IMPLEMENTATION OF ENTITY COMPONENT-SYSTEMS IN SCRIPTING

MARCUS

Abstract. As game development continues to be a lucrative industry, it has become increasingly important to make performant experiences while keeping production velocity high[1]. However, this is non-trivial when you have to deal with many objects that have converging behaviour. The traditional approach is to use object-oriented paradigms to construct massive and complicated inheritance paths to share a set of behaviour. However, with the additional indirections from code paths, the reusability of code decreases and performance suffers[2].

To combat these issues, the traditional object-oriented design can be replaced with a more data-oriented design approach utilising a composition-over-inheritance model where the data and logic are separated, called Entity-Component-System (ECS). In this approach, where components and systems can be added to a complex program without interfering with existing logic. This flexibility sets it apart from the aforementioned traditional object-oriented approaches based on heterogeneous collections of explicitly defined object types, where implementing new combinations of behaviours can require far-reaching changes.

The purpose of this thesis is to research the impacts of the memory arrangement and how that affects implementation. Then pivot to explore various elements of ECS, highlighting its advantages, and discussing potential implementations on a conceptual level.

Through comparative analysis, a well-designed ECS completely separates from a naive implementation by leveraging optimized memory layouts and caching to achieve significant performance improvements. These findings provide valuable insights for game developers seeking to build an efficient ECS.

Contents

1. Introduction 2
1.1. Background 2
1.2. ECS Libraries 2
1.3. Purpose 3
1.4. Research Question 3
2. Method 3
2.1. Research Approach 3
2.2. Research Process 3
3. Theory 3
3.1. Entity Component System Architecture 4
3.2. Cache Locality 4
3.3. Data Layouts 4
3.4. SIMD 4

Date: DEADLINES: Draft March 2 and Final version April 4, 2024.

3.5. Vectorization 4
3.6. Archetype 5
3.7. Sparse Set 6
4. Implementation 7
4.1. Data Structures 7
4.2. Functions 8
5. Analysis 10
5.1. Random Access 10
5.2. Updating Component Data 10
5.3. Queries 11
6. Conclusions 12
7. Acknowledgments 12
References 12

1. Introduction

1.1. **Background.** In modern game development, optimizing for performance and production velocity is crucial, However, traditional object-oriented programming (OOP) approaches often encounter challenges when dealing with the complexity of game systems, particularly in managing large amounts of data efficiently. One significant issue is the lack of data locality, which refers to how closely related data elements are stored in memory.

In 2011, Robert Nystrom published his book Game Programming Patterns. In the chapter on optimization patterns, Nystrom expounds on the importance of data locality. This concept, while not revolutionary, underscores the fundamental role of data storage, referencing, and manipulation is the impetus for creating any program. Without efficient data locality, programs may suffer from increased cache misses and slower memory access times, leading to performance bottlenecks and decreased frame rates. This problem becomes more pronounced as games become more sophisticated and demand higher fidelity graphics, complex physics simulations, and larger virtual worlds. It was this chapter that motivated the research into the relationship between data locality and the implementation of Entity Component System.

1.2. ECS Libraries.

1.2.1. *Matter.* Matter, an ECS library written in Lua, provided with a debugger and scheduler that has been developed specifically for Roblox, makes it easy to use and understand.

Matter was selected for this paper to provide a baseline threshold to benchmark against.

1.2.2. *Flecs.* Flecs is an efficient ECS made for games and simulations with many entities. It also has an elaborate query engine that is capable of finding entities by relationships[3] and can embed multitudes of operations into its queries.

Flecs was chosen because of its exhaustive API coupled with an involved community and in-depth documentation.

1.2.3. *Hecs.* Hecs, a lightweight ECS that aims to be unobtrusive by being a library and not a framework.

Hecs also has an archetypal storage and is the main inspiration for Matter. This was the reason for why it was chosen for this paper.

1.3. Purpose.

The traditional OOP paradigm, with its emphasis on class hierarchies and inheritance, often results in poor data locality due to how objects and their associated data are stored in memory. As a result, game developers are turning to data-oriented design (DOD) principles to address these performance issues.

By adopting a data-oriented approach, such as the ECS architecture, developers can restructure their code to prioritize data locality. ECS separates game entities into discrete components, each containing only the data relevant to a specific aspect of gameplay. Systems then operate on these components in a data-driven manner, promoting cache efficiency and reducing memory access overhead.

