Универзитет у Београду

Електротехнички факултет



Предности и имплементација Entity Component System архитектуре

Дипломски рад

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

Београд, Септембар 2020.

Садржај

[Садржај i](#_Toc50623227)

[1. Увод 1](#_Toc50623228)

[2. Традиционална архитектура 2](#_Toc50623229)

[2.1. Објектно-орјентисано програмирање 2](#_Toc50623230)

[2.2. Развој видео игара коришћењем OOP 2](#_Toc50623231)

[2.3. Предности 3](#_Toc50623232)

[2.4. Мане 3](#_Toc50623233)

[2.4.1. Одржавање 3](#_Toc50623234)

[2.4.2. Преформансе 5](#_Toc50623235)

[3. Архитектура орјентисана на податке 6](#_Toc50623236)

[3.1. Историја 6](#_Toc50623237)

[3.2. Entity-Component-System архитектура 7](#_Toc50623238)

[3.3. Поређење са објектно-орјентисаним дизајном 7](#_Toc50623239)

[3.3.1. Добјиање на брзини коришћењем кешa 7](#_Toc50623240)

[3.3.2. Поређење преформанси 11](#_Toc50623241)

[4. Детаљи имплементације 12](#_Toc50623242)

[4.1. Entity-Component-System 12](#_Toc50623243)

[4.1.1. Управљање ентитетима и компонентама 13](#_Toc50623244)

[4.1.2. Управљање системима 13](#_Toc50623245)

[4.2. ECS примјери 13](#_Toc50623246)

[5. Додатне могућности 14](#_Toc50623247)

[6. Порблеми 15](#_Toc50623248)

[7. Закључак 16](#_Toc50623249)

[Литература 17](#_Toc50623250)

[Списак скраћеница 18](#_Toc50623251)

[Списак слика 19](#_Toc50623252)

[Списак табела 20](#_Toc50623253)

1. Увод

Игре су специфична врста апликација због тога што се од њих константно очекује висок ниво преформанси. 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 преформансе 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 преформансе трпе.

Да би се рјешио овај проблем, традиционални објектно орјентисани 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 орјентисан 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 орјентисаног на податке, а не објекте. Имплементирати платформу која ће детаљно приказати начин функционисања ЕCS архитектуре и на који начин она доприноси флексибилности и бољим преформанса саме апликације.

У наставку рада, конкретно у поглављу 2 биће објешњен традиционални начин кодирања, како би се боље могла направити паралела са нови дизајном орјентисаним на податке. У поглављу 3...

1. Традиционална архитектура

Објектно-Орјентисано програмирања (*eng. Object-Oriented programming ООP*) је начин да се организује код при развоју видео игара. У овом поглављу наведени су основни принципи Објектно-Орјентисаног 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-орјентисано 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* четри ноге и слично.

**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 објшњава 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-орјентисани 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 *GameObject* 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 јадан од основих концепата објектно-орјентисаног програмирања.

* 1. Предности

Велика предност Објектно-Орјентисаног програмирања је што помаже у стварању одрживог кода који је лако размљиви 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 врста објеката који међусобно комуницирају и ажурирају своја стања.

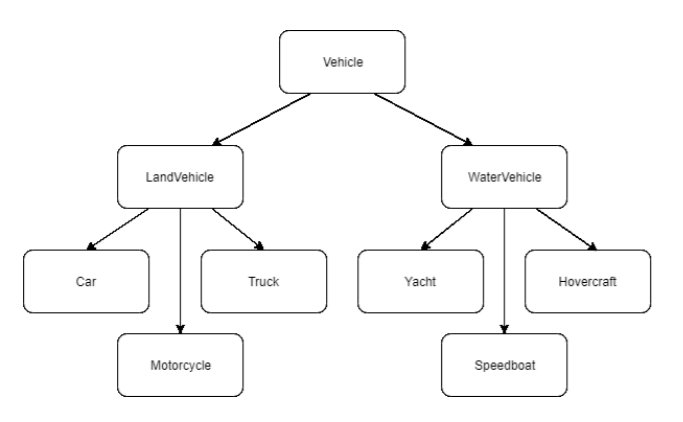
* 1. Мане

Посотоје два озбиљна недостатка у објектно-орјентисаном дизајну који могу створити проблеме играма, а то су одржавање и преформансе.

