Fachhochschule Salzburg University of Applied Sciences

Building a Bridge between the Entity-Component-System and Data-Oriented Design

BACHELORARBEIT 1

StudentIn Alessandro Morio, 1410601014
BetreuerIn DI Dr. Markus Tatzgern

Salzburg, 06.06.2016

Eidesstattliche Erklärung

Hiermit versichere ich, Alessandro Morio, geboren am **26.03.1992** in **Freilassing**, dass ich die Grundsätze wissenschaftlichen Arbeitens nach bestem Wissen und Gewissen eingehalten habe und die vorliegende Bachelorarbeit von mir selbstständig verfasst wurde. Zur Erstellung wurden von mir keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Ich versichere, dass ich die Bachelorarbeit weder im In- noch Ausland bisher in irgendeiner Form als Prüfungsarbeit vorgelegt habe und dass diese Arbeit mit der den BegutachterInnen vorgelegten Arbeit übereinstimmt.

Salzburg, am **06.06.2016**

Unterschrift

Vorname Familienname	Personenkennzeichen

Kurzfassung

Diese Bachelorarbeit untersucht Methoden des *Data-Oriented Designs* (DOD) und Elemente des *Entity-Component-Systems* (ECS) bezüglich der Spieleentwicklung. Darüber hinaus werden Methoden des DOD auf die Elemente des ECS angewendet um die Performance in einem ECS in Bezug auf ein Cache freundliches Design zu verbessern.

Aufgrund der *komponentenbasierten Architektur* des ECS, welches das objektorientierte Prinzip *Komposition an Stelle von Vererbung* verwendet, ist der Quellcode gut strukturiert, wiederverwendbar und erweiterbar. Das ist notwendig, wenn man große und komplexe Spiele entwickelt. Das ECS besteht aus drei Hauptbestandteilen: den Entitäten, den Komponenten und den Systemen. Eine Entität repräsentiert ein Element in der Spielwelt. Komponenten stellen aufgabenspezifische Daten dar und haben keine oder wenige Abhängigkeiten. Aufgabenspezifische Systeme verwalten und verarbeiten ihre jeweiligen Komponenten. Das Verhalten einer Entität wird durch die Komponenten bestimmt aus denen sie besteht. Komplexe Entitäten bestehen einfach aus mehr Komponenten als Einfache. Die Wiederverwendbarkeit des Quellcodes wird durch die Komponenten gewährleistet. Diese können in anderen Projekten mit gar keiner oder wenig Änderungen wiederverwendet werden. Während eines Spiels fragmentiert der Arbeitsspeicher aufgrund der inkohärenten Erstellung und Löschung von Spielelementen. Das verlangsamt die Geschwindigkeit mit der eine CPU arbeiten kann.

Um der Fragmentierung des Arbeitsspeichers entgegenzuwirken werden Methoden des DOD angewendet. Die Grundlagen der Cache Architektur einer CPU werden erläutert um ein besseres Verständnis des DOD zu bieten. Durch Methoden des DOD wie *Contiguous Arrays*, *Packed Data* und *Hot/Cold Splitting* wird eine wesentliche Steigerung der Performance erreicht. Diese Methoden organisieren Daten auf eine Cache freundliche Art und Weise und werden auf die Elemente des ECS angewandt.

Schlagwörter:

Entity-Component-System, Data-Oriented Design, Spieleentwicklung, Komponentenbasierte Architektur, Cache Ausnutzung, Komposition an Stelle von Vererbung

Abstract

This bachelor thesis examines practices of *Data-Oriented Design* (DOD) and elements of the *Entity-Component-System* (ECS) in terms of game development. Furthermore, practices of DOD will be applied to elements of the ECS to enhance the performance of an ECS concerning a cache-friendly design.

Due to the *component-based architecture* (CBA) of the ECS, which uses the object-orientated programming principle *composition over inheritance*, the code base is well structured, reusable and extendible. That is mandatory when building large and complex games. The ECS consists of three principle elements: entities, components and systems. An entity represents an element in the game world. Components represent task-specific data and have no or few dependencies. Task-specific systems host and process its components. The behaviour of an entity is determined by the components it is composed of. Complex entities just hold more components than simple ones. The reusability of the code base is granted by the components. They can be reused in other projects with no or little change. During a game memory becomes fragmented due to the incoherently creation and deconstruction of game objects. That slows the processing speed of a CPU down.

To prevent the fragmentation of memory practices of DOD are applied. Basics of a CPU's cache architecture will be specified to provide a better understanding of DOD. Through DOD practices like *contiguous arrays*, *packed data* and *hot/cold splitting* a significant performance boost is achieved. These practices organize data in a cache-friendly way and will be applied to elements of the ECS.

Keywords:

Entity-Component-System, Data-Oriented Design, game development, component-based architecture, cache utilization, composition over inheritance

Contents

1 Introduction

The video game industry's budgets and sales figures increase year by year (Hight and Novak 2008, 8-17). In the early 1980s the budget of a video game project ranged from $1,000 up to $10,000. Most games were developed by a single programmer in a few weeks. Project management was informal and the only goal was to finish the game. In the 1990s game budgets started with $50,000 and rose up to $1,000,000. Team sizes increased to 6 - 20 people. At that time project management became an important part of the game development process. At the beginning of the 21st century 3D game development spread widely. Specialized project managers for every aspect of the game development cycle were needed. 30 – 80 people were involved in finishing a game project. The budgets of those video game projects ranged from $5 million to $30 million. In recent years, from 2010 to 2014, video games worth about $80 billion were sold in the U.S. (Entertainment Software Association 2015, 12). Hence, video game development has become a serious business over the years.

In order to successfully finish game projects, a game specific software framework is needed which saves time and money (DeLoura 2009). A framework that makes game development easier in terms of software engineering and working with art assets. Such a framework is called a *game engine*. Before the first game engines were developed, all the game's logic had to be implemented in hardware (Madhav 2013, 2-5). With the introduction of the Atari Video Computer System (Atari 2600) in 1977, a standardized platform for video games was introduced. Since then, game development was more about programming than designing complex hardware. At that time hardware limitations were a fact game developers had to consider and games were written in Assembler. There were no development tools for debugging. Therefore, the game programming effort was higher. By the 1990s the programming language C spread widely (Ritchie 1993) and debugging software was available on the PC and the Macintosh (Hight and Novak 2008, 13). That enabled game projects to become more complex. In the mid-1990s the video game *Doom* was released by id Software (Gregory 2014, 11-30). Doom's software architecture defined a clear separation between the core software components, art assets, game world and game logic. Due to the clear separation, new games of the same genre were developed by just creating new art, game worlds and game logic with minor changes to the engine software. Therefore, developers saved time and money because they could reuse core software components. At that time the term *game engine* was associated with Doom. In the late 1990s first-person shooter (FPS) games like Quake III Arena and Unreal were designed in terms of reusability. The Quake engine could be customized with a scripting language called *Quake C*. With these engines, nearly every game of the FPS genre could be created. Current game engines like Unity, Unreal Engine and Source Engine allow the developer to create large and complex games of a vast variety of genres. They implement diverse subsystems like Havok Physics (Madhav 2013, 4). Therefore, programmers do not need to spend time writing and maintain specific subsystems.

Modern game engines need a way to manage thousands of game objects (Gregory 2014, 340-341, 848). They are rendered and updated multiple times per second. Due to that fact a well-designed software architecture is needed. Software architecture is the way a program is organized and how code base is designed (Nystrom 2014, 9-17). A well-designed software architecture is flexible, reusable and extendible. It expects changes and provides a clean way to do so. The *Entity-Component-System* (ECS) is a software architecture that is capable of integrating changes in a clean way and produces reusable code (Entity Systems Wiki, 2014a). The ECS is a *component-based architecture* (CBA). Modern game engines like Unity are designed as CBAs (Unity Technologies 2016c).

Assume a well-designed game engine architecture provides a clean way a game is organized. It provides access to collections of game objects (Gregory 2014, 51-52). Moreover, the architecture manages the construction and deconstruction of game objects and the communication system. All the game objects are placed in RAM so that the CPU can access and process them (Nystrom 2014, 269-278, 305). Furthermore, thousands of game objects are created and deleted incoherently during a game. The memory is fragmenting. The performance of a CPU that requests fragmented data drops by quite a lot due to the fact that it has to access memory frequently whenever the data is not stored contiguously. Accessing RAM is slow compared to the processing power of a modern CPU. Therefore, a method for organizing fast process able data is needed. That method is called *Data-Oriented Design* (DOD) (Collin 2014). In terms of game development, DOD arranges game object data in *contiguous arrays* to achieve a defragmented memory layout (Nystrom 2014, 275-284). Another DOD practice is the compression of game object data into a mandatory collection of attributes. That leads to maximum of data objects a CPU can retrieve.

2 Game Engine Architecture: Object Model Architectures

The *object model architecture* is a large and important component of a game engine architecture (Gregory 2014, 854-873). Game object model architectures determine how game objects are modelled and simulated in the game world. They define a variety of game elements, their behaviour and properties that exist in the virtual world. There are many ways how game object model architectures are designed. However, there are two basic architectures: CBAs and *object-oriented architectures* (OOAs).

