Encoding 2-gMaps using Morton codes and bit flips

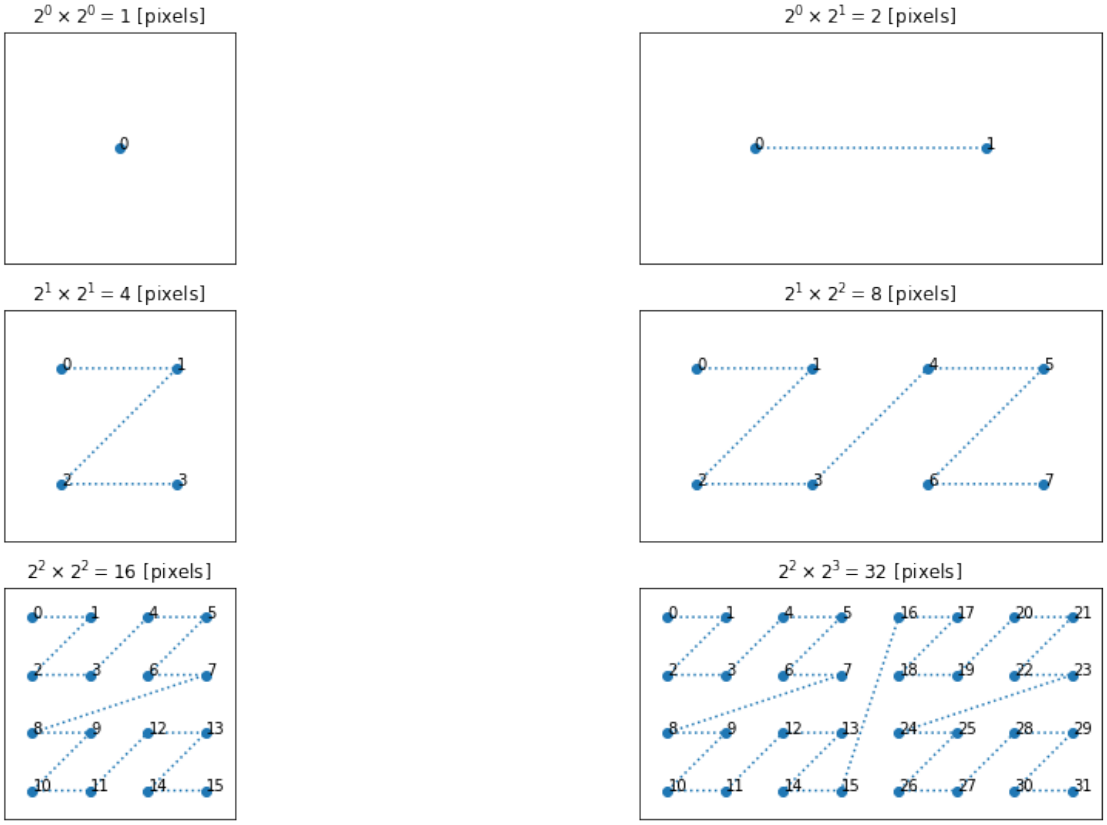
Jiří Hladůvka

April 30, 2021

Motivation: For an advanced caching strategies of pixel/voxel scenes, space-filling curves have been utilized to number, store, and access the pixels/voxels [TODO refs]. The most prominent curves in this regard are Hilbert curve and the Morton curve (a.k.a Z-curve). The later one utilizes a faster mapping between 2D/3D coordinate and the index, which can be expressed in bit operations reference here. For that this encoding has been has been widely used, e.g., for texture mapping in computer graphics [TODO refs].

1 Morton curve

For $n \in \{0, 1, ...\}$, Morton codes can efficiently order pixels of $2^n \times 2^n$ or $2^n \times 2^{n+1}$ images by interleaving $2n$ or $2n + 1$ bits, respectively.



2 Involutions for 2-gMaps (a recap)

As we aim at edge removal and/or contraction, it is of advantage that the 4 darts of one edge are kept together. Walter proposed earlier how this can be achieved by implicit encoding of α_0 and α_2 using simple arithmetics.