* + 1. Одржавање

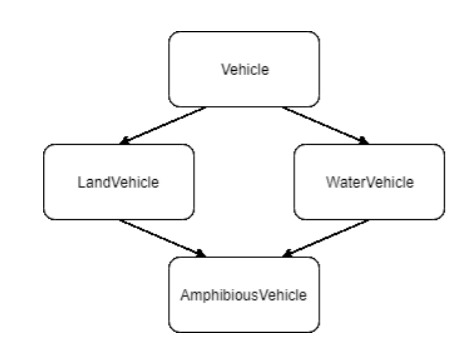
Проблем који се појављује код овог дизајна је проблем класификације објеката. Постоји могућност класификација објеката на основу само једног скупа критеријума. На пријмјер, уколико се организам класификује према генетским особинама, боја организма није узета у обзир на било који начин. Да би се организам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

У стварно животу честе 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 у када је хијерархија класа била оргинално прављена. [2]



Слика 2.4.1.1 Хијерархија класа аутомобила

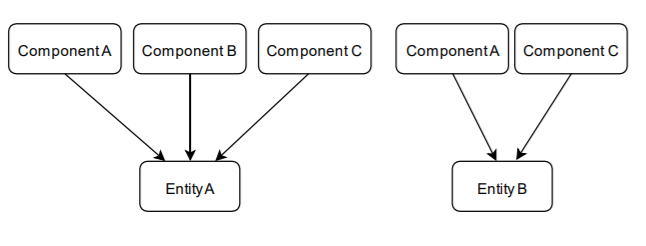
На слици 2.4.1.1 је приказана хијерархиија објеката аутомобила у објектно-орјентисаном дизајну. Иако се оваква хијерахија чини прихватљивом, може да дође до проблема уколико дизајнер видео игре одлучи да дода још једну врсту возила, која могу да се крећу и по копну и по земљи. Таква класа би морала да 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 наслеђвањем 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 оснвоне 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 нам су потребни нови типови класа, а потпуно преуређивање хијерархиија представља неприхватљиво рјешење.

Рјешење у оваком случају је да се одступи од објектно-орјентисаног 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-орјентисана композиција.



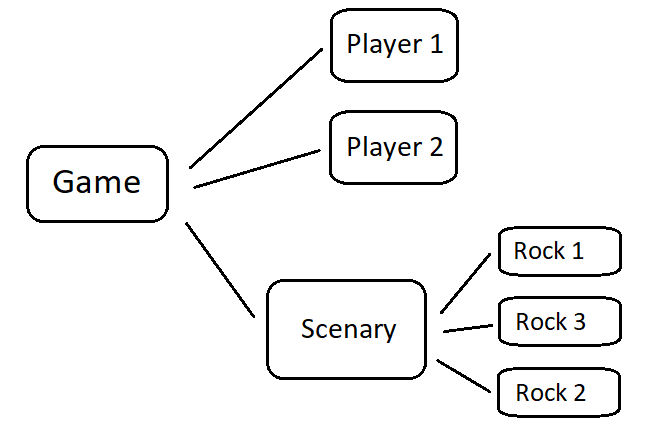
Слика 2.4.1.3 Хијерархија објектно-орјентисане композиције

На слици 2.4.1.3 приказано је како су ентитети састављени од малих компоненти пружајући ентитету оређену 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. Преформансе

Када су у питању преформансе битно је да се спомене чињеница да брзина процесроа се повећава знатно брже него брзина 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-орјентисани 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 добрим приступом меморији названа *Data-oriented technology stack* DOTS. Главна идеја оваквог приступа је уредити податке у меморији како би се максимизирала локалност података и направити код који користи велике блокове података одједном, умјесто да се ради над једним објектном у неком временском тренутку.

