



MICK WEST

THE INNER PRODUCT

EXPONENTIAL OPTIMIZING

MANY TASKS IN GAME PROGRAMMING

involve dealing with a large set of objects, where any object in that set can theoretically be affected by the state of any other object in that set. A good example is any type of physics object, such as bricks that need to potentially collide with all other bricks. A linear increase in the number of objects will result in an exponential increase in the potential number of interactions that need to be considered.

Solutions to this problem usually involve some kind of partitioning, so that the entire set of objects does not need to be considered at once. A brick need only check for collisions with other bricks that it knows are nearby. The ways in which the exponential increase in complexity can be addressed by partitioning the data are theoretically reasonably well understood. However, the practical realities of target hardware limitations and of the types of events and situations we have to handle in computer games make for some interesting complications. I'll explore a couple of simple examples of this partitioning, and discuss potential complications.

SORTING

Sorting is not usually considered an interesting problem in computer science. The basic algorithms were all discovered many decades ago, and they are very well understood. However, the specifics of computer games and

computer game hardware (consoles in particular) make an investigation of sorting quite interesting. While the theoretical performance of algorithms is straightforward, your actual results are affected by several complex factors.

Before you dive head first into optimizing your sorting functions, you might want to consider whether you're optimizing prematurely. If you hardly ever do sorting on a per-frame basis, and it's only for small lists of a few tens of items, then it's unlikely you need to optimize. If, on the other hand, you have a list of 5,000 particles that you need to sort for a transparency draw order every frame, then it's a pretty safe bet that the sort time is going to be an issue. Because a change in a sort algorithm can result in a significant increase in memory usage and potential new bugs, it partially qualifies as a mature optimization in that it is best done early in the project, before you know if you actually need it. [See "Mature Optimization," January 2006.]

The most common trivial sort algorithm I use is the selection sort, where you simply scan the list, find the lowest item, put it at the start of the list, then repeat for the rest of the list. It's pretty easy to implement, and for short lists it's reasonably efficient, as it has no overhead and minimizes memory writes. Figure 1 shows what happens in the selection sort. For each pass, each item is compared to the lowest value found so far and is then inserted at the start of the list. Note: In these examples, I'm just sorting an array of numbers into ascending order; more typically you'd be sorting larger items based on some value or key.

It's easy to see the run time of this algorithm is going to be exponentially proportional to the length of the list. It requires a fixed $n*(n-1)/2$ comparison: it is essentially $O(n^2)$. This makes it impractical for large lists.

QUICKER SORTING

All good programmers know that there are $O(n^2)$ and $O(n \log n)$ sort algorithms, so your next step is going to be to implement something like the quicksort algorithm. Quicksort is slightly more complex than selection sort, but not excessively so. It's well documented, and there are many implementations. You basically choose one element in the list and create lists of higher and lower elements, and then recursively do the same to those lists until all the lists are one item long, at which point you join all the lists together, and they will be sorted.

CONTINUED ON PG 38

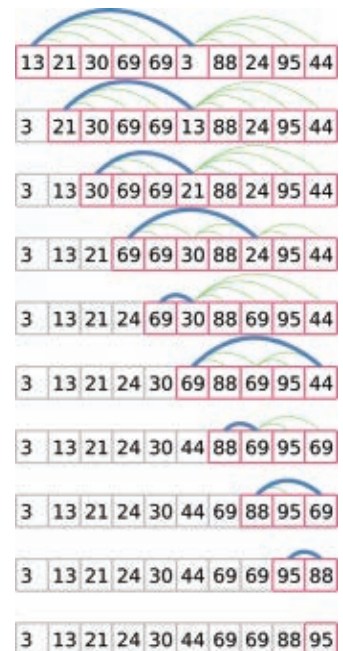


FIGURE 1 Selection sort: All items in the list are tested by comparing them to the lowest value found so far (faint arcs), and the smallest is moved to the start of the list (bold arcs). This is repeated until the whole list is sorted.

MICK WEST was a co-founder of Neversoft Entertainment. He's been in the game industry for 19 years and currently works as a technical consultant. Email him at mwest@gdmag.com.

CONTINUED FROM PG 36

You can do this in place, so you won't need any additional storage.

Why would you not use quicksort? For very small lists, typically fewer than 15 items, it's actually slower than the selection sort. This is due to the additional overhead and the increased number of memory writes. The last four lines of Table 1 show the typical number of memory reads and writes for simple implementations of quicksort (QS) and selection sort (SS) for 10, 20, 40, 80, etc., items. Note that for 10 items, the selection sort uses fewer reads and writes, and for 20 items or more it always uses fewer writes.

The analysis of memory reads and writes is important if you want to understand why an algorithm performs as it does in terms of execution time. On some architectures, memory writes are slower than memory reads, so the total time is not simply a sum of memory accesses. Cache obviously plays a big part, and as such the patterns with which memory are accessed can play a role.

