COMPUTER SCIENCE & ENGINEERING DEPARTMENT
PES UNIVERSITY
100 FEET RING ROAD, BSK III STAGE, BANGALORE-560085

Data-Oriented Design Guidelines

May 1st 2021

Rishit Chaudhary
PES University
3rd Year, Department of CSE
PES1201800316
rishit.c.rc@gmail.com

Raghav Goyal
PES University
3rd Year, Department of CSE
PES1201800111
raghavb144@gmail.com

Tanishq Vyas
PES University
3rd Year, Department of CSE
PES1201800125
tanishqvyas069@gmail.com

1 ABSTRACT

This document illustrates and identifies a set of different rules/patterns of Data-Oriented Design philosophy in the field of computer science. These rules/patterns are meant to be employed in order to enhance the code's cache friendliness, by reducing the number of L1 & L2 cache misses, improving the CPU utilization and reducing the overall runtime for the program. The various sub sections illustrate each one of these actionable rules/patterns that developers could read, comprehend and integrate these practices into their software development lifecycle. Each sub section contains the intent, an applicable example in the form of code/use case scenario accompanied by detailed description of the pros and cons for the same based on the obtained results. The results prove that proper usage of these

rules/patterns while writing code helps improve the performance by a significant margin, statistics for which can be found in the section 5 Results & Discussions, in the latter half of this document. All the code and related documents can be found in our GitHub repository: ***https://github.com/tanishqvyas/Data-Oriented-Design-Guidelines***.

2 Introduction

The main principle behind Data-Oriented Design [1–4] or DoD for short, is to design architectures and write piece of code with a prevalent focus on the representation of the data itself considering the aspects of efficient memory layouts and enhanced memory access. The main intent is to identify memory access patterns and accompanying layouts. One could also refer to this as coding for the hardware. This intuition is correct as in Data Oriented Design we carefully select and deeply understand a finite set of hardware that we target. In Data Oriented Design we firmly believe that *"The hardware is the platform"*. This design philosophy is extensively used in the domain of game development in order to increase the performance of the system.

A very basic example of this principle could the choice of using a **vector** over a **list** for the purpose of making an iterable. This would then help us eliminate the overhead caused due to dereferencing the pointers while iteration as well as enhance the data locality thus increasing the cache performance. When we a run a simple code snippet for iteration over the container for 1000 elements and assigning them a value, the number of cache misses (read & write misses) for a vector are *2.5 million* and the execution time is *0.166s* whereas that for a list is *14.5 million* and *1.419s* respectively. We can clearly notice how much performance boost we could gain by keeping in mind the data layout and the kind of manipulation that needs to be done on the same. This is a heavily used design philosophy in game development industry as well as the game engines themselves. Although, the philosophy is flexible enough to be adopted in almost any kind of project in order to enhance the performance of the system.

3 Design Guidelines

This section illustrates a set of guidelines/patterns one must follow and keep in mind while developing a piece of code for certain niche situations. Inclusion of these principles into the development pipeline would help improve the overall performance of the code in terms of cache utilization, execution time and CPU utilization.

3.1 AoS vs SoA

One of the most far reaching concept in data oriented design is the comparison of Array of structures and Structure of arrays. The basic idea is that layout of data can improve cache locality and allow faster access of data and operations on it. As an example consider the following structure:

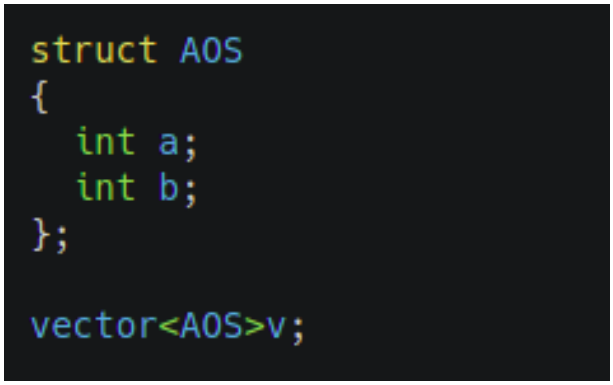


Figure 3.1: Array of Structures

For array, as shown in Fig : 3.1, the layout of structure components will be as follows

a b <pad> a b <pad> a b <pad> a b ...

where, *<pad>* is the padding that is present between each of the successive elements for memory alignment, which varies based on the CPU architecture and word-size.

