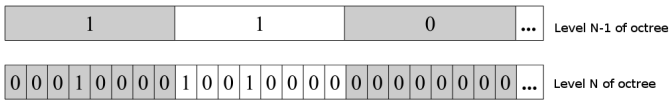
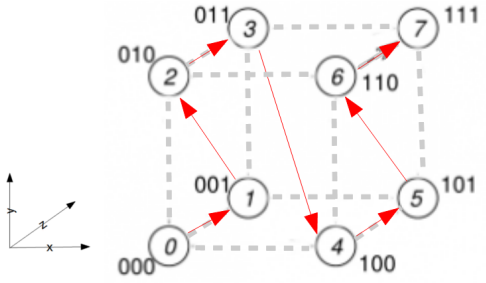
Morton encoding/decoding through bit interleaving: Implementations

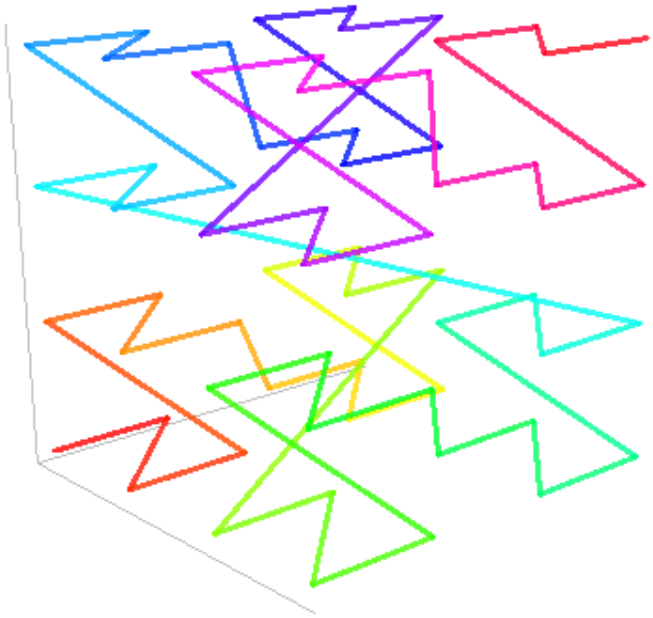
UPDATE 2016: I've bundled and improved a lot of these methods in a library called libmorton. **Read about it here**. The code used in this post is outdated and might have bugs. Keeping this post online for reference, but I highly suggest using **libmorton** for actual implementations.

In my research on building Sparse Voxel Octrees, I often use Morton codes. The Morton order is a mapping from an n-dimensional space onto a linear list of numbers. If you apply it to coordinates, the morton order defines a space-filling curve which is Z-shaped – that's why the Morton order is often called Z-order or Z-curve as well. The curve has some nice locality properties: coordinates which are close to eachother in the N-dimensional space have morton numbers which are close to eachother too.



Morton – Encoded Array





The Z-order curve inspired a lot of people to do great stuff with compression and parallel data construction. In my High Performance Graphics paper, I use the property that Morton order is a post-order depth-first traversal of a multi-dimensional tree to efficiently build a Sparse Voxel Octree. Go read it ;)

If you want to convert a certain set of integer coordinates to a Morton code, you have to convert the decimal values to binary and interleave the bits of each coordinate:

- (x,y,z) = **(5,9,1)** = (0101,1001,0001)
- Interleaving the bits results in: 010001000111 = **1095** th cell along the Z-curve.

So in order to do anything interesting with the Morton order, we need an efficient way of interleaving bits of a three-dimensional coordinate. For the following functions, I assume:

- The **morton code** is stored as a **64-bit integer**.
- **x, y** and **z** are three **unsigned, 32-bit integers**. Only 21 bits (starting from the right) will be used, because 3 x 21 bits is 63 bits, which is the maximum we can fit in a 64-bit morton code. So yes, still one bit free for a custom flag of your choosing! (hint: In a voxel-based system, this can be your "filled" boolean)

We'll be using a lot of bitwise operations in the following code, so read up if you're not familiar with them. We'll mainly be using left and right shifts (<< and >>) and bitwise and (&) and or (|).

Update (nov 2013): **Alexandre Avanel** made a great LUT-based implementation as well. Available in this github repo. Also thanks to Alexandre for an optimization in the for-loop based method below.

Update (apr 2014) Another great SIMD-based implementation **here**.

