Primjena ECS arhitekture u izradi računalnih igara

Adamek, Jurica

VISOKO UČILIŠTE ALGEBRA

DIPLOMSKI RAD

Primjena ECS arhitekture u izradi računalnih igara

Jurica Adamek

Zagreb, mjesec 2018.

Predgovor

Zahvaljujem svojem mentoru dr. sc. Goran Đambić, v. pred. koji mi je pomogao pri definiranju ideje, savjetovao kada je bilo potrebno te imao puno strpljenja za ovaj rad.

Zahvaljujem također svojim kolegama koji su bili dobro i zanimljivo društvo tijekom ove dvije godine.

Na kraju zahvaljujem svojim roditeljima, bratu i djevojci koji su mi bili podrška tijekom diplomskog studija.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme diplomskog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ideja rada je stvoriti okolinu za analizu te testiranje performansi različitih programskih arhitektura. Arhitekture koje će se analizirati su OOP te ECS. Stvorit će se tri iste igre na različitim platformama te spomenutim arhitekturama u Unity razvojnom okruženju. Igre će se analizirati na Android te Windows platformi. Postoji niz parametara po kojima će se igre uspoređivati te će testovi biti adekvatno vizualizirani.

Ključne riječi: Unity, OOP, ECS, analiza, igra.

Summary

Idea of this thesis is to create an environment for analysis and testing of the performance of different software architectures. Architectures that will be analyzed are OOP and ECS. For purpose of testing, the same game will be made for different platforms and said architectures on Unity framework. Games will be analyzed on Android and Windows platforms. A set of testing factors will be determined and the results of them will be properly visualized.

Key Words: Unity, OOP, ECS, analysis, game.

Sadržaj

1. Uvod

Kako se programiranje razvija kao znanost i struka, tako se osmišljavaju različiti pristupi problemima. Kroz njihovo rješavanje primjećuje se kako su određena rješenja bolja u određenim okolnostima naspram drugih. Sve to rezultira osmišljavanju različitih paradigmi, arhitektura te obrazaca. Od reaktivnog programiranja, objektnog ili funkcionalnog, svako malo se izrodi novi način rješavanja određenih problema.

Tema ovog diplomskog rada je analiza i testiranje performansi ECS (engl. *Entity Component System*, skraćeno ECS) arhitekture u odnosu na tradicionalne OOP (engl. *Object oriented programming*, skraćeno, OOP) sustave. Na temelju krajnjeg rezultata testova, prikazat će se kako navedene arhitekture pristupaju problemima te koja je za ovaj specifičan problem prikladnija. Ideja i cilj je stvoriti platformu za testiranje tih sustava koja će na objektivan način dati rezultate performansi te pokazati koji način bi se u kojem trenutku trebao koristiti. Kroz ovaj rad, implementirat će se igra na tri različita načina pomoću različitih arhitektura te testirati na dvije platforme: Windows te Android operacijskim sustavima.

Rad će biti odvojen po logičkim cjelinama kroz koje će se postepeno kretati od uvoda u problematiku preko testiranja prema zaključku. Počet će se od uvoda i teorijske obrade arhitektura te implementacijskim detaljima ECS arhitektura koje će se koristiti u radu. Nakon toga se implementira vlastiti ECS sustav sa znanjem iz prethodnih cjelina te igra na tom sustavu. Da bi se mogao sustav uspoređivati s drugim ECS sustavima, igra će biti napisana u Unity ECS okviru. Na kraju će igra biti napisana po tradicionalnom OOP načinu pisanja. Nakon implementacije, slijedi testiranje igara s paralelnom analizom dobivenih rezultata. Igre će biti testirane na dvije platforme u različitim uvjetima.

Očekuje se da će se Unityjeva ECS implementacija pokazati kao najbolje rješenje s visokim performansama s obzirom da je razina optimizacije na toj arhitekturi puno veća nego ona koju je moguće postići kroz ostale načine programiranja. Igra u vlastitom ECS okviru te OOP rješenje trebali bi imati slične performanse, no ECS bi trebao imati nešto bolje rezultate.

2. Analiza OOP i ECS arhitekture

Arhitektura je struktura kôda, stvorena s ciljem da bude što prilagodljivija svojoj okolini i drugim sustavima s kojima mora zajedno funkcionirati. Tijekom godina, pokazalo se da je arhitektura programa vrlo kompleksna tema kojoj se može pristupiti na više načina. Problem arhitekture javlja se u svim programerskim jezicima, implementacijama i projektima. Ovisno o prirodi problema, različite arhitekture mogu dominirati nad ostalima, zbog svojih specifičnosti koje možda u drugim slučajevima ne bi bile primjenjive. Kod viših jezika kao što su C#, Java ili C++, OOP se nameće kao standardna paradigma pisanja kôda, dok se kod jezika, kao što su Go ili C, preferira pisanje proceduralnog kôda. Moderni jezici mogu podržavati više paradigmi što povećava fleksibilnost tog jezika da se koristi u različitim problemima i situacijama.

Arhitekturu možemo gledati kao implementaciju paradigme. Više implementacija iste paradigme može se razlikovati, ali još uvijek teže prema istom cilju, odnosno paradigmi. Tako, C# jezik podržava niz programerskih paradigmi uključujući OOP, funkcionalno programiranje, DDP (engl. *Data Driven Programming*, skraćeno DDP) i druge. Kod inicijalne verzije C# jezika dodavala se podrška za rad funkcionalnog programa, reaktivnog, programa s više dretava i sl. Konstantnim razvijanjem C# jezika povećava se broj mogućih paradigmi koje su programeru na raspolaganju.

Također bi se trebala napraviti distinkcija između arhitektura te uzorka dizajna kôda. Neki uzorci kao *Factory* ili *Singleton* mogu se koristiti da bi se neka arhitektura uspješno implementirala, ali oni sami ne predstavljaju arhitekturu.

U ovom radu će se objasniti i analizirati dvije arhitekture u dvije različite paradigme; OOP arhitektura te ECS koji predstavlja DDP paradigmu. Obje implementacije su napisane u C# jeziku, bit će analizirane i uspoređene preko više prevoditelja. Svaka implementacija igre bit će prikazana s istom funkcionalnosti i dizajnom, ali će u pozadini imati različite implementacije arhitektura. Igre su specifične vrste aplikacije zbog toga što se od njih konstantno očekuje visoka razina performansa. Jedna igra bi trebala imati prosječan broj sličica u sekundi od 30 FPS-a (engl. *Frames Per Second*, skraćeno FPS) te će se preko implementacija vidjeti koliko se dobro arhitektura drži pod ovim zahtjevom.

2.1. Uvod u OOP

Objektno orijentirano programiranje je paradigma proizašla iz iskustva programerske zajednice koja je tijekom godina identificirala programerske probleme koji se konstantno pojavljuju i definirala potencijalna rješenja za njih. OOP možemo nazvati skup praksi kako pristupiti programiranju tako da napisani kôd bude što robusniji, da se može ponovno iskorištavati te jednostavno nadograđivati. OOP pokušava približiti kôd samom programeru, tako da on lakše prepozna što se u programu događa, ali time stvara kôd koji je teži za interpretaciju i izvođenje samom računalu. Akademska zajednica se usuglasila da se OOP temelji na 3 glavna koncepta iz kojega sve ostalo proizlazi, a to su: enkapsulacija, nasljeđivanje te polimorfizam.[1] Osim tih koncepata zajednica je prepoznala niz principa kojima se programer može služiti da bi kôd ostao čitljiv i robustan kao SOLID, no takvi principi se mogu primijeniti na razne paradigme, ne samo OOP.

Neke od arhitekture koje primjenjuju OOP su: MVC, MVP, MVVM i ostali.

2.1.1. Enkapsulacija

OOP kaže da bi svaki segment aplikacije trebao u sebi sadržavati svoju logiku, a na van pokazivati drugim segmentima samo svoju suštinu. Ponekad su to samo *getteri* za privatne varijable, a nekada su javne metode u kojima se izvodi neka poslovna logika. Enkapsulacija je upravo to.

Enkapsulacija objašnjava da su određeni segmenti nevidljivi vanjskim korisnicima te su im dani drugačiji načini pristupa. Na primjer, klasa ne bi trebala imati javne varijable, jer joj bilo koji korisnik tog segmenta može pristupiti. Nego bi u većini slučajeva trebalo napraviti *gettere* i *settere* umjesto direktnog pristupa varijablama. Koncept enkapsulacije olakšava samom programeru rad jer točno zna s čime može raditi. Negativna strana enkapsulacije je teže shvaćanje performansi od strane programera te duže vrijeme izvođenja na procesoru. Primjer u C# jeziku pokazuje izvršavanje 200.000 puta dohvat varijable putem *gettera* te 200.000 puta dohvat druge varijable koja je `public`.

```
static void Main(string[] args)
{
      Test test = new Test();
      int local;
      var watchA = Stopwatch.StartNew();
      for (int i = 0; i < 200000; i++) local = test.A;
```

```
            watchA.Stop();
            Console.WriteLine(watchA.ElapsedTicks);
            var watchB = Stopwatch.StartNew();
            for (int i = 0; i < 200000; i++) local = test.b;
            watchB.Stop();
            Console.WriteLine(watchB.ElapsedTicks);
      }
      class Test
      {
            public int A { get; set; }
            public int b;
      }
```

Kôd 2.1 Program za testiranje performansi *gettera* i `public` varijable

Kôd 2.1 pokazuje rezultate u procesorskim otkucajima. Bilo koji drugi način bilježenja bio bi prevelik da bi se vidjela razlika u brzinama. Rezultati pokazuju da su oba načina vrlo brza u izvršavanju te ne troše puno procesorskog vremena. Usprkos tome, kako vidimo u Tablici 2.1, *getter* je u prosjeku 1.5 puta sporiji od čistog pristupanja varijabli.

