Asger Hoedt

Contact Info:
email: asgerhoedt@gmail.com
tlf: +45 6051 3669

Address:
Vejledalen 246
2635 Ishøj
Denmark

Morton Codes

Posted on **October 19, 2012**

I've recently had to use 3D and 5D Morton codes for spatial datastructure construction. Unfortunately I found that the material on the net actually describing how to encode, decode and interprete Morton codes is non-existent to shallow at best. The best sources I've found are wikipedia and an excellent blog post by Fabian "ryg" Giesen. This spurred me to create this blog post about my own experience with Morton codes and link to resources with more information.

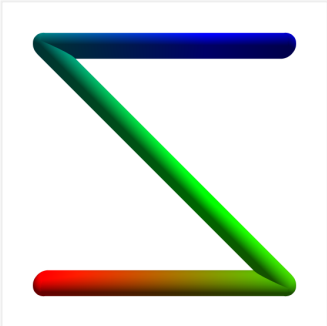
Introduction

Morton codes provides an ordering along a space-filling curve while preserving data locality. Lets break that down a bit shall we?

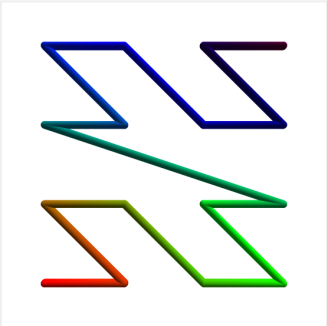
A space-filling curve is a curve whose range contains the entire N-dimensional hypercube enclosing your data domain mapped to discrete cells. When working with 3-dimensional data the hypercube is usually just an axis aligned bounding box. What this all means is that the data is enclosed by the hypercube and that hypercube is then subdivided along each dimension, creating an N-dimensional grid, lets call that a hypergrid. A space-filling curve that contains the entire hypercube is guaranteed to pass through each cell of the grid.

A space-filling curve preserves data locality if similar data gets mapped within the same range on the curve. This means that a cell in the hypergrid gets mapped to a range on the curve and therefore all the data inside that cell gets mapped to a position inside that range. A good space filling curve will also map neighbouring cells close each other.

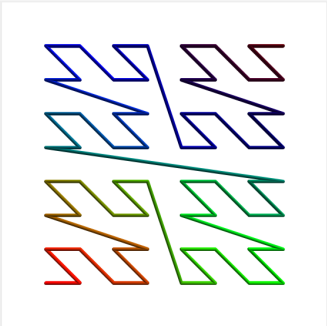
Morton codes provides such a spatially local mapping. Below are three iterations of the Morton order in 2- and 3-dimensions, where each iteration adds a new subdivision to the cells of the previous iteration.



