Component-Based Entity Systems

Modular Object Construction and High Performance Gameplay

Faculty of Arts
Department of Game Design

Olof Wallentin
Degree Project in Game Design, 15c
Supervisors: Hans Svensson, Mikael Fridenfalk
Examiner: Masaki Hayashi
2014-06-08
Abstract

This bachelor thesis examines design implementation and differences between game entity systems, with a focus on a component-based structure. How each will affect the other will be discussed from both a technical and design point of view, including possible drawbacks or advantages regarding game design iteration and performance.

Since the focus is on component-based entity systems, a clarification on traditional entity systems are required, thus this thesis focuses on entity systems that are traditional, property-based, container-based, and aggravated component-based.

The design and implementation of each system was founded from different generations of programming paradigms which resulted in specific compositional structure based on each specific era of hardware configuration. This thesis analyses the progress of hardware alongside game entity system design to further understand its progression and evolution into today’s standards and implementation. Details on each system is provided from a design perspective for the traditional entity system and with an in-depth view for the component-based entity systems.

Keywords

Programming, game object, entity, C++, component, game design.
## Contents

1 Introduction........................................................................................................... 4

2 Theory .................................................................................................................... 4

3 Purpose .................................................................................................................. 5

4 Materials and Methods ......................................................................................... 5

5 Terminology ........................................................................................................... 6

6 Overview ................................................................................................................. 7

   6.1 Traditional Entity System ................................................................................. 7

       6.1.2 General Design Overview ....................................................................... 7

       6.1.3 Entity Deep Hierarchies .......................................................................... 7

       5.1.4 Deep Complex Entity Hierarchies ......................................................... 8

       6.1.5 Composition ............................................................................................ 10

6.2 Component-Based Entity System ...................................................................... 11

   6.2.1 General Design Overview ........................................................................... 11

   6.2.2 Entities, Components and Properties ....................................................... 12

   6.2.3 Pure Containers .......................................................................................... 13

   6.2.4 Entities as Pure Aggregation ....................................................................... 14

6.3 Data-Driven Programming .................................................................................. 15

6.4 Resistance ........................................................................................................... 15

7 Analysis ................................................................................................................... 15

    7.1 Traditional Entity System .............................................................................. 16

       7.1.1 Creation .................................................................................................. 16

       7.1.2 Communication ...................................................................................... 16

       7.1.3 Performance ........................................................................................... 16

7.2 Component-based Entity System (Properties) ................................................... 17

    7.2.1 Creation .................................................................................................... 17
1 Introduction

Due to the ambition to create larger and more immersive games, the need for flexible and powerful game engines continues to grow. To meet the needs of developers around the world, the industry is continuously changing the way engines are designed. Since modern game engines are made to power a variety of different genres, it is required to feature a modular design at its core, and this is where component-based entity systems come into play.

In the past, game entities were programmed specifically for its sole purpose, making code reuse and design changes difficult to perform. A component-based entity system attempts to change this approach by constructing game entities from a set of modular components, thereby removing, and adding or altering components can create new game entities at the blink of an eye.

With this in mind, the approach to implementing a component based entity system in itself becomes a modular task. This idea is relatively new and has only been around for years, so many developers have their own vision of its design. It is not only performance versus modularity that is a hot topic, but also communication between components, which essentially makes the game entity whole. All of these questions will be examined, researched and discussed in this thesis, as well as examples of implementations by myself and other professionals alike.

While modularity is important for flexibility, so is performance, and the question arises on how to implement a component-based entity system which favors both of them. A variety of game engines today, such as Unity, has taken this approach into account and sacrificed one for the other. Is it possible to have both? If so, what are the consequences, and how will the implementation look?

Another question which has become relevant in recent years is whether we now can sacrifice performance for the sake of modularity. Progression of processing power has enabled high-level languages like C# and Java to be a part of the game industry and many engines have started using them as the main source of communication.

