UDC 004.05

ARCHITECTURAL DESIGN PATTERN ENTITY-COMPONENT-SYSTEM

DZMITRY SUSCHEUSKI*, *IRYNA BURACHONAK
Polotsk State University, Belarus

In the presented article, the architectural design pattern Entity-Component-System is investigated. It describes its main features that improve the architecture of the game application and solve problems related to the performance and flexibility of the software.

One of the main problems of the present time is the ever-increasing complexity of software. This leads to an increase in the probability of making a mistake. To solve these problems, new approaches to writing software, design patterns and rules to facilitate system support are created. Games, like other applications, should work quickly, and their design should allow easy expansion of functionality.

In the presented article we will take a closer look at one of the interesting approaches of the Entity Component System (ECS) which is based on composition. ECS is an architectural pattern that is mainly used in game development. ECS adheres to composition, not inheritance, which allows for greater flexibility in defining objects. Objects consist of one or more components that add additional behavior or functionality. Therefore, the behavior of an object is easy to change at run time by adding or removing components [1]. Many large companies, such as Unity, Epic, Ctytek use this template. This approach is data-oriented and reduces the number of cache miss [2].

The component approach is increasingly recognized in game development. The main idea of the approach is to divide the functional into separate components, which are mostly independent of each other. Standard deep inheritance hierarchies are not used. Instead of traditional hierarchies of objects, collections are created, collections of independent components. Each object has only those functions that it needs. Any new functionality is easily implemented by adding new components.

The cache controller manages the contents of the cache, retrieving data from the RAM, transfers it to the processor, and returns the results of the calculations to RAM. When the processor core accesses the controller for data, the controller checks if this data is in the cache memory. If the data is contained in the cache, then it is given to the processor. If the data is not found, the kernel must wait for the moment when this data will be loaded from RAM, which is a very resource-intensive process. The situation when the cache does not have the necessary data is called a cache miss. The controller tries to keep the cash misses to a minimum. Processor cache size is very small compared to RAM. Usually it stores a small part of the data taken from the RAM. As a rule, all modern processors have caches of different levels (Figure 1); the higher the level, the less memory it has and the more efficient the data acquisition rate is. If the data is not found in the uppermost level of L1, then there is an appeal to the level that is below L2 and the necessary data is searched, if this data is not there, then the last level of L3 is accessed, if no data is found in it, then the controller accesses the RAM. Such complex architectural solutions for working with memory were chosen for the reason that usually the bottleneck in programs is working with memory. Therefore, it is very important to work with memory as efficiently as possible. To do this, the program should be as cache-friendly as possible to achieve maximum performance.

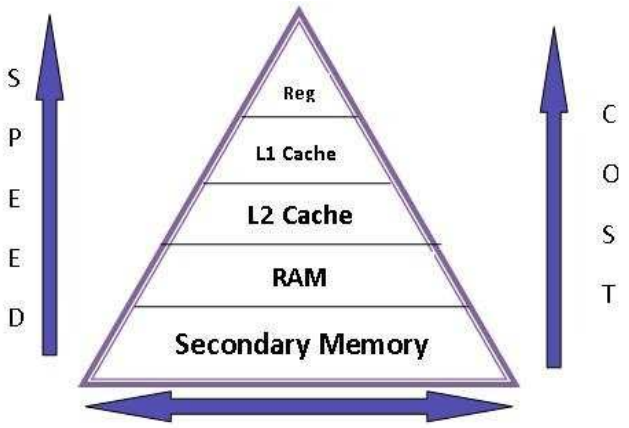

Figure 1 – Computer memory levels

The pattern can be divided into three parts.

1. Entities are container objects that do not have properties that act as storage for "components". An entity is an implicit aggregation of components. Storage is usually a simple data container. Usually, an approach is used in conjunction with entities, called object pooling (of the object pool). This reduces the cost of allocating and redistributing memory, which is a resource-intensive process. The idea of this approach is to allocate a large memory size, and if necessary, take it from the reserve and use it to create objects, and after the object is not needed, you need to return it back.