On the contrary look at the following structure:

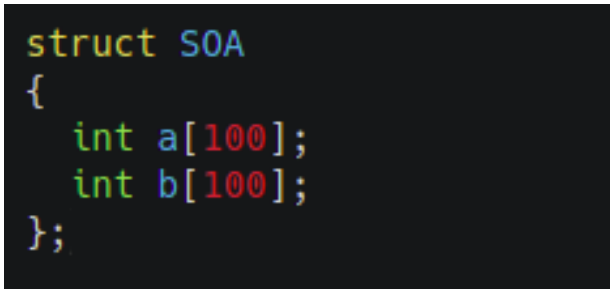


Figure 3.2: Structure of Arrays

Now we have a structure of array defined. The layout now is like

aaaaaaa... <pad> bbbbbb...

The direct advantage of this approach is lesser usage of padding in memory. This allows us to have twice the number of component “a”. Also the memory location of components “a” and “b” is continuous and allows faster accessing of data parallely. This is in contrast to the AOS technique, wherein accessing a single component of a structure is non-continuous in memory. In order to benchmark and validate the results we have created a `Player` class and a `Player` structure for $N = 10000$ players for AoS and SoA respectively. The code for the same can be found at our code repository. The time of execution for array of structures came to be *1.225s*

In contrast the time of execution for structure of arrays was *1.012s*.
The performance improvement can be even better understood by looking at the cache misses. For array of structures for the total references of *5,207,173,601* there were *100,070,993* misses. In contrast, by using structure of arrays, for the total references of *4,608,177,074* there were only *48,692* misses. This clearly depicts the huge leap in performance caused by using efficient data layouts.

3.2 Boolean in a Class, Bane or Boon ?

Consider a `Player Class` which has a couple of attributes amongst which we have a boolean `player_is_alive` and an integer `player_points`. Lets say we wish to update the points of all the *alive players* for each render cycle. Traditionally, one would have a vector or an array of player objects and would iterate over the players' list. Within the iteration a check will be performed using `player_is_alive`, if true, we do `++player_points`, otherwise we won't be performing the update.
Even though this method has linear time complexity, it still has a lot of scope for optimization. In this method we are wasting a lot of cache data. Suppose we have 64 byte cache line, we are reading in 64 bytes of information. However we are using just 1 bit of information, i.e. the boolean `player_is_alive`.
This is where Data-Oriented Design comes into picture. The suggested changes would be to completely remove the `player_is_alive` attribute from the class and instead maintain a separate array or vector of type boolean which contains the information about whether the players are alive or not. Iterate over this `status` vector/array and continue with the previously stated process. This new way of handling the booleans does not differ from the previously proposed solution in terms of space and time complexity. However, the results are still much better than the former method. This is mainly due to the fact that when we bring in the `status` vector into the cache, the data locality increases greatly thus resulting in lesser cache misses and better hardware utilization. This gives the overall performance a boost.
As an experiment we created a *Player* class which represents a player in the game consisting of various attributes such as health and name. The main attribute in consideration is the boolean variable *player_is_alive* which specifies whether the player is alive or not. As an example, if the player is alive we perform some processing, otherwise we do nothing. We iterate over a vector of size *1000000* and perform the processing on each of the players. In this OOP based implementation we get the time of processing as *0.825s* with D1 miss rate of *1.6%*. However, by using the DOD approach wherein the boolean variables are not member of the class but a separate contiguous array. We got a time of *0.794s* with D1 miss rate of *1.5%* which is an overall improvement over the OOP based approach.

3.3 Collection for States Pattern

Let us consider a simple example, from [5], where we have **N** players in a game scene. Each player has a health regeneration ability which helps them regain health over time. The general way one would go about doing this would be to iterate over the vector of players and check if the player has taken damage. If yes, then we would call the `regenerateHealth()` function