Tablica 2.1: Mjerenje performansi dohvata *gettera* i `public` varijable u procesorskim ciklusima

	1.	2.	3.	4.	5.	6.	7.	8.	9.	Prosjek
Getter	2038	1918	1919	1919	1969	1923	2135	2751	2006	2064
Public	1226	1225	1314	1318	1269	1314	1672	1472	1397	1356

Razlika u rezultatima se možda ne čini toliko velika, ali u stvarnom primjeru, logika može biti puno kompleksnija s više pristupa varijabli te se pomoću `public` varijable može uštedjeti mnogo CPU (engl. *Central Processing Unit*, skraćeno CPU) vremena. Također se može činiti da prosjek od nekoliko tisuća otkucaja nije velik za današnje standarde gdje su norma procesori s 2 do 3 GHz takta, ali kada se ovakav način dohvaćanja varijabli preseže kroz cijeli kôd igre ili koje druge aplikacije koja mora konstantno biti performantna, osjeti se razlika.

2.1.2. Nasljeđivanje

Koncept nasljeđivanja omogućuje programeru ponovno iskorištavanje kôda kreiranjem generalne klase te jedne ili više specijaliziranih klasa. Isto tako se približava kôd programeru stvarajući hijerarhiju koja je lakša za praćenje. Primjer životinje kao bazne

klase i psa, kao specijalizirana klasa, opisuje temeljno nasljeđivanje. Nasljeđivanje samo po sebi nema velike cijene što se tiče performansi, ali određene funkcionalnosti koje se dobivaju korištenjem toga mogu biti zahtjevnije za procesor. Tehnike poput virtualnih funkcija, kao što je prikazano u kôdu 2.2 nešto su zahtjevnije jer u pozadini koriste virtualne tablice (V-Tablice)[2]. V-Tablica se koristi da program zna koju metodu pozvati kada je u pitanju lanac od jedne ili više virtualnih metoda. Pri pozivu svake metode, na takvom objektu koji mijenja virtualnu metodu svoje bazne klase, program mora proći po lancu od bazne klase sve do odgovarajuće metode.

```
class Zivotinja
{
      public virtual void glasajSe(){}
      public void drugoGlasanje(){}
}
class Pas : Zivotinja
{
      public override void glasajSe(){}
}
```

Kôd 2.2 Primjer nasljeđivanja

Kod prevođenja kôda, današnji moderni prevoditelji kao *Roslyn* ili *Mono*[3] dodatno optimiziraju kôd tako da virtualne metode i ostale ekstenzije koje donosi nasljeđivanje imaju vrlo malo odstupanje u performansama od klasičnih metoda, no kao i kod enkapsulacije, postoje razlike.

Tablica 2.2: Mjerenje performansi između virtualne i obične metode u procesorskim ciklusima

	1.	2.	3.	4.	5.	6.	Prosjek
Virtualna metoda	7049	5719	7452	6402	6408	6624	6609
Obična metoda	6071	5083	4946	5123	5390	4959	5262

Test je napravljen između virtualne i obične metode koje se pozivaju 200.000 puta te rezultat vizualiziran u tablici. Tablica 2.1 pokazuje da je virtualna metoda oko 20% sporija. Isto kao i kod prethodnog testiranja enkapsulacije i ovaj test pokazuje vrlo mala

odstupanja, no kada bi se cijeli program pisao ovom logikom, moguće je da korisnik primijeti pad performansi aplikacije.

2.1.3. Polimorfizam

Polimorfizam se definira kao mogućnost mijenjanja stanja tijekom vremena izvođenja programa. To se dobiva korištenjem nasljeđivanja i sučelja. Sučeljem se definira skup ponašanja koja se trebaju implementirati u klasi, a objekt tog tipa sučelja može se mijenjati kroz život programa. Polimorfizam također ima veze s prethodnim konceptom nasljeđivanja. Ako postoji bazna klasa, varijabla tog tipa može mijenjati svoju vrijednost na bilo koji objekt specijalizirane klase. Kôd 2.3 prikazuje primjer gdje imamo sučelje `IGlasanje` te dvije klase koje implementiraju to sučelje, `Pjevač` te `Govornik`. Varijabla tipa `IGlasanje` u bilo kojem trenutku može zamijeniti svoju implementaciju te kôd nastavlja normalnim radom.

```
public interface IGlasanje { void glasajSe();}
public class Pjevac {...}: IGlasanje
public class Govornik {...}: IGlasanje

public void main()
{
	IGlasanje covjek = new Pjevac();
	covjek.glasajSe();

	covjek = new Govornik();
	covjek.glasajSe();
}
```

Kôd 2.3 Primjer polimorfizma

2.2. Uvod u ECS

Kao što je bilo napisano u Uvodu, ECS nije paradigma, već je implementacija jedne, odnosno, to je arhitektura. Pošto je to arhitektura, u ECS-u se bavimo implementacijskim detaljima te pojmovima. ECS je dio DDP paradigme, ono se bavi time da su podaci glavni fokus arhitekture, u odnosu na OOP gdje su u fokusu sami objekti.[4] To znači da se cijeli program piše s podacima u vidu te se trebaju definirati prije nego što se kreću rješavati problemi. U ECS-u podaci se predstavljaju s entitetima (E u ECS nazivu) te

komponentama (C u ECS nazivu). Kada se definiraju podaci, kreće se pisati logika koja manipulira tim podacima, što je ovdje predstavljeno sa sustavima (S u ECS nazivu).

Entitete možemo gledati kao „vreću“ za komponente. Oni samo služe da se grupira niz komponenti zajedno. Ovisno o implementaciji, to mogu biti liste ili neka druga struktura podataka. Entitet može sadržavati još neke podatke, ali zbog performansi i same suštine ECS-a, entitet se ne bi trebao predstavljati kao ništa drugo osim kao držač za komponente.

Komponente su sami podaci. U ECS arhitekturi sva stanja koja se mijenjaju trebala bi biti u komponentama. Kako je prikazano u kôdu 2.4 i 2.5, postoje dvije vrste komponenata: podatkovna i označna. U podatkovnoj se nalaze stanja programa, dok označna služi samo da označi entitet kojem je pridijeljena. Na primjer, ako entitet ima komponentu `Player` koja nema nikakva stanja na sebi, taj entitet je označen tom komponentom. Označne komponente služe za filtriranje entiteta koje ćemo proći kod opisa koncepta sustava.

```
public struct Bullet : IComponentData
{
	public float speed;
	public float timeToDestroy;
}
```

Kôd 2.4 Primjer podatkovne komponente

```
public struct Player : IComponentData{}
```

Kôd 2.5 Primjer označne komponente

Sustavi su sama logika programa. Kada se sustav aktivira, djeluje u 2 koraka: filtriranje i obavljanje logike. U prvom se koraku pretražuju entiteti u bazi svih entiteta tražeći one s određenim komponentama. Ako se u sustavu traže entiteti s komponentama `Player` i `Move`, kao povratna vrijednost će biti svi entiteti koji imaju te komponente na sebi. U nekim implementacijama postoje mogućnosti da se dohvate svi entiteti koji nemaju određenu komponentu ili entiteti koji imaju samo određene komponente i ni jedne druge. U drugom koraku sustav izvodi neke akcije, odnosno logiku nad tim entitetima i njihovim komponentama. Kod ECS arhitekture, čest je slučaj da svaka komponenta ima svoj sustav koji je obrađuje. To omogućava jednostavnu kombinaciju komponenata tako da se entitet ne mora brinuti koje komponente mora imati na sebi da se određena logika provede. U većini slučajeva, sustav neće filtrirati entitete po jednoj komponenti već po više njih. To je generalno jedna ili više baznih komponenti (pozicija, rotacija, prikazivač (engl. *renderer*)) te jedna komponenta na koju se odnosi sustav (npr. sustav kontroliranja igrača ima pristup

komponenti `Input`). Tako sustav ima pristup nekim generalnim podacima entiteta kao pozicija te prema tome može određivati kakva će se logika izvršavati.

Jedna od stvari zašto se ECS koristi u praksi je brzina izvođenja.[5] Tok kôda je vrlo jednostavan za izvođenje na procesoru te, ovisno o implementaciji, sustavi mogu biti brži nego klase s implementiranom logikom u nekoj drugoj arhitekturi. ECS isto omogućuje mijenjanje logike nad podacima. U OOP rješenjima, kada neka klasa implementira neko sučelje, ne može u trenutku izvođenja kôda promijeniti to sučelje. U ECS-u se to lagano može implementirati pomoću dodavanja i micanja komponenti na entitetu. Isto tako se mogu kombinirati komponente da istovremeno izvode nekakvu logiku.

Na primjer, imamo entitet `Sword` koji može imati `Fire` i/ili `Magic` svojstvo te u svakom trenutku može dodati ili maknuti neko od tih svojstva. U OOP rješenju mogli bi koristiti neki od obrazaca poput *Decorator* ili *Builder*, kao što je prikazano u kôdu 2.6, ali ni s njima ne dobivamo sve što donosi ECS. Problem s takvim pristupom je taj što trebamo predefinirati takve funkcionalnosti u kôdu, dok u ECS-u samo dodamo ili maknemo komponente s entiteta, kao što je vidljivo iz kôda 2.7. Ovdje možemo vidjeti još jedno svojstvo u ECS-u, a to je da se komponente mogu ponovno iskorištavati na različitim entitetima. U bilo kojem trenutku komponentu `Fire`, s entiteta `Sword`, možemo dodati i nekom drugom entitetu, recimo `Arrow`. Sustav koji obrađuje entitete s komponentom `Fire` filtrirat će sve s tom komponentom i odraditi nad njima ono što je u sustavu definirano. Recimo, kada `Sword` zadaje udarac proizvesti će vatrenu eksploziju, a isto to će napraviti i `Arrow` kada udari u svoj cilj.

```
public class Sword
{
      public Sword addFire(){...}
      public Sword addMagic(){...}
      public Sword removeFire(){...}
      public Sword removeMagic(){...}
}

Sword sword = new Sword().addFire().addMagic();
sword.removeMagic();
```

Kôd 2.6 Moguća implementacija u OOP arhitekturi

```
public class Magic : Component{}
public class Fire : Component{}
```

```
Entity sword = new Entity();
addComponent(sword, new Magic());
addComponent(sword, new Fire());
removeComponent(sword, typeof(Fire));
```

Kôd 2.7 Moguća implementacija u ECS arhitekturi

U radu će se promatrati dvije implementacije ECS sustava, jedna od strane Unityja, dok će druga biti vlastita.

