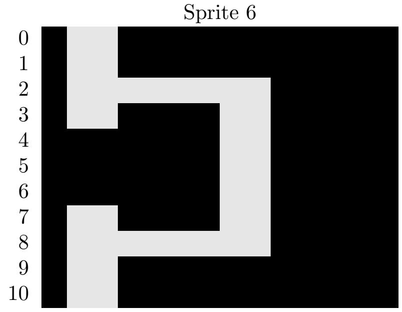
Sprite 4



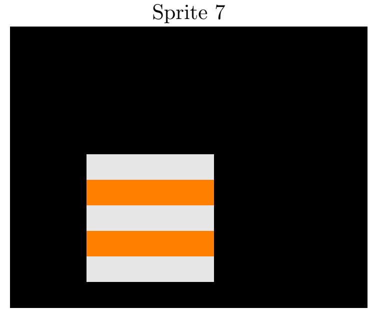
Sprite 5



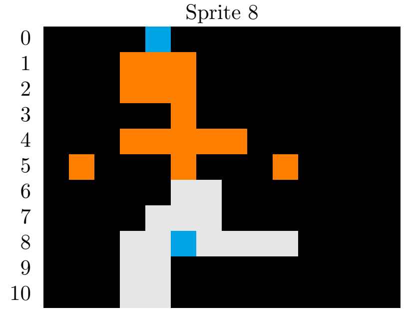
Sprite 6



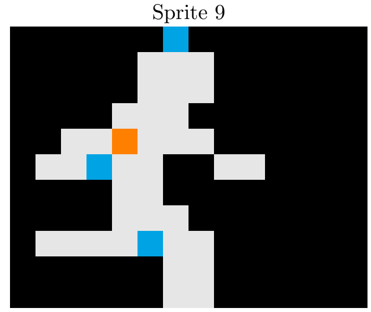
Sprite 7



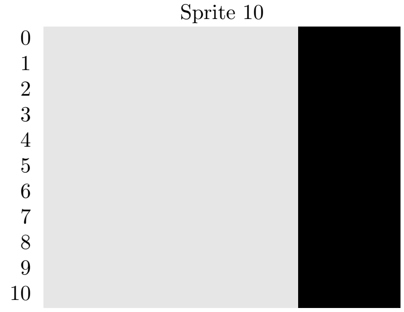
Sprite 8



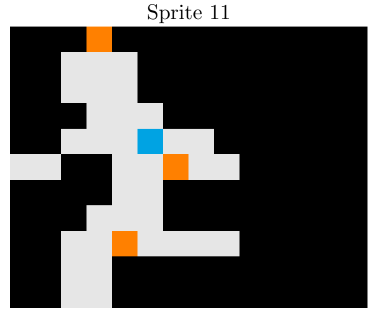
Sprite 9



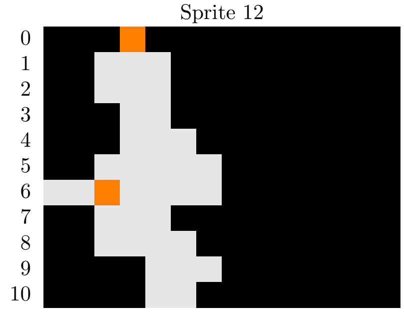
Sprite 10



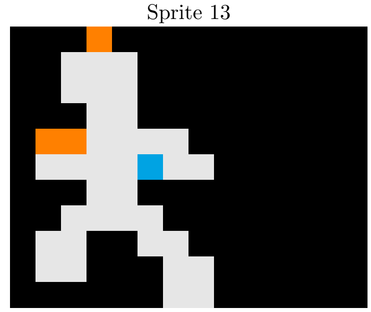
Sprite 11



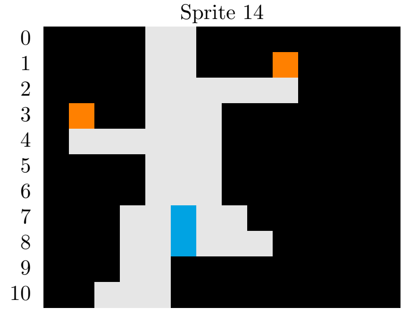
Sprite 12



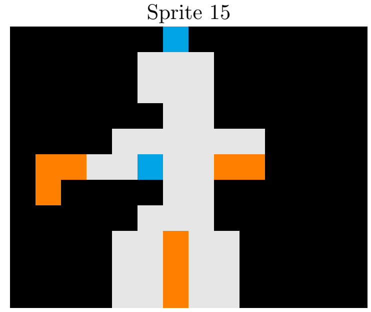
Sprite 13



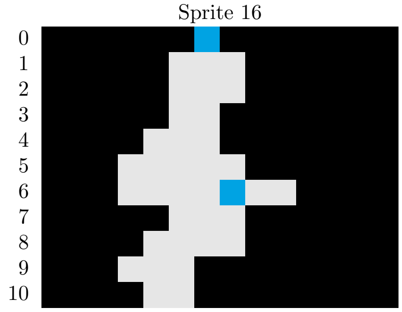
Sprite 14



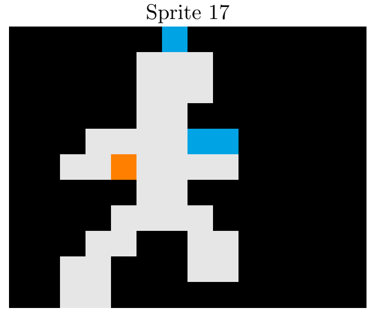
Sprite 15



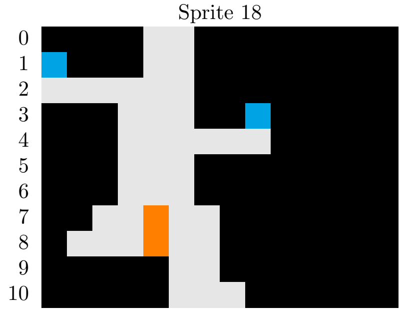
Sprite 16



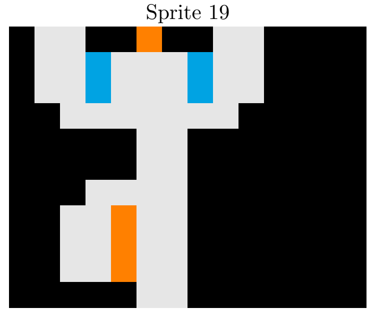
Sprite 17



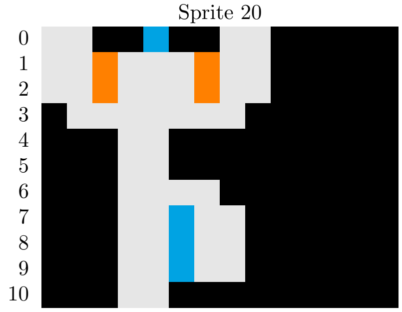
Sprite 18



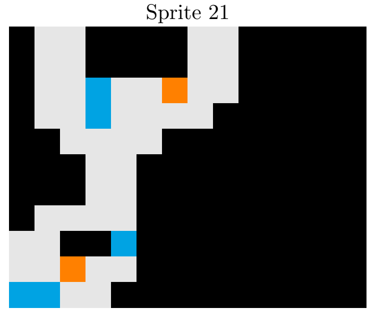
Sprite 19



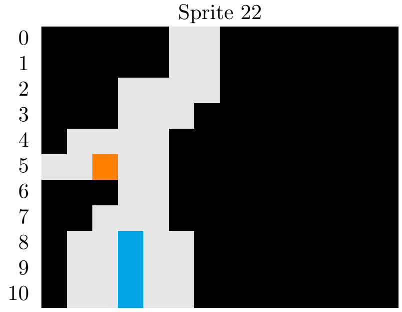
Sprite 20



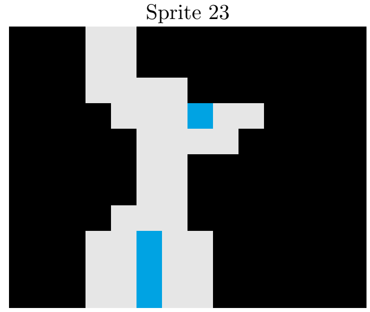
Sprite 21



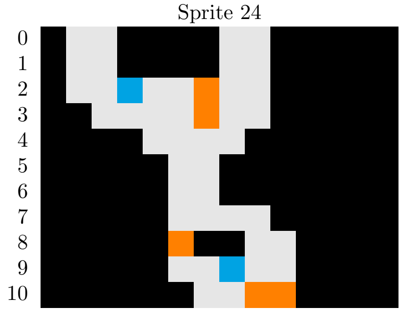
Sprite 22



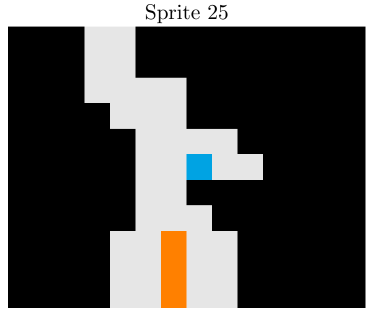
Sprite 23



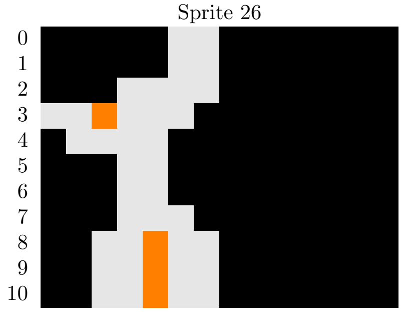
Sprite 24



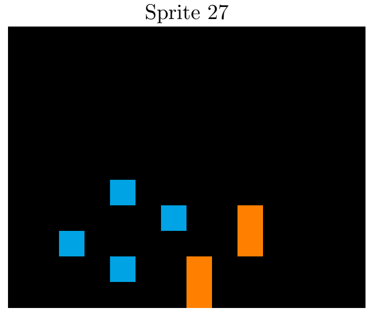
Sprite 25



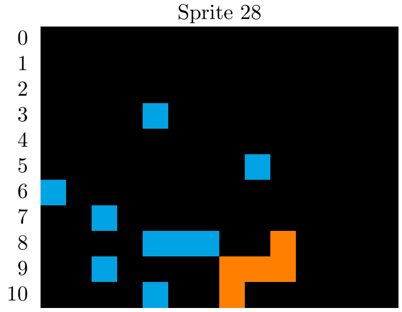
Sprite 26



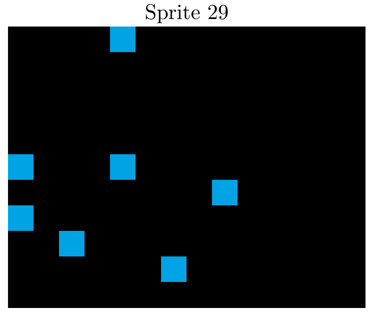
Sprite 27



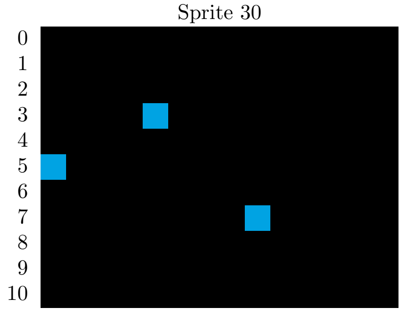
Sprite 28



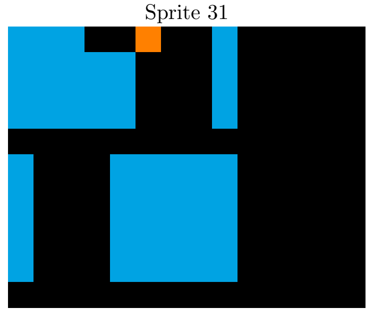
Sprite 29



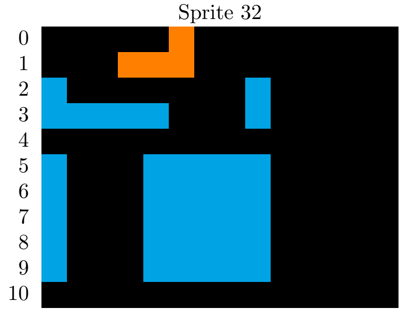
Sprite 30



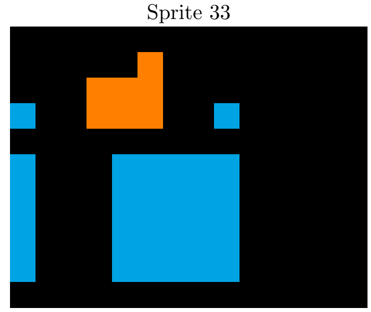
Sprite 31



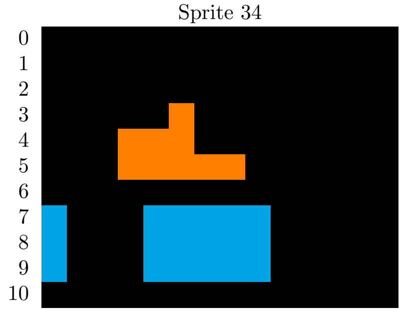
Sprite 32



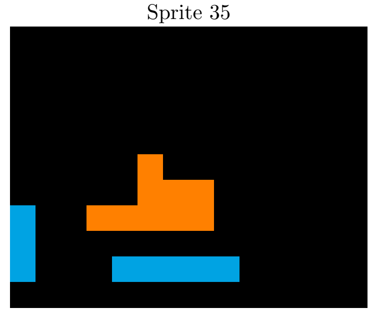
Sprite 33



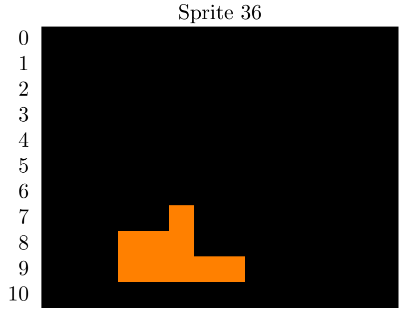
Sprite 34



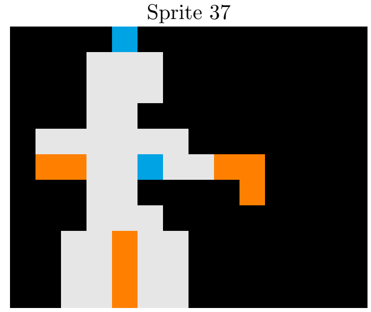
Sprite 35



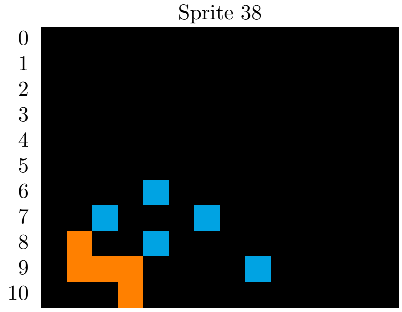
Sprite 36



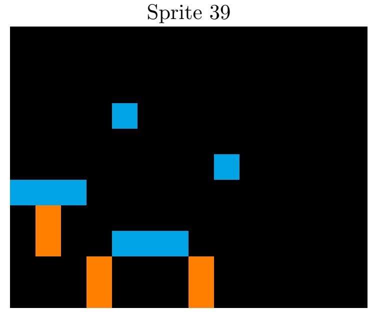
Sprite 37



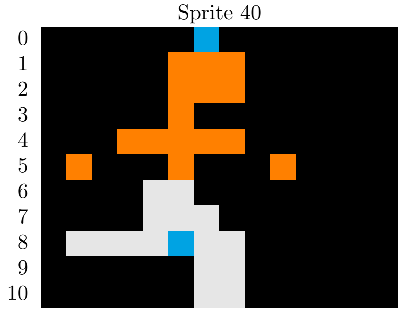
Sprite 38



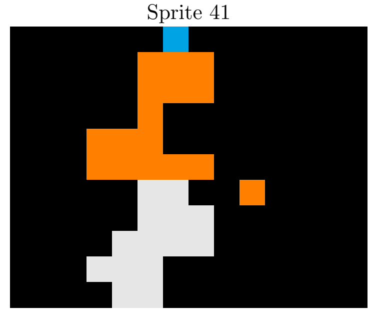
Sprite 39



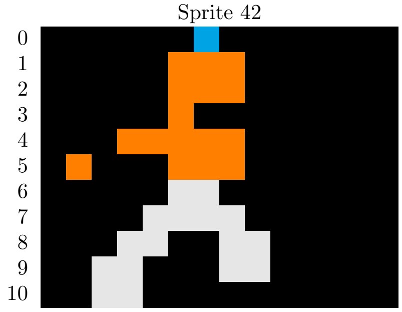
Sprite 40



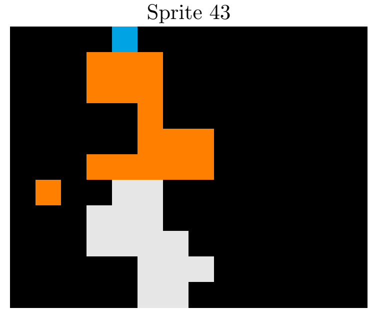
Sprite 41



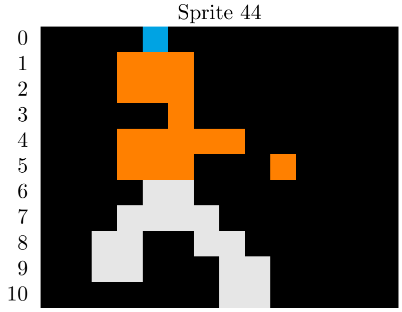
Sprite 42



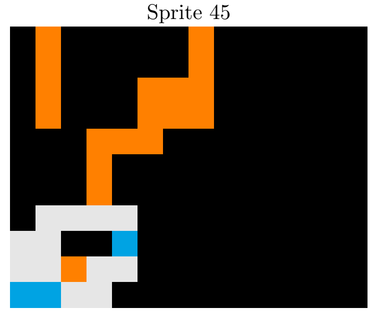
Sprite 43



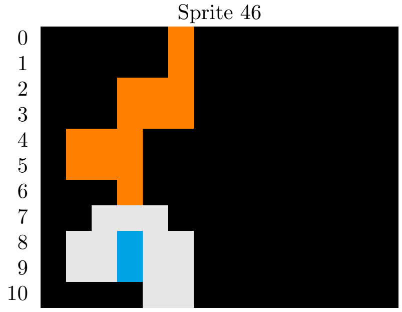
Sprite 44



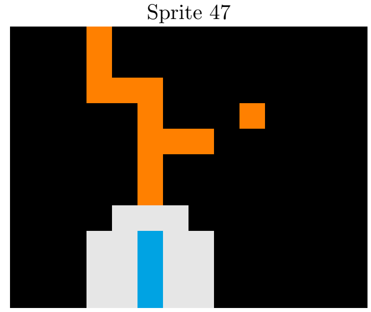
Sprite 45



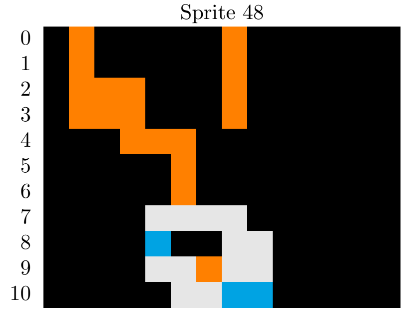
Sprite 46



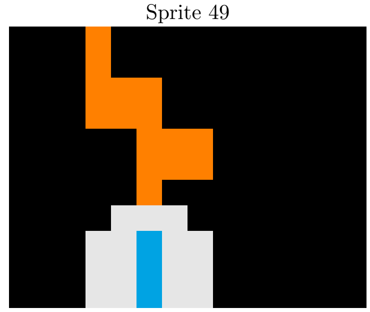
Sprite 47



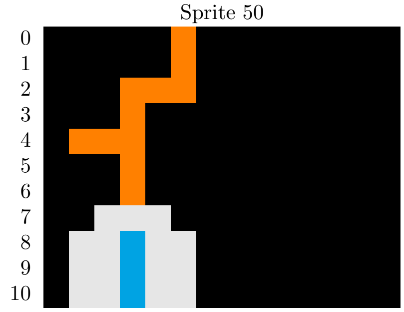
Sprite 48



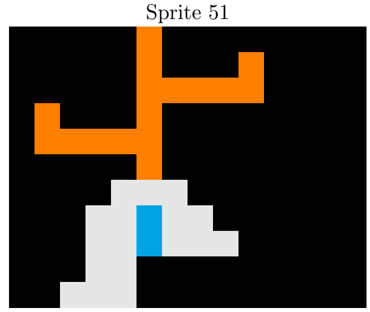
Sprite 49



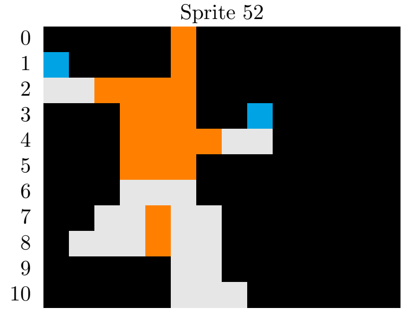
Sprite 50



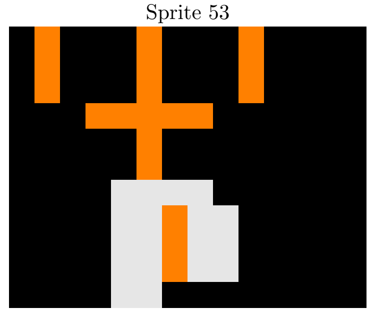
Sprite 51



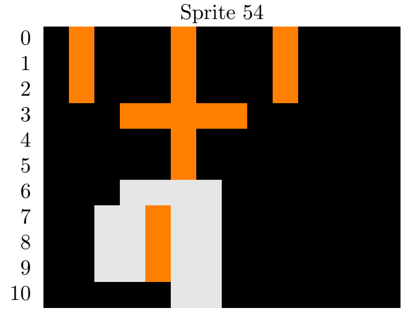
Sprite 52



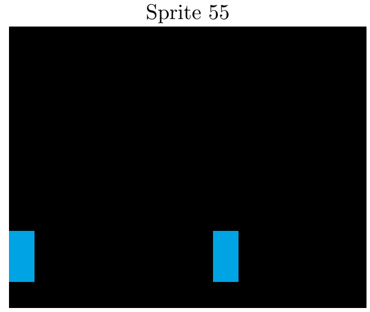
Sprite 53



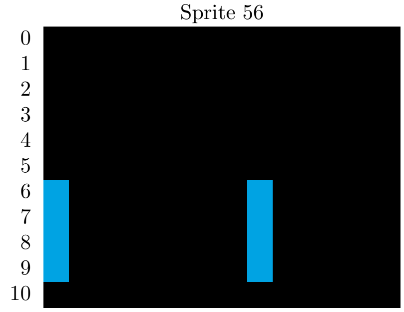
Sprite 54



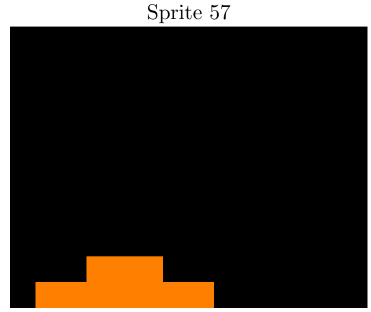
Sprite 55



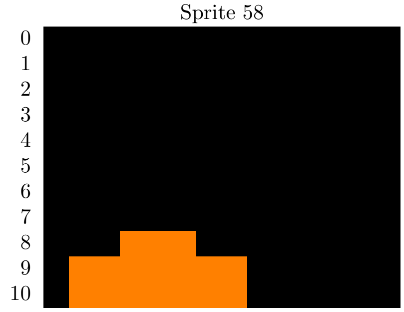
Sprite 56



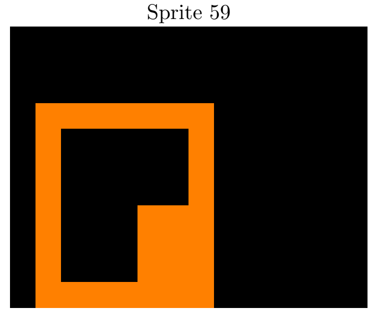
Sprite 57



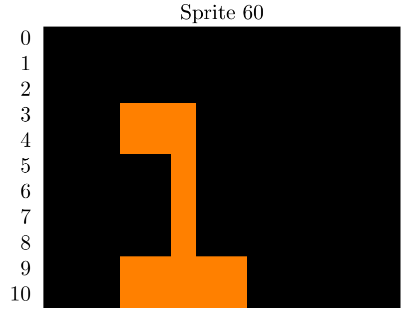
Sprite 58



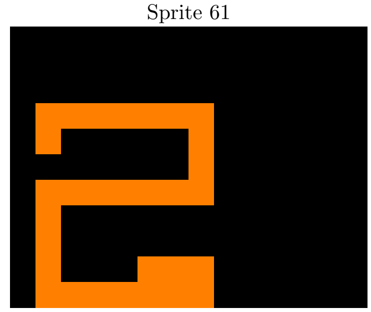
Sprite 59



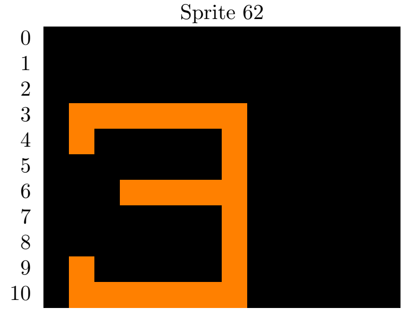
Sprite 60



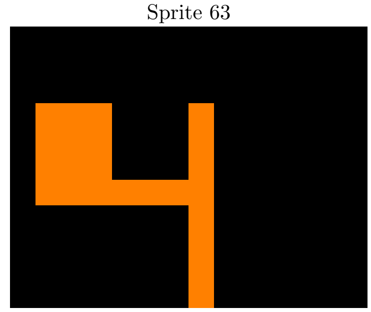
Sprite 61



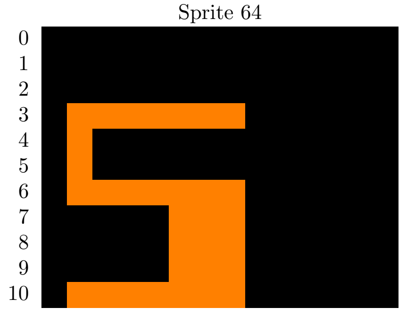
Sprite 62



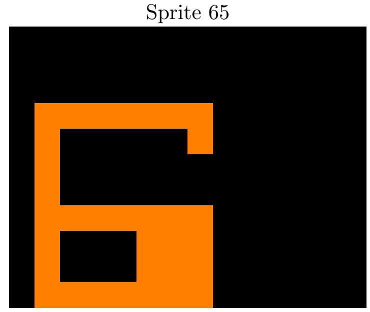
Sprite 63



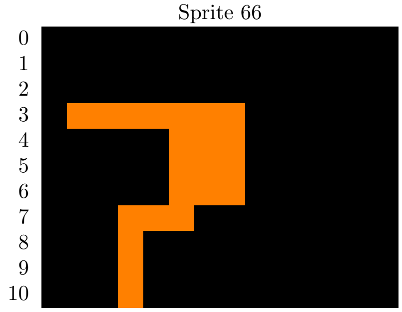
Sprite 64



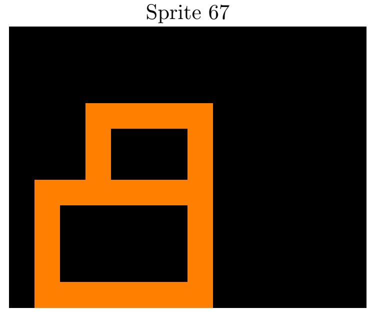
Sprite 65



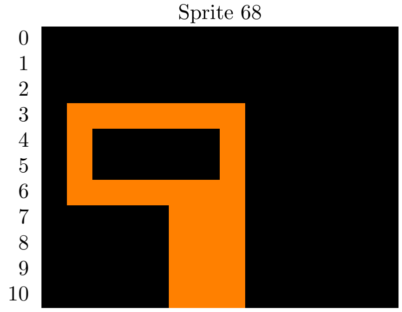
Sprite 66



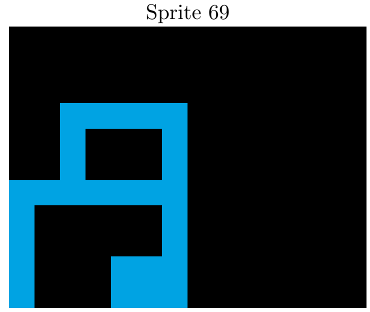
Sprite 67



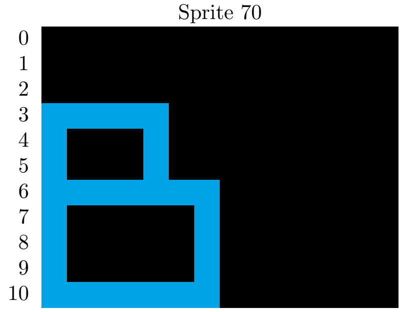
Sprite 68



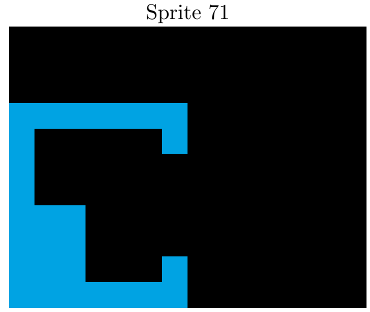
Sprite 69



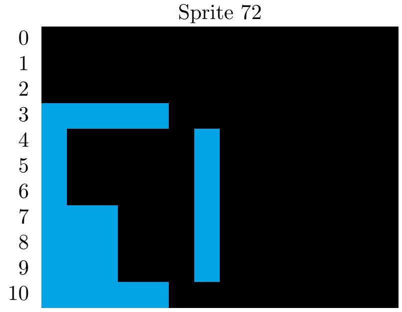
Sprite 70



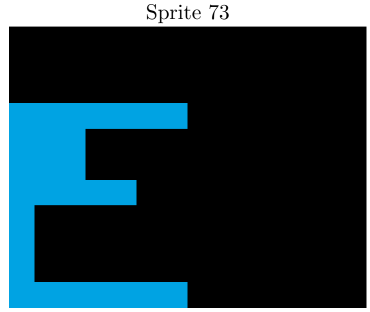
Sprite 71



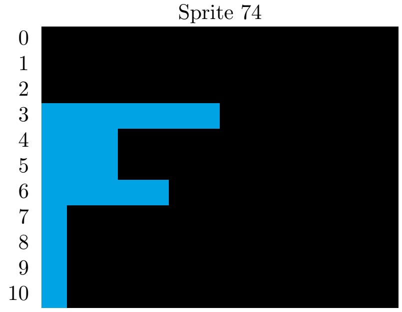
Sprite 72



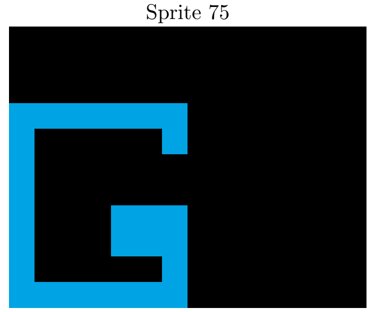
Sprite 73



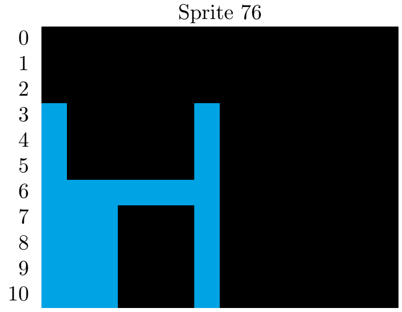
Sprite 74



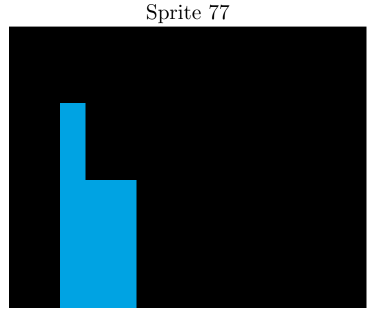
Sprite 75



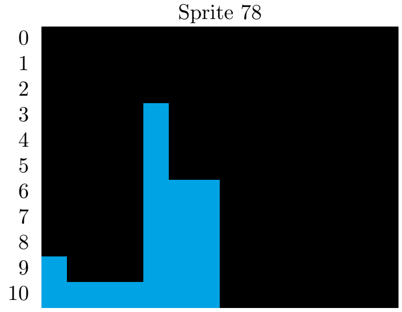
Sprite 76



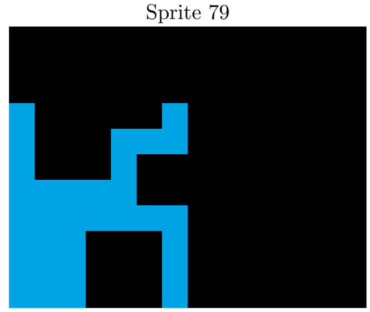
Sprite 77



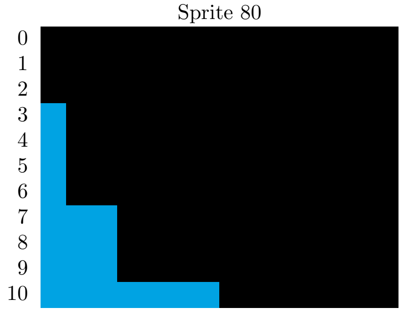
Sprite 78



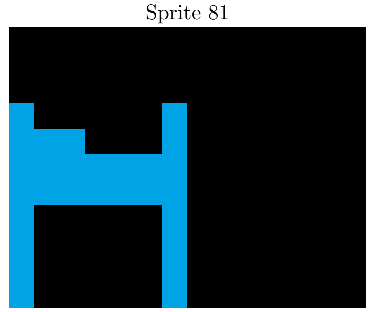
Sprite 79



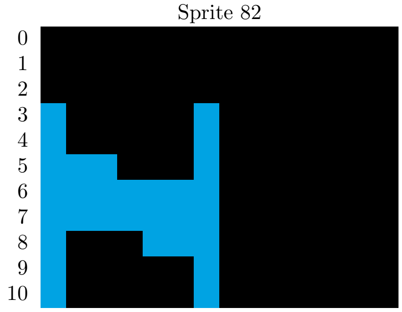
Sprite 80



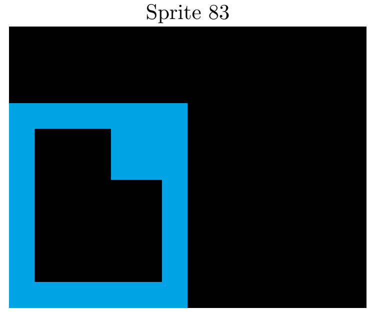
Sprite 81



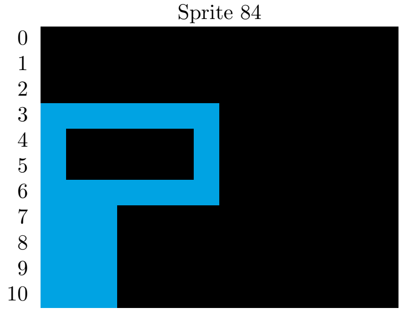
Sprite 82



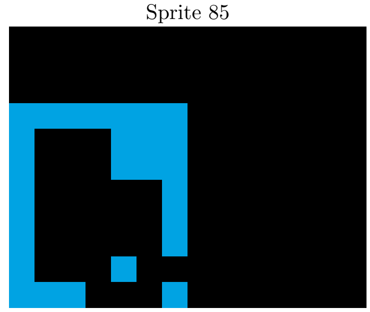
Sprite 83



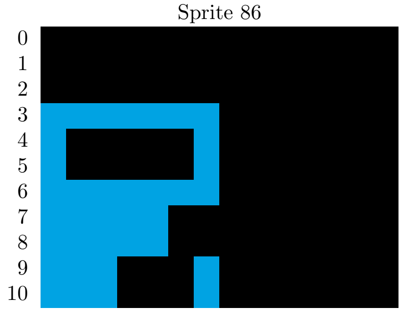
Sprite 84



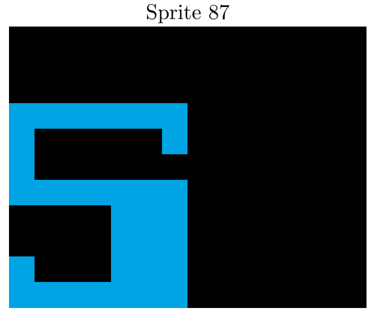
Sprite 85



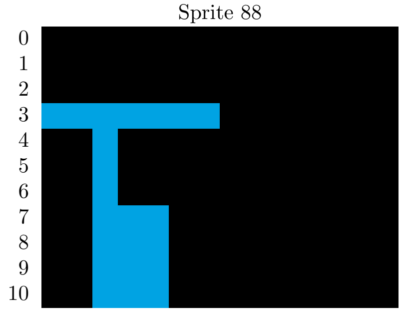
Sprite 86



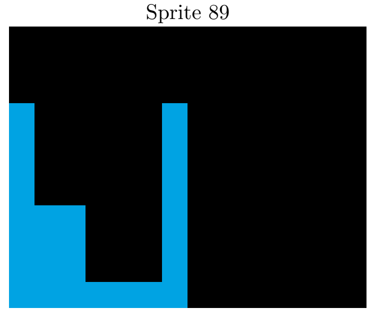
Sprite 87



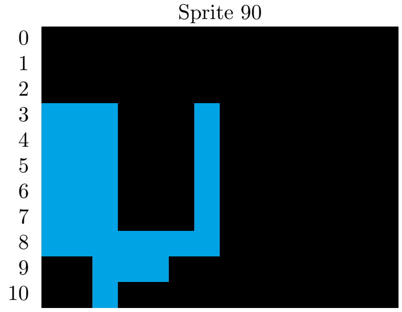
Sprite 88



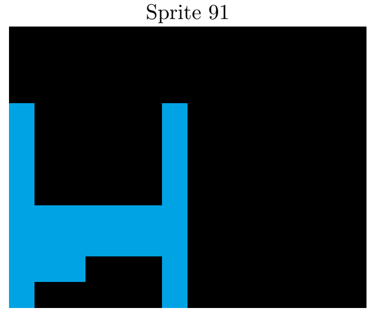
Sprite 89



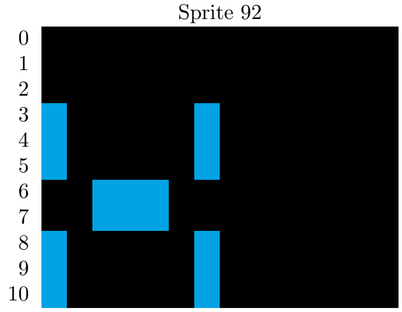
Sprite 90



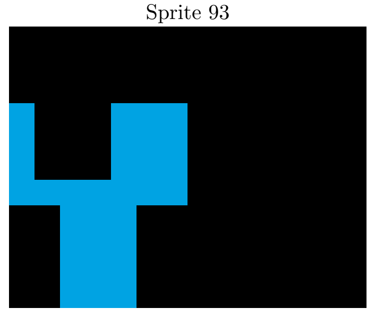
Sprite 91



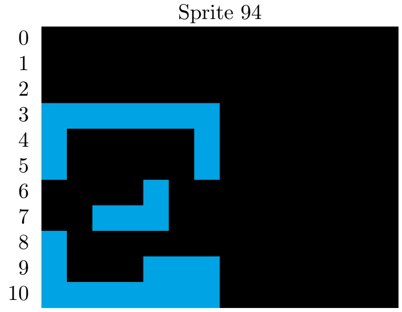
Sprite 92



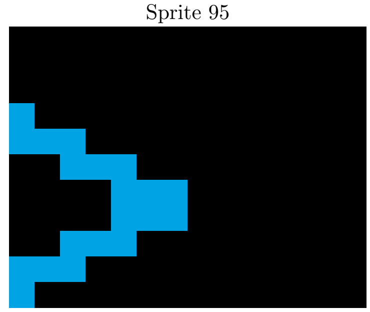
Sprite 93

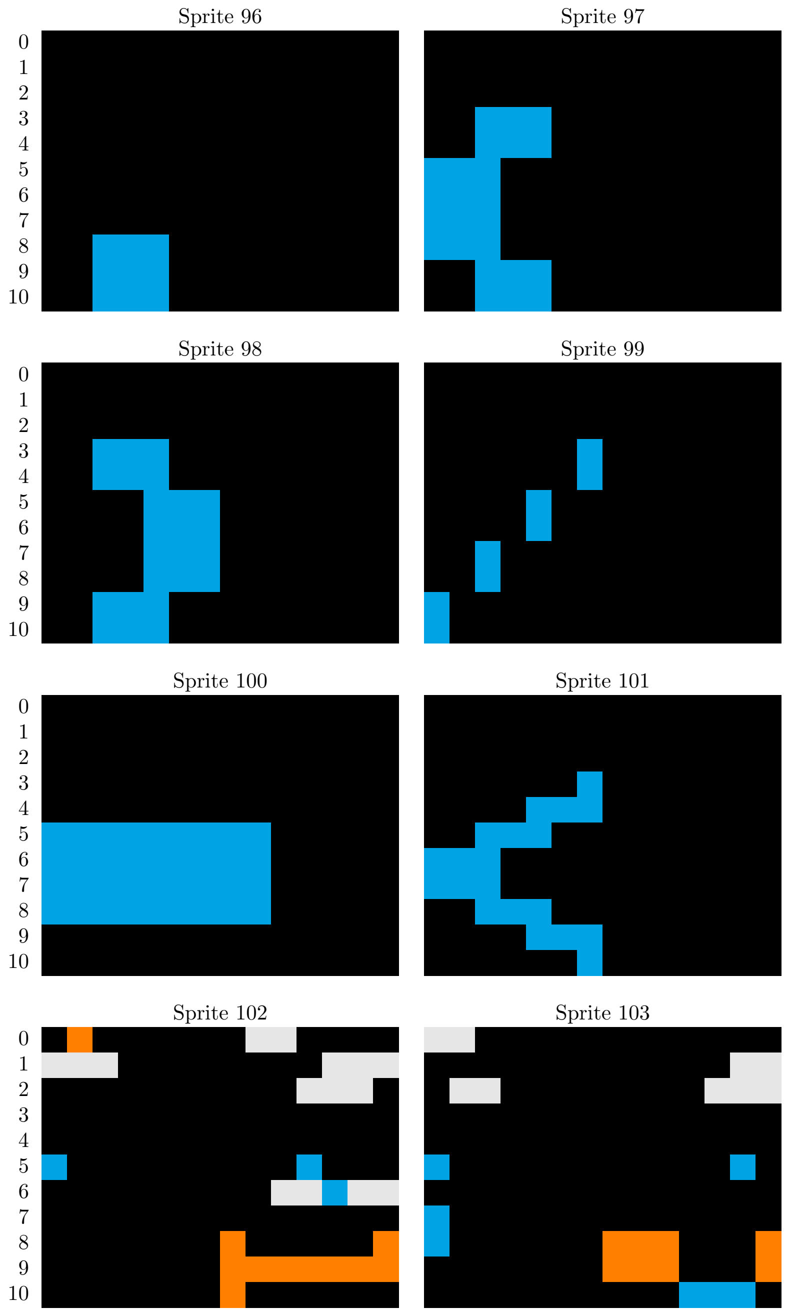


Sprite 94



Sprite 95





2.3 Shifting sprites

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

21 ⟨*defines* 3⟩+≡ (215) ◁3 23c▷

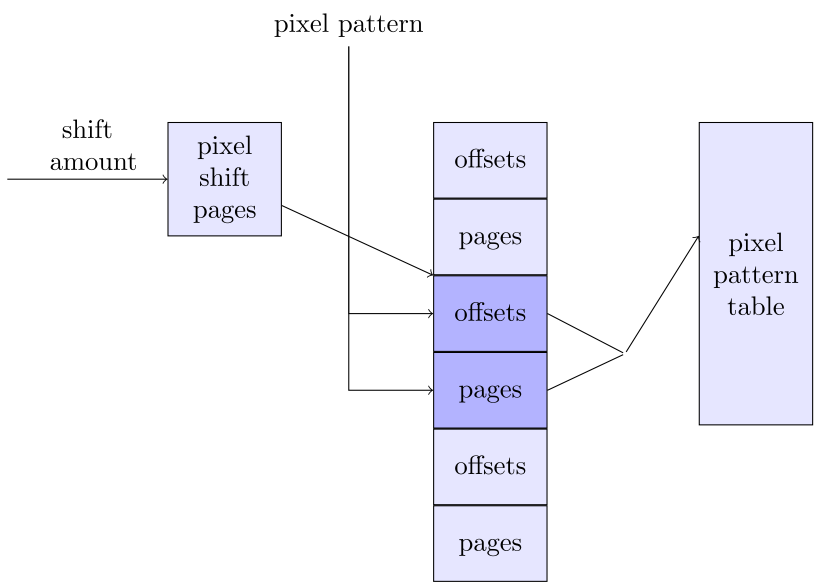
```
      ORG     $DF
BLOCK_DATA      DS      33
```

Defines:
BLOCK_DATA, used in chunks 24, 33, 36, and 39.

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⟩+≡ (215) ◁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⟩+≡ (215) ◁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.

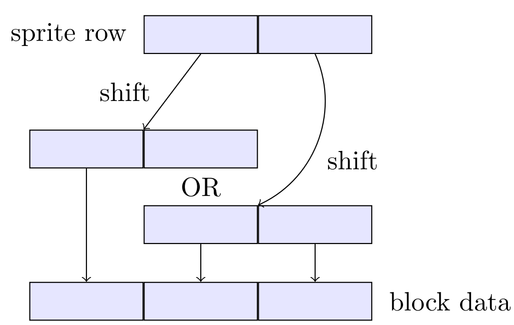
23b ⟨*tables* 7⟩+≡ (215) ◁23a 26a▷

```
    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⟩+≡ (215) ◁21 26b▷

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

Defines:
`ROW_COUNT`, used in chunks 24, 33, 36, 39, and 185.
`SPRITE_NUM`, used in chunks 24, 33, 36, 39, 125a, 129, and 171b.

24 ⟨*routines 4*⟩+≡ (215) ◁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, 36, and 39.

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⟩+≡ (215) ◁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⟩+≡ (215) ◁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, 117, and 203.
ROW_ADDR, used in chunks 26c, 27, 33, 36, 39, 83, 96a, 106, 118, and 205.
ROW_ADDR2, used in chunks 27, 36, 39, 83, and 96a.

26c ⟨*routines* 4⟩+≡ (215) ◁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, 118, and 205.
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⟩+≡ (215) ◁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, 39, and 92–95.
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⟩+≡ (215) ◁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⟩+≡ (215) ◁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, 127, 157, 160, 174, and 179.
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⟩+≡ (215) ◁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⟩+≡ (215) ◁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 chunks 36 and 39.
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⟩+≡ (215) ◁28 31b▷

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

Defines:
ROW_OFFSET_TABLE, used in chunk 31a.

31a ⟨*routines* 4⟩+≡ (215) ◁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 chunks 125a and 171b.
Uses GET_SCREEN_COORDS_FOR 29a and ROW_OFFSET_TABLE 30b.

31b ⟨*tables* 7⟩+≡ (215) ◁30b 32b▷
```
      ORG     $889D
  COL_OFFSET_TABLE:
      HEX     FE FF 00 01 02
```
Defines:
  COL_OFFSET_TABLE, used in chunk 31c.

31c ⟨*routines* 4⟩+≡ (215) ◁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 chunks 125a and 171b.
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⟩+≡ (215) ◁26b 38▷

```
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, 36, and 39.
  COLNUM, used in chunks 33, 36, and 39.
  GAME_COLNUM, used in chunks 33, 44a, 46a, 49, 51, 71, 78b, 82b, 84, 110, 112b, 127, 135, 157, 160, 163, 174, 179, 185, 189, 199, and 208.
  GAME_ROWNUM, used in chunks 33, 44a, 49, 51, 74, 79–82, 84, 108, 111a, 112b, 117, 118, 120c, 121c, 124b, 127, 135, 157, 160, 163, 174, 179, 185, 189, 199, 205, and 208.
  MASK0, used in chunks 33 and 183.
  MASK1, used in chunk 33.
  ROWNUM, used in chunks 33, 36, and 39.

32b ⟨*tables* 7⟩+≡ (215) ◁31b 75a▷

```
    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⟩+≡ (215) ◁31c 88▷

```
    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 44a, 46a, 68, 157, 160, and 163.
DRAW_SPRITE_PAGE2, used in chunks 44a, 46a, 67, 82b, 84, 127, 135, 163, 174, and 179.

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 erases a sprite at a given screen coordinate. It does this by drawing the inverse of the sprite on page 1, then drawing the sprite data from page 2 onto page 1.

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 ⟨*erase sprite at screen coordinate* 36⟩≡ (213)

```
    ORG     $8336
ERASE_SPRITE_AT_PIXEL_COORDS:
    SUBROUTINE
    ; Enter routine with A set to sprite number to draw,
    ; Y set to the screen row to erase it at, and X
    ; set to *half* the screen column to erase 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             ; screen[COLNUM] =
                                     ;   (screen[COLNUM] & (BLOCK_DATA[i] ^ 0x7F)) | screen2[COLNUM]

    INX                              ; X++
    INY                              ; Y++
    LDA     BLOCK_DATA+1,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y             ; screen[COLNUM+1] =
                                     ;   (screen[COLNUM+1] & (BLOCK_DATA[i+1] ^ 0x7F)) | screen2[COLN
```

```
    INX                         ; X++
    INX                         ; X++
    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        ; screen[COLNUM] =
                                ;   (screen[COLNUM] & (BLOCK_DATA[i] ^ 0x7F)) | screen2[COLNUM]

    INX                         ; X++
    INY                         ; Y++
    LDA     BLOCK_DATA+1,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+1] =
                                ;   (screen[COLNUM+1] & (BLOCK_DATA[i+1] ^ 0x7F)) | screen2[COLN