2 Theory

Hardware continues to evolve and the way software is designed is changing along with it. Designing systems for managing and creating game entities has been the same since object-oriented-programming was introduced in the game industry. In a way it can make logical
sense to construct an entity from an abstracted real world point of view, but when game applications become larger, more performance demanding and the need for iteration and alterations are requested, the need for change is required.

One of the answers to the calling is component-based entity systems which is, compared to traditional entity systems, designed from a data driven perspective, which construct entities from pure components. With components as the building blocks, it becomes easy to switch, add or remove parts of a game entity and thus, changing its behavior completely.

If implemented properly, component-based entity systems can also be more performance effective in memory allocation and usage compared to traditional entity systems.

3 Purpose

The main purpose of this thesis is to provide a discussion, analysis and theoretical implementation of a component-based entity system. In order for this to be possible, this thesis also has a second purpose.

The second purpose is to analyze and discuss various game entity systems. It is to provide examples of why certain systems are designed in a certain way, how it can affect performance and overall project structure, re-design and iteration. This is required in order to understand why component-based entity systems are designed the way they are, and why it will bring benefits to a game project.

As iteration and change is an important rule for game design, a set of questions concerning this topic is investigated in this thesis.

1. How do we create a modular game entity which is easy to change?
2. Define what easy means within the definition of creating and modifying game entities.
3. Can we increase performance of game entity execution and memory access?
4. What type of system might be suitable in what situation?

4 Materials and Methods

Because the subject that this thesis discusses is relatively new to the industry, few books have been written about it, and more practical implementation examples such as post mortems and game developer conference presentations are more valid research materials. To provide more
data for the discussion, the author of this thesis has created several implementations of component-based entity systems himself and is working at a company were the in-house engine is using such a system.

5 Terminology
Terminology used in this thesis which is specific to game programming or general game development.

**Debug/debugging:** Action performed by a programmer trying to locate problems in the code or analysis.

**C++:** The most common programming language used for game development.

**Entity:** Sometimes referred to as game object, a container of behaviors and functionality which define how certain things operate within a game world.

**Class:** The definition of an object or entity. It contains member functions and data which is required to perform the requested operations.

**Polymorphism:** When a class inherits from another class, building upon or extending existing features.

**Array:** A block of memory which contains a set number of similar data types in a consecutive order.

**Cache miss:** When the processor is trying to access memory, fails to find the requested location but is retrying until a correct result is returned.

**Unity:** Free and commercial cross-platform game engine. Uses a component-based entity system.

**Standard data type:** standard data types are variables that exist by default in the programming language. Examples for standard data types are floating point numbers, integers and characters.

**Pointer:** A standard data type in C++ which holds the address of a memory location.

**Overhead:** This is a term used when a feature is extended for additional functionality, but causing it to operate slower.

**Stack:** A location on the computer which enables fast memory allocation but is small and therefore mostly used for temporary objects.
6 Overview
The following describes the thinking and decisions made behind several game entity systems.

6.1 Traditional Entity System
A traditional entity system is simply often referred to as entity system and is based on classic object oriented programming with deep hierarchies. Entities are created as a single pre-defined entity containing the necessary data required to perform its requested operations and inherits functions and/or data from interfaces or other entities. It is common that entities made with a traditional system is composed by using a mixture of “is-a” and “has-a” design [4].

6.1.2 General Design Overview
From a programming perspective, each game entity will be named according to what it represents and includes, in an abstract, the type of objects and data requested by the game design. As such, an example of game entities in a Super Mario game, can by name be the following.

- Mario (The player entity)
- Gomba (Enemy entity)
- Red Mushroom
- Block

The picture to the right displays these game entities as an active part of a game. Entities like these are often hard-coded chunks of code created for a single purpose. To enable different features of the entity such as collision or physics, an interface is inherited which contains member functions which will enable the programmer to further specify what will happen during those stages in the game for this particular entity [6].

