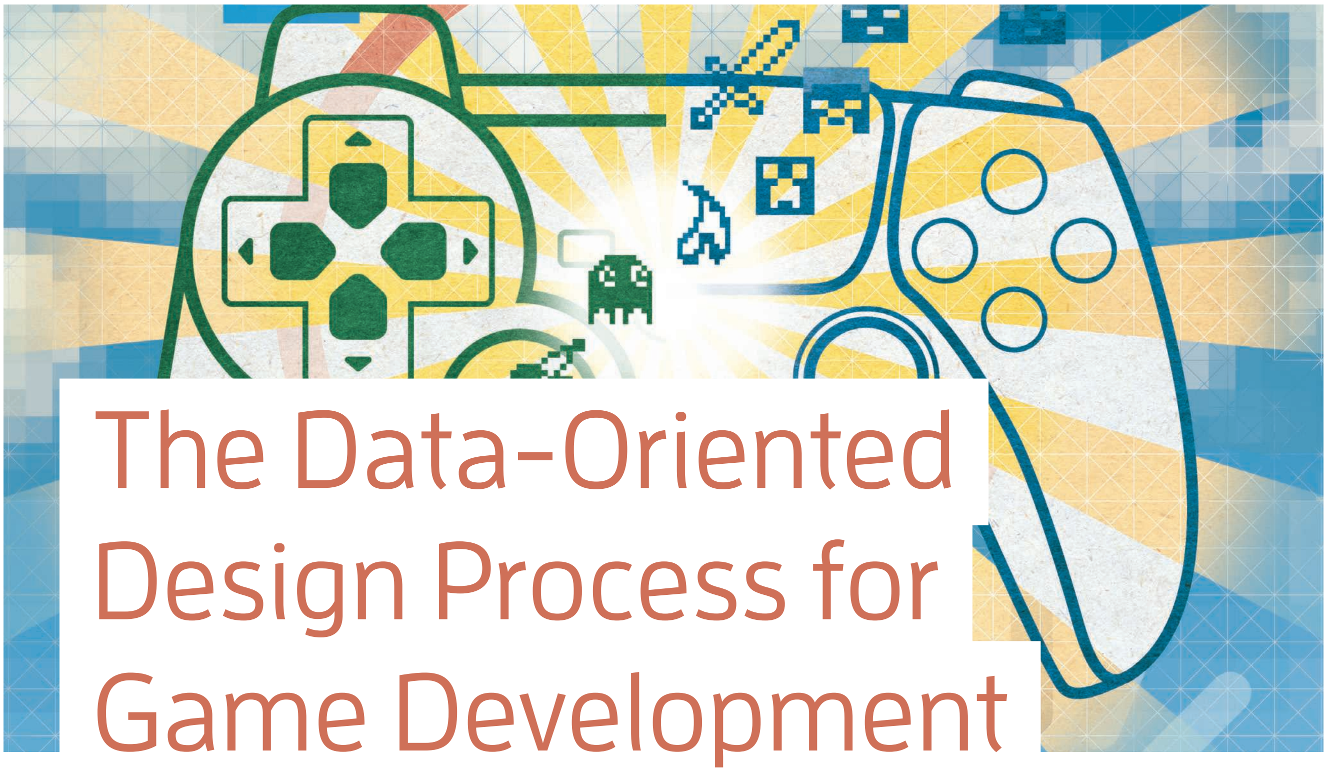

The Data-Oriented Design Process for Game Development

Jessica D. Bayliss, Rochester Institute of Technology & Unity Technologies

Data-oriented design is a growing software development process for games that has not been well studied in academia. It seeks to subtract complicated design methods from problem solving and leverage the simplicity of what computer architecture is designed to do: input, transform, and output data.

Data-oriented design (DOD) grew when game developers needed to use modern hardware architectures for performant games, and existing software processes did not meet their needs. The DOD process reduces software to a basic goal of computer architecture: to input, transform, and output data.

To properly explain DOD, we first define DOD and compare it to similar processes. A history of how DOD evolved is introduced, and core concepts in the DOD process are further discussed through several relevant examples. The Unity Technologies Data-Oriented Tech Stack (DOTS) is brought up as a canonical use of DOD, and the conclusion mentions the use of DOD outside of game development as well as its future.