2. Components are data blocks that define the possible properties of any game objects or events. All this data is grouped and processed by a certain logic. They are objects with a simple data structure (plain old object, POD). Each type of component can be attached to an entity to determine its characteristics. Components contain no logic, sometimes they are empty markers for processing the system. The component can be compared with the structure in the C programming language, it has no methods and is capable of storing data. Each component describes a specific aspect of an object and its parameters. The components themselves make little sense, but when combined with entities and systems, they become an extremely powerful tool for solving development problems. For example, entities can be assigned the "health" property, which is a regular integer or fractional value in memory. The component should not have a large size, because this will cause problems in processing speed.

3. System is responsible for the processing of components, in which the work of all logic takes place. The system has a list of components with certain types, where it scans them and processes them. As a rule, the system never has one element, it deals with a collection, and processes the elements in turn, but this does not mean that the collection cannot be empty or have only one element. This approach eliminates the problems if in the future it will be necessary to add, for example, a new character. For example, the system can work with position, speed. Each system will be updated in a logical order.

Intersystem communication can be done in a variety of ways. For example, the way to send data between systems is to store certain data in components. In the game, the position of the object can be constantly updated. However, this approach is not always good when events occur rarely and it is necessary to somehow keep the current state. The most common are status flags, but this has a big drawback. Systems at each iteration will read the flags and check the availability of the event, which may be ineffective, one of the reasons may be the branch prediction.

The branch prediction is a mechanism that is a part of microprocessors, with a pipeline architecture, where the prediction is carried out whether a conditional transition will be performed in an executable program. The reason is that modern processors perform many operations in parallel, which allows to reduce the downtime of the conveyor due to preloading and execution of instructions that must be performed after the conditional transition. Branch prediction plays a critical role [3]. This can be a problem in ECS, since the components are treated as a continuous pipeline with low latency. For example, in Unity, the Fixed Update method runs 50 times per second. There is also an Update method, the speed of which can vary from the power of the computer, i.e. maybe both 100 and 10 calls per second. In this case, the problem may be critical. Therefore, as a rule, there are several subsystems in the system that update entities. To solve this problem, a design pattern can be used - an observer. All systems that depend on an event subscribe to it. Thus, the action from the event will be executed only once at the moment when it happens and does not require constant polling with checks.

The Entity-Component-System approach solves the problem with multiple invocations of update methods that surround all gaming applications. A good example is the Unity game engine. Native calls of programming language C ++ occur in it. There is a performance test of one of the Unity developers, in which he showed how native calls affect performance. In these tests, the Unity developer made 10,000 calls to the Update method and called the update method, which is not native. Links to components were added to the collection, where they were called in turn [4]. Table 1 presents the results of performance measurements.

Table 1. – Measurements of the performance of native and non-native methods in Unity

Mono

Methods	**iPhone 6**	**iPhone 4s**
Update	2.8ms	
Manager	0.52ms	2.1ms

IL2CPP

Methods	**iPhone 6**	**iPhone 4s**
Update	5.4ms	10.91ms
Manager (Dynamic Array)	1ms	2.52ms
Manager (Array)	0.22ms	1.15ms

As a result of the study, it can be concluded that the use of the ECS approach not only reduces code connectivity, but also significantly increases program performance. The proposed approach allows us to simplify further support and provide a simple way of connecting the game components to each other.

The advantages of the ECS approach include: flexibility, scalability, efficient memory use, easy access to objects, and ease of testing.

In the future, it is planned to introduce this approach into the development of a casual multiplayer gaming application. Since Unity is initially used is not fully implemented in ECS approach, because of which there are certain performance problems, with a large number of objects.

REFERENCES

1. Wikipedia. Entity-Entity-Component-System. [Electronic resource] / Wikipedia. – Access mode: https://en.wikipedia.org/wiki/Entity-component-system/. – Access date: 10.09.2018.
2. Game Programming Patterns. [Electronic resource] / http://gameprogrammingpatterns.com/data-locality.html/. – Access date: 10.09.2018.
3. Branchprediction. [Electronic resource] / Danluu. – https://danluu.com/branch-prediction/. – Access date: 10.09.2018
4. 10 000 вызовов Update. [Electronic resource] / Unity. – Access mode: https://blogs.unity3d.com/ru/2015/12/23/1k-update-calls/. – Access date: 10.09.2018.