6.1.3 Entity Deep Hierarchies
Traditional entity systems rely on classic object oriented programming, and as such, large games with many different entities will inherit from huge number of interfaces and classes. The following is a diagram which shows the natural progression of inheritance hierarchy.
Logic dictates that every game entity is, in its original form, an “Entity”, which every foundation derives from. As such, variations are defined, the higher the hierarchy goes, such as the Mario and the Luigi entity which derives from Player. In the example tree above it is certain that major differences exist between both the Mario and the Luigi entities, which is why there are two specific implementations.

In the game Super Mario Bros, the only difference between Mario and Luigi was visual, i.e. the graphics. This made it unnecessary to create specific entities like the example hierarchy above. Instead, the Player entity is created with different initialization data. Two identical entities are kept in our game with the only difference consisting of a color value and an identification value for a joystick.

Later, in Super Mario Bros. 2 there are major differences in not just graphics but in behavior and control for Mario and Luigi. This makes it more ideal to create two unique entities which derives from a base Player class. The base Player class will have simple functions for input as this is a feature which is required by all entities controlled by a player.

5.1.4 Deep Complex Entity Hierarchies

Since games are real-time applications, and speed and performance are vital parts of the development process, we want a game entity to do as little useless work as possible. In the deep hierarchy tree, every entity has the basic properties of rendering, collision and physics.
If we want to increase the abstraction level of some of these parts, since certain entities will not be affected by physics but intended to be rendered in the scene, it is possible to inherit interfaces or classes which will enable the programmer to add specific functionalities.

As specified in the tree above, the Mario and the Luigi entity both derives from the Player entity, just like before. Since the Player entity will be rendered on the screen, it also encapsulates the IRender interface which enables render functionality. If the IRender interface was removed from the Player, Mario would no longer be rendered on the screen.

IPhysics provides a physics interface so the entity is affected by gravity and the ICollision interface adds collision events for other ICollision entities. Because the coin in Super Mario Bros. 3 floats in the air, it is clear to assume that if this game used a complex entity hierarchy system, it would not inherit the IPhysics interface. Doing so would make the coin fall to the ground.

No matter if the game entity system is using a complex or deep hierarchy, every base entity contains a member function often referred to as update [7]. It is in the update function that everything specific about our game entity is being handled. The class of the base entity update function is declared as virtual or pure virtual, meaning it exists as a template for derived classes where the programmer will implement its functionality.
In the deep hierarchy example of entity -> player -> Mario, the player class can implement basic functionality such as input. The Mario class will then call on its player base class update function and proceed with its own update afterwards.

In a simple deep hierarchy system every entity is expected to render on the screen, which means that the base entity class also has a member function called render. When we want an entity to not render, we simply do not implement the draw function or add a condition for that functionality. This also means that no matter what, the draw function will always be called for every entity, even if it is not used. This is what the complex hierarchy system is trying to optimize by placing each inherited interfaces into separate containers and running them individually.

6.1.5 Composition

When using deep hierarchies for constructing game entities in a modular fashion, a common term known as the diamond of death often occurs [7]. This is when an entity is created in a deep hierarchy which inherits from an interface which itself inherits from the same interface. This makes the entity inherit from itself and will causes all sorts of problems. To use hierarchies and inheritance in traditional entity systems, compositions is sometimes a preferred suggestion [2].

It is possible to take complex deep hierarchy trees further and make multiple inheritance interfaces in the following way.

![Diagram of deep hierarchy with entities and interfaces](image)

**Figure 7:** The diamond of death, a common problem when using too much multiple inheritance
In this example we can see that Player inherits the IJump interface, and then we have another interface called IDoubleJump which inherits from IJump. Because we want both Mario and Luigi to be able to Jump, it is logical to make the Player class derive from IJump. However, we want to give Mario the ability to double jump so this class inherits from IDoubleJump, which in its own turn inherits from IJump, but Player already inherits from IJump, which IDoubleJump also inherits from. This is called a diamond of death.

When the diamond of death becomes a problem, the programmer can go back and make changes to entity inheritance. Another design approach is to rely less on multiple inheritance and instead create entities using composition [8].