Digital Object Identifier 10.1109/MC.2022.3155108
Date of current version: 6 May 2022

When discussing software processes, it is important to consider that we are likely biased when solving problems using a computer. Current research suggests that we overlook subtractive changes in problem-solving in comparison with additive changes.[1] For example, when given a Lego block bridge that has a one-block difference between the left and right sides, most people under a time constraint will choose to add a block to one side rather than remove a block from the other side. It makes sense that this bias would also impact how we make decisions in developing software. "Feature creep" is a known potential issue, and most software is developed within time constraints. This bias can lead to bloated and slow software, incompatible with soft real-time systems, such as games, which are required to consistently run at 30, 60, or even 100 (such as for virtual reality applications) frames/s.

The emphasis on data as a design driver allows DOD to reduce unnecessary complexity and emphasizes that transforming data well means that one must understand characteristics of the data as well as the whole supply chain of development (for example, hardware and compilers) that implements data transformations. Historically, viewing data as core to the software development process is not a new concept. For example, data flow programming was conceived in the 1960s and concentrates on the flow of data through software algorithms, primarily for parallel computation.[2,3] The concept of data flow is related to DOD, but, in data flow, the emphasis is not truly on knowledge of data but on the flow of data from one algorithm to another.

DOD is an imperative design process due to its emphasis on program state changes; however, it is different from similar-sounding design processes, such as data-driven design, which is also related to data flow but allows the input data to control the state of the program, sometimes even at a computer architecture level.[4] In software design, it is commonly used in games to increase flexibility. As an example, the data for a game level may contain information about special effects and door state changes that are read in and executed by a data-driven game program.

DOD does not emphasize data control or data flow in a program, only that the program be defined in terms of data input, transformations, and output. The core that ties all of the patterns in DOD together is that programs only input, transform, and output data. All elements and patterns involved with DOD may be understood through this focus.

CURRENT RESEARCH SUGGESTS THAT WE OVERLOOK SUBTRACTIVE CHANGES IN PROBLEM-SOLVING IN COMPARISON WITH ADDITIVE CHANGES.

DOD asks detailed questions about the data and uses answers to design software. Examples include asking about

- type
- distribution
- count
- storage
- accuracy.

In modeling, knowledge of the data can change the way programs are created. As an example, it becomes possible to consider which program system to create next from how often that system is run and how much data it transforms. Within a game context, if a character spends most of his or her time walking around the game world, one could foresee that walking is very important and should be a high priority in development.

Deep knowledge of data and the problem being solved leads to concrete solutions but does not preclude flexibility in software. Flexibility is part of the design process and should be planned out and carefully thought through rather than just "thrown in" as part of a "generic" solution. Solving for a problem that needs a flexible solution is a different concrete problem than solving for nonflexible cases.

In the search to understand data transformations (especially when they work poorly), the whole supply chain for software development, including both hardware and tools, is considered. As an example, hardware is considered primarily because it provides the physical means to transform data. A DOD proponent does not seek to know all hardware specifics but to specifically understand how the details of the hardware impact the constraints that the software needs to meet. The most common hardware considerations in DOD are cache performance and multicore processing, for this reason. Both of these hardware elements greatly impact the performance of the input, transformation, and output of data.

The core of DOD is not about optimization or making fast programs through hardware consideration; it is about organizing programs around a deep knowledge of data and its transformation. A slow program can be based around modeling data and ignoring the supply chain view that DOD proponents use. If performance is not important for the application, then the knowledge of data can still be used to create software. However, it cannot be understated that engineering software well requires understanding transformations. Hardware knowledge is required primarily because models of software and compilers are abstractions, and those abstractions are leaky and inconsistent.[5]

THE HISTORY OF DOD

The movement toward data orientation in game development occurred after the PlayStation (PS) 3 game console was released in the mid-2000s. The game console used the Cell hardware architecture, which contains a PowerPC core and eight synergistic processing elements, of which game developers traditionally used six. This forced game developers to make the move from a single-threaded view of software development to a more parallel way of thinking about game execution to push the boundaries of performance for games on the platform. At the same time, large-scale (AAA) game development was growing in complexity, with an emphasis on more content and realistic graphics.