Jiri showed that the same can be efficiently achieved by bit-flipping:

- $\alpha_0(d) = d$ xor 1 ... flip bit 0 of d
- $\alpha_2(d) = d$ xor 2 ... flip bit 1 of d

Seeing α_0 and α_2 as bit flips, it is easy to see that α_0, α_2, and $\alpha_0 \circ \alpha_2$ are all involutions which is a necessary requirement from the definition of 2-gMap.

What remains to define is α_1. Here we have a relative freedom: it is up to the design, how the edges are stitched together by α_1.

To utilize the Morton code, we can use a one-to-one correspondence between pixel and its two edges, i.e., North and West ones. Using this scheme we encode all edges except for those of East and South boundary of the image. Referring to the following figure (see separate PDF file for larger version) we'll refer to these two cases as the color-zone and the gray-zone.

3 2×4 baselevel example

We'll need 7 bits, numbered 6 to 0.

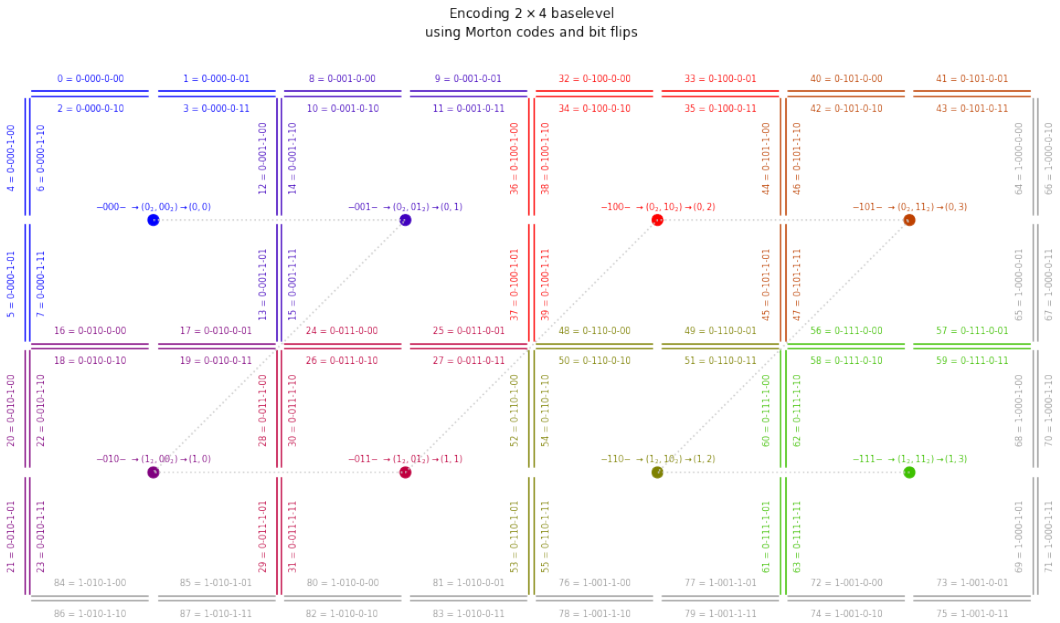
- number of rows R=2,
- number of columns C=4.

To encode the darts of the color-zone we need 6 bits (note the left-most bit 6 is set to zero)

- 3 bits = $\log_2(R \times C) = \log_2(2 \times 4)$ to Morton-encode the pixel coordinates
- 1 bit to encode if the edge is North or East one
- 2 bits to encode the 4 dart of the edge

To encode the darts of the gray-zone we need additional

- $4 \times (R + C) = 24$ darts
- which requires an additional bit (note the left-most bit 6 is set to 1). Within the gray-zone the edges can be organized independently of the Morton code. The example shows North-to-South followed by East-to-West ordering of edges.