    INX                         ; X++
    INY                         ; Y++
    LDA     BLOCK_DATA+2,X
    EOR     #$7F
    AND     (ROW_ADDR),Y
    ORA     (ROW_ADDR2),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+2] =
                                ;   (screen[COLNUM+2] & (BLOCK_DATA[i+2] ^ 0x7F)) | screen2[COLN

    INX                         ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .need_3_bytes
    RTS
```

Defines:
ERASE_SPRITE_AT_PIXEL_COORDS, used in chunks 127, 146, 148, 150, 153, 157, 160, 164, and 174.

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.

And then there's the corresponding routine to draw a sprite at the given coordinates. The routine also sets whether the active and the background screens differ in SCREENS_DIFFER.

38 ⟨*defines* 3⟩+≡ (215) ◁32a 43▷

```
SCREENS_DIFFER      EQU     $52
```

Defines:
SCREENS_DIFFER, used in chunks 39 and 41.

39 ⟨*draw sprite at screen coordinate* 39⟩≡ (213)
```
    ORG     $83A7
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
    STX     SCREENS_DIFFER      ; SCREENS_DIFFER = 0
    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     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER          ; SCREENS_DIFFER |=
                                    ;   ( (screen[COLNUM] ^ screen2[COLNUM]) & BLOCK_DATA[i])
    LDA     BLOCK_DATA,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM] |= BLOCK_DATA[i]

    INX                             ; X++
    INY                             ; Y++
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA+1,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER          ; SCREENS_DIFFER |=
                                    ;   ( (screen[COLNUM+1] ^ screen2[COLNUM+1]) & BLOCK_DATA[i+1])
    LDA     BLOCK_DATA+1,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y            ; screen[COLNUM+1] |= BLOCK_DATA[i+1]
```

```
    INX                         ; X++
    INX                         ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .loop1
    RTS

.need_3_bytes:
    LDY     ROWNUM
    JSR     ROW_TO_ADDR_FOR_BOTH_PAGES
    LDY     COLNUM
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM] ^ screen2[COLNUM]) & BLOCK_DATA[i])
    LDA     BLOCK_DATA,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM] |= BLOCK_DATA[i]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA+1,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM+1] ^ screen2[COLNUM+1]) & BLOCK_DATA[i+1])
    LDA     BLOCK_DATA+1,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+1] |= BLOCK_DATA[i+1]

    INX                         ; X++
    INY                         ; Y++
    LDA     (ROW_ADDR),Y
    EOR     (ROW_ADDR2),Y
    AND     BLOCK_DATA+2,X
    ORA     SCREENS_DIFFER
    STA     SCREENS_DIFFER      ; SCREENS_DIFFER |=
                                ;   ( (screen[COLNUM+2] ^ screen2[COLNUM+2]) & BLOCK_DATA[i+2])
    LDA     BLOCK_DATA+2,X
    ORA     (ROW_ADDR),Y
    STA     (ROW_ADDR),Y        ; screen[COLNUM+2] |= BLOCK_DATA[i+2]

    INX                         ; X++
    INC     ROWNUM
    DEC     ROW_COUNT
    BNE     .loop1
    RTS
```

Defines:
DRAW_SPRITE_AT_PIXEL_COORDS, used in chunks 41, 157, 160, 174, and 179.
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, SCREENS_DIFFER 38, and SPRITE_NUM 23c.

There is a special routine to draw the player sprite at the player's location. If the two pages at the player's location are different and the player didn't pick up gold (which would explain the difference), then the player is killed.

41 ⟨*draw player* 41⟩≡ (213)
```
    ORG     $6C02
DRAW_PLAYER:
    SUBROUTINE

    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    LDA     SCREENS_DIFFER
    BEQ     .end
    LDA     DIDNT_PICK_UP_GOLD
    BEQ     .end
    LSR     ALIVE       ; Set player as dead
.end
    RTS
```
Defines:
DRAW_PLAYER, used in chunks 146, 150, 153, 157, 160, and 164.
Uses ALIVE 106d, DIDNT_PICK_UP_GOLD 126, DRAW_SPRITE_AT_PIXEL_COORDS 39, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, and SCREENS_DIFFER 38.

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.

42 ⟨*char to sprite num* 42⟩≡ (213)

```
    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 chunks 44a and 185.

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

43 ⟨*defines* 3⟩+≡ (215) ◁38 44b▷

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

Defines:
DRAW_PAGE, used in chunks 44a, 46a, 51, 112b, 116, 117, 185, 189, and 208.

44a ⟨*put char* 44a⟩≡ (213)

```
    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 115b.
PUT_CHAR, used in chunks 45, 113c, 114b, and 185.
Uses CHAR_TO_SPRITE_NUM 42, DRAW_PAGE 43, 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.

44b ⟨*defines* 3⟩+≡ (215) ◁43 46b▷

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

Defines:
SAVED_RET_ADDR, used in chunks 45 and 56.

45 ⟨*put string* 45⟩≡ (213)

```
    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 51, 70a, 113, 114, 189, 192, 195, 208, 210a, and 211.
Uses `PUT_CHAR` 44a and `SAVED_RET_ADDR` 44b.

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.

46a ⟨*put digit* 46a⟩≡ (213)
```
    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 49, 51, 71, and 113–15.
Uses DRAW_PAGE 43, 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.

46b ⟨*defines* 3⟩+≡ (215) ◁44b 48b▷
```
    ORG     $C0
HUNDREDS        DS      1
TENS            DS      1
UNITS           DS      1
```
Defines:
HUNDREDS, used in chunks 47, 51, 71, and 114c.
TENS, used in chunks 47–49, 51, 71, 114c, and 115a.
UNITS, used in chunks 47–49, 51, 71, 114c, and 115a.

47 ⟨*to decimal3* 47⟩≡ (213)

```
    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 51, 71, and 114c.
Uses `HUNDREDS` 46b, `TENS` 46b, and `UNITS` 46b.

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

48a ⟨*bcd to decimal2* 48a⟩≡ (213)

```
      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 49 and 115a.
Uses TENS 46b and UNITS 46b.

2.7 Score and status

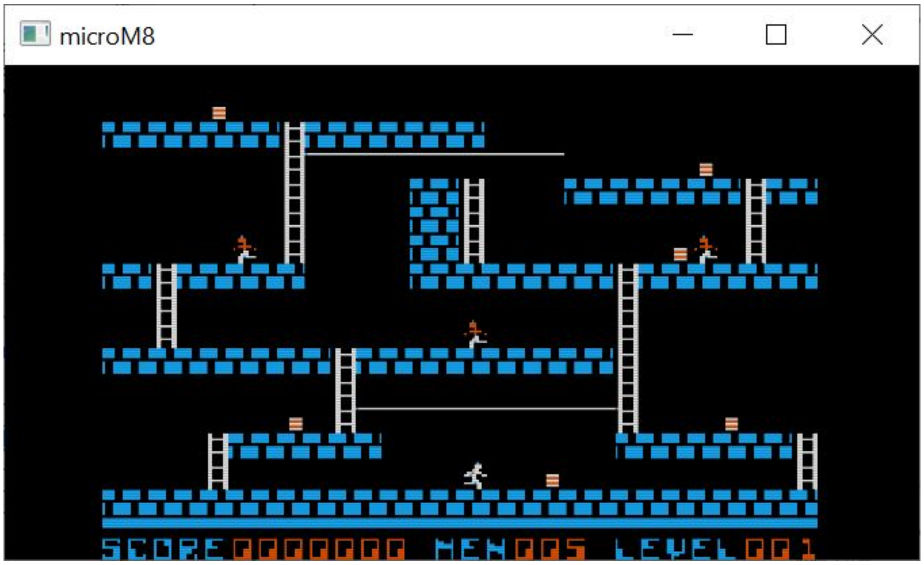
Lode Runner stores your score as an 8-digit BCD number.

48b ⟨*defines* 3⟩+≡ (215) ◁46b 50▷

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

Defines:
  SCORE, used in chunks 49, 51, 113a, 127, 174, 185, 195, 197, 200, and 208.

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.

49 ⟨*add and update score* 49⟩≡ (213)

```
    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 chunks 51, 127, 174, and 200.
Uses BCD_TO_DECIMAL2 48a, GAME_COLNUM 32a, GAME_ROWNUM 32a, PUT_DIGIT 46a, SCORE 48b, TENS 46b, and UNITS 46b.

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

50 ⟨*defines* 3⟩+≡ (215) ◁48b 55▷

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

Defines:
LEVELNUM, used in chunks 51, 71, 106b, 122a, 123a, 130a, 185, and 200.
LIVES, used in chunks 51, 130, 131c, 138, 197, 200, and 208.

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.

51 ⟨*put status* 51⟩≡ (213)

```
    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

    ORG     $79AD
PUT_STATUS:
    SUBROUTINE

    JSR     CLEAR_HGR1
    JSR     CLEAR_HGR2
    LDY     #$27
    LDA     DRAW_PAGE
    CMP     #$40
    BEQ     .draw_line_on_page_2

.draw_line_on_page_1:
    LDA     #$AA
    STA     $2350,Y
    STA     $2750,Y
```

```
    STA     $2B50,Y
    STA     $2F50,Y
    DEY
    LDA     #$D5
    STA     $2350,Y
    STA     $2750,Y
    STA     $2B50,Y
    STA     $2F50,Y
    DEY
    BPL     .draw_line_on_page_1
    BMI     .end        ; Unconditional

.draw_line_on_page_2:
    LDA     #$AA
    STA     $4350,Y
    STA     $4750,Y
    STA     $4B50,Y
    STA     $4F50,Y
    DEY
    LDA     #$D5
    STA     $4350,Y
    STA     $4750,Y
    STA     $4B50,Y
    STA     $4F50,Y
    DEY
    BPL     .draw_line_on_page_2

.end:
    LDA     #$10
    STA     GAME_ROWNUM
    LDA     #$00
    STA     GAME_COLNUM

    ; "SCORE        MEN    LEVEL   "
    JSR     PUT_STRING
    HEX     D3 C3 CF D2 C5 A0 A0 A0 A0 A0 A0 A0 A0 CD C5 CE
    HEX     A0 A0 A0 A0 CC C5 D6 C5 CC A0 A0 A0 00

    JSR     PUT_STATUS_LIVES
    JSR     PUT_STATUS_LEVEL
    LDA     #$00
    TAY
    JMP     ADD_AND_UPDATE_SCORE        ; tailcall
```

Defines:
PUT_STATUS, used in chunk 197.
PUT_STATUS_LEVEL, used in chunk 86.
PUT_STATUS_LIVES, used in chunks 86, 130b, and 200.
Uses ADD_AND_UPDATE_SCORE 49, CLEAR_HGR1 4, CLEAR_HGR2 4, DRAW_PAGE 43, GAME_COLNUM 32a, GAME_ROWNUM 32a, HUNDREDS 46b, LEVELNUM 50, LIVES 50, PUT_DIGIT 46a, PUT_STRING 45,

SCORE 48b, TENS 46b, TO_DECIMAL3 47, and UNITS 46b.

Chapter 3

Sound

3.1 Simple beep

This simple beep routine clicks the speaker every 656 cycles. At approximately 980 nsec per cycle, this would be a period of about 0.64 milliseconds, or a tone of 1.56 kHz. This is a short beep, playing for a little over 0.1 seconds.

54 ⟨*beep* 54⟩≡ (213)

```
      ORG     $86CE
  BEEP:
      SUBROUTINE

      LDY     #$C0

  .loop:
      ; From here to click is 651 cycles. Additional 5 cycles afterwards.
      LDX     #$80            ; 2 cycles

      ; delay 640 cycles
  .loop2:
      DEX                     ; 2 cycles
      BNE     .loop2          ; 3 cycles

      LDA     ENABLE_SOUND    ; 3 cycles
      BEQ     .next           ; 3 cycles
      LDA     SPKR            ; 3 cycles

  .next:
      DEY                     ; 2 cycles
      BNE     .loop           ; 3 cycles
      RTS
```

Defines:
  `BEEP`, used in chunks 70a, 71, 185, 208, and 210a.
Uses `ENABLE_SOUND` 57b and `SPKR` 57b.

3.2 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.

55 ⟨*defines* 3⟩+≡ (215) ◁50 57b▷

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

Defines:
NOTE_INDEX, used in chunks 56, 57a, 60, and 199.
SOUND_DURATION, used in chunks 56, 57a, and 60.
SOUND_PITCH, used in chunks 56, 57a, and 60.

56 ⟨*load sound data* 56⟩≡ (213)

```
    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 chunks 127, 174, and 200.