2.1 Object-Oriented Architecture (OOA) Models

A traditional and intuitive game object model architecture sets up a class hierarchy for game objects (see figure 1) (Gregory 2014, 873-881). OOAs are made of *is-a relationships*. This relationship is known as *inheritance*. In that object-oriented approach every game object is represented by a single or few class instances. Properties and methods of

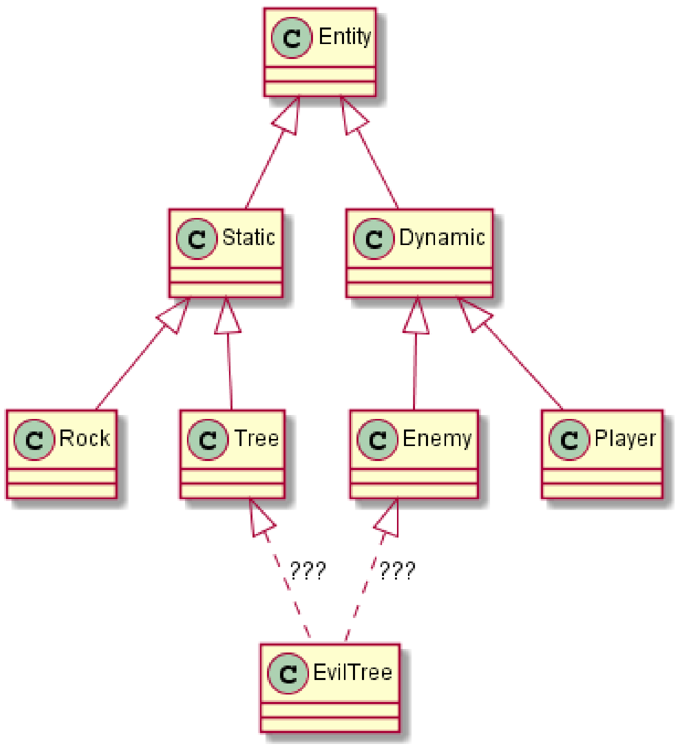

Figure 1: Example class hierarchy in a game (Gregory 2014, 854-879). Functionality is added through *inheritance*. Class hierarchies may cause several problems like the *deadly diamond*.

parent classes are inherited to child classes. At first glance that means less code duplication. Initially game class hierarchies are simple and meaningful. However, as they grow they become wide and deep. Moreover, there arise several problems with wide and deep hierarchies.

First of all the maintenance and modification of deep and wide class hierarchies cause problems (Gregory 2014, 877-878). If a derived class has to be changed, it and all of its parent classes have to be understood. Changing an overwritten function may interfere with the parent classes' intent of the function. Therefore, without knowing about parent classes, changing a derived class may cause serious bugs.

Another problem with class hierarchies is *multiple inheritance* (Gregory 2014, 99, 879). Assume there is a new class which needs the behaviour and properties of two hierarchy axes. Multiple inheritance enables a class to derive from two or more base classes. At first multiple inheritance seems to solve the issue. However, multiple inheritance converts a simple tree structure to a potentially complex graph. Additionally, a graph can lead to the *deadly diamond* (see figure 2). The deadly diamond describes the problem in which a derived class has two copies of the same grandparent base class. A final disadvantage with deep and wide class hierarchies is that they may cause *The Blob* (McCormick 1998) also known as *the bubble-up effect* (Gregory 2014, 880-881). Initially the root of class hierarchies is simple and only contains mandatory features. Though, as class hierarchies

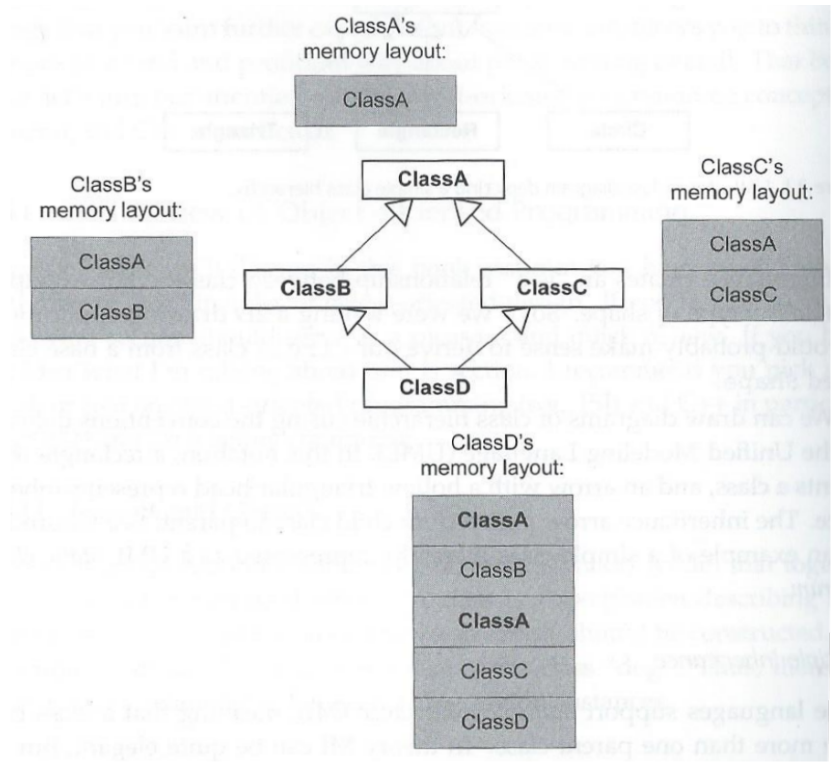

Figure 2: The *deadly diamond* and the resulting class memory layout (Gregory 2014, 99).

grow and new functionality is added, some code has to be shared by two or more unrelated classes. One possibility is to use multiple inheritance. However, that may cause the deadly diamond. The other possibility is to move the new functionality to a common parent class. Over time all the important functionality is moved to the root classes. That leads to big classes which no one dares to change. Additionally, the blob class appears in nearly every subsystem. This is problematic because everyone who is working on the project has to deal with that class.

2.2 Component-Based Architecture (CBA): Composition over Inheritance

In contrast to OOAs there are CBAs (Gregory 2014, 881). CBAs consist of *has-a relationships*. That relationship is known as *composition* (see figure 3). Composition means that a source class holds an object, pointer or reference to another linked class. New functionality is added through composition instead of inheritance. The source class owns the linked class. That means the lifetime of the linked class depends on the source class.

CBAs favour *composition over inheritance* because there are several advantages of using composition. A big advantage of composition is that objects can change their behaviour at runtime (Gamma et al. 2007, 18-20). Behaviour changes through adding and removing references to other objects. Another effect of composition is that classes are more focused

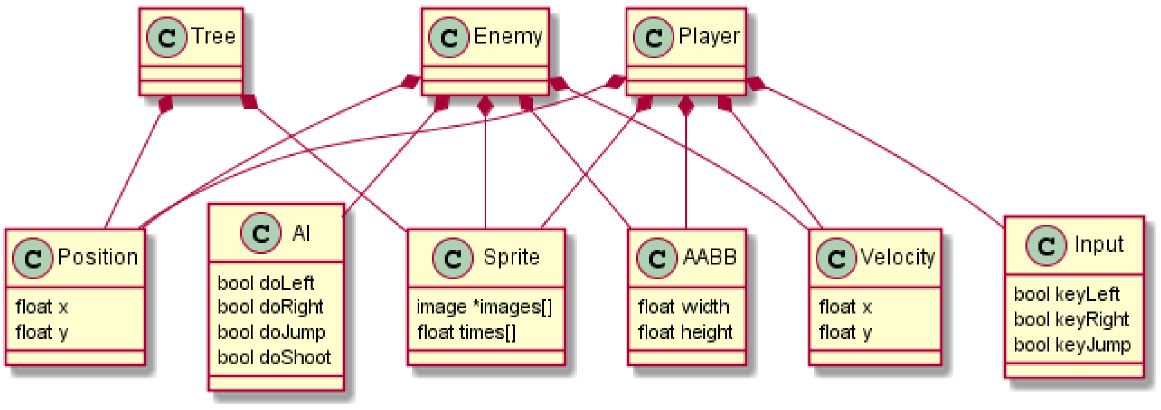

Figure 3: Example class composition in a game (Gregory 2014, 881). Functionality is added through *composition instead of inheritance*.

on one task. The first step in a CBA is to split software into small chunks of task-specific data and functionality (Allan et al. 2006, 164). These small units of data and functionality are called *components*. Components are plugged together to create complex applications. Small task-specific units of software are easier to understand and maintain than complex class hierarchies. Additionally, well-designed components can be reused in different applications with little or no change. Moreover, components are designed task related. That means modifying a component requires no or little knowledge about other tasks. That saves time and money when working with a CBA.

There are game specific frameworks using a CBA like *Artemis-odb* (Papari 2016) and *Entitas* (Schmid 2016). The concrete CBA both frameworks use is the ECS. *Artemis-odb* is a high performance ECS framework written in Java. It supports HTML5, Android and iOS. *Entitas* is also an ECS framework especially made for C# and Unity. *Entitas* uses internal caching and garbage collector specific optimizations to enhance the performance. There also exists a Unity module which provides editor extensions to work with *Entitas*.