Within that environment, data parallelism and throughput were very important. As practices coalesced around techniques used in game development, the term *DOD* was created and first mentioned in an article in *Game Developer* magazine in 2009.[6]

In 2017, Unity Technologies, known best for the Unity Real-Time Development Platform used by many game developers, hired DOD proponents Mike Acton and Andreas Fredriksson from Insomniac Games to "democratize data-oriented programming" and to realize a philosophical vision with the tagline of "performance by default."[7] The result has been the introduction of a DOTS, which is a canonical use of DOD techniques.

To date, many blogs and talks have discussed DOD since the original article, but very little has been studied in academia regarding the process. Richard Fabian, a practitioner from industry, published a book on DOD in 2018, although it existed for several years in draft form online.[8]

In 2019, a master's thesis was published by Per-Morten Straume at the Norwegian University of Science and Technology that investigated DOD.[9] Straume interviewed several game industry proponents for DOD in the thesis and concluded that, while they differed in their characterizations of DOD, the core characteristics of DOD were to focus on solving specific problems rather than generic ones, the consideration of all kinds of data, making decisions based on data, and an emphasis on performance in a wide sense.

Both Fabian and Straume discuss DOD elements without fully tying those elements together to form a design practice that can be used to create software. The overarching theme that ties all of the elements together is that software exists to input, transform, and output data.

DOD

The light switch problem

In DOD, questions about both the problem and data used for the problem need to be asked before attempting a solution. The light switch problem addresses the action of turning on a light and exemplifies how to think of data representation. In DOD, questions about both the problem and data used for the problem need to be asked before attempting a solution. For example:

- Is the light meant to be variable in intensity or just on and off?
- What hardware is available to help with turning the light on/off?
- What data are necessary for turning the light on/off?

If the light is meant to be a simple light that can be turned on and off, then it really only requires a single bit of data to represent on (one) and off (zero). An abstract solution can easily obscure necessary data for the problem solution. Figure 1 contains an image of a smart-home device that can control multiple lights through verbal commands, but, for a user who just wants to turn a light on/off, it is not readily apparent how to control a single light unless one has already memorized the name of the light and the process for controlling it. It also does not work when the house has Internet problems. This represents an overcomplicated solution for the initial problem that requires extra data for what is really a single-bit operation.

This is similar to what happens when software is constructed that tries to abstract away from the core problem it tries to solve. A complicated list of many other problems may be solved, but the initial reason for the software to exist may no longer be apparent or easy to use. Another example of this in physical hardware would be the PS4 game console power button. A lot of work was obviously put into making sure that the

FIGURE 1. A smart-home device that is capable of turning multiple lights on/off as well as controlling blinds through verbal commands.

game controller can turn the PS4 on/off, but people not using the game controller may have to look up an image to see where the power on/off switch is located, as it is hidden under a decorative panel on the front of the PS4.

Given the complexity of most software problems that need much more than a single bit of data, it is important not to allow extra complexity into the solution, as that extra complexity commonly creates more complicated solutions that need to be fully tested/debugged. Additional levels of complexity also equal additional requirements for testing and validation.

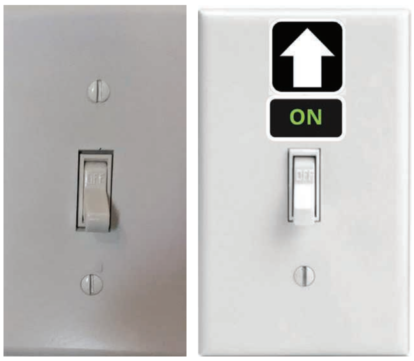

FIGURE 2. A switch (a) that minimizes the data necessary for turning on a light and (b) with slightly more data that allows a user to reason about which direction of the light switch is the ON position.

One bit that represents the on/off behavior of the light is necessary to turn on the light. Subtracting all of the extra data available yields an interface to the light bit that could look something like Figure 2(a). This particular solution is very similar to the solutions that DOD proponents seek in that it well represents the data and transformation of those data.

