Profiling a Prototype Game Engine Based on an Entity Component System in C++

Dennis Nilsson & Anton Björkman
Malmö University
2019-06-23

Faculty of Science Department of Computer Science
Degree Project in Game Development, 15 credits
Supervisor: Carl Johan Gribel
Examiner: Olle Lindberg
2019-06-23

We would like to thank our supervisor Carl Johan Gribel and examinator Olle Lindeberg for supporting us while writing this thesis at Malmö University.

***Abstract*—This paper introduces a performance comparison between an object-oriented artifact and a data-oriented artifact in the context of using an entity component system.**

A game engine must handle a variety of game objects which may share common attributes such as a transformation, collision and input management. Instead of solving this via multiple inheritance, entity component system uses composition to decouple traits into data (components) and systems operating on them. Game objects (entities) then serve as containers for all components necessary for a particular purpose, such as a player character of a piece of platform. Composition is not necessarily data oriented however, as evident by recent development in e.g. the Unity engine [13]. This thesis will implement two artifacts. One data-oriented artifact using an entity component system. One object-oriented artifact using an entity component system. These two artifacts will be stress tested to explore the performance.

Keywords: Data-oriented design, Object-oriented design, Entity Component System

I. Introduction

Games are becoming larger for each generation inevitably resulting in more computations for the hardware to perform. One reason for the continuing enlargement of games is the aspiration of achieving higher realism than previous generations and or adding more content. Unity is currently component-based but is introducing a new component which is an entity component system. This is because component-based systems did not age well because it did not align data in memory possibly resulting in low cache coherence [13]. Programming using an object-oriented design without considering the data allocation can result in inefficient allocation of data in memory due to factors such as encapsulation of data, inheritance, polymorphism. Encapsulation of data refers to bundling of data and methods operating on that data in the same class. Logic and data bundled together in a class could lead to low cache coherence. A low cache-coherence can possibly require more fetches from memory from the central processing unit (CPU) affecting performance negatively. A data-driven design such as an entity component system has the property of aligning data locally which could lead to improved cache-coherence [10], [6].

Today's games require efficient computation of data to maintain stable frames per second (fps). Unity currently uses a component-based entity system. This simplifies game development since its intuitive to add components to objects. A consequence of this is, in order to be able to create or destroy objects in Unity, a global list must be modified containing the names of each object. This will require a mutex lock. Another consequence is an extension of the previous consequence. Every game object has a C# wrapper pointing to the C++ object. There is no control over where in memory it is allocated, meaning it could be anywhere [13]. This could potentially increase the amount of cache misses. A cache miss happens when data required for the CPU's next operation does not exist in the cache. The CPU then has to fetch new data from memory in order to continue its work. Data-oriented design such as an entity component system focuses on aligning data locally. This can possibly reduce the amount of cache misses. The purpose of this study is to compare the performance of a data-oriented artifact and an object-oriented artifact. This will be done by conducting benchmarks on two artifacts, a data-oriented and an object-oriented. The benchmarks will record the time it takes to render frames in milliseconds. By analyzing the results of the benchmarks, this thesis aims to contribute by presenting a performance comparison between a data-oriented or an object-oriented artifact. The relevance of this thesis is providing important insight for developers when performance is a prioritized quality attribute. Future work extending our thesis could be examining the number of CPU cache misses per benchmark. Furthermore benchmarks in 3D environments would be of great importance for contemporary developers within the game industry.

The structure of this paper is as follows: This section, the introduction. Section II takes up background and related work. Section III explains the method used conducting the experiment. Section IV presents results and data. Section V & VI includes discussion and conclusion. Abbreviations will be avoided to minimize confusion.

II. Background and related work

The complexity of games striving for a realistic experience graphics-wise are increasing for each generation of consoles. This derives partly from the increasing amount of polygons in striving for realistic graphics [12]. It also derives from the fact that the more data a game scene contains, the more computation power is required to maintain for example 60 frames per second, as opposed to a game scene containing less data. Stable frames per second relies on computations being finished within the time limit of the game loop [15]. As an example, to be able to maintain $\sim$60 fps, $\sim$16 milliseconds (ms) is the game loops time limit. Not being able to achieve stable frames per second is considered as game-breaking in the game industry.

