Универзитет у Београду

Електротехнички факултет



Развој апликација заснован на архитектури оријентисаној на подацима

Дипломски рад

|  |  |
| --- | --- |
| Ментор: | Кандидат: |
| доц. др Дражен Драшковић | Маја Скоко 2016/0491 |

Београд, септембар 2020.

Садржај

[Садржај i](#_Toc52099085)

[1. Увод 1](#_Toc52099086)

[2. Традиционална архитектура 2](#_Toc52099087)

[2.1. Објектно оријентисано програмирање 2](#_Toc52099088)

[2.2. Развој видео игара коришћењем *OOP* 2](#_Toc52099089)

[2.3. Предности 3](#_Toc52099090)

[2.4. Мане 3](#_Toc52099091)

[2.4.1. Одржавање 3](#_Toc52099092)

[2.4.2. Перформансе 5](#_Toc52099093)

[3. Архитектура оријентисана на подацима 6](#_Toc52099094)

[3.1. Историја 6](#_Toc52099095)

[3.2. *Entity-Component-System* архитектура 7](#_Toc52099096)

[3.3. Поређење са објектно оријентисаним дизајном 8](#_Toc52099097)

[3.3.1. Добијање на брзини коришћењем кешa 8](#_Toc52099098)

[3.3.2. Поређење перформанси 12](#_Toc52099099)

[4. Опис реализације 13](#_Toc52099100)

[4.1. *Entity Component System* 13](#_Toc52099101)

[4.1.1. Управљање ентитетима и компонентама 14](#_Toc52099102)

[4.1.2. Управљање системима 17](#_Toc52099103)

[4.1.3. Архитектура као цјелина 17](#_Toc52099104)

[4.2. *ECS* примјери коришћења 18](#_Toc52099105)

[4.3. *ECS* тестови перформанси 23](#_Toc52099106)

[5. Резултати анализе и дискусија 25](#_Toc52099107)

[5.1. Додатне могућности 25](#_Toc52099108)

[5.2. Недостаци архитектуре 25](#_Toc52099109)

[6. Закључак 27](#_Toc52099110)

[Литература 29](#_Toc52099111)

[Списак скраћеница 30](#_Toc52099112)

[Списак слика 31](#_Toc52099113)

[Списак табела 32](#_Toc52099114)

[A. Имплементација 33](#_Toc52099115)

[A.1. Entity 33](#_Toc52099116)

[A.2. Component 33](#_Toc52099117)

[A.3. System 34](#_Toc52099118)

[A.4. EntityManager 35](#_Toc52099119)

[A.4.1. Chunk 37](#_Toc52099120)

[A.4.2. Archetype 37](#_Toc52099121)

[A.5. SystemManager 39](#_Toc52099122)

[A.6. Timer 40](#_Toc52099123)

[A.7. Еngine 41](#_Toc52099124)

[A.8. API 41](#_Toc52099125)

1. Увод

Игре су специфична врста апликација због тога што се од њих константно очекује висок ниво перформанси. Велике компаније попут *Unity*, *Epic* и *Crytek* су увидјеле да им је потребна нова програмска архитектура, како би се омогућио развој видео игара које имају знатно боље перформансе него што је то био случај раније. Велике хијерархије класа, сложене путање наслеђивања и додавања нових типова објеката учинили су да флексибилност, поновна употреба кода и перформансе трпе.

Да би се рјешио овај проблем, традиционални објектно оријентисани дизајн замјењен је новом архитектуром која је названа *Еntity Component Sytem* (*ECS*) и управо њеном имплементацијом се бави и овај рад. *Entity Component System* представља архитектуру која омогућава велику флексибилност при дизајнирању игара. *ЕCS* је оријентисан на саме податке, концепт објекта се више не користи већ су подаци и логика одвојени. Наравно, објекат као инстанца класе или структуре се мора створити, јер су то ограничења програмског језика, али у њему се налазе само подаци и стање које је потребно, док се остала логика дефинише у другим дјеловима кода.

Циљ овог рада је упознавање са концептом *Entity Component System* архитектуре, као дизајна оријентисаног на податке, а не објекте. Имплементирати платформу која ће детаљно приказати начин функционисања *ЕCS* архитектуре и на који начин она доприноси флексибилности и бољим перформанса саме апликације.

У наставку рада, конкретно у поглављу 2 биће објашњен традиционални начин кодирања, како би се боље могла направити паралела са новим дизајном оријентисаним на податке. У поглављу 3 биће представљена сама структура *ECS* архитектуре, на који начин *ECS* функционише и главне предности у односу на традиционални дизајн. Након тога биће детаљно приказани детаљи имплементације и начини употребе исте, са примјером коришћења и одређеним тестовима перформанси. За сами крај споменуте су додатне могућности и потенцијални проблеми и мане *Entity Component System* архитектуре.

1. Традиционална архитектура

Објектно оријентисано програмирања (енг*. Object-Oriented Programming ООP*) је начин да се организује код при развоју софтвера, укључујући и видео игре. У овом поглављу наведени су основни принципи објектно оријентисаног програмирања и како се они могу користити за креирање програмског кода, зашто је овај дизајн користан и како га је могуће искористи у развоју видео игара.

* 1. Објектно оријентисано програмирање

У *ООP* имплементацији кода сваки логички објекат креиран је као инстанца класе, и такве инстанце познате су под именом објекти. Сваки објекат садржи информације и о **стању** и о **понашању** истих.

**Стање** представља карактеристике објекта и обично је у форми *„садржи/има“* или *„је“*. Као примјер може се навести компјутер који *је* укључен или искључен, столица која *има* четири ноге и слично. [1]

**Понашање** су ствари које објекат *„може радити“* или акције које објекат *„може изводити“*. За примјер се може узети човјек који *може да* сједи, користи компјутер или чита чланак. [1]

**Енкапсулација** у *OOP* објашњава да одређени сегменти апликације у себи садрже сву своју логику, док осталим сегментима показују само своју суштину. То значи да су одређени сегменти невидљиви вањским корисницима, те су им дати другачији начини приступа. **Наслеђивање** је процес код кога нову класу креирамо на основу постојеће класе. Објекти новокреиране класе, садрже све атрибуте (стање) и методе (понашање) базне класе. На овакве класе могуће је примјенити **полиморфизам**, што значи да функције наслеђене од базне класе, свака изведена класа може да редефинише и да извршава на свој начин.

* 1. Развој видео игара коришћењем *OOP*

Данас, велики број програма за развијање видео игра (енг*. Game Engine*) користи управо објектно оријентисани дизајн. Као примјер можемо узети програм *Unity*. Свака скрипта написана, без обзира који се програмски језик користи, представља једну објектну класу. Свака класа садржи информације о стању и понашању самог објекта из игре (енг. *Game Object*). Сваком објекту из игре могуће је додати различите врсте компоненти које садрже написане скрипте које дефинишу понашање. Када су нам потребна два објекта која имају исти интерфејс, али различиту имплементацију користи се наслеђивање који је један од основих концепата објектно оријентисаног програмирања.

* 1. Предности

Велика предност објектно оријентисаног програмирања је што помаже у стварању одрживог кода који је лако разумљиви и уредан. Такође, врло је погодан за директно превођење објеката из „реалног свијета“ и њихових интеракција у објекте у коду, јер игре се односе на различите врсте објеката који међусобно комуницирају и ажурирају своја стања.

* 1. Мане

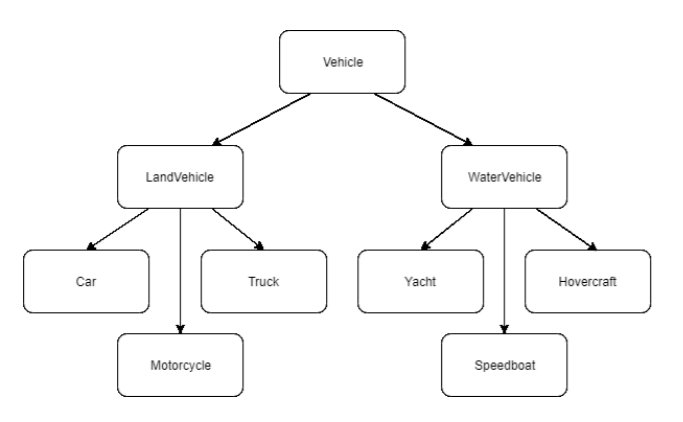
Постоје два озбиљна недостатка у објектно оријентисаном дизајну који могу створити проблеме приликом креирања и извршавања играма, а то су одржавање и перформансе. Како би дошли до рјешења таквих проблема у наредним потпоглављима детаљно су објашњени узроци настајања истих.

* + 1. Одржавање

Проблем који се појављује код овог дизајна је проблем класификације објеката. Постоји могућност класификација објеката на основу само једног скупа критеријума. На примјер, уколико се организам класификује према генетским особинама, боја организма није узета у обзир на било који начин. Да би се организам класификовао по боји, потребно је да се креира потпуно ново структурно „стабло“.

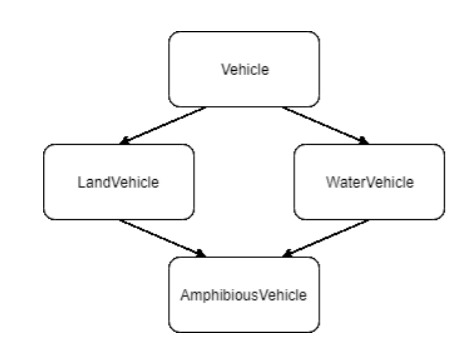
У стварном животу чести су примјери објеката када је потребно да нека класа интегрише више различитих критеријума класификације у себе. Такође, дешава се да је потребно да се направи додатан простор за нове типове класа који нису били предвиђени када је хијерархија класа била оргинално прављена. [2]

На слици 2.4.1.1. приказана је хијерархија класа возила, у објектно оријентисаном дизајну, који се најприје дјеле на копнена и водена возила, а затим свака од тих класа има и своје конкретне врсте.



Слика 2.4.1.1 Хијерархија класа аутомобила

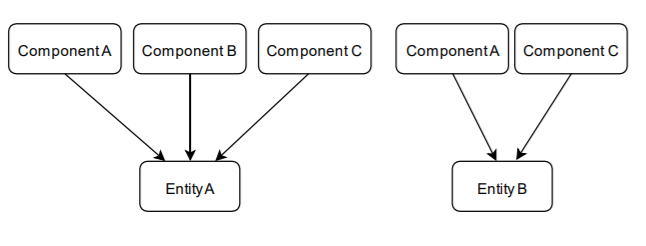
Иако се оваква хијерахија чини прихватљивом, може да дође до проблема уколико дизајнер видео игре одлучи да дода још једну врсту возила, која могу да се крећу и по копну и по земљи. Таква класа би морала да наслиједи особе из двије класе, класе копнених возила и водених возила. Међутим, таким наслеђивањем врло брзо би се уочио проблем изазван вишеструким наслеђивањем од истог родитеља. Такозвана дијамант структура добијена овавим наслеђивањем приказана је на слици 2.4.1.2. Оваква хијерархија може врло брзо постати компликована за одржавање и проширивање. Намеће се и питање која копија основне класе „*Vehicle*“ ће се кориситити у новододатој класи. Да би рјешили овакав проблем потребно је потпуно преуредити хијерархију класа, што ће довести до понављања кода.



Слика 2.4.1.2. Дијамант структура класа

И управо ово представља споменути проблем одржавања кода. Настајање овакве структуре представља проблем, јер приликом развој апликације врло често се дешава да су нам потребни нови типови класа, а потпуно преуређивање хијерархиија представља неприхватљиво рјешење.

Рјешење у оваком случају је да се одступи од објектно-оријентисаног дизајна, и да се умјесто наслеђивања атрибута, уведу компоненте. Сваки објекат тј. ентитет може да комбинује и преклапа компоненте како би се изградило потребно понашање. Овакав дизајн би се могао назвати објетно оријентисана композиција и приказан је на слици 2.4.1.3.



