Spatial binning and hashing

In the smoothed particle hydrodynamics simulation, particles interact only with those particles that are within a circle of radius h of them. In the naive reference code, a substantial fraction of the total time is spend finding which pairs of particles interact, and the cost of finding interacting pairs scales quadratically with the number of particles in the simulation. But the particles in the simulation never get too close together, and so any given particle will typically only interact with a bounded number of neighbors. We can therefore make the simulation much faster by checking for interactions only between particles that are close enough that they could conceivably interact.

One simple way to limit the number of interactions we check is to partition space into fixed-size bins. In particular, if we partition space into bins with side length $l \geq h$, then a particle in a particular bin can interact with at most the other particles in that bin and in the 26 neighboring bins. If $l \geq 2h$, then a particle in a particular bin can interact with at most the other particles in that bin and particles in 7 other neighboring bins; see Figure 1

I do something very simple: I represent each particle as a structure and use a linked list structure for each bin. The particle structure has the form

```
typedef struct particle_t {
  // Position, velocity, acceleration, etc.
  struct particle_t* next; // Next in bin
} particle_t;
```

and I have an array of head pointers `particle_t* hash[]`, one for each bin. Before computing interactions at each time step, I clear the `bins` pointers, and then re-build the bins with a loop of the form

```
for (int i = 0; i < n; ++i) {

    // Figure out bin for particle i
    int b = ...;

    // Add particle to the start of the list for bin b
    part[i].next = bins[b];
    bins[b] = part[i];
}
```

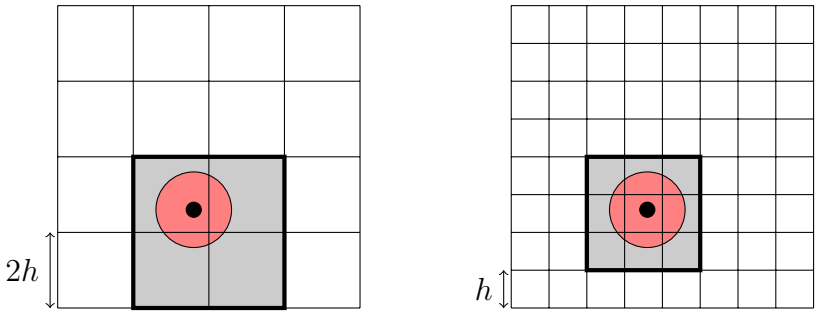

Figure 1: In 2D, if we partition space into bins of width $2h$, we need to check four bins for particles that might interact with a given particle (left). If we use bins of width h, we need to check eight bins, though the total area covered is smaller (right). In 3D, we would need to check eight or 27 bins, respectively.

Then when I want to compute interactions for particle i, I only need to look up those particles in the bin where particle i resides, along with a few neighboring bins.

Partitioning space into bins that are $2h$ on a side is reasonable when a large fraction of the total volume is filled with particles, as is the case for the dam break simulation that we run by default. But this is not such a reasonable approach if only a small fraction of the volume in the computational domain is filled with particles. In this case, we could spend an inordinate amount of memory on empty bins if we were to represent them all explicitly. Instead, we could use *spatial hashing*, which is a fancy way of saying that we index the buckets by a hash of the bin identifier. As long as the hash table is around the same size as the number of particles, we expect not to have too many collisions; and for any given particle, it is still easy to look up the hash buckets that would contain any particles with which it might interact.

For partitioning work across processors, or to get a good key for spatial hashing, it makes sense to map each triple of bin indices (or pairs in 2D) into a single index in a way that maintains some spatial locality. A particularly good choice is the *Hilbert index* (a classic space-filling curve), but a choice that is both easy to understand and to compute is the Z-Morton curve. We show a two-dimensional example in Figure 2. The Z-Morton curve effectively interlaces the bits of the individual coordinate indices in order to get a com-

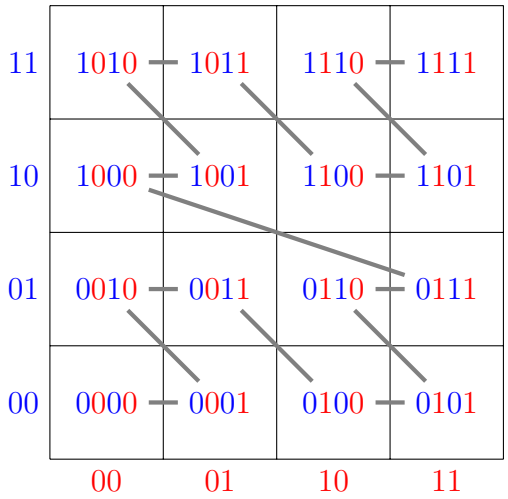

Figure 2: Z-Morton ordering in 2D. By interleaving the bits of the x and y bin indices, we obtain a bin ordering that maintains some spatial locality.

bined index; visually, this gives a curve that looks like a collection of nested "Z"s, which explains the name. The bit-fiddling to actually accomplish this index is a bit opaque, but I've provided you with some routines to take care of it in the reference code.

Of course, giving you this much code still leaves you with a number of practical issues to address, such as how to index the bins and how to keep using symmetry in the computations of the interactions between particles! And there's nothing that says that you have to structure your computation in the same way that I do. For example, you could eschew pointers entirely and just use a bucket sort algorithm[1] to re-order your particles at each step so that particles in the same bin are in contiguous array locations[2]. Or you could use a Hilbert curve rather than a Z-Morton curve. Or you could periodically compute a list of potential interactions for the next several steps, and partition the interaction forces rather than partitioning particles.

[1] I'm happy to tell you about bucket sort if you come ask me, but you might be better of just doing a Google search (or looking in a favorite algorithms textbook).

[2] I actually do a sort in my code in order to improve locality, but only once every hundred steps or so