3 Entity-Component-System (ECS)

This thesis examines the ECS because it offers advantages which are mandatory for building large and complex video games (Allan et al. 2006, 164). When developing a video game there has to be space for changes and extensions. The produced code may be reused in other projects. The ECS is a software architecture which provides flexibility, reusability and extendibility in terms of software development (Entity Systems Wiki, 2014a). That is important for large projects like video games. The ECS is a CBA which mainly consists of three principle elements: *entities*, *components* and *systems* (Entity Systems Wiki, 2014b). In terms of a CBA the components of the ECS represent task-specific data. Systems modify their respective components and represent the task-specific functionality. Entities group different component instances and form an element in the game world.

Entity

An entity or game object is an element in the game world (Gregory 2014, 848). The game world can consist of a vast amount of entities. Traditionally, game elements can be divided into static and dynamic elements. Static elements don't actively move or interact with gameplay. Examples for static elements are trees, rocks and walls. Dynamic elements include heroes, monsters and spells. Furthermore, an entity does not always have to be visible (Bilas 2002). A camera and a trigger area can also exist as an entity in the game world.

In OOAs entities are represented by class hierarchies (Gregory 2014, 873-881). However, there occur several problems when setting up class hierarchies for entities like the deadly diamond. In a CBA an entity mainly consists of a unique identifier and a collection of attributes respectively components (Gregory 2014, 854, 871-873, 886). The components of an entity determine the behaviour in the game world. Assume there is a *health component* for example. That component determines whether the entity is damaged, currently losing health or is dying. Depending on the state, the behaviour of the entity can change. The unique identifier is used to identify or search for a particular object. In most cases the unique identifier is an integer or a hashed string variable due to performance issues (Gregory 2014, 276-277). Additionally, an entity can be classified with different tags. Those tags can be used to query entities of a specific tag (Unity Technologies, 2016b).

The classic way to grant access to entities, is to store them in a global accessible list (Entity Systems Wiki 2014c). The *EntityManager* class contains that list of entities and grants access via functions. Moreover, the *EntityManager* manages the creation, modification and deletion of its entities. In some CBAs the entity class is entirely omitted (Gregory 2014, 886-887). The *EntityManager* no longer holds collections of entities. The methods for working with entities are built around their components.

Component

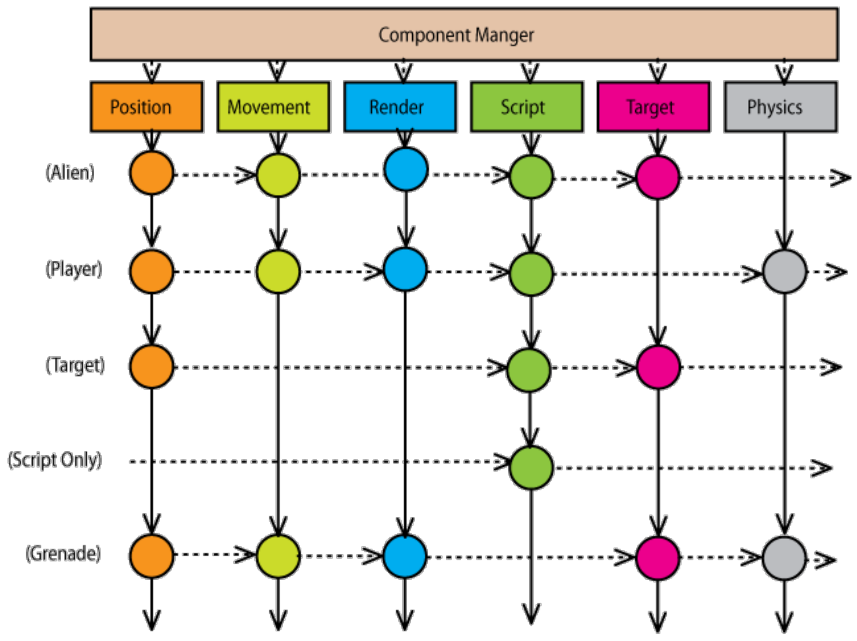

Figure 4: This figure shows how entities can be composed of different components (Gregory 2014, 873-877). The behaviour of an entity changes depending on the components it is composed of.

The classic approach to provide an entity with abilities or behaviour is through inheritance (Gregory 2014, 873-877). In a CBA an entity is extended through *composition*. A component represents an ability or behaviour an entity can consist of (see figure 4) (Gregory 2014, 881-891). Components add features to entities through *composition instead of inheritance*. The usage of components decouples software features, which in a class hierarchy, have to interact with each other due to inheritance (Nystrom 2014, 213-217). Components stand for their own. They don't know each other. When designing a new component it should have no or in exceptional cases few dependencies. Therefore, modifying components is easy. The person who changes a component should not care about other components. At some point components have to interact. However, that can be directly controlled and not happens due to inheritance.

There are three ways how components can be organized (Gregory 2014, 881-887):

- Direct component references
- Generic components
- Pure component model

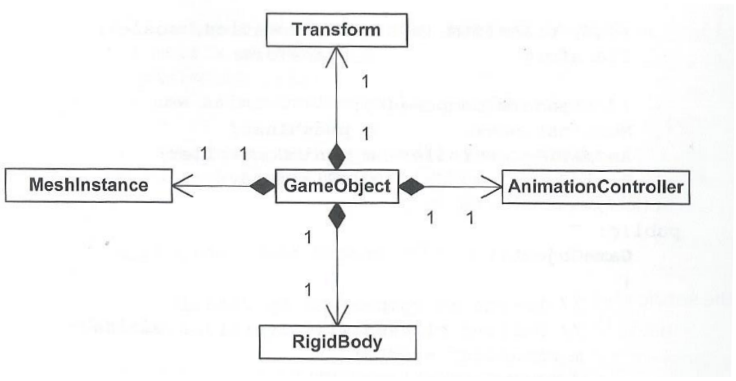

Figure 5: A game object class with *direct component references* (Gregory 2014, 881-885). A simple and straight forward approach.

Using *direct component references* is a simple method to implement components (see figure 5). Each component is represented by one class (Gregory 2014, 881-885). Each class contains attributes and methods which determine the behaviour of an entity at runtime. One possible implementation is a base entity class which holds references to mandatory components like a *transformation component*. Classes which derive from base entity class define new component references to append new features. The components are created in the constructor of derived entity classes. This approach uses hard-coded component references. Therefore, it is not very flexible.

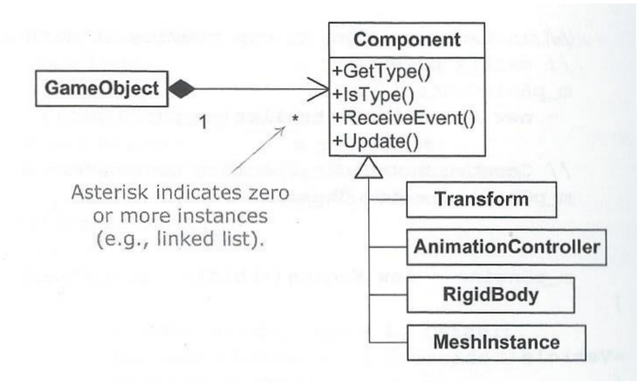

Figure 6: A game object class with a one-to-many relationship to *generic components* (Gregory 2014, 885-886). Generic components are more flexible than direct component references.

More flexible is the usage of *generic components* (see figure 6). Each concrete component derives from a base component class (Gregory 2014, 885-886). The entity class holds a list of generic components. When iterating over the list of generic components, polymorphic operations are performed. Such polymorphic operations include checking the type of component and passing events to each component for a possible processing. Instead of a fixed number of components per class, this design allows to dynamically create and add only the components to an entity which are needed. Moreover, deriving from specific entity classes is now unnecessary.

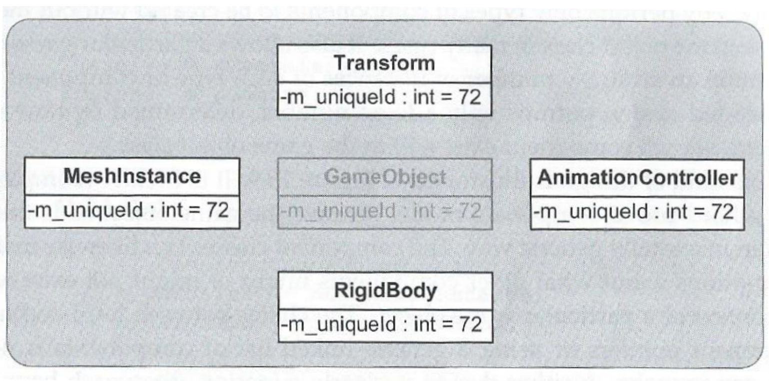

Figure 7: *The pure component model* (Gregory 2014, 886-887). The components themselves now hold the unique identifier of their parent entity. The entity class is omitted.

The *pure component model* goes one step further (see figure 7) (Gregory 2014, 886-887). An entity only consist of a unique identifier and a collection of components. There is no other logic or important data in the entity class. Therefore, the pure component model eliminates the entity class completely. Instead of the entity, the respective components hold the unique id of the entity. Due to the fact that there is no entity class any more, the creation of components needs to be considered. One solution is to use a factory pattern which is responsible for the creation of components. Another challenge is the communication between the components of an entity. Before the entity class was omitted, components could communicate over the parent entity object. Therefore, there has to be a new way components communicate with each other. One solution is a look-up function to connected components by the entity's unique identifier. The look-up function must be fast because it is executed frequently. Another approach is to add the sibling components into a linked list.