For-loop based method

The first way of tackling this is to use a for-loop with shifts. As you can see, we make sure the bits from x are right-most, then the ones from y in the middle and z to the left. We incrementally build the answer by shifting in new bits form each of the input coordinates.

```
#include <stdint.h>
#include <limits.h>
using namespace std;

inline uint64_t mortonEncode_for(unsigned int x, unsigned int y, unsigned int z){
uint64_t answer = 0;
for (uint64_t i = 0; i < (sizeof(uint64_t)* CHAR_BIT)/3; ++i) {
answer |= ((x & ((uint64_t)1 << i)) << 2*i) | ((y & ((uint64_t)1 << i)) << (2*i + 1)) | ((z & ((uint64_t)1 << i)
}
return answer;
}
```

This method is easy to implement, compact, and relatively easy to read (though you might panic when you're not used to bitwise operations).

"Magic Bits" method

Inspired by this blogpost by fgiesen, Sean Eron Anderson's Bit Twiddling Hacks and this StackOverflow discussion, I generated this method for interleaving 32-bits integers into a 64-bit morton code.

This is a bit harder to implement / understand, and isn't that straightforwardly extendable for more bits / other input sizes, but it is a whole lot faster than the previous method (see performance comparison further down) and has the added benefit of being nice and small compared to the LUT implementation.

```
#include <stdint.h>
#include <limits.h>
using namespace std;

// method to seperate bits from a given integer 3 positions apart
inline uint64_t splitBy3(unsigned int a){
uint64_t x = a & 0x1fffff; // we only look at the first 21 bits
x = (x | x << 32) & 0x1f00000000ffff; // shift left 32 bits, OR with self, and 0001111100000000000000000000000000
x = (x | x << 16) & 0x1f0000ff0000ff; // shift left 32 bits, OR with self, and 0001111100000000000000001111111100
x = (x | x << 8) & 0x100f00f00f00f00f; // shift left 32 bits, OR with self, and 0001000000001111000000001111000000
x = (x | x << 4) & 0x10c30c30c30c30c3; // shift left 32 bits, OR with self, and 0001000011000011000011000011000011
x = (x | x << 2) & 0x1249249249249249;
return x;
}

inline uint64_t mortonEncode_magicbits(unsigned int x, unsigned int y, unsigned int z){
uint64_t answer = 0;
answer |= splitBy3(x) | splitBy3(y) << 1 | splitBy3(z) << 2;
return answer;
}
```

Lookup Table (LUT) method

This is basicly a divide-and-conquer method. We can precompute splitting a certain subset of bits (1 byte = 8 bits = decimals 0 -> 255). And then split the input integers byte-by-byte, and shift the results in place.

For an even further optimization, I also precomputed the shifts for y and z. So the extra tables are basicly the same as the Morton256_x table, but shifted to the left by 1 bit (for y) and 2 bits (for z). This seems like a trivial optimization, but it saves on doing 6 shifts, which can make a difference if computing morton codes is on your critical path.