1. Архитектура орјентисана на податке

Имплементација коришћењем дизајна орјентисаног на податке може се сматрати неком врстом проширења објектно-орјентисаног 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 искористион на најбљи начин, потребно је предузети главни развојни помак, користити дизајн орјентисан на податке. DOTS је потпуно орјентисан на податке: код мора да буде дизајниран око података, а не обрнуто, што представља одступање од предходних система који су били објектно-орјентисани. Када се правилно користи, DOTS 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 орјентисаног на податке у софтверима великих размјера је био 1998 године у видео игри по називом „Thief: The Dark Project“, у којој су програмери имали филозофију стварања изузето употребљивих компоненти. Посвећивање овом експериметалном дизајну нагарђено 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 звршну 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 орјентисан 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 Адамз 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 ту тему ријетко могу наћи, тренутно постоји безброј 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 преформанси, флексибилности и продуктивности.

ECS се састоји од три основна елемента која међусобно комуницирају:

* Ентитети: које је Адам Мартин деинисао 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.

Напиши још нешто овдје...

* 1. Поређење са објектно-орјентисаним дизајном

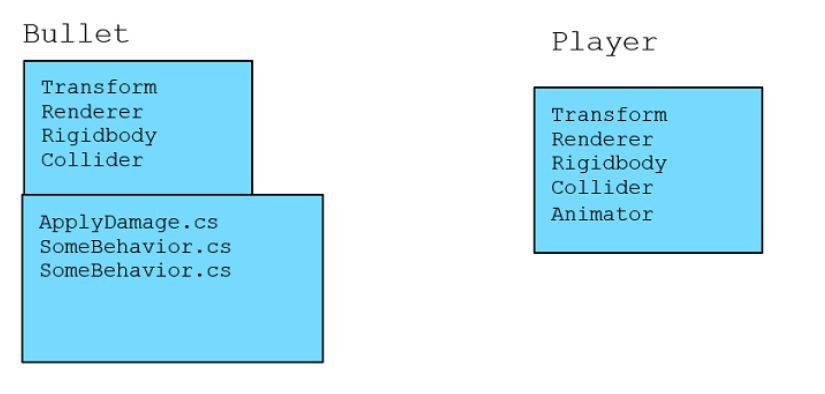
Главна разлика ове двије архитектуре је у томе што се OOP фокусира на, како му и само име каже, објекте. Цијели програм се пише с објектима на уму, те се сви остали концепти прилагођавају објектима. Објект можемо дефинисати као скуп података и функционалности који манипулишу тим подацима на једном мјесту.

Како би боље разумјели OOP и упоредили га са архитектуром орјентисаном 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 времену извршавања
  + 1. Добјиање 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, SIMD векторска једница.

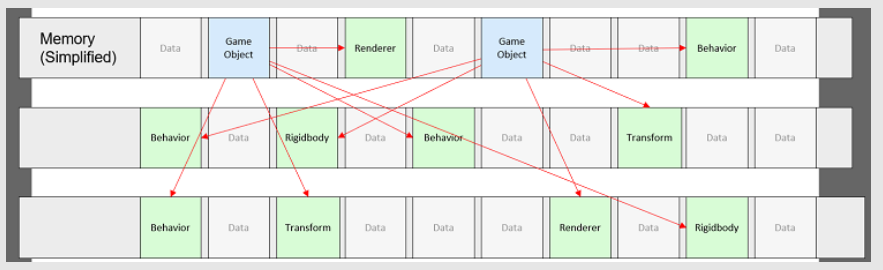


Слика 3.3.1.1 Примјер више *GameObject*-а и компоненти у ООP дизајну *Unity*

Приступ подацима из системске меморије је много спорији него припступ кеш меморији. То је меморија гдје до изражаја долази дохватања података унаприје. 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 преформанси 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 временз извршавања.

