15

High-Performance Programming with Data-Oriented Design

Noel Llopis
Snappy Touch

Common programming wisdom used to encourage delaying optimizations until later in the project, and then optimizing only those parts that were obvious bottlenecks in the profiler. That approach worked well with glaring inefficiencies, like particularly slow algorithms or code that is called many times per frame. In a time when CPU clock cycles were a good indication of performance that was a good approach to follow. Things have changed a lot in today's hardware, and we have all experienced the situation where, after fixing the obvious culprits, no single function stands out in the profiler but performance remains subpar. Dataoriented design helps address this problem by architecting the game with memory accesses and parallelization from the beginning.

15.1 Modern Hardware

Modern hardware can be characterized by having multiple execution cores and deep memory hierarchies. The reason for the complex memory hierarchies is due to the gap between CPU power and memory access times. Gone are the days when CPU instructions took about the same time as a main memory access. Instead, this gap continues to increase and shows no signs of stopping (see Figure 15.1).

Different parts of the memory hierarchy have different access times. The smaller ones closer to the CPU are the fastest ones, whereas main memory can be really large, but also very slow. Table 15.1 lists some common access times for different levels of the hierarchy on modern platforms.

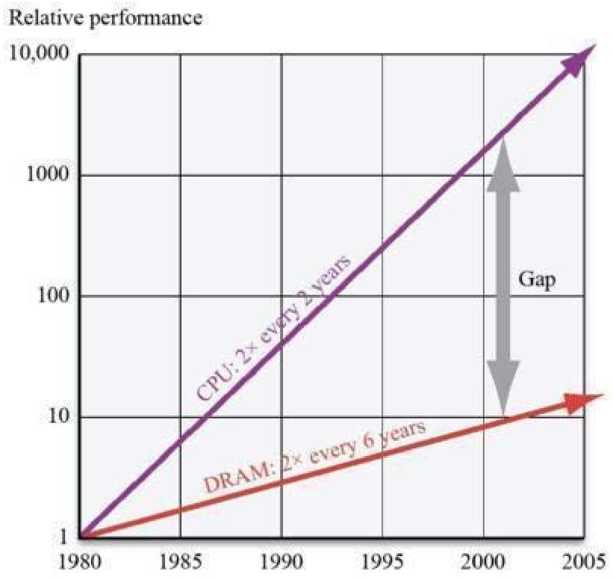

Figure 15.1. Relative CPU and memory performance over time.

With these kinds of access times, it's very likely that the CPU is going to stall waiting to read data from memory. All of a sudden, performance is not determined so much by how efficient the program executing on the CPU is, but how efficiently it uses memory.

Barring a radical technology change, this is not a situation that's about to change anytime soon. We'll continue getting more powerful, wider CPUs and larger memories that are going to make memory access even more problematic in the future.

Looking at code from a memory access point of view, the worst-case situation would be a program accessing heterogeneous trees of data scattered all over memory, executing different code at each node. There we get not just the constant data cache misses but also bad instruction cache utilization because it's calling different functions. Does that sound like a familiar situation? That's how most modern games are architected: large trees of different kinds of objects with polymorphic behavior.

What's even worse is that bad memory access patterns will bring a program down to its metaphorical knees, but that's not a problem that's likely to appear anywhere in the profiler. Instead, it will result in the common situation of everything being slower than we expected, but us not being able to point to a particular spot. That's because there isn't a single place that we can fix. Instead, we need to change the whole architecture, preferably from the beginning, and use a dataoriented approach.

Table 15.1. Access times for different levels of the memory hierarchy for modem platforms.

Memory Level	CPU Cycles per Access
Register	1
L1 cache	5-8
L2 cache	30-50
Main memory	500+

15.2 Principles of Data-Oriented Design

Before we can talk about data-oriented design, we need to step back and think about what a computer program is. One of the most common definitions of a computer program is "a sequence of instructions to perform a task." That's a reasonably good definition, but it concentrates more on the hows rather than on the whys. What are those instructions for? Why are we writing that program?

A more general definition of a computer program is "something that transforms input data into output data." At first glance, some people might disagree with this definition. It might be true for the calculations in a spreadsheet, but is it really a good description for a game? Definitely. In a game, we have a set of input data: the clock value, the game controller state, network packets, and the state of the game during the previous frame. The outputs we're calculating are a new game state, a set of commands for the graphics processor, sound, network packets, etc. It's not very different from a spreadsheet, except that it runs many times per second, at interactive rates.