2.3. Usporedba OOP-a i ECS-a

Kao što je bilo prije napisano, OOP je paradigma pa ćemo za potrebe ovog poglavlja predočiti arhitekturu koja koristi prethodno navedena tri glavna koncepta OOP-a.

Glavna razlika ove dvije arhitekture je u tome što se OOP fokusira na, kako mu i samo ime kaže, objekte. Cijeli program se piše s objektima na umu te se svi ostali koncepti prilagođavaju objektima. Objekt možemo definirati kao skup podataka i funkcionalnosti koji manipuliraju tim podacima na jednom mjestu.[6] Implementacija spomenute definicije može se vidjeti u kôdu 2.8. Ako se mora odvijati neka akcija nad objektom, ideja je da je ona enkapsulirana u njemu te da je objekt može izvršavati sam, bez drugih sudionika. Ako je potreban koji drugi sudionik, on se dobavlja iz različitih izvora, bilo kao argument, neka tvornica (engl. *Factory*) ili nekako drugačije. Na tom drugom objektu zovu se metode pa u nekom trenutku taj objekt može zvati sljedeći objekt i slično. Ovdje vidimo taj lanac objekata kojima se gradi sam program.

Budući da se ECS orijentira na same podatke više nego na OOP, možemo ga svrstati u DDP paradigmu. Kod takvih arhitektura, fokus je na samim podacima koji su potrebni za izvođenje programa, a sve ostalo se bazira na tim podacima. ECS objekt je vidljiv u kôdu 2.9. Koncept objekta se ne bi trebao koristiti u ECS-u, već bi podaci i logika trebali biti odvojeni. Podrazumijeva se da se `Objekt` kao instanca klase ili strukture mora stvoriti jer je to ograničenje jezika, ali u njemu bi trebali biti samo podaci i stanje koje je njemu potrebno. Ostala logika se definira u drugim dijelovima kôda, što su u kontekstu ECS-a sustavi.

```
public class OOPObject
{
	private int valueA;
	public int getValueA()
	{
		return valueA;
	}
	public void addToA(int value){
		this.valueA += value;
	}
}
```

Kôd 2.8 Primjer OOP objekta

```
public class ECSObject
{
	public int valueA;
	public int valueB;
}
```

Kôd 2.9 Primjer ECS objekta

OOP i ECS dva su različita pristupa različitim problemima. Ukoliko se prikaže potreba performansama, brzom pristupu i mijenjanju objekata (instanca klasi i struktura), bolja opcija je ECS. Ako problem koji programer mora riješiti uključuje nadograđivanje programa, kombinaciju logika te ponovno iskorištavanje određenog djela logike, bolji pristup je OOP.

Ove dvije arhitekture nisu međusobno isključive. Postoji opcija da se dio kôda napiše u OOP arhitekturi, dok se dio koji mora biti performantniji napiše u DDP arhitekturi. Postoje sustavi koji su već sazrjeli u određenoj paradigmi te uvođenje nove može značiti nanovo pisanje komponenti koje već provjereno rade. Ako se nešto mora mijenjati po pitanju arhitektura, jedno od mogućih rješenja su hibridne implementacije koje spajaju najbolje od dva arhitekturna svijeta.

3. Unity API

Razvojno okruženje Unity omogućuje razvoj igara na više načina. Budući da se kôd u Unityju piše u C# jeziku, moguće je raditi po raznim paradigmama. Od Unity verzije 2018, Unity je razvio prevoditelja koji nativno podržava ECS. To znači da se ECS kôd koji se napiše u C# jeziku dodatno optimizira bude performantniji.

Do te verzije, jedini način pisanja kôda je bio s `MonoBehaviour` komponentama, koji teži prema OOP načinu pisanja. `MonoBehaviour` komponentu možemo gledati kao OOP objekt. Ona ima neka svoja stanja koja se mogu podešavati iz Unity urednika (engl. *Unity Editor*) te ima logiku koja je enkapsulirana unutar same komponente.

U vremenu pisanja ovog rada, Unity ECS sustav je još u *preview* statusu razvoja. To znači da je još u procesu izrade te nije za korištenje u produkciji. Zbog toga je Unity omogućio programerima da koriste takozvani *Hybrid ECS*[7] s klasičnim korištenjem `MonoBehaviour` komponenata. Također, niz sustava još nije prebačeno na ECS sustav, uključujući fiziku, sustav čestica, sustav stvaranja korisničkog sučelja, odnosno UI-a (engl. *User Interface*, skraćeno UI) i drugi. Zbog tog problema, moraju se pronaći drugačija rješenja kad se radi s takvim sustavima u ECS kôdu.

3.1. Standardni način rada u Unity okruženju

U standardnom načinu izrade igara, svi objekti koje igrač vidi dio su jedne ili više trenutno učitanih scena. Scene možemo gledati kao spremnik svih objekata te definira sve ostale dodatne parametre potrebne za normalan rad scene kao što je svjetlost, pečene teksture i sl.

U svakoj sceni postoji niz `GameObject` objekata. `GameObject` sam po sebi nema nikakvog utjecaja na scenu, ali se pomoću dodatnih komponenata definiraju njegova statička i dinamička svojstva. Kako je scena spremnik `GameObject` objekata, tako `GameObject` možemo gledati kao spremnik komponenata. `GameObject` može biti jedno drvo koje treba imati `MeshRenderer` i `MeshFilter` komponentu za iscrtavanje na ekranu ili može biti upravljač igrača koji ima neke korisničke definirane komponente.

Komponente su najmanji dio u Unity arhitekturi i one definiraju samo ponašanje i izgled igre. Unity dolazi s velikim brojem predefiniranih komponenata, kao što je `MeshRenderer`, `AudioSource` ili `AudioListener`, a moguće je raditi i svoje komponente. Komponenta se kreira tako da se stvori klasa koja nasljeđuje `MonoBehaviour` klasu. Svaka komponenta ima svoj životni ciklus te se može pretplatiti na niz događaja koje je Unity pripremio[8]:

- `Awake()` - Prva metoda koja se poziva kod pokretanja skripte. U ovoj metodi se najčešće inicijaliziraju članovi klase.
- `Start()` - Druga metoda koja se poziva kod pokretanja skripte. Ova metoda služi tome da se pristupi članovima drugih klasa prije početka igre.
- `Update()` - Beskonačna petlja koja poziva svaku sličicu (engl. *frame*). Koristi se najčešće za manipuliranje ulaznih signala (pritisak gumba, pokret miša).
- `FixedUpdate()` - Beskonačna petlja koja se poziva nekoliko puta u jednoj sličici. Koristi se najčešće za implementaciju fizike u igri.
- `OnGUI()` - Također beskonačna petlja koja služi za prikaz grafičkog sučelja. Samo u ovoj metodi se mogu pozivati metode klase `GUI` koja služi za kreiranje `GUI` objekta.
- `OnDestroy()` - Metoda koja se poziva kada je pozvana `Destroy()` metoda na objektu ili kada se izlazi iz scene.
- Ostale metode - postoji još desetak metoda koje služe za preciznije upravljanje skriptama.

Da bi se neka komponenta skriptirala, Unity daje na raspolaganje API (engl. *Application Programming Interface*, skraćeno API) u `MonoBehaviour` klasi, `GameObject` klasi i drugima. Na primjer, da se pristupi nekom objektu na sceni po imenu, potrebno ga je naći pomoću `GameObject.Find(string name)` metode.

Sve ove klase daju potporu za rad na OOP način. Jedna komponenta je suzdržljiva cjelina koja se pretplaćuje na određene događaje u svom životnom ciklusu, ima svoja stanja, javne metode te komunicira s drugim objektima preko njihovih javnih metoda.

Unity u pozadini radi niz dodatnih provjera pri pozivanju svake od metoda životnog ciklusa pa ovakvo pisanje kôda može za posljedicu imati lošije performanse. U igri koja je kreirana za potrebe ovog rada, na jednoj sceni istovremeno postoje više tisuća objekata te svaki ima potrebu zvati `Update()` na jednoj od svojih komponenata. Jedna moguća

opcija smanjenja broja `Update()` funkcija u kôdu je ta da postoji jedan glavni `Update()` koji u sebi onda zove druge vlastite `doUpdate()` funkcije. Pristup s vlastitim funkcijama je zasigurno brži jer se zaobilaze dodaci koje Unity postavlja, ali je također i manje siguran. Kada se koristi `Update()` funkcija, Unity prvo provjerava sljedeće stvari[9]:

1. Iterira po svim `Update()` funkcijama koristeći `SafeIterator` klasom koja osigurava, ukoliko se promjeni lista usred jedne iteracije, da ništa ne pukne u igri.
2. Pregledava je li sam poziv funkcije uredu. Ovdje se gleda postoji li `gameObject` na kojem je pozvana metoda još uvijek ili ne.
3. Unity se sprema pozvati funkciju. Kreira se objekt koji sadrži sve argumente za funkciju.
4. Poziva se funkcija.

U sceni s 11.000 objekata koji su imali logiku u beskonačnoj petlji, igra je imala rezultate prikazane u tablici 3.1:

Tablica 3.1 Komparacija `Update` i `doUpdate` u milisekundama (ms)

	1.	2.	3.	4.	5.	6.	7.	8.	9.	Prosjek
`doUpdate`	20.08	17.58	17.71	17.50	17.29	18.20	17.88	18.18	17.07	161.49
`Update`	17.96	18.20	19.30	18.10	26.23	24.97	35.34	16.93	18.80	195.83

Kao što vidimo, `Update` je sporiji za otprilike 20% od vlastite `doUpdate` funkcije.

3.2. ECS način rada u Unity okruženju

Kao što je prethodno bilo napisano, za ECS su potrebna tri koncepta: entiteti, komponente i sustavi. Pošto se cijela logika odrađuje u sustavima, nema potrebe za standardnim `MonoBehaviour` komponentama, ali to znači da se na neki drugi način moramo pretplatiti na događaje unutar igre kao što su `Start()`, `Update()` i slično. Da bi kompenzirali ovakve situacije, Unity je pripremio određen broj već definiranih sustava i dodatnih funkcionalnosti da je moguće raditi ECS projekt bez `MonoBehaviour` komponenti.