3.1 Systems: Game Functionality and Organization

A game engine consists of many interacting systems (Gregory 2014, 340-341, 916). Most systems provide a periodically processing service for a particular task. In the ECS systems host and update their respective components (Cohen 2010, 34-60). Each system's update method is called in the game loop (Gregory 2014, 340-346). In one update step of a system all of its associated and active objects are iterated and updated when required. There are systems needed for rendering, animation, collision detection, movement and so on. The *animation* and *rendering system* for example need to be synchronized at a rate of 30 or 60 Hz to perform a fluent graphical representation.

The systems of a game engine provide functionality (Gregory 2014, 340-346). Each system provides different functionality. A *render system* for example provides the logic for displaying entities on the screen. Another example of a system is the *input system.* This system processes the input for a game. There is a variety of input types: mouse, keyboard, controller, touch and so on. Depending on the input different game logic is executed.

Assume there is a game with astonishing graphics, lots of game logic and functionality. At some point new functionality must be added to the game (Gregory 2014, 344). Therefore, there has to be a clean way to implement new functionality to the game over and over again. However, the game loop and the systems are already written. In order to add new functionality in a clean way the systems provide a possibility to add callback functions to their update method. That enables the programmer to add as much functionality as he or she wants by just adding new callback functions.

Some content of the game has to be processed every frame or update step. Examples are render able and animate able content. Besides that, there is also content which must only be updated under certain circumstances (Gregory 2014, 345). For example if a player presses a button or an explosion is going off. Those circumstances are called *events.* An *event system* processes the events of a game. The event system permits other systems to register for specific events, fire events and allows them to respond to fired events. Actually some event systems allow registering periodically events. Moreover, these periodic events can be used to implement periodic updates by firing an event 30 or 60 times a second.

Due to the fact there are many interacting systems, they have to be organized (Gregory 2014, 231). Hence, before a system can work, it has to be configured and initialized. Some systems depend on other ones. A *2D animation system* could depend on a *texture system.* Therefore, the order of initialization is important. When the applications exits, the systems typically are shut-down in reverse order.

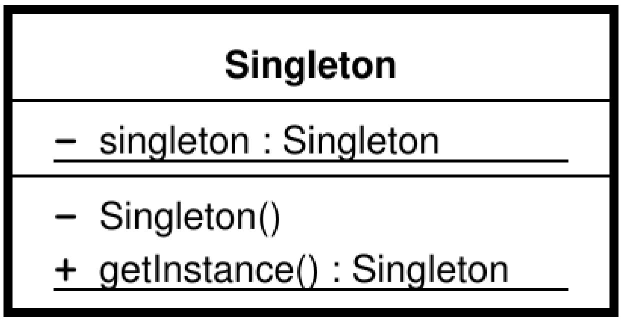

Figure 8: Singleton pattern (Gregory 2014, 232-234). The static *getInstance()* function is used to retrieve the only static instance of a class.

In order to organize inter-connected systems, globally accessible systems are required. A straight forward way would be to create a static instance for each system (Gregory 2014, 232-234). However, there is no control about how many systems of one kind can exist. For most systems there should only be one instance. In order to accomplish that issue the *singleton pattern* is one solution (see figure 8). Additionally, there is no need to create a global instance. The system's instance is globally accessible by a static class function.

Hence, there can only be one system. A method for start-up and shut-down is needed

(Gregory 2014, 234-236). Therefore, each system has a start-up and a shut-down function. The systems' start-up functions are called in the required order before the game loop starts. In those functions the required systems are retrieved with the singleton pattern. After the game loop, the systems' shut-down functions are called in reverse order.

Inside the game loop systems process their respective objects. Though some systems, also depend on other ones in terms of processing (Gregory 2014, 922-925). As the order of start-up and shut-down system functions is a vital part, the order of the systems' updating function also is. A *physics system* may has to be updated first. Moreover, a physics system may need a more frequently called update function to calculate physics more precisely.

3.2 Communication between Entities, Components and Systems

In a game entities, components and systems have to communicate (Gregory 2014, 52). Otherwise there would be no change or progress in the game. In terms of the ECS there are two main types of communication:

- Inner-entity communication
- Inter-entity communication

The communication inside an entity, the communication of sibling components, needs to be considered when designing an ECS. The reason for the need of communication between sibling components is because components are isolated and stand for their own (see section 3 Component). They don't know each other. A simple and fast but tightly coupled approach would be to store references to required components in the respective component classes (Nystrom 2014, 227-228). Another more dynamic approach is that the entity class provides an access function to return its components (Unity Technologies, 2016a). The component could hold a reference to the parent entity. That reference can be used to retrieve all the components the parent entity holds. In a pure component model that access becomes a little trickier (Gregory 2014, 886-887). There is no parent entity. The sibling components only share the same entity unique identifier. With a smart look-up function, which determines the position of sibling components in their respective systems list by the entity's identifier, the access to sibling components can be granted. There is another more complex possibility how sibling components can communicate (Nystrom 2014, 229-231). A little messaging system between sibling components can be implemented. Each component would have a receiving function to process incoming messages. The parent entity broadcasts the redirected message from a component to all sibling components.

The second type of communication, the *inter-entity communication*, needs a specific system that handles the communication between entities (McShaffry et al. 2009, 33). By default, there are no connections between entities. The system that performs that task is called *event system* (see section 3.1 Systems: Game Functionality and Organization). That system allows other systems to register for specific events, fire events and allows them to respond to fired events. When an entity is created or updated some systems may want to react to that event (McShaffry et al. 2009, 33). To realize that, an event is fired when an entity is created or updated in specific circumstances. The event system is a clean way to handle communication. Instead of registering for specific subsystems to listen to specific events there is one point where the events are handled. When for example a collision occurs in the *collision system*, it simply sends an event to the event system and all systems which registered for an event of that type will be notified. Therefore, any type of inter-entity communication is possible. There can be global events where every system responds to. There also can be entity and component specific events.

3.3 Design Patterns

The ECS also reveals several design patterns. The most significant patterns are:

- Model-View-Controller (MVC)
- Strategy
- Observer

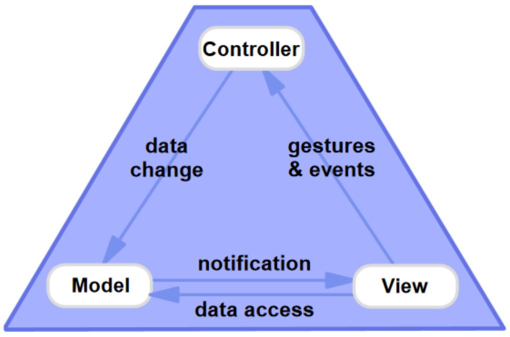

Figure 9: Model-View-Controller pattern (Potel 1996, 1-2). This pattern is always present in the ECS.

The MVC design pattern (see figure 9) was used for the implementation of *Smalltalk's*[1] graphical user interface (Potel 1996, 1-2). The *model* in the MVC represents data. The *view* defines how the data of the model is displayed on the screen. The *controller* determines how user interface interactions and events change the model. Since game development is not primarily about user interfaces, the MVC has to be adapted to fit the ECS. In the ECS the MVC pattern is always present. The model is represented by the

1. Programming language

components (see section 3 Component). Components only hold data and have no or few dependencies. A *render system* represents the view (see section 3.1 Systems: Game Functionality and Organization). The render system accesses the data of render components and displays them on the screen. The controller is represented by a variety of systems. Since systems update their respective components, they are responsible how their components are changed. However, modern game engine architectures may implement the *Model-View-Presenter* (MVP) pattern (Potel 1996, 1-8). The MVP is a design pattern which is based on the MVC. The MVP is a more modern approach which specifies data management, events and the presentation of data in a more concrete way.

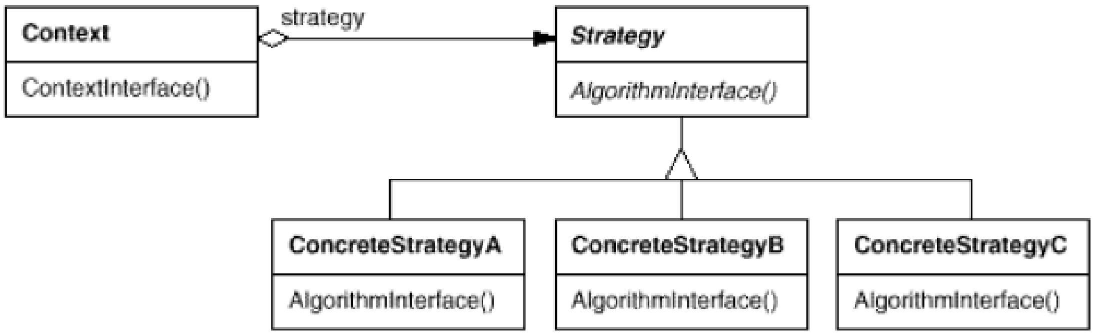

Figure 10: Strategy pattern (Gamma et al. 2007, 315-317). Is applied to the ECS when an entity class holds a list of generic components (see sections 3 Entity and 3 Component).

