**УДК 004.05**

# АРХИТЕКТУРНЫЙ ШАБЛОН ПРОЕКТИРОВАНИЯ ENTITY-COMPONENT-SYSTEM

***Д.В. СУЩЕВСКИЙ***
***(Представлено: канд. тех. наук, доц. И.Б. БУРАЧЁНОК)***

*Исследован архитектурный шаблон проектирования Entity-Component-System. Описаны его основные возможности, улучшающие архитектуру игрового приложения и решающие проблемы, связанные с производительностью и гибкостью программного обеспечения.*

Одной из главных проблем настоящего времени является постоянно растущая сложность программного обеспечения (ПО). Это приводит к увеличению вероятности допустить ошибку. Для решения этих проблем создаются новые подходы к написанию ПО, шаблоны проектирования и правила, облегчающие поддержку системы. Игры, как и другие приложения должны работать быстро, а их дизайн должен позволять легко расширять функционал.

В представленной статье подробнее рассмотрим один из интересных подходов Entity Component System (ECS) в основе которого лежит композиция. ECS – это архитектурный шаблон, который в основном используется при разработке игр. ECS придерживается композиции, а не наследования, что позволяет повысить гибкость в определении объектов. Объекты состоят из одного или нескольких компонентов, которые добавляют дополнительное поведение или функциональность. Поэтому поведение объекта легко изменить во время выполнения, путём добавления или удаления компонентов [1]. Многие крупные компании, такие как, Unity, Epic, Ctytek используют этот шаблон. Данный подход является ориентированным на данные и позволяет уменьшить количество кэш промахов (cache miss) [2].

Компонентный подход получает всё большее признание в разработке игр. Главная идея подхода состоит в том, чтобы разделять функционал на отдельные компоненты, которые в основном не зависят друг от друга. Стандартные глубокие иерархии наследования не используются. Вместо традиционных иерархий объектов создаются совокупности, коллекции независимых компонентов. Каждый объект обладает только теми функциями, которые ему необходимы. Любой новый функционал легко реализуется путём добавления новых компонентов.

Контроллер кэш памяти управляет содержанием кэша, получая данные из оперативной памяти, передаёт ее процессору и возвращает результаты вычислений в оперативную память. При обращении ядра процессора к контроллеру за данными, контроллер проверяет есть ли эти данные в кэш-памяти. Если данные содержаться к кэш-памяти, то они отдаются процессору. Если данные не найдены, то ядру необходимо ожидать момента, когда эти данные будут загружены из оперативной памяти, что является весьма ресурсоёмким процессом. Ситуация, когда в кэш-памяти не оказывается необходимых данных, называется кэш-промахом. Контроллер пытается свести кэш-промахи к минимуму. Размер кэша процессора по сравнению с оперативной памятью очень мал. Обычно в нём хранится небольшая часть данных, взятых из оперативной памяти. Как правило у всех современных процессоров есть кэши различных уровней (рис.), чем выше уровень, тем у него меньше памяти и эффективней скорость получения данных.

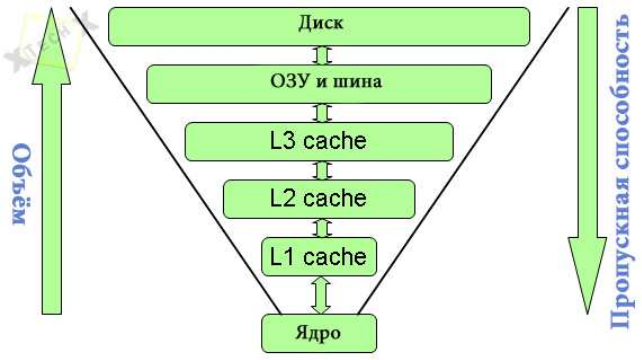


**Рисунок. – Уровни памяти компьютера**

Если данные не найдены в самом верхнем уровне L1, то идёт обращение к уровню, что находится ниже L2 и происходит поиск нужных данных, если этих данных нет и там, то происходит обращение к последнему уровню L3, если данных не найдено и в нём, то контроллер осуществляет обращение к оперативной памяти. Такие сложные архитектурные решения для работы с памятью были выбраны по причине того, что обычно самое узкое место в программах, это работа с памятью. Поэтому очень важно, чтобы работа с памятью была максимально эффективной. Для этого программа должны быть максимально кэш дружелюбной (cache friendly), чтобы достичь максимальной производительности.

Шаблон можно разделить на три составляющие.