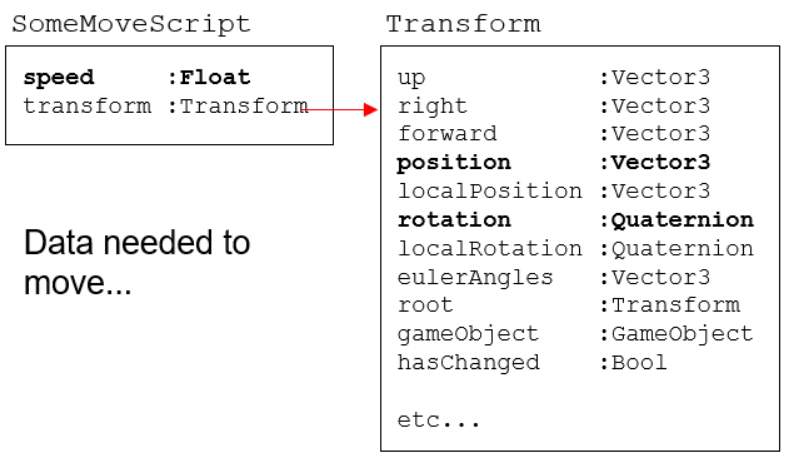
Баш такав примјер имамо у OOP дизајну при развоју видео игара, као што је приказано на слици 3.3.1.2. Слика приказује случајну спорадичну природу ове методе складиштења података. Сваки *GameObject* садржи одређен броји компоненти, које се алоцирају у *heap* меморији, тако да имамо разбаацане објекте са великим бројем референци. Приступ компонентама је врло чест, а често је потребно и да приступимо више компоненти 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 корсити, па је овакав дизајн критичан према преформансама.



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.

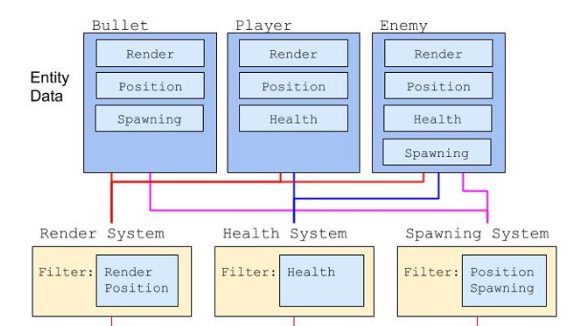
Како би се извршило помјерање *GameObject* коришћењем неке написане скрипте потребно је да приступимо позицији и ротацији, који су чланови компоненте *Transform*. Када хардвер дохвата податке из меморије, кеш меморија се попуњава и са свим осталим члановима ове компоненте који представљају бескорисне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



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

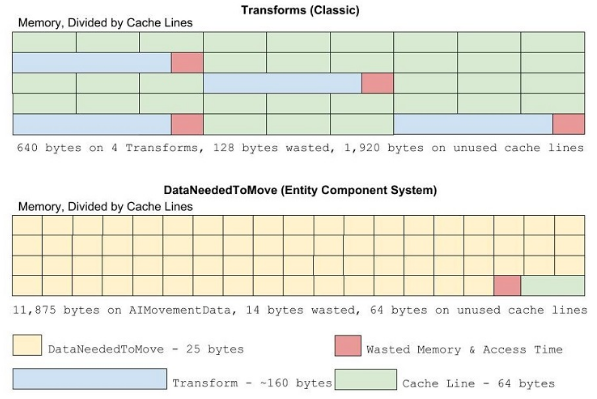
Entity Component System архитектура помаже у уклањању неефикасног референцирања објеката. Уместо *GameObject-*аса сопственом колекцијом компоненти, размотримо ентитет који садржи само податке који су му потребни.