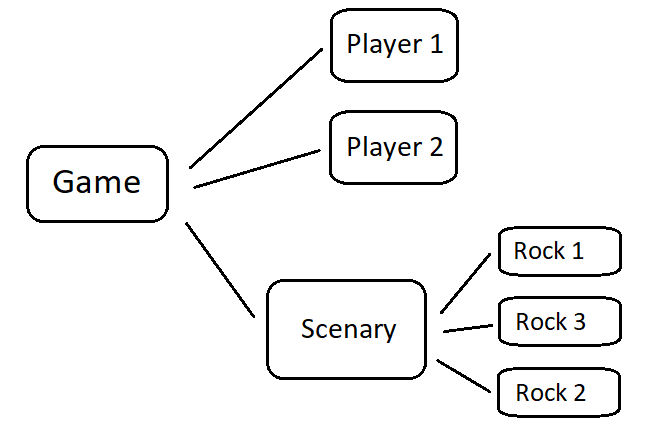
Слика 2.4.1.3 Хијерархија објектно оријентисане композиције

На слици 2.4.1.3 приказано је како су ентитети састављени од малих компоненти пружајући ентитету одређену функционалност, а уједно рјешавајући се сложених хијерархија класа. Компоненте се могу поново користити између различитих ентитета. Ово представља прикладније рјешење приликом програмирања сложених апликација, али и даље не пружа могућност раздвајања података и логике, због чега имамо значајне трошкове због полиморфизма током извођења.

* + 1. Перформансе

Када су у питању перформансе битно је да се спомене чињеница да брзина кеш меморије која је ближа процесору значајно већа од брзине системске меморије. Пажљиво коришћење кеш меморије може бити од невјероватне важности за брзину апликације.

Објекти у *ООP* обично су повезани са многим индиректним везама и референцама између самих објеката, али такође и референцама приликом претраживања свих виртуелни табела које су настале приликом позива методе неког објекта. Примјер је приказан на слици 2.4.2.1. Објектно оријентисани објекти се такође алоцирају по потреби и шире се по читавом меморијском простору. Оваква насумична расподјела потребних података по меморији доводи до тога да процесор понаша неефикасно, јер непрестално чека на приступ меморији.



Слика 2.4.2.1 Објекти у ООP повезани великим бројем референци

Како би се рјешили овакви проблеми, постији програмска парадигма која се бави другачијим приступом меморији названа *Data-oriented design* (*DOD*). Главна идеја оваквог приступа је уредити податке у меморији како би се повећала локалност података и направити код који користи велике блокове података одједном, умјесто да се ради над једним објектном у неком временском тренутку.

1. Архитектура оријентисана на подацима

Имплементација коришћењем дизајна оријентисаног на податке може се сматрати неком врстом проширења објектно оријентисаног дизајна композиције који је споменут у поглављу 2.4.1, али примјена и употреба захтјева потпуно нов начин размишљања од корисника.

Да би се хардвер машине искористио на најбољи начин, потребно је предузети главни развојни помак, користити дизајн оријентисан на податке. *DOD* је потпуно оријентисан на податке: код мора да буде дизајниран око података, а не обрнуто, што представља одступање од предходних система који су били објектно оријентисани. Када се правилно користи, *DOD* може омогућити да се искористе предности паралелизације и већи проценат погодака кеш меморије о чему ће се касније детаљније говорити. Додатне предности укључују модуларност, лакше умрежавање и лакшу сериализацију. [3]

* 1. Историја

Један од првих случајева употребе дизајна оријентисаног на податке у софтверима великих размјера је био 1998. године у видео игри под називом „*Thief: The Dark Project*“, у којој су програмери имали филозофију стварања изузето употребљивих компоненти. Посвећивање овом експериметалном дизајну награђено је програмом гдје „... није постојала било која врста хијерархије објеката заснованих на коду“. Приступ је функционисао тако добро да је тим успио да користи исту завршну датотеку кроз већи дио развоја програма „*Thief*“ и „*System Shock*“, двије веома различите игре, само избором различите хијерерхије објеката и скупа података у вријеме превођења. [4]

Још један познат примјер је игра под називом „*Dungeon Siege*“ из 2002. године која представља беспрекоран свијет без икаквих учитавајућих екрана, што је омогућио његов систем оријентисан на податке, систем који има велику сличност са *ECS* (који ће детаљније бити објашњен у наставку) иако термин још није званично био скован.

„*Dungeon Siege*“ је представио преко 7300 јединствених објеката од 7 типова, као и преко 100 000 објеката смештених између двије мапе [5]. Континуирани свијет ових размјера у то вријеме био је изузетно постигнуће, а модуларност систем заснован на компонентама дозволила је флексибилно управљање меморијом током извршавања.

Касније, овакав дизајн је озваничен захваљујући Адаму Мартину, британском програмеру, који је створио и ширио своје идеје и смјернице за овакав дизајн у индустрији, популаришући га посебно у контексту развоја програма за прављење видео игара. Као такве, већина парадигми *Entity Component System* приписује се Мартину, чак и ако су други тимови већ експериментисали и радили на програмима заснованим на сличним обрасцима.

Иако се академски радови и публикације на ову тему ријетко могу наћи, тренутно постоји велики број извјештаја и студија случаја успјешне употребе *ECS* архитектуре у играма и апликацијама, у распону од *VFX* и рачунарске графике, па све до независних развоја игара.

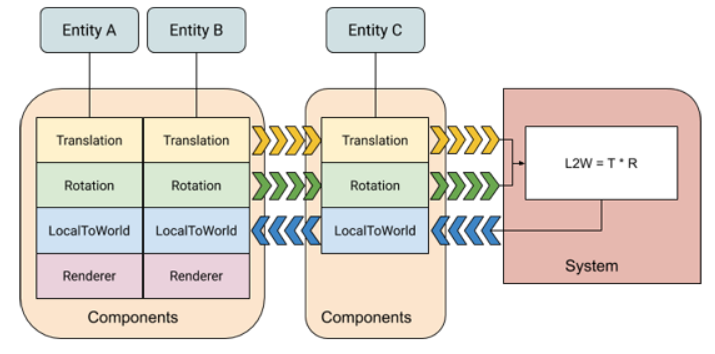
* 1. *Entity-Component-System* архитектура

*Entity-Component-System* представља архитектурални пројектни узорак за развој сложених апликација или игара, које имају користи од дефинисања објеката у мање дијелове, који се могу искористи више пута. *Entity-Component-System* се заснива на принципу „композиција изнад наслеђивања“. Рјешава проблеме настале наслеђивањем, при чему настају велике хијерархије класа, и уводећи значајне предности у погледу перформанси, флексибилности и продуктивности.

*ECS* се састоји од три основна елемента која међусобно комуницирају:

* Ентитети: које је Адам Мартин дефинисао као „фундаменталне коцептуалне градивне блокове“ система, који представљају конкретне објекте. Не садрже податке и логику специфичну за апликацију.
* Компоненте: мале класе/структуре које је могуће искористити више пута и које „граде“ ентитете. Цитирајући опет Адама Мартина можемо их рећи да „означавају да ентитет посједује одређени аспект“. Компоненте садрже податке, али не и логику апликације.
* Системи: омогућавају логику апликације за ентитете који садрже одређени скуп компоненти.

Архитектура *Entity Component System* раздваја идентитет (ентитете), податке (компоненте) и понашање (системи). Системи трансформишу податке из улазног у излазно стање читањем података из компонентни које се индексирају на основу ентитета. У овом приступу, ентитетске инстанце су сведене са објеката на цјелобројне идентификаторе, који служе само у сврху обједињавања компоненти у једну цјелину. Компоненте у овој парадигми су једноставне, мале и безрезервни прикази својстава и података ентитета. Логиком се баве одвојени системи, који су увијек у потрази за активним компонентама. Не садрже никакве податке у себи и користе се само за трансформацију улазних податка на жељени начин. Три елемента су састављена заједно уз помоћ менаџера, који омогућава различитим дијеловима дизајна да комуницирају једни с другима. Менаџери су задужени за вођење евиденције које компоненте припадају којим ентитетима и која се користи за прослеђивање потребних података одговарајућим системима.



Слика 3.2.1 Ентитети, компоненте и системи као цјелина

* 1. Поређење са објектно оријентисаним дизајном

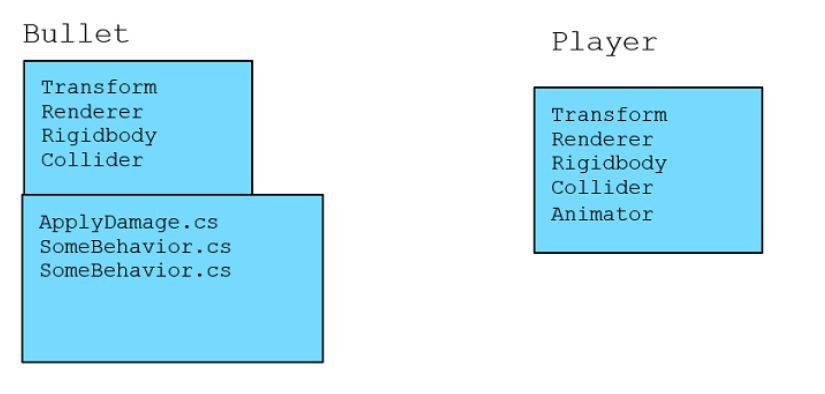
Главна разлика ове двије архитектуре је у томе што се *OOP* фокусира на, како му и само име каже, објекте. Цијели програм се пише с објектима на уму, те се сви остали концепти прилагођавају објектима. Објект можемо дефинисати као скуп података и функционалности који манипулишу тим подацима на једном мјесту.

Како би боље разумјели *OOP* и упоредили га са архитектуром оријентисаном на податке, најбоље погледати ток рада за креирање објеката на сцени у програму за креирање игрица *Unity*:

* креирање празног објекта *GameObject* на сцени,
* додавање компоненти које објекту дају жељене особине,
* креирање и додавање скрипти објекту које ће да манипулишу и мјењају стање компоненти у вријеме извршавања.
  + 1. Добијање на брзини коришћењем кешa

Може се примјетити да постоји низ мана оваквог тока рада. Неке од њих су већ навођене, али значајно је примјетити да је обрада везана за врло специфичан скуп података, тако да је поновна употреба кода ријетка. Као додатак на то, овакав систем је веома зависан од типова података који садрже референце на објекте (енг. *Reference types*).

На слици 3.3.1.1 приказана су два *GameObject* и њихове компоненте. *Bullet* је зависан од *Transform, Renderer, Rigidbody* и *Collider* референци. Објектни на које се упућује скриптама налазе у дијелу меморије који се назива *heap*. Као последицу, имамо да се подаци не трансформишу у облик којим може да оперише бржи дио меморије и *SIMD* векторска јединица. [7]

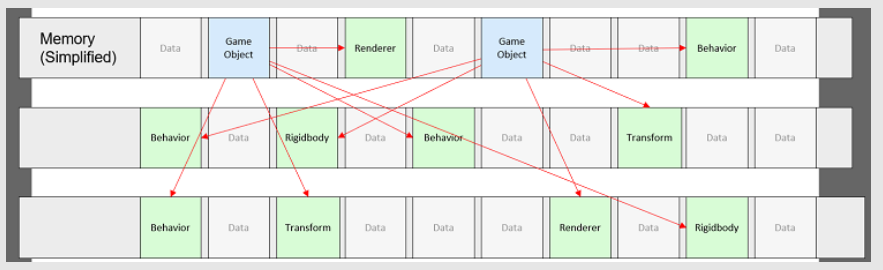


Слика 3.3.1.1 Примјер више *GameObject*-а и компоненти у ООP дизајну *Unity*

Приступ подацима из системске меморије је много спорији него припступ кеш меморији. То је меморија гдје до изражаја долази дохватања података унапријед. То уствари представља ситуацију када компјутерски хардвер предвиђа којим подацима ће се приступати наредни пут када се затражи приступ меморији, а затим се превентивно повлачи из системске, спорије меморије, у бржу како би подаци били спремни одмах када затребају. Коришћењем оваквог хардвера добија се лијеп пораст перформанси на предиктивним израчунавањима.