There are some extra considerations involved with that solution outside of the initial questions asked, though. For instance, the wall plate acts as a safety measure and covers the wires on the back side of the light switch so that people cannot accidentally touch them. It is normal for extra considerations to come up in the design and implementation of a solution, but each consideration should be carefully thought about before being added to the solution.

In Figure 2(a), the solution assumes the user knows that up is the on position and down is the off position, as the state change is not labeled. While this is true in some countries, in others, the opposite is true.

This hinders the usability of the switch, as the light switch on the left does not contain all necessary data for a user to know how to turn it on and off. A final solution to the light problem may look something like Figure 2(b), where ON is displayed, and the user is given all of the information necessary to turn on the light.

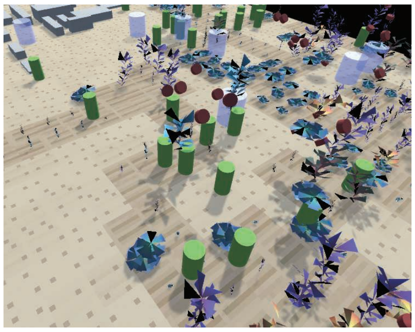

FIGURE 3. The Autofarmers simulation, where robotic farmers break rocks, till soil, plant procedurally generated crops, and take those crops to the market.

The Autofarmers simulation problem

The Autofarmers simulation is shown in Figure 3 and consists of farmers breaking rocks, tilling soil, planting crops, and selling those crops to a store for money. The full simulation is too complex for presentation, but it is useful to show how to model software from a data perspective.

How does one begin to view this in a DOD way, and how does using DOD alter the software development of the program? Ignoring the 3D models in the program (which each have their own data), one potential set of main simulation data includes

- position (x, y)
- speed
- direction
- target
- state
- scale.

These are the data necessary to perform the simulation part of the program. Rather than considering each “thing” in the game as its own object with separate activities, writing out data information allows operations across the game to be batched where possible and functionality used by multiple sets of data.

As an example, everything in the game has a position and can be placed at the same time. Only plants and farmers move. (Plants are carried to the store by farmers.) This allows for the same movement functionality to be used on farmers and plants. DOD looks at common data as well as operations and allows for the batching of those data with those operations.

In viewing the simulation in a DOD manner, planning would also look at which transformations are made the most often. As an example, farmers in the simulation spend most of their time walking to different places. Hence, a DOD-based solution would seek to understand the exact data necessary for the input to movement (for example, the position, speed, direction, and target) and construct the movement transformation early in writing the software for the simulation.

This would allow for experiments to be made that measure pathing/movement and potential solutions for any problems found. Using a poor solution for pathing/movement means that the simulation may run poorly due to the amount of time spent moving in the simulation, and this is a priority problem for software development due to the frequency of the data transformation in the program.

Using minimal data for a concrete solution. Most data used in the implementation of Autofarmers are 2D since the simulation is 2D, and the height is a constant. Developing a 2D simulation is a different problem than creating a 3D one, in terms of both data and algorithms. This is not a case where three dimensions should be solved for "just in case" since the algorithms and data are different for each one. On average, 3D computations are more expensive than 2D ones, as well.

DOD SOFTWARE DESIGN

A key element of design is data, so one designs data before code and views the data early and often. The hardware and compiler information are used along with any other tools that impact data transformations to understand the transformation and how to best make that transformation.

An image transformation problem

Say that the problem is to transform image data from color to gray scale. For a DOD solution, one must first start with the data, which should be designed before the code. For this problem, the input data are in bytes that range from 0 to 255. Each individual byte is in a quadruplet that represents the red, green, blue, and alpha values in the color image. The image being transformed will be in an array of 1,024 × 1,024, meaning that there are 1,048,576 pixels (each with a red–green–blue–alpha quadruplet) to be transformed.

One possible transformation from color to gray scale mathematically is the byte average of the red, green, and blue color information. The average for each pixel's color information will replace each pixel's red, green, and blue information, while the value of alpha will remain the same. The output is an array of size 1,024 × 1,024.