Uses `NOTE_INDEX` 55, `SAVED_RET_ADDR` 44b, `SOUND_DURATION` 55, and `SOUND_PITCH` 55.

There's also a simple routine to append a single note to the sound buffer. The routine gets called with the pitch in A and the duration in X.

57a ⟨*append note* 57a⟩≡ (213)
```
    ORG     $87D5
APPEND_NOTE:
    SUBROUTINE

    INC     NOTE_INDEX
    LDY     NOTE_INDEX
    STA     SOUND_PITCH,Y
    TXA
    STA     SOUND_DURATION,Y
    RTS
```
Defines:
APPEND_NOTE, used in chunks 61b, 157, and 160.
Uses NOTE_INDEX 55, SOUND_DURATION 55, and SOUND_PITCH 55.

3.3 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.

57b ⟨*defines* 3⟩+≡ (215) ◁55 59b▷
```
ENABLE_SOUND    EQU     $99     ; If 0, do not click speaker.
SPKR            EQU     $C030   ; Access clicks the speaker.
```
Defines:
ENABLE_SOUND, used in chunks 54, 58, 123a, and 132a.
SPKR, used in chunks 54 and 58.

58 ⟨*play note* 58⟩≡ (213)
```
    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 60 and 164.
Uses ENABLE_SOUND 57b, SPKR 57b, and TMP_PTR 3.

3.4 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.

59a ⟨*sound delay* 59a⟩≡ (213)

```
    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 60.

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 + FRAME_PERIOD.

The routine is designed to delay approximately the same amount regardless of sound duration. The delay is controlled by FRAME_PERIOD. This value is hardcoded to 6 initially, but the game can be sped up, slowed down, or even paused.

59b ⟨*defines* 3⟩+≡ (215) ◁57b 61a▷

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

Defines:
FRAME_PERIOD, used in chunks 60 and 133.

60 ⟨*play sound* 60⟩≡ (213)

```
    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     FRAME_PERIOD
    SEC
    SBC     SOUND_DURATION,Y        ; A = FRAME_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     FRAME_PERIOD
    INX
    INX                     ; duration = FRAME_PERIOD + 2
    JSR     PLAY_NOTE

    CLC
    RTS

.end:
    LDX     FRAME_PERIOD
    JSR     SOUND_DELAY

    CLC
    RTS
```

Defines:
PLAY_SOUND, used in chunks 61b and 200.

Uses FRAME_PERIOD 59b, NOTE_INDEX 55, PLAY_NOTE 58, SOUND_DELAY 59a, SOUND_DURATION 55, and SOUND_PITCH 55.

Another routine is just for when a level is cleared. It appends a note based on a scratch location, and then plays it.

61a ⟨*defines* 3⟩+≡ (215) ◁59b 62▷

```
SCRATCH_5C      EQU     $5C
```

Defines:
SCRATCH_5C, used in chunks 61b and 200.

61b ⟨*append level cleared note* 61b⟩≡ (213)

```
    ORG     $622A
APPEND_LEVEL_CLEARED_NOTE:
    SUBROUTINE

    LDA     SCRATCH_5C
    ASL
    ASL
    ASL
    ASL                         ; pitch = SCRATCH_5C * 16
    LDX     #$06                ; duration
    JSR     APPEND_NOTE
    JMP     PLAY_SOUND
```

Defines:
APPEND_LEVEL_CLEARED_NOTE, used in chunk 200.
Uses APPEND_NOTE 57a, PLAY_SOUND 60, and SCRATCH_5C 61a.

Chapter 4

Input

4.1 Joystick input

Analog joysticks (or paddles) on the Apple //e are just variable resistors. The resistor on a paddle creates an RC circuit with a capacitor which can be discharged by accessing the `PTRIG` location. Once that is done, the capacitor starts charging through the resistor. The lower the resistor value, the faster the charge.

At the start, each `PADDL` value has its high bit set to one. When the voltage on the capacitor reaches `2/3` of the supply voltage, the corresponding `PADDL` switch will have its high bit set to zero. So, we just need to watch the `PADDL` value until it is non-negative, counting the amount of time it takes for that to happen.

In the `READ_PADDLES` routine, we trigger the paddles and then alternately read `PADDL0` and `PADDL1` until one of them indicates the threshold was reached. If the `PADDL` value hasn't yet triggered, we increment the corresponding `PADDLE_VALUE` location.

Once a `PADDL` triggers, we stop incrementing the corresponding `PADDLE_VALUE`.

Once both `PADDL` have been triggered, we end the routine.

62 ⟨*defines* 3⟩+≡ (215) ◁61a 64▷

```
PADDLE0_VALUE       EQU     $65
PADDLE1_VALUE       EQU     $66
PADDL0              EQU     $C064
PADDL1              EQU     $C065
PTRIG               EQU     $C070
```

Defines:
- `PADDL0`, used in chunk 63.
- `PADDL1`, used in chunk 63.
- `PADDLE0_VALUE`, used in chunks 63, 65, and 140.
- `PADDLE1_VALUE`, used in chunks 63, 65, and 140.

63 ⟨*read paddles* 63⟩≡ (213)

```
    ORG     $8746
READ_PADDLES:
    SUBROUTINE

    LDA     #$00
    STA     PADDLE0_VALUE
    STA     PADDLE1_VALUE       ; Zero out values
    LDA     PTRIG

.loop:
    LDX     #$01                ; Start with paddle 1

.check_paddle:
    LDA     PADDL0,X
    BPL     .threshold_reached
    INC     PADDLE0_VALUE,X
.check_next_paddle
    DEX
    BPL     .check_paddle

    ; Checked both paddles
    LDA     PADDL0
    ORA     PADDL1
    BPL     .end                ; Both paddles triggered, then end.
    LDA     PADDLE0_VALUE
    ORA     PADDLE1_VALUE
    BPL     .loop               ; Unconditional

.threshold_reached:
    NOP
    BPL     .check_next_paddle      ; Unconditional

.end:
    RTS
```

Defines:
  READ_PADDLES, used in chunks 65 and 140.
Uses PADDL0 62, PADDL1 62, PADDLE0_VALUE 62, and PADDLE1_VALUE 62.

The INPUT_MODE location tells whether the player is using keyboard or joystick input.

The CHECK_JOYSTICK_OR_DELAY routine, if we are in joystick mode, reads the paddle values and checks to see if any value is below 0x12 or above 0x3A, and if so, declares that a paddle has a large enough input by setting the carry flag and returning.

If neither paddle has a large enough input, we also check the paddle buttons, and if either one is triggered, we set the carry and return.

Otherwise, if no paddle input was detected, or we're in keyboard mode, we clear the carry and return.

64 ⟨*defines* 3⟩+≡ (215) ◁62 66▷

```
INPUT_MODE  EQU     $95          ; 0xCA = Joystick mode (J), 0xCB = Keyboard mode (K)
    ORG     $95
    HEX     CA                   ; Start in joystick mode
JOYSTICK_MODE   EQU     #$CA
KEYBOARD_MODE   EQU     #$CB

BUTN0       EQU     $C061        ; Or open apple
BUTN1       EQU     $C062        ; Or solid apple
```

Defines:

BUTN0, used in chunks 65, 121a, 135, 138, 140, and 205.
BUTN1, used in chunks 65, 121a, 135, 138, 140, and 205.
INPUT_MODE, used in chunks 65, 121a, 129, 132b, 135, 138, 205, and 208.

65 ⟨*check joystick or delay* 65⟩≡ (213)

```
    ORG     $876D
CHECK_JOYSTICK_OR_DELAY:
    SUBROUTINE

    LDA     INPUT_MODE
    CMP     KEYBOARD_MODE
    BEQ     .delay_and_return       ; Keyboard mode, so just delay and return

    JSR     READ_PADDLES

    LDA     PADDLE0_VALUE
    CMP     #$12
    BCC     .have_joystick_input            ; PADDLE0_VALUE < 0x12
    CMP     #$3B
    BCS     .have_joystick_input            ; PADDLE0_VALUE >= 0x3B

    LDA     PADDLE1_VALUE
    CMP     #$12
    BCC     .have_joystick_input
    CMP     #$3B
    BCS     .have_joystick_input

    LDA     BUTN1
    BMI     .have_joystick_input
    LDA     BUTN0
    BMI     .have_joystick_input

    CLC
    RTS

.have_joystick_input:
    SEC
    RTS

.delay_and_return:
    LDX     #$02
.loop:
    DEY
    BNE     .loop
    DEX
    BNE     .loop
    CLC
    RTS
```

Defines:
CHECK_JOYSTICK_OR_DELAY, used in chunks 67 and 68.
Uses BUTN0 64, BUTN1 64, INPUT_MODE 64, PADDLE0_VALUE 62, PADDLE1_VALUE 62, and READ_PADDLES 63.

4.2 Keyboard routines

66 ⟨*defines* 3⟩+≡ (215) ◁64 70b▷

```
SCRATCH_A1      EQU     $A1
    ORG     $8745
CURSOR_SPRITE:
    HEX     06
```

Defines:
`CURSOR_SPRITE`, used in chunks 67 and 68.

67 ⟨*wait for key* 67⟩≡ (213)

```
     ORG     $85F3
WAIT_FOR_KEY:
     SUBROUTINE
     ; Enter routine with A set to cursor sprite. If zero, sprite 10 (all white)
     ; will be used.

     STA     CURSOR_SPRITE

.loop:
     LDA     #$68
     STA     SCRATCH_A1
     LDA     CURSOR_SPRITE
     BNE     .draw_sprite
     LDA     #$0A            ; all-white sprite
.draw_sprite:
     JSR     DRAW_SPRITE_PAGE2

.loop2:
     LDA     KBD
     BMI     .end            ; on keypress, end

     JSR     CHECK_JOYSTICK_OR_DELAY
     DEC     SCRATCH_A1
     BNE     .loop2

     ; Draw a blank
     LDA     #$00
     JSR     DRAW_SPRITE_PAGE2
     LDA     #$68
     STA     SCRATCH_A1

.loop3:
     LDA     KBD
     BMI     .end
     JSR     CHECK_JOYSTICK_OR_DELAY
     DEC     SCRATCH_A1
     BNE     .loop3
     JMP     .loop

.end:
     PHA
     LDA     CURSOR_SPRITE
     JSR     DRAW_SPRITE_PAGE2
     PLA
     RTS
```

Defines:
  WAIT_FOR_KEY, used in chunks 70a and 185.
Uses CHECK_JOYSTICK_OR_DELAY 65, CURSOR_SPRITE 66, DRAW_SPRITE_PAGE2 33, and KBD 121b.

68 ⟨*wait for key page1* 68⟩≡ (213)
```
    ORG     $8700
WAIT_FOR_KEY_PAGE_1:
    SUBROUTINE
    ; Enter routine with A set to cursor sprite. If zero, sprite 10 (all white)
    ; will be used.

    STA     CURSOR_SPRITE

.loop:
    LDA     #$68
    STA     SCRATCH_A1
    LDA     #$00
    LDX     CURSOR_SPRITE
    BNE     .draw_sprite
    LDA     #$0A            ; all-white sprite
.draw_sprite:
    JSR     DRAW_SPRITE_PAGE1

.loop2:
    LDA     KBD
    BMI     .end            ; on keypress, end

    JSR     CHECK_JOYSTICK_OR_DELAY
    BCS     .end

    DEC     SCRATCH_A1
    BNE     .loop2

    LDA     CURSOR_SPRITE
    JSR     DRAW_SPRITE_PAGE1
    LDA     #$68
    STA     SCRATCH_A1

.loop3:
    LDA     KBD
    BMI     .end

    JSR     CHECK_JOYSTICK_OR_DELAY
    BCS     .end

    DEC     SCRATCH_A1
    BNE     .loop3
    JMP     .loop

.end:
    PHA
    LDA     CURSOR_SPRITE
    JSR     DRAW_SPRITE_PAGE1
    PLA
```

```
    RTS
```

Defines:
WAIT_FOR_KEY_PAGE_1, used in chunks 69 and 71.
Uses CHECK_JOYSTICK_OR_DELAY 65, CURSOR_SPRITE 66, DRAW_SPRITE_PAGE1 33, and KBD 121b.

This routine is used by the level editor whenever we need to wait for a key. If the key isn't the escape key, we can immediately exit, and the caller interprets the key. However, on escape, we abort whatever editor command we were in the middle of, and just go back to the main editor command loop, asking for an editor command.

69 ⟨*editor wait for key* 69⟩≡ (213)

```
    ORG     $823D
EDITOR_WAIT_FOR_KEY:
    SUBROUTINE

    LDA     #$00
    JSR     WAIT_FOR_KEY_PAGE_1
    STA     KBDSTRB
    CMP     #$9B        ; ESC
    BNE     .return
    JMP     EDITOR_COMMAND_LOOP

.return
    RTS
```

Defines:
EDITOR_WAIT_FOR_KEY, used in chunks 192, 195, 208, and 211.
Uses EDITOR_COMMAND_LOOP 208, KBDSTRB 120b, and WAIT_FOR_KEY_PAGE_1 68.

70a ⟨*hit key to continue* 70a⟩≡ (213)
```
    ORG     $80D8
HIT_KEY_TO_CONTINUE:
    SUBROUTINE

    ; "\r"
    ; "\r"
    ; "HIT A KEY TO CONTINUE "
    JSR     PUT_STRING
    HEX     8D 8D C8 C9 D4 A0 C1 A0 CB C5 D9 A0 D4 CF A0 C3
    HEX     CF CE D4 C9 CE D5 C5 A0 00

    JSR     BEEP
    STA     TXTPAGE2
    LDA     #$00
    JSR     WAIT_FOR_KEY
    STA     KBDSTRB
    STA     TXTPAGE1
RETURN_FROM_SUBROUTINE:
    RTS
```
Defines:
HIT_KEY_TO_CONTINUE, used in chunk 189.
RETURN_FROM_SUBROUTINE, used in chunk 190a.
Uses BEEP 54, KBDSTRB 120b, PUT_STRING 45, TXTPAGE1 119b, TXTPAGE2 115c, and WAIT_FOR_KEY 67.

The GET_LEVEL_FROM_KEYBOARD is used by the level editor to ask the user for a 3-digit level number. The current level number, given by DISK_LEVEL_LOC, is put on the screen. Note that DISK_LEVEL_LOC is 0-based, while the levels the user enters are 1-based, so there's an increment at the beginning and a decrement at the end.

The routine handles forward and backward arrows. Hitting the escape key aborts the editor action and dumps the user back into the editor command loop. Hitting the return key accepts the user's input, and the level is stored in DISK_LEVEL_LOC and LEVELNUM.

70b ⟨*defines* 3⟩+≡ (215) ◁66 75b▷
```
SAVED_GAME_COLNUM       EQU     $824E
```
Defines:
SAVED_GAME_COLNUM, used in chunk 71.

71 ⟨*get level from keyboard* 71⟩≡ (213)

```
      ORG     $817B
  GET_LEVEL_FROM_KEYBOARD:
      SUBROUTINE

      LDY     DISK_LEVEL_LOC
      INY
      TYA
      JSR     TO_DECIMAL3     ; make 1-based
      LDA     GAME_COLNUM
      STA     SAVED_GAME_COLNUM

      ; Print current level
  .loop:
      LDA     HUNDREDS,Y
      STY     KBD_ENTRY_INDEX     ; save Y
      JSR     PUT_DIGIT
      LDY     KBD_ENTRY_INDEX     ; restore Y
      INY
      CPY     #$03
      BCC     .loop

      LDA     SAVED_GAME_COLNUM
      STA     GAME_COLNUM
      LDY     #$00
      STY     KBD_ENTRY_INDEX

  .loop2
      LDX     KBD_ENTRY_INDEX
      LDA     HUNDREDS,X
      CLC
      ADC     #$3B            ; sprite = '0' + X
      JSR     WAIT_FOR_KEY_PAGE_1
      STA     KBDSTRB
      CMP     #$8D            ; return
      BEQ     .return_pressed

      CMP     #$88            ; backspace
      BNE     .check_for_fwd_arrow

      LDX     KBD_ENTRY_INDEX
      BEQ     .beep           ; can't backspace past the beginning

      DEC     KBD_ENTRY_INDEX
      DEC     GAME_COLNUM
      JMP     .loop2

  .check_for_fwd_arrow:
      CMP     #$95            ; fwd arrow
      BNE     .check_for_escape
```

```
    LDX     KBD_ENTRY_INDEX
    CPX     #$02
    BEQ     .beep           ; can't fwd past the end

    INC     GAME_COLNUM
    INC     KBD_ENTRY_INDEX
    JMP     .loop2

.check_for_escape:
    CMP     #$9B            ; ESC
    BNE     .check_for_digit
    JMP     EDITOR_COMMAND_LOOP

.check_for_digit:
    CMP     #$B0            ; '0'
    BCC     .beep           ; less than '0' not allowed
    CMP     #$BA            ; '9'+1
    BCS     .beep           ; greater than '9' not allowed

    SEC
    SBC     #$B0            ; char - '0'
    LDY     KBD_ENTRY_INDEX
    STA     HUNDREDS,Y
    JSR     PUT_DIGIT
    INC     KBD_ENTRY_INDEX
    LDA     KBD_ENTRY_INDEX
    CMP     #$03
    BCC     .loop2

    ; Don't allow a fourth digit
    DEC     KBD_ENTRY_INDEX
    DEC     GAME_COLNUM
    JMP     .loop2

.beep:
    JSR     BEEP
    JMP     .loop2

.return_pressed:
    LDA     SAVED_GAME_COLNUM
    CLC
    ADC     #$03
    STA     GAME_COLNUM
    LDA     #$00
    LDX     HUNDREDS
    BEQ     .add_tens

    CLC
.loop_hundreds:
```

```
    ADC     #100
    BCS     .end
    DEX
    BNE     .loop_hundreds

.add_tens:
    LDX     TENS
    BEQ     .add_units

    CLC
.loop_tens:
    ADC     #10
    BCS     .end
    DEX
    BNE     .loop_tens

.add_units:
    CLC
    ADC     UNITS
    BCS     .end

    STA     LEVELNUM
    TAY
    DEY
    STY     DISK_LEVEL_LOC
    CPY     #$96

.end:
    RTS
```

Defines:
GET_LEVEL_FROM_KEYBOARD, used in chunks 210a and 211.
Uses BEEP 54, EDITOR_COMMAND_LOOP 208, GAME_COLNUM 32a, HUNDREDS 46b, KBD_ENTRY_INDEX 185, KBDSTRB 120b, LEVELNUM 50, PUT_DIGIT 46a, SAVED_GAME_COLNUM 70b, TENS 46b, TO_DECIMAL3 47, UNITS 46b, and WAIT_FOR_KEY_PAGE_1 68.

Chapter 5

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.

5.1 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:

74 ⟨*level draw routine* 74⟩≡ (213) 78a ▷

```
      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 109.
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.

75a ⟨*tables* 7⟩+≡ (215) ◁32b 79a▷
```
    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 75–77 and 146.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES, used in chunks 75–77 and 146.
CURR_LEVEL_ROW_SPRITES_PTR_PAGES2, used in chunks 75–77.

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

75b ⟨*defines* 3⟩+≡ (215) ◁70b 77c▷
```
PTR1        EQU     $06     ; 2 bytes
PTR2        EQU     $08     ; 2 bytes
```
Defines:
PTR1, used in chunks 75–79, 84, 110, 146, 148, 150, 153, 157, 160, 163, 164, 174, and 179.
PTR2, used in chunks 75–77, 79–81, 110, 127, 135, 146, 148, 150, 153, 164, 174, and 179.

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, I think a "background" page that contains only non-moving elements.

These are very useful fragments, and appear all over the place in the code. This fragment sets PTR1 to the current active level's row sprite data.

75c ⟨*set active row pointer* PTR1 *for* Y 75c⟩≡ (77d 84 146 148 163 174)
```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
    STA     PTR1+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, and PTR1 75b.

This fragment sets PTR2 to the current background level's row sprite data.

75d ⟨*set background row pointer* PTR2 *for* Y 75d⟩≡ (77e 127 135 164 174)
```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
    STA     PTR2+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 75a, and PTR2 75b.

And this fragment sets PTR1 to the active row and PTR2 to the background row.

76a ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y 76a⟩≡ (76c 78a 109 146 148 150 153 174 179)
```
        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 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 75a, PTR1 75b, and PTR2 75b.

Occasionally the sets are reversed, although the effect is identical, so:

76b ⟨*set active and background row pointers* PTR2 *and* PTR1 *for* Y 76b⟩≡ (174)
```
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS,Y
        STA     PTR1
        STA     PTR2
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2,Y
        STA     PTR2+1
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES,Y
        STA     PTR1+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 75a, PTR1 75b, and PTR2 75b.

There's even a routine which does this, but it seems that there was a lot of inlining instead. Presumably the cycles were more important than the space.

76c ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y *routine* 76c⟩≡ (213)
```
        ORG     $884B
    GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA:
        SUBROUTINE

        ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
        RTS
```
Defines:
GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA, used in chunks 157 and 160.

Occasionally we want to get the next row (i.e. for Y+1). In that case we use these fragments.

76d ⟨*set active row pointer* PTR1 *for* Y+1 76d⟩≡ (148 164)
```
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
        STA     PTR1
        LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES+1,Y
        STA     PTR1+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, and PTR1 75b.

77a ⟨*set background row pointer* PTR2 *for* Y+1 77a⟩≡ (77f)
```
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS+1,Y
    STA     PTR2
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
    STA     PTR2+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 75a, and PTR2 75b.

77b ⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y+1 77b⟩≡ (164)
```
    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
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES2+1,Y
    STA     PTR2+1
```
Uses CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES2 75a, PTR1 75b, and PTR2 75b.

We also keep track of the player's sprite column and row.

77c ⟨*defines* 3⟩+≡ (215) ◁75b 78d▷
```
  PLAYER_COL      EQU     $00
  PLAYER_ROW      EQU     $01
```
Defines:
PLAYER_COL, used in chunks 77, 81c, 82c, 108, 125a, 127, 146, 148, 150, 153, 157, 160, 164, and 199.
PLAYER_ROW, used in chunks 77, 81c, 125a, 127, 146, 148, 150, 153, 157, 160, 163, 164, 199, and 200.

A common paradigm is to get the sprite where the player is, on the active or background page, so these fragments are repeated many times:

77d ⟨*get active sprite at player location* 77d⟩≡
```
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 75c⟩
    LDY     PLAYER_COL
    LDA     (PTR1),Y
```
Uses PLAYER_COL 77c, PLAYER_ROW 77c, and PTR1 75b.

77e ⟨*get background sprite at player location* 77e⟩≡ (146)
```
    LDY     PLAYER_ROW
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y
```
Uses PLAYER_COL 77c, PLAYER_ROW 77c, and PTR2 75b.

77f ⟨*get background sprite at player location on next row* 77f⟩≡ (146)
```
    LDY     PLAYER_ROW
    ⟨set background row pointer PTR2 for Y+1 77a⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y
```
Uses PLAYER_COL 77c, PLAYER_ROW 77c, and PTR2 75b.

78a ⟨*level draw routine* 74⟩+≡ (213) ◁74 78b▷
⟨*set active and background row pointers* PTR1 *and* PTR2 *for* Y 76a⟩

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

78b ⟨*level draw routine* 74⟩+≡ (213) ◁78a 78c▷

```
      LDY     27
      STY     GAME_COLNUM

  .col_loop:
```

Uses GAME_COLNUM 32a.

We load the sprite from the level data.

78c ⟨*level draw routine* 74⟩+≡ (213) ◁78b 78f▷

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

Uses PTR1 75b.

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.

These values are zeroed before the DRAW_LEVEL_PAGE2 routine is called.

78d ⟨*defines* 3⟩+≡ (215) ◁77c 78e▷

```
  GUARD_COUNT     EQU       $8D
  GOLD_COUNT      EQU       $93
  LADDER_COUNT    EQU       $A3
```

Defines:
GOLD_COUNT, used in chunks 80a, 108, 127, 174, 179, and 200.
GUARD_COUNT, used in chunks 80b, 108, 135, 172a, 174, and 199.
LADDER_COUNT, used in chunks 79b, 108, and 179.

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.

78e ⟨*defines* 3⟩+≡ (215) ◁78d 81b▷

```
  VERBATIM        EQU       $A2
```

Defines:
VERBATIM, used in chunks 78f, 82c, and 107a.

78f ⟨*level draw routine* 74⟩+≡ (213) ◁78c 79b▷

```
      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 78e.

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.

79a ⟨*tables* 7⟩+≡ (215) ◁75a 96b▷
```
      ORG     $0C00
  LADDER_LOCS_COL     DS      48
  LADDER_LOCS_ROW     DS      48
```
Defines:
LADDER_LOCS_COL, used in chunks 79b and 179.
LADDER_LOCS_ROW, used in chunks 79b and 179.

79b ⟨*level draw routine* 74⟩+≡ (213) ◁78f 79c▷
```
      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 78d, LADDER_LOCS_COL 79a, and LADDER_LOCS_ROW 79a.

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

79c ⟨*level draw routine* 74⟩+≡ (213) ◁79b 80a▷
```
  .remove_sprite:
      LDA     #0
      STA     (PTR1),Y
      STA     (PTR2),Y

  .draw_sprite1
      BEQ     .draw_sprite        ; Unconditional jump.
```
Uses PTR1 75b and PTR2 75b.

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

80a ⟨*level draw routine* 74⟩+≡ (213) ◁79c 80b▷

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

    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 78d.

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.

80b ⟨*level draw routine* 74⟩+≡ (213) ◁80a 81a▷

```
.check_for_guard:
    CMP     #$08
    BNE     .check_for_player

    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_0,X
    STA     GUARD_ANIM_STATES,X
    LDA     #$02
    STA     GUARD_X_ADJS,X
    STA     GUARD_Y_ADJS,X

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

Uses GAME_ROWNUM 32a, GUARD_ANIM_STATES 168, GUARD_COUNT 78d, GUARD_FLAGS_0 168, GUARD_LOCS_COL 168, GUARD_LOCS_ROW 168, GUARD_X_ADJS 168, GUARD_Y_ADJS 168, and PTR2 75b.

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

81a ⟨*level draw routine* 74⟩+≡ (213) ◁80b 81c▷

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

Next we check for sprite 9, the player.

81b ⟨*defines* 3⟩+≡ (215) ◁78e 85▷

```
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 81c, 125, 157, 160, and 164.
PLAYER_X_ADJ, used in chunks 81c, 125a, 127, 143, 150, and 153.
PLAYER_Y_ADJ, used in chunks 81c, 125a, 127, 143, 146, 148, 164, and 200.
Uses SPRITE_ANIM_SEQS 124c.

81c ⟨*level draw routine* 74⟩+≡ (213) ◁81a 82a▷

```
.check_for_player:
    CMP     #$09
    BNE     .check_for_t_thing

    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 81b, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_X_ADJ 81b, PLAYER_Y_ADJ 81b, PTR2 75b, and SPRITE_ANIM_SEQS 124c.

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

82a ⟨*level draw routine* 74⟩+≡ (213) ◁81c 82b▷
```
  .check_for_t_thing:
      CMP     #$05
      BNE     .draw_sprite
      LDA     #$01                    ; Brick sprite

      ; fallthrough to .draw_sprite
```

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

82b ⟨*level draw routine* 74⟩+≡ (213) ◁82a 82c▷
```
  .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!

82c ⟨*level draw routine* 74⟩+≡ (213) ◁82b 83▷
```
      LDA     VERBATIM
      BEQ     .copy_page2_to_page1

      LDA     PLAYER_COL
      BPL     .reveal_screen

      SEC                             ; Oops, no player! Return error.
      RTS
```
Uses PLAYER_COL 77c and VERBATIM 78e.

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

83 ⟨*level draw routine* 74⟩+≡ (213) ◁82c 84▷

```
.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!

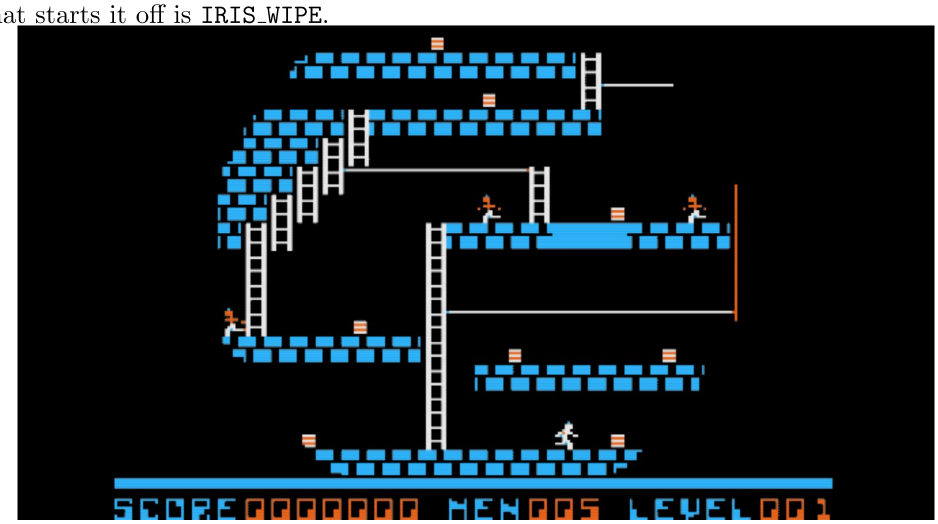
84 ⟨*level draw routine* 74⟩+≡ (213) ◁83

```
.reveal_screen
    JSR     IRIS_WIPE

    LDY     15
    STY     GAME_ROWNUM

.row_loop2:
    ⟨set active row pointer PTR1 for Y 75c⟩
    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 DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, IRIS_WIPE 86, and PTR1 75b.