However, despite the potential benefits of ECS and other data-oriented techniques, many developers still face challenges in understanding and implementing an ECS efficiently. This gap underscores the need for comprehensive research and documentation to explore the implications of data locality in game development and provide practical solutions for optimizing an ECS implementation.

1.4. Research Question.

How can the ECS architecture be optimized to address the limitations of traditional object-oriented techniques?

2. Method

2.1. Research Approach.

This research project is an exploratory study with the aim of gaining an understanding of the inner workings of ECS. This study will adopt a mixed-methods approach, with both inductive and deductive reasoning. This approach is chosen to provide a comprehensive understanding of the relationship between entity-component-systems, data locality, and performance.

2.2. Research Process.

The research process will start with a thorough study of the literature related to the field of ECS. Relevant research concepts will be summarized and presented in the Theory chapter to construct a theoretical framework. This framework will be used for analysis in the empirical part of the study.

The empirical part of the project consists of a comprehensive case study. Multiple ECS implementations will be tested and analysed, using the framework constructed in the theoretical part.

An implementation of an ECS from scratch will further be conducted in order to experiment with different storage layouts. Through this iterative process, insights into the optimal design and implementation of ECS will be gained, with a particular focus on addressing performance bottlenecks related to data access and manipulation.

3. Theory

This theory chapter is dedicated to forming the theoretical foundation of ECS architecture. The reader will get a fundamental understanding of what ECS is, what makes it useful, and what the key elements of the architecture are. Together, these parts form a theoretical framework which will be used as the base of both the empirical and implementation part of the study.

3.1. Entity Component System Architecture. The Entity Component System (ECS) architecture provides infrastructure for representing distinct objects with loosely coupled data and behaviour. Data is stored in contiguous storage types to promote cache optimality which benefits performance. An ECS world consists of any number of entities (unique IDs) associated with components, which are pure data. The world is then manipulated by systems that access a set of component types.

3.2. Cache Locality. When a CPU loads data from Random Access Memory it is stored in a cache tier (i.e. L1, L2, L3), where the lower tiers are allowed to operate faster relatively to how closely embedded it is to the CPU.[3] When a program requests some memory, the CPU grabs a whole slab, usually from around 64 to 128 bytes starting from the requested address, and puts it in the CPU cache, i.e. cache line. If the next requested data is in the same slab, the CPU reads it straight from the cache, which is faster than hitting RAM. Inversely, when there is a cache miss, i.e. it is not in the same slab then the CPU cannot process the next instruction because it needs said data and waits a couple of CPU cycles until it successfully fetches it. (Nystrom, 2011).

3.3. Data Layouts.

3.3.1. *Array Of Structs.* Array of Structs organizes data in a way where each struct is stored as elements within an array, arranged in rows (see code snippet). This memory arrangement is frequently utilized in object-oriented programming, mirroring how classes inherently structure their data members.[3]

```
struct AoS {
  foo: i64;
  bar: i32;
}

values: Vec<AoS>
```

3.3.2. *Struct of Arrays.* Struct of Arrays organizes data in a way where each field of an entity is stored in separate arrays or ”columns” (see code snippet). This memory arrangement in memory in a way that will be more beneficial to CPU performance as it can better predict the next memory access.[3]

```
struct SoA {
  foo: Vec<i64>;
  bar: Vec<i32>;
}

values: SoA
```

3.4. SIMD. Single Instruction Multiple Data (SIMD) is a type of parallel computing that performs the same operation on multiple values simultaneously.

3.5. Vectorization. Vectorization is where code meets the requirements to use SIMD instructions. Those requirements are that: - Data must be stored in contiguous arrays - The code should contain no branches or function calls

3.6. **Archetype.** Storing data in contiguous arrays to maximize vectorization and SIMD is the ideal situation, however it is a very complex problem in implementation. Below the ABC problem[4] is demonstrated where 3 entities all have the component `A` which can be stored in a single column:

```
0: [A]
1: [A]
2: [A]
```

Now suppose entity 0 and entity 2 have the component B, leaving a gap between the lower and upper bound entities in the component *B* array (column). The column is now non-contiguous which means it cannot be vectorized or use SIMD:

```
0: [A, B]
1: [A,  ]
2: [A, B]
```

The components in the rows are stored contiguously, but the traversal over the entities cannot be vectorized for code that requires both *A* and *B*. To make these components contiguous in memory again, the entities at indexes 1 and 2 are swapped, resulting in the following organization:

```
0: [A, B]
2: [A, B]
1: [A,  ]
```

