UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Development of an Entity Component System Architecture for Realtime Simulation

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Trevor Hollmann

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Kevin Keul, M.Sc. (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

The development of a *game engine* is considered a non-trivial problem. [3] The architecture of such simulation software must be able to manage large amounts of simulation objects in real-time while dealing with "crosscutting concerns" [3, p. 36] between subsystems.
The use of object oriented paradigms to model simulation objects in class hierarchies has been reported as incompatible with constantly changing demands during game development [2, p. 9], resulting in anti-patterns and eventual, messy re-factoring. [13] Alternative architectures using data oriented paradigms revolving around object composition and aggregation have been proposed as a result. [13, 9, 1, 11]
This thesis describes the development of such an architecture with the explicit goals to be simple, inherently compatible with data oriented design, and to make reasoning about performance characteristics possible. Concepts are formally defined to help analyze the problem and evaluate results. A functional implementation of the architecture is presented together with use cases common to simulation software.

Zusammenfassung

Die Entwicklung einer *Spiele-Engine* wird als nichttriviales Problem betrachtet. [3] Die Architektur einer solchen Simulationssoftware muss in der Lage sein eine große Ansammlung von Simulationsobjekten in Echtzeit zu verwalten und dabei Subsysteme mit horizontalen Abhängigkeiten ("crosscutting concerns" [3, p. 36]) zu koordinieren.
Simulationsobjekte mittels objektorienterter Modellierung in Klassenhierarchien einzuordnen ist als inkompatibler Ansatz für sich ständig im Wandel befindende Spiele-Software beschrieben worden [2, p. 9]. Das Auftauchen von Anti-Pattern und kompliziertes Code-Refactoring werden als Folgen genannt. [13] Als Maßnahme wurden Architekturen vorgeschlagen, die Objektkomposition mit datenorientierten Paradigmen verknüpfen. [13, 9, 1, 11]
Diese Arbeit beschreibt die Entwicklung einer solchen Architektur mit dem erklärten Ziel simpel und inhärent kompatibel mit einer datenorientierten Perspektive zu sein und Schlussfolgerungen über Performance-Charakteristiken überschaubar zu gestalten. Die Konzepte der Architektur werden formal definiert, um bei der Analyse der Aufgabe und der Bewertung der Ergebnisse zu helfen. Eine lauffähige Implementation dieser Architektur sowie typische Anwendungsbeispiele von Simulationen werden präsentiert.

Contents

1 Introduction

This section gives an overview of the problem domain of simulation software, introduces general concepts, defines terminology and briefly discusses known solutions for a selection of problems. Section 2 examines different concepts for the architecture of simulation software described by papers and articles. Based on these a novel architecture is proposed by this thesis and defined in section 3. As proof of concept this architecture is implemented. Section 4 describes the implementation including several use cases common to simulation software. The architecture is then evaluated in section 5 based on the insight gained from the implementation and the analysis of the problem domain in this section. Finally, the results are summarized for the conclusion and an outlook for future direction is given in section 7.

1.1 Simulation Software

A *game engine* is a piece of software that can be described as a *soft real-time interactive agent-based computer simulation*[7, p. 11]. *Soft real-time*, in the sense that clear time constraints exist, but a violation is not considered catastrophic. *Interactive* means that user input is reflected by the produced output. The relation between this input and output must be recognizeable, just as the behaviour of the *agents* must follow clear rules to make it a *simulation*. This bachelors thesis will consider all such simulation software, but will focus its examples on game engines for illustrative purposes.

1.2 Problem Domain

The task of simulation software is to compute changes in the state of simulation objects $e \in E$ over time. A simulation starts with an initial set E_0 of simulation objects and a set of rules S. The state of the simulation is changed by applying the rules $s_1..s_m \in S$ to the current set $e_1..e_n \in E_i$ of simulation objects, producing the next iteration of the state E_{i+1}.

$$E_{i+1} := \{E_i \oplus s \mid \forall s \in S\} \tag{1.1}$$

The set of rules S is fixed for the duration of the simulation. A simulation objects behaviour is defined by its transitions from one state to another. Some simulations require determinism of state transitions, others impose requirements on the resolution of the temporal axis. Game engines commonly compromise both for the benefit of runtime performance. The number of simulation objects and their ability to interact with each other affects the amount of computation required. The interaction between simulation objects can influence their state and cause the removal or addition of objects to or from the simulation.

1.2.1 Complexity of Interactions

For the purpose of this thesis an interaction is defined as the application of a rule $s \in S$ to a pair of simulation objects $E_s := \{e_j, e_k\} \subset E$

$$s(E_s) \to E'_s \mid E'_s \subset E \tag{1.2}$$

where the result is a set of simulation objects. Note that the elements of the pair might be the same $j = k \implies e_j = e_k \implies E_s = \{e_j\}$ and that the resulting set may be the empty set $\emptyset$. While the majority of objects do not interact with each other most of the time, the simulation has to test for the occurrence of *possible* interactions. When every simulation object can interact with every other the complexity of the simulation grows with the number $n := |E|$ of simulation objects and the number $m := |S|$ of rules. The rate of this growth depends on the type of interactions with the upper bound generally being factorial for each interaction.

$$p \leq n! * m \tag{1.3}$$

Where p is the number of possible object interactions for n simulation objects and $m := |S|$ different types of interaction. Complexity can vary greatly between different types of interaction, however. Collision tests can generally be implemented as commutative operation $a \oplus b = b \oplus a \mid \forall a, b \in E$ such that any two simulation objects a, b only need to be tested against each other once. For these types of interaction p is a weak upper bound. In contrast, visibility testing with directed vision is not commutative. a, b might be connected by a direct line of sight, but due to limited vision cones (eg. of cameras) the spatial orientation of a determines whether a has vision of b without affecting if b has vision of a. Thus, both interactions $a \oplus b$ and $b \oplus a$ have to be tested and only the interaction $a \oplus a$ of a simulation object with itself can be omitted. Simpler interactions such as applying gravity as a static velocity to each simulation object have linear complexity $O(n)$.

The order in which the interactions are resolved can affect the behaviour of the simulation objects and may be seen as an additional source for complexity. However, since the set of rules S is fixed for the runtime of the simulation, this order can be predetermined and will generally not be decided at runtime. In conclusion, no strong statement about the complexity of a simulation can be made without knowing the set of rules, which change considerably between different simulations.

1.2.2 Common Solutions

A common approach to these problems is to divide the set of simulation objects into smaller subsets based on suitable features and then limit interactions to objects within the same subset. This technique will be referred to as *grouping* throughout this thesis. Which features are suitable for grouping depends on the types of interaction. Spatial proximity works well for interactions with a limited range like the collision of bounding boxes. A variant of this commonly used in video games is to section the simulation world into levels.

On a finer grained scale the space may be subdivided using algorithms such as binary space partitioning [6, p. 337] or structures like octrees [7, p. 516]. These also prove suitable for spatially directed interactions such as ray tracing.[12]

In a simulation with different types of simulation objects some interactions only apply to objects of a certain type or with a specific aspect. This is true for simplified approximations of reality such as video games where some objects aren't affected by physics or suddenly become invisible due to gameplay conditions. Such aspects can be used to subdivide the set of simulation objects and

optimize the layout of data for the computation of certain interactions. However, the choice of aspects to group by is not as straight forward as the division by spatial proximity. Separating objects with a bounding box from those without may be beneficial for collision detection but detrimental to graphics rendering where grouping of objects with a visual representation is more important.

Table 1 gives an example of how simulation objects may be grouped based on their aspects. Equation 1.4 compares the complexity of computing a collision interaction with and without this grouping.

Subset	Moving	Collidable	Count	Example
A	✓	✓	2	Player characters
B	✓	-	2	Cameras
C	-	✓	5	Floors & walls
D	-	-	1	Particle effects

Table 1: A set E of 10 simulation objects grouped into 4 subsets based on two of their aspects.

To test for collisions in the scenario given by table 1, only simulation objects from subset $E_1 := A$ need to be tested against those from subset $E_2 := \{A \cup C\}$. Equation 1.4 shows that the number of tests can be reduced when simulation objects can be grouped into subsets based on their aspects. Figure 2 illustrates the difference for the example given by table 1.

$$\begin{aligned} \text{Let } E_1, E_2 \subset E \neq \emptyset \text{ and } n := \|E_1\|, m := \|E_2\| \\ \implies n < \|E\| \wedge m < \|E\| \\ \implies c_1 = n * m < \|E\|^2 = c_2 \end{aligned} \tag{1.4}$$

Figure 1: Where c_1, c_2 are the number of collision tests necessary for a set of entities E with and without grouping respectively.

Whether or not such optimizations can be implemented depends on the architecture of the simulation software. Specifically, the part of the architecture responsible for the management of simulation objects. A definition for this part has been proposed by Doherty [4] under the term *object system*: "The object system is responsible for maintaining the state information describing all objects in the game world." It is clearly separated from other parts of the architecture. Bilas [2] presents a definition in two parts: A "Game Object (Go)" is a "piece of logical interactive content" that performs "tasks like rendering, path finding, path following, speaking, animating, persisting". Then, a "Game Object System" "constructs and manages Go's", "maps ID's to object pointers", "routes messages" and is built "from many things" including a "Go database" and "static content database".

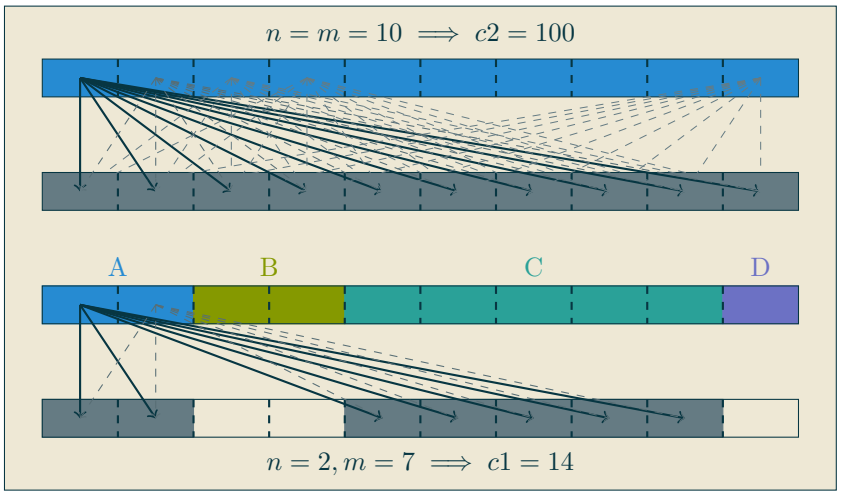

Figure 2: Illustation of how interactions are reduced by grouping simulation objects into categories based on the example in table 1. Equation 1.4 was used to derive exact numbers.

1.3 Core Concerns

From the above description we can derive a few abstract operations that are core concerns of any architecture meant to manage simulation objects and simulate their behaviour.

Con- & Destruction of simulation objects.
This operation adds or removes at least one element from the set of simulation objects.

Iteration over simulation objects.
This operation is a form of access to the set or a subset of the simulation objects such that each simulation object is visited once.

Addressing of simulation objects.
This operation is a form of access where a specific simulation object is selected from the set based on a unique address.

Grouping of simulation objects.
This operation divides the set of simulation objects into subsets based on some feature. The main purpose is to limit the range of elements that are operated on and thus improve performance.

Table 2: Core concerns for the design of the architecture of simulation software.

These operations can not be treated as independant from each other. For example, *grouping* is required to allow *iteration* over subsets of simulation objects. *Addressing* requires that *construction* includes the generation of some form of address, which is tied to the same simulation object until the *destruction* of that

simulation object.

It must also be noted that these operations may overlap. An access operation such as *iteration* might be able to *destroy* a simulation object if it has write-access. Similarly, *addressing* may be implemented with implicit *construction* of a simulation object if the specified address is unused. This is a common strategy for dictionary structures such as the "unordered_map"[1] from the *C++ Standard Template Library*, or the "dictionary"[2] in the *Python programming language*.

When designing an architecture for managing simulation objects, it is not enough to have established abstract operations. The architecture must also consider which operations to prioritize. If two operations pose conflicting goals for data organization, a balance must be found. For example, iteration is fastest on sequential data but searching profits from hierarchical organization.

1.4 Terminology

Term	Meaning
Simulation software	A software framework for implementing computer simulations.
Simulation engine	
Game engine	A specific type of simulation software.
Simulation object	An object being simulated.
Entity	
Simulation rule	A behaviour being simulated.
Interaction	Resolving operations between entities.
Con-/Destruction	The addition or removal of a simulation object to or from the simulation.
Iteration	Accessing a set of simulation objects sequentially.
Addressing	Regarding simulation objects, the access of a specific simulation object.
Grouping	Building subsets of simulation objects with a common trait.

Table 3: Recap of the terminology introduced in this section.