5.2 Iris Wipe

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



85 ⟨*defines* 3⟩+≡ (215) ◁81b 87▷

```
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 86 and 97–99.
`WIPE_MODE`, used in chunks 86 and 197.

86 ⟨*iris wipe* 86⟩≡ (213)

```
    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 84.
Uses IRIS_WIPE_STEP 90, PUT_STATUS_LEVEL 51, PUT_STATUS_LIVES 51, WIPE_COUNTER 85, and WIPE_MODE 85.

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.

87 ⟨*defines 3*⟩+≡ (215) ◁85 89▷

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

Defines:
MATH_TMPH, used in chunks 88, 100, and 101a.
MATH_TMPL, used in chunks 88, 100, and 101a.

88 ⟨*routines* 4⟩+≡ (215) ◁33 213▷

```
      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 98 and 99.
Uses `MATH_TMPH` 87 and `MATH_TMPL` 87.

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

89 ⟨*defines* 3⟩+≡ (215) ◁87 104a▷

```
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 97, 101, and 185.
- `WIPE1`, used in chunks 97 and 100–102.
- `WIPE10D`, used in chunks 94, 95, 99b, and 102b.
- `WIPE10R`, used in chunks 94, 95, 99b, and 102b.
- `WIPE2`, used in chunks 91, 97d, 98a, 100, and 101a.
- `WIPE3H`, used in chunks 93, 98b, and 102a.
- `WIPE3L`, used in chunks 93, 98b, and 102a.
- `WIPE4H`, used in chunks 95, 98c, and 103a.
- `WIPE4L`, used in chunks 95, 98c, and 103a.
- `WIPE5H`, used in chunks 94, 98c, and 103b.
- `WIPE5L`, used in chunks 94, 98c, and 103b.
- `WIPE6H`, used in chunks 92b, 98d, and 102d.
- `WIPE6L`, used in chunks 92b, 98d, and 102d.
- `WIPE7D`, used in chunks 94, 95, 98e, and 102c.
- `WIPE7R`, used in chunks 94, 95, 98e, and 102c.
- `WIPE8D`, used in chunks 92b, 93, 99a, and 103c.
- `WIPE8R`, used in chunks 99a and 103c.
- `WIPE9D`, used in chunks 92b, 93, 99a, and 102f.
- `WIPE9R`, used in chunks 92b, 93, 99a, and 102f.

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

90 ⟨*iris wipe step* 90⟩≡ (213) 91 ▷

```
    ORG     $88D7
IRIS_WIPE_STEP:
    SUBROUTINE

⟨WIPE0 = WIPE_COUNTER 97b⟩
⟨WIPE1 = 0 97c⟩
⟨WIPE2 = 2 * WIPE0 97d⟩
⟨WIPE2 = 3 - WIPE2 98a⟩

; 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 98b⟩
⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 98c⟩
⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 98d⟩

; 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 98e⟩
⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 99a⟩
⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 99b⟩
```

Defines:
IRIS_WIPE_STEP, used in chunk 86.

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.

91 ⟨*iris wipe step* 90⟩+≡ (213) ◁90

```
.loop:

⟨iris wipe loop check 97a⟩

    JSR     DRAW_WIPE_STEP

    LDA     WIPE2+1
    BPL     .89a7

⟨WIPE2 += 4 * WIPE1 + 6 100⟩
    JMP     .8a14

.89a7:

⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 101a⟩
⟨Decrement WIPE0 101b⟩
⟨Increment WIPE3 102a⟩
⟨Decrement WIPE10 modulo 7 102b⟩
⟨Increment WIPE7 modulo 7 102c⟩
⟨Decrement WIPE6 102d⟩

.8a14:

⟨Increment WIPE1 102e⟩
⟨Increment WIPE9 modulo 7 102f⟩
⟨Decrement WIPE4 103a⟩
⟨Increment WIPE5 103b⟩
⟨Decrement WIPE8 modulo 7 103c⟩
    JMP     .loop
```

Uses `DRAW_WIPE_STEP` 92a and `WIPE2` 89.

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.

92a ⟨*draw wipe step* 92a⟩≡ (213)
```
      ORG     $8A69
  DRAW_WIPE_STEP:
      SUBROUTINE

  ⟨Draw wipe for south part 92b⟩
  ⟨Draw wipe for north part 93⟩
  ⟨Draw wipe for north2 part 94⟩
  ⟨Draw wipe for south2 part 95⟩
```
Defines:
DRAW_WIPE_STEP, used in chunks 91 and 97a.

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

92b ⟨*Draw wipe for south part* 92b⟩≡ (92a)
```
      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 96a, ROW_TO_ADDR_FOR_BOTH_PAGES 27, WIPE6H 89, WIPE6L 89, WIPE8D 89, WIPE9D 89, and WIPE9R 89.

93 ⟨*Draw wipe for north part 93*⟩≡ (92a)

```
.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` 96a, `ROW_TO_ADDR_FOR_BOTH_PAGES` 27, `WIPE3H` 89, `WIPE3L` 89, `WIPE8D` 89, `WIPE9D` 89, and `WIPE9R` 89.

94 ⟨*Draw wipe for north2 part 94*⟩≡ (92a)

```
.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 96a, ROW_TO_ADDR_FOR_BOTH_PAGES 27, WIPE10D 89, WIPE10R 89, WIPE5H 89, WIPE5L 89, WIPE7D 89, and WIPE7R 89.

95 ⟨*Draw wipe for south2 part* 95⟩≡ (92a)

```
.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` 96a, `ROW_TO_ADDR_FOR_BOTH_PAGES` 27, `WIPE10D` 89, `WIPE10R` 89, `WIPE4H` 89, `WIPE4L` 89, `WIPE7D` 89, and `WIPE7R` 89.

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.

96a ⟨*draw wipe block* 96a⟩≡ (213)

```
    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 92–95.
Uses ROW_ADDR 26b, ROW_ADDR2 26b, WIPE_BLOCK_CLOSE_MASK 96b, and WIPE_BLOCK_OPEN_MASK 96b.

96b ⟨*tables* 7⟩+≡ (215) ◁79a 107c▷

```
    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 96a.
WIPE_BLOCK_OPEN_MASK, used in chunk 96a.

97a ⟨*iris wipe loop check* 97a⟩≡ (91)
```
      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 92a, WIPE0 89, and WIPE1 89.

5.2.1 Initialization

97b ⟨WIPE0 = WIPE_COUNTER 97b⟩≡ (90)
```
      LDA     WIPE_COUNTER
      STA     WIPE0
      LDA     #$00
      STA     WIPE0+1         ; WIPE0 = WIPE_COUNTER
```
Uses WIPE0 89 and WIPE_COUNTER 85.

97c ⟨WIPE1 = 0 97c⟩≡ (90)
```
      ; fallthrough with A = 0
      STA     WIPE1
      STA     WIPE1+1         ; WIPE1 = 0
```
Uses WIPE1 89.

97d ⟨WIPE2 = 2 * WIPE0 97d⟩≡ (90)
```
      LDA     WIPE0
      ASL
      STA     WIPE2
      LDA     WIPE0+1
      ROL
      STA     WIPE2+1         ; WIPE2 = 2 * WIPE0
```
Uses WIPE0 89 and WIPE2 89.

98a ⟨WIPE2 = 3 - WIPE2 98a⟩≡ (90)
```
      LDA     #$03
      SEC
      SBC     WIPE2
      STA     WIPE2
      LDA     #$00
      SBC     WIPE2+1
      STA     WIPE2+1         ; WIPE2 = 3 - WIPE2
```
Uses WIPE2 89.

98b ⟨WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER 98b⟩≡ (90)
```
      LDA     WIPE_CENTER_Y
      SEC
      SBC     WIPE_COUNTER
      STA     WIPE3L
      LDA     #$00
      SBC     #$00
      STA     WIPE3H          ; WIPE3 = WIPE_CENTER_Y - WIPE_COUNTER
```
Uses WIPE3H 89, WIPE3L 89, and WIPE_COUNTER 85.

98c ⟨WIPE4 = WIPE5 = WIPE_CENTER_Y 98c⟩≡ (90)
```
      LDA     WIPE_CENTER_Y
      STA     WIPE4L
      STA     WIPE5L
      LDA     #$00
      STA     WIPE4H
      STA     WIPE5H          ; WIPE4 = WIPE5 = WIPE_CENTER_Y
```
Uses WIPE4H 89, WIPE4L 89, WIPE5H 89, and WIPE5L 89.

98d ⟨WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER 98d⟩≡ (90)
```
      LDA     WIPE_CENTER_Y
      CLC
      ADC     WIPE_COUNTER
      STA     WIPE6L
      LDA     #$00
      ADC     #$00
      STA     WIPE6H          ; WIPE6 = WIPE_CENTER_Y + WIPE_COUNTER
```
Uses WIPE6H 89, WIPE6L 89, and WIPE_COUNTER 85.

98e ⟨WIPE7 = (WIPE_CENTER_X - WIPE_COUNTER) / 7 98e⟩≡ (90)
```
      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 88, WIPE7D 89, WIPE7R 89, and WIPE_COUNTER 85.

99a ⟨WIPE8 = WIPE9 = WIPE_CENTER_X / 7 99a⟩≡ (90)

```
      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 88, WIPE8D 89, WIPE8R 89, WIPE9D 89, and WIPE9R 89.

99b ⟨WIPE10 = (WIPE_CENTER_X + WIPE_COUNTER) / 7 99b⟩≡ (90)

```
      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 88, WIPE10D 89, WIPE10R 89, and WIPE_COUNTER 85.

5.2.2 All that math stuff

100 ⟨WIPE2 += 4 * WIPE1 + 6 100⟩≡ (91)

```
      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 87, MATH_TMPL 87, WIPE1 89, and WIPE2 89.

101a ⟨WIPE2 += 4 * (WIPE1 - WIPE0) + 16 101a⟩≡ (91)

```
      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 87, MATH_TMPL 87, WIPE0 89, WIPE1 89, and WIPE2 89.

101b ⟨*Decrement* WIPE0 101b⟩≡ (91)

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

Uses WIPE0 89.

102a ⟨*Increment* WIPE3 102a⟩≡ (91)
```
      INC     WIPE3L
      BNE     .89f2
      INC     WIPE3H          ; WIPE3++
  .89f2
```
Uses WIPE3H 89 and WIPE3L 89.

102b ⟨*Decrement* WIPE10 *modulo 7* 102b⟩≡ (91)
```
      DEC     WIPE10R
      BPL     .89fc
      LDA     #$06
      STA     WIPE10R
      DEC     WIPE10D
  .89fc
```
Uses WIPE10D 89 and WIPE10R 89.

102c ⟨*Increment* WIPE7 *modulo 7* 102c⟩≡ (91)
```
      INC     WIPE7R
      LDA     WIPE7R
      CMP     #$07
      BNE     .8a0a
      LDA     #$00
      STA     WIPE7R
      INC     WIPE7D
  .8a0a
```
Uses WIPE7D 89 and WIPE7R 89.

102d ⟨*Decrement* WIPE6 102d⟩≡ (91)
```
      DEC     WIPE6L
      LDA     WIPE6L
      CMP     #$FF
      BNE     .8a14
      DEC     WIPE6H
```
Uses WIPE6H 89 and WIPE6L 89.

102e ⟨*Increment* WIPE1 102e⟩≡ (91)
```
      INC     WIPE1
      BNE     .8a1a
      INC     WIPE1+1         ; WIPE1++
  .8a1a
```
Uses WIPE1 89.

102f ⟨*Increment* WIPE9 *modulo 7* 102f⟩≡ (91)
```
      INC     WIPE9R
      LDA     WIPE9R
      CMP     #$07
      BNE     .8a28
      LDA     #$00
      STA     WIPE9R
      INC     WIPE9D
  .8a28
```
Uses WIPE9D 89 and WIPE9R 89.

103a ⟨*Decrement* `WIPE4` 103a⟩≡ (91)
```
    DEC     WIPE4L
    LDA     WIPE4L
    CMP     #$FF
    BNE     .8a32
    DEC     WIPE4H
.8a32
```
Uses `WIPE4H` 89 and `WIPE4L` 89.

103b ⟨*Increment* `WIPE5` 103b⟩≡ (91)
```
    INC     WIPE5L
    BNE     .8a38
    INC     WIPE5H          ; WIPE5++
.8a38
```
Uses `WIPE5H` 89 and `WIPE5L` 89.

103c ⟨*Decrement* `WIPE8` *modulo 7* 103c⟩≡ (91)
```
    DEC     WIPE8R
    BPL     .8a42
    LDA     #$06
    STA     WIPE8R
    DEC     WIPE8D
.8a42
```
Uses `WIPE8D` 89 and `WIPE8R` 89.

5.3 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.

5.3.1 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 256-byte DISK_BUFFER buffer. This buffer is the same as the DOS read/write buffer, so that level data can be loaded directly from disk. Levels on disk are stored starting at track 3 sector 0, with levels being stored in consecutive sectors, 16 per track.

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

One additional feature is that you can start the routine with A being 1 to read a level, 2 to write a level, and 4 to format the entire disk. Writing and formatting is used by the level editor.

104a ⟨*defines* 3⟩+≡ (215) ◁89 106d▷

```
PREGAME_MODE            EQU     $A7
DISK_BUFFER   EQU     $0D00       ; 256 bytes
RWTS_ADDR               EQU     $24          ; 2 bytes
DISK_LEVEL_LOC          EQU     $96
```

Defines:
PREGAME_MODE, used in chunks 105, 117, 122–24, 129, 197, 199, 200, and 208.

104b ⟨*jump to RWTS indirectly* 104b⟩≡ (213)

```
    ORG     $0023
JMP_RWTS:
    SUBROUTINE

    JMP     $0000       ; Gets loaded with RWTS address later
```

Defines:
JMP_RWTS, used in chunk 105.

105 ⟨*load compressed level data* 105⟩≡ (213)

```
    ORG     $630E
LOAD_COMPRESSED_LEVEL_DATA:
    SUBROUTINE
    ; Enter routine with A set to command: 1 = read, 2 = write, 4 = format

    STA     IOB_COMMAND_CODE
    LDA     PREGAME_MODE
    LSR
    BEQ     .copy_level_data        ; If PREGAME_MODE is 0 or 1, copy level data

    ; Read/write/format level on disk
    LDA     DISK_LEVEL_LOC
    LSR
    LSR
    LSR
    LSR
    CLC
    ADC     3
    STA     IOB_TRACK_NUMBER            ; track 3 + (DISK_LEVEL_LOC >> 4)
    LDA     DISK_LEVEL_LOC
    AND     #$0F
    STA     IOB_SECTOR_NUMBER           ; sector DISK_LEVEL_LOC & 0x0F
    LDA     #$<DISK_BUFFER
    STA     IOB_READ_WRITE_BUFFER_PTR
    LDA     #$>DISK_BUFFER
    STA     IOB_READ_WRITE_BUFFER_PTR+1 ; IOB_READ_WRITE_BUFFER_PTR = 0D00
    LDA     #$00
    STA     IOB_VOLUME_NUMBER_EXPECTED  ; any volume

ACCESS_DISK_OR_RESET_GAME:
    LDY     #<DOS_IOB
    LDA     #>DOS_IOB
    JSR     JMP_RWTS
    BCC     .end
    JMP     RESET_GAME      ; On error

.end:
    RTS

.copy_level_data:
⟨Copy level data 106a⟩
```

Uses DOS_IOB 181, IOB_COMMAND_CODE 181, IOB_READ_WRITE_BUFFER_PTR 181, IOB_SECTOR_NUMBER 181, IOB_TRACK_NUMBER 181, IOB_VOLUME_NUMBER_EXPECTED 181, JMP_RWTS 104b, and PREGAME_MODE 104a.

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.

Since the game is supposed to come with 150 levels, there is not enough room to store all of it, so the rest of the level data must be on disk. Only the first few levels are in memory.

106a ⟨*Copy level data* 106a⟩≡ (105)

```
⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 106b⟩
⟨Copy data from ROW_ADDR into DISK_BUFFER 106c⟩
```

106b ⟨ROW_ADDR = $9E00 + LEVELNUM * $0100 106b⟩≡ (106a)

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

Uses LEVELNUM 50 and ROW_ADDR 26b.

106c ⟨*Copy data from* ROW_ADDR *into* DISK_BUFFER 106c⟩≡ (106a)

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

Uses ROW_ADDR 26b.

5.3.2 Uncompressing and displaying the level

Loading the level also sets the player ALIVE flag to 1 (alive). Throughout the code, LSR ALIVE simply sets the flag to 0 (dead).

106d ⟨*defines* 3⟩+≡ (215) ◁104a 107b▷

```
ALIVE       EQU     $9A
```

Defines:
ALIVE, used in chunks 41, 107a, 130a, 131c, 138, 172a, 174, and 200.

107a ⟨*load level* 107a⟩≡ (213)

```
      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 108⟩

      LDA     #1
      STA     ALIVE       ; Set player live
      JSR     LOAD_COMPRESSED_LEVEL_DATA

      ⟨uncompress level data 109⟩
```

Defines:
`LOAD_LEVEL`, used in chunks 111b and 199.
Uses `ALIVE` 106d and `VERBATIM` 78e.

107b ⟨*defines* 3⟩+≡ (215) ◁106d 112a▷

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

107c ⟨*tables* 7⟩+≡ (215) ◁96b 114a▷

```
      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.

108 ⟨*Initialize level counts* 108⟩≡ (107a)

```
    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     GUARD_FLAGS_5,X
    DEX
    BPL     .loop2
```

Uses GAME_ROWNUM 32a, GOLD_COUNT 78d, GUARD_COUNT 78d, LADDER_COUNT 78d, and PLAYER_COL 77c.

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!

109 ⟨*uncompress level data* 109⟩≡ (107a)

```
  .row_loop:
      ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
      ⟨uncompress row data 110⟩
      ⟨next compressed row for row_loop 111a⟩

      JSR     DRAW_LEVEL_PAGE2
      BCC     .end                  ; No error

  ⟨handle no player sprite in level 111b⟩

  .end:
      RTS

  .reset_game:
      JMP     RESET_GAME
```

Uses DRAW_LEVEL_PAGE2 74.

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

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).

110 ⟨*uncompress row data* 110⟩≡ (109)

```
    LDA     #0
    STA     GAME_COLNUM

.col_loop:
    LDA     TMP                         ; odd/even counter
    LSR
    LDY     LEVEL_DATA_INDEX
    LDA     DISK_BUFFER,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 75b, and PTR2 75b.

111a ⟨*next compressed row for* `row_loop` 111a⟩≡ (109)

```
      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 DISK_LEVEL_LOC is zero, we're going to jump to RESET_GAME. Otherwise, we set DISK_LEVEL_LOC to zero, increment \$97, set X to 0xFF, and retry LOAD_LEVEL from the very beginning.

111b ⟨*handle no player sprite in level* 111b⟩≡ (109)

```
      LDA     DISK_LEVEL_LOC
      BEQ     .reset_game

      LDX     0
      STX     DISK_LEVEL_LOC
      INC     GUARD_PATTERN_OFFSET
      DEX
      JMP     LOAD_LEVEL
```

Uses GUARD_PATTERN_OFFSET 196c and LOAD_LEVEL 107a.

Chapter 6

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 `HI_SCORE_DATA`.

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.

112a ⟨*defines* 3⟩+≡ (215) ◁107b 115c▷

```
HI_SCORE_DATA    EQU     $1F00      ; 256 bytes
```

Defines:
`HI_SCORE_DATA`, used in chunks 114, 115a, 183, 185, and 195.

112b ⟨*construct and display high score screen* 112b⟩≡ (213)

```
    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 113a⟩
    ⟨draw high score rows 113b⟩
    ⟨show high score page 116⟩
```

Defines:
`HI_SCORE_SCREEN`, used in chunks 124a, 135, and 185.
Uses `CLEAR_HGR2` 4, `DRAW_PAGE` 43, `GAME_COLNUM` 32a, and `GAME_ROWNUM` 32a.

113a ⟨*draw high score table header* 113a⟩≡ (112b)

```
    ; "    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 45 and SCORE 48b.

113b ⟨*draw high score rows* 113b⟩≡ (112b)

```
    LDA     #$01
    STA     $55             ; Used for row number
.loop:
    ⟨draw high score row number 113c⟩
    ⟨draw high score initials 114b⟩
    ⟨draw high score level 114c⟩
    ⟨draw high score 115a⟩
    ⟨next high score row 115b⟩
```

113c ⟨*draw high score row number* 113c⟩≡ (113b)

```
    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 44a, PUT_DIGIT 46a, and PUT_STRING 45.

114a ⟨*tables* 7⟩+≡ (215) ◁107c 124c▷
```
      ORG     $79A2
  HI_SCORE_TABLE_OFFSETS:
      HEX     00 08 10 18 20 28 30 38 40 48
```
Defines:
HI_SCORE_TABLE_OFFSETS, used in chunks 114b and 185.

114b ⟨*draw high score initials* 114b⟩≡ (113b)
```
      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 112a, HI_SCORE_TABLE_OFFSETS 114a, PUT_CHAR 44a, and PUT_STRING 45.

114c ⟨*draw high score level* 114c⟩≡ (113b)
```
      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 112a, HUNDREDS 46b, PUT_DIGIT 46a, PUT_STRING 45, TENS 46b, TO_DECIMAL3 47, and UNITS 46b.

115a ⟨*draw high score* 115a⟩≡ (113b)

```
      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` 48a, `HI_SCORE_DATA` 112a, `PUT_DIGIT` 46a, `TENS` 46b, and `UNITS` 46b.

115b ⟨*next high score row* 115b⟩≡ (113b)

```
  .next_high_score_row:
      JSR     NEWLINE
      INC     $55
      LDA     $55
      CMP     #11
      BCS     .end
      JMP     .loop
```

Uses `NEWLINE` 44a.

115c ⟨*defines* 3⟩+≡ (215) ◁112a 119b▷

```
  TXTPAGE2                EQU     $C055
```

Defines:
`TXTPAGE2`, used in chunks 70a and 116.

116 ⟨*show high score page* 116⟩≡ (112b)

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

Uses DRAW_PAGE 43 and TXTPAGE2 115c.

Chapter 7

Game play

7.1 Splash screen

117 ⟨*splash screen* 117⟩≡ (213)

```
     ORG     $6008
RESET_GAME:
     SUBROUTINE

     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     DISK_LEVEL_LOC   ; GAME_ROWNUM = DISK_LEVEL_LOC = PREGAME_MODE = 0
     LDA     #$20
     STA     HGR_PAGE
     STA     DRAW_PAGE        ; HGR_PAGE = DRAW_PAGE = 0x20

     ⟨splash screen loop 118⟩

     STA     TXTPAGE1
     STA     HIRES
     STA     MIXCLR
     STA     TXTCLR
     JMP     .long_delay_attract_mode
```

Uses `CLEAR_HGR1` 4, `DRAW_PAGE` 43, `GAME_ROWNUM` 32a, `HGR_PAGE` 26b, `HIRES` 119b, `MIXCLR` 119b, `PREGAME_MODE` 104a, `TXTCLR` 119b, and `TXTPAGE1` 119b.

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.

118 ⟨*splash screen loop* 118⟩≡ (117)

```
.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.

7.2 Startup code

The startup code is run immediately after relocating memory blocks.

119a ⟨*startup code* 119a⟩≡ (213)
⟨*set startup softswitches* 119c⟩
⟨*set stack size* 119d⟩
⟨*maybe set carry but not really* 120a⟩
⟨*ready yourself* 120c⟩

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.

119b ⟨*defines* 3⟩+≡ (215) ◁115c 120b▷

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

Defines:
`HIRES`, used in chunks 117 and 119c.
`MIXCLR`, used in chunks 117 and 119c.
`ROMIN_RDROM_WRRAM2`, used in chunk 119c.
`TXTCLR`, used in chunks 117 and 119c.
`TXTPAGE1`, used in chunks 70a, 117, 119c, 135, 197, and 208.

119c ⟨*set startup softswitches* 119c⟩≡ (119a)

```
    ORG     $5F7D

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

Uses `HIRES` 119b, `MIXCLR` 119b, `ROMIN_RDROM_WRRAM2` 119b, `TXTCLR` 119b, and `TXTPAGE1` 119b.

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.

119d ⟨*set stack size* 119d⟩≡ (119a)

```
    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.

120a ⟨*maybe set carry but not really* 120a⟩≡ (119a)

```
      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.

120b ⟨*defines* 3⟩+≡ (215) ◁119b 121b▷

```
  KBDSTRB     EQU     $C010
```

Defines:
KBDSTRB, used in chunks 69–71, 120c, 122b, 129, 131a, 135, 185, and 205.

120c ⟨*ready yourself* 120c⟩≡ (119a)

```
      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     JOYSTICK_MODE            ; Fake keypress 0x4A (J)
      JMP     CHECK_FOR_BUTTON_DOWN
```

Uses CHECK_FOR_BUTTON_DOWN 121a, GAME_ROWNUM 32a, and KBDSTRB 120b.

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.

121a ⟨*check for button down* 121a⟩≡ (213)
```
      ORG     $6199

  .check_input_mode:
      LDA     INPUT_MODE

  CHECK_FOR_BUTTON_DOWN:
      CMP     KEYBOARD_MODE
      BEQ     .no_button_pressed  ; If keyboard mode, skip check button presses.
      LDA     BUTN1
      BMI     .button_pressed
      LDA     BUTN0
      BMI     .button_pressed

      ; fall through to .no_button_pressed
```
Defines:
CHECK_FOR_BUTTON_DOWN, used in chunk 120c.
Uses BUTN0 64, BUTN1 64, and INPUT_MODE 64.

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.

121b ⟨*defines* 3⟩+≡ (215) ◁120b 126▷
```
  KBD         EQU     $C000
```
Defines:
KBD, used in chunks 67, 68, 121c, 123a, 129, 131a, 135, 138, and 205.

121c ⟨*no button pressed* 121c⟩≡ (213)
```
      ORG     $61A9

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

      ; fall through to .no_button_or_key_timeout
```
Uses GAME_ROWNUM 32a and KBD 121b.

If one of the joystick buttons was pressed:

122a ⟨*button pressed at startup* 122a⟩≡ (213)
```
    ORG     $6201

.button_pressed:
    LDX     #$00
    STX     DISK_LEVEL_LOC      ; DISK_LEVEL_LOC = 0
    INX
    STX     LEVELNUM            ; LEVELNUM = 1
    STX     $9D
    LDA     #$02
    STX     PREGAME_MODE
    JMP     .play_game
```
Uses LEVELNUM 50 and PREGAME_MODE 104a.

And if one of the keys was pressed:

122b ⟨*key pressed at startup* 122b⟩≡ (213)
```
    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 120b.

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

122c ⟨*ctrl-e pressed* 122c⟩≡ (213)
```
    ORG     $6211

.ctrl_e_pressed:
    JMP     START_LEVEL_EDITOR
```
Uses START_LEVEL_EDITOR 208.

122d ⟨*return pressed* 122d⟩≡ (213)
```
    ORG     $61E4

.return_pressed:
    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read hi score table

    ; fallthrough to .pregame_mode_2
```
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183.

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

123a ⟨*timed out waiting for button or keypress* 123a⟩≡ (213)

```
    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     ENABLE_SOUND
    STX     .restore_enable_sound+1    ; Save previous value of DNABLE_SOUND
    STA     ENABLE_SOUND
    JMP     .init_game_data

.restore_enable_sound:
    LDA     #$00            ; Fixed up above
    STA     ENABLE_SOUND
    LDA     KBD
    LDX     $AC
    BEQ     .key_pressed
    JMP     .long_delay_attract_mode
```

Uses ENABLE_SOUND 57b, KBD 121b, LEVELNUM 50, and PREGAME_MODE 104a.

123b ⟨*check game mode* 123b⟩≡ (213)

```
    ORG     $61DE

.check_game_mode:
    CMP     #$01
    BNE     .reset_game
    BEQ     .pregame_mode_2        ; Unconditional jump
```

123c ⟨*reset game if not mode 1* 123c⟩≡ (213)

```
    ORG     $61F3

.reset_game:
    JMP     RESET_GAME
```

Pregame mode 2 displays the high score screen.

124a ⟨*display high score screen* 124a⟩≡ (213)

```
      ORG     $61E9

  .pregame_mode_2:
      JSR     HI_SCORE_SCREEN
      LDA     #$02
      STA     PREGAME_MODE            ; PREGAME_MODE = 2
      JMP     .long_delay_attract_mode
```

Uses HI_SCORE_SCREEN 112b and PREGAME_MODE 104a.

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

124b ⟨*long delay attract mode* 124b⟩≡ (213)

```
      ORG     $618E

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

      ; fall through to .check_input_mode
```

Uses GAME_ROWNUM 32a.

7.3 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.

124c ⟨*tables* 7⟩+≡ (215) ◁114a 128b▷

```
      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 81 and 125a.

125a ⟨*get player sprite and coord data* 125a⟩≡ (213)
```
      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 chunks 41, 146, 148, 150, 153, 157, 160, and 164.
Uses GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR 31c, GET_SCREEN_ROW_OFFSET_IN_X_FOR 31a, PLAYER_ANIM_STATE 81b, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_X_ADJ 81b, PLAYER_Y_ADJ 81b, SPRITE_ANIM_SEQS 124c, 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.

125b ⟨*increment player animation state* 125b⟩≡ (213)
```
      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, used in chunks 146, 150, and 153.
Uses PLAYER_ANIM_STATE 81b.

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 background 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 background sprite data, increment the score by 250, erase the gold sprite on the background screen at the player location, and then load up data into the sound area.

There is also a flag `DIDNT_PICK_UP_GOLD` which tells us whether the player did not pick up gold during this move. This flag is set to `1` just before handling the player move.

126 ⟨*defines* 3⟩+≡ (215) ◁121b 128a▷

```
DIDNT_PICK_UP_GOLD      EQU     $94
```

Defines:
`DIDNT_PICK_UP_GOLD`, used in chunks 41, 127, and 164.

127 ⟨*check for gold picked up by player* 127⟩≡ (213)

```
    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
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y

    CMP     #$07                ; Gold
    BNE     .end

    LSR     DIDNT_PICK_UP_GOLD  ; picked up gold
    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 background screen

    LDY     PLAYER_ROW
    LDX     PLAYER_COL
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #$07                            ; Gold
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS    ; Erase 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
```

Defines:
`CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER`, used in chunks 143, 146, 150, 153, and 164.

Uses `ADD_AND_UPDATE_SCORE` 49, `DIDNT_PICK_UP_GOLD` 126, `DRAW_SPRITE_PAGE2` 33, `ERASE_SPRITE_AT_PIXEL_COORDS` 36, `GAME_COLNUM` 32a, `GAME_ROWNUM` 32a, `GET_SCREEN_COORDS_FOR` 29a, `GOLD_COUNT` 78d, `LOAD_SOUND_DATA` 56, `PLAYER_COL` 77c, PLAYER_ROW 77c, PLAYER_X_ADJ 81b, PLAYER_Y_ADJ 81b, PTR2 75b, and SCORE 48b.