Svaka igra mora imati neki kôd za inicijalizaciju. Bilo to učitavanje spremljenih podataka o igraču ili učitavanje modela iz resursa. U ECS kôdu to se izvodi tako da se jednoj ili više funkcija doda anotacija (engl. *annotation*) tipa `RuntimeInitializeOnLoadMethod`. Ta anotacija prima enumeraciju tipa `RuntimeInitializeLoadType` kao parametar koji može imati 2 stanja:

- `BeforeSceneLoad`: Izvršava se prije nego što se scena učitava, to je zamjena za `Awake()`.
- `AfterSceneLoad`: Izvršava se nakon što je scena učitana, zamjena za `Start()` funkciju.

Anotacija se postavlja na statičnu funkciju te se ne mora nigdje dodatno pozvati, Unity ECS sistem će ju sam pozvati u pravo vrijeme. Trenutna je praksa da u kôdu cijele igre postoji samo jedna klasa s takvim anotiranim funkcijama, te se iz njih sve ostalo izvodi. Primjer takve funkcije je prikazan u kôdu 3.1:

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterSceneLoad)]
public static void init(){
}
```

Kôd 3.1 Primjer funkcije s `AfterSceneLoad` parametrom

U ovoj implementaciji ECS-a, entitet se definira strukturom `Entity` koja predstavlja indeks u memoriji. Zbog nekih dodatnih funkcionalnosti koristi se struktura, ali ovo je mogao biti obični `int` i obavljao bi isti posao kao i trenutna implementacija. `Entity` možemo gledati kao vrlo lagani `GameObject`. On postoji na sceni te na sebi ima neke komponente. Razlika je u tome što `GameObject` na sebi ima još niz stanja kao što su njegovo ime, njegova oznaka (engl. *tag*) i druge varijable, dok `Entity` ne predstavlja ništa više nego samo indeks u memoriji. `GameObject` također u sebi sadržava metode koje služe za rad s njim i drugim `GameObject` objektima kao što je `GetComponent` ili `AddComponent`. U ECS-u, sva logika se nalazi u sustavima te se u `Entity` strukturi ne nalaze nikakve funkcije vezane uz igru.

Entiteti žive u konceptu svijeta (engl. *world*). Svijet možemo poistovjetiti s scenama u Unityju. Oni nam omogućuju da ih dijelimo i određenu logiku primjenjujemo na određenu skupinu. Svaki svijet ima svog upravitelja, odnosno instancu `EntityManagera`. Sve operacije kreiranja entiteta, dodavanja ili micanja komponenti ili rada na podacima prolaze kroz njega. Neke funkcionalnosti `EntityManagera` su prikazane u kôdu 3.2:

```
public Entity CreateEntity(EntityArchetype archetype);
public void DestroyEntity(Entity entity);
public T GetComponentData<T>(Entity entity);
public void RemoveComponent<T>(Entity entity);
```

Kôd 3.2 Dio `EntityManager` API-a

Kada se kreira entitet s određenim komponentama, praksa je da se stvori arhetip entiteta. Taj koncept je implementiran u strukturi `EntityArchetype`. Arhetip predstavlja jedan entitet kojemu su sve njegove komponente usko kreirane u memoriji. Ako stvaramo entitet s komponentama `Position`, `Scale` i `Rotation`, instanciranje arhetipa bi izgledalo kao u kôdu 3.3:

```
manager.CreateArchetype(new ComponentType[]
{
      typeof(Position),
      typeof(Rotation),
      typeof(Scale)});
}
```

Kôd 3.3 kreiranje arhetipa

U memoriji to možemo zamisliti ovako:

	Position	Rotation	Scale
1	10 15 23	0 0 0 0	10 10 10
2	42 42 22	0 0 0 0	1 1 1

Slika 3.1 Prikaz podataka u memoriji

Na slici 3.1 prikazana su dva entiteta, jedan s indeksom 1, drugi s indeksom 2. Svaki put kada se doda novi entitet, dodat će se na novo mjesto, a podaci u komponentama bit će pomaknuti za onoliko da budu na istom „stupcu“ u memoriji kao i ostale komponente. Ovo omogućuje brzo pretraživanje i iteraciju po entitetima istog arhetipa. Ukoliko se obriše neka komponenta ili doda nova, entitet prestaje biti dio ovog arhetipa, te se gubi beneficije brzog pristupa podacima.[10]

Komponente se definiraju tako da se u kôdu stvori nova struktura te se implementira sučelje `IComponentData` ili `ISharedComponentData`. Sučelja nemaju nikakve metode, te služe više kao oznaka da se radi o komponenti. Razlika između ova dva sučelja

je u tome da `IComponentData` komponenta u svojim varijablama može imati samo *blittable* tipove podataka[11], dok `ISharedComponentData` podržava sve tipove, uključujući i `MonoBehaviour` komponente i ostale referentne tipove.[12] Zbog svojeg ograničenja na samo *blittable* tipove podataka, u teoriji bi `IComponentData` trebala biti brža u radu jer se ne mora raditi nikakva konverzija podataka iz *managed* memorije u *unmanaged*[1]. Vlastiti testovi su pokazali da oba tipa komponenti imaju slične performanse kako je prikazano u tablici 3.2. Ovo može biti zbog toga što je ECS sustav još u razvoju i nije kompletno optimiziran.

Tablica 3.2 Komparacija brzine dohvata i mijenjanja podataka `IComponentData` i `ISharedComponentData`

	1.	2.	3.	4.	5.	6.	Prosjek
`IComponentData`	0.24	0.24	0.26	0.24	0.23	0.24	0.241
`ISharedComponentData`	0.31	0.23	0.22	0.22	0.24	0.23	0.242

U vremenu pisanja ovog rada, ECS nije podržan od niza sustava. Da bi se koristio na primjer *Cinemachine* sustav, potrebno je stvoriti komponentu koja nasljeđuje `ISharedComponentData` i raditi s virtualnim kamerama *Cinemachinea* unutar te komponente. Budući da se u takvim slučajevima radi o referentnim tipovima koji žive u *managed* memoriji, bilo kakvu promjenu ne treba slati natrag u memoriju preko `EntityManagera` već je promjena trenutačna.

Sustavi su zadnji sastavni dio ECS-a. Unity podržava niz sustava, ali ovdje ćemo se osvrnuti na jedan tip sustava, `ComponentSystem`. Svaki sustav se odvija u dva koraka: prikupljanje entiteta i rad nad njima. `ComponentSystem` je apstraktna klasa te se pri kreiranju sustava mora implementirati funkcija `OnUpdate()` koja izvršava svaku sličicu dok je sustav omogućen. Entiteti se dohvaćaju preko objekta tipa `ComponentGroup` što djeluje kao filter; u njega se unesu sve komponente koje se žele dohvatiti ili komponente koje moraju biti isključene tijekom pretrage. Da bi se dobilo pravo stanje entiteta i njihovih komponenti, praksa je da se entiteti dohvaćaju na početku svakog poziva `OnUpdate()` funkcije. Spomenuti životni ciklus ECS sustava je prikazana u kôdu 3.4:

[1] Managed memorija je ona memorija koju je potrebno ručno oslobađati, dok je unmanaged memorija o kojoj se brine GC(engl. *Garbage Collector*, skraćeno GC).