How much do these tables cost? It's 256 * 32 bits * 3 tables = ~**3 Kb**, so your executable size won't take a big hit. Of course, baking bigger tables results in a bigger speedup and bigger executable size.

```
#include <stdint.h>
#include <limits.h>
using namespace std;
```

```
static const uint32_t morton256_x[256] =
{
0x00000000,
0x00000001, 0x00000008, 0x00000009, 0x00000040, 0x00000041, 0x00000048, 0x00000049, 0x00000200,
0x00000201, 0x00000208, 0x00000209, 0x00000240, 0x00000241, 0x00000248, 0x00000249, 0x00001000,
0x00001001, 0x00001008, 0x00001009, 0x00001040, 0x00001041, 0x00001048, 0x00001049, 0x00001200,
0x00001201, 0x00001208, 0x00001209, 0x00001240, 0x00001241, 0x00001248, 0x00001249, 0x00008000,
0x00008001, 0x00008008, 0x00008009, 0x00008040, 0x00008041, 0x00008048, 0x00008049, 0x00008200,
0x00008201, 0x00008208, 0x00008209, 0x00008240, 0x00008241, 0x00008248, 0x00008249, 0x00009000,
0x00009001, 0x00009008, 0x00009009, 0x00009040, 0x00009041, 0x00009048, 0x00009049, 0x00009200,
0x00009201, 0x00009208, 0x00009209, 0x00009240, 0x00009241, 0x00009248, 0x00009249, 0x00040000,
0x00040001, 0x00040008, 0x00040009, 0x00040040, 0x00040041, 0x00040048, 0x00040049, 0x00040200,
0x00040201, 0x00040208, 0x00040209, 0x00040240, 0x00040241, 0x00040248, 0x00040249, 0x00041000,
0x00041001, 0x00041008, 0x00041009, 0x00041040, 0x00041041, 0x00041048, 0x00041049, 0x00041200,
0x00041201, 0x00041208, 0x00041209, 0x00041240, 0x00041241, 0x00041248, 0x00041249, 0x00048000,
0x00048001, 0x00048008, 0x00048009, 0x00048040, 0x00048041, 0x00048048, 0x00048049, 0x00048200,
0x00048201, 0x00048208, 0x00048209, 0x00048240, 0x00048241, 0x00048248, 0x00048249, 0x00049000,
0x00049001, 0x00049008, 0x00049009, 0x00049040, 0x00049041, 0x00049048, 0x00049049, 0x00049200,
0x00049201, 0x00049208, 0x00049209, 0x00049240, 0x00049241, 0x00049248, 0x00049249, 0x00200000,
0x00200001, 0x00200008, 0x00200009, 0x00200040, 0x00200041, 0x00200048, 0x00200049, 0x00200200,
0x00200201, 0x00200208, 0x00200209, 0x00200240, 0x00200241, 0x00200248, 0x00200249, 0x00201000,
0x00201001, 0x00201008, 0x00201009, 0x00201040, 0x00201041, 0x00201048, 0x00201049, 0x00201200,
0x00201201, 0x00201208, 0x00201209, 0x00201240, 0x00201241, 0x00201248, 0x00201249, 0x00208000,
0x00208001, 0x00208008, 0x00208009, 0x00208040, 0x00208041, 0x00208048, 0x00208049, 0x00208200,
0x00208201, 0x00208208, 0x00208209, 0x00208240, 0x00208241, 0x00208248, 0x00208249, 0x00209000,
0x00209001, 0x00209008, 0x00209009, 0x00209040, 0x00209041, 0x00209048, 0x00209049, 0x00209200,
0x00209201, 0x00209208, 0x00209209, 0x00209240, 0x00209241, 0x00209248, 0x00209249, 0x00240000,
0x00240001, 0x00240008, 0x00240009, 0x00240040, 0x00240041, 0x00240048, 0x00240049, 0x00240200,
0x00240201, 0x00240208, 0x00240209, 0x00240240, 0x00240241, 0x00240248, 0x00240249, 0x00241000,
0x00241001, 0x00241008, 0x00241009, 0x00241040, 0x00241041, 0x00241048, 0x00241049, 0x00241200,
0x00241201, 0x00241208, 0x00241209, 0x00241240, 0x00241241, 0x00241248, 0x00241249, 0x00248000,
0x00248001, 0x00248008, 0x00248009, 0x00248040, 0x00248041, 0x00248048, 0x00248049, 0x00248200,
0x00248201, 0x00248208, 0x00248209, 0x00248240, 0x00248241, 0x00248248, 0x00248249, 0x00249000,
0x00249001, 0x00249008, 0x00249009, 0x00249040, 0x00249041, 0x00249048, 0x00249049, 0x00249200,
0x00249201, 0x00249208, 0x00249209, 0x00249240, 0x00249241, 0x00249248, 0x00249249
};

// pre-shifted table for Y coordinates (1 bit to the left)
static const uint32_t morton256_y[256] = {
0x00000000,
0x00000002, 0x00000010, 0x00000012, 0x00000080, 0x00000082, 0x00000090, 0x00000092, 0x00000400,
0x00000402, 0x00000410, 0x00000412, 0x00000480, 0x00000482, 0x00000490, 0x00000492, 0x00002000,