Even though computation power has been increasing since the arrival of computers, there are complications such as the difference in clock speed between the CPU and memory [15]. Factors such as increased power consumption, heat and thermal losses is contributing in the challenge of developing faster CPU's [9]. While CPU's has mostly increased in clock speed, DRAM's been mostly increasing in size. This is a bottleneck in performance because the clock speeds in CPUs exceed the clock speeds in DRAM memory [2]. Possibly with the outcome that the CPU waits a couple of hundred cycles until the DRAM is able to provide the necessary data for processing [15]. Since cache storage exist in the CPU's for speeding up memory access, developers should take advantage of that. By utilizing as much of the hardware's capacity as possible. As Bob Nystrom mentions, "Modern CPUs have caches to speed up memory access. These can access memory adjacent to recently accessed memory much quicker. Take advantage of that to improve performance by increasing data locality - keeping data in contiguous memory in the order that you process it" [15].

Our definition of data-oriented design is the alignment of data in memory with the goal of increasing data locality. Furthermore it is the alignment of data in the order that it will be processed in run-time. A data-oriented design tends

to promote data locality. It is important to align data so that it can be processed after each other in the cache line. By following a data-oriented design developers could increase the cache coherence. An object-oriented design unfortunately tend to spread data randomly across the memory due to storing objects on the heap, unless a memory pool is implemented and used. A positive effect of an object-oriented approach is the readability. For instance a human would be described in one class, as one object with all the properties a human holds.

This thesis will address the performance of a data-oriented compared to an object-oriented artifact in a game-like environment when using an entity component system. The main contribution of the benchmarks is displaying the performance of, in render time in milliseconds per frame, a data-oriented artifact and an object-oriented artifact using an entity component system.

RQ: What is the performance of a data-oriented artifact compared to an object-oriented artifact simulating a game-like environment using an entity component system?

Hypothesis: The data-oriented artifact renders frames faster than the object-oriented artifact depending on better cache performance.

A. Object oriented

Object-oriented design bring several attributes such as readability, inheritance, polymorphism, encapsulation of data and reusability. The readability attribute can simplify the communication within a project. Object-oriented design strives after describing things as humans perceive it. It makes it easier for team members to familiarize with a project. A possible negative consequence of object-oriented design is the potential dynamic allocation of objects in run time. This can be counteracted by allocating memory for objects before run time. If there is not already allocated memory for an object, it must be instantiated leading to the allocation of memory on the heap of that object. Dynamically allocated objects not making use of a memory pool might lead to low cache coherence because the data is not sequential which in turn could lower performance [8]. Figure 1 shows an illustration of how unaligned data may look like in memory.

Figure 1: Visualization of unaligned data in memory.

B. Data oriented

Data-oriented design focuses on data and cache coherency with the aim of having fewer cache misses. The design pattern is especially important for systems running in real-time such as game engines [8]. Data is not fast or slow but the hardware operating on it is. Meaning where it resides in memory matters. It is comparable to a book where the text is the data and the reading speed is the processor. If a book has the same size and two readers, the one finishing first will be the one reading faster. But imagine if the pages were scrambled in one of them. The one finishing first would probably be the one with the coherent text, almost no matter the reading speed of the readers. Data-oriented design purpose is aligning data in memory so when code is running the things to process is next to each other in memory. Like reading a book with coherent text. Figure 2 shows an illustration of how aligned data may look like in memory.

Figure 2: Visualization of aligned data in memory.

C. Entity Component System

Entity Component System is an architectural pattern that in a sense is an inverted object-oriented design. Instead of the entities using its own implementations of logic, the entities are acted upon by components which in turn are controlled through systems [5]. Entity component systems are novel and there is not much research within the area. Unity is integrating entity component system as a component in their game engine. This is due to the ability of retaining the user-friendliness of the component-based system currently used meanwhile also gaining performance and parallelism [13]. An illustration of an entity component system is shown in Figure 3.

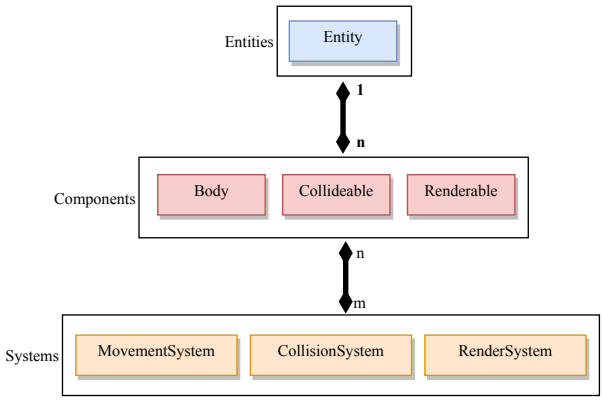

Figure 3: Visualization of an Entity Component System.