3.1 Example in the color-zone: darts of pixel $(y, x) = (0, 3)$

Meaning of the bits:

- bit 6 = 0 indicates a colored-zone (i.e, a north-or-west) edge of some pixel
- Morton bits 5…3
  - even subsequence thereof = bit 4 = $0_2 = 0_{10}$ encode the y coordinate of the pixel
  - odd subseqence thereos = bits 5 and 3 = $11_2 = 3_{10}$ encode the x-coordinate of the pixel

- bit 2
  - 0 for the north (horizontal) darts (40…43)
  - 1 for the west (vertical) darts (40…47)
- bits 1 and 0 … index $i \in \{0...3\}$ of the dart within the edge, optionally further decomposed as follows
  - bit 1 = 0 … upper dart on the north edge / left dart on the west edge
  - bit 1 = 1 … lower dart on the north edge / right dart on the west edge
  - bit 0 = 0 … left dart on the north edge / upper dart on the west edge
  - bit 0 = 1 … right dart on the north edge / lower dart on the west edge

3.2 Examples of α_0 and α_2 and 1-orbits (edges)

- α_0 … flips bit 0, e.g., $\alpha_0(42_{10}) = \alpha_0(0101010_2) = 0101011_2 = 43_{10}$.
- α_2 … flips bit 1, e.g., $\alpha_2(42_{10}) = \alpha_2(0101010_2) = 0101000_2 = 40_{10}$.

- 1-orbit (edge): set of 4 darts with identical bits 6..2, e.g., edge (43) = $\{01010{-}??_2\}$ = $\{01010-00_2,\ 01010-01_2,\ \ 01010-10_2,\ 01010-11_2\} = \{40, 41, 42, 43\}$

3.3 Rethinking α_1

Involutions α_0 and α_2 are given implicitly.

α_1 can be encoded implicitly, too, though it is more involved than bit flips.

The darts can be divided in 3 classes according to the valence of the incident vertex:

- 4-edges
- 3-edges
- 2-edges

3.3.1 α_1 for 4-edge darts

All such darts belong to the color-zone. The involution α_1 is here determined by bits 2,1, and 0, e.g., by means of an 8-entry lookup table:

old bits 210	new bits 210	Δy	Δx
000	111	-1	0
001	101	-1	+1
010	110	0	0
011	100	0	+1
100	011	0	-1
101	001	+1	-1
110	010	0	0
111	000	+1	0

Here Δy and Δx determine how the pixel coordinate changes undef α_1. This affects only the Morton part of the bit-code (bits 5,4,3 in this 2×4 image example).

The procedure for a dart d may look as follows:

```
def alpha_1 (d):
    i =  d & 0b000111              # mask lookup index from dart
    m = (d & 0b111000) >> 3        # mask Morton code  from dart

    y,x = decode (m)               # Morton code to pixel coordinates

    bits, dy, dx = Lookup [i]      # Lookup

    y += dy                        # pixel's y update
    x += dx                        # pixel's x update
    m = encode ((y,x))             # pixel to Morton code

    return (m << 3) | bits         # compose the things back, yield the dart
```