128a ⟨*defines* 3⟩+≡ (215) ◁126 134b▷

```
KEY_COMMAND      EQU     $9E
```

Defines:
KEY_COMMAND, used in chunks 129, 138, 140, 157, 160, 164, and 199.

128b ⟨*tables* 7⟩+≡ (215) ◁124c 137▷

```
    ORG     $6B59
VALID_CTRL_KEYS:
    ; ctrl-
    ; ^ @ [ R A S J K H U X Y M
    ; Esc:         ctrl-[
    ; Down arrow:  ctrl-J
    ; Up arrow:    ctrl-K
    ; Right arrow: ctrl-U
    ; Left arrow:  ctrl-H
    ; Return:      ctrl-M
    HEX     9E 80 9B 92 81 93 8A 8B 88 95 98 99 8D 00

    ORG     $6B67
CTRL_KEY_HANDLERS:
    ; These get pushed onto the stack, then an RTS is issued.
    ; Remember that the 6502's return stack contains the address
    ; to return to *minus 1*, so these values are actually one less
    ; than the function to jump to.
    WORD    CTRL_CARET_HANDLER-1
    WORD    CTRL_AT_HANDLER-1
    WORD    ESC_HANDLER-1
    WORD    CTRL_R_HANDLER-1
    WORD    CTRL_A_HANDLER-1
    WORD    CTRL_S_HANDLER-1
    WORD    DOWN_ARROW_HANDLER-1
    WORD    UP_ARROW_HANDLER-1
    WORD    LEFT_ARROW_HANDLER-1
    WORD    RIGHT_ARROW_HANDLER-1
    WORD    CTRL_X_HANDLER-1
    WORD    CTRL_Y_HANDLER-1
    WORD    RETURN_HANDLER-1
```

Defines:
CTRL_KEY_HANDLERS, used in chunk 129.
VALID_CTRL_KEYS, used in chunk 129.
Uses CTRL_A_HANDLER 131c, CTRL_AT_HANDLER 130b, CTRL_CARET_HANDLER 130a, CTRL_R_HANDLER 131c, CTRL_S_HANDLER 132a, CTRL_X_HANDLER 134a, CTRL_Y_HANDLER 134a, DOWN_ARROW_HANDLER 132b, ESC_HANDLER 131b, LEFT_ARROW_HANDLER 133, RETURN_HANDLER 135, RIGHT_ARROW_HANDLER 133, and UP_ARROW_HANDLER 132b.

129 ⟨*check for input* 129⟩≡ (213)

```
    ORG     $6A12
CHECK_FOR_INPUT:
    SUBROUTINE

    LDA     PREGAME_MODE
    CMP     #$01
    BEQ     CHECK_FOR_MODE_1_INPUT

    LDX     KBD
    STX     KBDSTRB
    STX     SPRITE_NUM
    BMI     .key_pressed

    LDA     INPUT_MODE
    CMP     KEYBOARD_MODE
    BEQ     .end                    ; If keyboard mode, end.

.check_buttons_:
    JMP     CHECK_BUTTONS

.key_pressed:
    CPX     #$A0
    BCS     .non_ctrl_key_pressed
    ; ctrl key pressed
    STX     SPRITE_NUM
    LDY     #$FF

.loop:
    INY
    LDA     VALID_CTRL_KEYS,Y
    BEQ     .non_ctrl_key_pressed

    CMP     SPRITE_NUM
    BNE     .loop

    TYA
    ASL
    TAY
    LDA     CTRL_KEY_HANDLERS+1,Y
    PHA
    LDA     CTRL_KEY_HANDLERS,Y
    PHA
    RTS                             ; JSR to CTRL_KEY_HANDLERS[Y], then return.

.non_ctrl_key_pressed:
    LDA     INPUT_MODE
    CMP     JOYSTICK_MODE
    BEQ     .check_buttons_         ; If joystick mode, check buttons.
```

```
        LDX     SPRITE_NUM
        STX     KEY_COMMAND
        STX     $9F

    .end:
        RTS
```

Defines:
CHECK_FOR_INPUT, used in chunks 130–35 and 164.
Uses CHECK_BUTTONS 140, CHECK_FOR_MODE_1_INPUT 138, CTRL_KEY_HANDLERS 128b, INPUT_MODE 64, KBD 121b, KBDSTRB 120b, KEY_COMMAND 128a, PREGAME_MODE 104a, SPRITE_NUM 23c, and VALID_CTRL_KEYS 128b.

Hitting `ctrl-^` increments both lives and level number, but also kills the player.

130a ⟨*ctrl handlers* 130a⟩≡ (213) 130b ▷

```
        ORG     $6A56
    CTRL_CARET_HANDLER:
        SUBROUTINE

        INC     LIVES
        INC     LEVELNUM
        INC     DISK_LEVEL_LOC
        LSR     ALIVE       ; set player dead
        LSR     $9D
        RTS
```

Defines:
CTRL_CARET_HANDLER, used in chunk 128b.
Uses ALIVE 106d, LEVELNUM 50, and LIVES 50.

Hitting `ctrl-@` increments lives.

130b ⟨*ctrl handlers* 130a⟩+≡ (213) ◁ 130a 131a ▷

```
        ORG     $6A61
    CTRL_AT_HANDLER:
        SUBROUTINE

        INC     LIVES
        BNE     .have_lives
        DEC     LIVES           ; LIVES = 255
    .have_lives:
        JSR     PUT_STATUS_LIVES
        LSR     $9D
        JMP     CHECK_FOR_INPUT
```

Defines:
CTRL_AT_HANDLER, used in chunk 128b.
Uses CHECK_FOR_INPUT 129, LIVES 50, and PUT_STATUS_LIVES 51.

Hitting ESC pauses the game, and ESC then unpauses the game.

131a ⟨*ctrl handlers* 130a⟩+≡ (213) ◁130b 131b▷
```
      ORG     $86A8
  WAIT_KEY:
      SUBROUTINE

      LDA     KBD
      BPL     WAIT_KEY
      STA     KBDSTRB
      RTS
```
Defines:
  WAIT_KEY, used in chunk 131b.
Uses KBD 121b and KBDSTRB 120b.

131b ⟨*ctrl handlers* 130a⟩+≡ (213) ◁131a 131c▷
```
      ORG     $6A76
  ESC_HANDLER:
      SUBROUTINE

      JSR     WAIT_KEY
      CMP     #$9B            ; key pressed is ESC?
      BNE     ESC_HANDLER
      JMP     CHECK_FOR_INPUT
```
Defines:
  ESC_HANDLER, used in chunk 128b.
Uses CHECK_FOR_INPUT 129 and WAIT_KEY 131a.

Hitting ctrl-R sets lives to 1 and sets player to dead, ending the game. Hitting ctrl-A shifts ALIVE, which just kills you.

131c ⟨*ctrl handlers* 130a⟩+≡ (213) ◁131b 132a▷
```
      ORG     $6A80
  CTRL_R_HANDLER:
      SUBROUTINE

      LDA     #$01
      STA     LIVES

  CTRL_A_HANDLER:
      LSR     ALIVE           ; Set player to dead
      RTS
```
Defines:
  CTRL_A_HANDLER, used in chunk 128b.
  CTRL_R_HANDLER, used in chunk 128b.
Uses ALIVE 106d and LIVES 50.

Hitting `ctrl-S` toggles sound.

132a ⟨*ctrl handlers* 130a⟩+≡ (213) ◁131c 132b▷

```
      ORG     $6A87
  CTRL_S_HANDLER:
      SUBROUTINE

      LDA     ENABLE_SOUND
      EOR     #$FF
      STA     ENABLE_SOUND
      JMP     CHECK_FOR_INPUT
```

Defines:
`CTRL_S_HANDLER`, used in chunk 128b.
Uses `CHECK_FOR_INPUT` 129 and `ENABLE_SOUND` 57b.

Hitting `ctrl-J` switches to joystick controls, and hitting `ctrl-K` switches to keyboard controls.

132b ⟨*ctrl handlers* 130a⟩+≡ (213) ◁132a 133▷

```
      ORG     $6A90
  DOWN_ARROW_HANDLER:
      SUBROUTINE

      LDA     JOYSTICK_MODE
      STA     INPUT_MODE
      JMP     CHECK_FOR_INPUT

      ORG     $6A97
  UP_ARROW_HANDLER:
      SUBROUTINE

      LDA     KEYBOARD_MODE
      STA     INPUT_MODE
      JMP     CHECK_FOR_INPUT
```

Defines:
`DOWN_ARROW_HANDLER`, used in chunk 128b.
`UP_ARROW_HANDLER`, used in chunk 128b.
Uses `CHECK_FOR_INPUT` 129 and `INPUT_MODE` 64.

Hitting the left arrow and right arrow decreases and increases the FRAME_PERIOD, effectively speed up and slowing down the game.

133 ⟨*ctrl handlers* 130a⟩+≡ (213) ◁132b 134a▷

```
      ORG     $6ABC
  RIGHT_ARROW_HANDLER:
      SUBROUTINE

      LDA     FRAME_PERIOD
      BEQ     .end
      DEC     FRAME_PERIOD

  .end
      JMP     CHECK_FOR_INPUT

      ORG     $6AC5
  LEFT_ARROW_HANDLER:
      SUBROUTINE

      LDA     FRAME_PERIOD
      CMP     #$0F
      BEQ     .end
      INC     FRAME_PERIOD

  .end
      JMP     CHECK_FOR_INPUT
```

Defines:
  LEFT_ARROW_HANDLER, used in chunk 128b.
  RIGHT_ARROW_HANDLER, used in chunk 128b.
Uses CHECK_FOR_INPUT 129 and FRAME_PERIOD 59b.

Hitting `ctrl-X` swaps `$6B81` and `$6B82`. Hitting `ctrl-Y` swaps `$6B83` and `$6B84`.

134a ⟨*ctrl handlers* 130a⟩+≡ (213) ◁133

```
      ORG     $6A9E
  CTRL_X_HANDLER:
      SUBROUTINE

      LDA     $6B81
      LDX     $6B82
      STA     $6B82
      STX     $6B81
      JMP     CHECK_FOR_INPUT

      ORG     $6AAD
  CTRL_Y_HANDLER:
      SUBROUTINE

      LDA     $6B83
      LDX     $6B84
      STA     $6B84
      STX     $6B85
      JMP     CHECK_FOR_INPUT
```

Defines:
`CTRL_X_HANDLER`, used in chunk 128b.
`CTRL_Y_HANDLER`, used in chunk 128b.
Uses `CHECK_FOR_INPUT` 129.

134b ⟨*defines* 3⟩+≡ (215) ◁128a 155▷

```
  SCRATCH_88      EQU     $88
  TABLE_0CA0      EQU     $0CA0       ; 31 bytes
  TABLE_0CC0      EQU     $0CC0       ; 31 bytes
```

Defines:
`SCRATCH_88`, used in chunk 135.

135 ⟨*return handler* 135⟩≡ (213)

```
    ORG     $77AC
RETURN_HANDLER:
    SUBROUTINE

    JSR     HI_SCORE_SCREEN     ; show high score screen
    LDX     #$FF
    LDY     #$FF
    LDA     #$04
    STA     SCRATCH_A1          ; loop 256x256x4 times

.loop:
    LDA     INPUT_MODE
    CMP     KEYBOARD_MODE                ; Keyboard mode
    BEQ     .check_keyboard

    LDA     BUTN1
    BMI     .button_pressed
    LDA     BUTN0
    BMI     .button_pressed

.check_keyboard:
    LDA     KBD
    BMI     .button_pressed

    DEX
    BNE     .loop
    DEY
    BNE     .loop
    DEC     SCRATCH_A1
    BNE     .loop

.button_pressed:
    STA     KBDSTRB
    STA     TXTPAGE1
    JSR     CLEAR_HGR2
    LDY     #$0F
    STY     GAME_ROWNUM

.loop2:
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     #$1B
    STY     GAME_COLNUM

.loop3:
    LDA     (PTR2),Y
    CMP     #$05
    BNE     .draw_sprite
    LDA     #$01                ; draw brick in place of T-thing
.draw_sprite:
```

```
    JSR     DRAW_SPRITE_PAGE2

    DEC     GAME_COLNUM
    LDY     GAME_COLNUM
    BPL     .loop3

    DEC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    BPL     .loop2

    LDX     #$1E
.loop4:
    STX     SCRATCH_88
    LDA     TABLE_0CE0,X
    BEQ     .next4

    LDY     TABLE_0CC0,X
    STY     GAME_ROWNUM
    LDY     TABLE_0CA0,X
    STY     GAME_COLNUM
    CMP     #$15
    BCC     .check_b

    LDA     #$00
    JSR     DRAW_SPRITE_PAGE2
    JMP     .next4

.check_b:
    CMP     #$0B
    BCC     .draw_sprite_56
    LDA     #$37
    JSR     DRAW_SPRITE_PAGE2
    JMP     .next4

.draw_sprite_56:
    LDA     #$38
    JSR     DRAW_SPRITE_PAGE2

.next4:
    LDX     SCRATCH_88
    DEX
    BPL     .next4

    LDX     GUARD_COUNT
    BEQ     .check_for_input

.loop5:
    STA     GUARD_FLAGS_5,X
    STX     SCRATCH_88
    BEQ     .next5
```

```
    LDY     GUARD_LOCS_COL
    STY     GAME_COLNUM
    LDY     GUARD_LOCS_ROW
    STY     GAME_ROWNUM
    CMP     #$14
    BCS     .next5

    CMP     #$0B
    BCC     .draw_sprite_58
    LDA     #$39                ; sprite 57
    BNE     .draw_sprite2       ; unconditional

.draw_sprite_58:
    LDA     #$3A

.draw_sprite2:
    JSR     DRAW_SPRITE_PAGE2

.next5:
    LDX     SCRATCH_88
    DEX
    BNE     .loop5

.check_for_input:
    JMP     CHECK_FOR_INPUT
```

Defines:
RETURN_HANDLER, used in chunk 128b.
Uses BUTN0 64, BUTN1 64, CHECK_FOR_INPUT 129, CLEAR_HGR2 4, DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, GUARD_COUNT 78d, GUARD_LOCS_COL 168, GUARD_LOCS_ROW 168, HI_SCORE_SCREEN 112b, INPUT_MODE 64, KBD 121b, KBDSTRB 120b, PTR2 75b, SCRATCH_88 134b, and TXTPAGE1 119b.

During pregame mode 1, we don't check for gameplay input. Instead, we use CHECK_FOR_MODE_1_INPUT for input. We first check if the user has pressed a key or hit a joystick button, and if so, we simulate killing the attract-mode player. However, if nothing was pressed, we check if the simulated player is pressing a key, and handle that.

137 ⟨*tables* 7⟩+≡ (215) ◁128b 156▷

```
    ORG     $6A0B
VALID_KEY_COMMANDS:
    HEX     C9      ; 'I'
    HEX     CA      ; 'J'
    HEX     CB      ; 'K'
    HEX     CC      ; 'L'
    HEX     CF      ; 'O'
    HEX     D5      ; 'U'
    HEX     A0      ; space
```

Defines:
VALID_KEY_COMMANDS, used in chunk 138.

138 ⟨*check for mode 1 input* 138⟩≡ (213)
```
    ORG     $69B8
CHECK_FOR_MODE_1_INPUT:
    SUBROUTINE

    LDA     KBD
    BMI     .key_pressed

    LDA     INPUT_MODE
    CMP     KEYBOARD_MODE
    BEQ     .nothing_pressed

    ; Check joystick buttons also
    LDA     BUTN1
    BMI     .key_pressed
    LDA     BUTN0
    BPL     .nothing_pressed

.key_pressed:
    ; Simulate killing the attact-mode player.
    LSR     $AC
    LSR     ALIVE
    LDA     #$01
    STA     LIVES
    RTS

.nothing_pressed:
    LDA     $AB
    BNE     .sim_keypress

    LDY     #$00
    LDA     ($A8),Y
    STA     $AA
    INY
    LDA     ($A8),Y
    STA     $AB
    CLC
    ADC     #$02
    STA     $A8
    LDA     $A9
    ADC     #$00
    STA     $A9

.sim_keypress:
    LDA     $AA
    AND     #$0F
    TAX
    LDA     VALID_KEY_COMMANDS,X
    STA     KEY_COMMAND
    LDA     $AA
```

```
        LSR
        LSR
        LSR
        LSR
        TAX
        LDA     VALID_KEY_COMMANDS,X
        STA     $9F
        DEC     $AB
        RTS
```

Defines:
CHECK_FOR_MODE_1_INPUT, used in chunk 129.
Uses ALIVE 106d, BUTN0 64, BUTN1 64, INPUT_MODE 64, KBD 121b, KEY_COMMAND 128a, LIVES 50, and VALID_KEY_COMMANDS 137.

140 ⟨*check buttons* 140⟩≡ (213)

```
    ORG     $6AD0
CHECK_BUTTONS:
    SUBROUTINE

    LDA     BUTN1
    BPL     .check_butn0
    LDA     #$D5
    BNE     .store_key_command    ; unconditional

.check_butn0:
    LDA     BUTN0
    BPL     .read_paddles
    LDA     #$CF

.store_key_command
    STA     KEY_COMMAND
    STA     $9F
    RTS

.read_paddles:
    JSR     READ_PADDLES
    LDY     PADDLE0_VALUE

    LDA     $6b82
    CMP     #$2E
    BEQ     .6afa

    CPY     $6b82
    BCS     .6b03
    LDA     #$CC
    BNE     .6b1e       ; unconditional

.6afa:
    CPY     $6b82
    BCC     .6b03
    LDA     #$CC
    BNE     .6b1e       ; unconditional

.6b03:
    LDA     $6b81
    CMP     #$2E
    BEQ     .6b13

    CPY     $6b81
    BCS     .6b1c
    LDA     #$CA
    BNE     .6b1e       ; unconditional

.6b13:
```

```
    CPY     $6b81
    BCC     .6b1c
    LDA     #$CA
    BNE     .6b1e       ; unconditional

.6b1c:
    LDA     #$C0

.6b1e:
    STA     $9F

    LDY     PADDLE1_VALUE

    LDA     $6b83
    CMP     #$2E
    BEQ     .6b32

    CPY     $6b83
    BCS     .6b3b
    LDA     #$C9
    BNE     .6b56       ; unconditional

.6b32:
    CPY     $6b84
    BCC     .6b3b
    LDA     #$C9
    BNE     .6b56       ; unconditional

.6b3b:
    LDA     $6b84
    CMP     #$2E
    BEQ     .6b4b

    CPY     $6b84
    BCS     .6b54
    LDA     #$CB
    BNE     .6b56       ; unconditional

.6b4b:
    CPY     $6b84
    BCC     .6b54
    LDA     #$CB
    BNE     .6b56       ; unconditional

.6b54:
    LDA     #$C0

.6b56:
    STA     KEY_COMMAND
    RTS
```

Defines:
  CHECK_BUTTONS, used in chunk 129.
Uses BUTN0 64, BUTN1 64, KEY_COMMAND 128a, PADDLE0_VALUE 62, PADDLE1_VALUE 62,
  and READ_PADDLES 63.

7.4 Player movement

Player movement is generally handled by functions which check whether the player can move in a given direction, and then either fail with carry set, or succeed, and the player is moved, with carry cleared.

Recall that the player is at the gross sprite location given by PLAYER_COL and PLAYER_ROW, but with a plus-or-minus adjustment given by a horizontal adjustment PLAYER_X_ADJ and a vertical adjustment PLAYER_Y_ADJ.

We will refer to the player as "exactly on" the sprite if the adjustment in the direction we're interested in is zero. Again, recall that the adjustment values are offset by 2, so an adjustment of zero is a value of 2, and the adjustment ranges from -2 to +2.

We will refer to the player as slightly above, below, left of, or right of, a sprite if the adjustment is not zero.

There are two routines which nudge the player towards an exact sprite row or column. Generally this is done when the player does something that has to take place on an exact row or column, such as climbing a ladder or traversing a rope, and serves to make the transition to an aligned row or column more smooth. Each time the player is nudged, we also check if the player landed on gold.

143 ⟨*try moving up* 143⟩≡ (213) 146 ▷

```
    ORG     $6C13
MOVE_PLAYER_TOWARDS_EXACT_COLUMN:
    SUBROUTINE

    LDA     PLAYER_X_ADJ
    CMP     #$02
    BCC     .player_slightly_left
    BEQ     .end

.player_slightly_right:
    DEC     PLAYER_X_ADJ        ; Nudge player left
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.player_slightly_left:
    INC     PLAYER_X_ADJ        ; Nudge player right
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.end:
    RTS

    ORG     $6C26
MOVE_PLAYER_TOWARDS_EXACT_ROW:
    SUBROUTINE

    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BCC     .player_slightly_above
```

```
    BEQ     .end

.player_slightly_below:
    DEC     PLAYER_Y_ADJ        ; Nudge player up
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.player_slightly_above:
    INC     PLAYER_Y_ADJ        ; Nudge player down
    JMP     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.end:
    RTS
```

Defines:
MOVE_PLAYER_TOWARDS_EXACT_COLUMN, used in chunks 146, 148, 157, and 160.
MOVE_PLAYER_TOWARDS_EXACT_ROW, used in chunks 150, 153, 157, and 160.
Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 127, PLAYER_X_ADJ 81b, and PLAYER_Y_ADJ 81b.

Now the logic for attempting to move up is:

- If the player location contains a ladder:
  - If the player is slightly below the sprite, then move the player up.
  - Otherwise, if the player is on row zero, the player cannot move up.
  - Otherwise, if the sprite on the row above is brick, stone, or T-thing, the player cannot move up.
  - Otherwise, the player can move up.
- Otherwise:
  - If the player is not slightly below the sprite, the player cannot move up.
  - Otherwise, if the sprite on the row below is not a ladder, the player cannot move up.
  - Otherwise, the player can move up.

The steps involved in actually moving the player up are:

- Erase the player sprite.
- Reduce any horizontal adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player vertically upwards by decrementing `PLAYER_Y_ADJ`.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for climbing, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Decrement `PLAYER_ROW`.
  - Put the player sprite on the active page at the new location.
  - Set the player's vertical adjustment to `+2`.
  - Update the player animation for climbing, and draw the player.

146 ⟨*try moving up* 143⟩+≡ (213) ◁143

```
    ORG     $66BD
TRY_MOVING_UP:
    SUBROUTINE

    ⟨get background sprite at player location 77e⟩
    CMP     #$03
    BEQ     .ladder_here

    LDY     PLAYER_Y_ADJ
    CPY     #$03
    BCC     .cannot_move        ; if PLAYER_Y_ADJ <= 2

    ; and if there's no ladder below, you can't move up.
    ⟨get background sprite at player location on next row 77f⟩
    CMP     #$03
    BEQ     .move_player_up

.cannot_move:
    SEC
    RTS

.ladder_here:
    LDY     PLAYER_Y_ADJ
    CPY     #$03
    BCS     .move_player_up             ; if PLAYER_Y_ADJ > 2

    ; If you're at the top, you can't move up even if there's a ladder.
    LDY     PLAYER_ROW
    BEQ     .cannot_move        ; if PLAYER_ROW == 0, set carry and return

    ; You can't move up if there's a brick, stone, or T-thing above.
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS-1,Y
    STA     PTR1
    LDA     CURR_LEVEL_ROW_SPRITES_PTR_PAGES-1,Y
    STA     PTR1+1
    LDY     PLAYER_COL
    LDA     (PTR1),Y                    ; Get the sprite on the row above.

    CMP     #$01
    BEQ     .cannot_move
    CMP     #$02
    BEQ     .cannot_move
    CMP     #$05
    BEQ     .cannot_move        ; If brick, stone, or T-thing, set carry and return

.move_player_up:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDY     PLAYER_ROW
```

```
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    JSR     MOVE_PLAYER_TOWARDS_EXACT_COLUMN
    DEC     PLAYER_Y_ADJ                ; Move player up
    BPL     TRY_MOVING_UP_check_for_gold

    ; PLAYER_Y_ADJ rolled over.

    ; Restore the sprite at the player's former location:
    ; If background page at player location is brick, put an empty at the
    ; (previous) player location on active page, otherwise copy the background
    ; sprite to the active page.
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .set_on_real_page
    LDA     #$00
.set_on_real_page:
    STA     (PTR1),Y

    DEC     PLAYER_ROW                  ; Move player up
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 75c⟩
    LDY     PLAYER_COL
    LDA     #$09
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$04
    STA     PLAYER_Y_ADJ        ; Set adjustment to +2
    BNE     TRY_MOVING_UP_inc_anim_state     ; unconditional

TRY_MOVING_UP_check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

TRY_MOVING_UP_inc_anim_state:
    LDA     #$10
    LDX     #$11
    JSR     INC_ANIM_STATE      ; player climbing on ladder
    JSR     DRAW_PLAYER
    CLC
    RTS
```

Defines:
TRY_MOVING_UP, used in chunk 164.

Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 127, CURR_LEVEL_ROW_SPRITES_PTR_OFFSETS 75a, CURR_LEVEL_ROW_SPRITES_PTR_PAGES 75a, DRAW_PLAYER 41, ERASE_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, INC_ANIM_STATE 125b, MOVE_PLAYER_TOWARDS_EXACT_COLUMN 143, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_Y_ADJ 81b, PTR1 75b, and PTR2 75b.

For attempting to move down, the logic is:

- If the player is slightly above the sprite, then move the player down.
- Otherwise, if the player is on row 15 or more, the player cannot move down.
- Otherwise, if the row below is stone or brick, the player cannot move down.
- Otherwise, the player can move down.

The steps involved in actually moving the player down are:

- Erase the player sprite.
- Reduce any horizontal adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player vertically downwards by incrementing `PLAYER_Y_ADJ`.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for climbing, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Increment `PLAYER_ROW`.
  - Put the player sprite on the active page at the new location.
  - Set the player's vertical adjustment to `-2`.
  - Update the player animation for climbing, and draw the player.

148 ⟨*try moving down* 148⟩≡ (213)

```
    ORG     $6766
TRY_MOVING_DOWN:
    SUBROUTINE

    LDY     PLAYER_Y_ADJ
    CPY     #$02
    BCC     .move_player_down   ; player slightly above, so can move down.

    LDY     PLAYER_ROW
    CPY     #$0F
    BCS     .cannot_move        ; player on row >= 15, so cannot move.

    ⟨set active row pointer PTR1 for Y+1 76d⟩
    LDY     PLAYER_COL
    LDA     (PTR1),Y
    CMP     #$02                ; stone
```

```
    BEQ     .cannot_move
    CMP     #$01                ; brick
    BNE     .move_player_down   ; Row below is stone or brick, so cannot move.

.cannot_move:
    SEC
    RTS

.move_player_down:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    JSR     MOVE_PLAYER_TOWARDS_EXACT_COLUMN
    INC     PLAYER_Y_ADJ                ; Move player down
    LDA     PLAYER_Y_ADJ
    CMP     #$05
    BCC     .check_for_gold_

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .set_on_real_page
    LDA     #$00
.set_on_real_page:
    STA     (PTR1),Y

    INC     PLAYER_ROW
    LDY     PLAYER_ROW
    ⟨set active row pointer PTR1 for Y 75c⟩
    LDY     PLAYER_COL
    LDA     #$09
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$00
    STA     PLAYER_Y_ADJ        ; Set adjustment to -2
    JMP     TRY_MOVING_UP_inc_anim_state

.check_for_gold_:
    JMP     TRY_MOVING_UP_check_for_gold
```

Defines:
TRY_MOVING_DOWN, used in chunk 164.
Uses ERASE_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, MOVE_PLAYER_TOWARDS_EXACT_COLUMN 143, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_Y_ADJ 81b, PTR1 75b, and PTR2 75b.

For attempting to move left, the logic is:

- If the player is slightly right of the sprite, then move the player left.
- Otherwise, if the player is on column 0, the player cannot move left.
- Otherwise, if the column to the left is stone, brick, or T-thing, the player cannot move left.
- Otherwise, the player can move left.

The steps involved in actually moving the player left are:

- Erase the player sprite.
- Set the PLAYER_FACING_DIRECTION to left (0xFF).
- Reduce any vertical adjustment, checking for gold pickup if not already exactly on a sprite column.
- Adjust the player horizontally to the left by decrementing PLAYER_X_ADJ.
- If the adjustment didn't roll over, check for gold pickup, then update the player animation for moving left, and draw the player.
- Otherwise:
  - Copy the background sprite at the player's sprite location to the active page, unless that sprite is a brick, in which case place an empty on the active page.
  - Decrement PLAYER_COL.
  - Put the player sprite on the active page at the new location.
  - Set the player's horizontal adjustment to +2.
  - Update the player animation for moving left, and draw the player.

The animation is either monkey-traversing if the player moves onto a rope, or running otherwise.

150 ⟨*try moving left* 150⟩≡ (213)

```
    ORG     $65D3
TRY_MOVING_LEFT:
    SUBROUTINE

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDX     PLAYER_X_ADJ
    CPX     #$03
    BCS     .move_player_left       ; player slightly right, so can move left.

    LDY     PLAYER_COL
```

```
    BEQ     .cannot_move            ; col == 0, so cannot move.

    DEY
    LDA     (PTR1),Y
    CMP     #$02
    BEQ     .cannot_move
    CMP     #$01
    BEQ     .cannot_move
    CMP     #$05
    BEQ     .move_player_left       ; brick, stone, or T-thing to left, so cannot move.

.cannot_move:
    RTS

.move_player_left:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDA     #$FF
    STA     PLAYER_FACING_DIRECTION             ; face left
    JSR     MOVE_PLAYER_TOWARDS_EXACT_ROW
    DEC     PLAYER_X_ADJ
    BPL     .check_for_gold

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .set_on_level
    LDA     #$00
.set_on_level:
    STA     (PTR1),Y

    DEC     PLAYER_COL
    DEY
    LDA     #$09
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$04
    STA     PLAYER_X_ADJ        ; Set adjustment to +2
    BNE     .inc_anim_state     ; Unconditional

.check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.inc_anim_state:
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$04                ; rope
    BEQ     .anim_state_monkeying

    LDA     #$00
```

```
    LDX     #$02
    BNE     .done               ; Unconditional

.anim_state_monkeying:
    LDA     #$03
    LDX     #$05

.done:
    JSR     INC_ANIM_STATE
    JMP     DRAW_PLAYER
```

Defines:
TRY_MOVING_LEFT, used in chunk 164.

Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 127, DRAW_PLAYER 41, ERASE_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, INC_ANIM_STATE 125b, MOVE_PLAYER_TOWARDS_EXACT_ROW 143, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_X_ADJ 81b, PTR1 75b, and PTR2 75b.