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

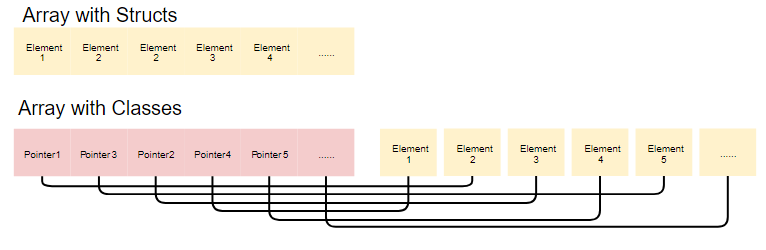
Наравно, није само систем за помјерање онај који има предност од оваког приступа. Још једна уобичајена компонента у многим играма су здравствени системи постављени у спектру непријатеља и савезника. Ови системи обично имају мало или нимало разлика међу врстама објеката, па су још један сјајан кандидат 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.



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.

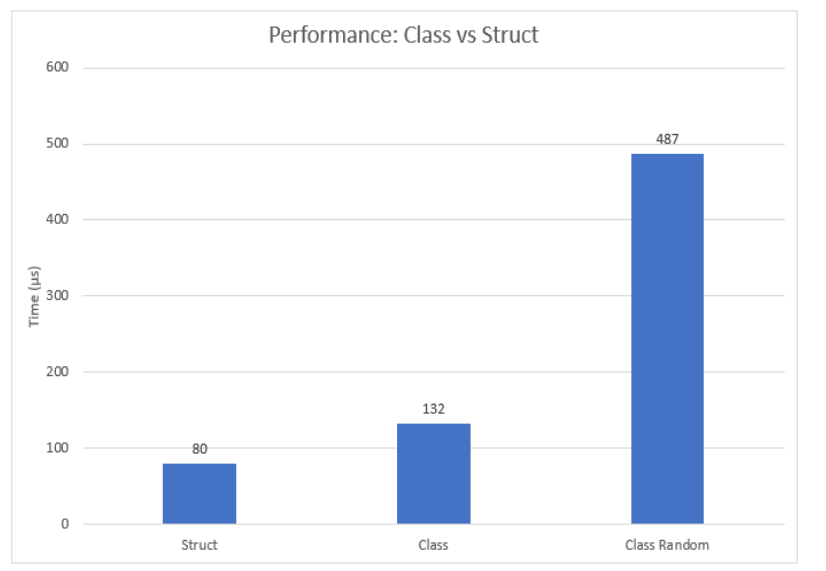
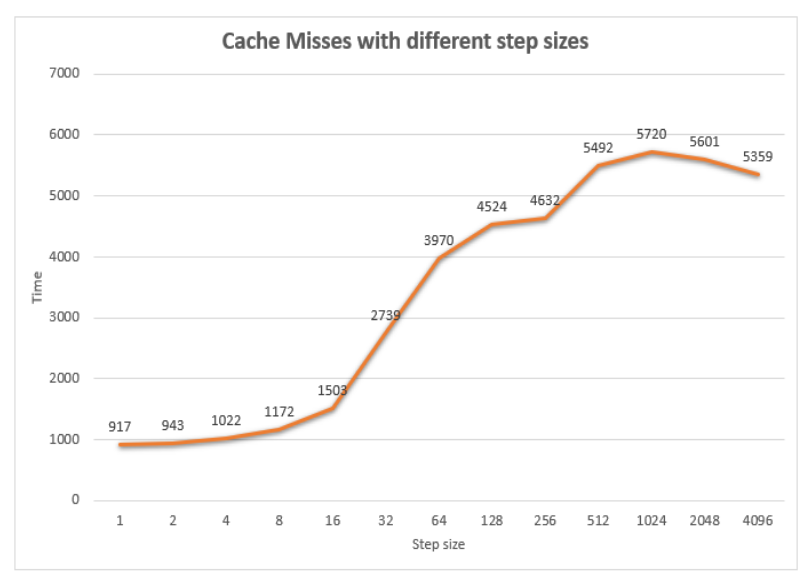
Примјер оваквог распоред у меморији приказан је на слици 3.3.1.6.



Слика 3.3.1.6 Распоред низоба и класа у меморији С#

* + 1. Поређење преформанси

На наредним сликама приказана је разлика у преформансама када се користи низ класа и низ структура. Тестирани низ садржи 1 милион елеменета што даје величину низа од 4MB [6].