The *strategy pattern* (see figure 10) consists of three parts (Gamma et al. 2007, 315-317). The first part is the *strategy* interface which defines a common interface for *concrete strategies*. The second part are concrete strategies which implement the interface. The final part is the *context*. The context has a reference to a strategy object and manages that reference. That reference of the context is configured with a concrete strategy. The strategy pattern has to be adapted to work with the ECS (see sections 3 Entity and 3 Component). Instead of one reference the context holds a list of references to concrete strategies. Additionally, the context calls the algorithm of all strategies. The strategy pattern is applied when the entity class is composed of generic components. The entity class represents the context. The strategy is represented by a base component class. Concrete strategies are represented by concrete components.

The last mentioned pattern is the *observer pattern* (see figure 11) (Gamma et al. 2007, 293-296). The observer pattern defines a *one-to-many dependency* between a *subject* and its *observers*. The subject provides functions to register, remove and notify observers. When the state of the subject changes, all observers are notified. That pattern can be used in the ECS to implement an *event system* (McShaffry et al. 2009, 272-301). The observer pattern has to be adapted to fit the requirements of an event system. Observers register for specific event types. That means another parameter has to be passed to the register function. When a specific event needs to be fired the subject's notify function is called and the event object is passed. All observers that registered for the fired event

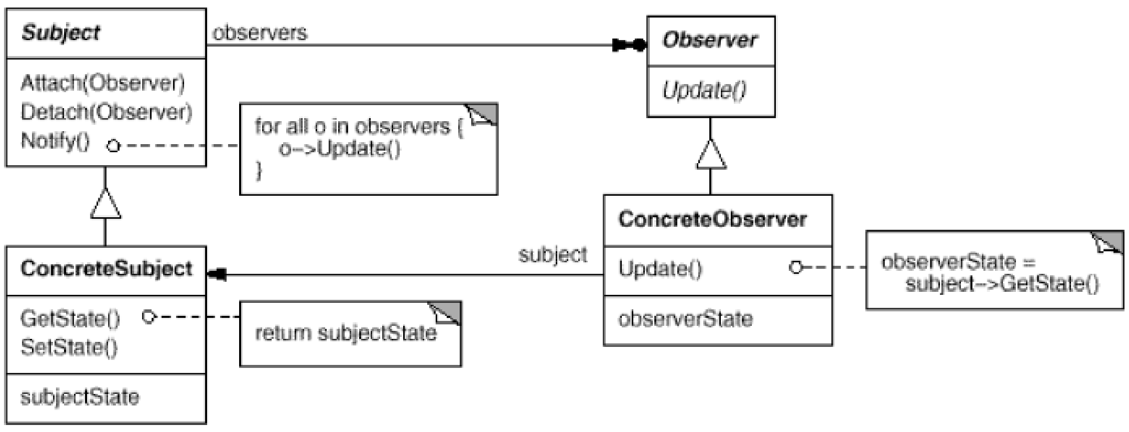

Figure 11: Observer pattern (Gamma et al. 2007, 293-296). Can be used to implement an event system (McShaffry et al. 2009, 272-301).

type will be notified and the event object is passed to registered observers for further processing.

3.4 Drawbacks

The ECS has many advantages due to its CBA. However, there are also disadvantages with the ECS. A system hosts and processes all of its respective components (see section 3.1 Systems: Game Functionality and Organization). That means a system iterates over a list of components every update step. Therefore, a system also iterates over objects that don't require an update because they may be inactive. That costs valuable time. Another problem could be the inner-entity communication (see section 3.2 Communication between Entities, Components and Systems). The component indirection costs time when a component accesses a sibling component. The sibling component has to be looked up. Depending on the look-up functions' implementation it costs more or less time. In an object-oriented architecture the functionality is inherited (Gregory 2014, 873-881). The required data or functionality is in the same object. There is no look-up required. A further point that needs to be considered is memory fragmentation. During a game many entities and components are created and deleted (Gregory 2014, 51-52). When a component is created, it is added to a system which processes components of the same type (Cohen 2010, 34-60). Due to the dynamic creation and deletion of components the memory becomes fragmented (Nystrom 2014, 275-278, 305). The CPU needs to fetch data from different locations in RAM when iterating over a list of pointers to objects which are scattered in RAM (Nystrom 2014, 271-273).

4 Data-Oriented Design (DOD)

Memory fragmentation is a problem DOD can solve (Nystrom 2014, 269-284). The way DOD organizes data prevents memory fragmentation. DOD introduces practices how to manage data in a way that it can be processed more cache-friendly. This thesis examines DOD due to the fact performance increases by just handling data differently. Through contiguous arranging, sorting and splitting data in a cache-friendly way a significant performance boost is achieved.

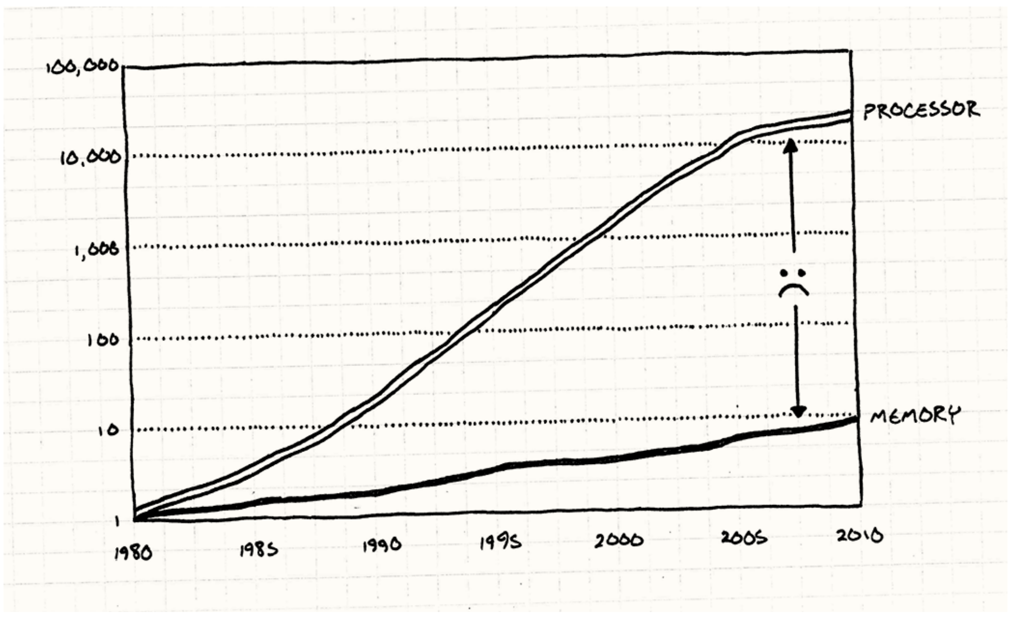

Figure 12: Performance gap between CPU and RAM (Nystrom 2014, 270). The figure displays the speeds of CPU and RAM relative to their speeds in 1980. The figure shows that in 2010 the speed of RAM increased by about 10 times and the speed of the CPU increased by over 10.000 times!

In the last few decades CPU speed constantly increased (Nystrom 2014, 269-270). That means CPUs can incredibly fast process data and perform calculations. Hence, before a calculation can be done, data has to be transmitted from RAM to the CPU. Over the years a performance gap arose between the CPU and RAM (see figure 12). Actually it can take hundreds of CPU cycles to retrieve a data package from RAM. Thus, the CPU spends lots of time waiting for data when data is not organized in a suitable way.

Due to the fact that the CPU has to wait for data from RAM, *caching* was invented (Nystrom 2014, 271-272). Modern CPUs have integrated memory chips. These memory chips are called *caches*. CPUs can access caches much faster than RAM. Modern CPUs like the *Intel Core i7* family have different caches to access and share data within the CPU cores (Levinthal 2009, 4-14). All cores of one *Intel Core i7* CPU socket share a common L3 cache. That L3 cache has a local integrated memory controller which accesses RAM.

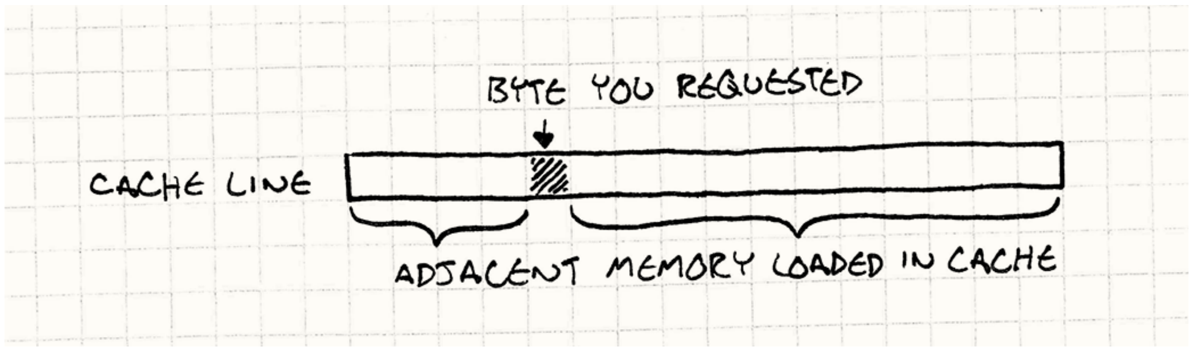

Figure 13: When the CPU requires one byte from RAM to continue processing, it automatically orders a contiguous chunk of memory around that byte and loads it into its cache (Nystrom 2014, 272). This contiguous chunk of memory is called *cache line*.