0x00002002, 0x00002010, 0x00002012, 0x00002080, 0x00002082, 0x00002090, 0x00002092, 0x00002400,
0x00002402, 0x00002410, 0x00002412, 0x00002480, 0x00002482, 0x00002490, 0x00002492, 0x00010000,
0x00010002, 0x00010010, 0x00010012, 0x00010080, 0x00010082, 0x00010090, 0x00010092, 0x00010400,
0x00010402, 0x00010410, 0x00010412, 0x00010480, 0x00010482, 0x00010490, 0x00010492, 0x00012000,
0x00012002, 0x00012010, 0x00012012, 0x00012080, 0x00012082, 0x00012090, 0x00012092, 0x00012400,
0x00012402, 0x00012410, 0x00012412, 0x00012480, 0x00012482, 0x00012490, 0x00012492, 0x00080000,
0x00080002, 0x00080010, 0x00080012, 0x00080080, 0x00080082, 0x00080090, 0x00080092, 0x00080400,
0x00080402, 0x00080410, 0x00080412, 0x00080480, 0x00080482, 0x00080490, 0x00080492, 0x00082000,
0x00082002, 0x00082010, 0x00082012, 0x00082080, 0x00082082, 0x00082090, 0x00082092, 0x00082400,
0x00082402, 0x00082410, 0x00082412, 0x00082480, 0x00082482, 0x00082490, 0x00082492, 0x00090000,
0x00090002, 0x00090010, 0x00090012, 0x00090080, 0x00090082, 0x00090090, 0x00090092, 0x00090400,
0x00090402, 0x00090410, 0x00090412, 0x00090480, 0x00090482, 0x00090490, 0x00090492, 0x00092000,
0x00092002, 0x00092010, 0x00092012, 0x00092080, 0x00092082, 0x00092090, 0x00092092, 0x00092400,
0x00092402, 0x00092410, 0x00092412, 0x00092480, 0x00092482, 0x00092490, 0x00092492, 0x00400000,
0x00400002, 0x00400010, 0x00400012, 0x00400080, 0x00400082, 0x00400090, 0x00400092, 0x00400400,
0x00400402, 0x00400410, 0x00400412, 0x00400480, 0x00400482, 0x00400490, 0x00400492, 0x00402000,
0x00402002, 0x00402010, 0x00402012, 0x00402080, 0x00402082, 0x00402090, 0x00402092, 0x00402400,
0x00402402, 0x00402410, 0x00402412, 0x00402480, 0x00402482, 0x00402490, 0x00402492, 0x00410000,
0x00410002, 0x00410010, 0x00410012, 0x00410080, 0x00410082, 0x00410090, 0x00410092, 0x00410400,
0x00410402, 0x00410410, 0x00410412, 0x00410480, 0x00410482, 0x00410490, 0x00410492, 0x00412000,
0x00412002, 0x00412010, 0x00412012, 0x00412080, 0x00412082, 0x00412090, 0x00412092, 0x00412400,
0x00412402, 0x00412410, 0x00412412, 0x00412480, 0x00412482, 0x00412490, 0x00412492, 0x00480000,
0x00480002, 0x00480010, 0x00480012, 0x00480080, 0x00480082, 0x00480090, 0x00480092, 0x00480400,
0x00480402, 0x00480410, 0x00480412, 0x00480480, 0x00480482, 0x00480490, 0x00480492, 0x00482000,
0x00482002, 0x00482010, 0x00482012, 0x00482080, 0x00482082, 0x00482090, 0x00482092, 0x00482400,
0x00482402, 0x00482410, 0x00482412, 0x00482480, 0x00482482, 0x00482490, 0x00482492, 0x00490000,
0x00490002, 0x00490010, 0x00490012, 0x00490080, 0x00490082, 0x00490090, 0x00490092, 0x00490400,
0x00490402, 0x00490410, 0x00490412, 0x00490480, 0x00490482, 0x00490490, 0x00490492, 0x00492000,
0x00492002, 0x00492010, 0x00492012, 0x00492080, 0x00492082, 0x00492090, 0x00492092, 0x00492400,
0x00492402, 0x00492410, 0x00492412, 0x00492480, 0x00492482, 0x00492490, 0x00492492
};

// Pre-shifted table for z (2 bits to the left)
static const uint32_t morton256_z[256] = {
0x00000000,
0x00000004, 0x00000020, 0x00000024, 0x00000100, 0x00000104, 0x00000120, 0x00000124, 0x00000800,
0x00000804, 0x00000820, 0x00000824, 0x00000900, 0x00000904, 0x00000920, 0x00000924, 0x00004000,
0x00004004, 0x00004020, 0x00004024, 0x00004100, 0x00004104, 0x00004120, 0x00004124, 0x00004800,
0x00004804, 0x00004820, 0x00004824, 0x00004900, 0x00004904, 0x00004920, 0x00004924, 0x00020000,
0x00020004, 0x00020020, 0x00020024, 0x00020100, 0x00020104, 0x00020120, 0x00020124, 0x00020800,
0x00020804, 0x00020820, 0x00020824, 0x00020900, 0x00020904, 0x00020920, 0x00020924, 0x00024000,
0x00024004, 0x00024020, 0x00024024, 0x00024100, 0x00024104, 0x00024120, 0x00024124, 0x00024800,
0x00024804, 0x00024820, 0x00024824, 0x00024900, 0x00024904, 0x00024920, 0x00024924, 0x00100000,