D. Related work

In Olof Wallentins paper [22], he analyzes a theoretical implementation of a component-based entity system. In order to do this, he made a comparison between several component-based entity systems. The comparison was made with an in-house game engine using different C++ component-based entity systems at the company he was working for. In his discussion he mentions that a data-oriented approach makes it harder for developers to understand and or lookup the associations within the system.

In Kim Svenssons and Tord Eliassons paper [11], they compare the functional and object-oriented paradigms in terms of memory usage and execution time. To compare these, they use Javascript and four algorithms binary search tree, shellsort, tower of hanoi and Dijkstra's algorithm. They state in their discussion that functional programming and object-oriented programming has different purposes but overall the functional programming performed worse than object-oriented programming. Their methodology is similar to ours as they conduct experiments and measure the different results in milliseconds. Their experiments involve different input data ranging from 1000 to 10 000 values.

In Walid Faryabis paper [8], he compares a data-oriented design to an object-oriented design by implementing his own entity component system. To compare these, he made performance tests in Unity with C# which consisted of simulating a sine wave and game objects with and without animations. Each test consisted of a set amount entities and measured the frames per second. The data collected was then calculated to a mean. His results shows that data-oriented design provides better average frames per seconds while using his entity component system. Our thesis will not provide results in form of frames per second. The results presented in this thesis will be time taken to render a frame in milliseconds. The chosen metric for this thesis is motivated by the difference between 90-100 fps not being equivalent to the difference between 20-30 fps. The difference is easily understood if one divides $1 \div 90$ and $1 \div 20$. Our thesis will be using EntityX [20] as the entity component system and implement a simple game-like artifact to conduct the experiments. Walid's comparisons are done with 3D models and animations using OpenGL. This thesis will make use of SFML (Simple Fast Media Library) to render 2D graphics [17].

III. Method

A. Methodological background

The methodology used in this thesis is experiments [16]. The independent variable in the experiments are the number of entities. The dependant variable is the time it takes, in milliseconds, per frame to render. The data generation method is observation. Each benchmark will be observed from which the data will be extracted. The data is analyzed quantitatively.

B. Research setting

All experiments were conducted in C++ on Windows 10 Home using the integrated development environment Visual Studio Community 2017 [14]. The computer's build-spec are Intel(R) Core(TM) i5-6300HQ, 16.0GB DDR4 RAM, 128GB SSD, Intel(R) HD Graphics 530. The CPU has 4 cores and 4 threads. The application cpu-z displays that each core has 128KBytes (4x32KBytes) L1 data cache and 128KBytes (4x32KBytes) L1 instruction cache, 1024KBytes (4x256KBytes) L2 cache and 6MBytes (4x1.5MBytes) L3 cache. The L1 data and instruction cache are 8-way associative. The L2 cache is 4-way associative. The L3 cache is 12-way associative [4].

1) Entity Component System - EntityX: EntityX is a type-safe C++ entity component system that includes its own benchmarks and is still maintained by the author [20]. Furthermore a benchmark suite has been created using up to 2 million entities in the benchmarks [1]. Projects has been made using EntityX [3], [19], [21], [7] to create 2D and 3D games. Therefore EntityX is a suitable framework for this thesis.

2) Simple and Fast Multimedia Library - SFML: EntityX was chosen as a framework for benchmarking. Since the benchmarks are measuring time per rendered frame an application for rendering also had to be chosen. An example using SFML was included in EntityX. This example was decided to be used as a foundation for the benchmarks. The reasons that motivated the use of SFML is the following. SFML is a cross-platform and multi-language framework consisting of five modules: system, window, graphics, audio and network. SFML has an active community which continues to implement new features and maintaining old features. SFML has official bindings for the C, .Net and other languages [17]. Several games has been made using SFML or some parts of SFML, some of these can be found on Steam [18].

3) Integrated SFML with EntityX on Windows 10 Home: EntityX included an example.cc using SFML for rendering. In this example there is a random function that did not function properly. The random function worked properly when rewritten as:

```
float r (int a, float b = 0)
{
      return ((float) std::rand())/RAND_MAX * a + b;
}
```

The random function original:

```
float r(int a, float b = 0)
{
        return static_cast<float>(std::rand() %
        (a * 1000) + b * 1000) / 1000.0;
}
```

C. Research approach

The chosen research approach is experiment. Hypothesis: The data-oriented artifact renders frames faster than the object-oriented artifact depending on better cache performance. The experiment is conducted in the following way. The process of the experiment is observing benchmarks while measuring and recording the metric render time per frame in milliseconds. 20 benchmarks will be conducted, each with different amounts of entities, explained in section III-C1. The benchmark results are compared. The prediction is that data-oriented artifact will have higher performance. The artifact with the lowest time per rendered frame is the one with the highest performance. The experiment is intended to show the performance of our artifacts.