With composition the programmer no longer uses multiple inheritance to the degree as seen in the IDoubleJump diamond of death example. In adding new features for game entities, a class object is added as a member variable, which can be the class DoubleJump. With this approach DoubleJump can still derive from Jump and even if Mario already has an instance of Jump, it can be overwritten by C++ variable inheritance rules, or simply ignored by the programmer.

6.2 Component-Based Entity System

The component-based entity system design is based on entities as containers or pure aggregations of components [4].

6.2.1 General Design Overview

When creating game entities, each functionality is kept within a component, and as a component is added to an entity, that functionality is also added. Entities themselves are nothing more than a simple container or identification which keeps all the components together.

- Render Component
- Physics Component
- Transform Component

The example components above will provide the entity with the functionality as described in the component name. By adding a render component to an entity, it will render a mesh within our scene and a physics component makes the entity react to the laws of physics, such as gravity. A transform component provides the entity with a position, rotation and scale within the world. Some components will require each other in order to function. In the case of
Vitei’s Usagi engine, a physics component will not function without a transform component but a render component can render a mesh without a transform component, which will render that mesh at position zero.

In the game engine Unity, which uses a component based entity system, each entity is automatically given a transform component which cannot be removed. This is a design decision which will make the overall communication between components simpler because entities in a scene do need a position in order to exist and operate properly.

6.2.2 Entities, Components and Properties

A common way to start creating a component-based entity system is to create an entity as an organized blob [4]. The entity is made out of sub-objects, common or unique properties which can be enabled or disabled, added or removed. If certain sub-objects are present or not, is verified dynamically at run-time with a null check before execution. With this approach it is possible to narrow down entity properties to what is actually needed to perform the functionality which is intended, and as each base is the same, the added sub-objects will construct the functionality with no need for deep hierarchies or polymorphism. A common design implementation using sub-object properties can look like the following.

```cpp
if (update)
    update->run(deltaTime);
if (physics)
    physics->run();
if (render)
    render->run();
```

The implementation example above is done using C++ syntax and checks if the corresponding sub-object property (update, physics, render) exists, and if so, executes its member function run.

While it is correct that the entity itself does not utilize polymorphism for deep hierarchies, the sub-object properties will have to in order for this design implementation to function properly. It is also important to note that this design will create processor overhead for each entity as it has to perform an if-case verification upon sub-object property execution.
6.2.2.3 Internal Structure

Besides the standard sub-objects (i.e. update, physics and render), this component design also includes a template container for member variables. Data for the entity can be integers, floating point values or characters. This is generated upon entity creation and set in an initialization phase where it is stored on the entity stack or can be received dynamically during an update. If not stored on the stack, the data is associated with a unique identifier such as a string. In doing so, the base for the entity will still be managed the same way as before, only the sub-object handles it differently.

6.2.3 Pure Containers

Another way to create entities with components is to store them in a dynamic list as a member variable for the base entity class. Each game entity then has an update member function which is executed and iterates through all components, executing its respective update member function as well.

```
Base Class Entity
<List of components>

Grab first component in the list  Update component

Grab next component in the list

Figure 8: Component update logic for the container design
```

In order for this implementation to work, each component must derive from a base component class and are therefore dependent on polymorphism. The entity becomes more or less a manager for its components, and components are aware of what entity is the owner. Since the entity works like a manager, knowing who owns who will make component removal and creation of other components simpler.

A game entity is defined by what types of components that have been added to the component list. Removing or adding components will alter the behavior of the entity.
6.2.4 Entities as Pure Aggregation

In this system design the concept of entities no longer exist. There is no such thing as a base class, a component list container or sub-objects, but only pure components. The only thing that binds them together is a single integer value.

<table>
<thead>
<tr>
<th>Component Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
</tr>
<tr>
<td>Render</td>
</tr>
<tr>
<td>Physics</td>
</tr>
<tr>
<td>Script</td>
</tr>
<tr>
<td>Target</td>
</tr>
</tbody>
</table>

(Playe) ![image](image)
(Enemy) ![image](image)
(Missile) ![image](image)

*Figure 9: Shows what defines a specific entity*

In figure 9 three entities are defined by their components. It is important to note that an entity called Player does not exist. It is only the components and the data that define the Player instance.