Ако као примјер узмемо итерирање кроз низ елемената, хардверска јединица за превентивно узимање података може да повуче групе података у брзу кеш меморију. Када дође вријеме да процесор приступи неком од наредних елемената низа, потребни подаци се већ налазе у кеш меморији и нема потребе да се приступа системској меморији и да се губи доста процесорког времена. За податке који су тијесно спаковани, тј. налазе се у меморији једни поред других у континуитету, као што су елементи низова, хардвер за повлачење података у кеш лако може довући праве податке. Међутим, уколико се ради о објектима који су разбацани у *heap* меморији, постаје немогуће да хардвер довуче податке који ће се заиста користити при следећем приступу меморији, и довлаче се бескорисни подаци, тако да немамо никакву уштеду у времену извршавања.

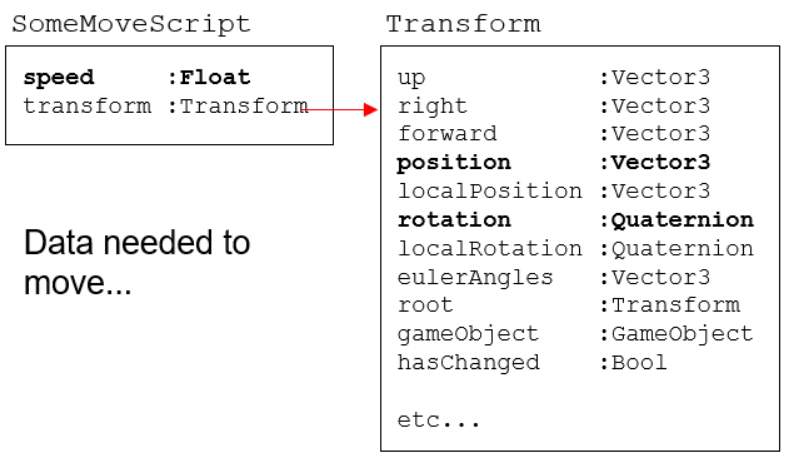
Баш такав примјер имамо у *OOP* дизајну при развоју видео игара, као што је приказано на слици 3.3.1.2. Слика приказује случајну спорадичну природу ове методе складиштења података. Сваки *GameObject* садржи одређен броји компоненти, које се алоцирају у *heap* меморији, тако да имамо разбацане објекте са великим бројем референци. Приступ компонентама је врло чест, а често је потребно и да приступимо већем броју компонената у истом тренутку када мјењамо стање неког објекта, а дешава се да је потребно промјенти и стање више сличних објеката који су такође разбацани по меморији. При сваком дохватању компоненти или објеката морамо да приступимо системској спорој меморији, јер је немогуће предвидјети која је то компонента која ће се следећа користити, па је овакав дизајн критичан за перформансе.



Слика 3.3.1.2 Референце у меморије између објекат, њиховог понашања и њихових компоненти

Као додатак на све то, сваки од тих референтних типова садржи пуно додатних података којима можда неће бити потребно да се приступити. Ови неискоришћени чланови такође заузимају драгоцјен простор у кешу процесора. Ако је потребно само неколико изабраних променљивих чланова постојеће компоненте, остало се може сматрати изгубљеним простором, као што је приказано на слици 3.3.1.3.

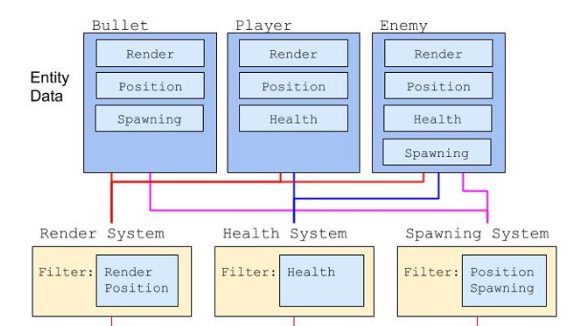
Како би се извршило помјерање *GameObject* коришћењем неке написане скрипте потребно је да приступимо позицији и ротацији, који су чланови компоненте *Transform*. Када хардвер дохвата податке из меморије, кеш меморија се попуњава и са свим осталим члановима ове компоненте који представљају бескорисне податке. Зато би било врло згодно када би било могуће да имамо само низ позиција и ротација за сваки *GameObject* за који постоји потреба да се помјери у неком тренутку.



Слика 3.3.1.3 Чланови који се стварно користе за операцију кретања и изгубљени простор

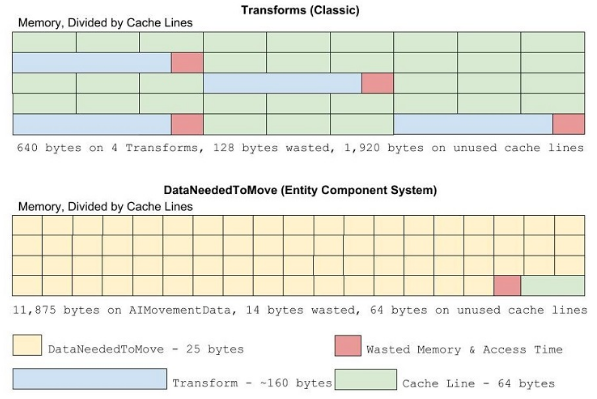
*Entity Component System* архитектура помаже у уклањању неефикасног референцирања објеката. Уместо *GameObject-*аса сопственом колекцијом компоненти, размотримо ентитет који садржи само податке који су му потребни.

На слици 3.3.1.4 можемо примјетити да *Bullet* ентитет није повезан са компонентом *Transform* или *Rigidbody*. Ентитет *Bullet* су само сирови подаци који су изричито потребни за рад приликом ажурирања.



Слика 3.3.1.4 *Еntity-Component-System* примјер

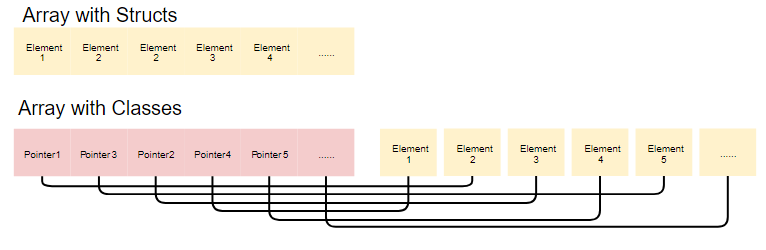
Наравно, није само систем за помјерање онај који има предност од оваког приступа. Још једна уобичајена компонента у многим играма су здравствени системи постављени у спектру непријатеља и савезника. Ови системи обично имају мало или нимало разлика међу врстама објеката, па су још један сјајан кандидат да искористе нови систем. Ентитет се користи за индексирање колекције различитих типова података који га представљају тј. компоненти. Системи могу филтрирати и радити на свим компонентама са потребним подацима без икакве помоћи програмера. Подаци су сви ефикасно организовани у чврсто спаковане сусједне низове. Предности овог система су огромне. Не само да побољшава вријеме приступа ефикасношћу кеш меморије, већ такође омогућава напредне технологије (аутоматска векторизација / *SIMD*) доступне у модерним процесорима којима је потребна ова врста поравнања података као на слици 3.3.1.5. Ово неоспориво даје перформансе у играма. Сваки кадар може учинити много више или учинити исто за много краће вријеме. [7]



Слика 3.3.1.5 Фрагментација у кешу и неискоришћени простор при ОО дизајну

Све о чему смо причали, може се примјетити на конкретном примјеру два низа у програмском језику C#. Први низ представља низ структура или низ примитивних типова, а величина оваквог низа позната је у вријеме превођења, па је елементе низа могуће спаковати као сусједе у меморију. Ово није случај и код другог низа који представља низ класа. Због полимофрмизма, сваки елемент има другачију величину, па представња немогуће да их спакујемо једне до других у меморији. Умјесто тога, низ чува показиваче (данас 64-битне) на друга мјеста на *heap*-у где се чувају стварни подаци. Елементи се насумично распоређују у зависности од тога када су створени помоћу *„new“* оператора - а не када је низ створен. Највјероватније ће неки од њих бити стављени на секвенцијалну меморијску локацију, али други ће бити складиштени изоловано.

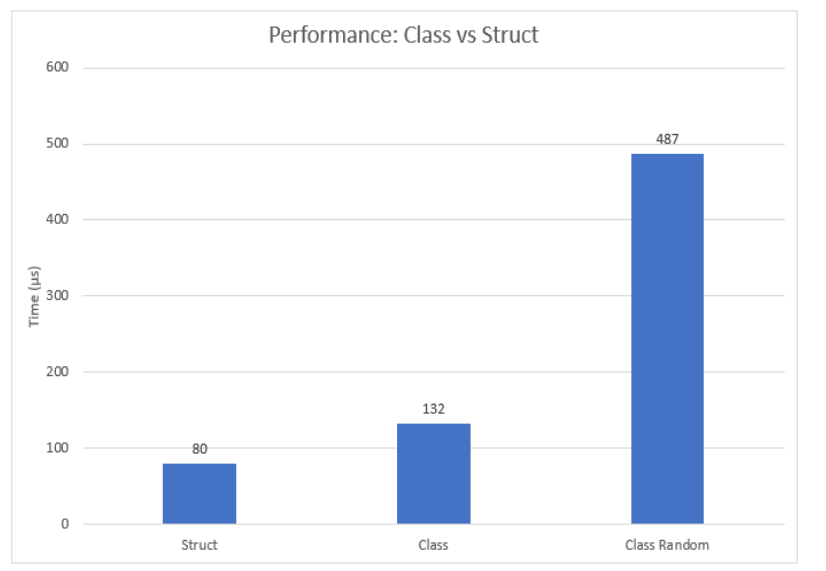
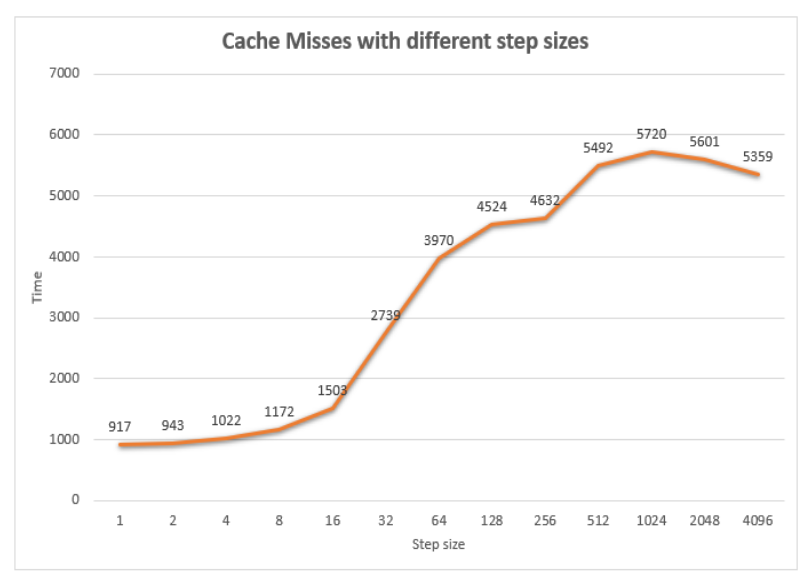
Примјер оваквог распоред у меморији приказан је на слици 3.3.1.6.