0x00100004, 0x00100020, 0x00100024, 0x00100100, 0x00100104, 0x00100120, 0x00100124, 0x00100800,
0x00100804, 0x00100820, 0x00100824, 0x00100900, 0x00100904, 0x00100920, 0x00100924, 0x00104000,
0x00104004, 0x00104020, 0x00104024, 0x00104100, 0x00104104, 0x00104120, 0x00104124, 0x00104800,
0x00104804, 0x00104820, 0x00104824, 0x00104900, 0x00104904, 0x00104920, 0x00104924, 0x00120000,
0x00120004, 0x00120020, 0x00120024, 0x00120100, 0x00120104, 0x00120120, 0x00120124, 0x00120800,
0x00120804, 0x00120820, 0x00120824, 0x00120900, 0x00120904, 0x00120920, 0x00120924, 0x00124000,
0x00124004, 0x00124020, 0x00124024, 0x00124100, 0x00124104, 0x00124120, 0x00124124, 0x00124800,
0x00124804, 0x00124820, 0x00124824, 0x00124900, 0x00124904, 0x00124920, 0x00124924, 0x00800000,
0x00800004, 0x00800020, 0x00800024, 0x00800100, 0x00800104, 0x00800120, 0x00800124, 0x00800800,
0x00800804, 0x00800820, 0x00800824, 0x00800900, 0x00800904, 0x00800920, 0x00800924, 0x00804000,
0x00804004, 0x00804020, 0x00804024, 0x00804100, 0x00804104, 0x00804120, 0x00804124, 0x00804800,
0x00804804, 0x00804820, 0x00804824, 0x00804900, 0x00804904, 0x00804920, 0x00804924, 0x00820000,
0x00820004, 0x00820020, 0x00820024, 0x00820100, 0x00820104, 0x00820120, 0x00820124, 0x00820800,
0x00820804, 0x00820820, 0x00820824, 0x00820900, 0x00820904, 0x00820920, 0x00820924, 0x00824000,
0x00824004, 0x00824020, 0x00824024, 0x00824100, 0x00824104, 0x00824120, 0x00824124, 0x00824800,
0x00824804, 0x00824820, 0x00824824, 0x00824900, 0x00824904, 0x00824920, 0x00824924, 0x00900000,
0x00900004, 0x00900020, 0x00900024, 0x00900100, 0x00900104, 0x00900120, 0x00900124, 0x00900800,
0x00900804, 0x00900820, 0x00900824, 0x00900900, 0x00900904, 0x00900920, 0x00900924, 0x00904000,
0x00904004, 0x00904020, 0x00904024, 0x00904100, 0x00904104, 0x00904120, 0x00904124, 0x00904800,
```

```
0x00904804, 0x00904820, 0x00904824, 0x00904900, 0x00904904, 0x00904920, 0x00904924, 0x00920000,
0x00920004, 0x00920020, 0x00920024, 0x00920100, 0x00920104, 0x00920120, 0x00920124, 0x00920800,
0x00920804, 0x00920820, 0x00920824, 0x00920900, 0x00920904, 0x00920920, 0x00920924, 0x00924000,
0x00924004, 0x00924020, 0x00924024, 0x00924100, 0x00924104, 0x00924120, 0x00924124, 0x00924800,
0x00924804, 0x00924820, 0x00924824, 0x00924900, 0x00924904, 0x00924920, 0x00924924
};

inline uint64_t mortonEncode_LUT(unsigned int x, unsigned int y, unsigned int z){
uint64_t answer = 0;
answer = morton256_z[(z >> 16) & 0xFF ] | // we start by shifting the third byte, since we only look at the first
morton256_y[(y >> 16) & 0xFF ] |
morton256_x[(x >> 16) & 0xFF ];
answer = answer << 48 | morton256_z[(z >> 8) & 0xFF ] | // shifting second byte
morton256_y[(y >> 8) & 0xFF ] |
morton256_x[(x >> 8) & 0xFF ];
answer = answer << 24 |
morton256_z[(z) & 0xFF ] | // first byte
morton256_y[(y) & 0xFF ] |
morton256_x[(x) & 0xFF ];
return answer;
}
```

Performance comparison

I used the following code to benchmark the methods:

```
#define MAX 256
int main(int argc, char *argv[]) {
Timer t;
t.reset(); t.start();
for(size_t i = 0; i < MAX; i++){
for(size_t j = 0; j < MAX; j++){
for(size_t k = 0; k < MAX; k++){
mortonEncode(i,j,k) ;
}
}
}
t.stop();
```

And these are the results, tested at MAX=64, 128 and 256. As you can see, the Magic Bits and LUT methods are an order of magnitude faster than the basic for loop method (times in seconds)

MAX = 64	MAX = 128	MAX = 256	
For-loop	0.2	1.6	13.13
Magic Bits	0.01	0.13	1.06
LUT	0.005	0.041	0.319

Conclusion

If it’s on your critical path, it’s probably a good idea to opt for the Magic Bits method for a quick speedup. If you’re willing to put in a bit more effort and generate the tables (you can do that using the splitBy3 or similar method, btw), the big old Lookup Table gives the best performance.