```
public class EnemySystem : ComponentSystem{
	ComponentGroup group;
	protected override void OnUpdate()
	{
		if(group == null)
		{
		group = GetComponentGroup(Bootstraper.enemyTypes);
		}
		var enemyEntities = group.GetEntityArray();
		...}
```

Kôd 3.4 Primjer sustava

3.3. Opis rada Burst prevoditelja

U standardnom načinu pisanja kôda u Unity okruženju, prevođenje kôda može se odvijati na 2 načina:

- Preko Mono prevoditelja
- Preko IL2CPP prevoditelja

Burst prevoditelj donosi novu alternativu prevođenju koja generira vrlo optimiziran kôd. Novi prevoditelj u pozadini koristi dijelove LLVM kôda.[13] LLVM je već postojeći lanac tehnologija koji se koriste za izradu i rad novih ili poboljšanje starih prevoditelja. LLVM u sebi sadrži biblioteke za optimiziranje, prevođenje, otklanjanje pogrešaka i druge. Burst je generalno nastao za potrebe *Unity Jobs* sustava koji je, ako se pravilno koristi, sposoban stvoriti vrlo efikasan i performantan kôd. Burst stvara kôd koji je siguran od konkurentnosti (engl. *thread safe*) te omogućuje jednostavno korištenje više dretava. Burst također može generirati performantan kôd, iako se ne koristi *Jobs* sustav, već neki drugi, standardniji načini pisanja kôda. Burst nudi ugrađene funkcije (engl. *intrinsic functions*) za `System.Math` biblioteku, pokazivače te nativna polja. Unity također nudi svoju `Unity.mathematics` biblioteku koja je samo omotač oko `System.Math` da bi programerima bio olakšan rad s matematičkim funkcijama. Unityjeva biblioteka za matematiku također donosi niz struktura koje se mogu optimizirati s Burst prevoditeljem kao što su `float2`, `float3` ili `quaternion`. Korištenjem takvih biblioteka i Burst prevoditelja može znatno poboljšati performanse nego s `UnityEngine.Mathf` te standardnim prevoditeljima. Prevoditelj ne može raditi s objektima koji žive u *managed* prostoru memorije, odnosno s referentnim tipovima. On isključivo radi s primitivnim

tipovima objekata te s još par struktura. Ova barijera je postavljena zato da u svakom trenutku Burst ima cjelovit pristup memoriji.

Također postoje izjave u kôdu koje se mogu optimizirati od strane prevoditelja, a postoje one koje ostaju neoptimizirane:[13]

Izjave koje se optimiziraju su:

- Izjave za kontrolu toka programa (`if` / `else` / `while`/…)
- Ekstenzijske metode
- Nesiguran C# kôd koji uključuje pokazivače
- Referentne parametre
- Čitanje `static readonly` varijabli

Izjave koje se ne optimiziraju:

- `DLLImport` naredba
- `Try / catch / finally` naredba
- `Foreach` naredba
- Pozivi metoda koji uključuju tipove koji žive u *managed* prostoru

Kod ECS kôda, Burst optimizira čitanje i pisanje podataka u komponentama zbog načina kako su posložene u memoriji. Burst može generirati kôd koji umjesto jedne dretve, može koristiti više dretava da bi pristupio podacima.

Da bi se Burst prevoditelj koristio u Unity igri, potrebno je preuzeti njegov paket s upravitelja paketima u Unity uredniku.

4. Izrada igara

Za potrebe komparacije ovih različitih arhitektura, ista igra bit će implementirana na različite načine. Igra o kojoj se radi je 3D igra iz trećeg lica. U igri igrač kontrolira svemirski brod koji mora zaustaviti neprijatelje dok oni putuju do matičnog broda u cilju da ga unište. Brod se na Windows platformi kontrolira s tipkama W (naprijed), A (lijevo), D (desno) te razmaknica za ubrzanje. Da brod napadne, potrebno je pritisnuti i držati lijevi gumb miša. Na Android platformi postoji virtualna kontrolna palica za kontroliranje broda. Što više neprijatelja brod uništi, to će imati jače oružje. Igrač će dobiti novo oružje kada uništi 1%, 2% te 6% neprijatelja na sceni. Oružja koja dobiva su:

- Na početku do 1% uništenih neprijatelja: sporo pucanje projektila
- Od 1% do 2% uništenih neprijatelja: brzo pucanje projektila
- Od 2% do 6% uništenih neprijatelja: zraka koja uništava neprijatelje kada ih dotakne
- Od 6% na dalje: zraka koja se povećava da prekrije sve neprijatelje

Kontrole će se replicirati na svim verzijama igre, no postoji mogućnost da će, zbog različitosti u implementacijama, biti drugačiji osjećaj u kontroliranju broda.

Najveći pokazatelj različitosti u performansama bit će prosječni broj FPS-a te broj istodobnih neprijatelja na sceni. Ostali pokazatelji uključuju vrijeme pokretanja, minimalni FPS kroz igru te maksimalni broj FPS-a tijekom cijele igre. Također će se mjeriti rad procesora tijekom igre. Taj broj će, uz prosječni broj FPS-a, pokazati koliko je arhitektura zapravo efikasna u korištenju resursa.

Zbog nekih ograničenja koje Unity ECS ima, morat će se posegnuti za alternativnim rješenjima u sustavima kao što je fizika ili sustav kamera. Ova ograničenja su privremena, pošto je ECS sustav još u razvoju te se očekuje da će se u sljedećim verzijama otkloniti.

Za izradu igre u vlastitom ECS sustavu, prvobitno će se stvoriti ECS sustav inspiriran Unity ECS-om te nekim popularnijim sustavima kao Entitas[14] i LeoECS[15]. Sustav bi trebao biti potpuno odvojen od bilo kakvog Unity kôda, te bi se u teoriji mogao koristiti i u drugim C# aplikacijama, ne samo u kontekstu Unity okruženja.

OOP igra će se graditi na tri glavna koncepta u OOP-u koja su bila prethodno objašnjena. U ovoj ćemo implementaciji vidjeti puno veću razinu semantike pošto će gotovo svaki koncept biti popraćen jednom klasom, ako ne i sučeljem, dok se prethodne implementacije bave samim podacima.

4.1. Izrada vlastitog ECS sustava

Kod implementacije vlastitog ECS sustava važno je prvobitno implementirati osnovne koncepte poput entiteta, komponente, sustava i upravitelja entiteta. Za ovu implementaciju, odlučeno je da će ECS sustav biti tanak sloj između Unity kôda i kôda za igru te će se sve ostalo odvijati u sustavima. Implementirane su osnovne funkcionalnosti koje se mogu naći u Unity ECS-u, te se koristi sličan tok kôda kao i u njemu:

1. Dohvat entiteta
2. Dohvat komponenata
3. Rad na podacima u komponentama

Razina optimizacije neće biti dostižna kao na Unity ECS-u zbog toga što se neće pisati dodatni prevoditelj kao što se koristi u Unity ECS-u. Ovdje će se kôd optimizirati korištenjem pravilnih struktura podataka te pisanjem standardnog ECS kôda. Koncepti kao entitet te komponenta bit će klase, dok će sustavi biti prestravljeni sučeljem. Implementacija spomenutih koncepata prikazana je u kôdu 4.1:

```
public class Entity{}
public class Component{}
public interface ISystem
{
      void update();
}
```

Kôd 4.1 Implementacija entiteta, komponente i sustava

Kao što vidimo, svi ovi tipovi podataka su referentni i minimalistički. Oni više služe semantički, da bi se odvojilo značenje zbog lakšeg čitanja kôda. Novi entitet je samo novi objekt koji ima svoju memorijsku lokaciju te se preko nje traže njegove komponente i slično.

Upravitelj entiteta `EntityManager` je srce ove implementacije. Preko njega se stvaraju entiteti, dodaju i uklanjaju komponente, radi filtriranje entiteta i sve ostalo što je povezano

s ECS sustavom. `EntityManager` je statična klasa kojoj za rad treba, na početku igre, pozvati `init()` statičnu funkciju, te svaku sličicu `update()` statičnu funkciju. Sve ostale funkcije koje su na raspolaganju sustavima kada je potreban rad s entitetima i komponentama su prikazane u kôdu 4.2:

```
createNewEntity(params Component[] components)
destroyEntity(Entity entity)
addComponent(Entity entity, Component component)
removeComponent(Entity entity, Component component)
getComponents(Entity entity)
getComponent<T>(HashSet<Component> components)
getFirstComponent<T>(Component[] filter)
getEntities(params Component[] filter)
registerSystem(ISystem system)
cancelSystem(ISystem system)
```

Kôd 4.2 Cijeli API `EntityManager` klase

Povezanost između entiteta i njegovih komponenti je u `EntityManager` skripti implementirano preko dva rječnika:

- `static Dictionary<Entity, HashSet<Component>> entitiesData;`
- `static Dictionary<Entity, int> entitiesCache;`

u rječniku `entitiesData` se nalaze sami podaci o komponentama svakog entiteta, a u `entitiesCache` se pohranjuju podaci koje se sve komponente nalaze na kojem entitetu. Ovo povećava kompleksnost kôda, ali se za uzvrat dobiva na performansama. Bez pričuvnog rječnika, maksimalni broj entiteta na Windows platformi je bio 5.000, a s dodatnim optimiziranjem je taj broj narastao na 15.000. Problem je bio u pretraživanju `HashSet<Component>` djela `entitiesData` rječnika. Makar je to operacija kompleksnosti O(1), u pozadini se dobavljaju sažeci (engl. *hash*) podataka te je to procesorski zahtjevna operacija. S optimizacijom, u pozadini svaki tip komponente dobije jedinstveni identifikacijski broj koji je potencija broja 2. Kada entitet dobije na sebe određenu komponentu, u `entitiesCache`, pod njegovim ključem, doda se taj identifikator. Isto tako se njegov broj smanji kada se komponenta uklanja. Ovo služi tome da je operacija pregleda svih komponenti koje entitet ima na sebi vrlo brza operacija, pošto se izvodi logika nad samim bitovima. Rad s bitovima u funkciji `getEntities` je prikazan u kôdu 4.3.

Recimo, ako entitet na sebi ima komponente `ComponentMoveForward` koja ima identifikator 4 te komponentu `ComponentTransform` s identifikatorom 32, njegov broj će u rječniku biti 36. U binarnom je to zapisano 100100 te prva jedinica s desna označava da ima `Transform` komponentu, a druga da ima `MoveForward`. Jedini način da ovaj sustav radi je taj da svaka komponenta ima drugačiji identifikator, te se zbog toga ne daje korisniku sustava da dodjeljuje brojeve, već se to izvodi tijekom prvog korištenja komponente. S trenutnom implementacijom, moguće je imati 32 različite komponente, gdje svaka komponenta predstavlja jedan bit u `int` djelu `entitiesCache` rječnika. Ovo može biti problem u većim igrama gdje je potrebno više komponenti, no u trenutnoj igri to je zanemarivo. Kao potencijalno rješenje tom problemu može biti korištenje neke druge strukture kao `long` ili neka druga vlastita implementacija.

```
public static List<Entity> getEntities(params Component[] filter)
{
	...
	Int filterCode = getFilterCodeFor(filter);
	foreach (var pair in entitiesCache)
	{
		if((pair.Value & filterCode) == filterCode)
		{
			output.Add(pair.Key);
		}
	}
	...
}
```

Kôd 4.3 Suština `getEntities` funkcije u `EntityManager` klasi

Komponente se definiraju tako da se kreira klasa u kojoj se naslijedi klasa `Component` kako je prikazano u kôdu 4.4. `Component` je više kao oznaka da se radi o komponenti pa ne zahtijeva nikakvu dodatnu implementaciju. U komponenti se ubacuju sva stanja koja su relevantna za tu komponentu. U ovoj implementaciji ne postoji razlika između komponenti koje rade s *blittable* tipovima te ostalim, već se svi tipovi podataka mogu staviti u

`Component`. Pošto se radi o klasama, bilo kakva promjena na varijablama je trenutačna te se ne mora vraćati promjena natrag u memoriju kao kod Unity ECS-a.

```
public class ComponentMoveForward : Component
{
	public float speed;
}
```

Kôd 4.4 Primjer implementacije komponente

Sustavi rade više - manje kao i na Unity ECS platformi. Oni traže entitete s određenim komponentama te imaju neku logiku koja manipulira tim entitetima. Za rad sustava se treba implementirati sučelje `ISystem` te metodu `update()`. Primjer implementacije sučelja `ISystem` se može vidjeti u kôdu 4.5. U ovoj implementaciji, potrebno je na početku igre, u nekom inicijalizacijskom kôdu, pretplatiti sve sustave koji se pokreću. To se registrira preko prethodno navedene `EntityManager` skripte. Na njoj postoji funkcija `registerSystem` te se kao parametar predaje objekt tipa `ISystem`. Nakon toga ECS sustav sam u pozadini zove registrirani sustav. Također je moguće maknuti sustav da se više ne izvodi preko funkcije `cancelSystem`. Ove dvije funkcije omogućuju to da se sustavi mijenjaju tijekom igre te da se igra dinamički optimizira.