[1] `http://www.cplusplus.com/reference/unordered_map/unordered_map/operator[]/`
[2] `https://docs.python.org/3/tutorial/datastructures.html#dictionaries`

2 Established Paradigms

This section discusses architectural paradigms for real time simulation software. Based on papers and articles a transition is described from a traditional - object oriented - paradigm of inheritance modeling towards a composition oriented paradigm based on entity-component relations.

2.1 Object Orientation with Inheritance

A traditional approach to implementing simulation objects in game engines is to construct an inheritance tree for an "object oriented decomposition of the set of entities" [13]. As noted by Bilas [2], there is more than one possible decomposition of the set of "game objects" into classes. It is pointed out that the class hierarchy can be considered a hard coded database [2, 12] but the "games constantly change" [2, 9] - meaning the hard coded database has to be adjusted by hand to fit the needs of evolving content. The static nature of a hard coded class hierarchy limits the interaction of simulation objects based on where a class is placed in the inheritance tree and what methods it has.

Mick West points out in [13] that "the traditional game object hierarchy ends up creating the type of object known as "the blob". The blob is a classic "anti-pattern" which manifests as a huge single class [...] with a large amount of complex interwoven functionality."[3] A similar observation from [8] also highlights the conflict of this anti-pattern with principles of object oriented design: "At the end, programmers often came up with gigantic classes that could handle every possible situation. This works but does not make good usage of programming patterns whose purpose is to save time and ease code management."[4]

2.2 Evolution: Composition Over Inheritance

To counter the problems described above, alternative architectures for structuring simulation object data have been explored. The solutions presented by [2, 9, 13, 8] can be categorized as composition based architectures in which simulation objects are represented as collection of components. In these architectures inheritance hierarchies are either flat or non-existent. The "is-a" relationship between simulation objects in an inheritance hierarchy is replaced by a "has-a" relationship between a single simulation object and multiple components. A simulation object is still an object in the context of the object oriented paradigm but is referred to as *entity*. Other object oriented practices such as polymorphism may be used for collecting components of sub-classed types. Composition and inheritance are not mutually exclusive and are commonly described alongside each other as tools for modeling relationships between simulation objects. Gregory introduces both as complementary practices in [7, pp.98-103].

One significant difference in the solutions presented here to a traditional approach is how the behavior of these entities is implemented. Instead of class

[3] `https://web.archive.org/web/20190116045950/http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/` (accessed Aug. 11, 2019)

[4] `https://web.archive.org/web/20180914193022/http://www.thepulsar.be/article/entities--components-and-message-handling-in-games` (accessed Aug. 11, 2019)

methods that modify member variables when called on an object, these entities do not define behavior themselves and the set of methods is reduced to constructors, destructors and data access such as iteration of the components held by the entity. It is the components that define behavior, either directly by providing methods or indirectly through mere presence. An entity with a *physics component* will be treated differently from an entity without it, thus having different behavior within the simulation.

[9] describes components as representing "aspects" of an entity: "Every in-game item has multiple facets, or aspects, that explain what it is and how it interacts with the world. [...] The Component does one really important thing: Labels the Entity as possessing this particular aspect".[5]

In [2] the components are derived from a single base class "GoComponent" and entities are instances of the "Go" class. Each "*owning*" a set of "GoComponents" in a "has-a" relationship.

[8] describes an architecture without hierarchy and where simulation objects "are described as collections of behaviors which are implemented into components." Here, communication between simulation objects is solved by passing message objects between components, which must implement the interface for message classes.

In [13] West describes an architecture where entities are pure aggregations of components and there is no class or instantiated object to represent the entity itself. All components are derived from a base class and share a common interface. A "component manager" exists to resolve access to component references, though other means to access components directly were later added due to performance concerns: "Ideally, components should not know about each other. [...] Initially we had all component references going through the component manager, however when this started using up over 5% of our CPU time, we allowed the components to store pointers to one another, and call member functions in other components directly."[6] No details are given on how components are associated as belonging to the same entity.

2.3 Entities, Components and Systems

The solutions discussed in section 2.2 can be considered half-way towards a pure Entity Component System. It was briefly mentioned that in some architectures the components do not implement behavior themselves, but their presence determines the behavior. The notion of components as "aspects" of an entity was introduced, but details about the implementation of behavior in these cases were left out. This section will focus on architectures of this kind, as described by Martin [9].

Here, behavior is implemented by "systems" - code that is well separated from the definition of entities or components. As Martin points out: "A System essentially provides the method-implementation for Components of a given aspect, but it does it back-to-front compared to OOP. OOP style would be for each Component to have zero or more methods, that some external thing has to

[5] `https://web.archive.org/web/20180617155000/http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/` (accessed Aug. 12, 2019)

[6] `https://web.archive.org/web/20190116045950/http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/` (accessed Aug. 12, 2019)

invoke at some point. ES style is for each Component to have no methods but instead for the continuously running system to run it's own internal methods against different Components one at a time."[7]

A system generally selects a subset of components based on type and processes their data. Changes made to this system can only affect components of that type. Conversely, adding a new type of component won't affect the simulation unless there is also at least one system processing it.

In this architecture the components may still be instantiated objects of a class definition with constructor and destructor, but they are no longer responsible for updating their member variables. The separation of data and logic allows for an implementation of components without classes or other common object oriented practices. Martin goes as far as stating that "If you think of Entities/Component's in OOP terms, you will never understand the ES, and you will screw up any attempt to program with it."[8]

He goes on to state that the "Entity System" in this form is a different paradigm than "Object Oriented Programming" and that "It's fundamentally different and incompatible."[9]

[7] https://web.archive.org/web/20180617155000/http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ (accessed Aug. 12, 2019)

[8] https://web.archive.org/web/20180617155000/http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ (accessed Aug. 13, 2019)

[9] https://web.archive.org/web/20180617155000/http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ (accessed Aug. 13, 2019)

3 The Entity Component System Architecture

This section introduces the architecture proposed by this thesis. First, the concept is described informally to convey the idea while choices are put in context of the discussion in section 2. Then, a precise definition of the architecture is given using the terminology from section 1.

3.1 Concepts

At the core the Entity Component System Architecture has three concepts that are very simple on their own.

Components are simple structs of *plain old data* [7, p. 336].

Entities identify sets of components to represent simulation objects.

Systems are algorithms that process the data stored in components.

The concept of how these are combined, including all arising implications, is the **architecture**.

Figure 3: Schematic illustration of the architecture. Components and entities are presented in a matrix where rows are component-arrays and columns are entities. Systems march through the arrays, one column (entity) at a time to process data stored in components.

Figure 3 illustrates the general concept of the architecture. Components are stored in "parallel arrays"[10], henceforth referred to as *component-arrays*. A simulation object is represented by an entity, which is just a set of components that label the entity as having certain aspects, similar to the design described in [9]. Arrangement of data in these arrays follows the SoA[11] principle with components of the same type packed together in memory. The behavior of entities is implemented by systems that iterate over these arrays sequentially to process the components, similar to the description in [9]. Each system has a narrow focus on a few aspects of behavior and iterates over the relevant component-arrays only.

There is *no* "Entity"-class that points to the components or stores them in a list. This is similar to the architectures described by [8, 13]. However, unlike the

[10] `https://en.wikipedia.org/wiki/Parallel_array` (accessed Aug. 13, 2019)
[11] `https://en.wikipedia.org/wiki/AoS_and_SoA` (accessed Aug. 13, 2019)

approach from [8], no message objects are passed between components and they aren't classes implementing a common interface. A central manager resolving all component references is described by [13] as too slow, their workaround is to allow direct pointers between components.

For implementing entities in the Entity Component System Architecture a simpler solution was chosen: plain integers. Components found at the same index across all arrays belong to the same entity, as can be seen in figure 3 with the "Entity at index i". Any system can associate multiple components by accessing different component-arrays at the same index. This approach allows for any system to access only the data it requires by selecting which arrays to iterate over.

This simplicity comes at the cost of unsolved problems:

1. How can indices be re-used after entities are destroyed?
2. How can entities have different aspects when every component-array has an index for every entity?
3. All component-arrays must have the same length (index range).
4. Therefore, adding a new component type would grow the matrix shown in figure 3 by an entire row, even if only a single entity needs that aspect.

Table 4: Problems arising from to the simplicity of parallel arrays.

The first problem in table 4 is equivalent to the question of how an entity can be identified if not by array index alone. Simplicity suggests to use the concepts already available: A type of component with the sole purpose to uniquely identify an entity. Thus it becomes possible to move all components at index i to index j of their respective component-array and still be able to identify them as the same entity by reading the value of the *unique entity id*-component. Section 3.2.4 defines the details and surrounding issues.

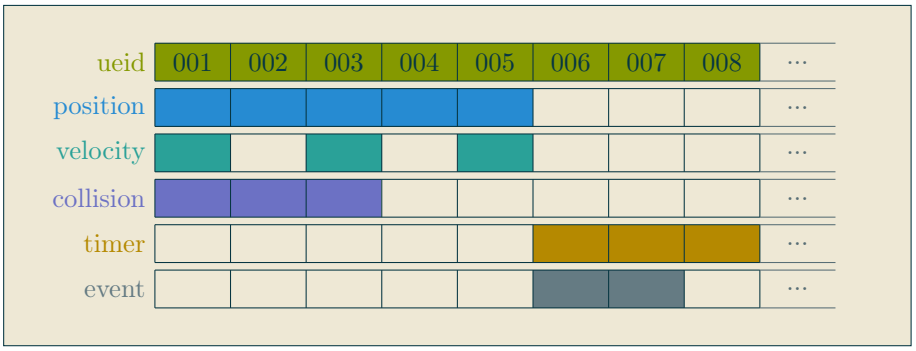

Figure 4: Illustration of component arrays with gaps caused by different types of entities with different aspects.

Solving the second problem is more nuanced than simply adding a new component. The architecture provides multiple, different ways to solve this and

they will be described in detail in section 3.2.5. Until then the quick answer is: component-arrays are *sparse*, as depicted in figure 4.

Problems three and four are different expressions of the same issue that is directly related to problem two. The cost of building relations between data records has not magically vanished, it was traded for the cost of memory. The concept so far could be called a trivial data base addressing scheme. All data of all entities can be found by choosing the right row and column of the entity-component-matrix. But the rows of the matrix are sparse and unless all entities have the same set of component types at least some space will be wasted. An increased number of component types decreases the chances of tightly packed component-arrays. This effect is referred to as *fragmentation* of component-arrays. To mitigate this the entity-component-matrix is *partitioned* into sectors in which entities with similar features are *grouped*. A sector is a logical partition of the simulation world that addresses multiple issues:

- Reducing the length and number of component-arrays.
- Providing meta data for a set of component-arrays.
- Grouping entities.
- Coordinating access to entities.

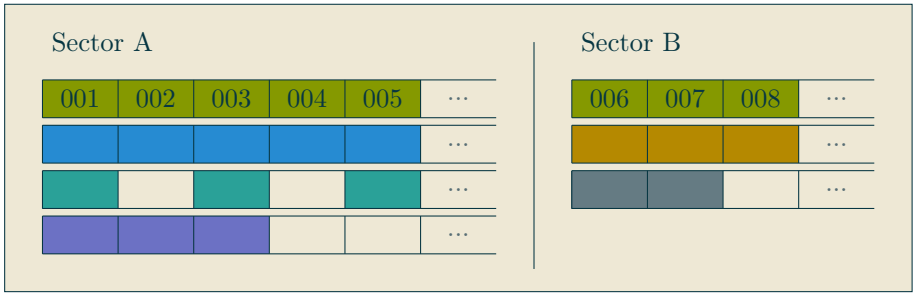

Figure 5: Illustration of how the component arrays shown in figure 4 can be distributed to multiple sectors, resulting in more densely packed arrays. Type names omitted, compare colors.

Compared to figure 4, figure 5 shows how the number of gaps in component-arrays can be reduced by allocating entities to sectors with different sets of component-arrays. In the first image, there are 24 gaps in the arrays combined. The entities with ids 1..5 don't share any component types except the obligatory "ueid" with the entities with ids 6..8. In the second image, the entities with ids 1..5 are in sector A, the others in sector B. Sector A doesn't need arrays for "timer" and "event" components while sector B doesn't need arrays for "position", "velocity" and "collision" components while none of the entities has lost components. There are only 5 unused slots in the component arrays of both sectors combined, 19 less than in figure 4.