Each of the component types (Position, Render, Physics, etc...) have their respective system which executes that particular component implementation [3]. With aggregation, components are updated per type, not per entity. This means that, for example, all active physics components are updated one after another, instead of updating an entity and then all of that particular entity’s components.

When constructing gameplay using aggregation, a specific system interface is created for this purpose. In figure 9 there is a component called Target, so in the engine there is also a system called Target System which will run for all entities that are connected to this component. The way game entities are constructed is by picking an index value in a large array and associate those components with that integer value.

In Vitei’s Usagi engine data structures for inputs and outputs are used to modify behaviors. Each system is capable of running code which can read or alter the data of components.
Whenever an entity has been updated or created, the associated system gets the required component so they can be used when the system is executed.

6.3 Data-Driven Programming
Even though component-based entity systems are founded on the basis of components, the fundamental idea comes from data-oriented programming [6]. With data-driven programming, functionality is composed of pure data instead of wrapping the required compositions into a class. The main reason for this approach lies within the bottlenecks of today’s hardware. As games tend to require powerful graphic cards to perform advanced real-time rendering, the processor is often used for optimization and cull rendering so the necessary entities can render in the fastest possible way. Updating game entities is, unless advanced artificial intelligence is present, a simple operation. The main issue is with memory access and cache misses. With data-oriented programming variables are stored in large block and accessed from start to end. This type of operation is friendly for the processor and minimizes cache misses. The aggravated component-based entity system is using this very fundamental implementation to optimize processor usage and memory access.

6.4 Resistance
Going away from something we are familiar with into new territory can be scary for some people. It has been reported that there are indeed resistance met from many when talking about switching from a traditional entity system to a component-based one [3,4].

*Since the code relied on the game objects being game objects, and very particular game objects at that, it proved to be a lot of work to make everything work as components.*

– Mick West

This experience is taken from the rewrite of the engine used in *Tony Hawk* games. Mick West also mentions that the executives at the company were strongly against the engine rewrite because it would consume unnecessary time when everything was already working anyway.

7 Analysis
This analysis will examine how entities are created, how they can be changed, and how performance is considered at the execution of their associated member functions. Since the goal is to create easily modifiable game entities, the approach for change is considered from the perspective of a designer.
7.1 Traditional Entity System

7.1.1 Creation

Entities are created as a disorganized blob which commonly contains bloated data and unused member functions due to deep hierarchies and multiple inheritance. Attempts to make it more modular with multiple inheritance has been made, but ultimately created more problems with the diamond of death. Even if the strategy with multiple inheritance is successful, a programmer will still be required to change or alter anything within the game entity. Some game engines allow for specific entity data (such as walk speed, jump length etc.) to be loaded from a text document outside of the compiled code. While this will enable the designer to alter how certain functionalities will behave for the entity, the core still remains closed and only open to a programmer. Even for a programmer, making changes can be a difficult task depending on how disorganized the blob has become. Entities in some games with no structure at all can be several thousand lines of code long, and finding or debugging problems in such a case is rarely an easy task.

7.1.2 Communication

Since most of the functionality is hard-coded, so can the communication between composite parts also be. It is a simple task (and good for performance) to send a pointer of an entity to another entity and simply directly access it from anywhere. One issue with this approach is the term dangling pointers, which occurs when an entity is deleted, but pointers still point to the memory location, which no longer exists. Accessing it will cause a crash unless taken care of manually. A solution to this is to use a message system, such as the observer pattern [2]. With a generic message system, entities can register and listen to specific messages which can notify when entities have been deleted, thus taking care of dangling pointers and other functionalities, such as multiple entity notifications for behavior changes.

Message systems can also become cumbersome the more entities that are present in the game. A possible solution is to simply not register every entity with the message system but doing so obviously removes that functionality as well as forcing the programmer to manually register entities in the code.