Theoretically, a bubble sort could be fastest in certain situations.

The quicksort scales very well for larger numbers. With more than 20 items, the quicksort is a clear winner over selection sort, which quickly explodes into an exponential number of memory reads (818,000 for selection sort, versus 18,260 for quicksort).

QUICKEST SORT

The biggest reason not to use quicksort is that it's not actually the quickest sort algorithm for list of the size that you typically find in video games. Now we enter the realm of partitioning, and hence to bin sort, also known as the bucket sort.

In this algorithm [see Figure 2], you simply divide up the key space into a number of bins and place each item into the corresponding bin. The bins are then sorted (which is often done as an insertion sort when adding the item), and then concatenated to give the sorted list.

Theoretically, this method can give us a run time of $O(n)$. But the actual time is greatly affected by the number of bins used. Too few bins and your time devolves back to the time taken to sort a bin; too many bins and there is excessive overhead in maintaining them. Table 1 shows the number of reads

(usually about the same as the number of writes) for a bin sort with various length lists, sorted using various bin sizes. You can see that with 10 bins, the performance is roughly comparable to a quicksort for low numbers, but begins to exponentially increase at a certain value. As we increase the number of bins, the performance increases for the longer lists. However, if the number of bins is increased too far, the performance can actually decrease. This is illustrated by the area of the table shaded in orange where the overhead of the large number of bins outweighs the performance gain.

It's interesting to compare the performance of the three algorithms. Selection sort is the fastest and simplest for a low number of elements (15 or fewer). Quicksort scales very well for large numbers. Bin sort can always outperform quicksort for a regular distribution, although you need to be careful in choosing the number of bins. Too few bins, and you'd be far better off using a quicksort. Having too many bins is better than having too few, but there's a sweet spot to which you would need to tune your algorithm. This depends on various factors of your implementation.

Another downside of the bin sort is that it requires quite a lot of additional memory. This is temporary memory, so you could perhaps use something that will be re-used later like a vertex

TABLE 1 Bin sort, quicksort, and selection sort.

Bins	Number of Elements to be Sorted									
	10	20	40	80	160	320	640	1280	2560	5120
10	42	88	193	459	1200	3803	12701	47148	174332	681551
20	52	87	170	389	945	2385	7620	25607	93950	351134
30	61	98	179	349	798	2099	5741	18721	65961	237101
40	71	105	177	343	770	1792	5079	14978	51437	187979
50	80	115	183	350	723	1712	4499	13061	42937	149393
60	91	122	196	343	734	1638	4146	11534	37584	129619
70	100	133	196	366	702	1576	3946	10821	33002	114995
80	111	143	211	350	694	1520	3747	10095	30694	102148
90	121	152	220	370	686	1485	3618	9431	28077	92399
100	130	164	232	367	696	1453	3496	9157	25976	86473
110	140	172	237	374	699	1425	3333	8613	24524	80850
120	150	182	248	379	680	1411	3262	8220	23794	74732
130	161	192	255	389	703	1400	3210	7875	22183	71207
140	170	201	262	396	715	1394	3157	7856	21625	67138
150	181	210	273	408	709	1379	3117	7493	20609	63250
160	190	222	285	420	712	1375	3034	7368	20332	59975
170	200	231	294	429	724	1381	2994	7241	19407	58497
180	211	241	304	437	730	1379	2998	7181	18905	56918
190	221	251	315	444	726	1396	2937	6952	18619	54967
200	230	260	325	456	736	1389	2952	6980	18091	52645
QS-read	48	110	279	728	1612	3668	8100	18260	42480	90478
QS-write	20	46	108	251	572	1296	2926	6365	13894	30424
SS-read	45	190	780	3160	12720	51040	204480	818560	3275520	13104640
SS-write	14	34	72	154	306	630	1260	2554	5094	10220

The table shows a comparison of the number of reads required to sort variously sized lists using bin sort, quicksort, and selection sort.

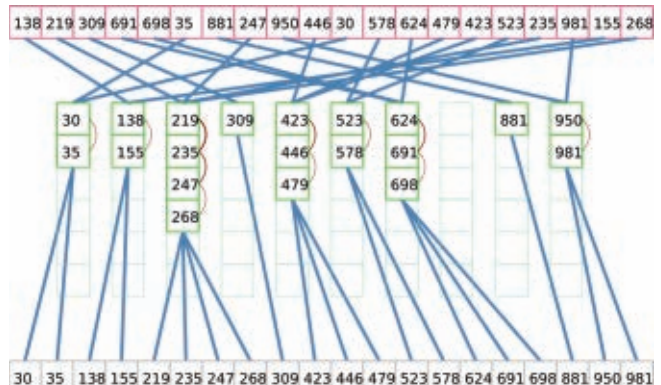


FIGURE 2 Bin sort: Items are placed in sorted buckets and are then concatenated. This method is more complex and memory-hungry, but generally works best for longer lists.

buffer. But this extra memory can still be a problem. Bin sort is also vulnerable to items being bunched up. If a large number of items goes into the same slot, then the performance will massively degrade, unlike quicksort. It's important to consider these cases when choosing an algorithm.