The reasons to chose simple arrays as main structure and associating components by integer indices rather than structures with indirections are related to performance. Mainly the discrepancy in cost between fetching data from

memory and processing the data. [10, 1] Reducing the number of cache misses has a substantial impact on performance. [1] Sequential reading of data is vastly more cache efficient than random reads following pointer indirections. [5, p. 24, table 3.1] But the premise for cache efficiency of sequential reading is that the arrays contain the data to be read without other data in btween. Therein lies the reason to split the arrays based on the type of component. A system that needs to apply *acceleration* to *velocity* can access the two arrays containing only components of these types. This adheres to the "principle of locality"[12], a pattern from data oriented design. As mentioned earlier, however, the component-arrays are sparse. Specifically, they contain "gaps" elements that can be ignored when read. This means the component-arrays will on average be less cache-efficient than plain arrays. To determine how much less efficient they are is to find out how many gaps there are. Exact results depend on size of cache-lines, number and size of CPU cache layers, associativity and other details of the hardware. [5, section 3.3] The content of the simulation also imposes limits on the optimizations possible.

3.2 Definitions

In general the definition of the Entity Component System Architecture does not include specific implementation details. Rather, guarantees are given that have to be met and algorithms are described that are to be used by any implementation trying to be compliant with the architecture. The definition tries to make as few assumptions as possible about the programming language being used; the minimum common denominator is that data structures and functions exist.

3.2.1 The Core

It is assumed that there is a central data structure for organizational purpose which will be referred to as the *core* in this definition. The following paragraphs define the required elements of the core structure.

Data A pool of memory that is allocated during initialization and used to store component-arrays and related meta data. The size of this pool is determined by two arguments given to the initialization function of the core:

```
1 int core_init(const sect_t num_sectors, const size_t sector_size_kb);
```

Listing 1: Initialization function of the core

$$k := \underbrace{n * (b_1 + b_2)}_{\text{meta data}} + n * 2^{10} s \tag{3.1}$$

Equation 3.1 defines the number of bytes k that must be allocated for the "Data" memory pool. Where n is the number of sectors, s the sector size in kibibytes [13] given as argument and b_1, b_2 are the sizes of the "sect_t" and "sector" types, which are defined in sections 3.2.3 and 3.2.5. The meta data is located at the beginning of the pool and will be used to describe the layout of sectors: An array for "sect_t" handles and one for "sector" structs, both the length of n. The remaining space of the "Data" pool is used for the data of component-arrays.

[12] https://en.wikipedia.org/wiki/Locality_of_reference
[13] https://en.wikipedia.org/wiki/Kibibyte

Sectors A structure with information about the partitioning of the "Data" memory pool. More precisely it describes the meta data at the beginning of the "Data" pool. It defines the size and count of sectors, which are simply the arguments given to "core_init" seen in listing 1, followed by a counter of currently active sectors and a pointer to the beginning of the "Data" pool where an array is maintained with the handles of all currently active sectors. Lastly, a pointer to the second array of the meta data, containing an array of "sector" structs, is given. How sectors are used is defined in section 3.2.5.

Counters Three integers used to generate "unique entity ids" and handles for "component types" and "entity types". The policy for generating these is to simply increase the counter when a new id or handle is needed.

Maps Five dictionary-like structures used to keep track of handles, types and paths. They may be implemented as trees or hash tables. Implementations

Name	Key	Value	reads	writes
comp_map	comp_t, string	comp_info	rare	none
sect_map	sect_t, string	sector pointer	regular	rare
ent_map	ueid_c	ent_path	frequently	many
comp_deps	comp_t	comp_dep	rare	none
ent_types	string	type_c	frequently	regular

Table 5: Definition of the cores key-value stores. Keys and values are data types defined within this section. Columns "reads" and "writes" are hints about the frequency of read and write operations per frame.

are free to chose any number of maps to record the same set of associations described here.

The **"comp_map"** stores the information about all component types that are known to the core. Entries are only added during the *init phase* and read occasionally when component type handles or names need to be resolved, eg. when a new sector is created or during initialization of systems. There are two keys for each value, the component type handle and the string name given when registering the component type (see section 3.2.3).

The **"sect_map"** is used to accelerate finding sectors based on their handle or name. It's only updated when sectors are created or destroyed, but read every time a sector is opened for access or filtered based on the component types it stores. Like with the comp_map there are two keys for each value, a sector handle and the name of the sector.

The **"ent_map"** keeps track of the sector and array index of every entity in the simulation. It is modified only during the cores maintenance phase (see "frame boundary" in section 3.2.2) but the number of updates can be significant and must complete in constant time. Systems implementing entity hierarchies that need traversal (such as the use case described in 4.5.2) are responsible for most access to the entity map. This access will occur every frame but likely only for a small subset of entities.

"comp_deps" records dependencies between component types. These are required by some types of component that rely on the presence of other types in

the same sector. This map is updated only during the *init phase* and accessed only during the creation of new sectors.

"ent_types" serves both as a cache for computed hashes and as a look-up table for known entity types. See section 3.2.4 for the definition of entity types.

3.2.2 Sequence of Events

Phase	Order	Time frame
Init	1	The initialization phase begins with a call to the cores "init"-function (listing 9) and ends with the first call to its "update"-function (listing 11).
Loop	2	Most of the runtime will be spent in the loop phase. It's entered as soon as the init phase ends and continues as long as the main event loop is running and the cores "update"-function is called.
Close	3	The close phase begins when the loop phase ends and is complete with a call to the cores "close"-function (listing 10).

Table 6: Definition of the three execution phases of the Entity Component System Architecture.

Phases of execution To provide some guarantees about the order of events the runtime is split into the three phases shown in table 6. Implementations are not required to enforce them, they are a guide line to help define the architecture. The main purpose of the phases is to limit the time frame during which certain events can occur. For example, the *registration* of new component types should only be possible during the *init* phase and the *construction* of entities only be completed during the *loop* phase.

Throughout the definition of the Entity Component System Architecture these phases are referenced to define the context of events. The following is a quick summary that is by no means exhaustive:

Init should be used to complete one-time tasks such as runtime configuration or acquiring handles for later use.

Loop is where the simulation transitions between states - one frame at a time - this is the main stage where systems process the data of components to model the behavior of entities.

Close is the clean-up phase in which finalization steps should happen to ensure consistency of written data like logs, saves etc.

Frames are discrete steps along the simulations temporal axis, approximating equidistant steps in real-time. The progress of the simulation is measured in these frames. Every system must update exactly once every frame. There are three time zones within one frame, as defined by table 7 and illustrated by figure 6. To allow frame rate independent simulation the time must be measured at the beginning of each frame and a *delta time* describing the real time duration of the previous frame must be passed to the update function of each system.

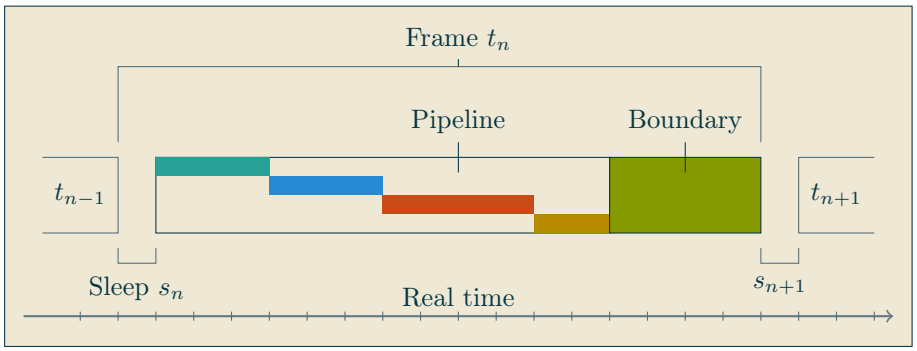

Figure 6: Timeline of a frame with the zones defined in table 7. The different colored blocks in the pipeline represent different systems being run.

Zone	Order	Event
Sleep	1	Buffer to approximate equidistant time steps.
Pipeline	2	Update of all systems except core.
Boundary	3	Update of core.

Table 7: Definition of time zones within a frame.

For the approximation of equidistant steps in real time a frames "Boundary" is followed by the next frames "Sleep" to create an artificial buffer if needed.

3.2.3 Handles and Types

Wherever possible the Entity Component System Architecture tries to replace more complex types with integer handles that are faster to process. To accomplish this, the interface provides several functions for registering information during the *init phase* that is then associated with a handle. For the rest of the runtime this information can then be referred to by this handle. Some of these handles are used as component types such as the "type_c" component described in section 3.2.4. Not all types are handles, notable exceptions are the ent_path and the sector_access, both of which are composite types (or structs).

The most common case for handles are the component types. During the *init phase* systems will call the function shown in listing 2 to register the size of a component struct with an identifier-string and acquire a component handle in return. During the *loop phase* they use this handle to tell the core which component array they want to access.

```
1 comp_t core_register_comp(const_bstring identifier, const uint16_t bytes);
```

Listing 2: Core interface for registering component types

3.2.4 Entities and Components

Components represent aspects of an entity and define all its state. An entity with a *position*-component is an entity that *has a position*. Which component types exist depends on the systems implementing the behavior that is simulated.

Name	Definition	Purpose
enti_t	uint32_t	Type for indices in component-arrays.
sect_t	uint16_t	Handle to uniquely identify sectors.
comp_t	uint16_t	Handle to uniquely identify component types.
type_c	uint16_t	Component to uniquely identify entity types.
ueid_c	uint64_t	Component to uniquely identify an entity.
ent_path	see 3.2.4	Composite type describing path to an entity as sector+index.
sector_access	see 3.2.5	Access token granting read/write privileges to the component-arrays of a sector.

Table 8: Definition of all types required to interface with the Entity Component System Architecture.

If the simulation objects should be able to collide with each other then a collision system needs to be implemented and it will have to define the component types that make an entity *collidable* (eg. bounding boxes).

There are only three component types that are defined by the Entity Component System Architecture itself:

ueid_c to uniquely identify an entity.

flag_c to signal certain states of the entity.

type_c to define the type of an entity.

Every entity has these automatically as they are silently added when an entity is created. Two of them have already been defined in section 3.2.3 as part of the types exposed by the cores interface. The "flag_c" component is defined as an enum with values "ENT_OK", "ENT_REMOVE" and "ENT_CREATE". Its purpose is to trigger a delayed action during the cores update function (see section 3.2.2). These three component types are needed by the core to manage entities, they may not be modified by any other system.

Entities are created with a call to the function shown in listing 19 from section 4.3.4. The creation happens in two steps. First, the three core components shown above are added to the component arrays of the specified sector. For the "ueid_c" component a new id is generated (see 3.2.1), "flag_c" is set to "ENT_CREATE" and "type_c" is given by the parameter. At this point the entity exists in the component-arrays but can not yet be searched for and will be outside the index range defined by the sectors access token. Only the system that created the entity will know it's there; it knows the index and can initialize the values of other components.

Since component-arrays are sorted between frames (see 3.2.5 and 3.2.2) the entity may be moved to a different index. To find it again the value of the "ueid_c" component must be remembered. The function shown in listing 22 will return the sector and array index of an entity with a given "ueid_c". It's utilizing the "ent_map" defined in 3.2.1.

The "ueid_c" value of 0 is reserved for the *neutral entity*, which is disregarded by all operations. It's named for being akin to the "neutral element" in algebraic structures.[14]

The entity type denoted by the "type_c" component is just a handle. It is mainly used for defragmenting sectors (see 3.2.5) and encodes which aspects the entity has. Listing 3 defines the signature of the function.

```
type_c core_get_ent_type(const comp_t *const array);
```

Listing 3: Core interface to computing an entity type handle based on the entities aspects

Internally, the given array of component type handles is sorted (they are integers) and used as key (interpreted as string) for the "ent_types" map defined in 3.2.1. If no entry for this key exists, a new handle is generated from the cores entity types counter (see 3.2.1) and added to the map before being returned. Implementations may use a different hashing algorithm to accomplish the same.

3.2.5 Sectors and Arrays

Component-arrays are implemented as a simple data structure "comp_array" (see listing 8) describing an array of component structs. All components in the array are of the same type. The only information stored in the "comp_array" is the type of the components as "comp_info" struct, the length of the array in bytes and a pointer to the beginning of the data. A "comp_info" combines a component type handle ("comp_t") and size in bytes as registered with the function shown in listing 2. The address pointed to is somewhere within the cores "Data" pool that has been allocated during the cores initialization, as described in paragraph "Sectors". Creation, initialization and destruction of component arrays is always managed by the sectors they are embedded in.

In section 3.1 component-arrays were described as sparse. This has no implications for the arrays themselves, only for the interpretation of their content. Elements of component-arrays are defined to be either valid components of the type declared by the component-arrays "comp_info"-member, or *gaps*. Gaps manifest as uninitialized components, which are equal to memory filled with zeroes. For some component types this is equivalent to a valid component, an example of this is described in section 4.5.1 for the movement physics system. The cores "Data" pool is initially allocated as zero-filled memory and sectors are required to overwrite their memory block as such when destroyed. When entities are moved within their sector, all component-arrays at their index are affected - including gaps - to guarantee that components from different entities don't get mixed and that gaps remain gaps.