Слика 3.3.1.6 Распоред низоба и класа у меморији С#

* + 1. Поређење перформанси

На наредним сликама приказана је разлика у перформансама када се користи низ класа и низ структура. Тестирани низ садржи 1 милион елемената што даје величину низа од 4 MB. Приступ низу структура је бржи, док приступ низу класа траје 65% више времена. Ако се изврши премјештање елемената низа, вријеме извршавања је чак 6 пута дуже. Резултати тестова јасно показују колико је важан секвенцијални приступати меморији. Тест случајног распореда елемената класе показује колико лоше могу да буду перформансе ако користите насумични приступ у RAM меморији [6].

Слика 3.3.2.1 Поређење перформанси C#

1. Опис реализације

Имплементација *Entity Component System* архитектуре рађена је у програмском језику C++ користећи окружење *Visual Studio 2019*. Дизајн и примјена детаљно ће би приказани у овом поглављу користећи одређен број исјечака програмских кодова и дијаграма који помажу читаоцима да разумију основне концепте и архитектуру.

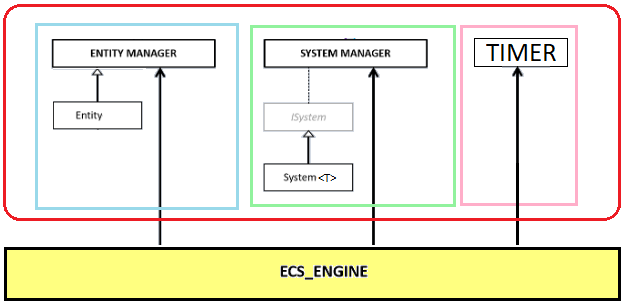
* 1. *Entity Component System*

Постоји више начина за имплементацију и примјену структура података које се користе. Сваки од тих начина свакако долази са својим предностима и слабостима. Заједнички циљ између свих приступа је тај да се покуша да се подаци складиште у меморији што је могуће непрекидније. Такође се поставља занимљиво питање: како ентитети, компоненте и системи међусобно комуницирају, ако су мање или више независни једни од других? Одговор се разликује у зависности од имплементације, али што се тиче начина који је образложен у овом поглављу главну везу између ентитета и система представљају саме компоненете.

Главне карактеристике имплементиране архитектуре:

* скалабилност - лако је додати нове типове ентитета, компоненти и система без било које унапријед задате горње границе, осим меморије система;
* флексибилност - не постоје зависности између ентитета, компоненти и система (ентитети и компоненте сигурно имају неку врсту зависности, али не садрже међусобну логику показивача);
* једноставно тражење/приступ објектима - лако проналажење ентитета и компоненти да би се извршило итерирање свих компоненти одређеног типа;
* контрола протока - системи могу зависити једни од других, стога се може успоставити тополошки редослијед њиховог извршавања;
* меморијска ефикасност – тежи се да се подаци смјештају што узастопније у меморији баш као што ће бити приказано у наредним потпоглављима.

На слици 4.1.1 приказана су четири различита подручја у боји. Свака област дефинише модуларни дио архитектуре. Модул за управљање ентитетима и компонентама (плаво подручје) и модул за управљање системима (зелено подручје). *Timer* представља модул који је заслужен за праћење протока времена у систему (розе подручје). Ови модули се углавном баве задацима управљања објектима. На врху је модул, *ECS\_Engine*. Овај глобални објекат оркестрира све остале модуле и брине се о иницијализацији и уништавању. Ово је кратак и врло апстрактан преглед, који ће бити детаљније објашњен.



Слика 4.1.1 Aрхитектура ECS система

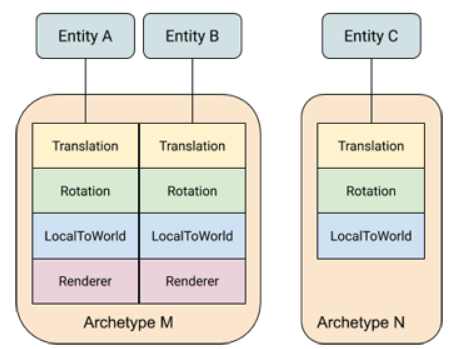
* + 1. Управљање ентитетима и компонентама

*EntityManager* је срце овог дијела имплементације. Преко њега се стварају и уништавају ентитети, додају и уклањају компоненате, ради филтрирање ентитета и све остало што је повезано са самим ентитетима и компонентама. Класа *EntityManager* треба да управља свим објектима ентитета током извршавања апликације.

Како би могли да схватимо како то *EntityManager* управља ентитетима и компонентама потребно је упознати се како су исти смјештени у меморији. Уводе се два важна појма, *Archetype* и *Memory* *chunk* који ће бити објашњени у нередним потпоглављима.

* + - 1. **Archetype** – прототип

Јединствена комбинација типова компоненти назива се *Archetype*. На примјер, 3Д објекат може имати компоненту за своју трансформацију тј. промјену позиције, једну за своје линеарно кретање, једну за ротацију и једну за визуелни приказ. Свака инстанца једног од ових 3Д објеката одговара једном ентитету, али зато што дјеле исти скуп компоненти, могу се класификовати као један *Archetype.*[8]На слици 4.1.1.1 приказан је примјер са два *archetypa* и ентитети који припадају једном од њих.



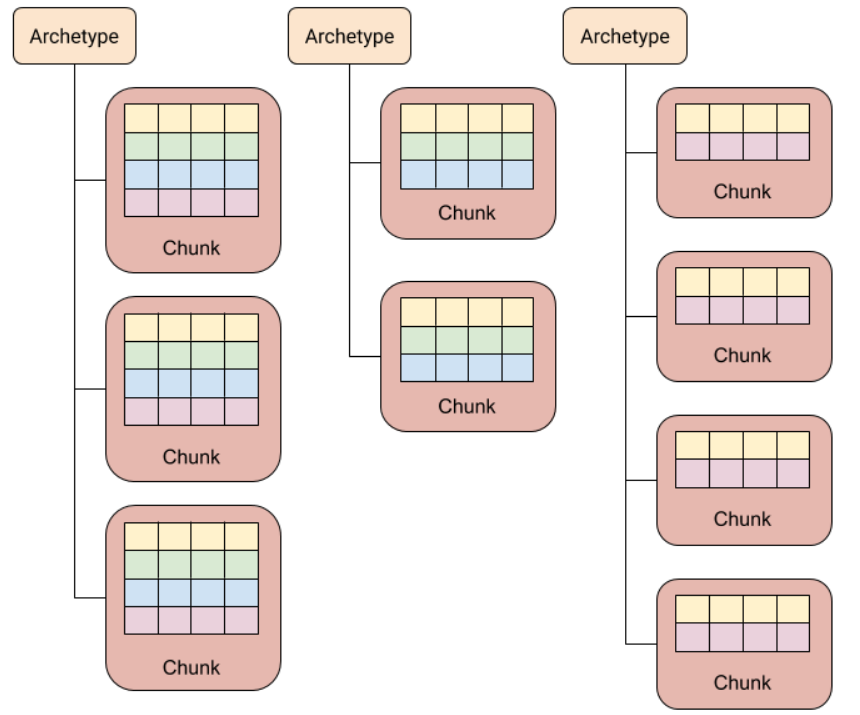
Слика 4.1.1.1 Archetypes у ECS-у

Ентитети *А* и *B* дјеле *archetype М*, док ентитет *C* има *archetype* *N*. Могуће је лако промјенити *archetype* ентитета додавањем или уклањањем компоненти током извршавања. На примјер, ако уклонимо компоненту *Renderer* из ентитета *B*, онда се *B* премјешта у *archetype N*.

* + - 1. **Memory** **chunk** –„ комад“ меморије

*Archetype* ентитета одређује где се чувају компоненте тог ентитета. ECS додељује меморију у „комадима“. *Chunk* увек садржи цјелине једног *archetype*-а. Када се *chunk* меморије напуни, нови дио меморије додjељује се за све нове ентитете створене са истим *archetype*. Ако се промјени *archetype* ентитета додавањем или уклањањем компоненти, компоненте за тај ентитет се премештају у други дио тј. у *chunk* новог *archetype-*а који има слободних мјеста .

На слици 4.1.1.2 приказан је начин на који су повезани *archetypes* и *chunks* у ECS-у.



Слика 4.1.1.2 Archetypes и Chunks у ECS-у

Ова организациона шема пружа однос један према више између *archetypes* и *chunks*. То такође значи да проналажење свих цјелина са датим скупом компонената захтјева само претрагу постојећих *archetypes*, који су обично малобројни, умјесто свих ентитета који су обично много већи.

Ентитети у „комаду“ меморије се не чувају у одређеном редоследу. Када се ентитет креира или промјени у нови *archetype*, он прелази у први „комад“ меморије који чува тај *archetype* који има мјеста. Компоненте у *chunk*-у ипак остају чврсто збијене; када се ентитет уклони из *archetype*, компоненте последњег ентитета у комаду премјештају се у новоослобођена мјеста у низовима компоненти.

„Комади“ меморије су имплементирани тако да заузимају 16 KB, како би могли да се поравнају са данашњом величином L1 и L2 кеша, како би се добиле најоптималније перформансе.

Компоненте у једном „комаду“ су смјештене на тај начин да иду све компоненте једне врсте, за тим све компоненте наредне врсте и тако за све компоненте које се налазе у том *archetype-*у. Овакво распоређивање компоненти је веома практично, јер се у низовима компоненти на истом индексу налазе компоненте истог ентитета. Као значајан додатак имплемантицији, треба поменути да се у *chunk*-у прије самих компоненти налазе сви ентити чије се компоненте налезе у том *chunk*-у.

* + - 1. **Ентитети и компоненте**

*EntityManager* управља свим ентитетима у систему. *EntityManager* одржава листу ентитета и организује податке повезане са ентитетима ради оптималних перформанси.

Иако ентитет нема тип, групе ентитета могу се категорисати по типовима компоненти података повезаних са њима. Док се креирају ентитети и додају им се компоненте, *EntityManager* прати јединствене комбинације компонената на постојећим ентитетима. Таква јединствена комбинација као што је већ споменуто назива се *archetype*. Постојећи *archetype*-ови се могу користити за стварање нових ентитета који су у складу са тим *archetype-*ом. Такође је могуће унапријед направити *archetype* користећи одговарајућу методу у класи *EntityManager* и помоћу тог *archetype* креирати ентитете.

У оквиру класе *EntityManager* постоје следеће методе за манипулацију ентитетима и компонентама:

* Креирање ентитета са компонентама користећи низ *ComponentType* објеката
* Уништавање једног ентитета на основу објекта тог ентитета
* Уништавање свих ентитета са одређеним скупом компоненти
* Додавање компоненте одређеног типа изабраном ентитету
* Уклањање компоненте одређеног типа изабраном ентитету
* Провјера да ли одређени ентитет има изабрану компоненту
* Креирање новог *archetype*-a
* Постављање или дохватање конкретне компоненти изабраном ентитету који већ садржи тај тип компоненте
* Креирање новог типа компоненте
* Постављање већ направљеног *archetype*-a изабраном ентитету
* Дохватање броја компоненти за одређени ентитет
* Дохватање свих ентитета који постоје у систему
* Дохватање свих ентитета који имају одређену компоненту
* Дохватање свих компоненти изабраних типова
  + 1. Управљање системима

Систем пружа логику која трансформише податке компоненти из тренутног стања у следеће стање, на примјер систем може ажурирати положаје свих покретних ентитета према њиховој брзини и временском интервалу од претходног ажурирања.

ECS покреће све системе дефинисане у игри и врши њихово ажурирање током извођења. Системи су организовани тако да је могуће успоставити редослијед њиховог извршавања.