1) The benchmarks: The benchmarks are implemented equivalently in regards of code. For the benchmarks, two artifacts were used. These artifacts implements the following two different paradigms, data-oriented and object-oriented design. The differences between the data-oriented artifact and the object-oriented artifact is shown in Figure 4. The data-oriented artifact uses three components Body, Collideable and Renderable. The object-oriented artifact only uses one component Circle consisting of Body, Collideable and Renderable. To understand the difference between the artifacts in memory allocation Figure 5 shows the memory layout for each one. In the data-oriented artifact the components are separated and placed one after another. The object oriented artifact is similar but there is only one component, Circle, that is aligned after each other. Each benchmark was executed for the same amount of time with different amount of entities. The amount of entities tested in the benchmarks range from 500 to 10000 entities in increments of 500 entities. During the benchmarks each frame's render time was logged. How many milliseconds it takes to render one frame has been

calculated to an average using the values retrieved from the benchmarks.

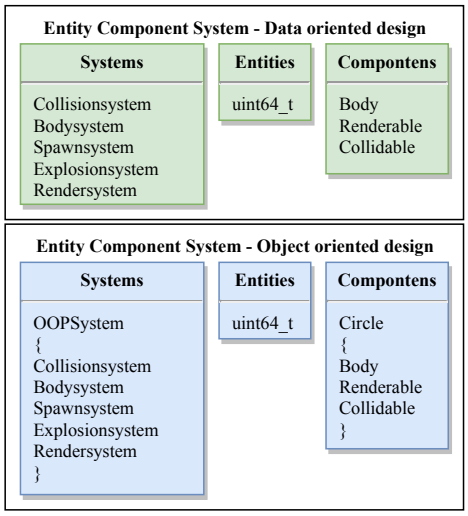

Figure 4: Describes the data-oriented and object-oriented artifact in terms of the Entities, Systems and Components.

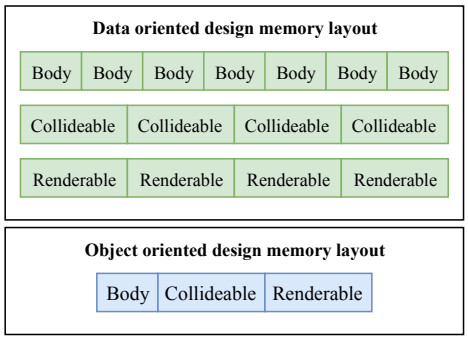

Figure 5: Describes the data-oriented and object-oriented memory layout.

2) Systems: Each system in our artifacts has a specific task. There are five systems CollisionSystem, BodySystem, SpawnSystem, ExplosionSystem and RenderSystem. The CollisionSystem handles collisions between the entities. The BodySystem handles the movement of an entity i.e. their position and direction. The SpawnSystem spawns entities into the environment. Each frame the SpawnSystem spawns as many entities as the total count of the initial value, which is set at the start of the application. The ExplosionSystem handles the removal of a colliding entities each frame i.e. making that entity inactive using a flag. The RenderSystem renders the entities.

3) Components: As shown in Figure 6: Body, Renderable, Collideable and Circle components used in our artifacts. The Body component has a size of 28 bytes. It consists of two vectors and three floats. The Collideable component consist of one float, having a size of 4 bytes. The Renderable component consist of a smart pointer to a shape, the size of this component is 8 bytes. The Circle component is built on the other components i.e. Body, Collideable and Renderable having a size of 40 bytes.

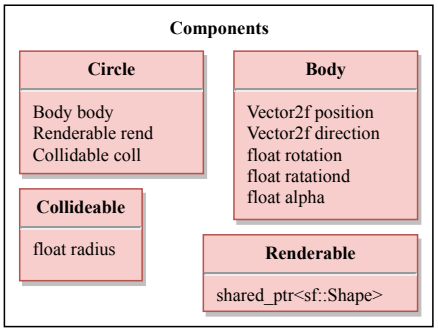

Figure 6: Describes the components used by our artifacts.