```
public class SystemMoveForward : ISystem
{
	Component[] filter = new Component[]
	{
		new ComponentMoveForward(),
		new ComponentTransform()
	};
	public void update()
	{
		var entities = EntityManager.getEntities(filter);
		for (int i = 0; i < entities.Count; i++){
			var components =
				EntityManager.getComponents(entities[i]);
			var moveForward =
				EntityManager.getComponent<ComponentMoveFor
			ward>(components);
			...
		}
	}
}
```

Kôd 4.5 Primjer implementacije sustava

Kao što vidimo, filter se izvodi preko novih polja objekata tipa `Component[]`, a u pozadini se pretražuje samo tip komponente koji se traži, a ne i referenca. To svojstvo sustava postoji zbog prethodno napisanih identifikacijskih brojeva komponenata. U sustavu se prvo dohvate svi entiteti s tim određenim filterom, nakon toga se dohvaćaju sve komponente na zasebnom entitetu te se na kraju filtriraju dobivene komponente da se dobe samo one koje su potrebne za logiku. Ovakav dohvat komponenata se forsira jedino zbog optimizacije rada igre. Da bi se pri svakom dohvatu jedne komponente nanovo dohvaćale sve komponente, performanse bi bile puno niže nego s trenutnom implementacijom.

4.2. Izrada igre na vlastitom ECS sustavu

Igra na ovoj arhitekturi se bazira na jednoj `MonoBehaviour` skripti te nekolicini sustava. `Game` je `MonoBehaviour` skripta koja se u ECS žargonu zove *Bootstrap* skripta te ona inicijalizira sve objekte, početne entitete te sustave. Također se preko nje zove `Update()` funkcija `EntityManager` klase.

U igri se koristi 13 komponenata te se popis svih nalazi u jednoj C# datoteci zvanoj `ComponentTypes`. Većina komponenta sadrži neke podatke kao npr. `ComponentMoveForward` s podatkom `speed` kao što vidimo u kôdu 4.6:

```
public class ComponentMoveForward : Component
{
      public float speed;
}
```

Kôd 4.6 Komponenta `ComponentMoveForward`

Postoji nekolicina komponenta koje služe samo kao oznake, odnosno entitetima se pridodaje ta komponenta samo da se entitet obradi u nekom sustavu. Sve definirane označne komponente su prikazane u kôdu 4.7:

```
public class ComponentDestroy : Component{}
public class ComponentEnemy : Component{}
public class ComponentTargetShip : Component{}
```

Kôd 4.7 Sve korištene označne komponente

Sustavi su podijeljeni tako da svaki radi jednu stvar, držeći se SRP (engl. *Single Responsebility Principle*, skraćeno SRP) principa. SRP predstavlja način pisanja kôda tako

da svaka cjelina radi samo jednu stvar.[16] Veličina te cjeline ovisi o okolini, arhitekturi, jeziku i ostalim faktorima. Ona može biti jedna klasa, jedna funkcija ili nešto drugo. Ovdje se primjenjuje SRP princip na ECS sustav kao cjelinu. Sustavi koji se koriste su sljedeći:

- `SystemMoveForward`: Ako se entitet obradi u ovom sustavu, pomaknut će se u svijetu za onoliko kolika mu je brzina definirana u `ComponentMoveForward`.
- `SystemEnemySetup`: Sustav koji gleda samo entitete s komponentom `ComponentEnemySetup`. Postavlja neprijatelje na određena mjesta te nakon obrade uklanja tu komponentu.
- `SystemEnemy`: Promjena inicijalne brzine neprijateljskog broda nakon što stigne na svoju početnu poziciju.
- `SystemInputMove`: Pomicanje entiteta u odnosu na kontrole.
- `SystemFireToInput`: Pucanje projektila ili svjetlosne zrake kada se pritisne određeni gumb na mišu.
- `SystemBullet`: Sustav odgovoran za provjeravanje kolizije između projektila i neprijatelja. U pozadini postoji dodatna `MonoBehavoiur` skripta `CollisionRule` koja prima događaje `OnCollisionEnter` i `OnTriggerEnter` te šalje informacije o koliziji sustavu.
- `SystemUI`: Sustav koji osvježava UI odnosno korisničko sučelje.
- `SystemDestroy`: Ukoliko je bilo kojem entitetu dodijeljena komponenta `ComponentDestroy`, ovaj sustav će ga procesirati i uništiti. Uništenje se događa u istoj sličici u kojoj je komponenta dodijeljena, samim tim što se `SystemDestroy` zove zadnji.

Svaki sustav ima istu strukturu i način odvijanja. U članskim varijablama se definiraju svi filtri koji se koriste za dohvat entiteta s određenim komponentama. `update()` funkcija u sebi prvobitno dohvaća sve entitete nad kojima se izvodi logika i, ako je potrebno, dohvaća komponente tih entiteta. Komponente se dohvaćaju tako da se prvo dohvati lista svih komponenata jednog entiteta te nakon toga traži pojedina komponenta. Ovime omogućuje bolje performanse jer se samo jednom dohvaćaju sve komponente. Definirani životni ciklus sustava može se vidjeti u kôdu 4.8:

```
public class SystemUI : ISystem
{
      Component[] filter = new Component[]
      {
```

```
        new ComponentUI()
};
public void update()
{
        var entities = getEntities(filter);
        for (int j = 0; j < entities.Count; j++)
        {
                var components = getComponents(entities[j]);
                var ui = getComponent<ComponentUI>(components);
                ...
        }
}
```

Kôd 4.8 Primjer sustava

Većinski je cijela igra napisana u ECS duhu, no kolizija je iznimka. Svaki objekt koji treba procesirati koliziju treba na sebi imati bilo koju `Collider` Unity komponentu te skriptu `CollisionRule`. Ta skripta služi kao brojač projektilima koliko su neprijateljskih brodova uništili, a uništava i same objekte brodova na sceni. U sustavu `SystemBullet`, kôd čita brojač koji se nalazi u `CollicionRule` skripti te povećava globalni broj uništenih brodova.

4.3. Izrada igre na Unity ECS sustavu

Kao i na prošloj ECS implementaciji i ova ima *Bootstrap* skriptu, odnosno skriptu koja inicijalizira sve objekte za ECS. S obzirom na to da se radi o čistom ECS sustavu, ta skripta je naziva `Bootstraper`. U njoj se anotira jedna funkcija koja se naziva `init()`, s anotacijom `RuntimeInitializeOnLoadMethod` te obaveznim parametrom vrijednosti `RuntimeInitializeLoadType.AfterSceneLoad`. U toj funkciji se definiraju svi arhetipovi koji se koriste u igri. Također se u `Bootstraper` skripti inicijaliziraju prvobitni entiteti za igrača, neprijatelje te ostale objekte. Da bi se entitet prikazivao na ekranu u Unity ECS sustavu, potrebno je na njega postaviti određene komponente. Jedna do njih je `MeshInstanceRenderer`. Tu komponentu možemo gledati kao zamjenu za `MeshRenderer` i `MeshFilter` u standardnom načinu rada Unityja. U samoj sceni definiraju se standardni `GameObject` objekti s `MeshInstanceRendererComponent`. To je `MonoBehaviour` skripta, te se na njoj definira model kojeg je potrebno prikazivati. Nakon toga se u `Bootstraper` skripti

dohvaćaju svi objekti s tom skriptom te preko `Value` svojstva dolazi se do `MeshInstanceRenderer` ECS komponente.

Ako se treba prikazati brod, u sceni se definira `GameObject` objekt s `MeshInstanceRendererComponent` objektom te određenim modelom kao parametrom. Nakon toga u skripti stvaramo entitet s `MeshInstanceRenderer` komponentom te vrijednosti modela koje se dohvaća iz scene.

Sve korisnički definirane komponente nalaze se u skripti `ComponentTypes`. Unutra se nalaze 2 vrste komponenta, `IComponentData` te `ISharedComponentData`. U ovoj implementaciji se ne koristi ni jedna komponenta označne naravi. S ukupno 7 komponenti definiraju se sva stanja koja su potrebna za rad igre. Ovdje je manje komponenti nego u prošloj implementaciji, a to je zbog toga što su se ovdje spajale neke komponente radi performansi.

Kao i s prošlom implementacijom, većina logike je u sustavima. Ovdje su neki sustavi spojeni u jedan, zbog nekih ograničenja u ECS sustavu koja bi mogla dovesti do velikih padova u performansama. Usprkos tome, postoje sustavi koji se drže SRP principa.

Sustavi koji se koriste u igri su:

- `BulletSystem`: Sustav koji je odgovoran za pomicanje projektila te uništavanje projektila ukoliko je živ na sceni više od 2 sekunde.
- `EnemySystem`: Kompleksniji sustav nego prošli, bavi se prvobitnim brzim pomicanjem neprijatelja do svog odredišta te naknadnim pomicanjem prema baznom brodu. Osim toga, gleda je li brod blizu bilo kojeg projektila ili ispred igrača dok igrač ispaljuje zraku koja ga uništava. U toj situaciji sustav uništava brod. Logika neprijatelja te projektila je spojena u ovaj sustav zbog toga što bi, da su odvojeni, svaki projektil trebao iterirati kroz polje svih neprijatelja, što ima velike posljedice na performanse. Ovako svaki neprijatelj iterira kroz sve projektile što je puno manji trošak. Prvobitno je ovaj problem nastao zbog toga što trenutno ne postoji sustav fizike u ECS-u pa se to aproksimira pomoću udaljenosti u prostoru između neprijatelja i projektila.
- `PlayerSystem`: Sustav pomoću kojeg se igrač pomiče, ispaljuje projektile te zrake.
- `SpawnerSystem`: Sustav koji postavlja entitete neprijatelja na nasumičnu poziciju.

- `UISystem`: Preko ovog sustava se korisničko sučelje osvježava novim postotkom uništenih neprijatelja potrebnih za novo oružje.

4.4. Izrada igre u OOP stilu

Cilj u ovoj implementaciji bio je imati što manje `MonoBehaviour` skripta na sceni. Kao i u ECS implementacijama, postoji jedna skripta koja inicijalizira objekte potrebne za rad igre. `Game` je skripta koja koristi `Start()` metodu Unityja da inicijalizira igrača, korisničko sučelje i neprijatelje te `Update()` metodu da zove njihovu logiku svaku sličicu. U istoj se skripti čuvaju globalni podaci poput broja koliko je neprijatelja uništeno, koje je trenutno oružje na igraču i slično.

Igrač je u igri definiran sučeljem `IPlayer`. U njemu je definiran popis metoda koje klasa mora implementirati za rad s igračem poput pomicanja ili mijenjanja oružja. Trenutno je definirana samo jedna implementacija u `Singleplayer` klasi. `Singleplayer` u sebi kao članske varijable sadržava objekte tipa `IInput` te `IWeapon` (relacija *has*).

`IInput` objekt služi da bi se kontrole igrača mogle promijeniti u bilo kojem trenutku, dok `IWeapon` predstavlja oružje koje igrač trenutno posjeduje. Od implementacija za `IInput` postoji `PCInputDevice` koji služi za korištenje tipkovnice i miša za kontroliranje igrača te `TouchInputDevice` za kontroliranje pomoću ekrana osjetljivog na dodir.

`IWeapon` je sučelje koje definira ponašanje oružja. Od mogućih oružja postoje `Rifle` te `Beam`. Svaki od njih ima svoje metke, `RifleBullet` te `BeamBullet`. Svi ovi koncepti su razdvojeni da bi se kasnije lakše moglo dodavati nove funkcionalnosti te da bi testiranje samih bilo jednostavnije.

Fizika je u ovoj implementaciji najjednostavnija jer ne postoje nikakvi dodatni slojevi kôda koji moraju komunicirati sa sustavom fizike. U skripti `CollisionRule` nalazi se sva logika fizike te ona direktno pristupa `Game` skripti ukoliko se dogodila kolizija između metka i neprijatelja.

5. Usporedba arhitektura i igara

Performanse igre najviše ovise o tome koliko na sceni ima neprijatelja koji se kreću prema baznom brodu. Neke arhitekture će moći izdržati više brodova, a neke manje, ali za procjenu performansi se moraju odrediti stvarni brojevi s kojima će se provoditi testiranje. Analizom svih arhitektura, procijenjeno je da će se najbolji rezultati pokazati ako se koriste 3 testiranja s različitim brojem neprijatelja: 10.000, 15.000 te 25.000.

Implementirane igre uspoređivat će se na nekolicini parametara, a to su:

- Prosječni FPS u prvih 10 sličica
- Prosječni FPS tijekom cijele igre
- Minimalni FPS tijekom cijele igre
- Maksimalni FPS tijekom cijele igre
- Prosječno iskorištavanje CPU-a
- Maksimalno iskorištavanje CPU-a

5.1. Praćenje podataka

Za potrebe testiranja, napisana je još jedna dodatna skripta, `PerformanceTest`. Ona ima funkcije za inicijaliziranje, te ako se zove svaku sličicu, prikupljat će podatke o performansama igre. Na kraju rada skripta ispiše prikupljene podatke u `output_log` datoteku Unity igre na Windows platformi ili na tok za otklanjanje neispravnosti (engl. *debug stream*) na Android uređaju. Za Unity ECS implementaciju također je dodan jedan sustav koji u `Update()` funkciji zove tu skriptu. Novi sustav je stvoren radi toga što ne postoji nekakvi globalni `Update()` poziv koji bi mogao odvijati tu logiku, već se sve izvodi preko sustava. Kod ostalih implementacija postoji središnja `MonoBehaviour` komponenta te se testiranje performansi odvija preko nje. U skripti se, ako se pozove `update()`, svaku sličicu gleda `Time.unscaledDeltaTime`. što je razmak u vremenu od prošle sličice koji nije skaliran preko `Time.timeScale`. Sama implementacija praćenja vremena je prikazana u kôdu 5.1. Preko toga se izračunava trenutni broj sličica te dobiveni izračun dodaje u određene skupove (minimalni FPS, maksimalni FPS, prosječni FPS itd.). Kada je vrijeme ispisivanja podataka, a to je u igrama

kada se uništi zadnji neprijatelj, treba se pozvati `finish()`. Ono podiže zastavicu (engl. *flag*) da skripta prestane pratiti FPS te ispisuje dobivene brojeve u konzolu. Na Android platformi je malo kompleksnije doći do ispisanih podataka jer se uređaj treba spajati na Android Monitor aplikaciju koja dolazi s Android SDK (engl. *Software Development Kit*, skraćeno SDK).