on that player object. However, as simple as it sounds, this approach isn't that good when it comes to performance. A better and data oriented way to do the same would be to maintain two separate vectors, namely, `allPlayers` and `damagedPlayers`. Upon dealing damage, a reference for that player is added to the `damagedPlayers`. This way on each render cycle, we could just call the `regenerateHealth()` function for all the players in the `damagedPlayers` parallely. The presence of the reference to that player in the `damagedPlayers` itself signifies that the function needs to be called for that player. Thus eliminating the need to perform an `if-else` check on the `Player` container while calling the function. Also, this enhances the data locality too as all the data that we need to operate on is stored in contiguous fashion. This small tweak helps boost the performance by a significant margin. Thus enhancing the performance of the game.
A point to be noted is that both of the above mentioned methods have time complexity of *O(N)* but the data oriented design philosophy results in better cache locality and removes conditional statements (*i.e. lesser branch prediction errors*) leading to better performance.
Having conditional constructs in the code such as having if-statements also leads to increase in the **Cyclomatic Complexity** [1] [6] of the code. This complexity deals with flow control of the code. General formula states that cyclomatic complexity equals the number of the conditions present added to one. This complexity signifies the ease of *debugging* and *testing* of code. However achieving low cyclomatic complexity requires having good form of data to start with and requires close inspection for null data. Lower cyclomatic complexity allows better estimation of the control-flow paths possible, work done and better optimizaton by compilers.

We ran a sample test program for 100000 Players with every 2000th player damaged. Upon using the iterative methodology the execution time turned out to be *0.074s* and D1 miss rate *0.5%* whereas in the case of the data oriented approach discussed above, it is *0.024s* and *0.2%* respectively. Thus we can see that there is a performance boost of roughly **3 times** in the execution time and 0.3% in D1 miss rate. The code for the sample test program can be found at our code repository.

3.4 Entity Component System

Entity Component System or ECS for short, is an architectural pattern that is built around the concept of Data-Oriented Design. Before one could proceed with understanding on how this relates to DoD, one must try to understand what ECS means. ECS comprises of the following three things:

- **Entity** [3] is a stateless and behaviour-less globally unique identifier that identifies an element available in the observable space.
- **Component** [3] is what defines the aspect of state pertaining to an individual entity or set of entities.

[1] Cyclomatic complexity refers to the number of linearly independent paths through a program's source code. The high the number, the more complex the code. Complex code is inefficient, defect prone, difficult to test, read and maintain.

- **System** [3] can be thought of as a global and independently running actors which are used and are accountable for operating on all the entities by means of attaching, detaching or altering it's component(s).

ECS is a heavily used architectural pattern in Game Engines and Games themselves. It can be also thought of as a parallel of an in memory relational database. Usage of this pattern results in much better cache line utilization as compared to the traditional OO based approach due to the spatial and temporal locality of the data we are trying to operate on.

Let us take a classical example [7] in the game development industry of managing Players, which in our case lets say `Birds`. While using the OO based approach, we tend to create an `Animal` parent class, which the `Bird` class inherits from, as shown in Fig 3.3. The `Animal` class wraps the attributes as well as the methods corresponding to the `Animal` object together. Now while iterating over each bird in order to modify their positions we need to bring in the whole Bird object in the cache line. This results in the wastage of the cache memory due to the fact that we bring in the data on which we do not wish to operate. This layout does not allow for an efficient packing of the *operation-related* data elements and thus results in a higher number of cache misses, as seen in Fig 3.5.

```
class Animal
{
private:
    // Data
    Vector3 Position;
    int EnergyPoints;
    bool bIsAlive;

public:
    // Behaviour
    Animal();
    void Eat(Animal food);
    void Move(float deltaTime);
};
```

Figure 3.3: OOP Class

On contrary, when using the DoD based ECS approach where we separate out the object as an entity, its attributes as components and the class methods as one global system; we could achieve a much more efficient memory layout, as seen in Fig 3.6, that increases the data locality. Since each component could be operated on by the system in an independent manner without bringing in the non essential data (*data related to the other components*), we get a considerable amount of boost in our performance.

The use of ECS pattern provides the developer with the flexibility of defining similar entities differently without breaking the architecture. This is not possible in the standard OOP based