The emphasis in Computer Science and Engineering is to concentrate on algorithms and code architecture. In particular, procedural programming focuses on procedure and function calls as its main element, while object-oriented programming deals mostly with objects (which are sets of data and the code that works on that data).

Data-oriented design turns that around and considers data first: how it is laid out and how it is read and processed in the program. Then, the code is something written to transform the input data into the output data, but it itself is not the focus.

As a consequence of looking at the input data carefully, we can apply another principle of data-oriented design: where there's one, there are more. How often have you had just one player in the game? Or one enemy? One vehicle? One bullet? Never! Yet somehow, we insist on treating each object separately, in isolation, as if it were the only one in the world. Data-oriented design encourages optimizing for the common case of having multiple objects of the same type.

15.3 Data-Oriented Design Benefits

Data-oriented design has three major performance benefits:

1. *Cache utilization*. This is the big one that motivated us to look at data in the first place. Because we can concentrate on data and memory access instead of the algorithms themselves, we can make sure our programs have as close to an ideal memory access pattern as possible. That means avoiding heterogeneous trees, organizing our data into large sequential blocks of homogeneous memory, and processing it by running the same code on all of its elements. This alone can bring a huge speed-up to our code.

2. *Parallelization*. When we work from the data point of view, it becomes a lot easier to divide work up into parts that different cores can process simultaneously with minimal synchronization. This is true for almost any kind of parallel architecture, whether each core has access to main memory or not.

3. *Less code*. People are often surprised at this one. As a consequence of looking at the data and only writing code to transform input data into output data, there is a lot of code that disappears. Code that before was doing boring bookkeeping, or getter/setters on objects, or even unnecessary abstractions, all go away. And simplifying code is very much like simplifying an algebraic equation: once you make a simplification, you often see other ways to simplify it further and end up with a much smaller equation than you started with.

When a technique promises higher performance, it often comes at a cost in some other department, usually in terms of readability or ease of maintenance. Data-oriented design is pretty unique in that it also has major benefits from a development perspective:

1. *Easier to test*. When your code is something that simply transforms input data into output data, testing it becomes extremely simple. Feed in some test input data, run the code, and verify the output data is what you expected. There are no pesky global variables to deal

with, calls to other systems, interaction with other objects, or mocks to write. It really becomes that simple.

2. *Easier to understand*. Having less code means not just higher performance but also less code to maintain, understand, and keep straight in our heads. Also, each function in itself is much simpler to understand. We're never in the situation of having to chase function call after function call to understand all the consequences of one function. Everything you want to know about it is there, without any lower-level systems involved.

To be fair and present all the sides, there are two disadvantages to dataoriented design:

1. *It's different*. So far, data-oriented design isn't taught in Computer Science curricula, and most developers aren't actively using it, so it is foreign to most team members. It also makes it more difficult to integrate with third-party libraries and APIs that are not data-oriented.

2. *Harder to see the big picture*. Because of the emphasis on data and on small functions that transform data, it might be harder to see and express the big picture of the program: When is an operation happening? Why is this data being transformed? This is something that might be addressed with tools, language extensions, or even a new programming language in the future. For now, we'll have to rely on examining the code and the data carefully.

15.4 How to Apply Data-Oriented Design

Let's get more specific and start applying data-oriented design. Eventually, we'd like the entire game to be architected this way, but we need to start somewhere. So pick a subsystem that needs to be optimized: animation, artificial intelligence, physics, etc.

Next, think about all the data involved in that system. Don't worry too much about how it is laid out in memory, just about what's involved. Apart from the explicit inputs and outputs, don't forget about data that the system accesses explicitly, such as a world navigation graph, or global handle managers.

Once you have identified all the data the system needs, carefully think about how each type of data is used and sort them into read-only, read-write, or writeonly. Those will become your explicit inputs and outputs. It will also allow you to make better decisions about how to lay out the data.

Also, at this point, it's important to think about the amount of data. Does this system ever process more than one of each type of data? If so, start thinking of it in terms of arrays of data, preferably as contiguous blocks of the same data type that can be processed at once.

Finally, the most important step is to look at the data you've identified as input and figure out how it can be transformed into the output data in an efficient way. How does the input data need to be arranged? Normally, you'll want a large block of the same data type, but perhaps, if there are two data types that need to be processed at the same time, interleaving them might make more sense. Or maybe, the transformation needs two separate passes over the same data type, but the second pass uses some fields that are unused in the first pass. In that case, it might be a good candidate for splitting it up into two types and keeping each of them sequentially in memory.