Each core has a L2 cache and a L1 cache for their own. The access latencies and sizes of a CPU's caches are different. The fastest but smallest cache is the L1 cache. The L2 cache is slower than the L1 but can store more data. The L3 cache is the biggest and slowest of all caches. Every time the CPU needs some data from RAM, it automatically orders a contiguous chunk of memory and puts it into the cache. That chunk of memory is called *cache line* (see figure 13). If the next byte one core needs is in its L1 cache, it directly reads it from there. When the next byte is not in the L1 cache, the CPU core looks up the byte in L2 cache and transfers its cache line to the L1 cache if it was found. That is called a *cache miss*. The same procedure is done with the L3 cache. If the required byte is not in the L3 cache, the L3 cache memory controller requests a cache line from RAM. Which takes many cycles to retrieve. Meanwhile, the *CPU stalls* because it can only continue working when the next piece of data is retrieved. Per cycle four x86 instructions can be decoded and issued. A current Intel CPU like the *Intel Core i7-5960X* can process about 298.19 GIPS (Giga-Instructions per second = 10ˆ9 instructions per second) (Williams 2014). The *Intel Core i7* family specifies the access latencies as shown in table 1.

Data Source	**Latency**
L1 cache hit	˜ 4 cycles
L2 cache hit	˜ 10 cycles
L3 cache hit (cache line unshared)	˜ 40 cycles
L3 cache hit (shared cache line in another core)	˜ 65 cycles
L3 cache hit (modified in another core)	˜ 75 cycles
Local DRAM	˜ 60 ns

Table 1: *Intel Core i7* access latencies (Levinthal 2009, 8-22)

Assume there is a component-based game architecture in which the entity class holds references to its components (see section 3 Entity). There also is an *EntityManager* which manages a global accessible list of entity pointers. When iterating over the list of entity pointers, every entity object has to be transmitted to the CPU's cache (Nystrom 2014, 275-278). Each entity pointer has to be traversed resulting in a cache miss. After

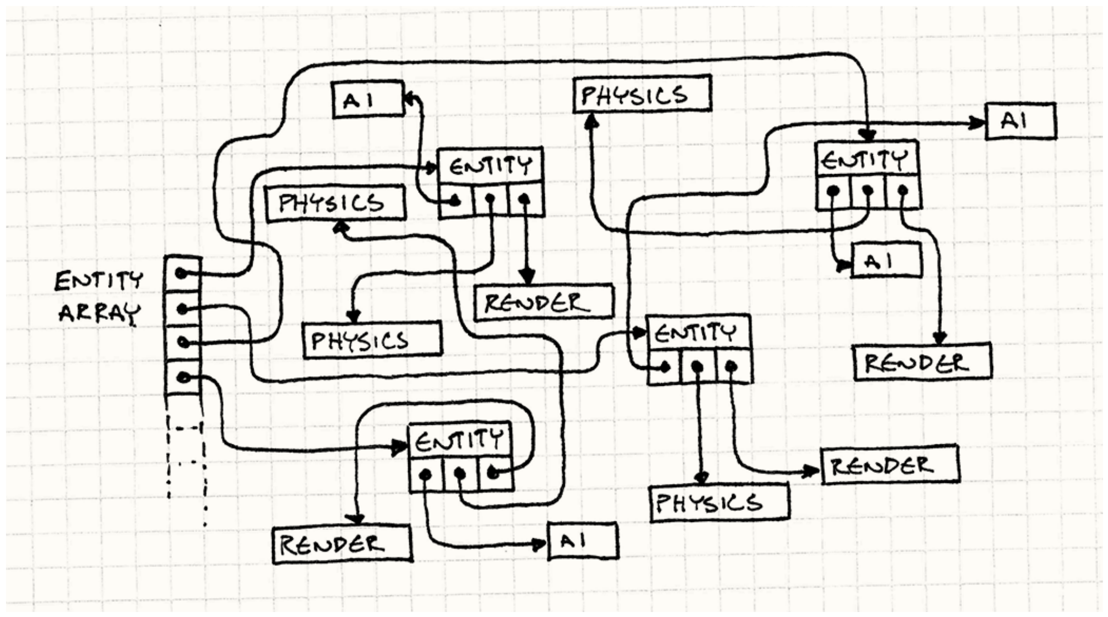

Figure 14: Memory fragmentation due to the incoherently creation and deconstruction of entities and components (Nystrom 2014, 275-278). The CPU has to chase pointers when processing an array of entities like this.

the entity object was fetched and processed, its components need to be processed by the CPU. Since an entity only holds references, not objects to its components, the data of its components also needs to be fetched from another location in RAM by the CPU. For each accessed component reference another cache miss occurs. Due to the incoherently creation and deletion of entities and their components, memory becomes fragmented and cache misses occur frequently (see figure 14).

The goal of DOD is to prohibit cache misses and CPU stalls by organizing data cache friendly (Nystrom 2014, 271-273). In other words, "organize your data structures so that the things you're processing are next to each other in memory" (Nystrom 2014, 273). One side effect of arranging data in contiguous arrays of homogeneous objects is that *parallelization* becomes easier (Llopis 2009). There is a small update function which mainly iterates over the contiguous data. Such a construct can easily split between multiple threads. Another side effect of DOD is *modularity.* When the code base is designed targeting data, its functions end up with no or few dependencies. In the end, DOD should only applied on performance critical code bases (Nystrom 2014, 274). Optimizing code that is rarely used costs more time refactoring than actually paying off.

4.1 Contiguous Arrays

The game loop processes all active entities and their components (Nystrom 2014, 275-280). In a straightforward software design entities hold references to their components. Additionally, there is a global accessible list of entity pointers, which can be iterated.

When accessing and iterating over these entities and their components cache misses occur and the CPU stalls due to memory fragmentation. Instead of creating one main list of entities, one array for each component type is created. The component arrays must hold objects and not pointers to utilize the cache. When initializing an array of objects and not an array of pointers, the entire memory for each object in the array is allocated. These objects are arranged next to each other in memory. When a loop iterates over an array of component objects the CPU can read the next component data from the cache because they are arranged contiguous in memory. That concept prohibits frequent cache misses and saves a lot of CPU cycles.

4.2 Packed Data

After storing component data in contiguous arrays, it can be cache friendly processed (Nystrom 2014, 280-283). The next step is to consider how unused or not activated components of entities are handled. An obvious approach would be to add a *Boolean* flag that determines whether a component is active or not. The component iteration process would check each component whether it is active or not and then processes only active components. That practice would load the flag and its object into the cache. That causes many cache misses when there are many inactive components that are skipped. The entire contiguous array approach becomes useless due to the fact the data in a contiguous array is not contiguous itself. Instead of checking a *Boolean* flag, the objects in an array are sorted by it. The update loop only iterates over active components, which are at the beginning of an array.

4.3 Hot/Cold Splitting

At the current state of cache friendly programming a contiguous array of contiguous data is processed rather rapidly. There is another possibility how the already optimized data can be further optimized (Nystrom 2014, 283-284). Assume there is a component with several member properties. A small set of properties is accessed every update step. The rest is needed occasionally. When the CPU accesses a small collection of properties the entire object and all of its properties are loaded in the cache line. Due to that fact less objects fit into a cache line and cache misses occur more frequently. To prevent cache misses, due to large objects, the data an object holds is split. The data is split into *hot and cold data*. *Hot* data is data that is accessed frequently. *Cold* data is only needed in specific cases. Hence, the component objects holds now only *hot* data and a reference to an outsourced object with *cold* data. Therefore, more small objects fit in a cache line and less cache misses occur.

4.4 Drawbacks

DOD offers obvious advantages due to its cache-friendly organization of data. However, there are also disadvantages which have to be mentioned. The first disadvantage is the lack of polymorphism (Nystrom 2014, 285). Polymorphism is a powerful tool that enables a collection of class instances of different types to be manipulated by a common interface (Gregory 2014, 101). Though data, is arranged in contiguous arrays of homogeneous objects (Nystrom 2014, 285-287). These object arrays are not capable of polymorphism. If an entity would hold a generic array of component pointers the code would be more flexible. Components in the array only would have to implement the interface. The dynamic dispatch would be used to determine which polymorphic operation to perform at runtime. However, the whole cache utilization would be omitted. Another problem occurs when sorting components by their activation state. When moving a component in memory to activate or deactivate it, the pointer of its entity can get broken. Hence, the entity's pointer has also to be updated when its components location in memory changes.

5 The Entity-Component-System and Data-Oriented Design

After the theory about the ECS and DOD, both are combined. This chapter examines an approach of the ECS in connection with DOD. The CBA is combined with practices of DOD to show how to utilize the cache in an ECS. In order to utilize the cache in an ECS the organization of entities and components is mandatory since they are accessed and processed frequently (see sections 3 Entity and 3 Component). Components have to be organized in a cache-friendly way. Practices of DOD are applied on entities and components.