Figure 3.4: Memory layout of child class Bird with parent class Animal

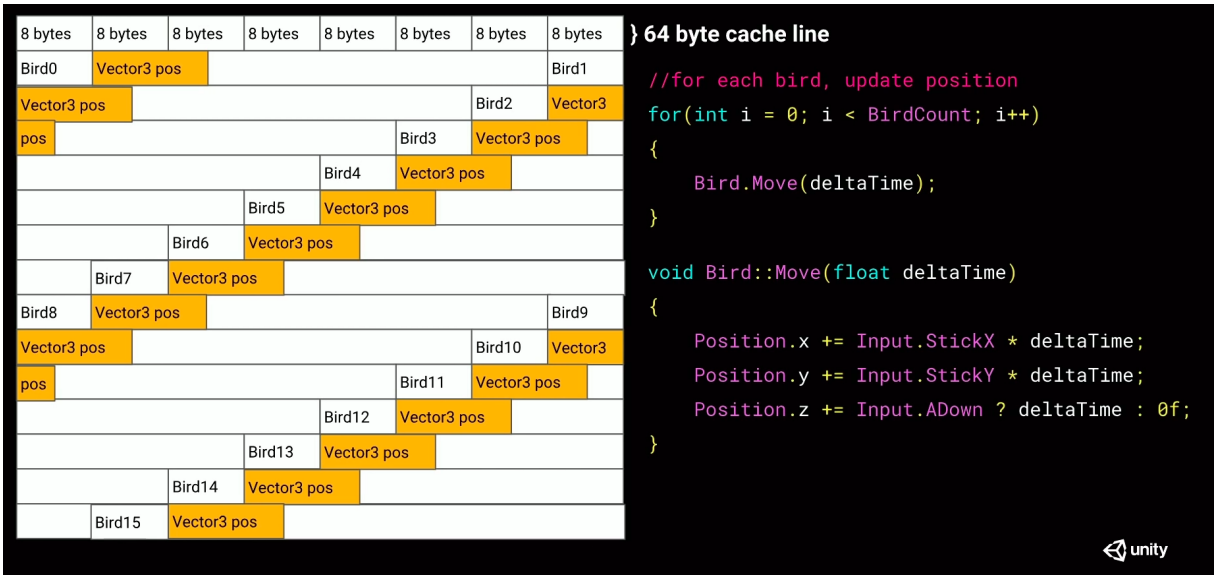


Figure 3.5: OOP Class Memory Layout

Figure 3.6: DOD Memory Layout

approach since we cannot have different attributes types for same object types. However the introduction of *Components*, allows us to do this. Thus we can add attributes to an entity only when we need it and can altogether stop worrying about setting things to default values as in the OOP based approach. [8]
Moreover, ECS based approach helps save memory since we have only one global system in contrast to having a copy of member functions of a class for every object we create.

4 Image Manipulation as an Example

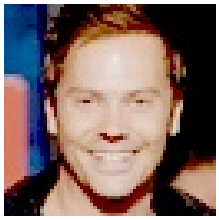
To apply the concepts of Data Oriented Design discussed above and to compare it with Object Oriented Design, we created a set of custom image manipulation filters. We applied the filters to the images of a format which we defined for simplicity. Our implementation consists of the filters: *Greyscale*, *Sepia channel Correction*, *Contrast*, *Negative* and *Gamma Correction*.
In the Data Oriented Design approach the images hold the file handle to the image file in the `original_images` and the `std::vector` of filters to apply on the images.
All the images and the corresponding filters to be applied are put into a `render_buffer` from where the system applies the filter one by one. The render buffer is a single object in the client code in which we use `std::array` to store the image pixel colour data and the list of filters to apply. This enables us to store the pixels and the filters to apply inside the object layout of the `render_buffer` object in contiguous memory locations. This gives better cache locality, as, when the `render_buffer` object is brought into the cache all the data needed for the process is also brought into memory with it. This makes sure that further references generated by the CPU during the processing of the filters will be to locations in the cache, i.e. either the L1 or higher-levels of the cache hierarchy.