However there are no operations that can fix the entity indexes when there are more than two columns and if there are every combination of components present in component storage:

```
0: [ , B,  ]
1: [ , B, C]
2: [A, B, C]
3: [A, B,  ]
4: [A,  ,  ]
5: [A,  , C]
6: [ ,  , C]
```

This problem is called the “ABC problem” which requires a relaxation in order to support vectorization. Which is what developers have found that archetypes solves.[4] Archetypes are semantically identical to “tables”. Each archetype contains only one type of entity, meaning each unique combination of components defining an entity has its own archetype. Below the following illustration is demonstrated where 2 entities only has component `A`, 2 entities with `A` and `B` and finally 2 entities with both `A` and `C` (see code snippet). It follows the same SoA principles where each component type has a column in the archetype. Rows in the archetype correspond to specific entities, with each entity intersecting components in the archetype.

```
1: [A]

2: [A, B]
3: [A, B]

4: [A, C]
5: [A, C]
```

This type of organization enables fast querying and iteration, however it also presents different challenges. Modifying entities, such as adding or removing components, or adding new entities, can be costly operations. Each change necessitates searching for the appropriate archetype, potentially creating a new archetype, and updating entity placements in archetypes which is really slow and requires traversal over every entity to find the archetype that the entity is in.

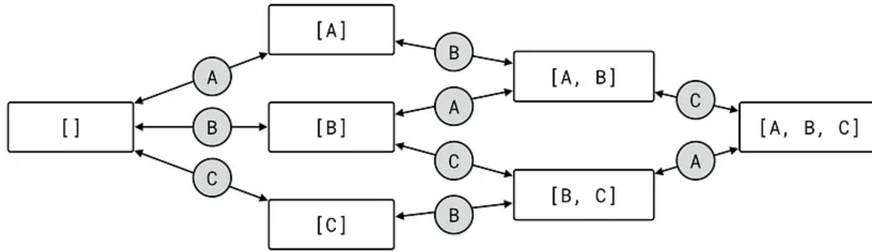
To move entities faster between archetypes, a common optimization is to keep references to the next archetype based on component types (see Figure 1). Each edge in the graph corresponds to a component that can be added, akin to an intersection operation on the archetype set.

$$A \cap (B \cap C)$$

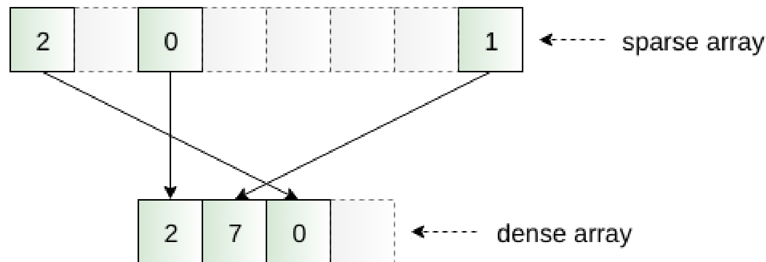
Removal of a component from the archetype is akin to a subtraction operation from the set.

$$A \cap (B \cap C) - C$$

‘. This archetype graph facilitates O(1) transitions between adjacent archetypes to mitigate the cost of fragmentation.

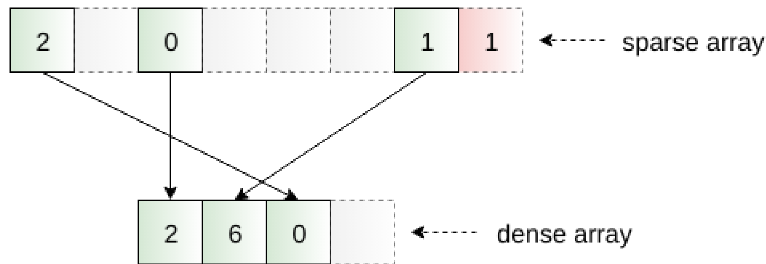


3.7. **Sparse Set.** Sparse sets organize each component type into its own densely packed array. A sparse set is composed of two arrays, one densely packed and one sparsely populated. The sparse array contains the position of an entity ID that is stored in the dense array (see Figure 2). Components are stored in parallel to entities which allows for insertions and removals of components at O(1) constant time as it is just setting a single value in an array. The trade-off is that it is less memory efficient as it revolves around many repeated random access and it requires ‘2n‘ memory units to store these indices in two arrays. However, they serve different purposes. The dense array is for operations over many entities such as iteration while the sparse array is for single entity lookup.