OBJECT INTERACTIONS

Another common task with $O(n^2)$ complexity is object interactions, particularly object collisions. Every object in the world can theoretically collide with every other object, but obviously they are limited to objects that are close enough. You need not consider objects that are half a mile away, and obvious optimizations arise where you can group objects that are close to each other.

Figure 3 shows a scattering of 200 objects in a world. Each object is a red dot, and the circles represent the distances at which the objects need to test each other for collisions. If two circles overlap, then tests must be performed. By only testing the overlapping objects, the number of tests is reduced from a theoretical 40,000 to a more manageable 265.

How do we know which circles overlap? The most common technique to use here is to have a grid that overlays the world. The grid is an array of lists of objects.

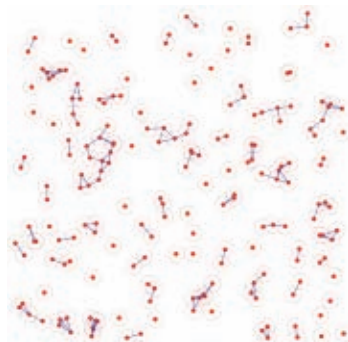


FIGURE 3 The red dots represent 200 objects in a world, and the circles show the distance at which the objects need to be tested for collision. Objects need a collision test if their circles overlap, but this requires n^2 tests.

Whenever an object moves, it tells the grid which cell it is in, and the grid updates the list of objects in that grid cell. This requires very little time to update.

Then, when wanting to find which objects are nearby, we only need to check the lists of objects that are in nearby cells. Figure 4 shows a grid of 10x10 cells. By checking only the objects in the same or adjacent cells, the number of checks is reduced from 40,000 to 1,614. This is still significantly more than the theoretical minimum of 265, but a vast improvement nonetheless.

As with the bin sort method, we can increase performance by increasing the resolution of our partitioning. Figure 5 shows a 20x20 grid, and the number of checks is now down to 536.

This grid method is similar to bin sort in that it uses simple criteria to place objects in slots. The simple $O(n^2)$ comparisons of every object against every other object is analogous to the selection sort. As with the selection sort, the simple method of checking all objects can actually be the fastest method when there are only a few objects. It's very simple, and generally requires few computations. The grid method, on the other hand, suffers from the same problems as the bin sort method—it uses up extra memory. Too fine a grid can actually slow things down, and if objects bunch together, their performance can degrade massively.

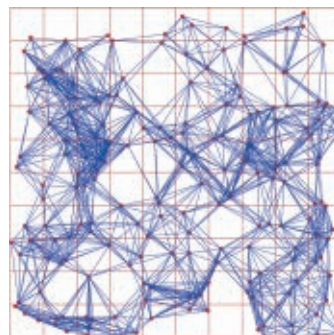


FIGURE 4 By partitioning the world, we only test those object in nearby cells.

TRADEOFFS

Optimization is about trade-offs. The examples in this article are quite simple, but are illustrative of the problems that can arise in typical game development. Here we've seen four primary trade-offs:

1. Faster algorithms are generally more complex. This increases your development time, but also increases the possibility of new bugs.
2. Speed can often only be bought at the expense of additional memory usage. This is a classic trade-off.
3. Faster algorithms have less predictable behavior. Quicksort is generally $O(n \log n)$, but can devolve to $O(n^2)$. Bin sort has an even wider range of possibilities, ranging from $O(n)$ to $O(n^2)$. The worst-case behavior may render pointless any optimization you got in other cases, and it might be best to go with a slow and reliable method.
4. Partitioning algorithms such as bin sort or grid-based proximity detection algorithms vary greatly in their results based on the number of elements, the resolution of the grid (or bins), and the spatial distribution of the elements. The ideal configuration is difficult to discern, and may require careful tuning. ❌

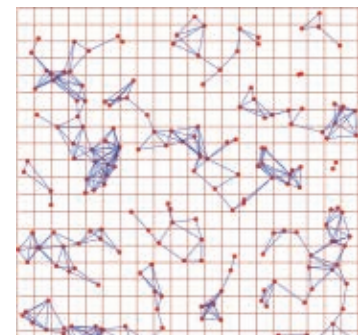


FIGURE 5 Using a higher resolution means fewer tests, but increases memory usage.