Once you have decided on the transformation, the only thing left is gathering the inputs from the rest of the system and filling the outputs. When you're transitioning from a more traditional architecture, you might have to perform an explicit gathering step-query some functions or objects and collect the input data in the format you want. You'll have to perform a similar operation with the output, feeding it into the rest of the system. Even though those extra steps represent a performance hit, the benefits gained usually offset any performance costs. As more systems start using the data-oriented approach, you'll be able to feed the output data from one system directly into the input of another, and you'll really be able to reap the benefits.

15.5 Real-World Situations

Homogeneous, Sequential Data

You are probably already applying some of the principles of data-oriented design in your games right now: the particle system. It's intended to handle thousands and thousands of particles. The input data is very carefully designed to be small, aligned, and fit in cache lines, and the output data is also very well defined because it probably feeds directly into the GPU.

Unfortunately for us, we don't have that many situations in game development where we can apply the same principle. It may be possible for some sound or image processing, but most other tasks seem to require much more varied data and lots of different code paths.

Heterogeneous Data

Game entities are the perfect example of why the straightforward particle approach doesn't work in other game subsystems. You probably have dozens of different game entity types. Or maybe you have one game entity, but have dozens, or even hundreds, of components that, when grouped together, give entities their own behavior.

One simple step we can take when dealing with large groups of heterogeneous data like that is to group similar data types together. For example, we would lay out all the health components for all entities in the game one right after the other in the same memory block. Same thing with armor components, and every other type of component.

If you just rearranged them and still updated them one game entity at a time, you wouldn't see any performance improvements. To gain a significant performance boost, you need to change the update from being entity-centric to being component-centric. You need to update all health components first, then all armor components, and proceed with all component types. At that point, your memory access patterns will have improved significantly, and you should be able to see much better performance.

Break and Batch

It turns out that sometimes even updating a single game entity component seems to need a lot of input data, and sometimes it's even unpredictable what it's going to need. That's a sign that we need to break the update into multiple steps, each of them with smaller, more predictable input data.

For example, the navigation component casts several rays into the world. Since the ray casts happen as part of the update, all of the data they touch is considered input data. In this case, it means that potentially the full world and other entities' collision data are part of the input data! Instead of collecting that data ahead of time and feeding it as an input into each component update, we can break the component update into two parts. The initial update figures out what ray casts are required and generates ray-cast queries as part of its output. The second update takes the results from the ray casts requested by the first update and finishes updating the component state.

The crucial step, once again, is the order of the updates. What we want to do is perform the first update on all navigation components and gather all the raycast queries. Then we take all those queries, cast all those rays, and save the results. Finally, we do another pass over the navigation components, calling the second update for each of them and feeding them the results of the ray queries.

Notice how once again, we managed to take some code with that tree-like memory access structure and turn it into something that is more linear and works over similar sets of data. The ray-casting step isn't the ideal linear traversal, but at least it's restricted to a single step, and maybe the world collision data might fit in some cache so we won't be getting too many misses to main memory.

Once you have this implemented, if the ray-casting part is still too slow, you could analyze the data and try to speed things up. For example, if you often have lots of grouped ray casts, it might be beneficial to first sort the ray casts spatially, and when they're being resolved, you're more likely to hit data that is already cached.

Conditional Execution

Another common situation is that not all data of the same type needs to be updated the same way. For example, the navigation component doesn't always need to cast the same number of rays. Maybe it normally casts a few rays every half a second, or more rays if other entities are closer by.

In that case, we can let the component decide whether it needs a second-pass update by whether it creates a ray-cast query. Now we're not going to have a fixed number of ray queries per entity, so we'll also need a way to make sure we associate the ray cast with the entity it came from.

After all ray casts are performed, we iterate over the navigation components and only update the ones that requested a ray query. That might save us a bit of CPU time, but chances are that it won't improve performance very much because we're going to be randomly skipping components and missing out on the benefits of accessing memory linearly.

If the amount of data needed by the second update is fairly small, we could copy that data as an output for the first update. That way, whenever we're ready to perform the second update, we only need to access the data generated this way, which is sequentially laid out in memory.

If copying the data isn't practical (there's either too much data or that data needs to be written back to the component itself), we could exploit temporal coherence, if there is any. If components either cast rays or don't, and do so for several frames at a time, we could reorder the components in memory so all navigation components that cast rays are found at the beginning of the memory block. Then, the second update can proceed linearly through the block until the last component that requested a ray cast is updated. To be able to achieve this, we need to make sure that our data is easily relocatable.

Polymorphism

Whenever we're applying data-oriented design, we explicitly traverse sets of data of a known type. Unlike an object-oriented approach, we would never traverse a set of heterogeneous data by calling polymorphic functions in each of them.