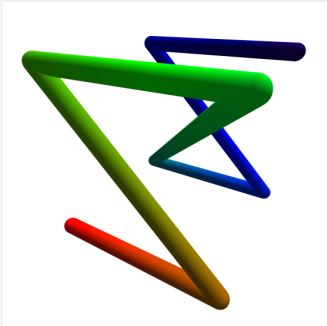
MortonCurve 2×2



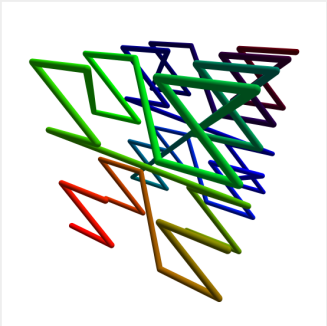
MortonCurve 4×4



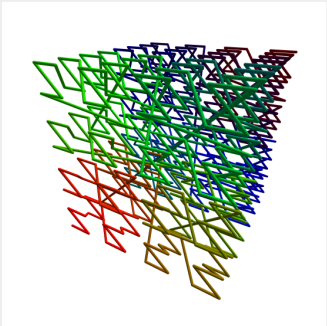
MortonCurve 8×8



MortonCurve 2x2x2



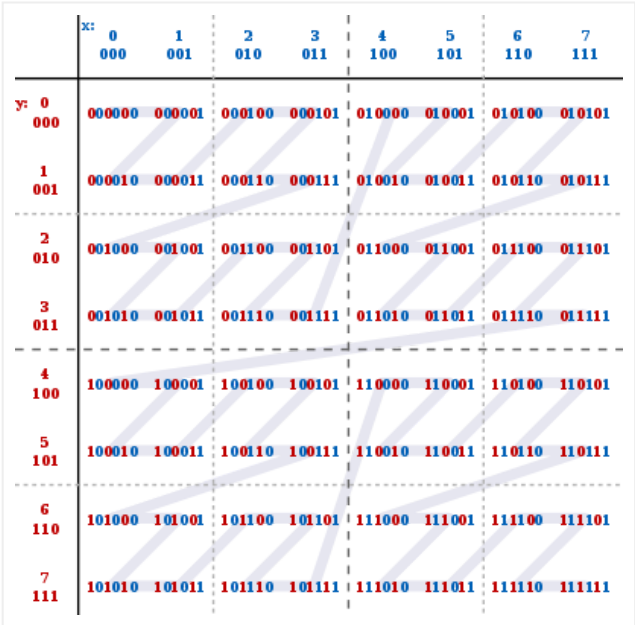
MortonCurve 4x4x4



MortonCurve 8x8x8

Encoding Morton Codes

Creating the Morton code for some data amounts to determining coordinates (x, y) of the data inside the hypergrid and interleaving those coordinates. The figure below shows the Morton codes and curve in the 2-dimensional case with a 8×8 grid. Interleaving the binary representation of a cell's coordinates yields the Morton code for the respective cell and connecting the cells in order of increasing Morton code connects the entire Morton curve.



Binary Morton curve creation

Implementing a function that can create Morton codes boils down to two problems: First seperate the least significant bits in c_x and c_y by N-1 and secondly combine them into a Morton code.

The implementation that separates the bits via a classical Divide-and-Conquer approach looks like this. In the comments after each line you can see the position of the significant bits.

```
unsigned int SeparateBy1(unsigned int x) {
    x &= 0x0000ffff;                  // x = ---- ---- ---- ---- fedc
    x = (x ^ (x <<  8)) & 0x00ff00ff; // x = ---- ---- fedc ba98 ----
    x = (x ^ (x <<  4)) & 0x0f0f0f0f; // x = ---- fedc ---- ba98 ----
    x = (x ^ (x <<  2)) & 0x33333333; // x = --fe --dc --ba --98 --76
    x = (x ^ (x <<  1)) & 0x55555555; // x = -f-e -d-c -b-a -9-8 -7-6
    return x;
}
```

Adapting the code to separate by two, three or four is pretty straightforward. Since I needed separation by 4 for a 5D ray representation I'll include that here.

```
unsigned int SeparateBy4(unsigned int x) {
    x &= 0x0000007f;                  // x = ---- ---- ---- ---- ----
    x = (x ^ (x << 16)) & 0x0070000F; // x = ---- ---- -654 ---- ----
    x = (x ^ (x <<  8)) & 0x40300C03; // x = -6-- ---- --54 ---- ----
    x = (x ^ (x <<  4)) & 0x42108421; // x = -6-- --5- ---4 ---- 3---
    return x;
}
```

Once we are able to separate the bits in the coordinate's binary representation, interleaving the result to obtain the Morton code is a simple matter of bit-shifting and or'ing.

```
MortonCode MortonCode2(unsigned int x, unsigned int y) {
    return SeparateBy1(x) | (SeparateBy1(y) << 1);
}

MortonCode MortonCode5(unsigned int x, unsigned int y, unsigned int z,
    return SeparateBy4(x) | (SeparateBy4(y) << 1) | (SeparateBy4(z) <<
}
```