To add an entity to the sparse set, it is pushed back onto the dense array and the sparse array is updated with the entity as the key, while the index representing its position in the dense array becomes its corresponding value. This ensures constant-time lookups to see whether an entity is contained in the sparse set: `dense[sparse[i]] == i`. However, removing an entity is more complicated as it involves swapping it with the last entity in the dense array and updating their respective positions in the sparse array. For instance, if entity 6 (indexed at 3 in the dense array) is to be removed, it is swapped with the last entity (e.g. entity 7), and the corresponding entry in the sparse array is adjusted to reflect this change. The removed entity is then simply removed from the end of the dense array. This operation ensures that the dense array remains tightly packed, facilitating efficient data management.

However, removing an entity is more complicated as it involves swapping it with the last entity in the dense dense array and updating its corresponding position in the sparse array. Using the previous Figure as an example, if the entity 6 (indexed at 3 in the dense array) is to be removed then it will be swapped with the last entity which is entity 7 and the corresponding entry in the sparse array will be updated. The removed entity is then simply removed from the end of the dense array (see Figure 3). This method ensures that the dense array remains tightly packed.[5]



The sparse set structure is beneficial for programs that frequently manipulate the component structures of entities. However, querying multiple components can become less efficient due to the need to load and reference each component array individually. In contrast to archetypes, which only needs to iterate over entities matching their query.

4. Implementation

The decision to use Lua scripting language for the ECS implementation was ultimately chosen because a pure Lua implementation confers distinct advantages in terms of compatibility and portability. By eschewing reliance on external C or C++ libraries or bindings, we ensure that our ECS framework remains platform-agnostic and compatible across various game engines. While some game engines offer support for integrating native code written in C or C++, not all engines provide this capability. Therefore, by keeping our implementation solely within the Lua environment, we maximize compatibility across different engines and platforms, including those that may lack native code integration capabilities.

4.1. **Data Structures.** The ECS utilize several key data structures to organize and manage entities and components within the ECS framework:

- **Archetype:** Represents a group of entities sharing the same set of component types. Each archetype maintains information about its components, entities, and associated records.
- **Record:** Stores the archetype and row index of an entity to facilitate fast lookups.
- **EntityIndex:** Maps entity IDs to their corresponding records.
- **ComponentIndex**: Maps IDs to archetype records.
- **ArchetypeIndex**: Maps type hashes to archetype.
- **ArchetypeMap:** Maps archetype IDs to archetype records which is used to find the column for the corresponding component.
- **Archetypes:** Maintains a collection of archetypes indexed by their IDs.

These data structures form the foundation of our ECS implementation, enabling efficient organization and retrieval of entity-component data.

4.2. **Functions.** The ECS needs to know which components an entity has and provide an interface to manipulate it and search for homogenous entities from a set of components quickly.

4.2.1. *get(entityId, . . .)* **Purpose:** The get function retrieves component data associated with a given entity. It accepts the entity ID and one or more component IDs as arguments and returns the corresponding component data.

```
local function get(entityId: i53, a, b, c, d, e)
    local id = entityId
    local record = entityIndex[id]
    if not record then
        return nil
    end

    return getComponent(record, a), getComponent(record, b) ...
end

local function getComponent(record: Record, componentId: i24)
    local id = record.archetype.id
    local archetypeRecord = componentIndex[componentId][id]

    if not archetypeRecord then
        return nil
    end

    local column = archetypeRecord.column

    return archetype.data.columns[column][record.row]
end
```

Explanation: This function retrieves the record for the given entity from `entityIndex`. It then calls `getComponent(record, componentId)` to fetch the data for each specified component `(a, b, c, d, e)` from the entity's archetype which is returned.

4.2.2. *entity().* **Purpose:** This function is responsible for generating a unique entity ID.

```
  local nextId = 0
  local function entity()
      nextId += 1
      return nextId
  end
```

Explanation: Generates a unique entity ID by incrementing a counter each time it is called.