The minimal design for this program consists of the input data (an array with all of the data elements), the gray scale transformation, and the output of the 1,024 × 1,024 transformed data. Here is a potential partial code transformation for the problem that will yield a solution:

```
for (int x = 0; x < width; x += 4){
   for (int y = 0; y < height; y++){
     byte avg =(byte)((image[y, x] +
        image[y, x + 1] +
        image[y, x + 2])/3);
     result[y, x] = avg;
     result[y, x + 1] = avg;
     result[y, x + 2] = avg;
     result[y, x + 3] = image[y, x + 3];
   }
}
```

The potential transformation takes the data, converts those data to gray scale through averaging the color information, and puts that data into an output array. Is this the best potential solution? It is unknown, as there is information about the problem and transformation that has not been stated!

DOD solutions require as much knowledge about the problem and transformations as possible. For example, is it required that the solution not overwrite the original image? If not, then two arrays are not necessary in the potential solution. What is the hardware, and what language is being used for the solution?

If the hardware is a modern PC architecture, then cache usage should be considered, along with multiprocessor capabilities. Both of these things are parts of hardware that can greatly impact the engineering of data transformations.

Organizing data for good cache usage. One of the main bottlenecks for performance on modern PCs is the bus and data transfer from storage to the processor. The concept of caching away data is fairly simple. If somebody is working late one night, and they think ahead of time that a snack might be good for an energy boost in the early hours of the morning, then they will grab the snack (assuming it doesn't need refrigeration—don't try this with ice cream) and put it near their desk. That way, they will not have to get up and go to the kitchen later. The food has been cached near the desk.

Computers also think ahead and precache data for the processor. That way, it can more quickly and easily move those data into and out of registers for processing. The time cost for an L1 cache (the closest cache to the processor) access is less than 1 ns, whereas random-access memory access off the chip is on the order of 100 ns. If the wrong data are brought into the cache because they are poorly organized in memory, then the penalty time to read the correct data is much larger than if the data were able to be prefetched for use. Additionally, the extra time cost for access to data farther away from the processor also turns into extra power usage, as moving data costs more in terms of power than processing those data.

Structurally, a common pattern for organizing data for good cache usage is to employ Structures of Arrays (SoA) to organize arrays of homogeneous data so that they can easily be read and used in programs. A common phrase in DOD says that "where there is one, there are many," as processing data in a batch can have multiple benefits, including good cache usage. This differs from some traditional approaches, where all data based around a single concept are organized in a class based around that concept, and instances of those classes are put into arrays and methods called on each individual instance. Figure 4 shows how the organization differs in SoA when compared to Arrays of Structures (AoS).

The image transformation problem and cache usage. Organizing data for good cache usage appears to have already been done for the potential solution in the image problem. However, in the C# language (the language chosen for the solution), data are in a row-major format, meaning that elements are laid out next to each other in rows, rather than column-major format (for example, Fortran), where columns are next to each other in memory. Taking this information into account means that, to properly lay out the image data for transformation, the two for loops in the piece of code should be swapped (image height should come first, with the inner loop on image width) for better cache utilization.

Better cache utilization is only needed for the solution because the requirement specifies that the data count is enough that cache utilization will make a difference in performance. It is possible that DOD can be used to accomplish goals other than performance, such as memory or power utilization considerations. For this application, if the image was a 16 × 16 image, then it would not matter which way the transformation was done because there are only 256 total pixels to transform, and they can fit within the cache on modern processors.

Multiprocessor usage. Multiprocessor performance has become more important as games have evolved. The DOD view of programs as data input, transformation, and output is helpful when designing for parallel processing. The need for synchronization is a large bottleneck to performance for games.

Since race conditions only happen when data can be changed or written to, knowledge of the read and write status for all data and when transformations happen in a program helps to avoid race conditions. The SoA format can be used to help batch jobs since parallelization needs a certain number of elements to process before parallel processing becomes advantageous.

The image transformation problem and multiprocessor usage. For the image transformation problem, using multiple processors likely will not help to solve the problem, as it is only a single small image that is being transformed, and the overhead for setting up parallel processing can be more than the transformation of a single image. In the case of a problem that required transforming a set of images, parallel processing could be very useful and save significant time in processing.

