HyperCard

Leon Rische

[2020-03-31 Tue 00:40]

Contents

1 Link Dump

- `https://hypercard.org/`
- `https://hypercard.org/hypercard_file_format_pierre/`
- `http://hypercardonline.tk/about`
- `https://blog.archive.org/2017/08/11/hypercard-on-the-archive-celebrating-30-years`
- `https://archive.org/details/mac_Danny_Goodmans_HyperCard_Developers_Guide_1988/`
- `https://ghostlevel.net/logoff/2015/11/11/hypercard-ressources-bookmark/`
- `https://cancel.fm/hyperjam/`

1.1 Hypertalk

- `http://www.jaedworks.com/hypercard/scripts/hypertalk-bnf.html`
- `https://en.wikipedia.org/wiki/HyperTalk`
- `https://wiki.xxiivv.com/site/hypertalk.html`
- `http://www.jaedworks.com/hypercard/HT-Masters/visual-effects.html`
- `http://web.csulb.edu/~murdock/hcindex.html` (List of commands / functions)

1.2 Emulator

- `https://www.gryphel.com/c/minivmac/`
- `https://www.gryphel.com/c/minivmac/extras/index.html`

1.3 Videos / Talks

- `https://vimeo.com/70833521`
- `https://www.youtube.com/watch?v=LVq67vYyAqo`
- `https://www.youtube.com/watch?v=FquNpWdf9vg`
- `https://www.youtube.com/watch?v=8i60_REoeIY`
- `https://www.youtube.com/watch?v=bdJKjBHCh18`

2 Setup using Mini vMac

1. Download from `https://www.gryphel.com/c/minivmac/dnld_std.html` (or build it yourself)
2. Download a ROM file (e.g. from `https://www.macintoshrepository.org/7038-all-macintosh-roms-68k-ppc-`, I'm using `mac-plus-3.rom`)
3. Rename the ROM file to `vMac.ROM`
4. Download `HyperCardBootSystem7.img` image from `https://archive.org/details/HyperCardBootSystem7`
5. Start the emulator `./Mini\ vMac HyperCardBootSystem.img`

3 Mounting Images on Linux

To mount to a folder:

1. `sudo losetup --find --show HyperCardBootSystem7.img` (returns the name of a loop device, for me that's `/dev/loop0`)
2. `sudo mount /dev/loop0 /mnt`

To unmount:

1. `sudo losetup -d /dev/loop0`
2. `sudo umount /mnt`

4 Mini vMac Hotkeys / Command

- `Ctrl` enters the control mode
- `H` to show help
- `M` to magnify the screen

5 HyperCard Tricks

- To detach a menu (e.g. "Tools" or "Patterns"), hold and drag it
- `Cmd-.` to stop execution of script / long running command
- Double-clicking on a pattern in the pattern menu allows editing it

6 HyperTalk

6.1 `choose <tool name> tool`

Tools:

- browse
- brush
- bucket
- button

- curve
- eraser
- field
- lasso
- line
- oval
- pencil
- rect[angle]
- reg[ular] poly[gon]
- round rect[angle]
- select
- spray
- text

6.2 `drag <pos from> to <pos to>`

Positions have the form `x,y` or `"x,y"` where `x` and `y` are numbers.

6.3 `repeat with x = 0 to 100 \n ... \n end repeat`

6.4 Random Numbers

`random(1, 10)` generates a random number between 1 and 10 (inclusive)

6.5 Variables

- `get <expr>`, evaluates `<expr>` storing the result in `it`
- `put <expr> into <var>`, evaluates `<expr>` storing the result in `<var>`
- `add <expr> to <dest>`
- `subtract <expr> from <dest>`
- `multiply <dest> by <expr>`
- `divide <dest> by <expr>`

6.6 Message Handlers / Functions

If you need a return value, use a function. If not, use a message handler instead.

```
function replaceStr pattern,newStr,inStr
  repeat while pattern is in inStr
    put offset(pattern,inStr) into pos
    put newStr into character pos to (pos +the length of pattern)-1 of inStr
  end repeat
  return inStr
end replaceStr
```

The return value of a function needs to be used as part of an expression, e.g. `put name(arg1, arg2) into void`.

Message handlers can take arguments, too. They differ from functions in that they don't return a value.

See `drawRect` in the next section for an example.

6.7 Comments

Lines beginning with `--` are commented out.

6.8 List Manipulation

Arrays seem to be 1-indexed.

language=C,label= ,caption= ,captionpos=b,numbers=none put "1,2" into list put 3 into item 3 of list – list is now "1,2,3" put item 1 to 2 of list – prints "1,2" in the message box put "a" into character 2 of list – list is now 1a2,3 put item 1 of pos – prints "1a2" in the message box

In addition to `item`, `character` can be used to manipulate strgs.

`length(str)` is the number of characters in `str`. Note that strings are used to represent lists so `length("1,2")` is `3`, not `2`.

To get the number of items in `list`, use `the number of items in list` ;)

6.9 Global Variables

Variables can be shared between elements / cards by declaring them as `global`

For example, in a button on one card:

language=C,label= ,caption= ,captionpos=b,numbers=none on mouseUp global var1, var2 put "Hello" into var1 put "World" into var2 end mouseUp

and then, in a button on a different card:

language=C,label= ,caption= ,captionpos=b,numbers=none on mouseUp global var1, var2 put var1 `" "` var2 end mouseUp

(Note: `&` concatenates two strings)

7 Useful Code / Functions

7.1 Checking if a string is a digit / number

language=C,label= ,caption= ,captionpos=b,numbers=none function isDigit s return offset(s, "0123456789") is not 0 end isDigit

function isNumber s return isDigit(char 1 of s) end isNumber

Note that `isNumber` only check the first digit.