4.2.3. *add(entityId, componentId, data).* **Purpose:** Adds a component with associated data to a given entity

```
local function add(entity, id, data)
  local record = ensureRecord(entityId)
  local source = record.archetype
  local destination = archetypeTraverseAdd(id, source)

  if not source == destination then
    moveEntity(entityId, record, destination)
    -- update query cache
  else
    if #destination.types > 0 then
      newEntity(entityId, record, destination)
    end
  end

  local archetypeRecord = destination.records[componentId]
  local columns = destination.data.columns
  columns[archetypeRecord.column][record.row] = data
end
```

Explanation: This function first ensures that the record exists for the given entity using `ensureRecord()`. It then determines the destination archetype from the current entity archetype and new component using `archetypeTraverseAdd()`. It will move the entity to a new archetype or if the entity does not have a record yet, initializes the record by calling `newEntity()`. Lastly it updates the data for the component in the corresponding column of the archetype's data.

4.2.4. *query(...)* **Purpose:** Performs a query against the entities that exists based on the specified components.

```
  local function query(a, b, c, ..)
    local entities = {}
    for archetype in archetypesWith(a) do
        if not archetypesWith(b)[archetype] then
          continue
        end
        if not archetypesWith(c)[archetype] then
          continue
        end
        ... -- match archetype if every archetype
        ... -- from the specified components are compatible
    end
```

```
    local i = 0
    return function()
        i+=1
        if i > #entities then
            return
        end
        local entity = entities[i]
        local record = entityIndex[entity]
        local archetype, row = record.archetype, record.row

        local columns = archetype.data.columns
        local id = archetype.id

        return entity,
            columns[componentIndex[a][id].column][row],
            columns[componentIndex[b][id].column][row],
            columns[componentIndex[c][id].column][row]
            ...
    end
end
```

Explanation: This function through retrieves all archetypes that have the first component in the specified component set through `archetypesWith()` that goes through the `ArchetypeMap` which maps a component to a set of all of the archetypes with that component. The query stacks operations that evaluate the conditions one by one with the subsequent components in the set. When an archetype gets matched, it iterates through the entities in that archetype and fetches the data for the component for each entity.

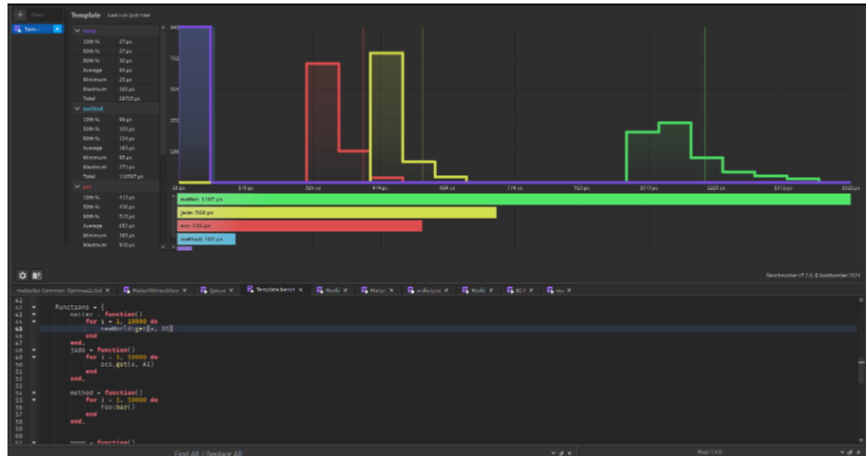
5. Analysis

There are three main operational aspects to measure for performance to evaluate the efficiency of the ECS, namely updating component data, random access and queries. Each of these aspects provide metrics to examine the performance characteristics and identify key areas for optimization.

5.1. **Random Access.** Retrieving component data associated with a specific entity is often slow because it requires multiple random access into memory due to map lookup. This is exemplified by Matter requiring multiple indirections to look through an entity in all of the storages with two subsequent map lookups using the entity archetype.