```
fpsDelta += (Time.unscaledDeltaTime - fpsDelta) * 0.1f;
float fps = 1.0f / fpsDelta;
```

Kôd 5.1 Način izračuna FPS-a u jednoj sličici

U istoj skripti za praćenje FPS-a implementirano je i praćenje opterećenosti CPU-a. To je implementirano preko `PerformanceCounter` klase. Tijekom testiranja ovim načinom, testovi su pokazali da je opterećenost procesora na 100% u bilo kojem trenutnu igre. To, naravno, nije točno te se zbog neispravnosti rada objekta mora naći drugi način kako bi se pratio ovaj faktor. Za Windows platformu je na kraju bio napisan novi program u obliku Windows konzolnog programa. Taj program također koristi `PerformanceCounter` objekt da prati opterećenost CPU-a. U konzolnom programu, objekt vraća brojeve koji variraju što pokazuje da nema nikakvih problema u praćenju opterećenja. Kôd 5.2 prikazuje kreiranje objekta tipa `PerformanceCounter` te mogućnost postavljanja vrijednosti parametara `InstanceName`. Parametru se može predati naziv procesa koji se prati, tako da se svaki puta kada se mijenja platforma koja se testira, treba promijeniti taj parametar.

```
cpuCounter.CategoryName = "Process";
cpuCounter.CounterName = "% Processor Time";
cpuCounter.InstanceName = "SpaceInvadersECS";
cpuCounter.NextValue();
...
```

Kôd 5.2 Inicijalizacija `PerformanceCounter` objekta

Konzolna aplikacija prestaje s radom u onom trenutku kada više ne može naći proces s imenom koji je dan u `InstanceName` parametru. Pri prestanku rada, aplikacija ispisuje dobivene podatke, odnosno prosječnu potrošnju te maksimalnu potrošnju.

Za Android praćenje opterećenosti CPU-a, korišten je Android Monitor. Na toj aplikaciji je moguće pratiti performanse android uređaja, te se preko njega može očitati traženi parametar iskorištenosti CPU-a.

5.2. Usporedba na Windows platformi

Za potrebe usporedbe na Windows platformi koristilo se računalo sa sljedećim specifikacijama:

- Procesor Intel Core i5-7500 s četiri jezgre i brzinom 3.40 GHz
- 16 GB radne memorije
- Nvidia GeForce GTX 1080 grafička kartica
- Windows 10 operacijski sustav

Sami brojevi testiranja su prikazani u tablici 5.1.

Tablica 5.1 Rezultati testiranja performansi na Windows platformi

Arhitekture	Prosječni FPS u prvih 10 sličica	Minimalni FPS	Maksimalni FPS	Prosječni FPS	Maksimalno CPU iskorištavanje	Prosječno CPU iskorištavanje
OOP 10.000	4.3075	6.6684	63.7872	47.0216	126%	180%
OOP 15.000	4.0002	5.9596	39.3618	28.4138	130%	183%
OOP 25.000	3.4693	4.6571	17.1447	14.4314	134%	180%
Vlastiti ECS 10.000	2.6151	3.6372	51.3402	45.1489	125%	175%
Vlastiti ECS 15.000	1.7166	1.9621	33.5725	29.4742	126%	157%
Vlastiti ECS 25.000	1.037	0.7112	19.3067	17.2996	127%	167%
Unity ECS 10.000	4.1384	6.4361	61.1817	56.3978	101%	151%
Unity ECS 15.000	4.1668	6.5124	56.3883	41.7737	102%	142%
Unity ECS 25.000	4.0392	6.1037	33.1737	24.5139	105%	146%

U prvom stupcu tablice je popis arhitektura te broj neprijatelja s kojim se trenutno testira. Ostali stupci predstavljaju broj sličica u sekundi osim zadnja dva koja predstavljaju postotak iskorištenosti procesorskog čipa. Glavni pokazatelj performansi arhitekture je zasigurno prosječni FPS. Kao što vidimo u podacima, gotovo sve arhitekture su blizu ili

preko 30 FPS-a. Taj broj je značajan zbog toga što je zamjena sličica tom brzinom oku nezamjetna te se ne vidi nikakvo zastajkivanje. Taj broj može varirati od osobe do osobe, neki vide razliku na 35 FPS-a, a neki ne vide na 24. Prosječan FPS se računa od početka igre sve dok nije zadnji neprijatelj uništen. Što se više neprijatelja uništava to je veći FPS, no za kraj igre je potrebno uništiti 6% neprijatelja tako da odstupanja u FPS-u nisu prevelika na početku i kraju igre. OOP implementacija i implementacija na vlastitom ECS sustavu imaju slične performanse, no ECS je malo performantniji. U testu s manje neprijatelja OOP je imao 2 FPS-a više, dok s više neprijatelja ECS ga je nadmašio za 3 FPS-a. Važno je imati na umu da se veća razlika između FPS-a primijeti kada je manji broj sličica po sekundi nego kada je veći, tako da ECS generalno daje ugodniji doživljaj igranja.