Слика 3.3.2.1 Поређење преформанси C#

1. Детаљи имплементације

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 кода и дијаграма који помажу читаоцима да разумију основне концепте и архитектуру.

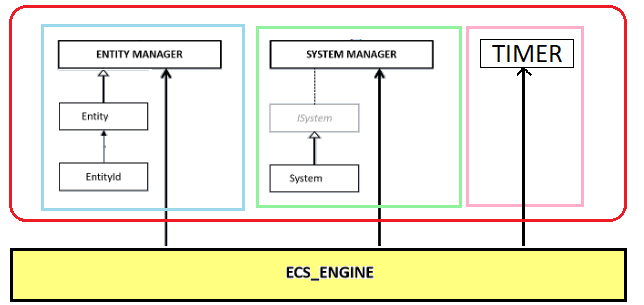
* 1. Entity-Component-System

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 ускладиште 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 моје 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 подпоглављима

На слици 4.1.1 приказана су четри 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 (розо 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

Аааа.

* + 1. Управљање системима

Аааа.

* 1. ECS примјери

Aaaa.

Табела 4.2.1 Образац за изглед табеле

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **величина 1** | **величина 2** | **величина 3** | **величина 4** | **величина 5** | **величина 6** | **величина 7** | **величина 8** |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

1. Додатне могућности

Свакако, осим имплементације саме Entity-Component-Sytem архитектуре, која увелико доприности преформансама комплексних видео игара, како би *DOTS* био потпун могуће је имплементирати још низ додатних могућности.

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 који CPU треба да обави. *Multithreading* је врста програмирања која користи способност CPU да истовремено обрађује више нити у више језгара. Уместо 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 побрино да пружи програмерима погодну платформу за писање сигурног кода који се извршава у више нити, што додатно доприноси преформансама. Посао или *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 пре него што се покрену.

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

Дизајн орјентисан 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. Порблеми

Ааа

1. Закључак

Преласком са објектно-орјентисаног дизајна на дизајн орјентисан на подакте, програмеру биће много лакше да поново користи свој код, а други да га разумију и раде на њему.

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).](https://gamedevelopment.tutsplus.com/tutorials/quick-tip-intro-to-object-oriented-programming-for-game-development--gamedev-1805" \l ":~: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.)

Списак скраћеница

|  |  |
| --- | --- |
| ECS | *Entity Component System* |
| OOP | *Object-Oriented Programming* |
| DOTS | *Data-Oriented Technology Stack* |
| VFX | *Visual effects* |
| SIMD | *Single instruction, multiple data* |
| CPU | *Central Processing Unit* |

Списак слика

[Слика 2.4.1.1 Хијерархија класа аутомобила 3](#_Toc50588616)

[Слика 2.4.1.2. Дијамант структура класа 4](#_Toc50588617)

[Слика 2.4.1.3 Хијерархија објектно-орјентисане композиције 4](#_Toc50588618)

[Слика 2.4.2.1 Објекти у ООP повезани великим бројем референци 5](#_Toc50588619)

[Слика 3.3.1.1 Примјер више *GameObject*-а и компоненти у ООP дизајну *Unity* 8](#_Toc50588620)

[Слика 3.3.1.2 Референце у меморије између објекат, њиховог понашања и њихових компоненти 9](#_Toc50588621)

[Слика 3.3.1.3 Чланови који се стварно користе за операцију кретања и изгубљени простор 9](#_Toc50588622)

[Слика 3.3.1.4 Еntity-Component-System примјер 10](#_Toc50588623)

[Слика 3.3.1.5 Фрагментација у кешу и неискоришћени простор при ОО дизајну 10](#_Toc50588624)

[Слика 3.3.1.6 Распоред низоба и класа у меморији С# 11](#_Toc50588625)

[Слика 3.3.2.1 Поређење преформанси C# 11](#_Toc50588626)

[Слика 4.1.1 Aрхитектура ECS система 13](#_Toc50588627)

Списак табела

[Табела 3.3.1. Образац за изглед табеле 13](#_Toc50328609)