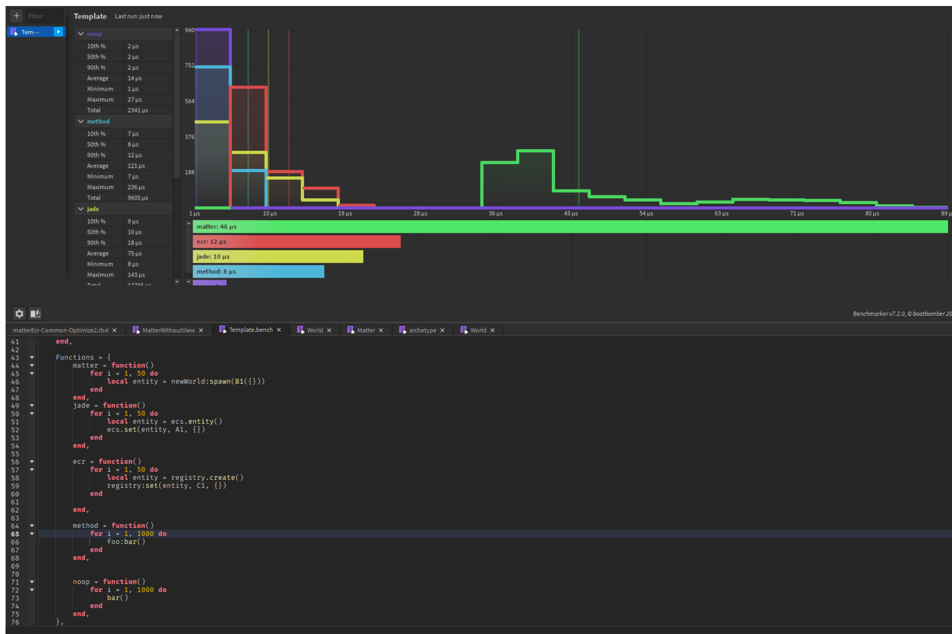
However, with specific locations of component data memoized by the column and row respectively as specified by Flecs, constant $\mathsf{O}(1)$ time data retrieval can be achieved by mostly array lookups as evident by Jade, an alias for the ECS implementation made during this paper that outperformed Matter by 98.25% (see below).

5.2. **Updating Component Data.** Insertions and Removals of component data being slow was expected due to that moving many overlapping components between archetypes costs a lot of computation when reconciling the columns and rows. Matter is especially slow here because it needs to naively look through every storage to find its old archetype and it has to create a new the new archetype every time



an entity is updated. Instead, Jade has amortized this cost by caching edges to adjacent archetypes on the graph (see Figure 1).

The result is that updating data was 360% faster than Matter (see below).



5.3. **Queries.** Matter is incapable leveraging very performant queries due to it is failing cache locality under adverse conditions as entities data is stored in AoS that requires heaps of random accesses, including many unnecessary hash lookups. It is also naively populating the query cache by iterating over every archetype in the world in linear time which scales poorly as there are always going to be more archetypes than components.

Jade saw a 93.9% increase in iteration speed by having memoized the entity locations by their column and row indices for fast indexing during contiguous traversal

over homogeneous entities. Query creations are also cheaper as populating the cache is cheaper when only iterating over archetypes with a common component.

6. Conclusions

Through the exploration of ECS and its performance characteristics, this research sheds light on crucial insights on various optimization strategies of highly abstracted memory layouts. The theoretical framework established highlights the significance of prioritizing data locality and separating data and logic in game systems. Additionally, the empirical analysis of various ECS implementations underscores the importance of memory arrangement and efficient data manipulation strategies.

Implementations such as Flecs exhibit superior performance by structuring memory layouts to minimize indirections, resulting in constant-time data retrieval for random access. Conversely, approaches like Matter, relying heavily on map lookups, experience performance penalties in random access operations. Implementations with poor cache locality, exemplified by Matter, struggle with slow query performance due to excessive random accesses during adverse locality conditions and emphasized the importance of caching strategies.

In conclusion, the ECS architecture offers a promising solution for addressing performance challenges in game development, however it needs to be implemented carefully in order to not have performance penalties.

7. Acknowledgments

I am grateful to Sanders Mertens for insightful discussions on archetypes and meticulous evaluations of the minimal ECS implementation iterations. My thanks also extend to Eryn L. K. and Lucien Greathouse for their invaluable guidance and contributions to the Matter project.

References

[1] Adam Martin (2007, September). Entity Systems are the future of MMOG development - Part 1. https://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/
[2] Casey Muratori (2014, June). Semantic Compression. https://caseymuratori.com/blog_0015
[3] Sanders Mertens (2019). Flecs. https://github.com/SanderMertens/flecs
[4] Carter Anderson (2022). Bevy. https://github.com/bevyengine/bevy

[5] Michele Caini (2020, August). ECS back and forth. https://skypjack.github.io/2020-08-02-ecs-baf-part-9/
[6] Robert Nystrom (2011). Game Programming Patterns.
[7] Scott Bilas (2002). A Data-Driven Object System (GDC 2002 Talk by Scott Bilas). https://www.youtube.com/watch?v=Eb4-0M2a9xE
[8] Matter, an archetypal ECS for Roblox https://matter-ecs.github.io/matter/