Sectors Each sector is described by a header located in the meta data region at the beginning of the cores "Data" pool. A header stores the handle and name of the sector together with a flag to indicate actions scheduled for the next update of the core. The number of component-arrays and length thereof is stored in the integers "rows" and "cols", resembling the image of rows and columns in the entity-component matrix of figure 3. There is an array of "comp_array" structs accompanied by a table "comp_lut" that resolves "comp_t" keys to the right index of that array. As with the associative structures of the core (defined

[14] https://en.wikipedia.org/wiki/Neutral_element

in 3.2.1) implementations are free to chose a suitable data structure for this table.

The sector struct also has a mutex and an access token, as defined in paragraph "Access" of this section.

Finally, the sector has an unsigned 64-bit integer defining a byte offset into the cores "Data" pool. The space of the pool after the section reserved for meta data is evenly divided into chunks and each sectors offset denotes the 16-byte-aligned start of its chunk. This layout does not change during runtime, which is why the sector structs are already set up when the core is initialized.

For this reason creating a new sector with the function shown in listing 4 does not involve the initialization of a new sector struct. Instead, the first unused sector in the cores "sectors" struct is chosen (see section 3.2.1). Where "unused" means that the sectors handle is set to 0. For the new sector, a handle is generated and assigned to the sector struct together with the name. The main work when creating a sector is to initialize its component-arrays.

```
1 sect_t core_mksector(bstring name, set *components);
```

Listing 4: Core interface for creating sectors

As listing 4 shows, the only arguments to the creation of a sector are a name and a set of component types. If dependencies for any of the component types in this set were registered (see "comp_deps" in 3.2.1) these are resolved and the set is expanded accordingly. The three core components ("ueid_c", "flag_c" and "type_c") are implicit (as defined in 3.2.4) and will be added to the set if missing. To compute the "cols" (length of component arrays) of the sector, the size of the sector is divided by the size of all component types accumulated. The number of "rows" is simply the size of the set of component types. Now the array of component-arrays and the "comp_lut" can be initialized, one entry for each entry in the set of component types received as argument. The order is not important except for the first three, which are always assigned the core components in the order given in 3.2.4. Finally, the sectors mutex and access token are initialized.

Access is only provided to the data of component-arrays, but not the sector itself. An access token is given out to a caller of the function shown in listing 15. Under the protection of the sectors mutex only one such token is handed out and only after it has been returned to the function shown in listing 16 can it be granted again. Thus, no more than one system can gain read-write access at a time. The access token itself provides all the information required to iterate any component-array of the sector: The indices of the first and last non-neutral entity in the sector as well as the total length of the arrays. To get the pointer to a component-array, the access token has to be passed to the function shown in listing 18.

3.2.6 Core Maintenance

Table 7 of section 3.2.2 defines the "Boundary" of a frame as "Update of core". This means the time spent during a call to the cores update function shown in listing 11. The reason why this can not overlap with the time in which other systems access the sectors and component arrays is that the guarantees listed in table 9 are not upheld while the core does maintenance of it's data structures.

Systems are able to create or destroy entities as defined in section 3.2.4 but this

No.	Guarantee
1	No entities are added or removed during a frame.
2	Entities remain at the same index during a frame.
3	All entities are within the index range reported by a sectors access token.
4	Sectors are not fragmented.

Table 9: Guarantees made about the simulation state while systems can access component data.

process is not finalized until the next call to "core_update". These changes are not reflected by the cores data structures used to find and access components. Even though one system has created new entities, no other system will know about it until the next frame. The same goes for the creation and destruction of sectors.

During its update, the core will iterate over all sector headers (see section 3.2.1) and update the register of active sectors. For all sectors that have a sector handle $\neq 0$ the core will first check if the sectors flag signals scheduled destruction. If so, all entities from the sector are removed from the cores "ent_map", the sectors entries are removed from the cores "sect_map" and the sector header itself is cleared by resetting its "comp_lut", "handle", "name", "flag", "rows", "cols" and "access" members.

Otherwise, the normal procedure for sector maintenance is to first update the cores "ent_map" for any entities that were created or destroyed during the frame and then defragment the sectors component-arrays.

Defragmenting Sectors is the process of sorting the entities based on their "type_c" component while filling any gaps left by recently removed entities. Implementations are free to chose a suitable sorting algorithm that fulfills the criteria listed in table 10. By which metric the entity types are ordered is not

1	No entity gaps	All non-neutral entities (ueid $\neq 0$) must come before the first neutral entity (ueid $= 0$).
2	No type fragmentation	No two entities with equal type may be separated by an entity of different type.
3	Stable sort	If guarantees 1 and 2 are fulfilled, no entities may be moved.

Table 10: Guarantees defined for defragmentation of component-arrays

defined and subject of future research. When entities are moved during the sorting process this needs to be reflected by an update of the cores "ent_map".

Note that the third guarantee in table 10 has an interesting implication for sectors in which no entities were created or destroyed during the frame: Their data won't be changed during the frame "Boundary" at all. The use case presented in section 4.5.4 makes use of this.

3.2.7 Systems

"The purpose of all programs, and all parts of those programs, is to transform data from one form to another."[1]

In the Entity Component System Architecture the transformation of data is carried out by the *systems*. The architecture provides two main ways of accessing the data of simulation objects: *Iteration* of component arrays (defined in section 3.2.5) and searching for entities by *address* (defined in section 3.2.4). For example, a system responsible for updating the count-down timers of all trigger entities would iterate all component-arrays for timer-components and change their data to reflect the new time. This transformation of data is equivalent to the concept of rules being applied to simulation objects as defined in section 1.2 as "interactions". The systems fulfill the role of "rules" here.

Before systems can be run (during the "loop" phase) they will generally need to initialize their state, which has to be done during the "init" phase (see definition of phases in section 3.2.2). If a system needs to perform clean-up work such as writing remaining log entries to a file this must be done during the "close" phase. These three steps make up the required interface for all systems and section 4.4 describes them in detail.

To prevent heisenbugs [15] between systems accessing data of the same entities concurrently, the architecture defines access to be protected by mutual exclusion on a per sector basis (see section 3.2.5). This should not be confused with support for concurrency between systems, it's a safety measure only. Running systems concurrently can lead to deadlocks[16] if any system tries to open two sectors at once. A simulation engine wanting to run systems concurrently would have to resolve dependencies between systems that could try to access the same data in order to prevent such deadlocks.

[15] `https://en.wikipedia.org/wiki/Heisenbug`
[16] `https://en.wikipedia.org/wiki/Deadlock`

4 Implementation

This section first presents an overview of the code base, then introduces some essential data structures and the interface of the implemented library. The basic concepts of implementing systems are introduced before the implementation of several different systems is described. Each of these covers an illustrative use case for working with the Entity Component System Architecture.

4.1 Code Base

The implementation is written in the *C* programming language (standard revision C11). It is organized in two parts: A shared library and surrounding code. The library implements the core part of the Entity Component System Architecture as defined in section 3 and is called the **core library** or **core** for short. The surrounding code builds a simulation engine using this library. Unit tests are included with the library. The decision what is part of the core library is guided by the question which parts will remain the same when implementing different simulations with the architecture. This distinction is simple and clear and resulted in moving all systems that implement entity behavior from the library to the surrounding code. Tables 11 and 12 give a brief overview of the code bases.

Module	sloc	fn	Brief outline
core	656	17	Implementation of the core Entity Component System Architecture.
structs	58	24	External data structures used by the library.
systems	61	4	Light wrapper functions to encapsulate init, update and close functions of systems with error checks.
utils	24	6	Miscellaneous utility functions, eg. to get the time, put a thread to sleep or align pointers.
headers	113	-	Structs, macros and signatures from the headers of all modules combined.
tests	582	26	Unit test for the core interface.
total	912	61	Accumulated stats (not including tests).

Table 11: Overview of the code base implementing the core library. Each row lists the lines of code and number of functions for one module, together with a short description of that module. (SLOC counted with sloccount(1))

4.2 Essential data structures

As discussed in section 3.1, all state and data of a simulation object is represented by its components. Component types are implemented as primitive- or composite data types and are identified by their size in bytes and their associated handle.

Section 3.2.4 mentions three component types that are essential to the functionality of the core: 1. ueid 2. flag and 3. type. It is mandatory for every sector

Module	sloc	Description
main	113	Main loop, see section 4.4.1.
sys_physics	78	Movement physics system (section 4.5.1)
sys_transform	76	Transform hierarchy system (section 4.5.2)
sys_renderer	294	Renderer system (section 4.5.3)
sys_input	189	Input system (section 4.5.4)
sys_collision	97	Collision system (section 4.5.5)
total	1798	Entire code base excluding external libraries.

Table 12: Overview of the surrounding code implementing the simulation engine on-top of the core library. Only the systems described in section 4.4 are listed explicitly. (SLOC counted with sloccount(1), headers and modules are combined)

to have component arrays for these types and they are always in the order of the above enumeration.

```
typedef uint64_t ueid_c;
```

Listing 5: Unique Entity Identifier Component

The Unique Entity Identifier shown in listing 5 is the first component type registered and subsequently always has the handle 1. The value 0 is reserved for the neutral entity described in section 3.2.4.

```
typedef uint16_t type_c;
```

Listing 6: Entity type component

Listing 6 shows the component identifying the type of an entity as defined in section 3.2.4. It's needed for sorting sectors (see section 3.2.5), constructing new entities as well as by some systems that rely on the entity type for building subsets to operate on. Since it is the second component type registered it always has the handle 2.

```
typedef enum {
        ENT_OK=0, ENT_REMOVE, ENT_CREATE
} flag_c;
```

Listing 7: Entity meta flag component

The third essential component type is a flag to mark entities for deferred operation during the next frame boundary. It always has the handle 3. Its value is set by the interface functions for creating and removing entities shown in listing 19 and 20.

```
typedef struct {
        comp_t handle;          // component type id
        uint16_t size;          // component size in bytes
} comp_info;

typedef struct {
        comp_info type;         // type of components in this array
        uint32_t mlen;          // length of memory assigned to this array
        void *start;            // pointer to the first component
} comp_array;
```

Listing 8: Struct for component-arrays

Listing 8 shows that component-arrays are deliberately kept simple with a pointer to the data, the length of the array and information about the type of components stored in the array.

4.3 Interface of the Core

This section will describe the interface of the core library. It provides functions to access entities and components, register data types, provide handles, create and destroy entities and sectors and to initialize and manage the core itself.

4.3.1 Init, Update and Close

```
int core_init(const sect_t num_sectors, const size_t sector_size_kb);
```

Listing 9: Interface for initializing the core library

Upon initialization the core allocates memory, sets up its data structures, creates sectors and distributes the allocated memory between them, as defined by section 3.2.1.

```
int core_close();
```

Listing 10: Interface for closing the core library

Its complement is the close function shown in listing 10. While it does free the main chunk of memory, it does not clean-up any of the data structures unless built in debug-mode. The core itself does not do any work that needs to be completed before shutdown and all remaining memory will be collected by the operating systems kernel anyway. Things like writing remaining data to open files are handled by the systems responsible for them during their own "close"-function. When compiled in debug mode however, the core will do the clean-up of all its structures which was verified with valgrind [17] during development.

```
int core_update(const double dt)
{
        if (!CORE.initialized) return ECSA_NOT_INITIALIZED;

        CORE.sectors.active = 0;
        for (sect_t i = 0; i < CORE.sectors.count; i++) {
                sector *s = &CORE.sectors.array[i];
                if (!s->handle) continue;
                if (s->flag == SEC_REMOVE) {
                        update_entmap(s);
                        clear_sector(s);
                        continue;
                }
                const sector_access *const sa = core_open_sector(s->handle);
                update_entmap(s);
                defrag_sector(s);
                core_close_sector(sa);
                CORE.sectors.idx[CORE.sectors.active++] = s->handle;
        }

        return ECSA_SUCCESS;
}
```

Listing 11: Update function of the core library

[17] http://www.valgrind.org/

As defined in section 3.2.6 the cores update function shown in listing 11 does maintenance of internal data structures. The loop from lines 6 to 19 marches over all sectors. First it clears all sectors that are scheduled for removal in lines 9 to 12. For the remaining sectors, the creation and destruction of entities is handled in line 15, followed by sorting the component arrays in line 16. Internal data structures such as the "ent_map" defined in section 3.2.1 are updated to reflect the new state of the sector.