D. Data collection

Each benchmark records the rendered time per frame in milliseconds. The recorded time is calculated and stored as floats in a vector. After the benchmark finishes the floats are written to a text file. The benchmarks are executed for 10 seconds with different amount of entities. Therefore each benchmark will produce a different amount of frames. A benchmark with few entities will produce many frames while a benchmark with many entities will produce less. In other words, the sample size will differ for each benchmark.

1) Implementation of render time measurement: Before the game loop the current time is taken by using the clock from the C++ standard library. After the game loop the elapsed time is stored and the time taken between is calculated. The measured time is stored in a vector using the push back function in the C++ standard vector library.

2) Implementation of writing to file: A C++ standard library vector of floats is declared globally. At the end of the benchmark i.e after 10 000 milliseconds the data is streamed to a text file.

E. A quantitative data analysis

The benchmarks in this thesis are measuring how much time it takes for a frame to be rendered in milliseconds. An average value is calculated and presented through diagrams. The diagrams show a comparison of both the artifacts in render time per frame, render time per entity and performance gained of the data-oriented artifact in milliseconds. The amount of entities we decided to test is based upon reasonable amounts of objects in a game scene.

F. Limitations

The benchmarks will only measure the time taken per rendered frame in milliseconds. This metric is stored as a float. The range in object's sizes is between 4 bytes and 40 bytes. All benchmarks are executed in the same environment for the same amount of time. The artifacts used in the benchmarks are equal in implementation other than one being data-oriented and the other object-oriented. An important aspect regarding the benchmarks is that we are assuming that CPU prefetches are made sequentially. Since prefetches are CPU dependant we cannot guarantee the actual process of prefetchning made by the CPU in the benchmarks without taking our CPU in account. This is the reason for choosing to only measure the

time per frame rendered in milliseconds and conducting all experiments using the same computer. Benchmarks range from from 500 entities to 10 000 entities, by the increments of 500 entities each benchmark.

IV. Data

A. Artifact

Pseudo code for the artifact below, describing every frame in the game loop. Figure 12 shows an artifact with all the systems implemented running 1000 entities.

```
Implementation per frame in pseudo code of both
artifacts
        int c = 0;
        Count entities, store in variable c

        for i := 0 to amountOfEntites - c
            Create an EntityX entity
            Add to EntityX memory pool
            Assign components to entities

        for i := 0 to entities
            Handle movement

        for i := 0 to entities
            Handle bouncing on screen bounds

        for i := 0 to entities
            Check collisions
                if(collision)
                    EntityX inactivates entity

        for i := 0 to entities
            Render entities with SFML
```

B. Benchmarks

Figure 7, 8 and 9 presents the results provided by running the benchmarks using our research approach. Figure 10 presents detailed information regarding the benchmarks. The measurements give an error of less than 1.2% with a 95% confidence interval. Figure 11 presents information about the amount of frames collected for each benchmark.

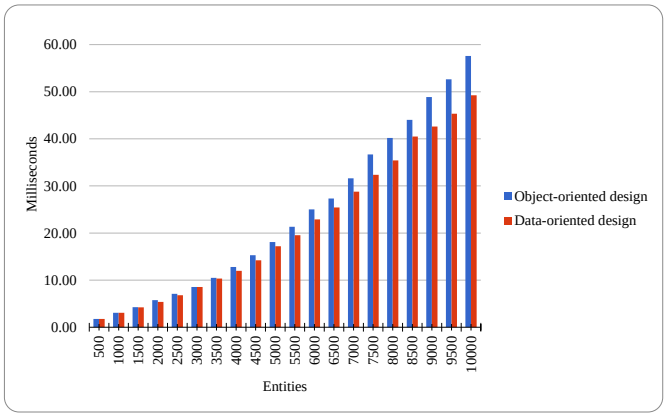

Figure 7: Benchmarks ranging from 500 entities to 10 000 entities. Average render timer per frame.

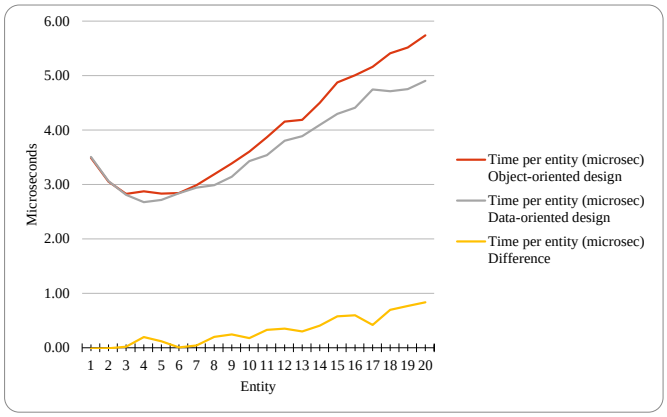

Figure 8: Average render time per entity per frame and difference in time per entity.

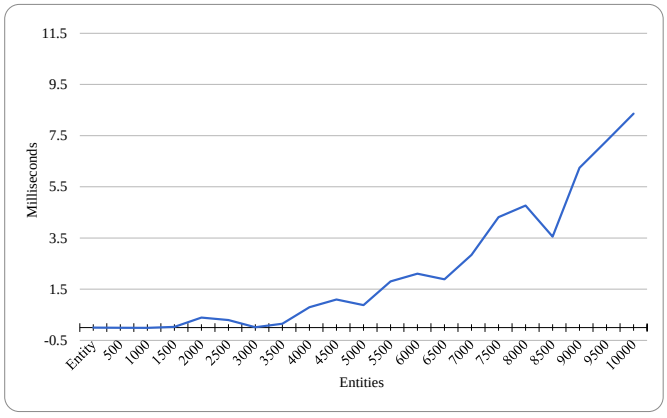

Figure 9: Performance gained per frame in milliseconds using the data-oriented artifact

Average render time per frame in milliseconds						
Entities	Object-oriented design	Confidence interval	Relative	Data-oriented design	Confidence interval	Relative
500	1,74	0,02	1,04%	1,75	0,02	0,91%
1000	3,05	0,02	0,58%	3,06	0,03	0,83%
1500	4,24	0,03	0,60%	4,21	0,04	0,86%
2000	5,74	0,04	0,66%	5,35	0,05	0,87%
2500	7,08	0,05	0,70%	6,79	0,06	0,83%
3000	8,53	0,05	0,60%	8,51	0,08	0,92%
3500	10,45	0,08	0,76%	10,30	0,10	0,94%
4000	12,74	0,09	0,72%	11,94	0,11	0,94%
4500	15,24	0,09	0,62%	14,14	0,14	0,96%
5000	18,02	0,15	0,83%	17,14	0,14	0,81%
5500	21,27	0,22	1,02%	19,46	0,18	0,92%
6000	24,92	0,29	1,15%	22,81	0,25	1,11%
6500	27,22	0,27	1,00%	25,27	0,26	1,03%
7000	31,49	0,27	0,87%	28,65	0,27	0,93%
7500	36,56	0,41	1,12%	32,22	0,31	0,95%
8000	40,03	0,43	1,07%	35,26	0,36	1,02%
8500	43,87	0,44	1,01%	40,32	0,47	1,17%
9000	48,67	0,53	1,10%	42,41	0,41	0,96%
9500	52,41	0,53	1,02%	45,13	0,43	0,96%
10000	57,38	0,57	0,99%	49,03	0,49	1,00%

Figure 10: Detailed information about Figure 7.

Frames collected		
Entities	Object-oriented design	Data-oriented design
500	5432	5541
1000	3158	3142
1500	2261	2290
2000	1670	1781
2500	1366	1417
3000	1130	1132
3500	929	931
4000	764	808
4500	634	684
5000	542	563
5500	457	494
6000	394	419
6500	355	383
7000	305	339
7500	265	298
8000	242	273
8500	221	240
9000	197	228
9500	185	214
10000	168	197

Figure 11: Collected frames for each benchmark.

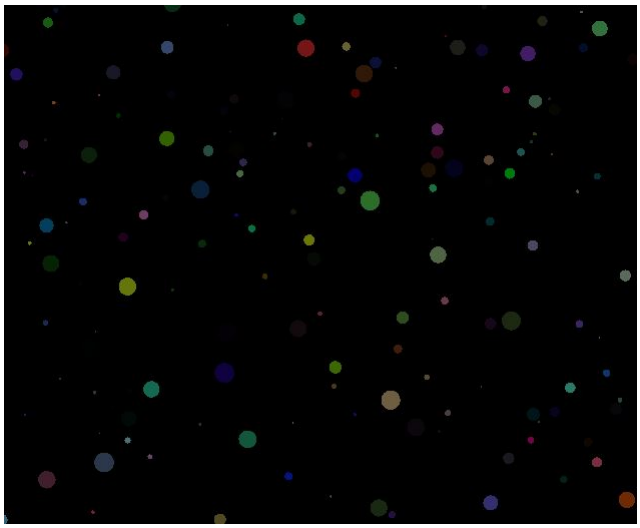

Figure 12: Screenshot of our artifact running 1000 entities.

V. Discussion

This section is about the performance of our data-oriented and object-oriented artifacts.

A. Prelude

What determines the reading and writing speed of data is the hardware connecting it and or storing it. Furthermore there is a difference in clock speed between the CPU and DRAM. The bottleneck introduced is that the CPU might have to wait hundreds of cycles for a single byte if the data does not reside in its own cache. Data locality could improve performance because it should minimize the amount of cache misses occurring in run time i.e. improving the cache coherence. Before writing this thesis, we suspected that data-oriented artifact would be faster than the object-oriented artifact. What we didn't know was if our data-oriented artifact was faster than our object-oriented artifact and if so how much faster.

A game running in $\sim$60 fps requires all computations to be finished within $\sim$16 ms. Suppose that a function takes 1ms and all functions in the systems are equally fast. This would mean that the budget is 16 functions. A function could be handling the movement of game objects among other tasks. These functions would have to be efficient to keep within the time limit. Failing to make them efficient will affect the fps. As shown in Figure 10, it takes about $\sim$15ms for the game loop to finish running with 4500 entities with the object-oriented artifact. That is about $\sim$1ms slower per rendered frame compared to the data-oriented implementation. This time could be spent better on other resources, such as controlling a population with artificial intelligence. The relation seen in the benchmarks did not correspond fully with our hypothesis. The data-oriented artifact renders frames faster than the object-oriented artifact when the amount of entities is over 3500.

B. Performance

By looking at the results of the benchmarks, the data-oriented artifact running over 3500 entities resulted in lower render times per frame than the object-oriented artifact. The object-oriented artifact running up to 3500 entities resulted in similar or lower render times per frame than the data-oriented artifact. A possible explanation for the performance gain of the object-oriented artifact when running an amount of entities under 3500 could be that relevant data resides in the cache. This is partly supported by looking at the result after multiplying 3000 entities with 40 bytes. This equals 120000 bytes which fits inside the level 1 cache. Therefore the performance of the two artifacts are very similar under 3500 entities. If this is the case, it means that the CPU does not have to fetch data from memory. But when the entities exceed 3500 in amount, it would seem that the required data for the next instruction does not longer fit in the CPU cache. By multiplying 3500 with 40 bytes which equals 140000, which is bigger than the level 1 cache storage. This possibly leads to a cache miss, meaning the CPU can't process its next instruction without fetching new data from memory. But in the case of the data-oriented artifact the data is possibly more efficiently aligned resulting in fewer cache misses when running a large amount of entities, in this case over 3500 entities. For clarification, the components, such as the body component, is stored in an array which in turn aligns the data efficiently. For example if data about the body component is required to perform an operation, the only array that needs to be traversed is the array containing body components. In the object-oriented artifact the components has been stripped down and put together into a single class to describe the object Circle. Therefore if in need of data regarding the body of Circle or any other property residing in Circle, the traversal of the array containing Circles is required to find the specific data and then use and or modify that data for an operation.

Figure 8 shows a quadratic curve. This is due to the collisions in the benchmarks having a quadratic time complexity. The collisions affected the benchmarks results but if the collisions would be removed, the artifacts wouldn't be as game-like. Since they affect, on average, the results in the same degree, the results are valid.

C. Summary

To summarize, the experiments show that there is a relation to data locality and performance. Our method of collecting data could possibly have been improved by collecting the same amount of data (frames) per benchmark. This would improve the confidence interval of the benchmarks. By conducting benchmarks on data-oriented and object-oriented artifacts using entity component system the research question could be answered. The contribution is showing that when running benchmarks with an amount of entities over 3500 our data-oriented artifact provided better performance than our object-oriented artifact when using EntityX.

VI. Conclusion

This paper set out to compare the performance of a data-oriented artifact and an object-oriented artifact. The benchmarks show that the data-oriented artifact is faster than the object-oriented running amounts of entities over 3500. Addressing this is an important contribution to contemporary game developers. This thesis has mostly focused on performance and the performance gained by data-oriented design artifacts implies that the findings are likely to be of importance to game developers. Furthermore it is important to developers having an interest of increasing performance or learning more about data locality.

In terms of future research we particularly suggest performing benchmarks while recording the number of CPU cache misses in an object-oriented designed artifact compared to a data-oriented. This is important to further present a possible performance gain by applying data-oriented design when developing performance-reliant applications such as games. The recording of the amount of CPU cache misses can be done within Visual Studio, an integrated development environment. Furthermore performing benchmarks in 3D environments, preferably in AAA video games [23], to be able to show the gain in performance of data-oriented design in large games. Furthermore about future work could be examining if bloating up the objects size to see if it breaks the cache performance.

In the following two sections, this thesis conclusions will be presented.

A. Results

1) Data-oriented artifact: Data-oriented artifact shown to be faster in benchmarks running over 3500 entities, presented in Figure 7, 8, 9, 10. This implies that its worth considering a data-oriented approach when developing games that are in need of performance.

2) Object-oriented artifact: The object-oriented artifact shown to be very similar in performance when running less than 3500 entities, presented in Figure 7, 8, 9, 10. This implies that the data fits in the CPU cache when running up to approximately 3500 entities in our artifact.

References

[1] Alex Beimler. Ecs benchmarks, 2014. Available at: https://github.com/abeimler/ecs_benchmark/. [Accessed: 2019-04-11].

[2] Carlos Carvalho. The gap between processor and memory speeds. Semantic Scholar, 2002. Universidade do Minho, Braga, Portugal.

[3] Giovanni Giuseppe Costa. Battlecity2014, 2014. Available at: https://github.com/ggc87/BattleCity2014/. [Accessed: 2019-04-16].

[4] CPUID. Cpu-z system information software. Available at: https://www.cpuid.com/softwares/cpu-z.html. [Accessed: 2019-06-12].

[5] Marc Erich Latoschik Dennis Wiebush. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. *In 8th on Software Engineering and Architectures for Realtime Interactive Systems*, 2015.

[6] Dice. Introduction to data-oriented design, 2014. Available at: https://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf. [Accessed: 2019-06-16].

[7] Giovani Milanez Espindola. Spacetd, 2015. Available at: https://github.com/giovani-milanez/SpaceTD/. [Accessed: 2019-04-16].

[8] Walid Faryabi. Data-oriented design approach for processor intensive games. Master's thesis, Norwegian University of Science and Technology, https://brage.bibsys.no/xmlui/, 2018. Available at: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2575669/18676_FULLTEXT.pdf?sequence=1. [Accessed: 2019-04-16].

[9] Alexander Fox. Why cpu clock speed isn't increasing, 2018. Available at: https://www.maketecheasier.com/why-cpu-clock-speed-isnt-increasing/. [Accessed: 2019-04-16].

[10] Gamesfromwithin. Data-oriented design (or why you might be shooting yourself in the foot with oop), 2009. Available at: http://gamesfromwithin.com/data-oriented-design. [Accessed: 2019-06-16].

[11] Tord Eliasson Kim Svensson Sand. A comparison of functional and object-oriented programming paradigms in javascript. DIVA, 6 2017. Blekinge Tekniska Högskola. Karlskrona, Sweden.

[12] Guy W. Lecky-Thompson. Video game design revealed. Cengage Learning, https://www.cengage.co.uk/, 2008. Page 117.

[13] Lucas Meijer. On dots: Entity component system, 2019. Available at: https://blogs.unity3d.com/2019/03/08/on-dots-entity-component-system/. [Accessed: 2019-04-16].

[14] Microsoft. Visual studio ide, 2019. Available at: https://visualstudio.microsoft.com/. [Accessed: 2019-04-10].

[15] Robert Nystrom. Game programming patterns, 2009. Available at: http://gameprogrammingpatterns.com/. [Accessed: 2019-04-16].

[16] Briony J Oates. *Researching Information Systems And Computing*. SAGE Publications, 2005.

[17] Simple and Fast Multimedia Library. Sfml. Available at: https://www.sfml-dev.org/. [Accessed: 2019-04-16].

[18] Simple and Fast Multimedia Library. Sfml projects, 2015. Available at: https://sfmlprojects.org/. [Accessed: 2019-04-16].

[19] Trans-Neptunian Studios. Triangulum, 2014. Available at: https://github.com/TransNeptunianStudios/Triangulum/. [Accessed: 2019-04-16].

[20] Alec Thomas. Entityx, 2014. Available at: Available at: https://github.com/alecthomas/entityx/ [Accessed: 2019-04-11].

[21] Will Usher. Asteroids, 2017. Available at: https://github.com/Twinklebear/asteroids/. [Accessed: 2019-04-16].

[22] Olof Wallentin. Component-based entity systems modular object construction and high performance gameplay. DIVA, 2014. Uppsala Universitet. Uppsala, Sweden.

[23] Wikipedia. Aaa (video game industry), 2019. Available at: https://en.wikipedia.org/wiki/AAA_(video_game_industry). [Accessed: 2019-04-19].