The only disadvantage with the Data Oriented Design approach here is that the size of the `std::array` is fixed at compile time. This is because the object layout must be fixed at compile time. This means that either the developer should be aware of the size that needs to be set for the `render_buffer` at compile time or the program could be specifically compiled with a certain value for the `render_buffer` size during installation.
The programs are compiled during installation based on the user's system specifications. The second method is used quite often in games where certain parts of the program which contain such fixed sized buffers. The buffer sizes can trivially be set using compile time flags. This method would work as the `pre-processing stage` is the first stage of the compiler.
When the client creates the render buffer from the `std::vector` of image objects. The image objects read out their file contents, i.e. the individual pixel's r, g and b value and writes them to single buffer to increase the memory locality.
The image objects also write their filters to a buffer in the render buffer object. The filter objects contain a `std::span` [2] to the area in the pixel colour buffer that it needs to apply the filter on. This helps maintain memory locality.
In the Object Oriented Design approach every object simply stores the `std::vector` of its image's pixels *(i.e. Red, Green and Blue values)* and filters. The `std::vector` may not seem to be a problem as we believe that it is contiguous memory. However a `std::vector` is dynamic in nature.The memory of a `std::vector` is stored on the heap. The only part of the `std::vector` that we are concerned with as is within the object layout, is the pointer to the location of the `std::vector` in the heap. This means that accessing elements in the a single `std::vector` will be fast but moving back-n-forth between vectors is expensive. Surprisingly, even if the vectors are in the same object, the program will incur a heavy run-time cost. This is because in both cases the CPU will have to bring-in the data of this newly-referenced `std::vector` into the cache as well.This puts the Object Oriented approach at a disadvantage.
In the above discussion it is assumed that the CPU cache is not capable of recognizing complex but consistent memory access patterns and bring them in cache. On the system[3] we used while working on the project we did not find any such advanced pattern recognition for the cache accesses by the CPU[3].
The filters use the concept of inheritance and polymorphism *(i.e. virtual functions)* so that we can create a `std::vector` of the pointers to the filter objects and then later using the pointer to the parent `class filter` and call apply using that pointer to render then filter on the image. This incurs a cost as, now, for every filter object access, the CPU needs to first access the virtual table in memory and then go to the correct implementation of the `void apply(std::vector<pixel_colour_t> &)`. This requires multiple memory accesses to different parts of memory thus greatly reducing the cache locality.
The render buffer only contains the data it needs to apply the filter. As the filters are of different types we use the `C++17` feature of `std::variant` and `std::visit` to handle different types of filters in the render buffer. This is done to avoid using virtual functions and inheritance.
The render buffer is of a fixed size which is only created once the client has selected all the filters to apply to each and every image. This is needed because if the render buffer was

[2] std::span is a `C++20` feature
[3] `Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz`

maintained from the very beginning, then for every filter the user applies or removes for the images the buffer will have to manipulated which could mean large movements in memory which is expensive and unnecessary when the user is still exploring the filters to apply. Only once the user has confirmed their selections do we create the render buffer tailor made to the user's selections. To view the results we created an image viewer in Python, which shows the processed images.

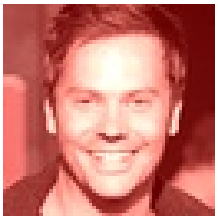
(a) Original Image



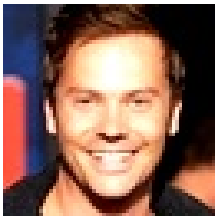
(b)

(c)

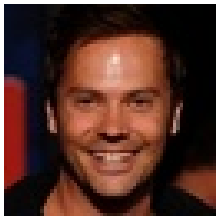
(d)



(e)

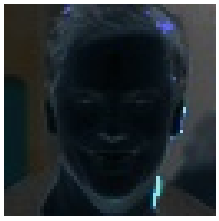


(f)

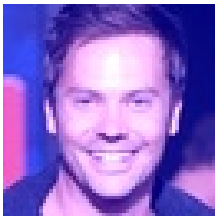


(g)

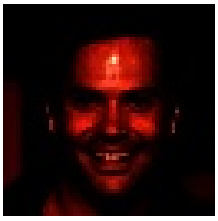
(h)



(i)



(j)



(k)

Figure 4.1: Image `(a)` is the original image used as input for applying the filters. The processed images `(b) - (k)` from the DoD and OOP Projects. Though the 10 processed images generated from both projects are identical, the performance and cache locality is vastly different.

As seen in Fig : 4.1, we have our input images (a) followed by the results of application of several filters on the same. The filters for the images have been chained together using the bean pattern in C++. This makes it easier to specify the filters on the images in the client code.

- Image Fig: 4.1 (b): *darkness channel adjustment*
- Image Fig: 4.1 (c): *grey scale, darkness channel adjustment, negative ,sepia*

- Image Fig: 4.1 (d): *grey scale, contrast, darkness channel adjustment, sepia, gamma correction*
- Image Fig: 4.1 (e): *negative, red channel adjustment, sepia, contrast*
- Image Fig: 4.1 (f): *sepia, gamma correction, darkness channel adjustment*
- Image Fig: 4.1 (g): *sepia, gamma correction, brightness channel adjustment*
- Image Fig: 4.1 (h): *contrast, freestyle channel adjustment, grey, sepia, darkness channel adjustment*
- Image Fig: 4.1 (i): *sepia, gamma correction, contrast, negative*
- Image Fig: 4.1 (j): *green channel adjustment, blue channel adjustment, gamma correction, contrast*
- Image Fig: 4.1 (k): *grey scale, contrast, darkness channel adjustment, gamma correction*

The OOP based approach for the image filters took a total execution time of *0.0199654 s* along with a L1 data cache miss rate of *0.1%*.
DoD in contrast executed significantly faster than OOP approach with execution time of *0.00945564 s* and L1 data cache miss rate of *0.0%*.
DoD shows significant less cache misses and faster time execution. This concretes the performance superiority of Data Oriented Design over Object Oriented Design.