Moving right has the same logic as moving left, except in the other direction.

153 ⟨*try moving right* 153⟩≡ (213)

```
    ORG     $6645
TRY_MOVING_RIGHT:
    SUBROUTINE

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDX     PLAYER_X_ADJ
    CPX     #$02
    BCC     .move_player_right      ; player slightly left, so can move right.

    LDY     PLAYER_COL
    CPY     #$1B
    BEQ     .cannot_move            ; col == 27, so cannot move.

    INY
    LDA     (PTR1),Y
    CMP     #$02
    BEQ     .cannot_move
    CMP     #$01
    BEQ     .cannot_move
    CMP     #$05
    BEQ     .move_player_right      ; brick, stone, or T-thing to right, so cannot move.

.cannot_move:
    RTS

.move_player_right:
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDA     #$01
    STA     PLAYER_FACING_DIRECTION             ; face right
    JSR     MOVE_PLAYER_TOWARDS_EXACT_ROW
    INC     PLAYER_X_ADJ
    LDA     PLAYER_X_ADJ
    CMP     #$05
    BCC     .check_for_gold

    ; adjustment overflow
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .set_on_level
    LDA     #$00
.set_on_level:
    STA     (PTR1),Y

    INC     PLAYER_COL
```

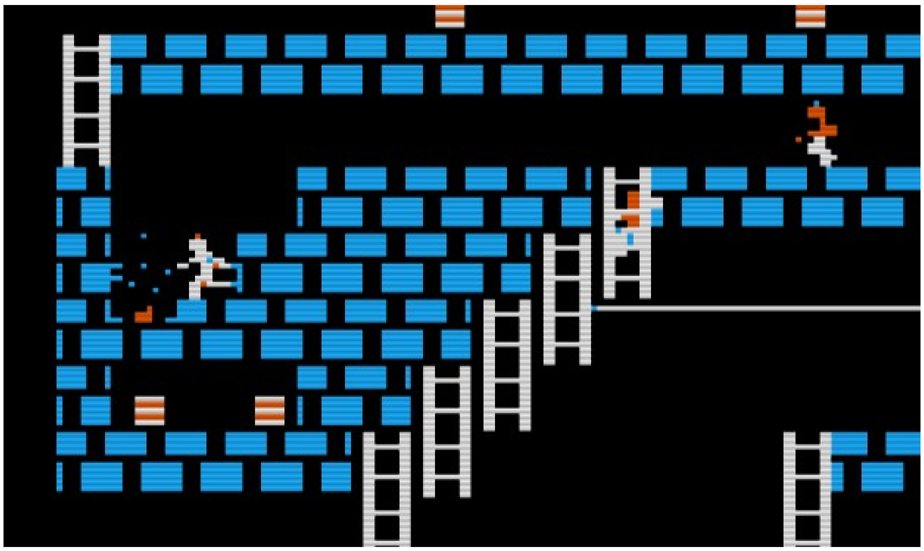
```
    INY
    LDA     #$09
    STA     (PTR1),Y            ; Write player sprite to active page.
    LDA     #$00
    STA     PLAYER_X_ADJ        ; Set adjustment to -2
    BNE     .inc_anim_state     ; Unconditional

.check_for_gold:
    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER

.inc_anim_state:
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$04                ; rope
    BEQ     .anim_state_monkeying

    LDA     #$08
    LDX     #$0A
    BNE     .done               ; Unconditional

.anim_state_monkeying:
    LDA     #$0B
    LDX     #$0D

.done:
    JSR     INC_ANIM_STATE
    JMP     DRAW_PLAYER
```

Defines:
TRY_MOVING_RIGHT, used in chunk 164.

Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 127, DRAW_PLAYER 41, ERASE_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, INC_ANIM_STATE 125b, MOVE_PLAYER_TOWARDS_EXACT_ROW 143, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_X_ADJ 81b, PTR1 75b, and PTR2 75b.

7.5 Digging

Provided there's nothing preventing the player from digging, digging involves a brick animation below and next to the player, and a "debris" animation above the dig site.



The DIG_DIRECTION location stores which direction we're digging in, and the DIG_ANIM_STATE location stores how far along in the 13-step animation cycle we are.

155 ⟨*defines* 3⟩+≡ (215) ◁134b 168▷

```
DIG_DIRECTION       EQU     $9C     ; 0xFF = left, 0x00 = not digging, 0x01 = right
DIG_ANIM_STATE      EQU     $A0     ; 00-0C
```

Defines:
DIG_DIRECTION, used in chunks 157, 160, 163, 164, and 199.

The DIG_DEBRIS_LEFT_SPRITES, DIG_DEBRIS_RIGHT_SPRITES and DIG_BRICK_SPRITES tables contain the sprites used during the animation. There's also a little sequence of notes that plays while digging, given by DIG_NOTE_PITCHES and DIG_NOTE_DURATIONS.

156 ⟨*tables* 7⟩+≡ (215) ◁137 171a▷

```
    ORG     $697A
DIG_DEBRIS_LEFT_SPRITES:
    HEX     1B 1B 1C 1C 1D 1D 1E 1E 00 00 00 00
DIG_DEBRIS_RIGHT_SPRITES:
    HEX     26 26 27 27 1D 1D 1E 1E 00 00 00 00
DIG_BRICK_SPRITES:
    HEX     1F 1F 20 20 21 21 22 22 23 23 24 24
DIG_NOTE_PITCHES:
    HEX     20 20 20 20 20 20 20 20 24 24 24 24 24
DIG_NOTE_DURATIONS:
    HEX     04 04 04 04 04 04 04 04 03 03 02 02 01
```

Defines:
DIG_BRICK_SPRITES, used in chunks 157 and 160.
DIG_DEBRIS_LEFT_SPRITES, used in chunks 157 and 160.
DIG_DEBRIS_RIGHT_SPRITES, never used.
DIG_NOTE_DURATIONS, used in chunks 157 and 160.
DIG_NOTE_PITCHES, used in chunks 157 and 160.

The player cannot dig to the left if they're on the bottom-most row or the leftmost column, or if there's no brick below and to the left. Also, there has to be nothing to the left of the player.

157 ⟨*try digging left* 157⟩≡ (213)

```
    ORG     $67D8
    SUBROUTINE

.cannot_dig_:
    JMP     .stop_digging

TRY_DIGGING_LEFT:
    LDA     #$FF
    STA     DIG_DIRECTION
    STA     KEY_COMMAND
    STA     $9F                 ; DIG_DIRECTION = KEY_COMMAND = 0xFF
    LDA     #$00
    STA     DIG_ANIM_STATE      ; DIG_ANIM_STATE = 0

TRY_DIGGING_LEFT_check_can_dig_left:
    LDY     PLAYER_ROW
    CPY     #$0F
    BCS     .cannot_dig_        ; row >= 15, so cannot dig.

    INY
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    BEQ     .cannot_dig_        ; col == 0, so cannot dig left.

    DEY
    LDA     (PTR1),Y
    CMP     #$01
    BNE     .cannot_dig_        ; no brick below and to the left, so cannot dig left.

    LDY     PLAYER_ROW
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    DEY
    LDA     (PTR1),Y
    CMP     #$00
    BNE     .not_empty_to_left  ; not empty to the left, so maybe cannot dig left.

    ; Can dig!
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     MOVE_PLAYER_TOWARDS_EXACT_COLUMN
    JSR     MOVE_PLAYER_TOWARDS_EXACT_ROW
    LDY     DIG_ANIM_STATE
    LDA     DIG_NOTE_PITCHES,Y
    LDX     DIG_NOTE_DURATIONS,Y
```

```
    JSR     APPEND_NOTE

    LDX     DIG_ANIM_STATE
    LDA     #$00                ; running left
    CPX     #$00
    BCS     .note_0             ; DIG_ANIM_STATE >= 0
    LDA     #$06                ; digging left
.note_0:
    STA     PLAYER_ANIM_STATE
    JSR     DRAW_PLAYER

    LDX     DIG_ANIM_STATE
    CPX     #$0C
    BEQ     .move_player_left
    CPX     #$00
    BEQ     .draw_curr_dig              ; Don't have to erase previous dig debris sprite

    ; Erase the previous dig debris sprite
    LDA     DIG_DEBRIS_LEFT_SPRITES-1,X
    PHA
    LDX     PLAYER_COL
    DEX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

    LDX     DIG_ANIM_STATE
.draw_curr_dig:
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    DEX
    STX     GAME_COLNUM
    LDY     PLAYER_ROW
    STY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS     ; Draw current dig debris sprite above dig site

    LDX     DIG_ANIM_STATE
    LDA     DIG_BRICK_SPRITES,X
    INC     GAME_ROWNUM
    JSR     DRAW_SPRITE_PAGE1               ; Draw dig brick sprite at dig site

    INC     DIG_ANIM_STATE
    CLC
    RTS

.not_empty_to_left:
```

```
    LDY     PLAYER_ROW
    INY
    STY     GAME_ROWNUM
    LDY     PLAYER_COL
    DEY
    STY     GAME_COLNUM
    LDA     #$01
    JSR     DRAW_SPRITE_PAGE1          ; Draw brick below and to the left of player

    LDX     DIG_ANIM_STATE
    BEQ     .stop_digging

    ; Erase previous dig debris sprite
    DEX
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDY     PLAYER_ROW
    LDX     PLAYER_COL
    DEX
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

.stop_digging:
    LDA     #$00
    STA     DIG_DIRECTION
    SEC
    RTS

.move_player_left:
    LDX     PLAYER_COL
    DEX
    JMP     DROP_PLAYER_IN_HOLE
```

Defines:
TRY_DIGGING_LEFT, used in chunk 164.

Uses APPEND_NOTE 57a, DIG_BRICK_SPRITES 156, DIG_DEBRIS_LEFT_SPRITES 156, DIG_DIRECTION 155, DIG_NOTE_DURATIONS 156, DIG_NOTE_PITCHES 156, DRAW_PLAYER 41, DRAW_SPRITE_AT_PIXEL_COORDS 39, DRAW_SPRITE_PAGE1 33, DROP_PLAYER_IN_HOLE 163, ERASE_SPRITE_AT_PIXEL_COORDS 36, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA 76c, GET_SCREEN_COORDS_FOR 29a, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, KEY_COMMAND 128a, MOVE_PLAYER_TOWARDS_EXACT_COLUMN 143, MOVE_PLAYER_TOWARDS_EXACT_ROW 143, PLAYER_ANIM_STATE 81b, PLAYER_COL 77c, PLAYER_ROW 77c, and PTR1 75b.

160 ⟨*try digging right* 160⟩≡ (213)

```
    ORG     $689E
    SUBROUTINE

.cannot_dig_:
    JMP     .stop_digging

TRY_DIGGING_RIGHT:
    LDA     #$01
    STA     DIG_DIRECTION
    STA     KEY_COMMAND
    STA     $9F                 ; DIG_DIRECTION = KEY_COMMAND = 0x01
    LDA     #$0C
    STA     DIG_ANIM_STATE      ; DIG_ANIM_STATE = 0x0C

TRY_DIGGING_RIGHT_check_can_dig_right:
    LDY     PLAYER_ROW
    CPY     #$0F
    BCS     .cannot_dig_        ; row >= 15, so cannot dig.

    INY
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    CPY     #$1B
    BCS     .cannot_dig_        ; col >= 27, so cannot dig right.

    INY
    LDA     (PTR1),Y
    CMP     #$01
    BNE     .cannot_dig_        ; no brick below and to the right, so cannot dig right.

    LDY     PLAYER_ROW
    JSR     GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA
    LDY     PLAYER_COL
    INY
    LDA     (PTR1),Y
    CMP     #$00
    BNE     .not_empty_to_right  ; not empty to the right, so maybe cannot dig right.

    ; Can dig!
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     MOVE_PLAYER_TOWARDS_EXACT_COLUMN
    JSR     MOVE_PLAYER_TOWARDS_EXACT_ROW
    LDY     DIG_ANIM_STATE
    LDA     DIG_NOTE_PITCHES-12,Y
    LDX     DIG_NOTE_DURATIONS-12,Y
    JSR     APPEND_NOTE

    LDX     DIG_ANIM_STATE
```

```
    LDA     #$08                ; running right
    CPX     #$12
    BCS     .note_0             ; DIG_ANIM_STATE >= 0x12
    LDA     #$0E                ; digging right
.note_0:
    STA     PLAYER_ANIM_STATE
    JSR     DRAW_PLAYER

    LDX     DIG_ANIM_STATE
    CPX     #$18
    BEQ     .move_player_right
    CPX     #$0C
    BEQ     .draw_curr_dig              ; Don't have to erase previous dig debris sprite

    ; Erase the previous dig debris sprite
    LDA     DIG_DEBRIS_LEFT_SPRITES-1,X
    PHA
    LDX     PLAYER_COL
    INX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

    LDX     DIG_ANIM_STATE
.draw_curr_dig:
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    INX
    STX     GAME_ROWNUM
    LDY     PLAYER_ROW
    STY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS     ; Draw current dig debris sprite above dig site

    INC     GAME_ROWNUM
    LDX     DIG_ANIM_STATE
    LDA     DIG_BRICK_SPRITES-12,X
    JSR     DRAW_SPRITE_PAGE1               ; Draw dig brick sprite at dig site

    INC     DIG_ANIM_STATE
    CLC
    RTS

.not_empty_to_right:
    LDY     PLAYER_ROW
    INY
    STY     GAME_ROWNUM
```

```
    LDY     PLAYER_COL
    INY
    STY     GAME_COLNUM
    LDA     #$01
    JSR     DRAW_SPRITE_PAGE1           ; Draw brick below and to the right of player

    LDX     DIG_ANIM_STATE
    CPX     #$0C
    BEQ     .stop_digging

    ; Erase previous dig debris sprite
    DEX
    LDA     DIG_DEBRIS_LEFT_SPRITES,X
    PHA
    LDX     PLAYER_COL
    INX
    LDY     PLAYER_ROW
    JSR     GET_SCREEN_COORDS_FOR
    PLA
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS

.stop_digging:
    LDA     #$00
    STA     DIG_DIRECTION
    SEC
    RTS

.move_player_right:
    LDX     PLAYER_COL
    INX
    JMP     DROP_PLAYER_IN_HOLE
```

Defines:
TRY_DIGGING_RIGHT, used in chunk 164.

Uses APPEND_NOTE 57a, DIG_BRICK_SPRITES 156, DIG_DEBRIS_LEFT_SPRITES 156, DIG_DIRECTION 155, DIG_NOTE_DURATIONS 156, DIG_NOTE_PITCHES 156, DRAW_PLAYER 41, DRAW_SPRITE_AT_PIXEL_COORDS 39, DRAW_SPRITE_PAGE1 33, DROP_PLAYER_IN_HOLE 163, ERASE_SPRITE_AT_PIXEL_COORDS 36, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_PTRS_TO_CURR_LEVEL_SPRITE_DATA 76c, GET_SCREEN_COORDS_FOR 29a, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, KEY_COMMAND 128a, MOVE_PLAYER_TOWARDS_EXACT_COLUMN 143, MOVE_PLAYER_TOWARDS_EXACT_ROW 143, PLAYER_ANIM_STATE 81b, PLAYER_COL 77c, PLAYER_ROW 77c, and PTR1 75b.

163 ⟨*drop player in hole* 163⟩≡ (213)

```
    ORG     $6C39
DROP_PLAYER_IN_HOLE:
    SUBROUTINE

    LDA     #$00
    STA     DIG_DIRECTION       ; Stop digging

    LDY     PLAYER_ROW
    INY                         ; Move player down

    STX     GAME_COLNUM
    STY     GAME_ROWNUM
    ⟨set active row pointer PTR1 for Y 75c⟩
    LDA     #$00
    LDY     GAME_COLNUM
    STA     (PTR1),Y                ; Set blank sprite at player location in active page
    JSR     DRAW_SPRITE_PAGE1
    LDA     #$00
    JSR     DRAW_SPRITE_PAGE2       ; Draw blank at player location on both graphics pages

    DEC     GAME_ROWNUM
    LDA     #$00
    JSR     DRAW_SPRITE_PAGE1       ; Draw blank at location above player
    INC     GAME_ROWNUM
    LDX     #$FF

.loop:
    INX
    CPX     #$1E
    BEQ     .end
    LDA     TABLE_0CE0,X
    BNE     .loop

    LDA     GAME_ROWNUM
    STA     TABLE_0CC0,X
    LDA     GAME_COLNUM
    STA     TABLE_0CA0,X
    LDA     #$B4
    STA     TABLE_0CE0,X
    SEC

.end:
    RTS
```

Defines:
DROP_PLAYER_IN_HOLE, used in chunks 157 and 160.
Uses DIG_DIRECTION 155, DRAW_SPRITE_PAGE1 33, DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, PLAYER_ROW 77c, and PTR1 75b.

The MOVE_PLAYER routine handle continuation of digging, player falling, and player keyboard input.

164 ⟨*move player* 164⟩≡ (213)

```
    ORG     $64BD
MOVE_PLAYER:
    SUBROUTINE

    LDA     #$01
    STA     DIDNT_PICK_UP_GOLD   ; Reset DIDNT_PICK_UP_GOLD

    ; If we're digging, see if we can keep digging.
    LDA     DIG_DIRECTION
    BEQ     .not_digging
    BPL     .digging_right
    JMP     TRY_DIGGING_LEFT_check_can_dig_left

.digging_right:
    JMP     TRY_DIGGING_RIGHT_check_can_dig_right

.not_digging:
    LDY     PLAYER_ROW
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$03
    BEQ     .check_for_keyboard_input_      ; ladder at background location?
    CMP     #$04
    BEQ     .check_if_player_should_fall    ; rope at background location?
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BEQ     .check_for_keyboard_input_      ; player at exact sprite row?

    ; player is not on exact sprite row, fallthrough.

.check_if_player_should_fall:
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BCC     .make_player_fall               ; player slightly above sprite row?

    LDY     PLAYER_ROW
    CPY     #$0F
    BEQ     .check_for_keyboard_input_      ; player exactly sprite row 15?

    ; Check the sprite at the player location
    ⟨set active and background row pointers PTR1 and PTR2 for Y+1 77b⟩

    LDY     PLAYER_COL
    LDA     (PTR1),Y
    CMP     #$00                    ; Empty
```

```
    BEQ     .make_player_fall
    CMP     #$08                    ; Guard
    BEQ     .check_for_keyboard_input_
    LDA     (PTR2),Y
    CMP     #$01                    ; Brick
    BEQ     .check_for_keyboard_input_
    CMP     #$02                    ; Stone
    BEQ     .check_for_keyboard_input_
    CMP     #$03                    ; Ladder
    BNE     .make_player_fall

.check_for_keyboard_input_:
    JMP     .check_for_keyboard_input

.make_player_fall:
    LDA     #$00
    STA     $9B                     ; $9B = 0
    JSR     GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER
    JSR     ERASE_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            ; Move down one
    LDA     PLAYER_Y_ADJ
    CMP     #$05
    BCS     .adjustment_overflow

    JSR     CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER
    JMP     DRAW_PLAYER            ; tailcall

.adjustment_overflow:
    LDA     #$00
    STA     PLAYER_Y_ADJ            ; Set vertical adjust to -2

    LDY     PLAYER_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y+1 77b⟩
    LDY     PLAYER_COL
    LDA     (PTR2),Y
    CMP     #$01                ; Brick
    BNE     .set_on_level
    LDA     #$00                ; Store empty sprite
.set_on_level:
    STA     (PTR1),Y
```

```
    INC     PLAYER_ROW          ; Move down
```

⟨*set active row pointer* `PTR1` *for* `Y+1` 76d⟩

```
    LDY     PLAYER_COL
    LDA     #$09            ; player facing right
    STA     (PTR1),Y
    JMP     DRAW_PLAYER     ; tailcall

.check_for_keyboard_input:
    LDA     $9B
    BNE     .check_for_key  ; $9B doesn't play note
    LDA     #$64
    LDX     #$08
    JSR     PLAY_NOTE       ; play note, pitch 0x64, duration 8.

.check_for_key:
    LDA     #$20
    STA     $A4
    STA     $9B
    JSR     CHECK_FOR_INPUT
    LDA     KEY_COMMAND
    CMP     #$C9            ; 'I'
    BNE     .check_for_K
    JSR     TRY_MOVING_UP
    BCS     .check_for_J    ; couldn't move up
    RTS

.check_for_K:
    CMP     #$CB            ; 'K'
    BNE     .check_for_U
    JSR     TRY_MOVING_DOWN
    BCS     .check_for_J
    RTS

.check_for_U:
    CMP     #$D5            ; 'U'
    BNE     .check_for_O
    JSR     TRY_DIGGING_LEFT
    BCS     .check_for_J
    RTS

.check_for_O:
    CMP     #$CF            ; 'O'
    BNE     .check_for_J
    JSR     TRY_DIGGING_RIGHT
    BCS     .check_for_J
    RTS

.check_for_J:
```

```
    LDA     $9F
    CMP     #$CA            ; 'J'
    BNE     .check_for_L
    JMP     TRY_MOVING_LEFT

.check_for_L:
    CMP     #$CC            ; 'L'
    BNE     .end
    JMP     TRY_MOVING_RIGHT

.end:
    RTS
```

Defines:
MOVE_PLAYER, used in chunk 200.

Uses CHECK_FOR_GOLD_PICKED_UP_BY_PLAYER 127, CHECK_FOR_INPUT 129, DIDNT_PICK_UP_GOLD 126, DIG_DIRECTION 155, DRAW_PLAYER 41, ERASE_SPRITE_AT_PIXEL_COORDS 36, GET_SPRITE_AND_SCREEN_COORD_AT_PLAYER 125a, KEY_COMMAND 128a, PLAY_NOTE 58, PLAYER_ANIM_STATE 81b, PLAYER_COL 77c, PLAYER_ROW 77c, PLAYER_Y_ADJ 81b, PTR1 75b, PTR2 75b, TRY_DIGGING_LEFT 157, TRY_DIGGING_RIGHT 160, TRY_MOVING_DOWN 148, TRY_MOVING_LEFT 150, TRY_MOVING_RIGHT 153, and TRY_MOVING_UP 146.

Chapter 8

Guard AI

168 ⟨*defines* 3⟩+≡ (215) ◁155 181▷

```
GUARD_LOCS_COL      EQU     $0C60       ; 8 bytes
GUARD_LOCS_ROW      EQU     $0C68       ; 8 bytes
GUARD_FLAGS_0       EQU     $0C70       ; 8 bytes
GUARD_X_ADJS        EQU     $0C78       ; 8 bytes
GUARD_Y_ADJS        EQU     $0C80       ; 8 bytes
GUARD_ANIM_STATES   EQU     $0C88       ; 8 bytes
GUARD_FLAGS_4       EQU     $0C90       ; 8 bytes
GUARD_FLAGS_5       EQU     $0C98       ; 8 bytes

GUARD_LOC_COL       EQU     $12
GUARD_LOC_ROW       EQU     $13
GUARD_ANIM_STATE    EQU     $14
GUARD_FLAG_4        EQU     $15
GUARD_FLAG_0        EQU     $16
GUARD_X_ADJ         EQU     $17
GUARD_Y_ADJ         EQU     $18
GUARD_NUM           EQU     $19

GUARD_PATTERN       EQU     $63
GUARD_PHASE         EQU     $64
```

Defines:
- GUARD_ANIM_STATE, used in chunks 170, 171b, and 174.
- GUARD_ANIM_STATES, used in chunks 80b and 170.
- GUARD_FLAG_0, used in chunks 170 and 174.
- GUARD_FLAG_4, used in chunks 170 and 174.
- GUARD_FLAGS_0, used in chunks 80b and 170.
- GUARD_FLAGS_4, used in chunk 170.
- GUARD_LOC_COL, used in chunks 170, 171b, and 174.
- GUARD_LOC_ROW, used in chunks 170, 171b, and 174.
- GUARD_LOCS_COL, used in chunks 80b, 135, and 170.
- GUARD_LOCS_ROW, used in chunks 80b, 135, and 170.
- GUARD_NUM, used in chunks 170 and 174.
- GUARD_PATTERN, used in chunk 172a.
- GUARD_PHASE, used in chunk 172a.

170 ⟨*guard store and load data* 170⟩≡ (213)

```
      ORG     $75A8
  STORE_GUARD_DATA:
      SUBROUTINE

      LDX     GUARD_NUM
      LDA     GUARD_LOC_COL
      STA     GUARD_LOCS_COL,X
      LDA     GUARD_LOC_ROW
      STA     GUARD_LOCS_ROW,X
      LDA     GUARD_X_ADJ
      STA     GUARD_X_ADJS,X
      LDA     GUARD_Y_ADJ
      STA     GUARD_Y_ADJS,X
      LDA     GUARD_FLAG_0
      STA     GUARD_FLAGS_0,X
      LDA     GUARD_FLAG_4
      STA     GUARD_FLAGS_4,X
      LDA     GUARD_ANIM_STATE
      STA     GUARD_ANIM_STATES,X
      RTS

  LOAD_GUARD_DATA:
      SUBROUTINE

      LDX     GUARD_NUM
      LDA     GUARD_LOCS_COL,X
      STA     GUARD_LOC_COL
      LDA     GUARD_LOCS_ROW,X
      STA     GUARD_LOC_ROW
      LDA     GUARD_X_ADJS,X
      STA     GUARD_X_ADJ
      LDA     GUARD_Y_ADJS,X
      STA     GUARD_Y_ADJ
      LDA     GUARD_ANIM_STATES,X
      STA     GUARD_ANIM_STATE
      LDA     GUARD_FLAGS_4,X
      STA     GUARD_FLAG_4
      LDA     GUARD_FLAGS_0,X
      STA     GUARD_FLAG_0
      RTS
```

Defines:
  LOAD_GUARD_DATA, used in chunk 174.
  STORE_GUARD_DATA, used in chunks 173 and 174.
Uses GUARD_ANIM_STATE 168, GUARD_ANIM_STATES 168, GUARD_FLAG_0 168, GUARD_FLAG_4 168, GUARD_FLAGS_0 168, GUARD_FLAGS_4 168, GUARD_LOC_COL 168, GUARD_LOC_ROW 168, GUARD_LOCS_COL 168, GUARD_LOCS_ROW 168, GUARD_NUM 168, GUARD_X_ADJ 168, GUARD_X_ADJS 168, GUARD_Y_ADJ 168, and GUARD_Y_ADJS 168.

171a ⟨*tables* 7⟩+≡ (215) ◁156 171c▷

```
      ORG     $6CCB
  GUARD_ANIM_SPRITES:
      HEX     08 2B 2C        ; running left
      HEX     30 31 32        ; monkey-traversing left
      HEX     36              ; falling left
      HEX     28 29 2A        ; running right
      HEX     2D 2E 2F        ; monkey-traversing right
      HEX     35              ; falling right
      HEX     33 34           ; climbing
```

Defines:
GUARD_ANIM_SPRITES, used in chunk 171b.

171b ⟨*get guard sprite and coords* 171b⟩≡ (213)

```
      ORG     $74DF
  GET_GUARD_SPRITE_AND_COORDS:
      SUBROUTINE

      LDX     GUARD_LOC_COL
      LDY     GUARD_X_ADJ
      JSR     GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR
      STX     SPRITE_NUM

      LDY     GUARD_LOC_ROW
      LDX     GUARD_Y_ADJ
      JSR     GET_SCREEN_ROW_OFFSET_IN_X_FOR
      LDX     GUARD_ANIM_STATE
      LDA     GUARD_ANIM_SPRITES,X

      LDX     SPRITE_NUM
      RTS
```

Uses GET_HALF_SCREEN_COL_OFFSET_IN_Y_FOR 31c, GET_SCREEN_ROW_OFFSET_IN_X_FOR 31a, GUARD_ANIM_SPRITES 171a, GUARD_ANIM_STATE 168, GUARD_LOC_COL 168, GUARD_LOC_ROW 168, GUARD_X_ADJ 168, GUARD_Y_ADJ 168, and SPRITE_NUM 23c.

171c ⟨*tables* 7⟩+≡ (215) ◁171a 172b▷

```
      ORG     $0060
  GUARD_PATTERNS:
      BYTE    %10000110
      BYTE    %00111110
      BYTE    %10000101
```

Defines:
GUARD_PATTERNS, used in chunks 172a and 199.

172a ⟨*move guards* 172a⟩≡ (213)
```
    ORG     $6C82
MOVE_GUARDS:
    SUBROUTINE

    LDX     GUARD_COUNT
    BEQ     .end

    ; Increment GUARD_PHASE mod 3
    INC     GUARD_PHASE
    LDY     GUARD_PHASE
    CPY     #$03
    BCC     .incremented_phase
    LDY     #$00
    STY     GUARD_PHASE
.incremented_phase:

    LDA     GUARD_PATTERNS,Y
    STA     GUARD_PATTERN

.loop:
    LSR     GUARD_PATTERN       ; Peel off the lsb
    BCC     .bit_done
    JSR     MOVE_GUARD          ; Move a guard
    LDA     ALIVE
    BEQ     .end                ; If player is dead, end.

.bit_done:
    LDA     GUARD_PATTERN
    BNE     .loop

.end:
    RTS
```
Defines:
MOVE_GUARDS, used in chunk 200.
Uses ALIVE 106d, GUARD_COUNT 78d, GUARD_PATTERN 168, GUARD_PATTERNS 171c, GUARD_PHASE 168, and MOVE_GUARD 174.

172b ⟨*tables* 7⟩+≡ (215) ◁171c 173▷
```
    ORG     $6E7F
GUARD_X_ADJ_TABLE:
    HEX     02 01 02 03 02 01
```
Defines:
GUARD_X_ADJ_TABLE, used in chunk 174.

173 ⟨*tables* 7⟩+≡ (215) ◁172b 182▷

```
      ORG     $6E97
  GUARD_FN_TABLE:
      WORD    STORE_GUARD_DATA-1
```

Defines:
  GUARD_FN_TABLE, used in chunk 174.
Uses STORE_GUARD_DATA 170.