Могуће је имплементирати једну или више наведених функција догађаја животног циклуса система:

* *OnCreate()* – позива се када се систем креира
* *OnUpdate()* – позива се на одређени временски интервал све док систем има посла и док је омогућен
* *OnDestroy()* – позива се када се систем уништава

За управљање системима у апликацији задужен је *SystemManager*. Он чува све постојеће системе у оквиру апликације као хеш мапу, гдје јединствени идентитфикатор система представља кључ, а сам систем вриједност. Могуће је да постоји само једна инстанца једног типа система. Када се систем креира он се мора додати преко *SystemManager*-а у споменуту хеш мапу. Сваки систем може да има један или више система од којих зависи његово извршавање па је неопходно да *SystemManager* одржава и листу редослиједа извршавања система.

У оквиру класе *SystemManager* класе постоје следеће методе за манипулацију системима:

* додавање система у листу свих система,
* дохватање инстанце система на основу типа система,
* омогућивање рада система одређеног типа,
* онемогућавање рада система одређеног типа,
* додавање зависности између система.
  + 1. Архитектура као цјелина

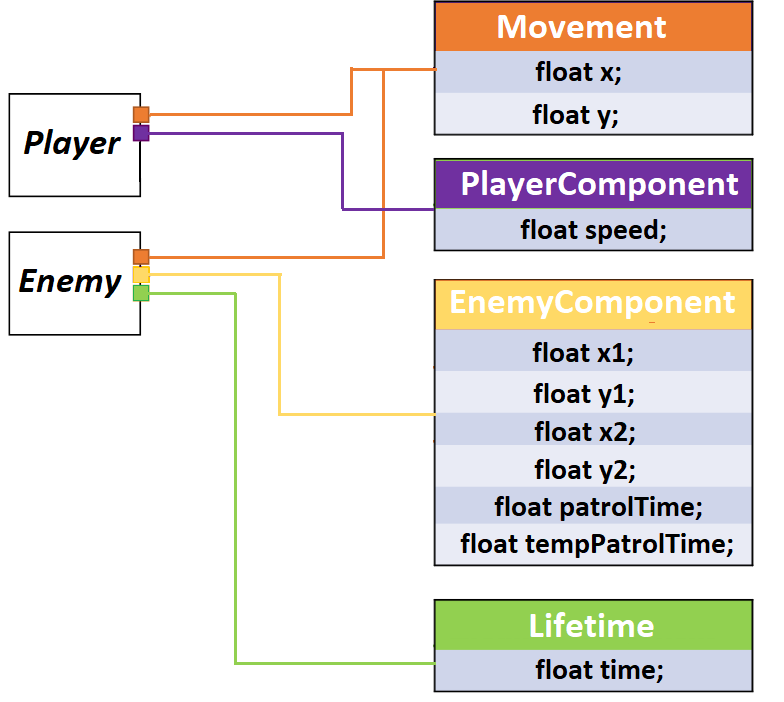
Како би се ентитети, компоненте и системи држали на окупу направљен је објекат класе *Engine*. Овај објекат осигурава лаку интеграцију и употребу у клијентском софтверу. Овај објекат садржи по инстанцу *EntityManager*-a, *SystemManager*-a и *Timer*-a. На страни клијента треба само укључити заглавље „ECS.h“ и позвати *ECS::Initialize()* методу. Позивом ове методе статички глобални објекат класе *Engine* ће бити иницијализован и може се користити на страни клијента да би се добио приступ свим класама менаџера. *ECS::Terminate()* треба позвати прије изласка из главног програма. Ово ће осигурати ослобађање свих стечених ресурса.

* 1. *ECS* примјери коришћења

*Еntity Component System* архитектура имплементирана на овај начин представља малу платформу за развој видео игара. Наравно прављене игре уз помоћу великих и отмјених програма за развој видео игра, попут *Unity* или *Unreal*, које обезбјеђују *GUI*, као и многе друге могућности, изгледају много боље и моћније, али у сврху демонстрације онога што је тема овог рада овдје ће бити посматран само *ECS*.

Идеја за овај примјер почива у чињеници да већина игара има двије основне врсте динамичких објеката, а то су главни играч и непријатељи. Због тога и у овом примјеру имамо двије врсте енитета *Player* и *Enemy.* Међутим, као што је већ речено ентитети у *ECS* представљају само класе са јединственим идентификатором, која већ постоји, тако да није било потребно прављене нових класа из којих би се правили ови објекти као што је то случај код објектно-оријентисаног програмирања. Овдје ће ентитети бити крерани заједно са информацијама о компонентама које садрже.

На слици 4.2.1 приказан је преглед споменутих ентитета и компоненти које имају дати ентитети. Гледајући ову слику може се лако видјети однос између ентитета и њихових компоненти.



Слика 4.2.1 Преглед ентитета и компоненти у ECS примјеру

Свим ентитетима заједничка је компонента *Movement.* Будући да се ентитети игара морају налазити негдје и имати своју позицију, они имају ову компоненту која описује положај ентитета. Положај ентитета је одређен на основу *x* и *y* координата, што значи да се играчи налазе у дводимензионалном простору. Као и све компоненте и ова компонента је изведен из класе *IComponent.*

class Movement : public IComponent {

public:

float x;

float y;

};

*Player* поред компоненте за положај, садржи компоненту *PlayerComponent* која садржи податке који су везани само за главног играча, а у овом случају је то брзина кретања. Брзина ће се користити приликом помјерања положаја самог ентитета.

class PlayerComponent : public IComponent {

public:

float speed;

};

*Еnemy* ентитети су обично они који су бројнији и постоји их велики број у игри. *Еnemy* поред компоненте за положај има још двије компоненте. Једна од тих компоненти је *Lifetime* која наводи преостало вријеме живота *Enemy* ентитета. Када вријеме у овој компоненти постане нула, ентитет ће бити уништен.

class Lifetime : public IComponent {

public:

float time;

};

Следећа компонента коју ентитети непријатељи садрже јесте *EnemyComponent* која чува податке који описују само ентитете овог типа. Ако пратимо редом промјенљиве у овој компоненти први пар *х1* и *y1* представљају координате прве позиције непријатеља који ће да се шета или „стражари“ између двије задате позиције. Други пар *x2* и *y2* представља координате друге позиције. Промјенљива *patrolTime* представља временски период који протекне између потребе да *Enemy* пређе са једне позиције на следећу. Промјенљива *tempPatrolTime* служи како би се чувало протекло вријеме од предходне промјене позиције ентитета.

class EnemyComponent : public IComponent {

public:

float x1;

float y1;

float x2;

float y2;

float patrolTime;

float tempPatrolTime;

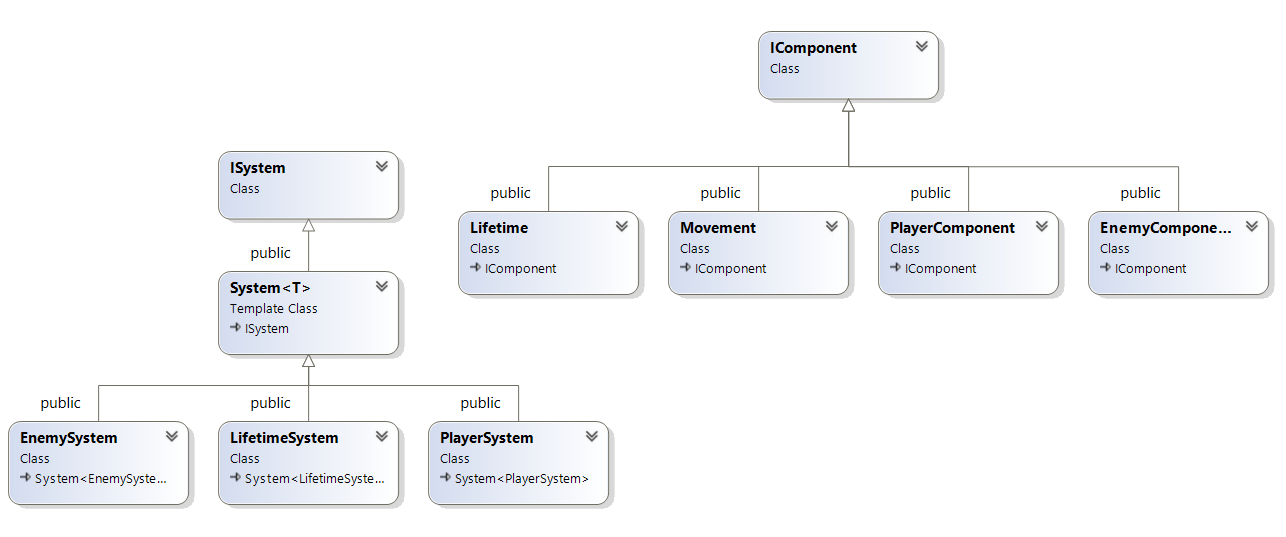
};

Имати све ове различите ентитете са њиховим појединачним компонената не довршава игру. Такође нам треба неко ко зна како да управља сваком од ових компоненти и ентитета. Ријеч је о системима, наравно. Системе користимо да цијелу логику игре подјелимо на много мање дијелове. Сваки дио се бави различитим аспектом игре.

У оквиру овог примјера коришћења имплементиране архитектуре постоје три система. Сваки систем има своју функцију, а оне су следеће:

* *PlayerSystem* – задужен за помјерање позиције главног играча
* *EnemySystem* – задужен за помјерање позиције непријатеља играча
* *LifetimeSystem* – задужен за контролисање времена живота свих непријатеља

Прије него што се крене на детаљно објашњавање система на слици 4.2.2 може се видјети класни дијаграм свих компоненти и система у овом примјеру, чији је циљ био представљање основних могућности ове архитектуре.



Слика 4.2.2 Преглед класа у примјеру коришћења ECS архитектуре

*PlayerSystem* као што је наведено управља свим ентитетима који имају *PlayerComponent*, иако је најчешће само један такав ентитет у игри. Његова улога јесте да контролише помјерање играча. Овај систем редефинише методу *OnCreate()* која је у случају овог система задужена да изврши иницијализацију енитета којим се управља овим системом. Док се у редефинисаној методи *OnUpdate()* врши ажурирање позиције играча на основу тастатуре, тзв. WASD управљање гдје се на основу тога који од тастера на тастатури корисник притисне врши помјерање играча у одговарајућем смјеру у 2Д простору.

void PlayerSystem::OnCreate(){

entityManager = ECS::ECS\_Engine->GetEntityManager();

movementComponent = entityManager->CreateComponentType<Movement>();

playerComponent = entityManager->CreateComponentType<PlayerComponent>();

Entity\*\*entities=entityManager->GetAllEntitiesWithType<PlayerComponent>(&numberOfEntites);

for (int i = 0; i < numberOfEntites; i++) {

Movement m = Movement();

m.x = 0; m.y = 0;

entityManager->SetComponent(\*entities[i], m);

**...** //init all components of entities

}

}

void PlayerSystem::OnUpdate(){

std::initializer\_list<ComponentType> types = { movementComponent, playerComponent };

uint8\_t\*\*\*playerComponents=entityManager->GetComponentsWithTypes(types,&numberOfEntites);

for (int i = 0; i < numberOfEntites; i++){

Movement\* m = (Movement\*)playerComponents[0][i];

PlayerComponent\* p = (PlayerComponent\*)playerComponents[1][i];

int ch = getch\_noblock();

if (ch!=-1) {

switch (ch){

case 'W':

m->x += 100 \* p->speed \* ECS::timeStep;

**...** //move entity in right direction

}

} //end if

} //end for

} //end method

Наредни дио којим управља један од система јесте промјена позиције непријатеља играча која функционише другаћије него кретање играча, па због тога постоји одвојен систем за то, *EnamySystem*. Непријатељи имају предефинисане двије позиције између којих се крећу, а транзиција између ове двије позиције дешава се на сваких *patrolTime*. Метода *OnCreate()* има врло сличну имплементацију као систем за играче, гдје се врши иницијализација компоненти непријатеља, па због тога није приказана имплементација ове методе. Са друге стране, метода *OnUpdate()* има другачију логику, али систем дохватања свих компоненти је исти само овај пут су нам потребне друге врсте компоненти.