5.1 Components and Contiguous Arrays

The first step to utilize the cache in an ECS is to organize the components in *contiguous arrays* (see section 4.1 Contiguous Arrays). Each component type has an own array of objects (see figure 15). The contiguous arrays of components are stored and iterated in their respective systems (see section 3.1 Systems: Game Functionality and Organization). Due to the fact that an array has a fixed size the maximum amount of components needs to be considered (Nystrom 2014, 305-308). On the one hand, too many available objects in the array waste memory. On the other hand when the array is too small, no additional objects are available. Measurements of the needed components in the game must be taken to estimate the maximum size.

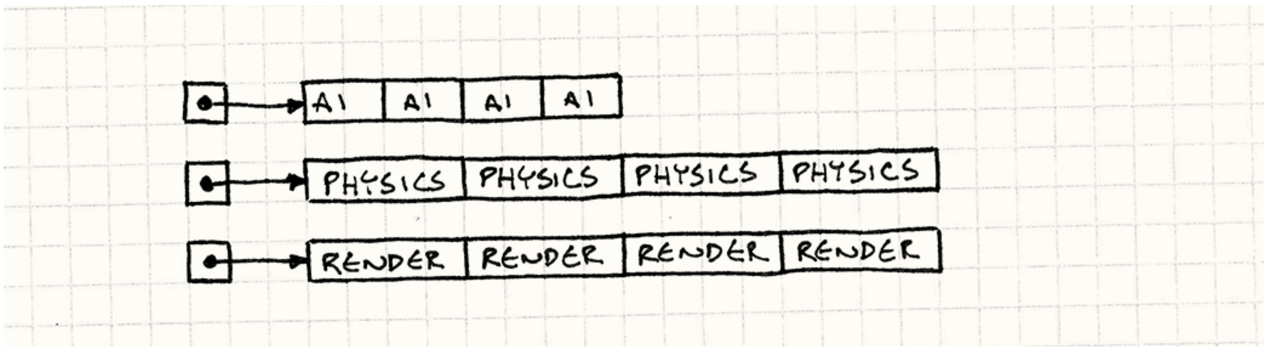

Figure 15: Contiguous component arrays (see section 4.1 Contiguous Arrays). The CPU can rapidly process these arrays because the components in this arrays are stored next to each other in memory.

5.2 Components and Packed Data

After the maximum size of the component arrays is found the activation status of components needs to be considered (see section 4.2 Packed Data). The contiguous arrays of components contain thousands of objects. However, not all the objects are needed at the same time. A system which iterates over its component array has to find a way not to visit all of them. At the start of a game fewer entities and components are typically initialized. As the game goes on more and more entities and components are needed. Thus, a way to separate needed components from available ones is required. A *Boolean* flag is not a suitable way to do so because that leads to cache misses. Therefore, the components are sorted by activation state (Nystrom 2014, 281-283). Active components are in the beginning of the array. There is a counter which determines how many active components are in the array. The associated system iterates only until that counter. Whenever a new component is required, a *swap operation* on a component array is performed. That operation exchanges an inactive component with the first inactive component in the array and increases the counter by one. Hence, the new active component can be associated to an entity. Additionally, whenever a component is deactivated another *swap operation* is performed. In this *swap operation* an active component is swapped for the last active component in the array and the counter is decreased by one.

5.3 Components and Hot/Cold Splitting

The next step in improving the ECS in terms of DOD is to split the components in *hot* and *cold* parts. *Hot/cold splitting* achieves less cache misses due to the separation of frequently accessed data from rarely accessed data (see section 4.3 Hot/Cold Splitting). When designing a component, that practice has to be considered. Assume there is a *PlayerComponent* in an existing game (see figure 16). The *PlayerComponent* has properties like *Name*, *Health* and *Controls*. Such properties are accessed frequently when updating the component. However, the *PlayerComponent* also consist of objects like *ItemCollection*

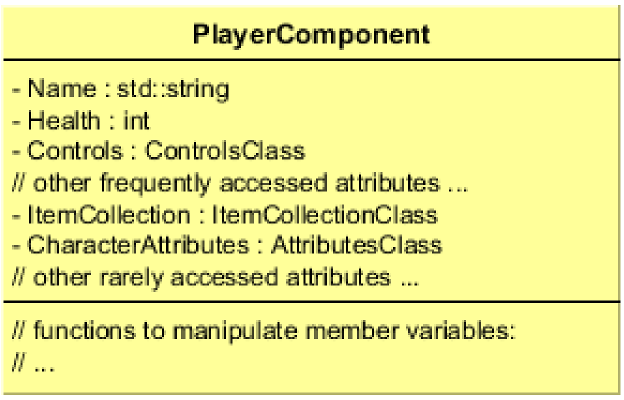
PlayerComponent
- Name : std::string
- Health : int
- Controls : ControlsClass
// other frequently accessed attributes ...
- ItemCollection : ItemCollectionClass
- CharacterAttributes : AttributesClass
// other rarely accessed attributes ...
// functions to manipulate member variables:
// ...

Figure 16: Example *PlayerComponent*. This component consists of frequently and rarely accessed attributes.

and *CharacterAttributes*. That are rarely accessed properties. The *ItemCollection* object is only needed when the player looks in his inventory. The *CharacterAttributes* object is needed when the player is in a fight. When the *PlayerComponentSystem* iterates over its components, the entire data in each *PlayerComponent* has to be fetched by the CPU. The bigger the objects the CPU has to fetch, the more cache misses occur. To minimize cache misses, properties like *ItemCollection* become a pointer or reference instead of an object.

5.4 An Entity is only an Id

Referring to a *pure component model* an entity is only an identifier (Gregory 2014, 886-887). Therefore, the entity class is omitted. There are only component class instances which can identified by the entity unique identifier. Assume there are contiguous arrays for each component type. At some point one component may need to access one of its sibling components. Consequently, the index position of the sibling component in its array has to be determined by the shared parent entity id. A hashing function would be appropriate to perform that task (Gregory 2014, 276-278, 333). The id of the parent entity would be used as a look-up key. Another approach is to organize the component arrays in a way that shared components of an entity have the same index in its array (Nystrom 2014, 289). The problem with both approaches is the sorting of components by their activation state.

6 Conclusion

This thesis examined a software architectural part of the game development process and the organization of data in cache-friendly way. Both parts of this thesis were combined to show how DOD is applied to the ECS.

In game development there are two principle game architectures: OOAs and CBAs. OOAs set up a class hierarchy for game objects. Behaviour of parent classes is inherited to child classes. At first class hierarchies are simple and clearly laid out. However, as they grow they become deep and wide. There arise several problems with deep and wide class hierarchies. The maintenance and modification of these hierarchies become expensive. Multiple inheritance can cause a *deadly diamond* and functionality of classes *bubble-up* and cause *The Blob*. On the other hand, there are CBAs. CBAs extend classes through *composition instead of inheritance*. CBAs split up software into small chunks of task-specific data. That leads to reusable and extendible software. Moreover, the maintenance of software is easy to perform due to the fact a class has no or few dependencies.

The ECS is a CBA which consists of three principle elements: *entities*, *components* and *systems*. The elements of the ECS represent objects in a game world, task-specific data an entity can consist of and game functionality. The ECS is a suitable software architecture for games due to its CBA which provides reusability, extendibility and flexibility in terms of software development. There are different ways entities and components can be designed. Most systems provide a periodically updating service for its respective components. Moreover, additional functionality can be added through callback functions to specific system to extend them. The communication between entities, components and systems is realized with an event system. In an event system different systems can register for specific events, fire events and respond to fired events.

Entities and their components are stored in memory. To process these objects the CPU has to request that data from RAM. The requested data is stored in the caches of a CPU to access that data rapidly. Every time the CPU needs a byte to continue processing, it automatically fetches a chunk of contiguous data around the required byte. That is called a *cache line*. If the next required byte is in the cache, the CPU can fetch the data from there. Accessing the cache is fast. However, when the required byte is not in the cache, the CPU has to request data from RAM. That is called a *cache miss*. Requesting data from RAM is slow compared to accessing caches. When the data of entities or components is arranged as a *contiguous array* of objects, the CPU can read the data from a cache and does not have to frequently fetch data from RAM. The goal of DOD is to organize data in a cache-friendly way. That means organizing data in contiguous arrays of objects, separating frequently accessed data from rarely accessed data and assuring that the objects in contiguous arrays are also contiguous themselves.

DOD enhances the performance in an ECS due to its practices which organize data in a cache-friendly way. The cache-friendly organization of components is an important task when designing an ECS in connection with DOD. First of all, the components are stored in contiguous arrays of objects in their respective systems to utilize the cache. Arrays have a maximum size. Therefore, a maximum size of needed components has to be determined. Due to the fact not all components of one array are needed, they have to be sorted by an activation state. A system only iterates over active components. That lowers the objects that a CPU has to fetch and prevents cache misses by deactivated components. After that the next task is to identify frequently and rarely accessed data and separate it. The separation of frequently and rarely accessed data leads to smaller objects a CPU has to fetch. Therefore, more objects fit into a cache line and less cache misses occur.