3.3.2 α_1 for 3-edge and 2-edge darts

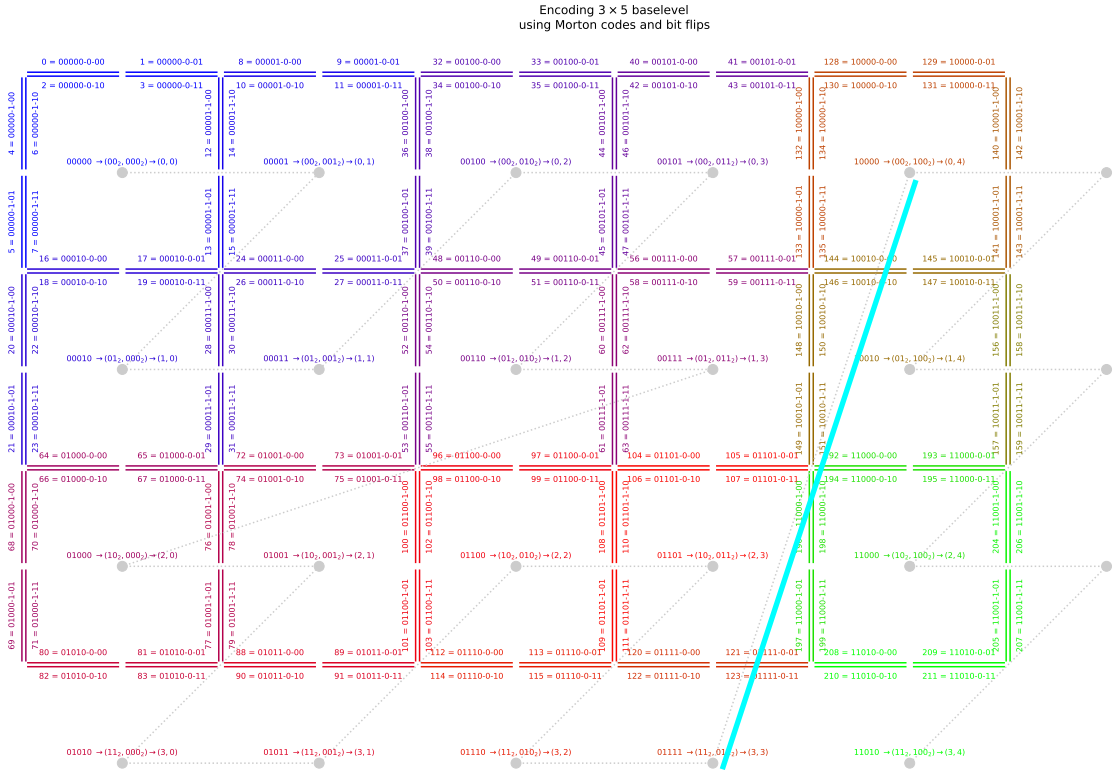
Under constructions: basically via look-up tables, too. Special attention to the gray-zone (left-most bit=1). The python notebooks currently uses maps and some logic. If no efficient implementation can be proposed (i.e, one w/o branching), the worst-case scenario would be to encode these cases explicitly.

4 Generalization to $2^n \times 2^n$ and $2^n \times 2^{n+1}$ images

… for nonnegative integer n is straightforward. Formalism is a work in progress.

5 Generalization to $R \times C$ images

can be achieved by dropping the requirement on having a continuous block of dart numbers. In the example 3×5 image below, the discontinuity starts with missing vertical edge of pixel (0,4), $D = \{0, \ldots 83, 88 \ldots 91, 96 \ldots 115, 120 \ldots, \}$ This, however does not pose a problem with additional storage for dart numbers, as the set of the darts is given implicitly, too.



5.1 Encoding

Referring to the above image the darts have the following encoding 8-bit encoding: `xyxyx-a-bb`.

- 2 bits (`bb`) to encode the position of a dart within edge
- 1 bit (`a`) to encode the horizontal/vertical edge (`a` stands for axis which can be 0 or 1)
- remaining bits to encode the pixel coordinate x,y (in the interleaving fashion for z-curve):
  - to account for the south and east borders of the image, we need to encode (R+1) rows $r \in \{0..R\}$, and (C+1) columns $c \in \{0..C\}$

5.2 Limits imposed by B-bit unsigned integers

Reserving the 3 rightmost bits (`abb`) to encode the 8 Nort-West darts of every pixel we have $B-3$ bits to encode the pixel coordinates. These are split as follows:

- $B/2-2$ bits for rows, constraining the number of rows as follows: $0 < R < 2^{B/2-2}$
- $B/2-1$ bits for columns, constraining the number of columns as follows $0 < C < 2^{B/2-1}$

Examples

bits	row bits	column bits	max Rows	max Columns
8	2	3	3	7
16	6	7	63	127
32	14	15	16383	32767
64	30	31	1073741823	2147483647

5.3 Implementation of α_1

The implementation of α_1 can be now more streamlined than that of 1-block set of darts: a branch-less (i.e., no `if-then-else` blocks) implementation that utilizes a precomputed 24-entry lookup table to cover all 3 categories of darts:

- outer boundary: corners
- outer boundary: straight
- interior

The identification is achieved by matching the given dart d dart to the first possible entry in the lookup. This can be achieved with pre-computed XOR-AND masks.