Even so, while we're transforming some well-known data, we might need to treat another set of data polymorphically. For example, even though we're updating the bullet data (well-known type), we might want to deliver damage to any entity it hits, independent of the type of that entity. Since using classes and inheritance is not usually a very data-friendly approach, we need to find a better alternative.

There are many different ways to go about this, depending on the kind of game architecture you have. One possibility is to split the common functionality of a game entity into a separate data type. This would probably be a very small set of data: a handle, a type, and possibly some flags or indices to components. If every entity in the world has one corresponding set of data of this type, we can always count on it while dealing with other entities. In this case, the bullet data update could check whether the entity has a damage-handling component, and if so, access it and deliver the damage.

If that last sentence left you a bit uncomfortable, congratulations, you're starting to really get a feel for good data access patterns. If you analyze it, the access patterns are less than ideal: we're updating all the current bullets in the world. That's fine because they're all laid out sequentially in memory. Then, when one of them hits an entity, we need to access that entity's data, and then potentially the damage-handling component. That's two potentially random accesses into memory that are almost guaranteed to be cache misses.

We could improve on this a little bit by having the bullet update not access the entity directly and, instead, create a message packet with the damage it wants to deliver to that entity. After we're done updating all of the bullets, we can make another pass over those messages and apply the damage. That might result in a marginal improvement (it's doubtful that accessing the entity and its component is going to cause any cache misses on the following bullet data), but most importantly, it prevents us from having any meaningful interaction with the entity. Is the entity bullet proof? Maybe that kind of bullet doesn't even hit the entity, and the bullet should go through unnoticed? In that case, we really want to access the entity data during the bullet update.

In the end, it's important to realize that not every data access is going to be ideal. Like with all optimizations, the most important ones are the ones that happen more frequently. A bullet might travel for hundreds of frames, and it will hit something at most in one frame. It's not going to make much of a difference if, during the frame when it hits, we have a few extra memory accesses.

15.6 Parallelization

Improving memory access patterns is only part of the performance benefits provided by data-oriented design. The other half is being able to take advantage of multiple cores very easily.

Normally, to split up tasks on different cores, we need to create a description of the job to perform and some of the inputs to the job. Unfortunately, where things fall down for procedural or object-oriented approaches is that a lot of tasks have implicit inputs: world data, collision data, etc. Developers try to work around it by providing exclusive access to data through locking systems, but that's very error prone and can be a big hit on performance depending on how frequently it happens.

The good news is that once you've architected your code such that you're thinking about the data first and following the guidelines in earlier sections, you're ready to parallelize it with very little effort. All of your inputs are clearly defined, and so are your outputs. You also know which tasks need to be performed before other tasks based on which data they consume and produce.

The only part missing is a scheduler. Once you have all of that information about your data and the transformations that need to happen to it, the scheduler can hand off tasks to individual cores based on what work is available. Each core gets the input data, the address where the output data should go, and what kind of transformation to apply to it.

Because all inputs are clearly defined, and outputs are usually new memory buffers, there is often no need to provide exclusive access to any data. Whenever the output data writes back into an area of memory that was used as an input (for example, entity states), the scheduler can make sure there are no jobs trying to read from that memory while the job that writes the output is executing. And because each job is very "shallow" (in the sense that it doesn't perform cascading function calls), the data each one touches is very limited, making the scheduling relatively easy.

If the entire game has been architected this way, the scheduler can then create a complete directed acyclic graph of data dependencies between jobs for each frame. It can use that information and the timings from each previous frame (assuming some temporal coherency) and estimate what the critical path of data transformation is going to be, giving priority to those jobs whenever possible.

One of the consequences of running a large system of data transformations this way is that there are often a lot of intermediate memory buffers. For platforms with little memory available, the scheduler can trade some speed for extra memory by giving priority to jobs that consume intermediate data instead of ones in the critical path.

This approach works well for any kind of parallel architecture, even if the individual cores don't have access to main memory. Since all of the input data is explicitly listed, it can be easily transferred to the core local memory before the transformation happens.

Also, unlike the lock-based parallelization approaches, this method scales very well to a large number of cores, which is clearly the direction future hardware is going toward.

15.7 Conclusion

Data-oriented design is a departure from traditional code-first thinking. It addresses head-on the two biggest performance problems in modern hardware: memory access and parallelization. By thinking about programs as instructions to transform data and thinking first about how that data should be laid out and worked on, we can get huge performance boosts over more traditional software development approaches.