4.3.2 Registering Components

The core library is agnostic towards different types of simulation and tries to be minimal. It only implements the component types defined in section 3.2.4 for internal operations. All other component types need to be registered with the core during the *init* phase (see table 6). During their initialization all systems are expected to register all component types they need to access later.

```
1  comp_t core_register_comp(const_bstring id, const uint16_t bytes)
2  {
3          if (!CORE.initialized) return ECSA_NOT_INITIALIZED;
4
5          // If the identifier is already registered, check if the size matches.
6          // A missmatch is an error because overwriting isn't allowed.
7          void *ptr = tree_search(&CORE.comp_map, id->data, id->slen);
8          if (anycast(ptr, comp_info).handle) {
9                  if (anycast(ptr, comp_info).size == bytes)
10                         return anycast(ptr, comp_info).handle;
11                 else
12                         return ECSA_ERROR;
13         }
14
15         // Get a new handle and add an entry to the registry.
16         comp_info i = {atomic_fetch_add(&CORE.comp_handle_counter, 1), bytes};
17
18         // The entry is added twice, using two different keys: The received
19         // string as well as the handle.
20         ptr = anycast(i, void*);
21         tree_insert(&CORE.comp_map, id->data, id->slen, ptr);
22         tree_insert(&CORE.comp_map, (uint8_t *)&i.handle, sizeof(comp_t), ptr);
23         return i.handle;
24 }
```

Listing 12: Interface for registering component types

Listing 12 shows the implementation of registering component types defined in section 3.2.3. The second argument is the size of the component type struct, which is used internally for memory management and pointer alignment by the core when constructing component-arrays. For the same combination of identifier string and size, the same component handle will be returned without performing another registration. An identifier string cannot be re-used with a different size, the association of a component type handle is fixed for the entire runtime.

```
1  comp_t core_get_comp_handle(const_bstring id)
2  {
3          void *p = get_comp_info(id);
4          comp_info inf = anycast(p, comp_info);
5          return inf.handle ? inf.handle : ECSA_ERROR;
6  }
```

Listing 13: Interface for acquiring component type handles

There is also a function to query the component type handle for a given identifier string without performing a registration (listing 13). It can be used when certain that a component type has already been registered.

4.3.3 Accessing Components

The main access pattern is to iterate over the component arrays of all sectors that contain a specified subset of component types. The following function of the core interface returns a list of sectors that contain all of the specified component types. To avoid complex data structures and those that require dynamically allocated memory, a simple zero-terminated array is used as argument.

```
sect_t *core_get_sectors(const comp_t *const c)
{
        size_t n = 0;
        sect_t *handles = calloc(CORE.sectors.count + 1, sizeof(sect_t));
        for (sect_t i = 0; i < CORE.sectors.active; i++) {
                const sect_t handle = CORE.sectors.idx[i];
                if (sector_has(handle, c)) handles[n++] = handle;
        }
        return handles;
}
```

Listing 14: Interface for retrieving a filtered list of sectors

The next step is to iterate the returned list of sector handles and use them to *open* the associated sectors.

```
const sector_access *const core_open_sector(const sect_t s)
{
        sector *sec;
        if (!(sec = get_sector(s))) return NULL;
        pthread_mutex_lock(&sec->rw);
        return &sec->access;
}
```

Listing 15: Interface for opening a sector with read-write access

Listing 15 and 16 show how sectors are opened and closed as defined in section 3.2.5. The sector access token shown in listing 17 is granted when opening a sector and is returned when closing it to yield exclusive access to the sector.

```
int core_close_sector(const sector_access *const sa)
{
        if (!sa || !(sa->s)) return ECSA_ERROR;
        sector *s;
        if (!(s = get_sector(sa->handle))) return ECSA_ERROR;
        pthread_mutex_unlock(&s->rw);
        return ECSA_SUCCESS;
}
```

Listing 16: Interface for returning a sectors access token

```
typedef struct {
        const void *const s;
        const sect_t handle;
        const enti_t length;
        enti_t first;           // index of first ent. (as of last core_update)
        enti_t last;            // index of last ent. (as of last core_update)
} sector_access;
```

Listing 17: Struct for handling restricted access to a sector

To access the component arrays, the token must be given to the function shown in listing 18.

```
void *core_access_array(const sector_access *const sa, const comp_t t)
{
        const sector *const s = sa->s;
        map_entry *e = map_search(s->comp_lut, voidcast(t));
        return e ? ((comp_array *)e->data)->start : NULL;
}
```

Listing 18: Interface for accessing component arrays

4.3.4 Working with Entities

While the main access pattern is to iterate over component arrays, it is also possible to search for a specific entity across all sectors. This access pattern becomes important when dealing with references between entities such as the hierarchical transformations demonstrated in section 4.5.2. Besides searching, the only other direct interactions with entities are the constructor and destructor functions shown in listing 19 and 20.

```
enti_t core_mkent(const sector_access *const sa, const uint32_t n, type_c type)
{
        // We don't need to test if n < sector.cols, the below algorithm
        // accounts for that and the corner case should not slow down the
        // common case with additional branching.
        const sector *const s = sa->s;
        ueid_c *c = core_access_array(sa, CORE.handle.ueid);
        flag_c *f = core_access_array(sa, CORE.handle.flag);
        type_c *t = core_access_array(sa, CORE.handle.type);

        // Find the first empty entity slot that is followed by n free slots
        enti_t i = 1;
find:
        while (c[i] && i < s->cols - n) i++;
        if (i >= s->cols - n) return ECSA_ERROR;        // == 0 aka invalid
        enti_t streak = 1;
        while (!c[i + streak] && streak < n) streak++;
        if (streak != n) goto find;

        // Create new entities by setting the ueid component
        for (enti_t j = i; j < i + n; j++) {
                c[j] = atomic_fetch_add(&CORE.ueid_counter, 1);
                f[j] = ENT_CREATE;
                t[j] = type;
        }

        return i;
}
```

Listing 19: Interface for constructing a new entity (error handling omitted)

Listing 19 shows the implementation of the function for entity construction. It schedules construction of n entities of the type given by handle t in the sector referenced by the received *sector_access* struct. No memory allocation or other system calls are involved in the construction. The algorithm to find n consecutive free entity slots (lines 15 to 21) only tests values in the component array of entity ids. Unless entities have been removed this is equal to an append operation. Lines 24 to 28 show that the construction of entities is limited to setting the three first components: *ueid*, *flag* and *type*. Note that the ueid is

retrieved with an atomic fetch-add operation, allowing multiple threads to call the constructor concurrently. The returned value indicates the index of the first constructed entity such that the caller can already set values to the components even though the entity is not yet visible to the search or included in the index range of the sectors access token that is used to iterate arrays. These changes are committed during the next call to *core_ update*.

```
enti_t core_rment(const sector_access *const sa, enti_t i, enti_t j)
{
#ifdef DEBUG
        if (!sa || !sa->s) return ECSA_ERROR;
#endif
        if (j >= sa->length) return ECSA_ERROR;

        flag_c *f = core_access_array(sa, CORE.handle.flag);
        enti_t e;
        for (e = i; e < j; e++) {
                f[e] = ENT_REMOVE;
        }
        return e - i;
}
```

Listing 20: Interface for destroying an entity

The implementation of the entity destructor function shown in listing 20 operates similar to the constructor in that it only modifies the meta data. Entities in the range $[i, j[$ are marked for removal during the next *core_ update*. They can still be found by *core_ search_ entity* and will be seen when iterating the component arrays based on the sectors access boundaries until the end of the current frame. The destructor is much simpler than the constructor since only the *flag* components are set and no search for indices is required. Note that since no other component types are touched, there is no automatic clean-up of data. This operation is meant to be as fast as possible and the caller is responsible for performing clean-ups should they be necessary. A guideline of the Entity Component System Architecture is that components should include all the data while also being light. Larger data such as loaded textures or meshes are only referenced by handle within the components. In this case the systems loading such data are responsible for tracking the handles (or pointers) and cleaning up this data after calling *core_ rment* themselves or by periodically searching for entities they are keeping track of and reacting when they can no longer be found. Since the entities can still be seen by other systems until the end of frame, care must be taken when cleaning up data.

```
/* Location of an entity in terms of sector handle and array index. */
typedef struct {
        sect_t sec;
        enti_t idx;
} ent_path;
```

Listing 21: Implementation of an entity path as returned by the search operation

```
ent_path core_search_entity(const ueid_c id)
{
        uint8_t *key = (uint8_t *)&id;
        void *p = tree_search(&CORE.ent_map, key, sizeof(ueid_c));
        ent_path loc = anycast(p, ent_path);
        return loc;
}
```

Listing 22: Interface for locating an entity based on ueid

Part	Lst.	Brief description
System state	23	Static struct for storing current state of the system as well as caching data such as handles to avoid repeated work.
Init function	24	Sets up the systems state and does all groundwork such as registering components, starting threads or creating sectors. Called during engines *init* phase.
Update function	25	Central part of each system. Updates state of system, iterates entities, works on components. Called once per frame during *loop* phase.
Close function	26	De-initializes state and does clean-up work such as freeing memory, closing file descriptors or joining threads. Called once during *close* phase.

Table 13: Basic structure of a system in the Entity Component System Architecture, see table 6 for information about the execution phases referenced here.

Lastly, entities can be searched based on their *ueid* component. As shown in listing 22 this is merely a look-up which must complete in constant time according to the definition in section 3.2.1. Since it is guaranteed by table 9 that entities are not moved during a frame, it is safe for multiple systems to query the search concurrently and use the result for the remainder of the current frame.

4.4 Implementing Systems

All systems of the Entity Component System Architecture have essentially the same structure. Table 13 presents a brief overview before each part is described in more detail below.

```
static struct {
        int initialized;
        struct { comp_t pos, ueid; } handle;
        /* ... */
} SYS;          // name of system struct, varies between systems
```

Listing 23: Prototype system state

The purpose of the state struct is to keep track of information between calls to the various functions as well as to cache results and avoid repeated work. It is static to make it local to file scope and to place it in statically allocated memory, which is initialized to 0 at compile time. Thus, testing whether a system has been properly initialized is as simple as checking the value of *SYS.initialized*, which is guaranteed to be 0 until explicitly changed. This happens at the end of the initialization function to signal that all relevant steps of the initialization have completed successfully.

```
int sys_init()          // prefix of function name varies between systems
{
        if (SYS.initialized) return ECSA_ALREADY_INITIALIZED;

        comp_t h = core_register_comp("pos", sizeof(pos_c));
        if (h == (comp_t)ECSA_ERROR) return ECSA_ERROR;
        SYS.handle.pos = h;
```

```
        SYS.handle.ueid = core_register_comp("ueid", sizeof(ueid_c));

        SYS.initialized = 1;
        return ECSA_SUCCESS;
}
```

Listing 24: Prototype system init function

The init function is called during the *init* phase. In it, the state of the system is set up, component types are registered and handles are retrieved. Some systems will perform more specific set ups such as creating a sector, initializing data structures or starting threads.

```
int sys_update(const double dt)
{
        if (!SYS.initialized) return ECSA_NOT_INITIALIZED;

        comp_t components[] = {SYS.handle.ueid, SYS.handle.pos, 0};
        sect_t *s = core_get_sectors(components);
        sect_t it;
        for (int i = 0; (it = s[i]); i++) {
                const sector_access *const sa = core_open_sector(it);
                ueid_c *u = core_access_array(sa, SYS.handle.ueid);
                pos_c *p = core_access_array(sa, SYS.handle.pos);
                for (enti_t j = 1; j <= sa->count; j++) {
                        /* Do something with ueid and pos components */
                }
                core_close_sector(sa);
        }
        free(s);
        return ECSA_SUCCESS;
}
```

Listing 25: Prototype system update function

The update function is called once per frame during the *loop* phase and receives the delta time since the last frame as argument. This is where access to the simulation objects happens. Usually, all sectors with a specified set of component types are opened and their component arrays iterated.

```
int sys_close()
{
        if (!SYS.initialized) return ECSA_NOT_INITIALIZED;
        /* Free memory, close file descriptors, unlock mutexes, etc */
        SYS.initialized = 0;
        return ECSA_SUCCESS;
}
```

Listing 26: Prototype system close function

Finally, the close function is called during *close* phase and covers finalizing steps, if there are any. Some systems might need to free data, others close files or flush streams.

4.4.1 Running Systems

How the systems are called is not actually part of the core library but the implementation built around it.

```
int QUIT;
void fn_quit(int pressed) { QUIT = 1; }

int main(const int argc, const char **const argv)
```

```
5  {
6          if (core_init(32, 2048) != ECSA_SUCCESS) return 1;    // INIT PHASE
7          if (!sys_init(input_init, "input")) return 1;
8
9          QUIT = 0;
10         input_register_fn("quit", fn_quit);
11         input_map('Q', 0, "quit");
12
13         double dt = 0.0, fps = 60.0;
14         core_update(dt);                                       // LOOP PHASE
15         double tlast = util_get_time();
16         for (;;) {
17                 sys_reset_frame();
18                 double now = util_get_time(); dt = now - tlast; tlast = now;
19
20                 sys_run(input_update, dt, "input");          // frame:pipeline
21                 sys_run(core_update, dt, "core");            // frame:boundary
22                 if (QUIT) break;
23                                                              // frame:sleep
24                 double elapsed = util_get_time() - tlast, desired = 1.0 / fps;
25                 if (elapsed < desired) {
26                         util_musleep((uint32_t)((desired - elapsed) * 1E6));
27                 }
28         }
29         sys_close(input_close, "input");                      // CLOSE PHASE
30         if (core_close() != ECSA_SUCCESS) return 1;
31         return 0;
32 }
```

Listing 27: Example main loop (includes omitted)

Listing 27 shows a minimal example to put the previous description of systems into context. It starts a simulation and waits for the press of a button to quit. The only system running besides the core is the input system documented in section 4.5.4.