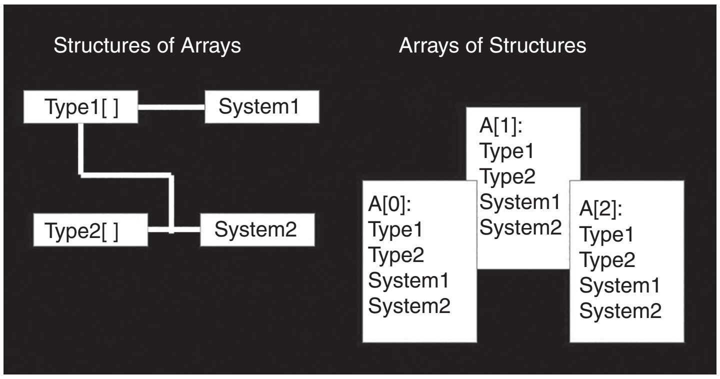

FIGURE 4. Structures of Arrays (SoA), where data are laid out in homogenous arrays fed into systems for data transformation, and Arrays of Structures (AoS), where data are generally laid out in instances of classes, and systems are called on single pieces of data.

View data early and often. Even though the first potential solution is not a good one when further information is seen regarding the problem and data, proposing the first solution allows us to reason about the data and improve on how it is input, transformed, and output. Many problems exist that do not have fully specified requirements. Examples in game development abound since it is a creative field, and game designs change frequently to "find the fun."

Viewing the data early and often aids in approximate solutions for incomplete data knowledge. As an example, DOD proponents commonly write small, experimental programs (or add debugging statements to existing programs) that seek to determine unknown knowledge, such as frequency or range. This knowledge can help refine a solution. In this way, progressive approximations can be made. While incomplete data can lead to solutions that must be iterated on, flexibility in an application is part of the design process and should be carefully considered separately from the experiments done for measuring data.

THE UNITY DOTS

Unity's DOTS is a large-scale industry example of DOD and consists of an experimental set of packages in Unity that were introduced after they were announced in 2017.[4] It is currently the most publicly available example of DOD in an industry product since the entity component system (ECS) source code is viewable online. The core components of DOTS are

- the burst compiler
- an ECS
- a job system
- testing and debugging support tools.

Burst compiler

The burst compiler is an excellent example of understanding and using the whole software development supply chain to create better solutions. It is an LLVM-based compiler technology that optimizes C# code for Unity's job system. It exists to allow for better overall game performance when using DOTS.

ECS

Unity's ECS implementation works closely with the job system. Entities are handles for a key chain of specific data components associated with the entity, components are struct data (in C#, a struct is a value type) for input/output, and systems are the data transformations necessary for solving problems.

If some of this information reminds people of databases, it is true that DOD can be compared to how data are organized in databases, and DOD-based frameworks commonly have the concept of a program-based query. An ECS allows for queries on components and commonly uses them as filters for jobs.

Job system

The job system exists to make data parallelism in C# easier. Jobs accept blittable (simple data types, such as float and int) structures, transform the data in those structures, and output results. The job system contains several different constructs that range from job-based parallel for loops that capture outside variables with a lambda expression to structures that have their own execute function.

Job inputs can be tagged as read only to allow for better knowledge regarding how the data are being input, transformed, and output. Jobs expect to obtain data laid out in an SoA manner, and options exist to determine various worker thread settings.

Testing and debugging support tools

One important part of viewing data early and often is to have support for adequately accessing the data. While profiler support of the job system in Unity is not one of the main selling features of DOTS, it supports DOD development efforts deeply. The profiler specifically profiles per frame and shows overall job system utilization as well as which jobs run at what time within the frame, and it shows this across worker threads.

An entity debugger allows runtime information to be displayed. Information about both components and systems is displayed, and data can be filtered to obtain information about a specific type of component. With these tools, it is easier to create small experiments to discover information about data during the runtime.

DOD concepts have been presented within the context of the game development field. DOD does not just exist within the game development field, and there are indications that it has relevance for other fields in software development. As an example, in 2014, CppCon: The C++ Conference invited DOD proponent Mike Acton to give a keynote speech to the larger C++ community,[10] and common DOD design patterns, such as using SoA rather than AoS, can aid any application that processes a lot of data programmatically.