void EnemySystem::OnUpdate(){

std::initializer\_list<ComponentType> types = { movementComponent, lifeTimeComponent, enamyComponent};

uint8\_t\*\*\* enemyComponents=entityManager->GetComponentsWithTypes(types,&numberOfEntites);

for (int i = 0; i < numberOfEntites; i++){

Movement\* m = (Movement\*)enemyComponents[0][i];

**...** //get another components too

if (e->patrolTime <= e->tempPatrolTime) {

e->tempPatrolTime = 0;

**...** //change position

}

e->tempPatrolTime += ECS::timeStep;

l->time -= ECS::timeStep;

}

}

Последњи систем јесте *LifetimeSystem* који дохвата све енитете који имају *Lifetime* компоненту, тј. имају ограничен вијек трајања у игри и врши њихово уништавање ако је истекло предефинисано вријеме. Овдје можемо видјети једну предност ECS архитектуре која се огледа у томе, ако нам се у некој наредној верзији игре појаве нове врсте ентитета које такође имају ограничен вијек трајања ми не морамо да кодирамо нове системе, јер већ имамо систем који то рјешава само је потребно да новим ентитетима додамо *Lifetime* компоненту. Из овог примјер се јасно види лакоћа поновне употребе кода.

void LifetimeSystem::OnUpdate(){

entityManager = ECS::ECS\_Engine->GetEntityManager();

Entity\*\* entities = entityManager->GetAllEntitiesWithType<Lifetime>(&numberOfEntites);

for (int i = 0; i < numberOfEntites; i++) {

Lifetime lifetime;

entityManager->GetComponent(\*entities[i], &lifetime);

if (lifetime.time <= 0) entityManager->DestroyEntity(\*entities[i]);

}

}

За крај овог примјера, остало је још да се представи како се цијела архитектуре покреће и како се ентитети, компоненте и системи додају у игру. У коду испод приказана је *main()* функција која у програмском језику С++ представља почетну тачку извршавања. То је мјесто гдје се покреће читав систем, тј. гдје се позивају методе *Initialize()* за покретање система и *Terminate()* за гашење истог, а које су споменуте у потпоглављу 4.1.3.

Позив методе *Game1()* представља мјесто гдје се врши иницијализације цијеле игре тј. креирање свих типова компоненти, самих компоненти, затим ентитета, система и свих потребних измјена истих користећи инстанце *EntityManager* и *SystemManager* класе и свих њихових метода описаних у потпоглављима 4.1.1 и 4.1.2.

int main() {

ECS::Initialize();// initialize global 'Engine' object

const float DELTA\_TIME\_STEP = 1.0f / 60.0f; // 60hz

ECS::timeStep = DELTA\_TIME\_STEP;

bool bQuit = false;

Game1();

// run main loop until quit

while (bQuit == false){

ECS::ECS\_Engine->Update(DELTA\_TIME\_STEP); // Update all Systems

std::this\_thread::sleep\_for(std::chrono::milliseconds(60));

}

ECS::Terminate();// destroy global 'Engine' object

return 0;

}

Посматрајући све поменуте функционалности, може се закључити да је ово добар почетак за развој једног програма за развој игара. Неке од ствари које би биле прве на реду за имплементацију у случају развоја таквог програма јесу следеће:

* *Editor* - визуелно управљање ентитетима, компонентама, системима
* *GUI* - изградњa интерактивног менија за игре
* *Networking* - слање догађаје преко мреже и подешавање режима за више играча
* *Resource Manager* - учитавање средстава као што су текстуре, фонтови и сл.
  1. *ECS* тестови перформанси

Као закључак на све споменуто до сада и како би потврдили све оно о чему је причано у предходним поглављима, извршено је пар тестова перформанси имплементиране архитектуре и исти су упоређени са перформансама при објектно оријентисаном дизајну.

Први примјер дефинише низ на два различита начина и мјери брзине итерирања кроз низ и ажурирање елемента тог низа. Низ је прво дефинисан као низ показвача на једну од компоненти система *PlayerComponent*, баш као што се то ради у објектно-оријентисаном дизајну. Извршено је итерирање кроз и низ и ажурирање промјењиве *float speed* која је поље ове класе компоненте. Након тога направељен је низ ентитета који садрже само компоненту *PlayerComponent* и дохватање тих комопоненти из меморије. Извршено је исто итерирање и ажурирање и кроз овај низ и добијени су следећи резултати приказани у табели 4.3.1.

Табела 4.3.1 Брзина итерирања и ажурирања низа у OOP и ECS

|  |  |  |
| --- | --- | --- |
| **Број елемената низа** | **Објектно-оријентисано (µs)** | **ECS (µs)** |
| 10 000 | 55 | 17 |
| 100 000 | 453 | 152 |
| 300 000 | 938 | 388 |
| 500 000 | 4859 | 852 |

Интересантно је било видјети шта се дешава са брзином итерирања и ажурирања над истим низовима ако се промјени величина компоненте. Резултати добијени за различите величине компоненти и низове са 100 000 и 500 000 елемената приказани су у табелама 4.3.2 и 4.3.3.

Табела 4.3.2 Брзина итерирања и ажурирања низа у OOP и ECS   
за различите величине компоненти за низ од 100 000 елемената

|  |  |  |
| --- | --- | --- |
| **Величина компонент (Byte)** | **Објектно-оријентисано (µs)** | **ECS (µs)** |
| 4 | 345 | 131 |
| 12 | 618 | 287 |
| 24 | 664 | 472 |

Табела 4.3.3 Брзина итерирања и ажурирања низа у OOP и ECS   
за различите величине компоненти за низ од 500 000 елемената

|  |  |  |
| --- | --- | --- |
| **Величина компонент** | **Објектно-оријентисано (µs)** | **ECS (µs)** |
| 4 | 1933 | 711 |
| 12 | 3264 | 1297 |
| 24 | 4126 | 2554 |

Из приложених резултата може се примјетити да коришћење ECS тј. дизајна оријентисаног на податке даје увијек запажајуће брже резултате. Код мањих величина компоненти, без обзира на дужину низа, за ову имплементацију то је чак и до 3 пута бржи приступ. Док за низове објеката које заузимају више меморије нпр. 24B имамо нешто лошије перформансе, али и даље до дупло брже него код објектно-оријентисаног дизајна. Нешто слабије перформансе код већих објеката јесу настале као утицај да у *chunk* меморије може да стане мањи број компоненти, па је потребно чешће узимати нови *chunk* што утиче на перформансе. Због тога, али и због лакоће управљања компонентама постоји препорука да компоненте буду мале групе података.

1. Резултати анализе и дискусија

Након описане саме реализације *Entity Component System* архитектуре и приказаних примјера коришћења исте, у наредним потпоглављима биће продискутоване додатне могућности како саме имплементације *ECS* архитектуре, тако и дизајна оријентисаног на податке генерално. Осим тога биће представљене и одређени недостаци ове архитектури који су примјећени у току рада.

* 1. Додатне могућности

Свакако, осим имплементације саме *Entity Component Sytem* архитектуре, која увелико доприности перформансама комплексних видео игара, како би се употпунио *ЕCS* могуће је имплементирати још низ додатних могућности. Таква архитектура окруженa низом побољшања назива се *Data-Oriented Technology Stack* *DOTS*, а примјере додатних могућности најлакше је пронаћи код великих компанија као што је *Unity* и њихових алата.

У рачунарском систему са једном нити, само по једна инструкција се може учитати у неком тренутку и излази само по један резултат. Вријеме учитавања и довршавања програма зависи од количине посла коју *CPU* треба да обави. *Multithreading* је врста програмирања која користи способност *CPU* да истовремено обрађује више нити у више језгара. Умјесто да се задаци или упутства извршавају један за другим, они се извршавају истовремено. Такав примјер можемо да пронађемо у *Unity* тзв. *C# Job System* који се побринуо да пружи програмерима погодну платформу за писање сигурног кода који се извршава у више нити, што додатно доприноси перформансама. Посао (енг. *Job*) је мала јединица рада која обавља један одређени задатак. Посао прима параметре и оперише подацима, слично ономе како се понаша позив методе. Послови могу бити самостални или могу зависити од других послова које треба обавити прије него што се покрену.

Наравно, не морамо да се зауставимо ни на овоме. *Unity* je одлучио да својим корисницима, пружи могућност додатних перформанси тако што је направио нови програмски преводилац такозвани *Burst Compiler*. Овај преводилац узима споменуте послове и производи високо оптимизовани машински код.

Дизајн оријентисан на податке, иако релативно нов дизајн, представља област за велики развој и напредак када је у питању развој видео игара. Развој аудио система, система за анимације, као и систем умрежавања код игара са више играча представљају само неке од поља на којима дизајн заснован на подацима тек треба да покаже своју моћ.

* 1. Недостаци архитектуре

Иако су до сада спомињање само предности ECS архитектуре, битно је знати да постоје одређене мане, не само у имплементацији описаној у овом раду већ и у осталим програмима који се користе за развој игара, а имају имплементиран ECS:

* Системи су међусобно зависни од редослиједа њиховог извршавања. Додавање новог система „између“ већ два постојећа може бити изазов и веома лоше за перформансе.
* Потребно је испланирати потребне компоненте и податке што је раније могуће, јер ће их потенцијално користити доста система, па би промјена садржаја компоненти могла утицати на рад више система.
* Иако је врло лако пратити извршавање система, са друге стране веома је тешко пратити стање појединачне компоненте и њене промјене. Такође не постоји глобални поглед на ентитет и шта се дешава са свим његовим промјенама.

Осим проблема везаних за саму имплементацију, постоје одређене мане везане за информисање о ECS-у и документацији:

* Крива учења је врло стрма и непрестано се наилази на проблеме које је тешко рјешити без тражења рјешења на форумима или без заобилажења.
* ECS је је још увек у развоју чак и у напредним програмим као *Unity*
* API се и даље непрестано мијења
* Много документације недостаје или је застарјело. Такође је тешко пронаћи детаљне информације на *Google*-у. Често да би схватили неке концепте потребно је да сами тумачимо изворни код других имплементација ECS-a које постоје.

Ако посматрамо са гледишта програмера, постоји још једна ствар о ECS архитектури, коју можемо сврстати у мане. Концепт ECS архитектуре је доста другачији од OOP програмирања на који смо сви навикли, и врло је тешко промјенити сам начин размишљања, па је прилагођавање програмера на рад са новом архитектуром отежавајуће.

1. Закључак

У овом раду, парадигма заснована на архитектури оријентисаној на подацима предложена је као алтернатива објектно оријентисаном програмирању приликом развоја видео игара. Теорија и пракса је до сада показала да је *ООP* једана од најраспрострањенијих архитектура због једноставности поимања програмских проблема, могућности проширивања кода и слично. Свим тим поступцима *ООP* је постајао лакши за рад програмеру, али тежи процесору. Како би се рјешио проблем перформанси и одржавања кода програмери су одлучили да развију и користe *ECS*.

У основи *ECS*-а је раздвајање објеката на три одвојена елемента, ентитете који су идентификационе цјелине, компоненте које садрже податке и системе који пружају логику. Главна идеја код дизајна оријентисаног на податке јесте секвенцијално смјештање компоненти у меморији и паметно коришћење кеш меморије, која је знатно бржа од системске меморије. Како би овакав начин управљања меморијом био могућ имплементирани су менаџери који су задужени за креирање, ажурирање и уништавање ентитета, компоненти и система.