There are further topics related to the ECS and optimizations concerning game engine development. One topic this thesis only mentioned is *parallelization*. The goal of parallelization is to utilize the hardware threads a CPU consists of (Gregory 2014, 361-372). There are different approaches how parallelization can be applied to a game engine. One approach is to assign one thread per system. A master thread manages and synchronizes all system specific threads. Another way how parallelization can be applied is through *divide-and-conquer* algorithms. The work is split into small units and is processed by different threads. The results have to be merged when the work is complete. One problem occurs when threads depend on each other and have to wait for one thread which has not finished working yet. To decouple depending threads, a *job system* can be implemented. A *job* is a small unit of relatively independent work. Jobs are managed by a job system and are stored in a queue. Available threads access jobs in the queue and process them. Therefore, threads no longer block each other. Current game engines do all the work by themselves like computing physics, AI and pathfinding (Gambetta 2016). The *Entity-Component-Worker* architecture introduces workers to the ECS and converts each system to a distributed system. Each distributed system consists of many workers that compute components. Physics workers compute physics components and pathfinding workers compute pathfinding components. Therefore, the size and content of a game world is no longer limited to processing power of a single sever. Hundreds of workers simulate different parts of the world.

To sum up, the ECS is a CBA which enhances the game development process in terms of software architecture and development. DOD improves the ECS in terms of a cache-friendly design. Both aspects, software architecture and data-oriented software development, are a major part in improving the game development process.

Abkürzungsverzeichnis

CBA	Component-based architecture
DOD	Data-Oriented Design
ECS	Entity-Component-System
FPS	First-person shooter
MVC	Model-View-Controller
MVP	Model-View-Presenter
OOA	Object-oriented architecture

List of Figures

Listings

List of Tables

References

Allan, B. A., R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, et al. 2006. "A Component Architecture for High-Performance Scientific Computing." *International Journal of High Performance Computing Applications* 20 (2): 163–202. ISSN: 1094-3420, accessed May 20, 2016. doi:10.1177/1094342006064488.

Bilas, Scott. 2002. "A Data-Driven Game Object System." Slides presented at the Game Developers Conference (GDC) 2002, San Jose, California, March 18-23. Accessed March 5, 2016. http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides_with_notes.pdf.

Cohen, Terrance. 2010. "A Dynamic Component Architecture for High Performance Gameplay - Insomniac Games." Slides presented at the Game Developers Conference Canada (GDC Canada) 2010, Vancouver, BC, May 06-07. Accessed March 3, 2016. http://d3cw3dd2w32x2b.cloudfront.net/wp-content/uploads/2011/06/6-1-2010.pdf.

Collin, Daniel. 2014. *Introduction to Data-Oriented Design.* Accessed March 3, 2016. http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf.

DeLoura, Mark. 2009. *Gamasutra: Mark DeLoura's Blog - The Engine Survey: Technology Results.* Accessed May 24, 2016. http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php.

Entertainment Software Association. 2015. *ESSENTIAL FACTS ABOUT THE COMPUTER AND VIDEO GAME INDUSTRY.* Accessed March 11, 2016. http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf.

Entity Systems Wiki. 2014a. *What's an Entity System? - Entity Systems Wiki.* Accessed May 3, 2016. http://entity-systems.wikidot.com/.

Entity Systems Wiki. 2014b. *ES Terminology - Entity Systems Wiki.* Accessed May 15, 2016. http://entity-systems.wikidot.com/es-terminology.

Entity Systems Wiki. 2014c. *Rdbms With Code In Systems - Entity Systems Wiki.* Accessed May 15, 2016. http://entity-systems.wikidot.com/rdbms-with-code-in-systems.

Gambetta, Gabriel. 2016. *Gamasutra: Gabriel Gambetta's Blog - The Entity-Component-Worker architecture and its use on massive online games.* Accessed May 31, 2016. http://www.gamasutra.com/blogs/GabrielGambetta/20160425/271221/The_EntityComponentWorker_architecture_and_its_use_on_massive_online_games.php.

Gamma, Erich, Richard Helm, Johnson Ralph, and Vlissides John. 2007. *Design Patterns: Elements of Reusable Object-Oriented Software.* 34. printing. Addison-Wesley professional computing series. Boston: Addison-Wesley. ISBN: 978-0201633610.

Gregory, Jason. 2014. *Game Engine Architecture.* 2. ed. Boca Raton Fla. u.a.: CRC Press. ISBN: 978-1-466-56001-7.

Hight, John, and Jeannie Novak. 2008. *Game Development Essentials: Game Project Management.* Clifton Park, NY: Thomson Delmar Learning. ISBN: 9781418015411.

Levinthal, David. 2009. *Performance Analysis Guide for Intel® Core^{TM} i7 Processor and Intel® Xeon^{TM} 5500 processors.* Technical report. Intel Corporation. Accessed May 5, 2016. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

Llopis, Noel. 2009. *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within.* Accessed March 3, 2016. `http://gamesfromwithin.com/data-oriented-design`.

Madhav, Sanjay. 2013. *Game Programming Algorithms and Techniques: A platform-agnostic approach.* Upper Saddle River, NJ: Addison Wesley. ISBN: 9780321940155.

McCormick, Hays W. 1998. *Development AntiPattern: The Blob.* Accessed May 26, 2016. `http://antipatterns.com/briefing/sld024.htm`.

McShaffry, Mike, James Clarendon, Jeff Lake, Quoc Tran, and David Graham. 2009. *Game Coding Complete.* 3rd ed. Australia, United States: Charles River Media/-Course Technology Cengage Learning. ISBN: 9781584506805.

Nystrom, Robert. 2014. *Game Programming Patterns.* s.l.: genever benning. ISBN: 9780990582908.

Papari, Adrian. 2016. *junkdog/artemis-odb.* Accessed May 8, 2016. `https://github.com/junkdog/artemis-odb`.

Potel, Mike. 1996. *MVP: Model-View-Presenter: The Taligent Programming Model for C++ and Java.* Technical report. Taligent, Inc. Accessed May 15, 2016. `http://www.wildcrest.com/Potel/Portfolio/mvp.pdf`.

Ritchie, Dennis. 1993. *The Development of the C Language.* Accessed May 8, 2016. `http://csapp.cs.cmu.edu/3e/docs/chistory.html`.

Schmid, Simon. 2016. *sschmid/Entitas-CSharp.* Accessed May 8, 2016. `https://github.com/sschmid/Entitas-CSharp`.

Unity Technologies. 2016a. *Unity - Scripting API: GameObject.GetComponent.* Accessed May 15, 2016. `http://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html`.

Unity Technologies. 2016b. *Unity - Manual: Tags and Layers.* Accessed May 24, 2016. `http://docs.unity3d.com/Manual/class-TagManager.html`.

Unity Technologies. 2016c. *Unity - Manual: Using Components.* Accessed May 26, 2016. http://docs.unity3d.com/Manual/UsingComponents.html.

Williams, Rob. 2014. *Core i7-5960X Extreme Edition Review: Intel's Overdue Desktop 8-Core Is Here.* Accessed May 5, 2016. http://techgage.com/print/core-i7-5960x-extreme-edition-review-intels-overdue-desktop-8-core-is-here/.

Anhang

Datensets

Archived Web Pages

List of Figures

http://web.archive.org/web/20160526143921/http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013

http://web.archive.org/web/20160526144132/http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

http://web.archive.org/web/20160526144213/http://gameprogrammingpatterns.com/data-locality.html

http://web.archive.org/web/20160526144326/https://upload.wikimedia.org/wikipedia/commons/b/b3/SingletonUML.png

References

http://web.archive.org/web/20160526144551/http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides_with_notes.pdf

http://web.archive.org/web/20160526144637/http://d3cw3dd2w32x2b.cloudfront.net/wp-content/uploads/2011/06/6-1-2010.pdf

http://web.archive.org/web/20160526144717/http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf

http://web.archive.org/web/20160526144741/http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php

http://web.archive.org/web/20160526144828/http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf

http://web.archive.org/web/20160526145037/http://entity-systems.wikidot.com/

http://web.archive.org/web/20160526144858/http://entity-systems.wikidot.com/es-terminology

http://web.archive.org/web/20160526144956/http://entity-systems.wikidot.com/rdbms-with-code-in-systems

http://web.archive.org/web/20160603152701/http://www.gamasutra.com/blogs/GabrielGambetta/20160425/271221/The_EntityComponentWorker

http://web.archive.org/web/20160526145347/https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

http://web.archive.org/web/20160526145419/http://gamesfromwithin.com/data-oriented-design

http://web.archive.org/web/20160526194501/http://antipatterns.com/briefing/sld024.htm

http://web.archive.org/web/20160526145525/https://github.com/junkdog/artemis-odb

http://web.archive.org/web/20160526145631/http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

http://web.archive.org/web/20160526145723/http://csapp.cs.cmu.edu/3e/docs/chistory.html

http://web.archive.org/web/20160526145823/https://github.com/sschmid/Entitas-CSharp

http://web.archive.org/web/20160526145933/http://docs.unity3d.com/Manual/class-TagManager.html

http://web.archive.org/web/20160526150015/http://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html

http://web.archive.org/web/20160526175528/http://docs.unity3d.com/Manual/UsingComponents.html

http://web.archive.org/web/20160526150057/http://techgage.com/print/core-i7-5960x-extreme-edition-review-intels-overdue-desktop-8-core-is-here/