Lines 1 and 2 show the definition of a global variable "QUIT" and a callback function to change its state. This callback is registered with the input system in line 10; a key mapping for that function follows in the next line and in line 22 the global variable is checked to see if its state was set and the loop should be exited. As described in section 4.5.4 the input system can record incoming input events asynchronously, but will handle the execution of callbacks in order when its update function is called (line 20).

But before any of that can happen, the core must be initialized as shown in line 6. Initializing the core marks the beginning of the *init phase* during which other systems may be initialized. This phase ends when "core_update" is called for the first time in line 14 and the *loop phase* begins. Behind the scenes, the input system has created a sector and requested the creation of entities during its "init"-function, as described in in section 4.5.4 and shown in listing 42. Section 3.2.6 defines that the creation of entities is only completed for the next frame.

To make the example code more intuitive to read, the loop from line 16 to 28 represents the order of a frame defined in table 7, which required moving the first call to "core_update" outside the loop.

In accordance with the definition of the frame zones in table 7, the loop from line 16 to 28 runs the "core_update" after all other systems (eg. the input system) and before the sleep. In an example with more systems, their update would be run between the input system and the core. The beginning of each

execution phase and frame zone is marked with comments in the respective lines.

4.5 Prototype Use Cases

To demonstrate concrete usage of the Entity Component System Architecture a few prototype use cases have been implemented and will be described in the following section. The focus of these use cases is not to implement sophisticated simulation techniques but to showcase the architecture in operation. As such all systems described here are greatly simplified. Table 14 briefly compares the use cases.

Use case	Access	Specific trait
Movement physics	producer & consumer	Presents a simple base case for iterating component arrays.
Transform hierarchy	producer & consumer	Demonstrates how entities can reference other entities to build hierarchic structures.
Graphics rendering	consumer	Constructs in internal representation of data read from component-arrays.
User input	producer	Receives input from asynchronous callbacks, delays execution of events.
Collision system	producer & consumer	Implements interaction between entities and filters for different subsets of components.

Table 14: Use cases with their access patterns and specific traits.

4.5.1 Movement physics

A concise and simple algorithm was chosen for computing the movement physics. It is complete enough to produce fluid movement based on user input but still simple enough to be described by the following formula:

$$\vec{v'} = \vec{v} + (\vec{a} * t), \tag{4.1a}$$

$$r = \begin{cases} \frac{m}{|\vec{v'}|} & \text{if } |\vec{v'}| > m \\ 1 & \text{otherwise} \end{cases}, \tag{4.1b}$$

$$\vec{v''} = \vec{v'} * r * (1 - \min(f * t, 1)), \tag{4.1c}$$

$$\vec{p'} = \vec{p} + \vec{v''} \tag{4.1d}$$

Where $\vec{a}, \vec{v}, \vec{p} \in \mathbb{R}^3$ are the acceleration, velocity and position, $t \in \mathbb{R} : t > 0$ is the delta time (since the last frame), $f \in \mathbb{R} : f \geq 1$ the base friction and $m \in \mathbb{R} : m \geq 1$ the maximum speed. The resulting $\vec{p'} \in \mathbb{R}^3$ is the new position.

The focus is not on the capabilities of this simulation but on demonstrating the usage of the Entity Component System Architecture. The same structure could be used to implement more complex physics simulations such as the impulse based physics models seen in the movement mechanics of many games. Similarly, ignoring rotation allows for a straight forward implementation based on $\mathbb{R}^3$-vectors that is easy to comprehend at a glance and can even be verified by hand.

```
static struct {
        int initialized;

        // Some hard-coded parameters that would emerge from components in a
        // more complex simulation. Here, they are just placeholders to satisfy
        // the inputs of the applied algorithm.
        float max_speed;
        float base_fric;

        struct { comp_t vel, pos, acc; } handle;
} P;
```

Listing 28: State struct of movement physics system

Listing 28 shows the state struct of the movement physics system. The *initialized* flag and the anonymous struct holding component handles is a repeated pattern throughout all systems. In contrast, the variables *max_speed* and *base_fric* are contrived examples. They are used here to show that any kind of information could be part of the state struct to influence the execution of the systems update function with values initialized by the systems initialization function.

Listing 29 shows the initialization function of the movement physics system. The component types are registered with the core and the returned handles are cached in the state struct. Once complete, the initialization flag is set, allowing other functions of this system to proceed when called. Error handling has been omitted for clarity.

```
int physics_init()
{
        if (P.initialized) return ECSA_ALREADY_INITIALIZED;

        struct tagbstring vel_s = bsStatic("vel"), pos_s = bsStatic("pos"),
                          acc_s = bsStatic("accel");
        P.handle.vel = core_register_comp(&vel_s, sizeof(vel_c));
        P.handle.pos = core_register_comp(&pos_s, sizeof(pos_c));
        P.handle.acc = core_register_comp(&acc_s, sizeof(acc_c));

        P.max_speed = 64.f;
        P.base_fric = 0.96f;

        P.initialized = 1;
        return ECSA_SUCCESS;
}
```

Listing 29: Init function of physics movement system

The update function seen in listing 30 resembles the structure of the prototype example in listing 25. It deviates only in the content of the inner loop, which implements the algorithm described in equation 4.1, as well as the set of component types used. The latter being dictated by the former. Inputs to the algorithm are supplied from three sources:

Delta time is given as argument to the update function.

Friction and maximum speed are defined in the systems state struct (listing 28) and were set during initialization (listing 29).

Position, velocity and acceleration are components.

Delta time, friction and maximum speed don't change during execution of the update function while the components differ for each entity. At the same time, the set of component types defines the filter for matching sectors. Line 5 and 6 of listing 30 show how this is accomplished using the function shown in listing 14. A list of all sectors containing the required set of component types is returned. It is not guaranteed that any of these sectors have entities with all of the required components at every frame. The returned sectors merely have component arrays for the required set of component types. A newly created sector may be empty or all components of a certain type might have been removed from the sector during the previous frame. The set of valid entities is a subset of the set of entities within the matched sectors. Further filtering may be required and the approach to this varies between different systems.

The movement physics system presents a simple and elegant case: The uninitialized state of all relevant component types is equal to the neutral element of the computation performed. In other words: Processing has no effect on uninitialized components. For example, an entity without an acceleration and velocity component will have zero-initialized data at its index of the component arrays for these component types. This is equivalent to the neutral element of addition for both acceleration and velocity components: $(0, 0, 0) \in \mathbb{R}^3$. Using equation 4.1 it can be verified that the position component will not be modified by the calculation in this case.

$$\begin{aligned} \vec{a} = \vec{v} = 0 \in \mathbb{R}^3 &\implies \vec{v'} = 0 \\ &\implies \vec{v''} = 0 \\ &\implies \vec{p'} = \vec{p} \end{aligned} \tag{4.2}$$

Neither will the velocity be modified if the entity has no acceleration component (eg. acceleration is uninitialized):

$$\vec{a} = 0 \in \mathbb{R}^3 \implies \vec{v'} = \vec{v} \tag{4.3}$$

An entity with acceleration and position components but no velocity will spontaneously gain a velocity component as its uninitialized state is identical to a velocity of $0 \in \mathbb{R}^3$, which in turn is valid input to equation 4.1:

$$\vec{v} = 0 \in \mathbb{R}^3 \implies \vec{v'} = \vec{a} * t \tag{4.4}$$

It should be pointed out that it is semantically questionable for an entity to have an acceleration but no velocity. Due to filtering the list of sectors it is guaranteed that the component array for velocity components is present in all sectors processed by the system. Sectors with acceleration components but no velocity components will simply be ignored. However, the core also provides a facility to guarantee the presence of a velocity component array for all sectors with acceleration components by registering a component dependency (see section 3.2.1, "Maps").

Another method of dealing with gaps in component arrays is shown by the renderer system in section 4.5.3, listing 39.

```
1  int physics_update(const double dt)
2  {
3          if (!P.initialized) return ECSA_NOT_INITIALIZED;
4
5          const comp_t comps[] = {P.handle.vel, P.handle.pos, P.handle.acc, 0};
6          sect_t *sectors = core_get_sectors(comps);
7          sect_t it;
8          for (int j = 0; (it = sectors[j]); j++) {
9                  const sector_access *const sa = core_open_sector(it);
10                 pos_c *p = core_access_array(sa, P.handle.pos);
11                 vel_c *v = core_access_array(sa, P.handle.vel);
12                 acc_c *a = core_access_array(sa, P.handle.acc);
13
14                 for (enti_t i = sa->first; i <= sa->last; i++) {
15                         // Apply acceleration to velocity
16                         vec3 accel;
17                         glm_vec3_scale(a[i], dt, accel);
18                         glm_vec3_add(v[i], accel, v[i]);
19                         const float speed = glm_vec3_norm(v[i]);
20                         if (speed > P.max_speed) {
21                                 const float factor = P.max_speed / speed;
22                                 glm_vec3_scale(v[i], factor, v[i]);
23                         }
24
25                         // Apply friction to velocity
26                         const float fric = fabsf(P.base_fric);
27                         const float m = 1.f - fminf(dt * fric, 1.f);
28                         glm_vec3_scale(v[i], m, v[i]);
29
30                         // Apply velocity to position
31                         glm_vec3_add(p[i], v[i], p[i]);
32                 }
33
34                 core_close_sector(sa);
35         }
36         free(sectors);
37         return ECSA_SUCCESS;
38 }
```

Listing 30: Update function of physics movement system

The outer loop of the update function iterates the list of sectors. Each is opened for read-write access (line 9) and access to the component arrays is gained (lines 10-12). In the inner loop all entities within the sector are iterated and the change of position is computed based on the algorithm from equation 4.1. The acceleration components are purely read while the velocity and position components are read and written. Once the inner loop has completed, the sector is closed again to yield exclusive access (line 34) before opening another sector in the next iteration of the outer loop.

Finally, the close function resets the *initialized*-flag of the state.

```
1  int physics_close()
2  {
3          if (P.initialized) P.initialized = 0;
4          return ECSA_SUCCESS;
5  }
```

Listing 31: Close function of physics movement system

4.5.2 Transform hierarchy

This system was chosen to demonstrate how hierarchies can be established between entities. Just like the system for movement physics described in section 4.5.1, the system for transform hierarchies iterates component arrays to read and write data. However, this system additionally uses a second method of accessing components: Searching based on unique entity identifiers (see listing 22).

```
static struct {
        int initialized;

        struct { comp_t ueid, pos, par, off; } handle;
} X;
```

Listing 32: State struct of transform system

The state of the transform system shown in listing 32 has no new concepts. The new component types represent spatial offset (listing 33) to the position of another entity and a reference to another entity as a parent (listing 34).

```
typedef vec3 off_c;
```

Listing 33: Position offset component

```
typedef ueid_c par_c;
```

Listing 34: Parent entity component

The initialization- and close-functions of the transform system don't introduce anything new and are omitted here.

```
int xform_update(const double dt)
{
        comp_t comps[] = {X.handle.pos, X.handle.par, X.handle.off, 0};
        sect_t *sectors = core_get_sectors(comps);
        sect_t s;
        for (int i = 0; (s = sectors[i]); i++) {
                const sector_access *const sa = core_open_sector(s);
                pos_c *pos = core_access_array(sa, X.handle.pos);
                par_c *par = core_access_array(sa, X.handle.par);
                off_c *off = core_access_array(sa, X.handle.off);
                for (enti_t j = sa->first; j <= sa->last; j++) {
                        par_c parent = par[j];
                        if (!parent) continue;

                        ent_path path = core_search_entity(parent);
                        pos_c ppos = GLM_VEC3_ZERO_INIT;
                        if (path.sec == s) {
                                glm_vec3_copy(pos[path.idx], ppos);
                        } else {
                                parent_pos_copy(path, ppos);
                        }
                        glm_vec3_add(ppos, off[j], pos[j]);
                }
                core_close_sector(sa);
        }
        free(sectors);
        return ECSA_SUCCESS;
}
```

Listing 35: Update function of the transform system

For the most part this is a pattern previously shown. Line 13 shows usage of the new component type *par_ c*. The parent component holds the id of another

entity to signal a parent-child relationship. The uninitialized form equals the neutral entity and is interpreted as having no parent. Otherwise the parent entity is searched as defined in section 3.2.4 using the function shown in listing 22 (line 15). The returned path is used to acquire the parents position component. Here, a distinction is made based on the sector found in the path. In the simple case, the parent entity is located in the same sector already opened, otherwise the sector of the parent entity must first be opened and access to its component arrays must be acquired. The latter case is wrapped in an extra function to avoid cluttering up the main update function. Finally, the offset is added to the parents position to compute the new position (line 24).

Just like the movement physics described in section 4.5.1 the transform hierarchy system can treat all entities the same, whether they define a parent and offset or not.

In this implementation only one parent is resolved per child per frame. This means a hierarchy with a depth of 2 will be resolved over the course of two frames. This is intentional as the implemented demonstration is meant to have a delay, but other hierarchical systems may have different requirements. If transform hierarchies are meant to be resolved without delay, the system could easily be changed to not resolve one layer of parents each update, but instead build a tree modeling the hierarchy. Section 4.5.3 describes a system that builds its own ordered representation of data while iterating the component arrays.

4.5.3 Graphics rendering

Like the other systems presented here, the renderer is kept relatively simple. It features only one shader program, no material system and no optimization to speak of. Due to the nature of the OpenGL API the renderer is still much more complex than any other system. This section will therefore not explain the entire renderer system but only those parts relevant to this use case of the core library. Specifically, how the renderer constructs an internal representation of the data it gathers while iterating component arrays.

Two component types are defined by the renderer system to represent meshes (listing 36) and materials (listing 37).

```
1 typedef struct {
2         GLuint vao;
3         uint32_t elem_count;
4         uint32_t elem_type;
5 } mesh_c;
6 #define MESH_INIT {GL_INVALID_VALUE, 0, 0}
```

Listing 36: Mesh component type

```
1 typedef struct {
2         vec4 color;
3         GLuint shd;
4         GLuint dif;
5 } mat_c;
6 #define MAT_INIT {{1, 1, 1, 1}, GL_INVALID_VALUE, GL_INVALID_VALUE}
```

Listing 37: Material component type

Internally, the renderer system defines another struct to combine information gathered from multiple different component types. This *drawable* struct shown in listing 38 stores all members of the *mesh_c* component as well as the color

from the *mat_ c* component and a model matrix derived from position- and scale components.

```
typedef struct {
        GLuint vao;
        uint32_t elem_count, elem_type;
        mat4 model;
        vec4 color;
} drawable;
```

Listing 38: Renderer internal drawable struct

Note that at least two pieces are missing for drawing objects: The shader- and texture handle from the material component. These are not included in the drawable struct but encoded in the internal representation of the scene that is built during the update function shown in listing 39.

```
int r_update(const double dt)
{
        tree_destroy(&R.scene);         // Reset scene
        tree_init(&R.scene);
        R.drawables.len = 0;

        comp_t components[] = {
                R.handle.mesh, R.handle.mat, R.handle.pos, R.handle.scale, 0
        };
        sect_t *s = core_get_sectors(components);
        sect_t *it = s;
        while (*it) {
                /* [...] */
                mesh_c *m = core_access_array(sa, R.handle.mesh);
                mat_c *a = core_access_array(sa, R.handle.mat);
                pos_c *p = core_access_array(sa, R.handle.pos);
                scale_c *z = core_access_array(sa, R.handle.scale);
                for (enti_t i = sa->first; i <= sa->last; i++) {
                        if (!m[i].vao || m[i].vao == GL_INVALID_VALUE)
                                continue;
                        // shader/diffuse/sector/entity
                        uint8_t key[14];
                        *((GLuint *)&key[0]) = a[i].shd;
                        *((GLuint *)&key[4]) = a[i].dif;
                        *((sect_t *)&key[8]) = *it;
                        *((enti_t *)&key[10]) = i;
                        drawable *d = &(R.drawables.data[R.drawables.len++]);
                        d->vao = m[i].vao;
                        d->elem_count = m[i].elem_count;
                        d->elem_type = m[i].elem_type;
                        glm_mat4_identity(d->model);
                        glm_translate(d->model, p[i]);
                        glm_scale(d->model, z[i]);
                        glm_vec4_copy(a[i].color, d->color);
                        tree_insert(&R.scene, (uint8_t *)&key, 14, d);
                }
                core_close_sector(sa);
                it++;
        }
        free(s);
        return ECSA_SUCCESS;
}
```

Listing 39: Update function of the renderer system (several parts omitted)

The inner loop in lines 18 to 35 of presents three differences to previously shown systems:

1. Entities without a mesh component are detected and skipped.

2. Data is read from component arrays but no data is written to component arrays, making the renderer system a pure consumer.

3. An alternate representation of the data is constructed and kept internally for later processing.

The first point is illustrated in lines 19 and 20. Unlike the movement physics system described in section 4.5.1 the renderer system can not treat all entities as valid candidates. The array of mesh components is tested for specific values that signal either uninitialized data or data explicitly marked as invalid for rendering and skips these to avoid faulty draw calls.

The component arrays are treated as read-only because the renderer is not supposed to change the simulation, merely present the current state. To do that the renderer gathers all information from the components necessary to construct a draw call and collects it in a tree. This tree is sorted to minimize the number of context switches between draw calls. Lines 22 to 26 show the method of sorting: Handles and indices are concatenated to form a path in which the shader is closest to the root and thus least frequently changed when iterating the tree. The leaves of the tree then point to the drawable structs constructed from the component data in lines 27 to 34. As previously mentioned, these drawable structs (listing 38) are missing shader- and texture handles needed by draw calls. Both are encoded in the first 8 bytes of the path to the leaves (lines 32, 33). How this tree is iterated to issue draw calls has little relevance to this use case and is omitted.

4.5.4 User input

Table 14 lists the input system with the access pattern "*producer*", suggesting it writes without ever reading data from component arrays. This is only true conceptually from the perspective of information exchange between systems. In practice the input system needs to keep track of input received asynchronously through its callback function until the next call to its update function. To do so it creates its own sector for buffering input events as components. Once per frame it will iterate over these components and handle the events in order, thus effectively reading data *like a consumer*. The input system only reads data it wrote itself, therefore not *consuming* information *produced* by other systems which makes it a pure *producer* in the context of information exchange between systems. This particular way to use a sector and its component arrays is the focus of this section as it highlights the importance of some guarantees made by the definition of the Entity Component System Architecture.

Just like the other systems documented here, the input system only covers a minimal set of features required to demonstrate its use case. Input events are limited to keyboard actions and consist of three integers as shown in listing 40.

```
1 typedef struct {
2         int key, action, mod;
3 } input_c;
```

Listing 40: Input event component with three integers encoding that a specific key was pressed or released

The state struct of the input system (listing 41) is more complex than those of the transform (listing 32) and movement (listing 28) systems previously shown.

```
1  static struct {
2          int initialized;
3
4          struct {
5                  comp_t input;
6                  sect_t sect;
7                  type_c ent_type;
8          } handle;
9
10         map *fnreg;             // mapping id to function pointer
11         map *keyreg;            // mapping id to int
12         map *keymap;            // mapping int to function pointer
13
14         _Atomic(uint64_t) seq;  // current input event sequence number
15         uint64_t handled;       // seq. number of last handled input event
16         enti_t sector_length;   // num. entities in the event sector
17
18         input_c *input;         // pointer to component array in event sector
19
20         mtx_t mapping_lock;
21 } I;
```

Listing 41: State struct of the input system

The first difference is the anonymous struct holding handles in lines 4 to 8. Where the other systems have only stored component handles, the input system also keeps a handle for the sector it created in line 6 of listing 42 and the type of entities representing input events (line 7). Lines 10 to 12 of listing 41 show three hash maps, which are initialized in the input systems init function in line 20. With these the input system can map associations between keys received as input events and registered callback functions.

```
1  int input_init()
2  {
3          /* ... */
4          set *comps = set_str();
5          set_insert(comps, &input_s);
6          I.handle.sect = core_mksector(bfromcstr("input events"), comps);
7          set_destroy(comps, NULL);
8
9          comp_t type_set[] = {I.handle.input, 0};
10         I.handle.ent_type = core_get_ent_type(type_set);
11         const sector_access *const sa = core_open_sector(I.handle.sect);
12         if (!core_mkent(sa, sa->length - 2, I.handle.ent_type)) {
13                 core_close_sector(sa);
14                 return ECSA_ERROR;
15         }
16         I.input = core_access_array(sa, I.handle.input);
17         I.sector_length = sa->length;
18         core_close_sector(sa);
19
20         I.fnreg = map_str(); I.keyreg = map_str(); I.keymap = map_int();
21
22         atomic_init(&I.seq, 0);
23         if (mtx_init(&I.mapping_lock, mtx_plain) != thrd_success)
24                 return ECSA_ERROR;
25         /* ... */
26 }
```

Listing 42: Initialization function of the input system (trivial parts omitted)

The remaining members are all related to the special usage of the input systems own sector for input events. By filling this sector with entities of the same type (line 12) it meets the guarantees from table 9. As a result, the component arrays of this sector won't be modified by the core. Since this sector is not accessed by any other system, the input system can forego safety measures such as opening and closing the sector through the core interface. Access to the array of input event components is simply stored in the state struct (listing 41, line 18) next to the length of the sector. To orchestrate access between the asynchronous callback adding new input events and the update function that sequentially processes input events, only two indices are needed. One to record the progress of processing events, seen in line 15. The other to mark the index of the most recently added input event component (line 14). Since new components are added through an asynchronous callback, this marker needs to be an atomic type and changes to it must be atomic operations as shown in listing 43, line 4 and listing 44, line 7.

```
1  void input_parse(const int key, const int action, const int mod)
2  {
3          uint64_t seq;
4          seq = atomic_fetch_add_explicit(&I.seq, 1, memory_order_seq_cst);
5          // compute index in range [1..n[
6          const enti_t ix = (seq % ((uint64_t)I.sector_length - 1)) + 1;
7          I.input[ix].key = key;
8          I.input[ix].action = action;
9          I.input[ix].mod = mod;
10 }
```

Listing 43: Callback function to receive input events

```
1  int input_update(const double dt)
2  {
3          if (!I.initialized) return ECSA_NOT_INITIALIZED;
4
5          mtx_lock(&I.mapping_lock);
6          const uint64_t sector_length = (uint64_t)I.sector_length;
7          uint64_t seq = atomic_load(&I.seq);
8          while (I.handled != seq) {
9                  enti_t ix = (I.handled++ % (sector_length - 1)) + 1;
10                 input_c *in = &I.input[ix];
11
12                 map_entry *e = NULL;
13                 if ((e = map_search(I.keymap, voidcast(in->key))) == NULL)
14                         continue;
15
16                 void (*fn)(int) = (void (*)(int))e->data;
17                 switch (in->action) {
18                 case GLFW_PRESS:
19                 case GLFW_RELEASE:
20                         fn(in->action);
21                         break;
22                 default:
23                         break;
24                 }
25         }
26         mtx_unlock(&I.mapping_lock);
27
28         return ECSA_SUCCESS;
29 }
```

Listing 44: Update function of the input system

Due to the guarantees made by the architecture this orchestration of receiving input events and processing them requires no mutex to be locked. In contrast, the mapping of input events to callback functions shown in needs to be protected by a lock (listing 41, line 20) while the update function is processing events (listing 44 line 5).

4.5.5 Collision system

Section 1.2 discusses the computational complexity that arises in systems that model interactions between simulation objects. Among the solutions mentioned to avoid the worst case complexity defined by equation 1.3 was the division of simulation objects into subsets. The Entity Component System Architecture supports multiple methods of dividing entities into subsets, most prominently the concept of sectors described in section 3.2.5 and demonstrated in all use cases so far. Another method that has been shown already is the filtering of entities based on component types (see listing 14 in section 4.3.3). The collision system is used to demonstrate how these methods can be utilized to reduce the complexity of solving interactions between pairs of entities.