Decoding Morton Codes

Given a Morton code it can be useful to know which coordinates were used to generate it by deconstructing or decoding it. Doing this amounts to reversing the encoding, which is done by inverting the shifts and reversing the order of the masks, yielding the following functions.

```
unsigned int CompactBy1(unsigned int x) {
    x &= 0x55555555;                  // x = -f-e -d-c -b-a -9-8 -7-6
```

```
    x = (x ^ (x >>  1)) & 0x33333333; // x = --fe --dc --ba --98 --76
    x = (x ^ (x >>  2)) & 0x0f0f0f0f; // x = ---- fedc ---- ba98 ----
    x = (x ^ (x >>  4)) & 0x00ff00ff; // x = ---- ---- fedc ba98 ----
    x = (x ^ (x >>  8)) & 0x0000ffff; // x = ---- ---- ---- ---- fedc
    return x;
}

unsigned int CompactBy4(unsigned int x) {
    x &= 0x42108421;                  // x = -6-- --5- ---4 ---- 3---
    x = (x ^ (x >>  4)) & 0x40300C03; // x = -6-- ---- --54 ---- ----
    x = (x ^ (x >>  8)) & 0x0070000F; // x = ---- ---- -654 ---- ----
    x = (x ^ (x >> 16)) & 0x0000007F; // x = ---- ---- ---- ---- ----
    return x;
}
```

Using the CompactByN functions, we can construct a MortonDecodeN function similar to MortonCodeN seen above.

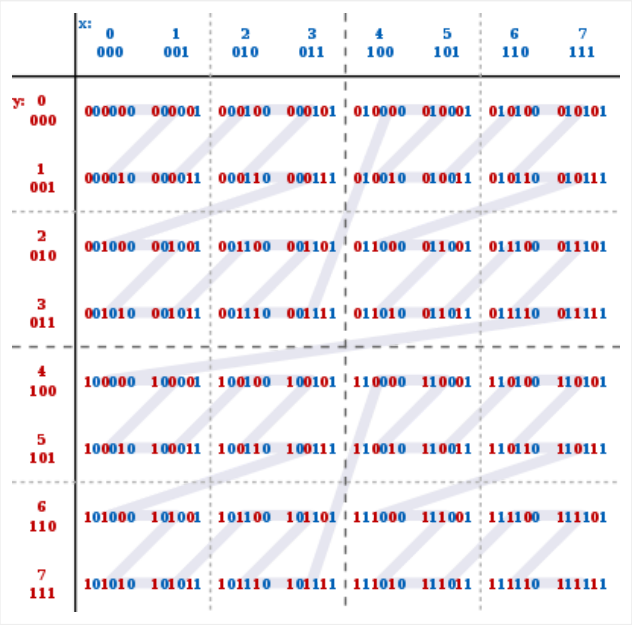
```
void MortonDecode2(MortonCode c, unsigned int x, unsigned int y) {
    x = CompactBy1(c);
    y = CompactBy1(c >> 1);
}

void MortonDecode5(MortonCode c, unsigned int x, unsigned int y, unsig
    x = CompactBy4(c);
    y = CompactBy4(c >> 1);
    z = CompactBy4(c >> 2);
    u = CompactBy4(c >> 3);
    v = CompactBy4(c >> 4);
}
```

Constructing Acceleration Datastructures

Recently a lot of methods for creating acceleration datastructures for ray tracing have focused on using space-filling curves to speed up construction time, fx LBVH and HLBVH. The idea is simple: Map your multi-dimensional data onto the one-dimensional space-filling curve, sort the data according to its position on the curve and then create an acceleration structure on top of the cells. Due to the recursive nature of Morton codes, creating the acceleration structure is very simple.

An interesting observation about Morton codes and acceleration structures is that the binary representation of a Morton code, *m*, can be interpreted as a traversal of an implicit binary tree to reach *m*'s cell. The traversal is performed by looking at each bit in turn, starting with the most significant bit. Each bit represents a splitting of the domain's data along the axis that the bit derived its value from. The value of the bit will then tell us which side of that splitting plane our traversal should proceed to, with 0 meaning we traverse the 'lower' child and 1 meaning the 'upper' child.