7.1.3 Performance

Entities can be created and stored in a single large chunk of memory, iterated from start to finish. As long as the entity do not reference pointers from other blocks and member variables are created on the stack, it becomes efficient from the point of view of memory
management. This is because the processor will read memory from beginning to end instead of jumping around and access different parts, which can cause cache misses.

7.2 Component-based Entity System (Properties)

7.2.1 Creation
The main difference from the traditional entity system and properties is that properties is more organized and less bound to the diamond of death, because systems are separated into sub-objects. However, it still requires a programmer to change any of the fundamental behaviors of the entity. This puts properties in the same category as traditional entity systems when it comes to being designer friendly.

7.2.2 Communication
Since the entity is separated into several sub-objects, communication cannot be as hard-coded as traditional entity systems. For example, if the update sub-object would like to communicate with the render sub-object it would either have to keep a pointer to the entity or send a message using a message system. In the traditional entity systems, everything is a large chunk, so if the programmer wanted to do the same thing, it can be done within the same class, instead of referencing an entity located somewhere else in memory.

7.2.3 Performance
The first overhead caused by properties is the initial null-verification which must be done for each entity to check if it has a system present in any of the sub-objects. It is not a major deal for modern processors with cache prediction but still something to consider. Since each sub-entity is forced to use polymorphism it must be allocated on the heap so memory will be scattered and accessed randomly.

7.3 Component-Based Entity System (Container)

7.3.1 Creation
Since the functionality of this system is based on what components that lie in the entity list, the creation process can be managed by a script or a simple text file. It is possible to hand this script or text file to a designer and let that person add or remove any components to experiment with different behavior. Compared to previous entity systems, this component-based container system is the first that actually enables a non-programmer to create and change entities without touching a single line of C++ code.
7.3.2 Communication

Some game engines such as Unity allows the designer to write scripts which will define the behavior based on the active components. In this case, certain component behaviors are programmed in pure C++ code which will enable the entity to render, be affected by gravity or have a position in the scene. The actual functionality is made possible by writing what the components will do.

Cistron, another component-based engine that is managed purely in C++ code and the way it affects component behavior is by using a get component function. This will search a list of components and get the requested type each time it is performed. It is also possible to make it store a pointer to a component instead of searching for it each update cycle, but doing so can lead to dangling pointers if the component has been removed. Searching and null checking is a safer way, but requires more processor power.

Finally it is also possible to use a message system to communicate to other components what it should do. Doing so enables communication without components actually being aware of other components. For example, the physics component can send a message that it has moved certain steps in the simulation to the transform-component which translates the entity to a new position.

7.3.3 Performance

The container type of component-based entity system makes every entity function as a manager for its components. This means that when updating every entity it first accesses the entity itself to be able to access its components to be able to update them properly. Compared to the aggravated system which updates every component type after each other, components in the container are updated based on their owner (i.e. the entity). Not only will the memory access jump back and forth, but components are updated in a random order, making it difficult to predict logical behaviors of entities.

7.4 Component-Based Entity System (Aggravated)

7.4.1 Creation

Creating an entity with the aggravated system is done by creating components using an integer identification. The integer is what is linking the components together, effectively creating a modular game entity. Although components themselves are created in a low-level programming language such as C++, scripting languages can also be used for certain functionalities.
The creation process itself can be abstracted to a subsidiary toolset, such as a graphical user interface or a text file suitable for a designer. This will enable the designer to create game entities without the need to touch any code at all. While the designer cannot change the internal configuration, it becomes a simple task to experiment with different components to create new game entities.

7.4.2 Communication
Since the only thing keeping a set of components together is an integer value, communication can be tricky. One possibility is to wrap the integer identification into a data structure and send messages to all components associated with it. If the component also knows who the owner is, it becomes possible to get a component and make modifications from there.

When getting input and output data in Vitei’s usage engine the associated system performs this operation once whenever the entity has been created or updated. It is possible in this case to get components from other entities such as parents or children and modify them directly.