In terms of philosophy, some of the concepts of DOD are at odds with object-oriented design (OOD), although it depends on which definition of OOD is used, and it is highly dependent on the problem being solved. A discussion of the many OOD definitions is outside the scope of this article, but, since DOD requires that the data be considered first and foremost for software development, simulated objects representing the problem space are unlikely to be used. The philosophy of DOD does not state that objects representing the problem space cannot be used if they happen to well represent the data for input, transformations, and output.

DOD promotes solving concrete problems as opposed to generic ones, which

ABOUT THE AUTHOR

JESSICA D. BAYLISS is a professor in the School for Interactive Games and Media, Golisano Computing College, Rochester Institute of Technology (RIT), Rochester, New York, 14623, USA, as well as a data-oriented design research and applications engineer at Unity Technologies. Her research interests include game development as well as co-creating the B.S./M.S. degrees in game design and development at RIT. Bayliss received a Ph.D. in computer science from the University of Rochester. She is a Member of IEEE, the Association for Computing Machinery, and the International Game Developers Association. Contact her at jessica.bayliss@unity3d.com or jdbics@rit.edu.

seems to indicate that it is against polymorphism and many inheritance uses in programs. Certainly, there are engineering reasons to reject deep hierarchies and virtual functions in favor of other ways of development that better align with hardware architectures on large game projects with performance requirements. The philosophy of DOD does not, in and of itself, reject all uses of these elements since there are many problems that need flexibility in their solutions, and the Unity DOTS example does make use of concepts such as interfaces to set out contractual obligations in program code.

DOD use has grown in the last decade, and, as it becomes more prevalent beyond its use in large game programs, it is important that DOD be recognized and studied in academia. DOD focuses on developing software based around data input, transformation, and output. The emphasis on design through considering data first can aid in clarifying the minimal set of data needed to solve a problem and allow reasoning about concepts, such as data parallelism for multicore solutions.

DOD has been used for large-scale projects in industry, with Unity's DOTS as a prime example meant to encourage the future development of performant games. Future work includes further research into the problems encountered with this approach for large-scale systems as well as the benefits for using DOD for such systems. C

ACKNOWLEDGMENTS

The author would like to thank Mike Acton for conversations about the history of DOD and encouraging the use of phrases that he has spoken many times in reference to DOD. The author is also thankful to the people at Unity Technologies for the chance to study a working example of core DOD and Sebastian Schöner for conversations regarding this article. Early conversations about this material, especially regarding the DOTS job system, are also based upon the effort to support more parallel and distributed computing education at all levels of the curriculum as supported by the National Science Foundation under grant NSF-1730527.

REFERENCES

1. G. S. Adams, B. A. Converse, A. H. Hales, and L. E. Klotz, "People systematically overlook subtractive changes," *Nature*, vol. 592, no. 7853, pp. 258–261, 2021, doi: 10.1038/s41586-021-03380-y.
2. D. A. Adams, "A computation model with data flow sequencing," Comput. Sci. Dep., Stanford Univ., Stanford, CA, USA, Tech. Rep. CS 117, 1968.
3. J. B. Dennis, "First version of a data flow procedure language," in *Proc. Programming Symp., Colloque sur la Programmation*, B. Robinet, Ed. Berlin, Germany: Springer-Verlag, 1974.
4. P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Comput. Surv.*, vol. 14, no. 1, pp. 93–143, 1982, doi: 10.1145/356869.356873.
5. J. Spolsky, "The law of leaky abstractions?" Joelonsoftware.com. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/ (Accessed: Mar. 23, 2022).
6. N. Llopis, "Data-oriented design (or why you might be shooting yourself in the foot with OOP)," *Game Developer Magazine*, vol. 16, no. 8, Sep. 2009.
7. Unity Technologies, San Francisco, CA, USA, *Unite Europe 2017 Keynote*. (2017). [Online Video]. Available: https://www.youtube.com/watch?v=-jZdMFPACpU
8. R. Fabian, *Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules*. Richard Fabian, 2018.
9. P. Straume, "Investigating data-oriented design," M.S. thesis, Norwegian Univ. Sci. Technol., Trondheim, Norway, 2019 [Online]. Available: https://hdl.handle.net/11250/2677763
10. M. Acton, *Data-Oriented Design and C++*. (2014). [Online Video]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc