Tampereen yliopisto

Toni Härkönen

ADVANTAGES AND IMPLEMENTATION OF ENTITY-COMPONENT-SYSTEMS

Faculty of Information Technology and Communication Sciences

Bachelor of Science Thesis

April 2019

ABSTRACT

Toni Härkönen: Advantages and Implementation of Entity-Component-Systems
Bachelor of Science Thesis, 28 pages
Tampere University
April 2019
Bachelor's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Tiina Schafeitel-Tähtinen

Entity-Component-System (ECS) is a relatively new design pattern that has become a major actor in the field of modern *Realtime Interactive Systems*. Its objective is to decouple objects into separated data and logic. This is achieved by separating the objects into three different elements: *Entities*, *Components*, and *Systems*. This thesis examines the common problems in commonly used object-oriented methods of encoding entities, and introduces the reader to a way to bypass these issues by shifting to a more Data-Oriented programming pattern.

The thesis is for the most part implemented as a literature study by exploring different sources both available. Due to the nature of the subject, scientifically surveyed papers regarding the paradigm are limited. However, a variety of texts and presentations by professionals of the topic are freely available online, and the subject is also covered in multiple books focused on software engine design, especially game engines.

The aim for this study is to introduce the reader to an architecture suitable for processing massive amounts of data in a Realtime Interactive System, and present a conception of how the paradigm can be implemented. Object-oriented techniques often suffer from heavily encapsulated data and inefficient cache management. Both of these issues are extensively discussed in this thesis and solved with a change to Data-Oriented Design pattern.

Keywords: Data-Oriented Design, Entity-Component-System, Object-Oriented, implementation, Decoupling

TABLE OF CONTENTS

1. INTRODUCTION

Trying to implement complex applications using a traditional object-oriented programming architecture can often be a very arduous task. Massive class hierarchies, deriving from the same root class, complicated inheritance paths and addition of new entity types can make the codebase needlessly complex. At the same time, the flexibility and re-usability of the code decreases, and performance suffers. The encapsulation of data in object-oriented oriented techniques is also an issue which requires attention to attain good performance when working with a large number of entities.

To combat these issues, the traditional object-oriented design can be replaced with a more data-oriented design approach utilizing a composition-over-inheritance model where the data and logic are separated, called *Entity-Component-System* (ECS). In this approach, the properties of entities can be defined with small, reusable and generic *components* that do not contain any logic within themselves. Instead, the logic is handled by distinct *systems*, that are matched against the components, and continuously perform their internal methods on them in the background. [11]

Some of the more prevalent issues encountered using object-oriented techniques are first introduced in this thesis. Then the focus shifts to addressing the different elements of ECS, the advantages gained from using Data-Oriented Design techniques, and how this approach could be implemented on a conceptual level.

The aim of this thesis is to further introduce the reader into the concept of the Entity-Component-System architectural pattern in the context of data-oriented design. The chapter 2 will focus on the more traditional ways of encoding entities, as well as what problems these methods introduce. In chapter 3, the reader is intoduced to the structure of the pattern, as well as descriptions of the different elements the architectural pattern consists of. Chapter 4 focuses on the advantages gained with the usage of Data-Oriented Design over Object-Oriented Design. The more common implementations of data structures and multi-threading in the context of Entity-Component-Systems are covered in chapter 5. Chapter 6 explains common issues one may face when working with Entity-Component-Systems. All the examples provided are written in C++.

2. ENCODING ENTITIES – THE TRADITIONAL WAY

The term *"entity"* does not necessarily have a formal definition which applies for every use case. There are some properties which are often thought to be defining components of what makes an Entity. An entity should represent a singular, distinguishable concept. [16, 4] In object-oriented programming, an object can be thought of as an entity. This chapter focuses on the more common ways of encoding entities, and where these conventions may fall short.

2.1 Object-Oriented Programming

In Object-Oriented Programming implementation, each logical object is created as either an instance of a class, or a collection of interconnected class instances. A hierarchy of classes is classified by a system of criteria known as *taxonomy*. For example, a *biological taxonomy* looks for genetic similarities in eight levels: domain, kingdom, phylum, class, order, family, genus and species. At each level of the tree, different living things are separated to more specific and accurate groups by some criterion belonging to that level. [5]

A problem that arises with this classification, is that each level of the hierarchy can only classify an object according to one particular set of criteria. As such, if for instance an organism is classified according to its genetic traits, the color of the organism is not taken into account in any manner. To classify organisms by their color, a completely new tree structure would have to be created.

Similar problems which manifest themselves from the limitations of hierarchical classification are commonly seen in Object-Oriented Programming. A single structure of class hierarchy often tries to integrate multiple separate classification criteria into itself, or make room for new types of classes, which couldn't have been predicted when the class hierarchy was originally designed. Let's take a look at *figure 1* showing a structure of class hierarchy for example:

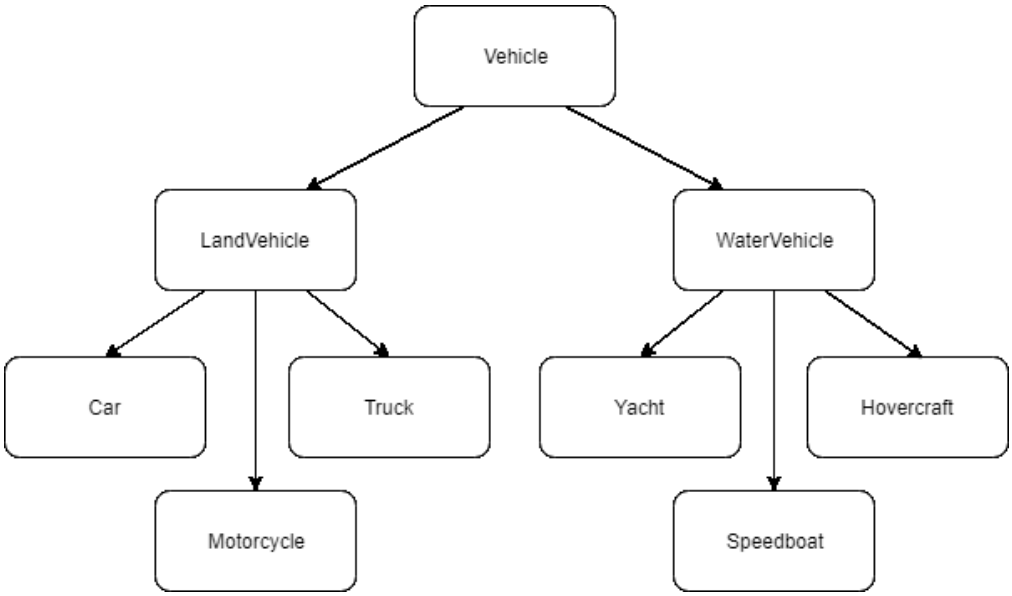

Figure 1. Motorized vehicle class hiearchy, based on [5]

While this hierarchy might look fine at first, what if the designers decide to add in an amphibious vehicle? The existing taxonomy does not accomodate for one in its current form, and trying to achieve the goal with a hack of some sort can easily lead into ugly, error-prone results. [5]

2.2 Multiple Inheritance and the Deadly Diamond

One solution to the problem of amphibious vehicles is a C++ practice called *multiple inheritance*, which in the example case means to simply inherit from both *LandVehicle* and *WaterVehicle* classes. While it may seem like an easy and good solution to the problem at first, issues regarding this practice will quickly arise. One of these issues is caused by multiple children inheriting from the same parent, and then trying to form a connected child class again, as can be seen in *figure 2*.

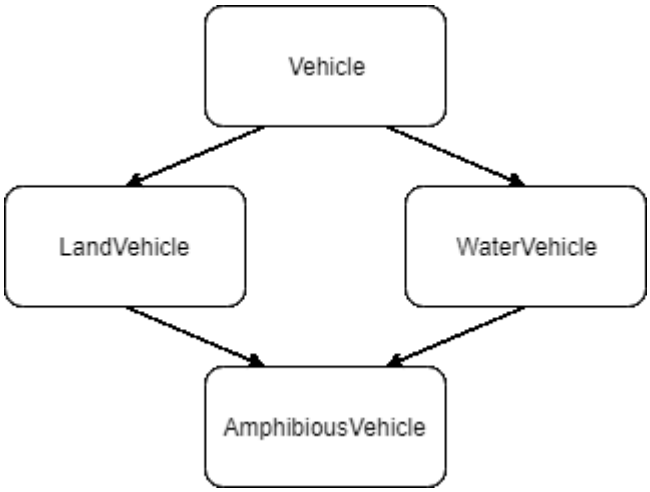

Figure 2. Deadly Diamond of Death, based on [5]

As *figure 2* shows, there is an ambiguity between *LandVehicle* and *WaterVehicle*, with both inheriting from the same base class *Vehicle* forming a so called *Deadly Diamond of Death*, leading to multiple copies of the base class' members. [5] This kind of hierarchy can very quickly become laborious to maintain and expand, as issues like which copy of the base class will be used have to be solved. To solve this problem, one will either need to entirely rearrange the hierarchy to get rid of the diamond, likely increasing code repetition, or use a language specific niche solution, such as C++'s *virtual inheritance.* [16]

2.3 Object-Oriented Composition

An intermediate approach is to factor the data and logic of an object into components separate from the entity. In this implementation, an entity is defined as a container for the components, which can be added and removed at run-time. By looking at *figure 1*, it is seen that the inheriting classes are simply adding new features to the already existing parent [4]. Separating these features into components is already a more flexible solution for the issue, as this approach requires no inheritance for the aggregation of an object, solving the earlier issue of Deadly Diamond of Death. A visualization of the hierarchy is provided in *figure 3*:

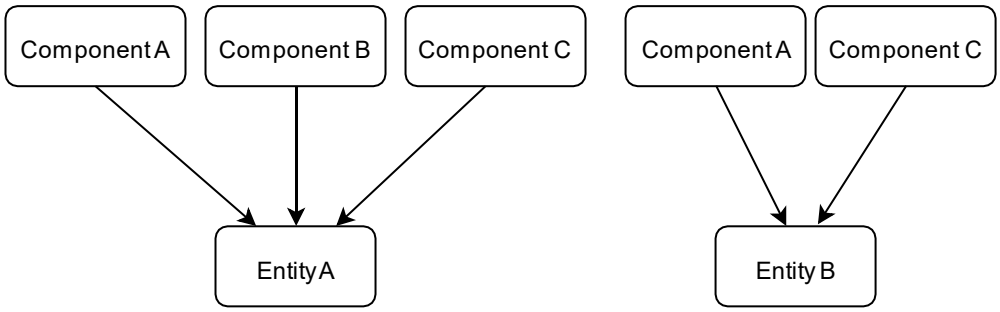

Figure 3. Hierarcy of object-oriented composition

In *figure 3,* it can be seen how the entities are composed of small components each providing the entity with certain functionality, getting rid of complex class hierarchies, as the components can be reused between different entities. While Object-Oriented Composition is already a more fitting solution for the purposes of complex large-scale programs, it is still somewhat lacking as it doesn't support the separation of data and logic, introduces significant overhead due to run-time polymorphism and it is not possible to optimize it for effective CPU caching. [16]

Since neither Object-Oriented Programming or Object-Oriented Composition seem to fit the needs of computationally heavy software, a third solution has been developed: a Data-Oriented architectural pattern called *Entity-Component-System* (ECS), which separates the data and logic from each other. With this approach, all data of the objects is decoupled to components, and can be processed as separate chunks between multiple entities at the same time instead of accessing a single objects data at once. This allows for highly optimizable memory management, parallelization, and flat, easy to manage class hierarchies. [14, 15]

3. ENTITY-COMPONENT-SYSTEM

Entity-Component-System is a software development architectural pattern which follows composition over inheritance principle, aiming to achieve a flexible and dynamic way to manage entities in a large scale real-time applications. By separating data from logic, ECS achieves a modular system, which allows for memory friendly storage of data in contiguous memory areas, and easily parallelized application logic. With modularity, entity systems can avoid pitfalls of Object-Oriented Programming (OOP), such as ambiguity in *multiple inheritance*.

It is good to conceive the distinction between entity systems and object-oriented programming from the beginning. While entity systems are often implemented with object-oriented languages, trying to encapsulate data and logic in same objects in ECS environment will ultimately end up in a failure. [10] Even if it is possible, and even encouraged, to have OOP and ECS used within same project, a clear separation between layers needs to be made, and any given layer should be implemented using either OOP or ECS [11].

The focus of this chapter is introducing the different elements of ECS implementation. Advantages of the paradigm are further discussed in chapter 4.

3.1 History

One of the first use cases of composition-based data-oriented design in a large-scale software was in a 1998 video game called *Thief: The Dark Project*, where the developers had a philosophy of creating a highly re-usable game-engine components. Dedicating to this experimental design was rewarded with an engine where ”...there was no code-based game object hierarchy of any kind.” The approach worked so well, that the team was able to use same executable file through much of the development for *Thief* and *System Shock*, two very different games, just by choosing a different object hierarchy and data set at run time. [18]

Another famous example for the ECS is a 2002 game called *Dungeon Siege*, which featured a seamless world without any loading screens, made possible by its ”Data-Driven Game Object System”, an engine with heavy resemblance to ECS despite the term not being officially coined yet. Dungeon Siege featured over 7300 unique object

types, as well as over 100 000 objects placed between two maps [1, 2]. A continuous world of this scale at the time was a remarkable achievement, but the modularity of the component-based system allowed for flexible memory management in run-time.

At later date, the pattern was officialized after Adam Martin, a British software developer, who created and spread his ideas and guidelines of the pattern among the industry, popularising it especially in the context of game engine development. As such, most of the Entity-Component-System paradigm is credited to Martin, even if other teams had already been experimenting and working with engines based on similar patterns.

3.2 Structure

The entity system implementation can be thought of an extension of the Object-Oriented Composition model, but implementing and using it requires a completely new way of thinking from the user. In this approach, the entity instances are reduced from being objects to mere numerical integer identifiers, only serving for the purpose of unifying components into a single being. The components in this paradigm are simple, small and logicless representations of an entity's properties and data.

The logic is handled by separate systems, which are always on the lookout for active components. They do not contain any data within themselves, and are only used for transforming the input data in desired manner.

The three elements are composed together with the help of a context manager, which lets the different parts of the pattern communicate with each other. The context manager is in charge of keeping a record of which components belong to which entities, and is used to pass the required properties to their respective systems.

3.3 Entities in Entity Systems

At the core of the paradigm are entitities, which can be thought of as the fundamental conceptual building blocks of the system. Each entity should represent a specific concept, but only on an ideological level, since they are actually not functional agents by themselves. In fact, an entity consists of nothing more than a unique identifier which distinguishes the coarsely specified entities from each other [17]. It does not contain any data or methods, nor is it an instance of a class. Ideally, an entity shouldn't be a list or an array of components either. The only function for the entities is to compose related components together to form them into a more concrete actor. [11] This is done by giving

the components an ability to be marked with the identifier to indicate which entities they are active in [10].

At the simplest, new entities can be generated with a simple static incrementation function, which generates a new entity id by adding 1 to the latest entity's id:

```
1       std::size_t generate_entity()
2       {
3               static std::size_t entityId = 0;
4               return entityId++;
5       }
```

The entity id does not necessarily have to be any more complex than a positive number, sometimes having some logic behind the identification may be beneficial in structuring the data. For example, as the id could for example be used as an index in a component array displaying which entities have that specific component active.

3.4 Components

Components are the data segment of ECS. They are small, generic and reusable types, which are used to define an entity's properties and how it can interact with other entities. They contain all the data belonging to their respective entities. As per Adam Martin's definition, components store data, but do not contain any logic. Each component represents a different aspect of an entity by providing it the data required to possess that particular aspect. [11] For instance, a skeletal soldier entity in a video game could possess components such as:

- Made of bone
- Undead
- Enemy
- Coordinates

None of these components contain any logic within themselves. The purpose they serve is to tag the entity for its properties, as well as to be separate containers for entities' data so that they can be kept as simple identifiers.

3.5 Systems

In order to provide logic for the components, systems, or *processors* are used. They are agents with a global scope, which means, that unlike a method of a traditional method of an object, they can be invoked from anywhere within the code. Instead of directly targetting a specific entity, they are interested in the active components of different entities. The components are added to a context, which is then passed to a respective system as an argument to provide the bounds of operation. Operating separately outside the entity-component structure, a system is matched against a set of components possessing the same aspects as that system, and performs its internal methods on the components continuously in the background, one at a time. Systems do not return values, but instead simply change the states of different components by performing data transformations.

An example system [16]:

```
1   void example_system(context& c)
2   {
3       c.for_entities_with<a_type, c_type>([&c](auto eid)
4       {
5           auto& a_data = c.get_component<a_type>(eid);
6           auto& c_data = c.get_component<c_type>(eid);
7
8           perform_action(a_data, c_data);
9       });
10  }
```

The system above loops through every entity possessing a certain set of components. On lines 5 and 6, the actual data belonging to the components of the entity are retrieved, and on line 8, the system performs a transformation on that data.

The basic cycle of a software running on ECS architecture is to iterate through all existing systems. Each system is matched against a subset of Entity/Component *Binary Large Objects* (BLOB). The BLOBs are selected from a global reserve, and the data is sent to CPU along with the logic code.

3.6 Composing The Model

Programming with an entity-component-system can be likened to programming relational databases. An instance of an entity acts similarly to a key in database, like in any

relational database management system (RBMS). From abstract point of view, accessing to a component of a specific entity is nothing but a database query. [11] The concept can be further intuited by looking at *Table 1,* where each row represents an instance of an entity and each column represents a different component:

Table 1. Entity-Component relationship table, based on [16]

	Component A	Component B	Component C
Entity #0	X		X
Entity #1	X	X	
Entity #2			X

It can be seen how different components are tied to entities, and how an instance of a component can be shared between multiple entities. [16, 18] A *context manager* is used to stay up to date with which entity has which components, and availability of a component should be able to be checked with a method like [16]:

```
bool context::has_component<...>(entity_id)
```

where context is equivalent to the context manager class. Once the availability of the component is confirmed, accessing its data is done with a method similar to [16]:

```
auto& context::get_component<...>(entity_id)
```

which returns the instance of the component for the specified entity.

A more concrete example of programming with an entity-component-system can be seen in a simple code example provided in *Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library* [16]. New component types are created as structs, holding the variables of the respective component. For each component, an array is created to store all the entities using that component.

```
struct component_a { /* ... */ };
struct component_b { /* ... */ };
struct component_c { /* ... */ };

constexpr auto max_entities{10000};

class component_storage
{
private:
	std::array<component_a, max_entities> _a;
	std::array<component_b, max_entities> _b;
	std::array<component_c, max_entities> _c;

public:
	template <typename TComponent>
	 TComponent& get(entity_id eid);
};
```

Context manager *c* is used to pass the required components to the processor, which can then perform its operations on the entities with required component types.

```
struct system_ac
{
	void process(context& c)
	{
		c.for_entities_with<a_type, c_type>([&c](auto eid)
			{
				/* ... */
			});
	}
};
```

The surface level use of the paradigm may look something like this:

```
 int main()
{
	component_storage cs;
	context c{cs};

	system_ac s_ac;

	// ...create entities...
	// ...add components...

	while(running)
	{
		s_ac.process(c);
	}
}
```

The component storage class is initialized as an object and set to a context. The context is being continuously passed to the system s_ac, which performs transformations on the data in every cycle.

4. DATA-ORIENTED DESIGN

When developing computationally heavy software, one frequently comes across performance issues causing the program's execution times for different tasks to stall for unacceptably long times. There can be multiple reasons to why that is happening, the most obvious one being unoptimized execution algorithms, but this is not always the case. It is fairly common that the slow-downs are actually caused by inefficient memory management. [6, 9] For one to be able to manage their data more effectively, a switch from object-oriented thinking to *Data-Oriented Design* (DOD), which ECS is a subset of, can be made. DOD aims to completely shift the focus from objects to handling the actual data. Because programming by definition is about data transformations, it is not actually necessary to think how an isolated object would do things. Instead, have the methods do the transformations in generic ways, and try to organize the data as effectively for the hardware as possible. [9]

In larger scale, having the data positioned optimally in the memory can have a drastic impact on a software's performance, so it is important to pay attention to the types of data, and how it will be processed. Ideally the data is laid out as homogenously and contiguously as possible so that it can be processed sequentially.
The ideal layout can be achieved by breaking objects into different components, and grouping those components together in the memory by their type. This results in homogenous and sequentially processable data. The approach works very well on large groups of objects, unlike OOP, which mainly focuses on a single object at time. For this reason, ECS is very effective and commonly used in game development: game objects are usually processed in groups. Processing data in sets lets one organize it so that the processing can be optimized to deal with groups of same types. DOD also enables the user to utilize multi-threaded processing as well a higher percentage of cache hits. Due to the modular nature of ECS, the paradigm allows for extremely flexible entity hierarchies and eases networking and serialization of data. [9]

4.1 Advantages of Data-Oriented Design

In Object-Oriented Programming, splitting a process between multiple cores can be a painful process due to synchronization errors caused by different threads trying to

concurrently access the same data. Having the threads wait for their turn to access the data results in a lot of idling, and the returns in terms of performance increase can be unsatisfactory. DOD simplifies parallelization. Because data is processed in groups, it is easy to split the data between different threads and the code will not be running into synchronization related problems.

Another strong point of Data-Oriented Design is the possibility for optimized cache utilization. In modern hardware, a key point to achieving great performance is to order the data in memory so that it can be efficiently used over and over again. If the data is laid contiguously in the memory, the data can be processed with near perfect cache usage resulting in superb performance. While optimizing the algorithms used to transform the data is certainly important, by looking back a bit further to how the data is being handled can yield even greater results in the large perspective. [9] Different methods for structuring data in memory efficiently are further discussed in chapter 5.

An already discussed advantage of using Data-oriented composition is its modularity. While it does not bring any extra performance to the program, it makes the development process significantly easier to manage. Keeping the functions small and avoiding dependecies between different parts of the code keeps the codebase from branching out, which improves the readability of the code, and makes updating and rewriting it significantly easier. While modularity is not only limited to Data-Oriented Design, it is a major factor to take into account, and thus worth mentioning. [9]

The last major advantage of DOD is how easy it makes it to test the code. Because all the functions are focused on directly transforming the data, unit tests simply have to take in some kind of input data, perform the transformation and see if the output is as expected. No need to worry about dependencies between different parts of the code. [9]

4.2 Application

The application of Data-Oriented Design and ECS to a piece of software can be thought of as an iterative process. A specific part of the object encapsulated implementation is chosen, and is implemented in a data-oriented way. This is repeated until most of the code works focused on the data. Once a subject to be implemented is picked, the inputs and outputs need to be determined. It is important to consider whether the input data is read-only, read-write, or write-only, as it helps to determine how the data should be

stored, and how different dependecies will be handled. Read-only data can safely be distributed between multiple threads, but some careful designing might be required when handling transformable data. [9]

Once the parameters have been thought through, the actual implementation needs to be planned. The systems need to be generic enough to handle hundreds of entries, so the functionality cannot be dependent on the input data being exactly the same for each execution. The transformation of the data requires thorough planning in regards to how it can be effectively cached and parallelized. [9]

5. PERFORMANT IMPLEMENTATION

A major advantage of entity-component-systems over object-oriented programming are how they can be optimized to fit a program's needs. The user is free to implement memory management by themselves, and decide whether they want to go for a simplistic implementation which may not be very cache effective, or if they want to go for a more complex, highly optimized system, which uses memory to the fullest, but may also suffer from fragmenting data tables.

5.1 Data Structures for Effective Memory Management

There are multiple ways to implement the data structures used for an entity system. They each come with their strengths and weaknesses. A common denominator between all of the approaches is trying to store data in memory as contiguously as possible. The reason for this is that contiguously stored memory in RAM loads quickly onto the bus, from where CPU pre-fetches it and the data remains in cache for the CPU to use it without additional delay. The condition is that CPU must read and write the data in increasing order. Due to this, for entity-component-system data structures, important things to prioritize are:

- Data is being stored in RAM as contiguously as possible.
- All entity system *processors* (systems) process their data in the order it is in *Random Access Memory* (RAM).
- The data structures are kept simple.

With these details in mind, there are multiple ways of implementing the actual data structures. [13] Some of these approaches are discussed in the following sub-chapters.

5.1.1 Approach 1: BigArray per ComponentType

One of the more obvious ways to implement a data structure for an entity system is to store the entity ids for each row of memory (an array of M items, where M = number of entities, each entity presents their state for that component) into arrays representing a component. A visualization of a component array can be seen in *figure 4*, where the array is populated by entity ids as and index, and their *position* data structs as the value.

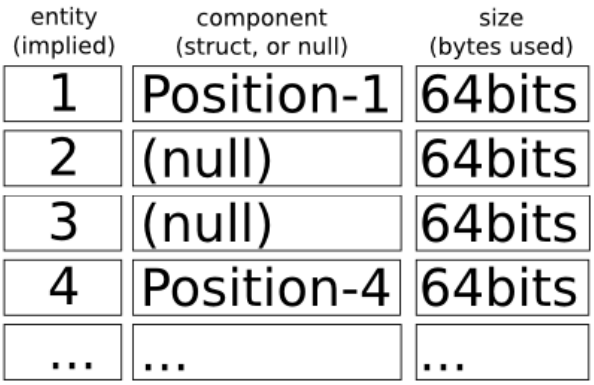

Figure 4. ”Position”-component array containing position data for each entity [13]

If entity ids are defined as integers, the array-indices can be used as the entity ids. The usage algorithm for this approach consists of the following phases:

- Pass the entire component array to a processor for iteration. If the processor requires access to *N* Components*, N* × *BigArrays* are passed.
- For random access, direct memory location jump can be made:
    - The memory location is: [Base address of the array] + [Component-size × EntityId]
    - The base address can be cached for the duration of the iteration, speeding up the process.

A problem that arises with this approach is that in a more complex software architecture, the method can be devastating for the cache. If the size of an instance is 500 bytes, and each BigArray consists of 20 000 instances, the size of one of the arrays totals into 10 MB. This goes way over what L1 and L2 caches can hold. Even modern L3 caches will struggle containing a single one of these BigArrays, when they should be able to hold multiple. [13] For the record, *Dungeon Siege,* an old AAA video game had over 100 000 entities at run-time, so even the 20 000 instances is on the lower end of the spectrum as far as large software goes [2].

Other problems that come with this approach include heavy memory usage and costly streaming. If the number of componentTypes for an entity is limited to 50, and the BigArray consists of 20 000 entities, the size for a single component type comes to *20 000* × *50 variables* × *8 bytes* = *8* MB. On modern machines this isn’t a huge problem anymore, but looking a few years back, it was common for mobile devices to be stocked

with around 512MB of RAM, which was something to consider when implementing the data structures. Having to stream large amounts of data from RAM to CPU can also introduce bottleneck even before the memory runs out. The transfer speed for DDR4 memory generally peaks at around 25 gigabytes per second, which is not nearly enough to be streaming arrays of 500 MB in every frame. [13]

5.1.2 Approach 2: Massive Interleaved Array

Another approach proposed by Adam Martin is interleaving the components for each entity. In practice, this means that all the components for a single entity reside adjacent to each other in memory. This allows for processors to have immediate access to the components they need, as they are always aware where they are being stored. A visualization of an interleaved array can be seen in *figure 5*, where an entity's components are all being stored directly under it.

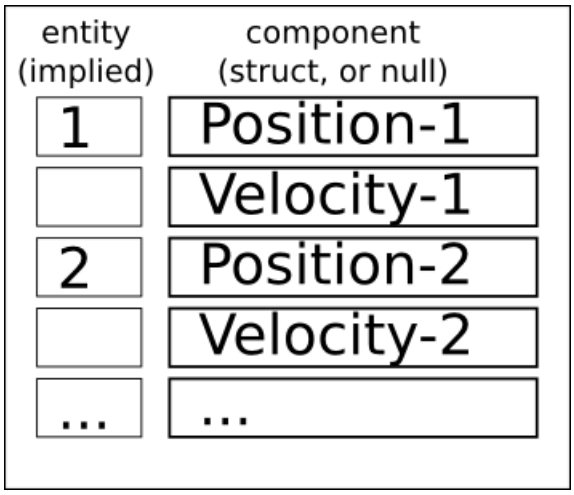

Figure 5. Interleaved component data [13]

A problem which arises with this approach is that once a certain set of components have been interleaved for a processor to access, they cannot be interleaved again, and if another processor needs to access some of these components, it will have to find the data it needs from the massive, semi-randomly scattered array. [13]

5.1.3 Approach 3: Separated Access and Component Arrays

The two previous approaches had some striking problems, which causes them to be unusable for software working with massive amounts of data. However, studying the

approaches is not fruitless, as the two can be refined and combined into a unified, more orgranized data structure, consisting of several smaller component templates, which contain interleaved component data for each entity. [13] *Figure 6* shows how this kind of table could be composed. On the left interleaved arrays not dissimilar to *Approach 2* can be seen. The array on the right consists of Component templates pointing to the interleaved arrays.

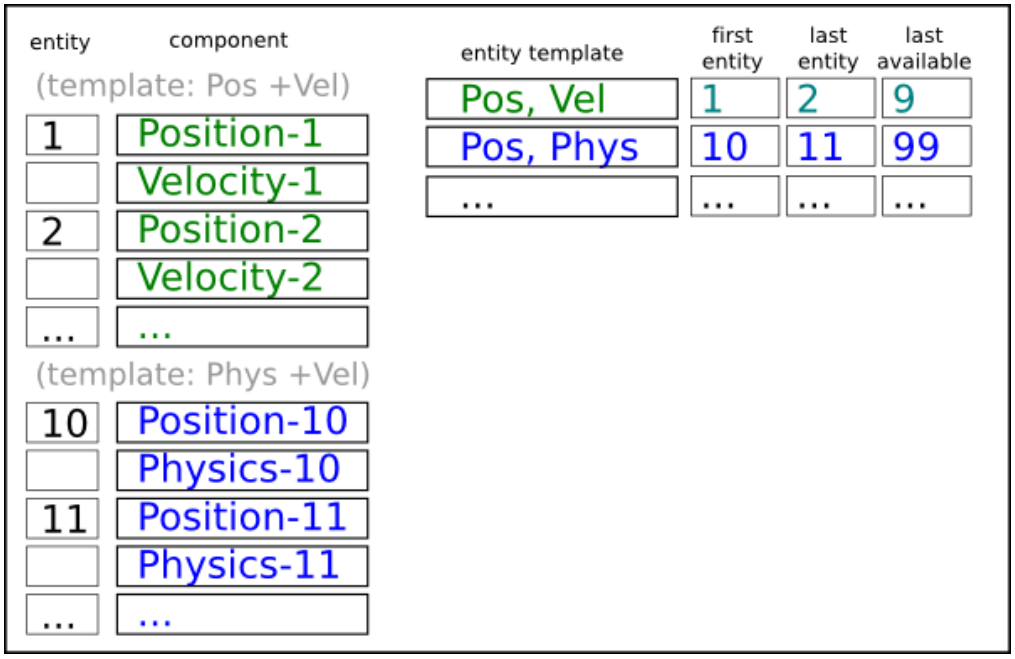

Figure 6. interleaved combination tables, MegaArray on the right [13]

To implement this approach, at the time of creation, the caller needs to give the entity some kind of information about the components it is likely to have in the future. Accordingly, the ECS will accommodate for entities with unique hint combinations by prereserving a new table everytime a new combination shows up. If a component table runs out of space, it can be extended by duplicating the table and attaching the new table to the end of the old one. These templates can be accessed from a *MegaArray* containing an entry for each combination, and under the combinations, the ids of the entities which contain the properties of the respective key. On every frame, a set of ranges within the MegaArray is sent for each of the processors to handle. The amount of overhead introduced by separating the component tables from the access data is negligible. [13]

While separating the components to combination tables does solve the flexibility problem caused by having a single massive array of interleaved component types, it is still not the perfect end-it-all solution. What if one wanted to introduce a massive amounts of

heterogenous entities into their software? Looking at *figure 6,* the tables accommodate for *Position* + *Velocity* combination as well as *Physics* + *Velocity* combination. What if one wanted to have an entity with the combination of all of these attributes, *Position* + *Physics* + *Velocity*? One solution would to have the combination templates be definitive, and simply create a new table for every newly introduced combination. However, this will quickly result into very fragmented tables, because every time a new component is either added or removed from an entity, it needs to be moved to a new table, causing a split in the mini-table range. Alternatively the tables could just be indicative of what components the entities should contain, but in this case additional information about the out of the norm combinations is required to not have them get mixed with entities that fully match their template. Another problem the templates come with is finding the components within their respective array. If an entity's position is known, it can be deduced that its specific components are some off-set away from that position, but unless all the components, as well as their adding order for the entity are known, it will be difficult to find a specific component. [13]

5.1.4 Approach 4: More Explicit Structure and More Tables

The fourth and final way to implementing a data structure to an entity system discussed in this thesis is focused on refinining the previously introduced approaches even further. This can be done by introducing two new types of tables to the system, one of which keeps track of where entities start, and the other states the offset of all the components from their respective entities. As a side effect, the tables also grants an index for which entities contain certain components. [13] A visualization of these tables can be seen in *Figure 7.*

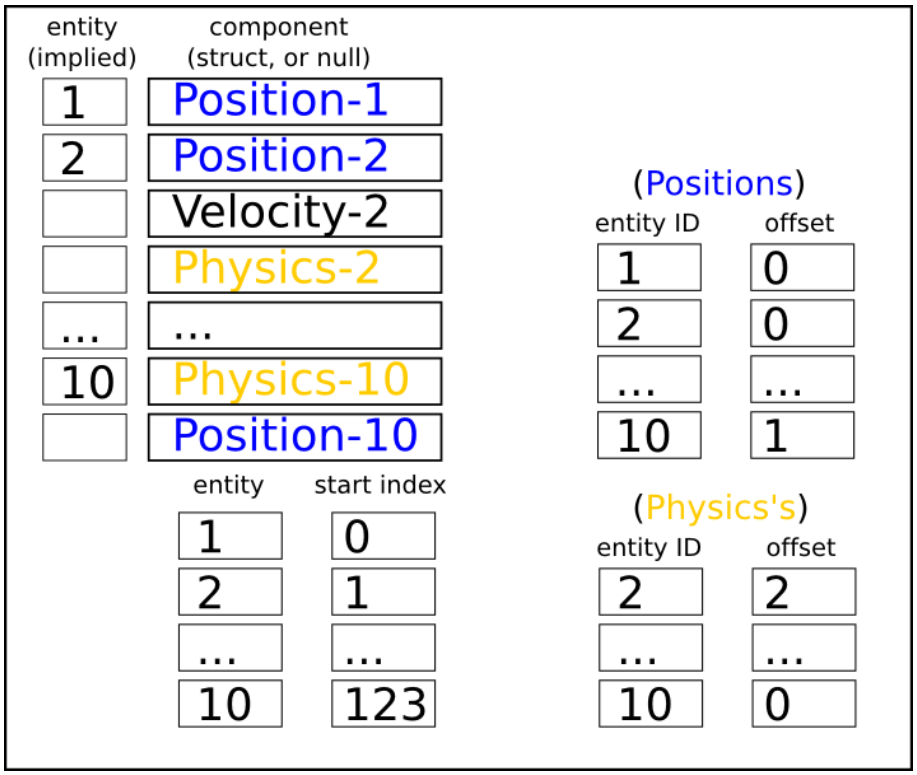

Figure 7. position and offset tables [13]

Having the data stuctured in this manner means losing some contiguity. To iterate over objects, a massive contiguous array holding the interleaved component data for each entity, as well as a number of non-contiguous side-arrays for the entity positions, as well as component offsets are now required. The saving grace here is that the component arrays are very small. Assuming an upper limit of 128 000 entities and a maximum 8 kb of component data for every entity. Now 17 bits are required for storing the entity ids, and 13 bits for the component offsets. With these limitations in mind, an entity's table size for all components will equal to 30 bits per component. This results in our side-arrays being around 1-40 kB in size, which small enough to be stored in cache multiple times, resulting in fast enough access times. [13]

5.2 Multi-Threading

Distributing a software to multiple threads has obvious performance benefits over having it run on a single thread. Concurrent operations drastically improve throughput times, responsiveness increases on both application and server side. Distributing only the required resources to the threads also lessens the overhead to creating, maintaining and managing them. [20]

To achieve a system that can be divided easily, as much of the data as possibe should aim to be read-only to avoid having to write multi-threaded code. Read-only data has the advantage of being accessible to multiple CPUs at the same time without having to worry about writing complex multi-threaded code, which can lead into difficult to debug mistakes taking extended periods of time to solve. A good rule of thumb to keep things simple is trying to write multi-threaded code in such a way, that there are no distinct rules to writing logic to the software. [14]

An opportunity multi-threading presents is running subsystems at different rates. For instance, video games usually aim to achieve rendering speed of over 60 frames every second. The same frequency is not necessarily required for other subsystems such as physics or artificial intelligence. These kinds of optimizations can result in significant performance gains, and are also helpful for flexible debugging. Since it is possible to slow down or stop certain systems while keeping others running, the data to debug can be adjusted to a more controllable amount. [14]

A reasonable way to implement such a system is a simple one. By duplicating all the required data to another place in the memory, there is no need to write multi-threaded code. The data can safely be managed, as it has no consequences to the actual data until the changes have been merged back to their original location. The process is quite similar to popular version control tools such as *Git*, *SVN* or *Mercurial*. To work with this kind of system, an Application Programming Interface (API) is in place to trivialize working with the data structures. The API should at least be able to [14]:

- Copy a set of data structures to represent all the data available.
- Copy specific components from specific entities at specific time.
- Merge data with others working on the same project by:
    - Updating the user's data by merging changes from *N* specific copies.
    - Update the data of *N* specific copies by merging the user's data with theirs.

The most simplistic way of implementing this is an event based one, where every change is stored to a list of events. While it is possible to do it this way, it is very inefficient due to having to do demanding processing to find out the simplest of changes in the software, and as such it should not be considered as a valid choice. A more sophisticated way would be to *copy the data on write.* This approach uses two sets of data, the *base* copy

and *changed* subset, to keep track of the data and how it has been modified. Some of the advantages of having the two different data structures are:

- Direct access to components in the *changed* subset. Fetching the current value of a variable in a specific component is fast.
- Merging data back is efficient by iterating the *changed* subset and overwriting the base values.
- It is easy to keep track on the number of changes.
- Memory usage is fairly efficient on average. The worst case scenario where all data is changed would result in twice the base memory usage, or three times in total.
- Deleting of entities and components in a thread leaves no dangling pointers or memory leaks. Merging multiple thread copies where some have deleted a component and others haven't requires some planning around though.

This approach has a drawback that arises when there are multiple copies with small changes to be dedicated to different threads. The RAM will be flooded with chunks of data very similar to each other, which is very inefficient and can lead to memory overflow. [14]

6. PROBLEMS

The entity-component-system paradigm is an effective way for encoding entities in its area of expertise, but does not come without its flaws. Because ECS' effectiveness comes from handling massive quantities of data at once by reusing different elements, it struggles when handling a singular entity. For example, in a single player game there will be only one instance of the player. Due to that, writing the systems required for controlling the player character or such becomes redundant, as these systems will never see use anywhere else, even if they have to account for it.

For the aforementioned reason, debugging entities can also become painstakingly difficult, as none of the code is designed around specific entities, and makes micro-management of the code very limited. Also, because the codebase of ECS has to be as generic as possible, writing systems for trivial functionality may not be as trivial anymore.

One of the largest drawbacks of ECS is that many people are not aware of its existence. The paradigm is fairly complex to understand, and fights against many programming conventions that are thought to be standard in the industry of software development. To get an inexperienced team working with this paradigm is a costly investment due to the training expenses, as well as the time it takes to legitimately assimilate the concepts. If the development team is not sure they are going to actually need this paradigm, it is probable that the risk of failure is not going to be taken. [6]

7. CONCLUSION

In this thesis Entity-Component-System, a paradigm based on Data-Oriented Design was proposed as an alternative to Object-Oriented Programming for complex large-scale software requiring heavy computing for multiple copies of similar data. At the core of the paradigm is decoupling objects into three separated elements, the identifying *entities*, the data containing *components*, and the logic providing *systems*. This approach provides an effective and highly adjustable solution to encapsulation and multi-threading related problems introduced by OOP, but it is not a perfect answer to everything. Writing code for isolated objects in ECS is tedious and introduces some overhead, and working with the paradigm itself is often difficult assimilate, which can delay the starting point of the project significantly.

The main advantage of the paradigm is that it allows the user to manage their data in flexible ways by decoupling objects into components, which can be managed in chunks for a multitude of entities at time. This technique is extremely useful when a mass of similarly constituted needs to be handled. Due to this, the paradigm has history and is to this day commonly used in game engines, where game objects are managed in large groups.

The other major feature of the pattern is the ease of parallelization. By keeping the data as much *read-only* as possible allows for thread-safe distribution between multiple processing threads without fear of synchronization errors. This can be achieved by building an *application programming interface* around the paradigm implementation, which automatically takes care that no illegal operations are made, and all of the programmers working on the projects stay within same guidelines. Due to the convenient parallelization, there have even been experiments of automating the parallelization of separate data transformation chains.

There are many facets to Entity systems which could not be discussed within the constraints of this thesis. One of the most important ones of these was the inter-component, inter-system, and inter-entity communication within the pattern. There are also many other ways of implementation for the data structures and parallelization other than the ones presented in this thesis, and the ones discussed are some of the more straight-forward ones.

REFERENCES

[1] S. Bilas, A Data-Driven Game Object System, Game Developer's Conference, USA, 2002.

[2] S. Bilas, A Data-Driven Game Object System, Game Developer's Conference, USA, 2002, 41 pp., available: https://www.gamedevs.org/uploads/data-driven-game-object-system.pdf.

[3] R. Fermier, Creating a Data Driven Engine, Game Developer's Conference, USA, 2002.

[4] D. Graham , M. McShaffry, Game Coding Complete, Fourth Edition, USA, pp. 155-174.

[5] J. Gregory, Game Engine Architecture, USA, 2018, pp. 1043-1062.

[6] M. Infantino, Entity-Component Systems & Data Oriented Design In Unity, 19 August 2018, available (referred to 28.4.2019): https://github.com/LifeIsGoodMI/ECS-And-DoD-In-Unity/blob/master/ECS_Unity_Article.pdf.

[7] P. Lange, R. Weller, G. Zachmann, Wait-Free Hash Maps in the Entity-Component-System Pattern for Realtime Interactive Systems, IEEE 9th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), USA, 2016, pp. 1-8.

[8] T. Leonard, Postmortem: Thief: Dark Project, 9 July 1999 [Online], available (referred to 11.2.2019): http://www.gamasutra.com/view/feature/131762/postmortem_thief_the_dark_project.php.

[9] N. Llopis, Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot with OOP), 4 December 2009 [Online], available (referred to 28.4.2019): http://gamesfromwithin.com/data-oriented-design.

[10] R. Lord, "Why use an Entity Component System architecture for game development?," 16 February 2012. [Online]. Available (referred to 15.2.2019): https://www.richardlord.net/blog/ecs/why-use-an-entity-framework.html.

[11] A. Martin, Entity Systems are the future of MMOG development, 3 September 2007, available (referred to 03.02.2019): http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/.

[12] A. Martin, Entity Systems: what makes good Components? Good Entities?, 16 March 2012, available (referred to 20.1.2019): http://t-machine.org/index.php/2012/03/16/entity-systems-what-makes-good-components-good-entities/.

[13] A. Martin, Data Structures for EntitySystems: Contiguous memory, 8 March 2014, available (referred to 28.4.2019): http://t-machine.org/index.php/2014/03/08/data-structures-for-entity-systems-contiguous-memory/.

[14] A.Martin, Data Structures for EntitySystems: Multi-threading and networking, 2 May 2015, available (referred to 28.4.2019): http://t-machine.org/index.php/2015/05/02/data-structures-for-entity-systems-multi-threading-and-networking/.

[15] A. Papari, Introduction To Entity Systems, 7 February 2016, available (referred to 20.1.2019): https://github.com/junkdog/artemis-odb/wiki/Introduction-to-Entity-Systems.

[16] V. Romeo, Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library, available (referred to 11.2.2019): https://github.com/SuperV1234/bcs_thesis/blob/master/final/rev1/web_version.pdf, Italy, 2015, pp.145.

[17] T. Stein, The Entity-Component-System - C++ Game Design Pattern, 21 November 2017, available (referred to 20.1.2019): https://www.gamedev.net/articles/programming/general-and-gameplay-programming/the-entity-component-system-c-game-design-pattern-part-1-r4803/.

[18] M. West, Evolve your Hierarchy, 5 January 2017, available (referred to 28.4.2019): http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/.

[19] D. Wiebusch, M. E. Latoschik, Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems, IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), France, 2015, pp. 1-8.

[20] Benefits of Multithreading, available (referred to 28.4.2019): https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqj/index.html.

[21] Unity Data-Oriented Tech Stack manual, available (referred to 20.1.2019): https://github.com/Unity-Technologies/EntityComponentSystemSamples/tree/master/Documentation~.