The algorithm to solve collisions was chosen for being as simple as possible while still being useful for the task at hand. It's based on the separating axis theorem in two dimensions and handles no other shape than axis aligned bounding boxes (listing 45).

```
1 typedef struct {
2         float w, h;
3 } aabb_c;
```

Listing 45: Axis aligned bounding box component

The update function shown in listing 46 is a wrapper to separate the code for collision solving from the surrounding logic for opening and closing sectors.

```
1  int coll_update(const double dt)
2  {
3          if (!C.initialized) return ECSA_NOT_INITIALIZED;
4
5          const comp_t c_static[] = {C.handle.pos, C.handle.aabb, 0};
6          const comp_t c_moving[] = {C.handle.pos, C.handle.aabb, C.handle.vel, 0};
7          sect_t *s_static = core_get_sectors(c_static);
8          sect_t *s_moving = core_get_sectors(c_moving);
9
10         sect_t a, b;
11         for (int i = 0; (a = s_moving[i]); i++) {
12                 const sector_access *const sa = core_open_sector(a);
13                 for (int j = 0; (b = s_static[j]); j++) {
14                         iter(sa, a == b ? sa : core_open_sector(b));
15                 }
16                 core_close_sector(sa);
17         }
18         free(s_moving);
19         free(s_static);
20
21         return ECSA_SUCCESS;
22 }
```

Listing 46: Update function of the collision system

In total there are four nested loops, two in the update function and two in a special iterator function called inside the inner loop of the update function. In

lines 5 and 6 we see two different sets of components defined that are used in the following lines to get two filtered lists of sectors. In previous use cases the systems only ever acquired one such list with all sectors containing entities relevant to them. For the collision system all entities with a position and bounding box are relevant. But the collision system implements an interaction between two entities and this list would contain entities that don't need to be tested against each other. Two entities that don't move also don't need to be tested for collision interactions against each other. Therefore the collision system acquires a second list of sectors with the additional requirement to contain velocity components. With these two lists it's possible to test all entities with a bounding box that are able to move against all entities with a bounding box. The two nested loops starting in line 11 of the update function (listing 46) implement this logic with the granularity of sectors. Within the inner loop, each pair of sectors is passed to the iterator function shown in listing 47 that implements the iteration over component arrays and solves collision between pairs of entities.

```
1  int iter(const sector_access *const a, const sector_access *const b)
2  {
3          pos_c *pos1 = core_access_array(a, C.handle.pos);
4          pos_c *pos2 = core_access_array(b, C.handle.pos);
5          aabb_c *box1 = core_access_array(a, C.handle.aabb);
6          aabb_c *box2 = core_access_array(b, C.handle.aabb);
7          ueid_c *u1 = core_access_array(a, C.handle.ueid);
8          ueid_c *u2 = core_access_array(b, C.handle.ueid);
9          vel_c *v = core_access_array(a, C.handle.vel);
10         if (v == NULL) return ECSA_ERROR;
11
12         int count = 0;
13         for (enti_t i = a->first; i <= a->last; i++) {
14                 const float x1 = pos1[i][0], y1 = pos1[i][1];
15                 const float hw1 = box1[i].w * 0.5f, hh1 = box1[i].h * 0.5f;
16                 if (hw1 * hh1 == 0.0f) continue;
17                 const ueid_c ueid1 = u1[i];
18                 for (enti_t j = b->first; j <= b->last; j++) {
19                         float hw2 = box2[j].w * 0.5f, hh2 = box2[j].h * 0.5f;
20                         if (hw2 * hh2 == 0.0f || u2[j] == ueid1) continue;
21                         float x2 = pos2[j][0], y2 = pos2[j][1];
22                         float dx = x1 < x2 ? x2 - x1 : x1 - x2;
23                         float dy = y1 < y2 ? y2 - y1 : y1 - y2;
24                         float gapx = dx - hw1 - hw2;
25                         float gapy = dy - hh1 - hh2;
26                         if (gapx >= 0 || gapy >= 0) continue;
27                         if (gapx > gapy) {
28                                 pos1[i][0] = x1 < x2 ? x1 + gapx : x1 - gapx;
29                                 v[i][0] = v[i][0] * -0.125f;
30                         } else {
31                                 pos1[i][1] = y1 < y2 ? y1 + gapy : y1 - gapy;
32                                 v[i][1] = v[i][1] * -0.125f;
33                         }
34                         count++;
35                 }
36         }
37
38         if (a->handle != b->handle) core_close_sector(b);
39         return count;
40 }
```

Listing 47: Implementation of collision solving between entity pairs found by iterating sectors

The collision system is the first one shown that opens two sectors at the same time. The definition in section 3.2.7 notes that this could cause deadlocks if systems would run concurrently. Even without concurrency the collision system could cause a deadlock in the special case of trying to open the same sector in the inner loop that was already opened in the outer loop. To prevent this, the update function compares the sector handles and omits opening in case of a match (listing 46, line 14). The iterator function does the same for closing sectors at the end (listing 47, line 38).

The first thing of note in this iterator is that it acquires access to two component arrays per component type for position, bounding box and ueid, as well as to a single array of velocity components. This reflects the filtering for two different subsets of sectors and that the iterator solves interactions between *pairs* of components of which only one has a velocity.

The algorithm combines X- and Y-coordinates of the position component with the width and height of the bounding box component to compute the four corners of the axis aligned bounding boxes. These are then used according to the separating axis theorem to detect overlapping regions. To resolve the collision a vector is computed - based on this overlap - by which one of the entities needs to be moved. This vector is applied to the position of the first entity, which is always movable due to having a velocity component. Additionally, the velocity component is scaled by a factor of -0.125 to simulate a bounce effect. Line 16 and 20 of listing 47 implement checks for opportunities to abort computation early. The first skips the inner loop entirely for the current entity if the bounding box has an area of 0. This also catches entities without a bounding box since the uninitialized value of the "aabb_c" component produces the same result. The second check does the same for the bounding box of the other entity as well as comparing the ueids of both entities to avoid collision checks of entities against themselves.

5 Evaluation

This section is dedicated to evaluating the Entity Component System Architecture defined by this thesis. The ability to implement a realtime simulation with this architecture is concluded. Insights into the subject of simulation software architecture are presented and limits and capabilities of the architecture are discussed based on the use cases implemented and described in section 4.

Implementing the Entity Component System Architecture The architecture defined in section 3 was implemented in a library in less than 1000 lines of C code (see section 4.1, table 11). With this library a simulation engine was developed that covers a variety of use cases common to game engines including, but not limited to, user input, collision detection, graphics rendering, movement physics and hierarchical transformations. As a result a working example simulation has been produced within the time frame of this bachelors thesis, proving that the architecture is indeed functional and sufficiently simple.

It has been shown that simulation objects can be represented by entities that associate multiple components by an index into parallel arrays. This SoA design adheres to the principle of locality known from data oriented design, making the architecture inherently compatible with this paradigm. These arrays can be iterated efficiently as described in section 3.1 and shown in all use cases (eg. listing 30). It was necessary to complement these arrays with associative structures that support the arrays by providing fast searching for addressed entities (see section 3.2.1, "Maps"). Among the five described use cases the hierarchical transform system was the only one to make use of this, all others were implemented using iteration of component-arrays as sole access method.

The systems could be implemented independently from each other and existing functionality did not suffer from the introduction of new component types or systems. Neither was it necessary to re-factor existing systems or component types at any point during development. This shows that (at least for the use cases presented) it is possible to reduce the effects of *crosscutting concerns* between systems.

Furthermore, all of the core concerns for simulation software defined in section 1.3 have been modeled with this architecture and the implementation described in section 4 was able to demonstrate that the majority of data access in common simulation use cases could be achieved with just these structures.

Grouping with Sectors To combat the complexity of simulation object interactions described in section 1.2.1 a method for *grouping* objects was needed. Section 3.2.5 has defined the concept of sectors to achieve this. In the use case of the collision system described in section 4.5.5 this has been demonstrated successfully. The input system described in section 4.5.4 has demonstrated how the efficiency of sectors can be maximized by reserving an entire sector for entities of the same type, resulting in component-arrays with 0 gaps and reduced maintenance cost for the core.

The notion of grouping entities based on segmentation of the simulation world (eg. levels) has been mentioned in section 1.2.2. Directly mapping levels to sectors would restrict the entity count in the level since the memory available to each sector is fixed for the runtime (see section 3.2.5). How efficiently this

memory is used depends on the component types of the entities. A more appropriate approach would be to analyze the set of entities in a world-segment for optimal distribution to multiple sectors. This optimization could even be done offline using algorithms not suited for realtime computation.

Sectors also provide access control to prevent race conditions when multiple systems try to access overlapping subsets of entities.

System	min	max	mean	stddev
core	0.000570	0.017615	0.001387	0.001056
physics	0.000002	0.000017	0.000004	0.000001
transform	0.000002	0.000021	0.000004	0.000001
renderer	0.000008	0.000039	0.000015	0.000002
input	0.000000	0.000015	0.000001	0.000001
collision	0.000002	0.000028	0.000005	0.000001

Table 15: Runtime of system update functions in seconds. Measurements from 632 frames over 3 executions. In a pool with 32 sectors of 2048KiB, 4 sectors were created. 11 regular entities were split (7+4) between two sectors for static and dynamic geometry. A sector for input events was filled with 80654 entities and one for log messages with 17182 entities. In all 3 runs the only spike was the update of core after all entities (97847) were added. (Hardware used: Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz)

Performance Characteristics The formal definition of the Entity Component System Architecture in section 3 has allowed identifying performance characteristics. Runtime performance can be separated between the execution of systems simulating entity behavior and the maintenance of the core.

Implementation of the use cases suggests that systems following the design described in section 4.4 run with predictable efficiency. No performance bottlenecks could be identified in the design of the architecture for systems. The mean and standard deviation of the measured execution times shown in table 15 suggest highly predictable performance for their update functions. For the performance of the cores maintenance a bottleneck could be identified in the number of modifications made to the cores "ent_map" described in section 3.2.1. Creation or destruction of a large number of entities is limited by the single-threaded modification of this map. The data structure used here is an Adaptive Radix Tree[18] from an external library[19]. As table 15 shows, adding roughly 100.000 entities in a single frame causes the cores update to take approximately 18 Milliseconds on the testing hardware. On average over 632 frames (including the spike) the cores maintenance took 0.001387 seconds. In that time the core has iterated the component-arrays of "type_c" and "flag_c" components across 4 sectors with 17182, 80654, 19054 and 15646 columns respectively, suggesting efficient use of the CPU's cache.

The measurements shown here support the premise of defining iteration of parallel arrays as main method to access data and relying on more complex structures only as optimization for some use cases. It is possible to reason about

[18] `http://www-db.in.tum.de/~leis/papers/ART.pdf`
[19] `https://github.com/armon/libart`

the performance characteristics of the implementation based on the definition of the architecture.

Limits of the Architecture The efficiency of iterating component arrays depends on the distribution of entities with different sets of components within these arrays. For the simple use cases described in section 4.5 the distribution of entities between sectors could be chosen by hand. But how an efficient distribution of entities to sectors can be accomplished for complex simulation worlds such as those of entire video games has not been explored.

While the implementation of the input system described in section 4.5.4 shows that systems can operate asynchronously and the architecture provides some protection from race conditions through sector access tokens (section 3.2.5), there is no explicit support for concurrent execution of systems.

6 Conclusion

The analysis of the problem domain and definition of core concerns in section 1 has provided valuable insight that has lead to a clear definition of a simple but capable architecture.

The Entity Component System Architecture has been defined in section 3 and successfully implemented in code. Working examples of common use cases for simulation software have been demonstrated in sections 4.5.1 to 4.5.5. Implementing the architecture is possible within a short time frame, making the effort manageable. Doing so can be justified with the performance characteristics shown in section 5 as well as the ability to reason about performance and bottlenecks reliably.

7 Outlook

Optimization of Sector Sorting The reason for sorting component-arrays within a sector is improved cache efficiency for sequential iteration. Currently, the sorting is defined to cluster entities with the same set of component types and leave no gaps (see table 10). What is not defined is the order of these clusters. But the similarity of entity types based on their set of components could prove a useful metric for ordering the clusters to reduce the number of gaps further. Since the "type_c" component already encodes an entities set of components as string, algorithms for string matching based on distance metrics should be a good starting point.

Concurrency and Access Control The Entity Component System Architecture does not provide a solution for running systems concurrently. How the definition of the architecture could be extended to take care of - or help with - the concurrent execution of systems is an interesting subject for future research. In particular, solving dependencies between systems accessing the same data to avoid dead-locks.

A related subject for future research would be more sophisticated schemes of access control. Simply differentiating between *read-only* and *read-write* has the potential to simplify concurrent access to data.

References

[1] Mike Acton. Data-oriented design and c++. *Luento. CppCon*, 2014.

[2] Scott Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.

[3] Jonathan Blow. Game development: Harder than you think. *Queue*, 1(10):28–37, 2004.

[4] Michael Doherty. A software architecture for games. *University of the Pacific Department of Computer Science Research and Project Journal (RAPJ)*, 1(1), 2003.

[5] Ulrich Drepper. What every programmer should know about memory. 2007.

[6] David H Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., 2004.

[7] Jason Gregory. *Game engine architecture, Second Edition*. AK Peters/CRC Press, 2017.

[8] Luc. Entities, components and message handling in games, November 2015.

[9] Adam Martin. Entity systems are the future of mmog development, September 2007.

[10] Paul E McKenney. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton*, 2010.

[11] Stoyan Nikolov. Oop is dead, long live data-oriented design. CppCon, 2018.

[12] Jorge Revelles, Carlos Urena, and Miguel Lastra. An efficient parametric algorithm for octree traversal. 2000.

[13] Mick West. Evolve your hierarchy, January 2007.