174 ⟨*move guard* 174⟩≡ (213)

```
    ORG     $6CDB
MOVE_GUARD
    SUBROUTINE

    ; Increment GUARD_NUM mod GUARD_COUNT, except 1-based.
    INC     GUARD_NUM
    LDX     GUARD_COUNT
    CPX     GUARD_NUM
    BCS     .guard_num_incremented
    LDX     #$01
    STX     GUARD_NUM
.guard_num_incremented:

    JSR     LOAD_GUARD_DATA
    LDA     GUARD_FLAG_0
    BMI     .check_sprite_at_guard_pos
    BEQ     .check_sprite_at_guard_pos

    DEC     GUARD_FLAG_0
    LDY     GUARD_FLAG_0
    CPY     #$0D
    BCS     .guard_flag_0_gt_12
    JMP     $6e65

.guard_flag_0_gt_12:
    LDX     GUARD_NUM
    LDA     GUARD_FLAGS_5,X
    BEQ     .guard_flag_5_zero
    JMP     STORE_GUARD_DATA            ; tailcall

.guard_flag_5_zero:
    JMP     $6db7

.check_sprite_at_guard_pos:
    LDY     GUARD_LOC_ROW
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y

    CMP     #$03
    BEQ     .ladder_
    CMP     #$04
    BNE     .not_rope_or_ladder
    LDA     GUARD_Y_ADJ
    CMP     #$02
    BEQ     .ladder_

.not_rope_or_ladder:
    LDA     GUARD_Y_ADJ
```

```
    CMP     #$02
    BCC     .blank_or_player           ; if GUARD_Y_ADJ < 2
    LDY     GUARD_LOC_ROW
    CPY     #$0F
    BEQ     .ladder_          ; Row == 15
    ⟨set active and background row pointers PTR2 and PTR1 for Y 76b⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y

    CMP     #$00
    BEQ     .blank_or_player
    CMP     #$09
    BEQ     .blank_or_player
    CMP     #$08               ; guard
    BEQ     .ladder_

    LDA     (PTR2),Y
    CMP     #$01               ; brick
    BEQ     .ladder_
    CMP     #$02               ; stone
    BEQ     .ladder_
    CMP     #$03               ; ladder
    BNE     .blank_or_player

.ladder_:
    JMP     .ladder

.blank_or_player:
    JSR     $74DF
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    JSR     $7582
    LDA     #$06
    LDY     GUARD_FLAG_4
    BMI     .set_guard_flag_3
    LDA     #$0D
.set_guard_flag_3
    STA     GUARD_ANIM_STATE

    INC     GUARD_Y_ADJ
    LDA     GUARD_Y_ADJ
    CMP     #$05
    BCS     $6dc0           ; If GUARD_Y_ADJ > 4

    LDA     GUARD_Y_ADJ
    CMP     #$02
    BNE     $6db7           ; If GUARD_Y_ADJ != 2

    LDY     GUARD_LOC_ROW
    ⟨set background row pointer PTR2 for Y 75d⟩
    LDY     GUARD_LOC_COL
```

```
    LDA     (PTR2),Y

    CMP     #$01
    BNE     $6db7           ; If background screen has brick

    LDA     GUARD_FLAG_0
    BPL     .6da2
    DEC     GOLD_COUNT
.6da2:
    LDA     $5F
    STA     GUARD_FLAG_0
    LDY     #$00
    LDA     #$75
    JSR     ADD_AND_UPDATE_SCORE        ; SCORE += 75

    ; Play the guard kill tune
    JSR     LOAD_SOUND_DATA
    HEX     06 20 04 30 02 40 00

    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA            ; tailcall

.6dc0:
    LDA     #$00
    STA     GUARD_Y_ADJ                 ; set vertical adjust to -2
    LDY     GUARD_LOC_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .set_real_sprite
    LDA     #$00
.set_real_sprite:
    STA     (PTR1),Y

    INC     GUARD_LOC_ROW               ; move guard down
    LDY     GUARD_LOC_ROW
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDY     GUARD_LOC_COL
    LDA     (PTR1),Y
    CMP     #$09
    BNE     .get_background_sprite
    LSR     ALIVE                       ; set player to dead
.get_background_sprite:
    LDA     (PTR2),Y
    CMP     #$01
    BNE     .place_guard_at_loc
    LDA     GUARD_FLAG_0
    BPL     .place_guard_at_loc
```

```
    ; What's above the guard?
    LDY     GUARD_LOC_ROW
    DEY
    STY     GAME_ROWNUM
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDY     GUARD_LOC_COL
    STY     GAME_COLNUM
    LDA     (PTR2),Y
    CMP     #$00
    BEQ     .drop_gold
    DEC     GOLD_COUNT
    JMP     .6e46

.drop_gold:
    LDA     #$07
    STA     (PTR1),Y
    STA     (PTR2),Y
    JSR     DRAW_SPRITE_PAGE2
    LDY     GAME_ROWNUM
    LDX     GAME_COLNUM
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #$07
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS

.6e46
    LDY     GUARD_LOC_ROW
    ⟨set active row pointer PTR1 for Y 75c⟩
    LDA     #$00
    STA     GUARD_FLAG_0
    LDY     GUARD_LOC_COL

.place_guard_at_loc
    LDA     #$08            ; guard
    STA     (PTR1),Y

    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
    JMP     STORE_GUARD_DATA            ; tailcall

.6e65:
    CPY     #$07
    BCC     .ladder
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     ERASE_SPRITE_AT_PIXEL_COORDS
    LDY     GUARD_FLAG_0
    LDA     GUARD_X_ADJ_TABLE-7,Y
    STA     GUARD_X_ADJ
    JSR     GET_GUARD_SPRITE_AND_COORDS
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS
```

```
    JMP     STORE_GUARD_DATA          ; tailcall

    ORG     $6E85
.ladder
    LDX     GUARD_LOC_COL
    LDY     GUARD_LOC_ROW
    JSR     $70D8
    ASL
    TAY
    LDA     GUARD_FN_TABLE+1,Y
    PHA
    LDA     GUARD_FN_TABLE,Y
    PHA
    RTS
```

Defines:
MOVE_GUARD, used in chunk 172a.
Uses ADD_AND_UPDATE_SCORE 49, ALIVE 106d, DRAW_SPRITE_AT_PIXEL_COORDS 39, DRAW_SPRITE_PAGE2 33, ERASE_SPRITE_AT_PIXEL_COORDS 36, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_SCREEN_COORDS_FOR 29a, GOLD_COUNT 78d, GUARD_ANIM_STATE 168, GUARD_COUNT 78d, GUARD_FLAG_0 168, GUARD_FLAG_4 168, GUARD_FN_TABLE 173, GUARD_LOC_COL 168, GUARD_LOC_ROW 168, GUARD_NUM 168, GUARD_X_ADJ 168, GUARD_X_ADJ_TABLE 172b, GUARD_Y_ADJ 168, LOAD_GUARD_DATA 170, LOAD_SOUND_DATA 56, PTR1 75b, PTR2 75b, SCORE 48b, and STORE_GUARD_DATA 170.

DO_LADDERS goes through the registered ladder locations from last to first. Recall that the ladder indices are 1-based, so that LADDER_LOCS_[0] does not contain ladder data. Instead, that location is used as scratch space by this routine.

Recall also that LADDER_LOCS_[X] is negative if there is no ladder corresponding to entry X.

For each ladder, if there's a non-blank sprite on the background sprite page for it, we set LADDER_LOCS_COL to 1.

However, if there is a blank sprite on the background sprite page for it, then set it to the ladder sprite, and if it's also blank on the active sprite page, set that to the ladder sprite, too. Then draw the ladder on the background and active graphics pages, remove the ladder from the registered locations, and keep going.

Once all ladder locations have been gone through, if LADDER_LOCS_COL is 1—that is, if there was a non-blank sprite on the background sprite page for any ladder location—then decrement the gold count. Since this routine is only called when GOLD_COUNT is zero, this sets GOLD_COUNT to -1.

179 ⟨*do ladders* 179⟩≡ (213)

```
    ORG     $8631
DO_LADDERS:
    SUBROUTINE

    LDA     #$00
    STA     LADDER_LOCS_COL     ; LADDER_LOCS_COL = 0

    LDX     LADDER_COUNT
    STX     .count              ; .count backwards from LADDER_COUNT to 0
.loop:
    LDX     .count
    BEQ     .dec_gold_count_if_no_ladder

    LDA     LADDER_LOCS_COL,X   ; A = LADDER_LOCS_COL[X]
    BMI     .next               ; If not present, next.

    STA     GAME_COLNUM         ; GAME_COLNUM = LADDER_LOCS_COL[X]
    LDA     LADDER_LOCS_ROW,X
    STA     GAME_ROWNUM         ; GAME_ROWNUM = LADDER_LOCS_ROW[X]
    TAY
    ⟨set active and background row pointers PTR1 and PTR2 for Y 76a⟩
    LDY     GAME_COLNUM
    LDA     (PTR2),Y            ; A = sprite at ladder loc
    BNE     .set_col_to_1

    LDA     #$03
    STA     (PTR2),Y            ; Set background sprite to ladder
    LDA     (PTR1),Y
    BNE     .draw_ladder        ; .draw_ladder if active sprite not blank
```

```
    LDA     #$03
    STA     (PTR1),Y            ; Set active sprite to ladder

.draw_ladder:
    LDA     #$03
    JSR     DRAW_SPRITE_PAGE2   ; Draw ladder on background page

    LDX     GAME_COLNUM
    LDY     GAME_ROWNUM
    JSR     GET_SCREEN_COORDS_FOR
    LDA     #$03
    JSR     DRAW_SPRITE_AT_PIXEL_COORDS ; Draw ladder on active page

    LDX     .count
    LDA     #$FF
    STA     LADDER_LOCS_COL,X       ; Remove ladder loc
    BMI     .next                   ; Unconditional

.set_col_to_1:
    LDA     #$01
    STA     LADDER_LOCS_COL         ; LADDER_LOCS_COL = 1

.next:
    DEC     .count
    JMP     .loop

.dec_gold_count_if_no_ladder:
    LDA     LADDER_LOCS_COL
    BNE     .end
    DEC     GOLD_COUNT

.end:
    RTS

.count:
    BYTE    0
```

Defines:
DO_LADDERS, never used.

Uses DRAW_SPRITE_AT_PIXEL_COORDS 39, DRAW_SPRITE_PAGE2 33, GAME_COLNUM 32a, GAME_ROWNUM 32a, GET_SCREEN_COORDS_FOR 29a, GOLD_COUNT 78d, LADDER_COUNT 78d, LADDER_LOCS_COL 79a, LADDER_LOCS_ROW 79a, PTR1 75b, and PTR2 75b.

Chapter 9

Disk routines

There appears to be a copy of the DOS RWTS loaded into the usual location at `$BD00`. In addition, the standard DOS IOB and DCT are used. Further details can be read in Beneath Apple DOS.

181 ⟨*defines* 3⟩+≡ (215) ◁168 184▷

```
DOS_IOB                     EQU     $B7E8
IOB_SLOTNUMx16              EQU     $B7E9
IOB_DRIVE_NUM               EQU     $B7EA
IOB_VOLUME_NUMBER_EXPECTED  EQU     $B7EB
IOB_TRACK_NUMBER            EQU     $B7EC
IOB_SECTOR_NUMBER           EQU     $B7ED
IOB_DEVICE_CHARACTERISTICS_TABLE_PTR        EQU     $B7EE   ; 2 bytes
IOB_READ_WRITE_BUFFER_PTR   EQU     $B7F0   ; 2 bytes
IOB_UNUSED                  EQU     $B7F2
IOB_BYTE_COUNT_FOR_PARTIAL_SECTOR   EQU     $B7F3
IOB_COMMAND_CODE            EQU     $B7F4
IOB_RETURN_CODE             EQU     $B7F5
IOB_LAST_ACCESS_VOLUME      EQU     $B7F6
IOB_LAST_ACCESS_SLOTx16     EQU     $B7F7
IOB_LAST_ACCESS_DRIVE       EQU     $B7F8

DCT_DEVICE_TYPE             EQU     $B7FB
DCT_PHASES_PER_TRACK        EQU     $B7FC
DCT_MOTOR_ON_TIME_COUNT     EQU     $B7FD   ; 2 bytes
```

Defines:
- `DCT_DEVICE_TYPE`, never used.
- `DCT_MOTOR_ON_TIME_COUNT`, never used.
- `DCT_PHASES_PER_TRACK`, never used.
- `DOS_IOB`, used in chunks 105 and 183.
- `IOB_BYTE_COUNT_FOR_PARTIAL_SECTOR`, never used.
- `IOB_COMMAND_CODE`, used in chunks 105, 183, and 192.
- `IOB_DEVICE_CHARACTERISTICS_TABLE_PTR`, never used.
- `IOB_DRIVE_NUM`, never used.
- `IOB_LAST_ACCESS_DRIVE`, never used.
- `IOB_LAST_ACCESS_SLOTx16`, never used.

IOB_LAST_ACCESS_VOLUME, never used.
IOB_READ_WRITE_BUFFER_PTR, used in chunks 105, 183, and 192.
IOB_RETURN_CODE, never used.
IOB_SECTOR_NUMBER, used in chunks 105, 183, and 192.
IOB_SLOTNUMx16, never used.
IOB_TRACK_NUMBER, used in chunks 105, 183, and 192.
IOB_UNUSED, never used.
IOB_VOLUME_NUMBER_EXPECTED, used in chunks 105 and 183.

ACCESS_HI_SCORE_DATA_FROM_DISK reads or writes—depending on A, where 1 is read and 2 is write—the high score table from disk at track 12 sector 15 into HI_SCORE_TABLE. We then compare the 11 bytes of HI_SCORE_DATA_MARKER to where they are supposed to be in the table.

If the marker doesn't match, then we return 0, indicating that the disk doesn't have a high score table.

If the marker does match, but the very last byte in the table is nonzero, then we return 1, indicating that this is a master disk (so its level data shouldn't be touched), otherwise we return -1, this being a data disk.

182 ⟨*tables* 7⟩+≡ (215) ◁173 191▷

```
    ORG     $63A8
HI_SCORE_DATA_MARKER:
    ; Spells out "LODE RUNNER".
    HEX     CC CF C4 C5 A0 D2 D5 CE CE C5 D2
```

Defines:
HI_SCORE_DATA_MARKER, used in chunks 183 and 192.

183 ⟨*access hi score data* 183⟩≡ (213)

```
    ORG     $6359
ACCESS_HI_SCORE_DATA_FROM_DISK:
    SUBROUTINE

    STA     IOB_COMMAND_CODE
    LDA     #$0C
    STA     IOB_TRACK_NUMBER
    LDA     #$0F
    STA     IOB_SECTOR_NUMBER
    LDA     #<HI_SCORE_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR
    LDA     #>HI_SCORE_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR+1
    LDA     #$00
    STA     IOB_VOLUME_NUMBER_EXPECTED
    LDY     #<DOS_IOB
    LDA     #>DOS_IOB
    JSR     INDIRECT_RWTS
    BCC     .no_error
    JMP     RESET_GAME

.no_error:
    LDY     #$0A
    LDA     #$00
    STA     MASK0       ; temp storage

.loop:
    LDA     HI_SCORE_DATA+244,Y
    EOR     HI_SCORE_DATA_MARKER,Y
    ORA     MASK0
    STA     MASK0
    DEY
    BPL     .loop

    LDA     MASK0
    BEQ     .all_zero_data

    LDA     #$00
    RTS

.all_zero_data:
    LDA     #$01
    LDX     $1FFF
    BNE     .end
    LDA     #$FF

.end:
    RTS
```

Defines:

ACCESS_HI_SCORE_DATA_FROM_DISK, used in chunks 122d, 185, 190a, 192, 195, and 197.
Uses DOS_IOB 181, HI_SCORE_DATA 112a, HI_SCORE_DATA_MARKER 182, INDIRECT_RWTS 196b, IOB_COMMAND_CODE 181, IOB_READ_WRITE_BUFFER_PTR 181, IOB_SECTOR_NUMBER 181, IOB_TRACK_NUMBER 181, IOB_VOLUME_NUMBER_EXPECTED 181, and MASK0 32a.

RECORD_HI_SCORE_DATA_TO_DISK records the player's score to disk if the player's score belongs on the high score list. It also handles getting the player's initials.

184 ⟨*defines* 3⟩+≡ (215) ◁181 190b▷

```
HIGH_SCORE_INITIALS_INDEX        EQU      $824D
```

Defines:
HIGH_SCORE_INITIALS_INDEX, used in chunk 185.

185 ⟨*record hi score data* 185⟩≡ (213)
```
    ORG     $84C8
RECORD_HI_SCORE_DATA_TO_DISK:
    SUBROUTINE

    LDA     $9D
    BEQ     .end

    LDA     SCORE
    ORA     SCORE+1
    ORA     SCORE+2
    ORA     SCORE+3
    BEQ     .end

    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
    ; Return value of 0 means the hi score marker wasn't present,
    ; so don't write the hi score table.
    BEQ     .end

    LDY     #$01
.loop:
    LDX     HI_SCORE_TABLE_OFFSETS,Y
    LDA     LEVELNUM
    CMP     HI_SCORE_DATA+3,X       ; level
    BCC     .next
    BNE     .record_it

    LDA     SCORE+3
    CMP     HI_SCORE_DATA+4
    BCC     .next
    BNE     .record_it

    LDA     SCORE+2
    CMP     HI_SCORE_DATA+5
    BCC     .next
    BNE     .record_it

    LDA     SCORE+1
    CMP     HI_SCORE_DATA+6
    BCC     .next
    BNE     .record_it

    LDA     SCORE
    CMP     HI_SCORE_DATA+7
    BCC     .next
    BNE     .record_it

.next:
    INY
```

```
    CPY     #$0B
    BCC     .loop

.end:
    RTS

.record_it:
    CPY     #$0A
    BEQ     .write_here
    STY     $56

    ; Move the table rows to make room at index $56
    LDY     #$09
.loop2:
    LDX     HI_SCORE_TABLE_OFFSETS,Y

    ; Move 8 bytes of hi score data
    LDA     #$08
    STA     ROW_COUNT       ; temporary counter
.loop3:
    LDA     HI_SCORE_DATA,X
    STA     HI_SCORE_DATA+8,X
    INX
    DEC     ROW_COUNT
    BNE     .loop3

    CPY     $56
    BEQ     .write_here
    DEY
    BNE     .loop2

.write_here:
    LDX     HI_SCORE_TABLE_OFFSETS,Y
    LDA     #$A0
    STA     HI_SCORE_DATA,X
    STA     HI_SCORE_DATA+1,X
    STA     HI_SCORE_DATA+2,X
    LDA     LEVELNUM
    STA     HI_SCORE_DATA+3,X
    LDA     SCORE+3
    STA     HI_SCORE_DATA+4,X
    LDA     SCORE+2
    STA     HI_SCORE_DATA+5,X
    LDA     SCORE+1
    STA     HI_SCORE_DATA+6,X
    LDA     SCORE
    STA     HI_SCORE_DATA+7,X
    STY     WIPE0               ; temporary
    LDA     HI_SCORE_TABLE_OFFSETS,Y
    STA     .rd_loc+1
```

```
    STA     .wr_loc+1
    JSR     HI_SCORE_SCREEN

    LDA     #$40
    STA     DRAW_PAGE
    LDA     WIPE0
    CLC
    ADC     #$04
    STA     GAME_ROWNUM
    LDA     #$07
    STA     GAME_COLNUM

    LDX     #$00
    STX     HIGH_SCORE_INITIALS_INDEX
.get_initial_from_player:
    LDX     HIGH_SCORE_INITIALS_INDEX
.rd_loc:
    LDA     HI_SCORE_DATA,X     ; fixed up to add offset from above
    JSR     CHAR_TO_SPRITE_NUM
    JSR     WAIT_FOR_KEY
    STA     KBDSTRB
    CMP     #$8D
    BEQ     .return_pressed
    CMP     #$88                ; backspace/back arrow
    BNE     .other_key_pressed

    ; backspace pressed
    LDX     KBD_ENTRY_INDEX
    BEQ     .beep       ; can't backspace/back arrow past the beginning

    DEC     HIGH_SCORE_INITIALS_INDEX
    DEC     GAME_COLNUM
    JMP     .get_initial_from_player

.other_key_pressed:
    CMP     #$95            ; fwd arrow
    BNE     .check_for_allowed_chars
    LDX     KBD_ENTRY_INDEX
    CPX     #$02
    BEQ     .beep       ; can't fwd arrow past the end

    INC     GAME_COLNUM
    INC     KBD_ENTRY_INDEX
    JMP     .get_initial_from_player

.check_for_allowed_chars
    CMP     #$AE        ; period allowed
    BEQ     .put_char
    CMP     #$A0        ; space allowed
    BEQ     .put_char
```

```
    CMP     #$C1
    BCC     .beep       ; can't be less than 'A'
    CMP     #$DB
    BCS     .beep       ; can't be greater than 'Z'

.put_char
    LDY     KBD_ENTRY_INDEX
.wr_loc:
    STA     HI_SCORE_DATA,Y     ; fixed up to add offset from above
    JSR     PUT_CHAR
    INC     KBD_ENTRY_INDEX
    LDA     KBD_ENTRY_INDEX
    CMP     #$03
    BCC     .get_initial_from_player

.beep:
    JSR     BEEP
    JMP     .get_initial_from_player

.return_pressed:
    LDA     #$20
    STA     DRAW_PAGE
    LDA     #$02
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; write hi score table
    JMP     $618E

    ORG     $824C
KBD_ENTRY_INDEX:
    HEX     60
```

Defines:
KBD_ENTRY_INDEX, used in chunk 71.
RECORD_HI_SCORE_DATA_TO_DISK, used in chunk 200.

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183, BEEP 54, CHAR_TO_SPRITE_NUM 42, DRAW_PAGE 43, GAME_COLNUM 32a, GAME_ROWNUM 32a, HI_SCORE_DATA 112a, HI_SCORE_SCREEN 112b, HI_SCORE_TABLE_OFFSETS 114a, HIGH_SCORE_INITIALS_INDEX 184, KBDSTRB 120b, LEVELNUM 50, PUT_CHAR 44a, ROW_COUNT 23c, SCORE 48b, WAIT_FOR_KEY 67, and WIPE0 89.

189a ⟨*bad data disk* 189a⟩≡ (213)

```
      ORG     $8106
  BAD_DATA_DISK:
      SUBROUTINE

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

      ; "DISKETTE IN DRIVE IS NOT A\r"
      ; "LODE RUNNER DATA DISK."
      JSR     PUT_STRING
      HEX     C4 C9 D3 CB C5 D4 D4 C5 A0 C9 CE A0 C4 D2 C9 D6
      HEX     C5 A0 C9 D3 A0 CE CF D4 A0 C1 8D CC CF C4 C5 A0
      HEX     D2 D5 CE CE C5 D2 A0 C4 C1 D4 C1 A0 C4 C9 D3 CB
      HEX     AE 00

      JMP     HIT_KEY_TO_CONTINUE
```

Defines:
BAD_DATA_DISK, used in chunks 190a, 195, and 197.
Uses CLEAR_HGR2 4, DRAW_PAGE 43, GAME_COLNUM 32a, GAME_ROWNUM 32a, HIT_KEY_TO_CONTINUE 70a, and PUT_STRING 45.

189b ⟨*dont manipulate master disk* 189b⟩≡ (213)

```
      ORG     $8098
  DONT_MANIPULATE_MASTER_DISK:
      SUBROUTINE

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

      ; "USER NOT ALLOWED TO\r"
      ; "MANIPULATE MASTER DISKETTE."
      JSR     PUT_STRING
      HEX     D5 D3 C5 D2 A0 CE CF D4 A0 C1 CC CC CF D7 C5 C4
      HEX     A0 D4 CF 8D CD C1 CE C9 D0 D5 CC C1 D4 C5 A0 CD
      HEX     C1 D3 D4 C5 D2 A0 C4 C9 D3 CB C5 D4 D4 C5 AE 00

      ; fallthrough to HIT_KEY_TO_CONTINUE
```

Defines:
DONT_MANIPULATE_MASTER_DISK, used in chunk 190a.
Uses CLEAR_HGR2 4, DRAW_PAGE 43, GAME_COLNUM 32a, GAME_ROWNUM 32a, HIT_KEY_TO_CONTINUE 70a, and PUT_STRING 45.

The level editor has a routine to check for a valid data disk, meaning it has a high score table and is not the master disk. In case of a disk that is not a valid data disk, we abort the current editor operation, dumping the user right into the level editor by jumping to START_LEVEL_EDITOR. Otherwise we jump to RETURN_FROM_SUBROUTINE, which apparently saved a byte over having a local RTS instruction.

190a ⟨*check for valid data disk* 190a⟩≡ (213)

```
    ORG     $807F
CHECK_FOR_VALID_DATA_DISK:
    SUBROUTINE

    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
    CMP     #$00        ; bad table
    BNE     .check_for_master_disk

    JSR     BAD_DATA_DISK
    JMP     START_LEVEL_EDITOR

.check_for_master_disk:
    CMP     #$01        ; master disk
    BNE     RETURN_FROM_SUBROUTINE

    JSR     DONT_MANIPULATE_MASTER_DISK
    JMP     START_LEVEL_EDITOR
```

Defines:
CHECK_FOR_VALID_DATA_DISK, used in chunks 210a and 211.
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183, BAD_DATA_DISK 189a, DONT_MANIPULATE_MASTER_DISK 189b, RETURN_FROM_SUBROUTINE 70a, and START_LEVEL_EDITOR 208.

Initializing a disk first DOS formats it. This zeros out all data on all tracks and sectors. Once that's done, we write track 0 sector 0 with the data from DISK_BOOT_SECTOR_DATA. Then we read the Volume Table of Contents (VTOC) at track 17 sector 0, which will contain all zeros because of the initial format. We then stick SAVED_VTOC_DATA in the disk buffer and write it to the VTOC. We do the same thing with the catalog sector at track 17 sector 15 and SAVED_FILE_DESCRIPTIVE_ENTRY_DATA.

The final step is to create a blank sector at track 12 sector 15, with the special "LODE RUNNER" marker HI_SCORE_DATA_MARKER near the end.

190b ⟨*defines* 3⟩+≡ (215) ◁184 196c▷

```
DISK_BOOT_SECTOR_DATA   EQU     $1DB2       ; 256 bytes
```

Defines:
DISK_BOOT_SECTOR_DATA, used in chunk 192.

191 ⟨*tables 7*⟩+≡ (215) ◁182 198▷

```
    ORG     $8250
SAVED_VTOC_DATA:
    HEX     60 02 11 0F 04 00 00 FE 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    HEX     00 00 00 00 00 00 00 00 7A 00 00 00 00 00 00 00
    HEX     00 FF FF 00 00 23 0F 00

    ORG     $8289
SAVED_FILE_DESCRIPTIVE_ENTRY_DATA:
    HEX     22 ; Track of first track/sector list sector (T34)
    HEX     0F ; Sector of first track/sector list sector (S15)
    HEX     88 ; File type and flags: locked, S-type file
    ; File name: "^H^H^H^H^H^HLODE RUNNER DATA DISK  "
    HEX     88 88 88 88 88 88 88 CC CF C4 C5 A0 D2 D5 CE CE
    HEX     C5 D2 A0 C4 C1 D4 C1 A0 C4 C9 D3 CB A0 A0
```

Defines:
`SAVED_FILE_DESCRIPTIVE_ENTRY_DATA`, used in chunk 192.
`SAVED_VTOC_DATA`, used in chunks 192 and 211.

192 ⟨*editor initialize disk* 192⟩≡ (213)

```
      ORG     $7D5D
  EDITOR_INITIALIZE_DISK:
      SUBROUTINE

      ; "\r"
      ; ">>INITIALIZE\r"
      ; "  THIS FORMATS THE DISKETTE\r"
      ; "  FOR USER CREATED LEVELS.\r"
      ; "  (CAUTION. IT ERASES THE\r"
      ; "   ENTIRE DISKETTE FIRST)\r"
      ; "\r"
      ; "  ARE YOU SURE (Y/N) "
      JSR     PUT_STRING
      HEX     8D BE BE C9 CE C9 D4 C9 C1 CC C9 DA C5 8D A0 A0
      HEX     D4 C8 C9 D3 A0 C6 CF D2 CD C1 D4 D3 A0 D4 C8 C5
      HEX     A0 C4 C9 D3 CB C5 D4 D4 C5 8D A0 A0 C6 CF D2 A0
      HEX     D5 D3 C5 D2 A0 C3 D2 C5 C1 D4 C5 C4 A0 CC C5 D6
      HEX     C5 CC D3 AE 8D A0 A0 A8 C3 C1 D5 D4 C9 CF CE AE
      HEX     A0 C9 D4 A0 C5 D2 C1 D3 C5 D3 A0 D4 C8 C5 8D A0
      HEX     A0 A0 C5 CE D4 C9 D2 C5 A0 C4 C9 D3 CB C5 D4 D4
      HEX     C5 A0 C6 C9 D2 D3 D4 A9 8D 8D A0 A0 C1 D2 C5 A0
      HEX     D9 CF D5 A0 D3 D5 D2 C5 A0 A8 D9 AF CE A9 A0 00

      JSR     EDITOR_WAIT_FOR_KEY
      CMP     #$D9        ; Y
      BNE     .end

      NOP     ; NOP x 15
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP

      LDA     DISK_LEVEL_LOC
      PHA

      ; Format the disk
      LDA     #$04
      JSR     LOAD_COMPRESSED_LEVEL_DATA
```

```
    ; Write the boot sector (T0S0)
    LDA     #<DISK_BOOT_SECTOR_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR
    LDA     #>DISK_BOOT_SECTOR_DATA
    STA     IOB_READ_WRITE_BUFFER_PTR+1
    LDA     #$00
    STA     IOB_TRACK_NUMBER
    STA     IOB_SECTOR_NUMBER
    LDA     #$02
    STA     IOB_COMMAND_CODE
    JSR     ACCESS_DISK_OR_RESET_GAME   ; write T0S0 with DISK_BOOT_SECTOR_DATA.

    ; Read the VTOC (T17S0)
    LDA     #$E0
    STA     DISK_LEVEL_LOC              ; ends up being T17S0 (the VTOC)
    LDA     #$01
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    ; Copy from SAVED_VTOC_DATA to DISK_BUFFER and write it.
    LDY     #$37
.loop:
    LDA     SAVED_VTOC_DATA+1,Y
    STA     DISK_BUFFER,Y
    DEY
    BPL     .loop

    LDA     #$02
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    ; Read the first catalog sector (T17S15)
    LDA     #$EF
    STA     DISK_LEVEL_LOC
    LDA     #$01
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    ; Copy from SAVED_FILE_DESCRIPTIVE_ENTRY_DATA the first file descriptive
    ; entry to DISK_BUFFER and write it.
    LDY     #$20
.loop2:
    LDA     SAVED_FILE_DESCRIPTIVE_ENTRY_DATA,Y
    STA     DISK_BUFFER+11,Y
    DEY
    BPL     .loop2

    ; Write it back
    LDA     #$02
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    ; Read the high score sector
```

```
    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK

    ; Copy from HI_SCORE_DATA_MARKER and write it.
    LDY     #$0A
.loop3:
    LDA     HI_SCORE_DATA_MARKER,Y
    STA     $1FF4,Y
    DEY
    BPL     .loop3

    ; Write it back
    LDA     #$02
    JSR     LOAD_COMPRESSED_LEVEL_DATA

    PLA
    STA     DISK_LEVEL_LOC
.end:
    JMP     EDITOR_COMMAND_LOOP
```

Defines:
EDITOR_INITIALIZE_DISK, used in chunk 207b.
Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183, DISK_BOOT_SECTOR_DATA 190b, EDITOR_COMMAND_LOOP 208, EDITOR_WAIT_FOR_KEY 69, HI_SCORE_DATA_MARKER 182, IOB_COMMAND_CODE 181, IOB_READ_WRITE_BUFFER_PTR 181, IOB_SECTOR_NUMBER 181, IOB_TRACK_NUMBER 181, PUT_STRING 45, SAVED_FILE_DESCRIPTIVE_ENTRY_DATA 191, and SAVED_VTOC_DATA 191.