7.4.3 Performance
Memory for components can be allocated at startup and in one large array. This makes it easy to iterate through and is friendly for the processor cache. The integer identification value is also an index value which specifies which component in the array it belongs to. This means that when creating the first entity, its value will be zero and every component in every component array at index zero belongs to that entity if it has been created.

While it can be argued that this approach is wasting memory since entities will not use every component in its designated index, however, the speed gained through effective memory iteration is enough to justify this approach. Game entities usually use pre-allocated memory arrays in any way and by having already defined all components for it, removing or adding new components becomes an effective operation.

7.5 System differences
The follow diagrams represents a visual implementation of the different systems.

7.5.1 Traditional Entity System
The basic implementation of the traditional entity system is composed of large blocks of code, one after another or intervened into each other, built into the entity and everything in this chunk is defining its functionality.
7.5.2 Component-Based Entity System

Although there are variations between different component-based entity systems, they are all formed from the base of building behaviors by combining components rather than hard-coding everything into one large chunk of data.

The main purpose of this system design is to be able to remove or add new components during runtime, thus changing the behavior of the entity. Since all of the components are class objects themselves, they won’t create conflicts with other components and it becomes a trivial task to experiment with different entity behaviors.

If we also look at the traditional entity system, every functionality is hard-coded into one large chunk, here we split everything into smaller sub-objects.
8 Conclusion

8.1 Modular Entity Creation
One of the questions asked in this thesis was how we can create a modular entity which is easy to change. By definition, it becomes easy to change an entity if it is constructed using components rather than putting everything in a large chunk of code. To do this, we can use a container or aggravated-based component systems. The reason for this is because a traditional entity system creates a disorganized blob which doesn’t isolate its functionality.

When the functionality of a game entity becomes advanced it is important to approach it with a modular implementation. Creating such an entity as a disorganized blob can quickly become a problem, not only for performance but for iteration. The larger it is, the more difficult it becomes to change, and altering any behavior can also break another functionality somewhere else.

When separating behaviors and functionalities into components and combining chunks of data, it becomes easier to remove or add additional behavior. It also creates a cleaner interface because everything is divided into specific groups. This approach opens up the opportunity for designers and other non-programmers so that they can work together and test and tweak game features easily. From a traditional perspective it is understandable that the traditional entity system is still around in many game engines at companies these days.

8.2 Data-Oriented Approach
One of the main fundamental differences between a traditional entity system and a component-based is the programming paradigms. Traditional entity systems is built from object-oriented programming which makes sense from a logical point of view. A car is a car which has wheels and the wheels have rubber etc. Each game entity is an abstraction from the real world equivalent. With a data-oriented approach (which is what an aggravated component based-entity system is based on) everything simply becomes a chunk of the associated data structures. A car is not a car. It is rather an entity constructed from several pieces put together in an abstract form. The data-oriented approach can be unclear on how everything really is associated and shared with other entities. Debugging an entity in a traditional system, it is clear where each operation starts and ends, but with components, the level of abstraction and thereby the risk for confusion regarding which entity that is currently running, is increased.
8.3 Choosing a system

It is important to consider what system may fit the project and also consider if it can be reused for future games. A smaller game with simple game entities may work very well with a traditional entity system, but even so the code written for the game will be specific for that project and code recycling becomes difficult. Creating behavioral entities, using a component-based entity system enables easy reuse due to code separation.

Time is always an issue for every project and deciding to go down the road of a component-based entity system can be seen as a gamble. As previously stated, building a component-based entity system is time consuming because a lot of new problems will occur.

Communication between entities are not as straight-forward as with a traditional entity system. The component-based entity system in Vitei’s game engine, was developed as the project carried on, meaning that certain functionalities were considered but altered when the assumed implementation did not give the expected result. This meant that the system changed alongside with the development, which caused game play implementations to be re-written whenever the core of the system was altered.
References


[8] Inheritance (IS-A) vs. Composition (HAS-A) Relationship [retrieved January 22, 2014].

[9] Usagi – In-house game engine developed at Vitei, inc.