7.2 Drawing Rectangles

Draw a rectangle filled with pattern `p` and size `(sx,sy)` at position `(ox,oy)`

language=C,label= ,caption= ,captionpos=b,numbers=none on drawRect ox,oy,sx,sy,p set filled to true set pattern to p choose rect tool drag from ox,oy to (ox+sx),(oy+sy) choose browse tool end drawRect

This function (message handler) can be used like this: `drawRect 0,0,100,100,12`.

When drawing multiple rectangles, it's faster to `choose rect tool` only once at the start, then reset it to `browse` at the end.

Patterns are indexed starting from 1 (top left in the pattern menu), the pattern at the start of the second row has index 2.

Pattern 12 is black.

7.3 Drawing Lines

language=C,label= ,caption= ,captionpos=b,numbers=none choose line tool drag from x1,y1 to x2,y2

7.4 +/- Counter

Use a field named `"name"` with the default value as content, and two buttons

1. `add 1 to the first word of card field "name"`

2. `subtract 1 from the first word of card field "name"`

8 Processing Stack Files

Stack data is encoded in **big endian** format.

Using Python's `struct.unpack`, we can process the blocks of the file using `struct.unpack(">Icccc", data[offset:8])` (big endian, 4 bytes size, 4 characters type) until we encounter a "TAIL" block.

Note that a block size includes its 4 byte size and its 4 byte name.

8.1 Reading BMAP Blocks

I've filled a cards background with the grid pattern and extracted the stack file from the image.

Here's what the BMAP block looks like:

```
00001440: 0000 00e0 424d 4150 0000 0f0d 0000 0000  ....BMAP........
00001450: 0000 0000 0001 0000 0000 0000 0156 0200  .............V..
00001460: 0000 0000 0156 0200 0000 0000 0156 0200  .....V.......V..
00001470: 0000 0000 0000 0000 0000 0000 0000 0090  ................
00001480: 8283 8083 8083 8083 8083 8083 8083 8082  ................
00001490: a784 82a7 8482 a784 82a7 8482 a784 82a7  ................
000014a0: 8482 a784 82a7 8482 a784 82a7 8482 a784  ................
000014b0: 82a7 8482 a784 82a7 8482 a784 82a7 8482  ................
000014c0: a784 82a7 8482 a784 82a7 8482 a784 82a7  ................
000014d0: 8482 a784 82a7 8482 a784 82a7 8482 a784  ................
000014e0: 82a7 8482 a784 82a7 8482 a784 82a7 8482  ................
000014f0: a784 82a7 8482 a784 82a7 8482 a784 82a7  ................
00001500: 8482 a784 82a7 8482 a784 82a5 84ff ffff  ................
00001510: ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

`0000 00e0 424d 4150 0000 0f0d 0000 0000`

1. Size, 4 bytes (224)
2. Type, 4 bytes, "BMAP"
3. ID, 4 bytes, 0xf0d
4. Filler, 4 bytes

`0000 0000 0001 0000` 4 x 2 bytes, Unknown, here 0, 0, 1, 0

`0000 0000 0156 0200`, `0000 0000 0156 0200`, `0000 0000 0156 0200` top, left, bottom, right of the card, mask and image rectangles. In this case, each rectangle has size (512,342) × (512,342)

`0000 0000 0000 0000` 2 x 4 bytes unknown, usually 0

`0000 0000 0000 0090` 4 bytes, size of the mask data (0) 4 bytes, size of the image data (144)

Mask and image data are stored separately to support transparency, using the mask for white pixels and the image for black pixels.

There is no mask data, so we treat all pixels in the mask as white (1).

At this point, we are at 16 + 8 + 24 + 8 + 8 = 64 bytes of data and we're left with 160 bytes of image data.

It seems like block sizes are rounded to multiples of 16 or 32, so the `0xff` in the last row would be filler data.

Now to the interesting part, decoding the image data:

```
00001480: 8283 8083 8083 8083 8083 8083 8083 8082
00001490: a784 82a7 8482 a784 82a7 8482 a784 82a7
000014a0: 8482 a784 82a7 8482 a784 82a7 8482 a784
000014b0: 82a7 8482 a784 82a7 8482 a784 82a7 8482
000014c0: a784 82a7 8482 a784 82a7 8482 a784 82a7
000014d0: 8482 a784 82a7 8482 a784 82a7 8482 a784
000014e0: 82a7 8482 a784 82a7 8482 a784 82a7 8482
000014f0: a784 82a7 8482 a784 82a7 8482 a784 82a7
00001500: 8482 a784 82a7 8482 a784 82a5 84ff ffff
```

Before compressing / decompressing the image data, the left side of the bounding box is rounded down to the nearest multiple of 32, the right side is rounded up.

Our bounding box is 512 pixels wide, so we don't need to do any rounding.

This encoding uses a custom compressed data format.

Each byte encodes 8 pixels of a row , so we need a total of 64 bytes to encode a full row.

`82` Our image starts with `0x82`, one black row.

`83 8083 8083 8083 8083 8083 8083 80` The next instructions are a bunch of `0x83 0x80`, each filling one row by repeating the byte `0x80` (`0b01000000`).

Next comes another `82` black row.

`a7` repeats the next instruction 7 times, `84` fills one row with a repeated byte of data previously used, looking up the byte in a 8-byte lookup table that is updated each time a `83` instruction is encountered.

In our case, this means that 7 rows are filled with repeated `0x80`

Next comes another black row and more repeated `0x80`.

The combination of `a784` is repeated a few more times, ending with a `a584` when only 5 rows are left to be filled.

At this point, we can see that the background is a grid pattern without even decoding it.