Преласком са објектно оријентисаног дизајна на дизајн оријентисан на податке, програмеру постаје много лакше да поново користи свој код, а другима да га разумију и раде на њему. Изненађујуће је лако додати нове ствари у игру, нове компоненте и системе. Када се једном постави богат темељ ентитета, компонената и система, готово је дјечија игра спојити их и из њих направити игру. Главна предност ове парадигме је што омогућава кориснику да управља својим подацима на флексибилан начини раздвајањем објеката на компоненте, којима се може управљати у дјеловима за мноштво ентитета у исто вријеме. Ова техника је изузетно корисна када маса треба поступати на сличан начин. Због овога, парадигма има историју и данас се користи у развоју игара.

Битно је споменути да постоје и многи други начини имплементације структура података од приказаних у овом раду, али је овај начин изабран као један од оних са најоптималнијим перформансама. Смјештање компоненти секвенцијално у меморији у комаде меморије који су величине L1 кеша, дали си три до четри пута брже перформансе код компоненти мање величине од 4В до 12В, док код нешто већих компоненти имамо мало слабије убрзање, али опет два до три пута брже него што је то у *ООP.* Такође, из добијених резултата може се примјетити да са повећањем броја елемената низа перформансе *ECS*-а не слабе у односу на перформансе *ООP*.

Поред свих предности *ECS* ипак није савршен одговор на све. Писање кода за изоловане објекте у *ECS* архитектури уводи одређене режијске трошкове и често се тешко навићи на рад са самом парадигмом, што може одложити почетну тачку пројекат значајно. Свакако ако је циљ постизање оптималних перформанси, што у развоју AAA игара врло често и јесте, *ЕCS* архитектура је неприкосовенa.

Једна веома битна могућност, као будући корак, јесте лакоћа паралелизације. Чувањем података што је више могуће само за читање омогућава дистрибуцију између вишеструко безбједних нити без страха од грешака у синхронизацији. Сваки систем би могао да покреће по једну такву нит у овкиру сваког позива *OnUpdate()* методе што би додатно убрзало извршавање. Такође развој аудио система, система за анимације, као и система умрежавања код игара са више играча представљају само неке од система које је могуће имплементирати како би се корисницима олакшао рад у *ECS*-у.

Циљ овог рада јесте био имплементација *Entity Component System* архитектуре како би се приказале њене предности у односу на *ООP* при развоју игара. Свако то није крајња тачка овог рада, јер оваква имплементација има озбиљан потенцијал да прерасте у мали g*ame engine* тј. програм за развој видео игара додавањем едитора, графичког корисничког интерфејса и свих остали дјелова које чине један такавпрограм.

Литература

1. Steven Lambert, "Intro to Object-Oriented Programming for Game Development" (<https://gamedevelopment.tutsplus.com/tutorials/quick-tip-intro-to-object-oriented-programming-for-game-development--gamedev-1805#:~:text=What%20Is%20Object%2DOriented%20Programming,Crysis%20has%20over%20a%20million).>,05.09.2020.)
2. Toni Härkönen, "Advantages and Implementation of Entity-Component-System" (<https://trepo.tuni.fi/bitstream/handle/123456789/27593/H%C3%A4rk%C3%B6nen.pdf?sequence=4&isAllowed=y>, 05.09.2020)
3. Vittorio Romeo, "Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library", July 2016. (<https://www.researchgate.net/publication/305730566_Analysis_of_entity_encoding_techniques_design_and_implementation_of_a_multithreaded_compile-time_Entity-Component-System_C14_library>, 06.09.2020.)
4. M. West, "Evolve your Hierarchy", 5 January 2017 (<http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>, 06.09.2020.)
5. S. Bilas, "A Data-Driven Game Object System, Game Developer’s Conference", USA 2002. (<https://www.gamedevs.org/uploads/data-drivengame-object-system.pdf>., 06.09.2020.)
6. Arne Held, "Unity's “Preformance by Default“ under the hood", December 9,2019 (https://tech.innogames.com/unitys-performance-by-default-under-the-hood/, 09.09.2020.)
7. *Get Started with the Unity\* Entity Component System (ECS), C# Job System, and Burst Compiler*,(https://software.intel.com/content/www/us/en/develop/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler.html/, 15.09.2020.)
8. *Entity Component System*,([https://docs.unity3d.com/Packagescom.unity.entities@0.2/ manual/index.html/](https://docs.unity3d.com/Packagescom.unity.entities@0.2/%20manual/index.html/) , 15.09.2020.)
9. Tobias Stein, "The Entity-Component-System – An awesome game-design pattern in C++", November 22, 2017 ( [https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/ The\_EntityComponentSystem\_\_An\_awesome\_gamedesign\_pattern\_in\_C\_Part\_1.php](https://www.gamasutra.com/blogs/TobiasStein/20171122/310172/%20The_EntityComponentSystem__An_awesome_gamedesign_pattern_in_C_Part_1.php), 16.09.2020)

[10] *The bountyhunter game*, ([https://tsprojectsblog.wordpress.com/2017/11/16/the-bounty hunter-game/](https://tsprojectsblog.wordpress.com/2017/11/16/the-bounty%20%20%20hunter-game/), 16.09.2020)

[11] *ECS Memory Layout*, ([https://forum.unity.com/threads/ecs-memory-layout.532028](https://forum.unity.com/threads/ecs-memory-layout.532028%20) , 18.09.2020)

Списак скраћеница

|  |  |
| --- | --- |
| AAA | *Triple-A* |
| API | *Application Programming Interface* |
| CPU | *Central Processing Unit* |
| DOD | *Data-Oriented Design* |
| DOTS | *Data-Oriented Technology Stack* |
| ECS | *Entity Component System* |
| GUI | *Graphical User Interface* |
| OOP | *Object-Oriented Programming* |
| RAM | *Random Access Memory* |
| SIMD | *Single instruction, multiple data* |
| VFX | *Visual effects* |

Списак слика

[Слика 2.4.1.1 Хијерархија класа аутомобила 3](#_Toc51870505)

[Слика 2.4.1.2. Дијамант структура класа 4](#_Toc51870506)

[Слика 2.4.1.3 Хијерархија објектно оријентисане композиције 4](#_Toc51870507)

[Слика 2.4.2.1 Објекти у ООP повезани великим бројем референци 5](#_Toc51870508)

[Слика 3.2.1 Ентитети, компоненте и системи као цјелина 7](#_Toc51870509)

[Слика 3.3.1.1 Примјер више *GameObject*-а и компоненти у ООP дизајну *Unity* 8](#_Toc51870510)

[Слика 3.3.1.2 Референце у меморије између објекат, њиховог понашања и њихових компоненти 9](#_Toc51870511)

[Слика 3.3.1.3 Чланови који се стварно користе за операцију кретања и изгубљени простор 10](#_Toc51870512)

[Слика 3.3.1.4 *Еntity-Component-System* примјер 10](#_Toc51870513)

[Слика 3.3.1.5 Фрагментација у кешу и неискоришћени простор при ОО дизајну 11](#_Toc51870514)

[Слика 3.3.1.6 Распоред низоба и класа у меморији С# 12](#_Toc51870515)

[Слика 3.3.2.1 Поређење перформанси C# 12](#_Toc51870516)

[Слика 4.1.1 Aрхитектура ECS система 14](#_Toc51870517)

[Слика 4.1.1.1 Archetypes у ECS-у 14](#_Toc51870518)

[Слика 4.1.1.2 Archetypes и Chunks у ECS-у 15](#_Toc51870519)

[Слика 4.2.1 Преглед ентитета и компоненти у ECS примјеру 18](#_Toc51870520)

[Слика 4.2.2 Преглед класа у примјеру коришћења ECS архитектуре 20](#_Toc51870521)

Списак табела

[Табела 4.3.1 Брзина итерирања и ажурирања низа у OOP и ECS 23](#_Toc51870502)

[Табела 4.3.2 Брзина итерирања и ажурирања низа у OOP и ECS за различите величине компоненти за низ од 100 000 елемената 23](#_Toc51870503)

[Табела 4.3.3 Брзина итерирања и ажурирања низа у OOP и ECS за различите величине компоненти за низ од 500 000 елемената 24](#_Toc51870504)

1. Имплементација

У оквиру овог поглавља биће представљен код који је имплементација Entity Component System архитектуре. Значење и функција појединачних дијелова ECS архитектуре објашњени су у предходним поглављима, док ће овдје бити приказана имплементација у програмског језику С++. Фокус ће бити на приказивању интерфејса, док ће имплементација бити приказана само за значајније методе.

* 1. Entity

Сваки ентитет представља објекат класе *Entity* која од података садржи само цјелобројни идентификациони број. Као што је раније објашњено ентитети не садрже ни податке ни логику извршавања.

class Entity {

private:

uint32\_t id;

public:

uint32\_t GetID();

void SetID(uint32\_t i);

};

Како би се сачувале додатне информације о „комаду меморије“ у ком се чува одређени енитет и позицији у истом користи се додатна класа *EntityInChunk*. У оквиру ове класе чува се показивач на *chunk* у коме се чува ентитет и индекс ентитета у том *chunk*-у.

class EntityInChunk {

private:

Chunk\* chunk;

int indexInChunk;

public:

ECS::Chunk\* GetChunk();

void SetChunk(Chunk\* chunk);

int GetIndexInChunk();

void SetIndexInChunk(int index);

};

* 1. Component

Свака компонента која постоји у систему мора да буде изведена из класе *IComponent*. Ова класа нема поља и методе, већ ће корисник наслеђивањем ове класе да дода потребне податке.

class IComponent { };

Да би лакше било разликовати типове компоненти при манипулацији са компонентама, користи се класа *ComponentType* која има информацију о типу компоненте добијену на основу оператора *typeid(type);* из заглавља <typeinfo>. Поред информације о типу чува се величина компоненте и индекс тј. јединствени идентификатор компоненте.

class ComponentType {

private:

static int INDEX;

int index;

int size;

const type\_info\* type;

public:

ComponentType() {}

ComponentType(const type\_info& type, int size);

ComponentType& operator=(const ComponentType& compType);

int GetIndex();

int GetSize();

const type\_info& GetTypeInfo();

};

* 1. System

Имплементација систем се заснива на двије класе. Прва је апстрактна класа *ISystem* који представља интерфејс који сви будући системи треба да имају. Од атрибута садржи промјенљиву која објашњава да ли је систем омогућен или неомогућен за извршавање. Такође, садржи и три апстрактне методе које су објашњење у потпоглављу 4.1.2, те апстрактну методу за дохватање јединственог индетификатора система, која ће бити редефинисана у нардној класи *System*.

class ISystem {

private:

bool enabled;

public:

ISystem();

~ISystem();

virtual inline const size\_t GetSystemTypeID() const = 0;

virtual void OnCreate() = 0;

virtual void OnDestroy() = 0;

virtual void OnUpdate() = 0;

bool IsEnabled();

void Enable();

void Disable();

};

Наредна класа која је значајна за имплементацију система је класа *System* која је изведена из изнад наведене апстрактне класе. У оквиру ове класе редефинисане су све виртуелне апстракне методе тако да садрже празно тијело, у случају да корисник у новонаправљним системима не редефинише исте оне ће имати подразумјевано понашање, тј. неће радити ништа. Веома значајан детаљ јесте да је ова класа, *template* класа, и ствара се по једна за сваки тип система. Тиме је омогућено да може да све класе истог типа система имају исти идентификациони број, који се добија на помоћу *template* класе *TypeID*.

template<class T>

class TypeID

{

private:

static size\_t s\_count;

public:

template<class U>

static const size\_t Get(){

static const size\_t STATIC\_TYPE\_ID{ s\_count++ };

return STATIC\_TYPE\_ID;

}

static const size\_t Get(){ return s\_count; }

};

template<class T>

class System : public ISystem {

friend class SystemManager;

public:

static const size\_t STATIC\_SYSTEM\_TYPE\_ID;

private:

SystemManager\* systemManagerInstance;

public:

System() { }

virtual ~System(){ OnDestroy(); }

virtual inline const size\_t GetStaticSystemTypeID() const{

return STATIC\_SYSTEM\_TYPE\_ID;