1. Сущности (Entity) – объекты контейнеры, не обладающими свойствами, которые выступают в роли хранилища для «компонентов». Сущность представляет собой неявное агрегирование компонентов. Как правило, хранилище – простой контейнер данных. Обычно совместно с сущностями используется подход, называемый пулинг объектов (object pool). Это позволяет сократить расходы на выделение и перераспределение памяти, что является ресурсоёмким процессом. Идея данного подхода состоит в том, чтобы выделить большой размер памяти, а при необходимости брать её из резерва и использовать для создания объектов, а после того как объект не нужен, необходимо вернуть его назад.
2. Компоненты (Component) – это блоки данных, определяющие возможные свойства любых игровых объектов или событий. Все эти данные сгруппированы и обрабатываются определенной логикой. Они являются объектами с простой структурой данных (plain old object, POD). Каждый тип компонента можно прикрепить к сущности, чтобы определить её характеристики. Компоненты не содержат логики, иногда они являются пустыми маркерами для обработки системы. Компонент можно сравнить со структурой в языке программирования C, он не имеет методов и способен хранить данные. Каждый компонент описывает определённый аспект объекта и его параметры. Сами по себе компоненты практически не имеют смысла, но в сочетании с сущностями и системами, они становятся чрезвычайно мощным средством решения задач в разработке. Например, сущности можно присвоить свойство «здоровье», которое является обычным целочисленным или дробным значением в памяти. Компонент не должен иметь большой размер, т.к. это будет вызывать проблемы в скорости обработки.
3. Система (System) – отвечает за обработку компонентов, в них и происходит работа всей логики. У системы имеется список компонентов с определенными типами, где она просматривает их и обрабатывает. Как правило, у системы никогда нет одного элемента, она имеет дело с коллекцией, и обрабатывает элементы поочередно, но это не значит, что коллекция не может быть пуста или иметь всего один элемент. Такой подход избавляет от проблем, если в дальнейшем необходимо будет добавить, например, нового персонажа. Например, система может работать с позицией, скоростью. Каждая система будет обновляться в логическом порядке.

Межсистемная связь может осуществляется множеством способов. Например, способом отправки данных между системами – хранить определенные данные в компонентах. В игре положение объекта может постоянно обновляться. Однако данный подход не всегда хорош, когда события происходят редко и необходимо каким-то образом хранить текущее состояние. Самое распространенное – флаги состояния, но это имеет большой недостаток. Системы на каждой итерации будут читать флаги и проверять доступность события, что может быть неэффективным, одной из причин может стать предсказатель переходов (branch prediction).

Предсказатель переходов – это механизм, входящий в состав микропроцессоров, с конвейерной архитектурой, где осуществляется предсказание, будет ли выполнен условный переход в исполняемой программе. Причиной является то, что современные процессоры выполняют многие операции параллельно, что позволяет сократить время простоя конвейера за счёт предварительной загрузки и исполнения инструкций, которые должны выполняться после выполнения условного перехода. Прогнозирование ветвлений играет критическую роль [3]. Это может стать проблемой в ECS, так как компоненты обрабатываются, как непрерывный конвейер с маленькими задержками. Например, в Unity метод FixedUpdate выполняется 50 раз в секунду. Также есть метод Update, скорость которого может варьироваться от мощности компьютера, т.е. может быть, как 100, так и 10 вызовов в секунду. В таком случае, проблема может быть критичной. Поэтому, как правило в системе присутствует несколько подсистем, которые обновляют сущности. Для решения этой проблемы может быть использован шаблон проектирования – наблюдатель. Все системы, зависящие от события, подписываются на него. Таким образом, действие из события будет выполнятся только однажды в тот момент, когда это произойдёт и не требуется постоянный опроса с проверками.

Подход Entity-Component-System решает проблему с множественными вызовами методов обновления, которые окружают все игровые приложения. Хороший пример – игровой движок Unity. В нём происходят нативные вызов языка программирования C++. Существует тест производительности одного

из разработчиков Unity, в котором он показал то, как нативные вызовы влияют на производительность. В этих тестах разработчик Unity осуществил 10000 вызовов метода Update и вызов метода обновления, не являющийся нативным. Ссылки на компоненты были добавлены в коллекцию, где вызывались поочередно [4]. В таблице представлены результаты замеров производительности.

Таблица. – Замеры производительности нативного и не нативного методов в Unity

Mono

| **Методы** | **iPhone 6** | **iPhone 4s** |
|---|---|---|
| Update | 2.8ms | |
| Manager | 0.52ms | 2.1ms |

IL2CPP

| **Методы** | **iPhone 6** | **iPhone 4s** |
|---|---|---|
| Update | 5.4ms | 10.91ms |
| Manager (Dynamic array) | 1ms | 2.52ms |
| Manager (Array) | 0.22ms | 1.15ms |

В результате исследования можно сделать вывод, что использование подхода ECS не только позволяет уменьшить связанность кода, но и существенно увеличить производительность программы. Предложенный подход позволяет упростить дальнейшую поддержку и обеспечить простой способ связи игровых компонентов между собой.

Из достоинств подхода ECS можно отметить: гибкость, масштабируемость, эффективное использование памяти, простой доступ к объектам и лёгкость тестирования.

В дальнейшем планируется внедрить данный подход в разработку казуального мультиплеерного игрового приложения. Так как в Unity изначально используется не чистый не полностью реализованный в ECS подход, из-за чего присутствуют определенные проблемы с производительностью, при большом количестве объектов.

ЛИТЕРАТУРА

1. Wikipedia. Entity-Entity-Component-System [Электронный ресурс] / Wikipedia. – Режим доступа: https://en.wikipedia.org/wiki/Entity-component-system/. – Дата доступа: 10.09.2018.
2. GameProgrammingPatterns [Электронный ресурс]. – Режим доступа: http://gameprogrammingpatterns.com/data-locality.html/. – Дата доступа: 10.09.2018.
3. Branch prediction [Электронный ресурс] / Danluu. – Режим доступа: https://danluu.com/branch-prediction/. – Дата доступа: 10.09.2018
4. 10 000 вызовов Update [Электронный ресурс] / Unity. – Режим доступа: https://blogs.unity3d.com/ru/2015/12/23/1k-update-calls/. – Дата доступа: 10.09.2018.