5 Results and Discussion

This section contains the performance results and comparison for the design patterns/guidelines mentioned in the **Design Guidelines** section. All the benchmarking has been done using *Valgrind's* [9] tool called `cachegrind` and the Linux `time` command.

The usage of boolean variable within a class in scenarios, such as the one discussed in **section 3.1**, is considered as an *anti pattern* by many people in the C++ community, such as the current VP DOTS (Data-Oriented Technology Stack) Architecture and Runtime at Unity Technologies, Mike Acton [4]. Increasing utilisation of cache data can be done by having a separate contiguous memory for boolean variables.

When talking about AoS vs SoA, the effect of using efficient data layout is significant. The number of cache misses are caused mainly due to lack of contiguous data storage. Structure of arrays is able to leverage this by packing *operation-related* data more strictly and using much lesser padding than Array of structures. Data locality for same component is increased and allows faster access of data, parallelly among the components.

The philosophy of collection for states pattern mainly focuses on separation of data to be processed on from the overall data. This results in elimination of `if statements` altogether

since the presence of the data in the different container itself accounts for the condition being checked. Now that we have all the data we need to operate upon placed in a contiguous layout, this increases data locality thus being able to leverage the cache memory, better. Although a noteworthy point would be that as the number of damaged players increases and reaches close the the total number of players, the performance of the two methods converges.

We have shown how entity component system uses data-oriented design principles and allows using those principles to optimise the performance. It is wildly different from the traditional OOP approach and is quintessential for gaming industry. ECS inherently allows greater data packing and low coupling between entities & components. While operating on the data, cache data is not wasted as only the relevant data is read. ECS is primarily referred to as an architectural pattern but transcends to data oriented design.

We have conclusively shown with an example of image manipulation using OOP based approach and the DoD based approach that the latter of the two results in significant improvement in terms of cache utilization and the execution time.

6 Conclusion

Data Oriented Design principles have been adopted and are prevalent, primarily in the game development industry from quite some time. Due to their contrasting nature from the more traditional and popular Object Oriented Programming, they are still not widespread and accepted in all the domains. DoD is mainly used in applications that require processing under time constraints and to make the code better utilize the cache. DoD is vast and so we have tried to summarize some of the key points that make DoD an incredible design pattern. Applying DoD pattern must be done cautiously, as it is a use case dependent pattern and improper usage of the same may even result in performance degradation.

References

[1] Daniele Bartolini. Data oriented design resources. `https://github.com/dbartolini/data-oriented-design`.

[2] NOKIA Technology Center Wrocław. code::dive conference 2014 - scott meyers: Cpu caches and why you care. `https://www.youtube.com/watch?v=WDIkqP4JbkE`.

[3] Per-Morten Straume. Investigating data-oriented design. Master's thesis, Norwegian University of Science and Technology, Gjøvik, 2019. Available from: `https://hdl.handle.net/11250/2677763`.

[4] Mike Acton. Cppcon 2014: Mike acton data-oriented design and c++. `https://www.youtube.com/watch?v=rX0ItVEVjHc`.

[5] Richard Fabian. Data-oriented design. *Verkkojulkaisu. Saatavissa: https://www.dataorienteddesign.com/dodmain.pdf [viitattu 10.5. 2016]*, 2013.

[6] Perforce. Cyclomatic complexity explained | how to calculate cyclomatic complexity with helix qac. https://www.youtube.com/watch?v=rIz1-imryqk.

[7] Unity. Understanding data-oriented design for entity component systems - unity at gdc 2019. https://www.youtube.com/watch?v=0_Byw9UMn9g.

[8] Stoyan Nikolov. Oop is dead, long live data-oriented design. https://youtu.be/yy8jQgmhbAU.

[9] Valgrind Developers. Cachegrind: a cache and branch-prediction profiler. https://valgrind.org/docs/manual/cg-manual.html#cg-manual.overview, 2009.