This is best clarified by example so again we will look at the figure with binary Morton codes.



Binary Morton code representation

Using cell (2, 3) as our example cell we are working with the Morton code 001110. Starting from the tree's root node, which spans the entire domain, (0,0) -> (7,7), the first bit **0**01110 tells us that our bounds should be split along the spatial y-median, between y=3 and y=4, and that the cell we are looking for is found among the lower numbered cells. Splitting (0,0) -> (7,7) along y's spatial median we get the new bounds (0,0) -> (7,3).

Next the second bit 0**0**1110 tells us that we should split the bounds along the spatial x-median, between x=3 and x=4, and again visit the lower child. This yields the new bounds (0,0) -> (3,3)

Repeating this process for all bit in the morton code we get the bounds
(0,0) -> (3,3) => (0,2) -> (3,3) => (2,2) -> (3,3) => (2,3) -> (3,3) => (2,3) -> (2,3)
where the last bounds enclose our starting cell (2,3) perfectly.

Advanced Operations

Usually, knowing how to encode and decode Morton codes is enough to sort data and create acceleration structures around that data. However, recently I've had to recursively assign geometry to a sorted list of rays, starting with assigning the geometry seen by all the rays to the root node and then recursively refine that for each child. In order to do that I found it helpful to do several things.

- Given two Morton codes, *m0* and *m1*, determine the Morton code that encodes the minimum coordinates of both *m0* and *m1*.
- Given two Morton codes determine the encoding of their maximum coordinates.

Minimum and maximum

Finding the Morton code that encodes the minimum coordinates of two Morton codes is trivial.

```
MortonCode Minimum2(MortonCode lhs, MortonCode lhs) {
   unsigned int lhsX, lhsY, rhsX, rhsY;
```

```
    // Decode the left hand side coordinates.
    MortonDecode2(lhs, lhsX, lhsY);

    // Decode the right hand side coordinates.
    MortonDecode2(rhs, rhsX, rhsY);

    // Find the minimum of the coordinates and re-encode.
    return MortonCode2(Min(lhsX, rhsX), Min(lhsY, rhsY));
}
```

But we can do better! Decoding and re-encoding means doing a lot of bit-shifting instructions that we'd like to avoid. First, realising that we're not interested in the actual values of x and y, but rather the minimum of their encoded values is a tremendous help. Secondly, SeperateBy1 preserves ordering, so if X < Y then SeperateBy1(X) < SeperateBy1(Y), as long as X and Y are valid coordinates. Using this we can construct a Minimum function that avoids a lot of instructions.

```
MortonCode Minimum2(MortonCode lhs, MortonCode lhs) {
    // Isolate the encoded coordinates.
    unsigned int lhsX = lhs & 0x55555555; // lhsX = -f-e -d-c -b-a -9-8
    unsigned int rhsX = rhs & 0x55555555; // rhsX = -f-e -d-c -b-a -9-8
    unsigned int lhsY = lhs & 0xAAAAAAAA; // lhsY = f-e- d-c- b-a- 9-8-
    unsigned int rhsY = rhs & 0xAAAAAAAA; // rhsY = f-e- d-c- b-a- 9-8-

    // Find the minimum of the encoded coordinates and combine them int
    return Min(lhsX, rhsX) + Min(lhsY, rhsY);
}
```

Maximum can be constructed in a similar manner.

This entry was posted in **Blog** and tagged **Bitflipping**, **BVH**, **datastructure**, **Morton Code**, **Ray Tracing** by **papaboo**. Bookmark the **permalink [http://asgerhoedt.dk/?p=276]** .

9 THOUGHTS ON “MORTON CODES”

Naseem Abbas
on **January 9, 2015 at 4:16 pm** said:

Hi, Thanks for such a informative article on Morton Codes. I have a question if you can provide me help.
You have used unsigned int for calculation of morton codes. I have a point cloud in which every point is represented by a float and and also have negative co-ordinate values with fractions. How can we calculate Morton codes while preserving the co-ordinates. Thanks in advance.

papaboo
on **March 20, 2015 at 9:01 pm** said:

Hey, thanks for reading the post.
To encode floating point positions as morton codes you need to map the coordinates in each dimension to an integer range and then encode them.
Fx, if you have a 2D point cloud with a bounding box, bb, [min: [-8,-8], max: [24,24]] and want to encode the point, p, (-4, 8). First you have to use the bounding box to remap p's coordinates into the range [0,1].
pNorm = (p – bb.min) / (bb.max – bb.min) = ((-4 + 8) / 32, (4 + 8) / 32) = (0.125, 0.375)
Now that you have a range of [0, 1] you have to decide how much precision you want. If you need a 32bit morton code, then it makes sense to split the bits 50/50 and use 16bits pr coordinate. So we remap once again, this time from [0,1] and into [0, (2^16)-1]. And voila, you've converted your points in the point cloud into an integer that can be used for creating morton codes.
For 3D points you can choose to use 10bit pr dimension to create a 30bit morton code. If you're feeling adventurous you can even use 11bit for two of the dimensions to get a full 32bit morton code.

Sven
on **January 21, 2017 at 1:54 pm** said:

pNorm = (p – bb.min) / (bb.max – bb.min) would yield a floating point value. floating point values are represented as sign, exponent and mantissa in binary form. How does the mapping work if we simply split those floating point values (taking the first 16 bit from a 32bit floating point value would give us a part from the mantissa)? Values which have the same mantissa but different exponents would have the same morton key or did is miss something?

papaboo
on **January 28, 2017 at 8:42 pm** said:

True. Taking the first 16 bits of a 32 bit floating point number is indeed a bad idea and cuts of parts of your mantissa. This is especially bad, since pNorm is in the range [0, 1], so the sign and exponent are rather unimportant.
What you want is to take pNorm, which is in the range [0, 1] and remap it to an integer in the range [0, 65535]. That way all you information is stored in exactly 16 bits with as high a precision as possible. And those 16 bits can then be encoded together with another 16 bit integer.

some person
on **April 3, 2016 at 9:16 am** said:

Best article on morton ordering out there.
Thank you for the clear and concise explanation!

Syd
on **February 3, 2017 at 5:26 pm** said:

Great Article!!
Any idea how to map the morton codes to a B-Tree for the n-dimensional data. There are many resources for the z-curve in-conjunction with B-Tree (i.e. UB-Tree), however, I can't seems to find any understandable algorithm for mapping codes to B-Tree and traversing it in a Z-Order for range queries. Cheers

papaboo
on **February 6, 2017 at 10:59 am** said:

I'm not entirely sure what it is you want. What is your use case? Finding the nearest K data points or finding all K data points within some distance?
I would assume that using Morton Codes and B-Trees in general just amounts to using the Morton Code as the key. If you want to ensure spatial locality then you could try to ensure that only spatially local points are grouped in the individual nodes in your tree. Fx if you have 3D data and 30bit morton codes, then only group morton codes that agree on the first 27 or 24 bits. Grouping them by leading 27bits will basically give you an octree though.

Syd
on **February 9, 2017 at 12:29 pm** said:

Sorry for quite an ambiguous questions. So what I would be interested in generating a morton code for N-Dimensions . That is, generalising the SeparateBy4 or SeparateBy3 function. Thus, providing N-Dimensions F({x_1, x_2, … x_n}) = Morton code. Would this result in an overflow? Such morton code can be used within a B+tree to provide range queries over N-Dimensions. Cheers

abeer
on **December 6, 2018 at 1:36 pm** said:

Wonderful Article.

I want to know some basics? Is it true that by using the morton codes we can get the highly correlated points in 3d. So if someone need the highly correlated points in the 3d point cloud. so what he can do? He can construct the octree and then order the points using morton code? Secondly is there a way to represent the interleaving of binaries in case for 3d to visualize the results?

Regards
Abeer