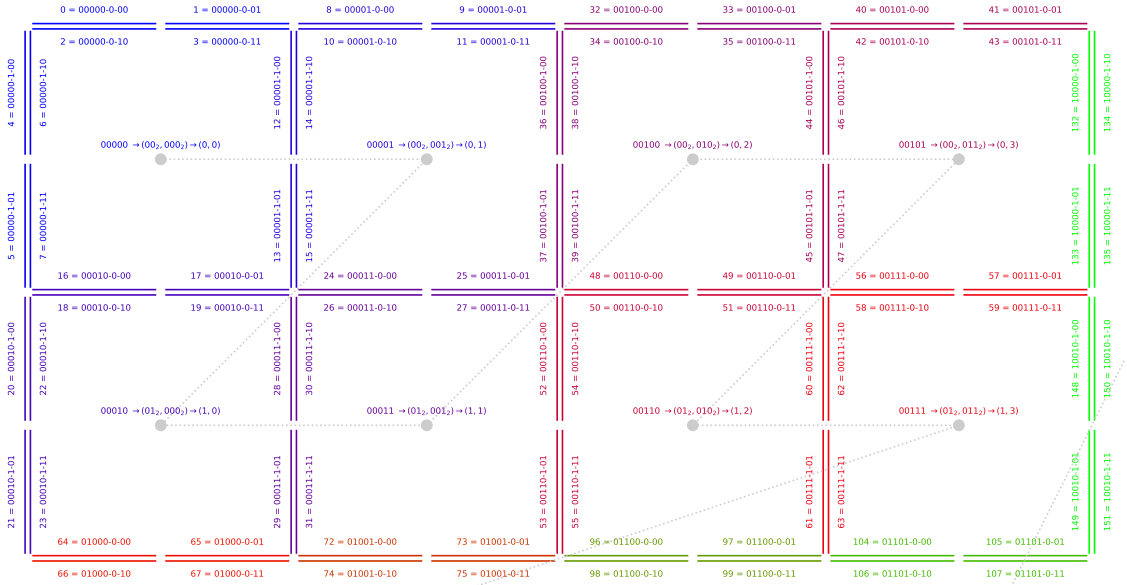
The lookup table additionally stores information on how to:

- updated the pixel coordinates $(\Delta x, \Delta y)$
- map the 3 leftmost bits

Given the information from the lookup table the procedure is as follows, see `02_pixelmap_z_curve_full.ipynb` for detail.

```
def alpha_1 (d):
    i = match (d)                                  # mask lookup index from dart
    dx,dy,newbits = DXs[i],DYs[i],NEWBITS[i]       # Lookup
    x,y = deinterleave2(d >> 3)                    # Morton code to pixel coords
    x+=dx; y+=dy;                                  # pixel update
    return (interleave2(x,y) << 3) | newbits       # pixel coord to Morton and bit composition
```

5.3.1 An example lookup table for 2×4 images



	d	Δx	Δy	$\alpha_1(d)$	
upper left horizontal	`00000-0-00`	0	0	`00000-1-00`	upper left vertical
upper right horizontal	`00101-0-01`	1	0	`10000-1-10`	upper right vertical
lower left horizontal	`01000-0-10`	0	-1	`00010-1-01`	lower left vertical
lower right horizontal	`01101-0-11`	+1	-1	`10010-1-11`	lower right vertical
border top left	`*0*0*-0-00`	-1	0	`*0*0*-0-01`	border top right
border bottom left	`*1*0*-0-10`	-1	0	`*1*0*-0-11`	border bottom right
border left upper	`0*0*0-1-00`	0	-1	`0*0*0-1-01`	border left lower
border right upper	`1*0*0-1-10`	0	-1	`1*0*0-1-11`	border right lower
interior	`*****-0-00`	0	-1	`*****-1-11`	interior
interior	`*****-0-01`	+1	-1	`*****-1-01`	interior
interior	`*****-0-10`	0	0	`*****-1-10`	interior
interior	`*****-0-11`	1	0	`*****-1-00`	interior

The lookup table defines:

1. matching patterns for all dart cases (the asterisks `*` stand for any of 0 or 1):
   - 8 outer-corner-darts (rows 1-4)
   - 8 outer-border-dart patterns (rows 5-8)
   - 8 interior-dart patterns (rows 9-12)
2. the mapping of the 3 rightmost bits
3. increment of coordinate of the representative pixel:
   - (x, y) `+=` $(\Delta x, \Delta y)$ when reading the line left-to-right
   - (x, y) `-=` $(\Delta x, \Delta y)$ when reading the line right-to-left