}

void SetSystemManagerInstance(SystemManager\* manager);

template<class Dependencie>

void AddDependencies(Dependencie dependencie){

this->systemManagerInstance->AddSystemDependency(this, dependencie);

}

virtual void OnCreate() override { }

virtual void OnDestroy() override { }

virtual void OnUpdate() override { }

};

template<class T>

const size\_t System<T>::STATIC\_SYSTEM\_TYPE\_ID = TypeID<ISystem>::Get<T>();

* 1. EntityManager

Као што је објашњено у поглављу 4.1.1 класа *EntityManager* је одговорна за управљене ентитетима и компонентама. Да би се омућило што лакше, брже и флексибилније управљање ова класа мора да садржи неке додатне структуре података. Најприје, потребно је било за сваки ентите запамтити коме *archetype*-у и *chunk*-у припада сваки ентитет. Због тога су увдене два низа један који памти показивач на *archetype,* Archetype\*\* archetypeByEntity и други који памти *chunk* и позицију ентитета у том *chunk-*у користећи класу *EntityInChunk* споменуту у поглављу А.1. EntityInChunk\* entityInChunkByEntity. У оба ова низа одговарајућем елементу приступамо на основу јединственог идентификатора енитета. Поред ових низова постоји низ у коме се памте сви до сада директно или индиректно креирани *archetype*-ови Archetype\* archetypes. Поред ових низива постоје само додатне информације о величинама иситих и тренутном броју елемената.

class EntityManager {

private:

Archetype\*\* archetypeByEntity;

EntityInChunk\* entityInChunkByEntity;

Archetype\* archetypes; //array of all archetypes which exists

int countArchetypes;

int capacityInArchetypeArray;

int entitiesCapacity;

int nextFreeEntityIndex;

Chunk\* GetChunkWithEmptySlots(Archetype\* archetype);

Archetype\* GetOrCreateArchetypeWithTypes(std::initializer\_list<ComponentType> types);

Archetype\* GetOrCreateArchetypeWithTypes(std::list<ComponentType> types);

template<class T> uint8\_t\* GetComponentPtr(Entity entity);

void IncreaseCapacity(int value);

public:

EntityManager();

~EntityManager();

template<class T> void GetComponent(Entity entity, T\* value);

template<class T> void SetComponent(Entity entity, T componentData);

template<class T> bool HasComponent(Entity entity);

bool HasComponent(Entity entity, ComponentType componentType);

int GetComponentCount(Entity entity);

bool AddComponent(Entity entity, ComponentType componentType);

bool RemoveComponent(Entity entity, ComponentType componentType);

Archetype\* CreateArchetype(std::initializer\_list<ComponentType> types);

Entity CreateEntity(std::initializer\_list<ComponentType> types);

void DestroyEntity(Entity entity);

void DestroyEntities(std::initializer\_list<ComponentType> types);

int CountEntities();

std::vector<Entity> GetAllEntities();

template<class T> ComponentType CreateComponentType();

void SetArchetype(Entity entity, Archetype\* archetype);

template<class T>T\*\* GetComponentsWithType(int\* count);

IComponent\*\*\* GetComponentsWithTypes(std::initializer\_list<ComponentType> types, int\* countComponents);

template<class T> Entity GetAllEntitiesWithType(int\* count);

};

* + 1. Chunk

*Сhunk* представља комад меморије који се састоји од заглавља и централног дијела у коме се чувају компоненте. У имплементацији *chunk* је представљен као класа која садржи основне информације о *chunk*, као што су показивач на *archetype* коме припада, укупан број енитетата који могу да се смјесте са свим компонентама у *chunk*, као и тренутан број таквих ентитета. Сваки *chunk* такође има свој јединствени идентификатор.

Значајно поље у овој класи јесте uint8\_t buffer[4] које се налази на почетку дјела *chunk* који садржи компоненте, па је дохватањем овог бафера врло лако прескочити заглавље и приступити компонентама.

class Chunk {

private:

static unsigned long SequenceNumber; // Incrementing automatically for each chunk

// Chunk header START

Archetype\* archetype;

uint32\_t count; //current number of entites in chunk

uint32\_t capacity; //number of entities which fits in chunk

unsigned long id;

// Chunk header END

static const int kBufferOffset = 64; // (must be cache line aligned)

uint8\_t buffer[4] = { 0,0,0,0 };

static const int kChunkSize = 16 \* 1024 - 256;

static const int kBufferSize = kChunkSize - kBufferOffset;

public:

void Init(Archetype\* arch);

static Chunk\* MallocChunk(){

return (Chunk\*)malloc(kChunkSize);

}

uint32\_t GetCount() ;

void IncrementCount;

void DecrementCount;

void SetCount(int i);

int GetCapacity();

void SetCapacity(uint32\_t capacity);

void SetArchetype(Archetype\* arch);

Archetype\* GetChunkArchetype();

static int GetChunkBufferSize();

uint8\_t\* GetBuffer() ;

int GetId;

bool Has(ComponentType componentType);

};

* + 1. Archetype

*Аrchetype* је представљен као класа који садржи низ типова који представљају дати *archetype.* За сваки тип компоненете у овој класи било је потребно запамтити и помјерај од почетка *chuka*-a до дате компоненте, као и величину исте компоненте у бајтовима. Осим тога, као што је већ објашњено сваки *archetype* садржи више *chunk*-ова у којим се чувају компоненте па је потребно сачувати низ *chunk*-oва за сваки *archetype*. Уколико се неки од *chunk*-oва у међувремену ослободи такав *chunk* потребно је сачувати у низ празних *chunk*-oва.

class Archetype {

private:

int entityCount;

int chunkCapacity; //Number of entites which fits in chunk of this archetype

ComponentType\* types;

int typesCount;

int\* offsets;

int\* sizeOfs;

ArchetypeChunkArray\* chunks;

Chunk\*\* chunksWithEmptySlots;

int numberOfChunksWithEmptySlots;

int capacityChunkWithEmptySlots;

int CalculateChunkCapacity(int bufferSize, int\* componentSizes, int count);

int CalculateSpaceRequirement(int\*componentSizes,int componentCount,int entityCount);

int Align(int size, int alignmentPowerOfTwo);

public:

Archetype(std::initializer\_list<ComponentType> componentTypes);

Archetype(std::list<ComponentType> componentTypes);

Archetype(const Archetype&);

Archetype(Archetype&&);

~Archetype();

Archetype& operator=(const Archetype& t);

int GetTypesCount();

ComponentType\* GetTypes();

void SetTypesCount(int count);

int GetChunkCapacity();

void IncerementEntityCount();

void DecrementEntityCount();

void SetCount(int i);

int GetEntityCount();

void IncEntityCount();

int GetOffset(int index);

int GetSizeOf(int index);

ArchetypeChunkArray\* GetChunksArray();

Chunk\* GetExistingChunkWithEmptySlots();

void AddToChunkListWithEmptySlots(Chunk\* chunk);

void AddToChunkList(Chunk\* chunk);

Entity\* AllocateEntity(ComponentType\* types);

int GetIndexInTypeArray(ComponentType type);

template<class T> int GetIndexInTypeArray();

ComponentType GetTypeInTypeArray(int index);

bool Compare(std::initializer\_list<ComponentType> componentTypes);

bool Compare(std::list<ComponentType> componentTypes);

string toString();

};

Како би *EntityManager*-у било једноставније да управља свих *archetype*-овима, направљена је додатна класа class ArchetypeChunkArray која чува низ *archetype*-ова и омогућава низ функционалности за управљање тим низом.

class ArchetypeChunkArray {

private:

Chunk\*\* chunks;

int count;

const int ChunkPtrCapacity = 10;

public:

ArchetypeChunkArray();

~ArchetypeChunkArray();

ArchetypeChunkArray(const ArchetypeChunkArray& oldArch);

ArchetypeChunkArray(ArchetypeChunkArray&& oldArch);

ArchetypeChunkArray& operator=(const ArchetypeChunkArray& oldArch);

int CalculateEntityCount();

void Add(Chunk\* chunk);

int GetNumberOfChunks();

Chunk\* GetChunkByIndex(int index);

Chunk\* GetCurrentChunk();

void SetNull(int id) ;

};

* 1. SystemManager

*SystemManager* представљен је као класа која је одговорна за управљање системима. У те сврхе у оквиру класе чува се хеш мапа која слика јединствени идентификатор сваког система, у сам показивач на објекат система unordered\_map<int, ISystem\*>. Ова мапа је потребна како би имали евиденцију у свим системима који су креирани и потребно је да се извршавају. Пошто је већ објашњено да системи зависе једни од других па је потребно знати и њихов редослијед извршавања чува се вектор система у редослиједу њиховг извршавања, као и матрица која представња граф зависности између система. Ова матрица је квадратна оних димензија колико има система. Вриједност true у неком пољу матрице означава да постоји зависност између два система са идентификаторима врсте и колоне матрице.

Методе тј. функционалности *SystemManager*-а објашњене су у поглављу 4.1.2.

class SystemManager {

private:

vector<ISystem\*> systemWorkOrder;

unordered\_map<int, ISystem\*> systems;

vector<std::vector<bool>> systemDependencyMatrix;

public:

SystemManager();

~SystemManager();

template<class T>

ISystem\* AddSystem();

template<class System\_, class Dependency\_>

void AddSystemDependency(System\_ target, Dependency\_ dependency);

template<class T>

T\* GetSystem() const;

template<class T>

void EnableSystem();

template<class T>

void DisableSystem();

void Update(float\_t dt\_ms);

void UpdateSystemWorkOrder();

};

* 1. Timer

Tајмер представља посебну цјелину која је представљен као класа class Timer и служи да би се у оквиру ње чувало вријеме система, на основу кога се врше ажурирања свих система у архитектури. Уз ову класу коришћена је и додатна унија која служи ради лакшег дохватња и праћења тренутног времена.

union TimeStamp{

float\_t asFloat;

uint32\_t asUInt;

TimeStamp() : asUInt(0U){}

TimeStamp(float\_t floatValue) : asFloat(floatValue){}

operator uint32\_t() const { return this->asUInt; }

inline const bool operator==(const TimeStamp& other) const;

inline const bool operator!=(const TimeStamp& other) const;

inline const bool operator<(const TimeStamp& other) const;

inline const bool operator>(const TimeStamp& other) const;

};

class Timer {

using Elapsed = std::chrono::duration<float\_t, std::milli>;

private:

Elapsed m\_Elapsed;

public:

Timer();

~Timer();

void Tick(float\_t ms);

void Reset();

inline TimeStamp GetTimeStamp() const {

return TimeStamp(this->m\_Elapsed.count());

}

};

* 1. Еngine

*Engine* представља класу која држи остале дјелове архитектуре на окупу. Она представља основу за даље коришћење архитектуре, јер садржи инстанцу класе *Timer*, *EntityManager* и *SystemManager*. Садржи методе за дохватање истих, као и методу упдате која покреће ажурирање и извршавање сви активних система.

class Engine {

private:

Timer\* engineTime;

EntityManager\* entityManager;

SystemManager\* systemManager;

public:

Engine();

~Engine();

EntityManager\* GetEntityManager() { return entityManager; }

SystemManager\* GetSystemManager() { return systemManager; }

void Update(float tick\_ms);

};

void ECS::Engine::Update(float tick\_ms){

// Advance engine time

engineTime->Tick(tick\_ms);

// Update all running systems

systemManager->Update(tick\_ms);

}

* 1. API

На почетку сваке апликације која би потенцијално користила ову имплеметацију потребно је да се позове метода void Initialize() која креира инстанцу класе *Еnginе* која је даље задужена за све компоненте архитектуре. Са друге стране, на крају такве апликације потребно је да се позове метода void Terminate() која уништава објекат класе *Еnginе*.

extern Engine\* ECS\_Engine;

void Initialize();

void Terminate();