To clear the high score table from a disk, we first read the sector where the high score table is supposed to be, and check to see if the buffer is a good high score table. If so, we zero out the first 80 bytes (the 10 high score entries) and write that back to disk.

If the disk didn't contain a good high score table, we display the BAD_DATA_DISK message and abort.

195 ⟨*editor clear high scores* 195⟩≡ (213)

```
    ORG     $7E75
EDITOR_CLEAR_HIGH_SCORES:
    SUBROUTINE

    ; "\r"
    ; ">>CLEAR SCORE FILE\r"
    ; "  THIS CLEARS THE HIGH\r"
    ; "  SCORE FILE OF ALL\r"
    ; "  ENTRIES.\r"
    ; "\r"
    ; "  ARE YOU SURE (Y/N) "
    JSR     PUT_STRING
    HEX     8D BE BE C3 CC C5 C1 D2 A0 D3 C3 CF D2 C5 A0 C6
    HEX     C9 CC C5 8D A0 A0 D4 C8 C9 D3 A0 C3 CC C5 C1 D2
    HEX     D3 A0 D4 C8 C5 A0 C8 C9 C7 C8 8D A0 A0 D3 C3 CF
    HEX     D2 C5 A0 C6 C9 CC C5 A0 CF C6 A0 C1 CC CC 8D A0
    HEX     A0 C5 CE D4 D2 C9 C5 D3 AE 8D 8D A0 A0 C1 D2 C5
    HEX     A0 D9 CF D5 A0 D3 D5 D2 C5 A0 A8 D9 AF CE A9 A0
    HEX     00

    JSR     EDITOR_WAIT_FOR_KEY
    CMP     #$D9        ; 'Y'
    BNE     .end

    LDA     #$01
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; read table
    CMP     #$00
    BNE     .good_disk
    JSR     BAD_DATA_DISK
    JMP     START_LEVEL_EDITOR

.good_disk:
    LDY     #$4F
    LDA     #$00

.loop:
    STA     HI_SCORE_DATA,Y
    DEY
    BPL     .loop

    LDA     #$02
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; write table
```

```
.end:
    JMP     EDITOR_WAIT_FOR_KEY
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183, BAD_DATA_DISK 189a, EDITOR_WAIT_FOR_KEY 69, HI_SCORE_DATA 112a, PUT_STRING 45, SCORE 48b, and START_LEVEL_EDITOR 208.

9.1 Initialization

196a ⟨*rwts targets* 196a⟩≡ (213)

```
    ORG     $0036
INDIRECT_TARGET:
    WORD    DEFAULT_INDIRECT_TARGET
DISABLE_INTS_CALL_RWTS_PTR:
    WORD    DISABLE_INTS_CALL_RWTS

DISABLE_INTS_CALL_RWTS      EQU     $B7B5
```

Defines:
DISABLE_INTS_CALL_RWTS, used in chunk 196b.
DISABLE_INTS_CALL_RWTS_PTR, used in chunk 197.
INDIRECT_TARGET, used in chunks 196b, 197, and 208.

196b ⟨*indirect call* 196b⟩≡ (213)

```
    ORG     $63A5
INDIRECT_RWTS:
    SUBROUTINE
    JMP     (INDIRECT_TARGET)

    ORG     $8E50
DEFAULT_INDIRECT_TARGET:
    SUBROUTINE
    JMP     DISABLE_INTS_CALL_RWTS
```

Defines:
INDIRECT_RWTS, used in chunk 183.
Uses DISABLE_INTS_CALL_RWTS 196a and INDIRECT_TARGET 196a.

196c ⟨*defines* 3⟩+≡ (215) ◁190b 207a▷

```
GUARD_PATTERN_OFFSET        EQU     $97
```

Defines:
GUARD_PATTERN_OFFSET, used in chunks 111b, 197, and 199.

197 ⟨*Initialize game data* 197⟩≡ (213)
```
    ORG     $6056

.init_game_data:
    LDA     #0
    STA     SCORE
    STA     SCORE+1
    STA     SCORE+2
    STA     SCORE+3
    STA     GUARD_PATTERN_OFFSET
    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 or splash screen),
    ; play the game.
    BEQ     .put_status_and_start_game

    ; We were displaying the high score screen or splash screen
    LDA     #1
    JSR     ACCESS_HI_SCORE_DATA_FROM_DISK      ; Read hi score data
    CMP     #$00
    BNE     .set_rwts_target
    JSR     BAD_DATA_DISK
    JMP     RESET_GAME

.set_rwts_target:
    LDA     $1FFF
    BNE     .use_dos_target
    LDA     INDIRECT_TARGET
    LDX     INDIRECT_TARGET+1
    BNE     .store_rwts_addr

.use_dos_target:
    LDA     DISABLE_INTS_CALL_RWTS_PTR
    LDX     DISABLE_INTS_CALL_RWTS_PTR+1

.store_rwts_addr:
    STA     RWTS_ADDR
    STX     RWTS_ADDR+1

.put_status_and_start_game:
    JSR     PUT_STATUS
    STA     TXTPAGE1
```

Uses ACCESS_HI_SCORE_DATA_FROM_DISK 183, BAD_DATA_DISK 189a, DISABLE_INTS_CALL_RWTS_PTR 196a, GUARD_PATTERN_OFFSET 196c, INDIRECT_TARGET 196a, LIVES 50, PREGAME_MODE 104a, PUT_STATUS 51, SCORE 48b, TXTPAGE1 119b, and WIPE_MODE 85.

198 ⟨*tables* 7⟩+≡ (215) ◁191 201▷

```
      ORG     $6CA7
GUARD_PATTERNS_LIST:
      HEX     00 01 01
      HEX     01 01 01
      HEX     01 03 01
      HEX     01 03 03
      HEX     03 03 03
      HEX     03 03 07
      HEX     03 07 07
      HEX     07 07 07
      HEX     07 07 0F
      HEX     07 0F 0F
      HEX     0F 0F 0F
```

Defines:
GUARD_PATTERNS_LIST, used in chunk 199.

199 ⟨*start game* 199⟩≡ (213)

```
      ORG     $609F

  .start_game:
      LDX     #$01
      JSR     LOAD_LEVEL
      LDA     #$00
      STA     KEY_COMMAND
      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     DIG_DIRECTION
      STX     NOTE_INDEX

      LDA     GUARD_PATTERN_OFFSET
      CLC
      ADC     GUARD_COUNT         ; GUARD_COUNT + $97 can't be greater than 8.
      TAY
      LDX     TIMES_3_TABLE,Y     ; X = 3 * Y (goes up to Y=8)
      LDA     GUARD_PATTERNS_LIST,X
      STA     GUARD_PATTERNS
      LDA     GUARD_PATTERNS_LIST+1,X
      STA     GUARD_PATTERNS+1
      LDA     GUARD_PATTERNS_LIST+2,X
      STA     GUARD_PATTERNS+2

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

Uses DIG_DIRECTION 155, GAME_COLNUM 32a, GAME_ROWNUM 32a, GUARD_COUNT 78d, GUARD_PATTERN_OFFSET 196c, GUARD_PATTERNS 171c, GUARD_PATTERNS_LIST 198, KEY_COMMAND 128a, LOAD_LEVEL 107a, NOTE_INDEX 55, PLAYER_COL 77c, PLAYER_ROW 77c, PREGAME_MODE 104a, and TIMES_3_TABLE 201.

200 ⟨*game loop* 200⟩≡ (213)

```
    ORG     $60E4
.game_loop:
    JSR     MOVE_PLAYER
    LDA     ALIVE
    BEQ     .died

    JSR     PLAY_SOUND

    LDA     GOLD_COUNT
    BNE     .still_gold_present
    JSR     $8631

.still_gold_present:
    LDA     PLAYER_ROW
    BNE     .not_at_top
    LDA     PLAYER_Y_ADJ
    CMP     #$02
    BNE     .not_at_top

    ; Reached top of screen
    LDA     GOLD_COUNT
    BEQ     .level_cleared
    CMP     #$FF
    BEQ     .level_cleared      ; level cleared if GOLD_COUNT == 0 or -1.

.not_at_top:
    JSR     $75F4
    LDA     ALIVE
    BEQ     .died
    JSR     PLAY_SOUND
    JSR     MOVE_GUARDS
    LDA     ALIVE
    BEQ     .died
    BNE     .game_loop

.level_cleared:
    INC     LEVELNUM
    INC     DISK_LEVEL_LOC
    INC     LIVES
    BNE     .lives_incremented
    DEC     LIVES               ; LIVES doesn't overflow.

.lives_incremented:
    ; Increment score by 1500, playing an ascending tune while doing so.
    LDX     #$0F
    STX     SCRATCH_5C

.loop2:
    LDY     #$01
```

```
    LDA     #$00
    JSR     ADD_AND_UPDATE_SCORE    ; SCORE += 100
    JSR     APPEND_LEVEL_CLEARED_NOTE
    JSR     APPEND_LEVEL_CLEARED_NOTE
    JSR     APPEND_LEVEL_CLEARED_NOTE
    DEC     SCRATCH_5C
    BNE     .loop2

.start_game_:
    JMP     .start_game

.died:
    DEC     LIVES
    JSR     PUT_STATUS_LIVES
    JSR     LOAD_SOUND_DATA
    HEX     02 40 02 40 03 50 03 50 04 60 04 60 05 70 05 70
    HEX     06 80 06 80 07 90 07 90 08 A0 08 A0 09 B0 09 B0
    HEX     0A C0 0A C0 0B D0 0B D0 0C E0 0C E0 0D F0 0D F0
    HEX     00

.play_died_tune:
    JSR     PLAY_SOUND
    BCS     .play_died_tune

    LDA     PREGAME_MODE
    LSR
    BEQ     .restore_enable_sound     ; If PREGAME_MODE is 0 or 1

    LDA     LIVES
    BNE     .start_game_    ; We can still play.

    ; Game over
    JSR     RECORD_HI_SCORE_DATA_TO_DISK
    JSR     SPINNING_GAME_OVER
    BCS     .key_pressed
```

Uses ADD_AND_UPDATE_SCORE 49, ALIVE 106d, APPEND_LEVEL_CLEARED_NOTE 61b, GOLD_COUNT 78d, LEVELNUM 50, LIVES 50, LOAD_SOUND_DATA 56, MOVE_GUARDS 172a, MOVE_PLAYER 164, PLAY_SOUND 60, PLAYER_ROW 77c, PLAYER_Y_ADJ 81b, PREGAME_MODE 104a, PUT_STATUS_LIVES 51, RECORD_HI_SCORE_DATA_TO_DISK 185, SCORE 48b, and SCRATCH_5C 61a.

201 ⟨*tables* 7⟩+≡ (215) ◁198 202▷

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

Defines:
TIMES_3_TABLE, used in chunk 199.

202 ⟨*tables* 7⟩+≡ (215) ◁201 207b▷

```
    ORG     $8C35
TABLE0:
    HEX     80 80 80 80 80 80 80 80 80 80 80 80 80 80
TABLE1:
    HEX     C0 AA D5 AA D5 AA D5 AA D5 AA D5 AA D5 80
TABLE2:
    HEX     90 80 80 80 80 80 80 80 80 80 80 80 80 82
TABLE3:
    HEX     90 AA D1 A2 D5 A8 85 A8 C5 A2 D4 A2 95 82
TABLE4:
    HEX     90 82 91 A2 C5 A8 80 88 C5 A2 94 A0 90 82
TABLE5:
    HEX     90 82 90 A2 C4 A8 80 88 C5 A2 94 A0 90 82
TABLE6:
    HEX     90 82 90 A2 C4 A8 81 88 C4 A2 D4 A0 95 82
TABLE7:
    HEX     90 A2 D1 A2 C4 88 80 88 C4 A2 84 A0 85 82
TABLE8:
    HEX     90 82 91 A2 C4 88 80 88 C4 AA 84 A0 85 82
TABLE9:
    HEX     90 82 91 A2 C4 88 80 88 C4 8A 84 A0 91 82
TABLE10:
    HEX     90 AA 91 A2 C4 A8 85 A8 85 82 D4 A2 91 82

    ORG     $8CCF
ADDRESS_TABLE:
    WORD    TABLE0-14
    WORD    TABLE1-14
    WORD    TABLE2-14
    WORD    TABLE3-14
    WORD    TABLE4-14
    WORD    TABLE5-14
    WORD    TABLE6-14
    WORD    TABLE7-14
    WORD    TABLE8-14
    WORD    TABLE9-14
    WORD    TABLE10-14
```

Defines:
`ADDRESS_TABLE`, used in chunk 205.

203 ⟨*anims* 203⟩≡ (213)

```
    ORG     $8B1A
SPINNING_GAME_OVER:
    SUBROUTINE

    LDA     #$01
    STA     ANIM_COUNT
    LDA     #$20
    STA     HGR_PAGE

.loop:
    JSR     ANIM5
    JSR     ANIM4
    JSR     ANIM3
    JSR     ANIM2
    JSR     ANIM1
    JSR     ANIM0
    JSR     ANIM1
    JSR     ANIM2
    JSR     ANIM3
    JSR     ANIM4
    JSR     ANIM5
    JSR     ANIM10
    JSR     ANIM9
    JSR     ANIM8
    JSR     ANIM7
    JSR     ANIM6
    JSR     ANIM7
    JSR     ANIM8
    JSR     ANIM9
    JSR     ANIM10
    LDA     ANIM_COUNT
    CMP     #100
    BCC     .loop

    JSR     ANIM5
    JSR     ANIM4
    JSR     ANIM3
    JSR     ANIM2
    JSR     ANIM1
    JSR     ANIM0
    CLC
    RTS

    ORG     $8B7A
ANIM0:
    JSR     SHOW_ANIM_LINE
    HEX     00 01 02 03 04 05 06 07 08 09 0A 02 01 00
ANIM1:
    JSR     SHOW_ANIM_LINE
```

```
    HEX     00 00 01 02 03 04 05 07 09 0A 02 01 00 00
ANIM2:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 01 02 03 04 09 0A 02 01 00 00 00
ANIM3:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 01 02 03 0A 02 01 00 00 00 00
ANIM4:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 01 03 0A 01 00 00 00 00 00
ANIM5:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 00 01 01 00 00 00 00 00 00
ANIM6:
    JSR     SHOW_ANIM_LINE
    HEX     00 01 02 0A 09 08 07 06 05 04 03 02 01 00
ANIM7:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 01 02 0A 09 07 05 04 03 02 01 00 00
ANIM8:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 01 02 0A 09 04 03 02 01 00 00 00
ANIM9:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 01 02 0A 03 02 01 00 00 00 00
ANIM10:
    JSR     SHOW_ANIM_LINE
    HEX     00 00 00 00 00 01 0A 03 01 00 00 00 00 00
```

Uses ANIM_COUNT 205, HGR_PAGE 26b, and SHOW_ANIM_LINE 205.

205 ⟨*show anim line* 205⟩≡ (213)

```
    ORG     $8CE5
SHOW_ANIM_LINE:
    SUBROUTINE

    PLA
    STA     TMP_PTR
    PLA
    STA     TMP_PTR+1           ; store "return" addr

    ; Fill 14 rows of pixel data from row 0x51 (81) through 0x5E (94).
    LDY     #$50
    STY     GAME_ROWNUM
    BNE     .next      ; unconditional

.loop:
    JSR     ROW_TO_ADDR
    LDY     #$00
    LDA     (TMP_PTR),Y
    ASL
    LDA     ADDRESS_TABLE,X
    STA     .loop2+1
    LDA     ADDRESS_TABLE+1,X   ; groups of 14 bytes
    STA     .loop2+2
    LDY     #$0D

    ; Copy 13 bytes of pixel data onto screen from
    ; addr+14 to addr+26
.loop2:
    LDA     $8D08,Y             ; fixed up from above
    STA     (ROW_ADDR),Y        ; pixel data
    INY
    CPY     #$1B
    BCC     .loop2              ; Y < 27

    ; Next row
.next:
    JSR     INCREMENT_TMP_PTR
    INC     GAME_ROWNUM
    LDY     GAME_ROWNUM
    CPY     #$5F
    BCC     .loop

    LDX     ANIM_COUNT
    LDY     #$FF
.delay:
    DEY
    BNE     .delay
    DEX
    BNE     .delay
```

```
    INC     ANIM_COUNT

    LDA     INPUT_MODE
    CMP     KEYBOARD_MODE
    BEQ     .check_for_keypress
    LDA     BUTN1
    BMI     .input_detected
    LDA     BUTN0
    BMI     .input_detected

.check_for_keypress:
    LDA     KBD
    BMI     .input_detected
    RTS

    ; Skip the rest of the big animation.
.input_detected:
    PLA
    PLA
    SEC
    LDA     KBD
    STA     KBDSTRB
    RTS

ANIM_COUNT:
    HEX     9D

    ORG     $8D4C
INCREMENT_TMP_PTR:
    SUBROUTINE

    INC     TMP_PTR
    BNE     .end
    INC     TMP_PTR+1
.end:
    RTS
```

Defines:
ANIM_COUNT, used in chunk 203.
INCREMENT_TMP_PTR, never used.
SHOW_ANIM_LINE, used in chunk 203.
Uses ADDRESS_TABLE 202, BUTN0 64, BUTN1 64, GAME_ROWNUM 32a, INPUT_MODE 64, KBD 121b, KBDSTRB 120b, ROW_ADDR 26b, ROW_TO_ADDR 26c, and TMP_PTR 3.

Chapter 10

Level editor

207a ⟨*defines* 3⟩+≡ (215) ◁196c 210b▷

```
    ORG     $7C77
SAVED_INPUT_MODE:
    HEX     00

    ORG     $7C54
EDITOR_RETURN_ADDRESS:
    HEX     5F 7C
```

Defines:
SAVED_INPUT_MODE, used in chunk 208.

207b ⟨*tables* 7⟩+≡ (215) ◁202

```
    ORG     $7C4D
EDITOR_KEYS:
    ; P (Play level)
    ; C (Clear level)
    ; E (Edit level)
    ; M (Move level)
    ; I (Initialize disk)
    ; S (clear high Scores)
    HEX     D0 C3 C5 CD C9 D3 00    ; P C E M I S
EDITOR_ROUTINE_ADDRESS:
    WORD    EDITOR_PLAY_LEVEL-1
    WORD    EDITOR_CLEAR_LEVEL-1
    WORD    EDITOR_EDIT_LEVEL-1
    WORD    EDITOR_MOVE_LEVEL-1
    WORD    EDITOR_INITIALIZE_DISK-1
    WORD    EDITOR_CLEAR_HIGH_SCORES-1
```

Defines:
EDITOR_KEYS, used in chunk 208.
EDITOR_ROUTINE_ADDRESS, never used.
Uses EDITOR_CLEAR_LEVEL 210a, EDITOR_INITIALIZE_DISK 192, and EDITOR_MOVE_LEVEL 211.

208 ⟨*level editor* 208⟩≡ (213)

```
    ORG     $7B84
LEVEL_EDITOR:
    SUBROUTINE

    LDA     #$00
    STA     SCORE
    STA     SCORE+1
    STA     SCORE+2
    STA     SCORE+3

    LDA     INDIRECT_TARGET
    STA     RWTS_ADDR
    LDA     INDIRECT_TARGET+1
    STA     RWTS_ADDR+1

    LDA     #$05
    STA     LIVES
    STA     PREGAME_MODE
    LDA     INPUT_MODE
    STA     SAVED_INPUT_MODE

    STA     TXTPAGE1

    LDA     DISK_LEVEL_LOC
    CMP     #$96
    BCC     START_LEVEL_EDITOR
    LDA     #$00
    STA     DISK_LEVEL_LOC

START_LEVEL_EDITOR:
    JSR     CLEAR_HGR1
    LDA     #$20
    STA     DRAW_PAGE
    LDA     #$00
    STA     GAME_COLNUM
    STA     GAME_ROWNUM

    ; "  LODE RUNNER BOARD EDITOR\r
    ; "----------------------------\r
    ; "  <ESC> ABORTS ANY COMMAND\r"
    JSR     PUT_STRING
    HEX     A0 A0 CC CF C4 C5 A0 D2 D5 CE CE C5 D2 A0 C2 CF
    HEX     C1 D2 C4 A0 C5 C4 C9 D4 CF D2 8D AD AD AD AD AD
    HEX     AD AD AD AD AD AD AD AD AD AD AD AD AD AD AD AD
    HEX     AD AD AD AD AD AD AD 8D A0 A0 BC C5 D3 C3 BE A0
    HEX     C1 C2 CF D2 D4 D3 A0 C1 CE D9 A0 C3 CF CD CD C1
    HEX     CE C4 8D 00

EDITOR_COMMAND_LOOP:
```

```
    LDA     GAME_ROWNUM
    CMP     #$09
    BCS     START_LEVEL_EDITOR

    ; "\r"
    ; "COMMAND>"
    JSR     PUT_STRING
    HEX     8D C3 CF CD CD C1 CE C4 BE 00

    JSR     EDITOR_WAIT_FOR_KEY
    LDX     #$00

.loop2:
    LDY     EDITOR_KEYS,X
    BEQ     .beep
    CMP     EDITOR_KEYS,X
    BEQ     .end
    INX
    BNE     .loop2

.beep:
    JSR     BEEP
    JMP     EDITOR_COMMAND_LOOP

.end:
    TXA
    ASL
    TAX
    LDA     EDITOR_RETURN_ADDRESS+1,X
    PHA
    LDA     EDITOR_RETURN_ADDRESS,X
    PHA
    RTS
```

Defines:
EDITOR_COMMAND_LOOP, used in chunks 69, 71, 192, 210a, and 211.
LEVEL_EDITOR, never used.
START_LEVEL_EDITOR, used in chunks 122c, 190a, and 195.

Uses BEEP 54, CLEAR_HGR1 4, DRAW_PAGE 43, EDITOR_KEYS 207b, EDITOR_WAIT_FOR_KEY 69, GAME_COLNUM 32a, GAME_ROWNUM 32a, INDIRECT_TARGET 196a, INPUT_MODE 64, LIVES 50, PREGAME_MODE 104a, PUT_STRING 45, SAVED_INPUT_MODE 207a, SCORE 48b, and TXTPAGE1 119b.

Clearing a level involves getting the target level number from the user, waiting for the user to insert a valid data disk, and then writing zeros to the target level on disk.

210a ⟨*editor clear level* 210a⟩≡ (213)

```
    ORG     $7C8E
EDITOR_CLEAR_LEVEL:
    SUBROUTINE

    ; "\r"
    ; ">>CLEAR LEVEL"
    JSR     PUT_STRING
    HEX     8D BE BE C3 CC C5 C1 D2 A0 CC C5 D6 C5 CC 00

    JSR     GET_LEVEL_FROM_KEYBOARD
    BCS     .beep
    JSR     CHECK_FOR_VALID_DATA_DISK

    LDY     #$00
    TYA
.loop:
    STA     DISK_BUFFER,Y
    INY
    BNE     .loop

    LDA     #$02
    JSR     LOAD_COMPRESSED_LEVEL_DATA      ; write level
    JMP     EDITOR_COMMAND_LOOP

.beep:
    JMP     BEEP
```

Defines:
EDITOR_CLEAR_LEVEL, used in chunk 207b.
Uses BEEP 54, CHECK_FOR_VALID_DATA_DISK 190a, EDITOR_COMMAND_LOOP 208, GET_LEVEL_FROM_KEYBOARD 71, and PUT_STRING 45.

Moving a level involves getting the source and target level numbers from the user, waiting for the user to insert the source data disk, reading the source level, waiting for the user to insert the target data disk, and then writing the current level data to the target level on disk.

210b ⟨*defines* 3⟩+≡ (215) ◁207a

```
    ORG     $824F
EDITOR_LEVEL_ENTRY:
    HEX     0F
```

Defines:
EDITOR_LEVEL_ENTRY, used in chunk 211.

211 ⟨*editor move level* 211⟩≡ (213)
```
    ORG     $7CD8
EDITOR_MOVE_LEVEL:
    SUBROUTINE

    ; "\r"
    ; ">>MOVE LEVEL"
    JSR     PUT_STRING
    HEX     8D BE BE CD CF D6 C5 A0 CC C5 D6 C5 CC 00

    JSR     GET_LEVEL_FROM_KEYBOARD
    BCS     .beep
    STY     EDITOR_LEVEL_ENTRY      ; source level

    ; " TO LEVEL"
    JSR     PUT_STRING
    HEX     A0 D4 CF A0 CC C5 D6 C5 CC 00

    JSR     GET_LEVEL_FROM_KEYBOARD
    BCS     .beep
    STY     SAVED_VTOC_DATA         ; convenient place for target level

    ; "\r"
    ; "  SOURCE DISKETTE"
    JSR     PUT_STRING
    HEX     8D A0 A0 D3 CF D5 D2 C3 C5 A0 C4 C9 D3 CB C5 D4 D4 C5 00

    JSR     EDITOR_WAIT_FOR_KEY
    ; Deny and dump user back to editor if not valid data disk
    JSR     CHECK_FOR_VALID_DATA_DISK
    LDA     EDITOR_LEVEL_ENTRY              ; source level
    STA     DISK_LEVEL_LOC
    LDA     #$01
    JSR     LOAD_COMPRESSED_LEVEL_DATA      ; read source level

    ; "\r"
    ; "  DESTINATION DISKETTE"
    JSR     PUT_STRING
    HEX     8D A0 A0 C4 C5 D3 D4 C9 CE C1 D4 C9 CF CE A0 C4 C9 D3 CB C5 D4 D4 C5 00

    JSR     EDITOR_WAIT_FOR_KEY
    ; Deny and dump user back to editor if not valid data disk
    JSR     CHECK_FOR_VALID_DATA_DISK
    LDA     SAVED_VTOC_DATA                 ; target level
    STA     DISK_LEVEL_LOC
    LDA     #$02
    JSR     LOAD_COMPRESSED_LEVEL_DATA      ; write target level
    JMP     EDITOR_COMMAND_LOOP

.beep:
```

```
    JMP     .beep
```

Defines:
EDITOR_MOVE_LEVEL, used in chunk 207b.
Uses CHECK_FOR_VALID_DATA_DISK 190a, EDITOR_COMMAND_LOOP 208, EDITOR_LEVEL_ENTRY 210b, EDITOR_WAIT_FOR_KEY 69, GET_LEVEL_FROM_KEYBOARD 71, PUT_STRING 45, and SAVED_VTOC_DATA 191.

Chapter 11

The whole thing

We then put together the entire assembly file:

213 ⟨*routines* 4⟩+≡ (215) ◃88

```
; Sprite routines

⟨erase sprite at screen coordinate 36⟩
⟨draw sprite at screen coordinate 39⟩
⟨draw player 41⟩
⟨char to sprite num 42⟩
⟨put char 44a⟩
⟨put string 45⟩
⟨put digit 46a⟩
⟨to decimal3 47⟩
⟨bcd to decimal2 48a⟩

; Screen and level routines

⟨add and update score 49⟩
⟨put status 51⟩
⟨level draw routine 74⟩
⟨set active and background row pointers PTR1 and PTR2 for Y routine 76c⟩
⟨splash screen 117⟩
⟨construct and display high score screen 112b⟩
⟨iris wipe 86⟩
⟨iris wipe step 90⟩
⟨draw wipe step 92a⟩
⟨draw wipe block 96a⟩
⟨load compressed level data 105⟩
⟨load level 107a⟩

; Sound routines

⟨beep 54⟩
```

```
⟨load sound data 56⟩
⟨append note 57a⟩
⟨play note 58⟩
⟨sound delay 59a⟩
⟨play sound 60⟩
⟨append level cleared note 61b⟩

; Joystick routines

⟨read paddles 63⟩
⟨check joystick or delay 65⟩

; Keyboard routines

⟨wait for key 67⟩
⟨wait for key page1 68⟩
⟨editor wait for key 69⟩
⟨hit key to continue 70a⟩
⟨get level from keyboard 71⟩

; Player movement routines

⟨get player sprite and coord data 125a⟩
⟨increment player animation state 125b⟩
⟨check for gold picked up by player 127⟩
⟨check for input 129⟩
⟨ctrl handlers 130a⟩
⟨return handler 135⟩
⟨check buttons 140⟩
⟨try moving up 143⟩
⟨try moving down 148⟩
⟨try moving left 150⟩
⟨try moving right 153⟩
⟨try digging left 157⟩
⟨try digging right 160⟩
⟨drop player in hole 163⟩
⟨move player 164⟩
⟨check for mode 1 input 138⟩

; Guard AI routines

⟨guard store and load data 170⟩
⟨get guard sprite and coords 171b⟩
⟨move guards 172a⟩
⟨move guard 174⟩

; Disk routines

⟨rwts targets 196a⟩
⟨jump to RWTS indirectly 104b⟩
```

⟨*indirect call* 196b⟩
⟨*bad data disk* 189a⟩
⟨*dont manipulate master disk* 189b⟩
⟨*access hi score data* 183⟩
⟨*record hi score data* 185⟩
⟨*check for valid data disk* 190a⟩
⟨*editor initialize disk* 192⟩
⟨*editor clear high scores* 195⟩

`; Startup code`

⟨*startup code* 119a⟩
⟨*check for button down* 121a⟩
⟨*no button pressed* 121c⟩
⟨*button pressed at startup* 122a⟩
⟨*key pressed at startup* 122b⟩
⟨*ctrl-e pressed* 122c⟩
⟨*return pressed* 122d⟩
⟨*timed out waiting for button or keypress* 123a⟩
⟨*check game mode* 123b⟩
⟨*reset game if not mode 1* 123c⟩
⟨*display high score screen* 124a⟩
⟨*long delay attract mode* 124b⟩

`; Game loop`

⟨*Initialize game data* 197⟩
⟨*start game* 199⟩
⟨*game loop* 200⟩
⟨*do ladders* 179⟩
⟨*anims* 203⟩
⟨*show anim line* 205⟩

`; Editor routines`

⟨*level editor* 208⟩
⟨*editor clear level* 210a⟩
⟨*editor move level* 211⟩

215 ⟨* 215⟩≡

```
PROCESSOR 6502
```

⟨*defines* 3⟩
⟨*tables* 7⟩
⟨*routines* 4⟩

Chapter 12

Defined Chunks

Chapter 13

Index