U vlastitom ECS sustavu te OOP implementaciji vidimo degradaciju FPS-a u prvih 10 sličica, dok je u Unity ECS-u oko stabilnih 5 FPS-a. U svim implementacijama je taj broj vrlo nizak zbog toga što se na početku stvaraju svi entiteti, uključujući sve neprijatelje, te se inicijaliziraju sve ECS komponente. Stvaranje objekata preko `GameObject.Instantiate()` ili `new GameObject()` puno sporiji nego `EntityManager.CreateEntity()` te zbog toga Unity ECS ima bolje performanse u tom području. U sljedećoj tablici - Tablica 5.2 - možemo vidjeti razliku u brzini između stvaranja 100.000 objekata u milisekundama (ms).

Tablica 5.2 Razlika u brzinama stvaranja 100.000 objekata (ms) na Windows platformi

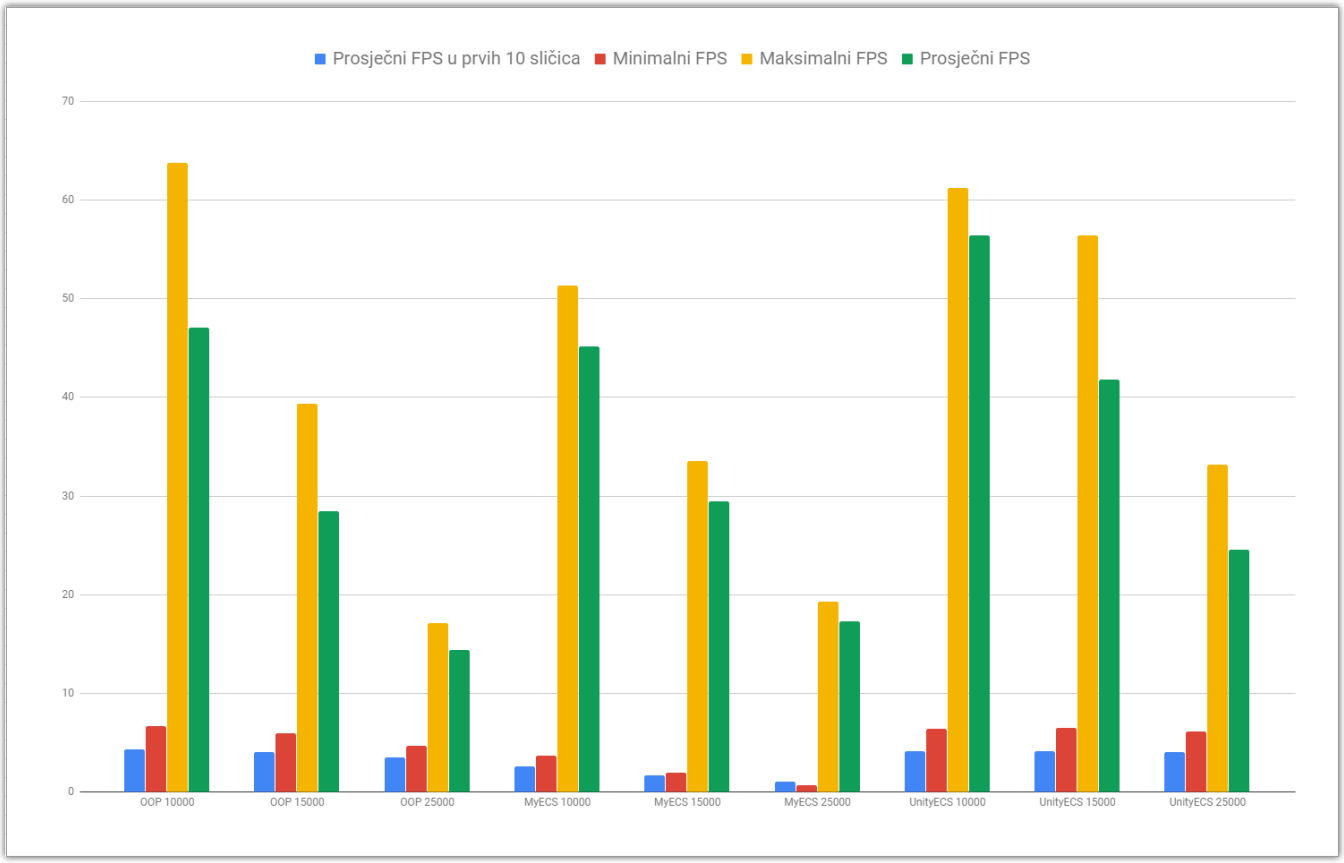
`EntityManager.CreateEntity()`	`new GameObject()`	`GameObject.Instantiate()`
74 ms	453 ms	634 ms

U minimalni FPS broj ne ulazi FPS prvih 10 sličica. Razlog zašto je minimalan FPS na svim testovima ovoliko mali, više-manje je oporavak računala od instanciranja objekata (rad GC-a), daljnje instanciranje neprijatelja koji se stvaraju s vremenskim odmakom od prvobitnih neprijatelja te pomicanje neprijatelja velikom brzinom do njihovih mjesta. Brzo prvobitno pomicanje neprijatelja je zahtjevno jer u pozadini koristi `Vector3.Distance()`, što u pozadini koristi korjenovanje te zbog toga ima loše performanse.

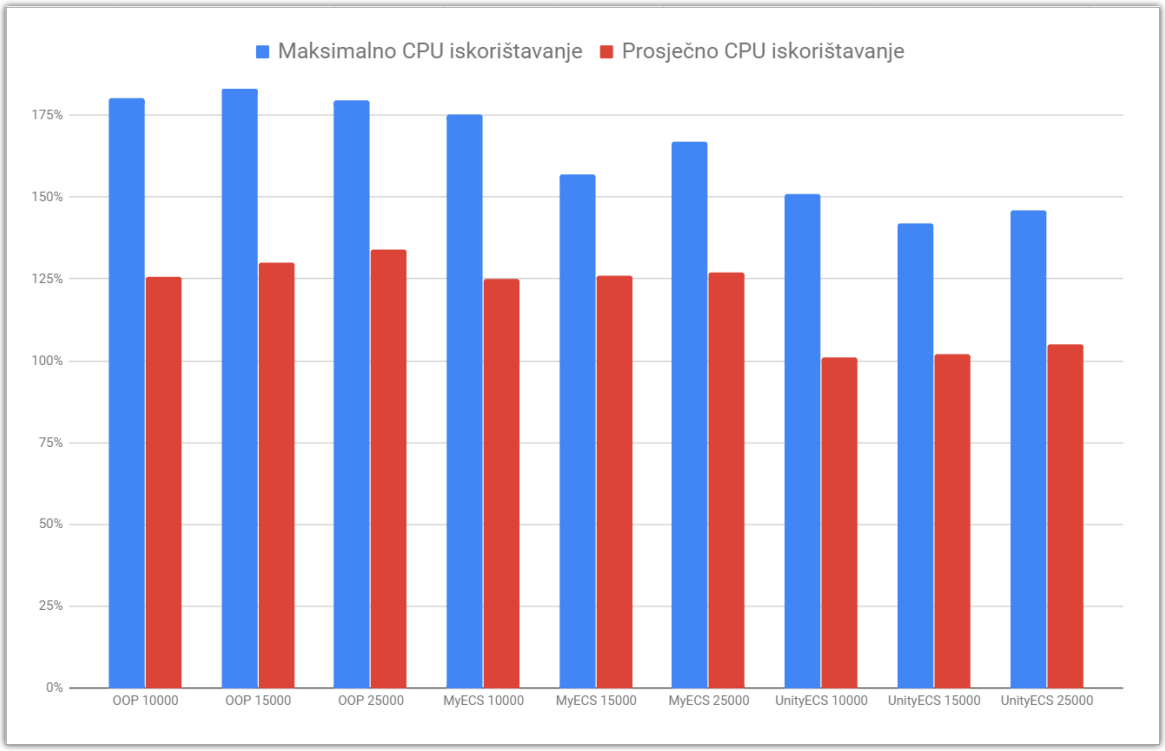
Postotak iskorištavanja CPU-a varira od testa do testa, ali možemo zaključiti da je na Unity ECS-u smanjena potrošnja CPU-a, što doprinosi manjoj potrošnji energije ili oslobađanje procesora za rad na drugim procesima. Razlog tome je do toga što je prevoditelj Unity

ECS-a optimiziran tako da bolje iskorištava procesorsku snagu, te stvara kôd koji je bolji u konkurentnosti (engl. *Multithreading*). Broj može premašiti 100% jer se dobiva zbrajanjem postotaka iskorištavanja svih jezgra procesora, tako da ako se jedna jezgra koristi 50% a druga 60%, a ukupna iskorištenost je 110%.

Radi lakše vizualizacije podataka, stvoreni su grafički prikazi u slikama 5.1 te 5.2 koji dobivene brojeve u testovima.



Slika 5.1 Grafički prikaz FPS-a na Windows platformi



Slika 5.2 Grafički prikaz iskorištavanja CPU-a na Windows platformi

5.3. Usporedba na Android platformi

Za potrebe testiranja na Android platformi, potrebno je nadograditi sve implementacije s kôdom za upravljanje brodom preko ekrana osjetljivog na dodir. Virtualna upravljačka ručica je dodana na ekranu kojom će igrač upravljati brodom, te kada se pritisne s drugim prstom bilo gdje, brod će pucati.

Android uređaj na kojem će se vršiti testiranje je sljedeći:

- Samsung S8+
- Procesor Exynos M2 s četiri jezgre s brzinom 2,30 GHz te Cortex A53 s četiri jezgre s brzinom 1.69 GHz
- 4 GB radne memorije
- GPU adapter za ARM SoC (engl. *System on a Chip*, skraćeno SoC): ARM Mali-G71
- Android 8.0.0 operacijski sustav
- Rezolucija FHD+ (2220 x 1080)

Zbog slabijih karakteristika uređaja u odnosu na računalo u prošlom testiranju, ovdje će se koristiti manji broj neprijatelja na sceni. Broj neprijatelja će biti otprilike oko 10 puta manji, odnosno 1.500, 2.500 te 3.000 neprijatelja na sceni.

Rezultati testiranja su prikazani u tablici 5.3:

Tablica 5.3 Rezultati testiranja performansi na Android platformi

Arhitekture	Prosječni FPS u prvih 10 sličica	Minimalni FPS	Maksimalni FPS	Prosječni FPS	Prosječno CPU iskorištavanje	Maksimalno CPU iskorištavanje
OOP 1.500	3.9328	5.9064	49.1334	23.7662	108%	156%
OOP 2.500	3.0188	1.1678	35.9381	16.1168	147%	156%
OOP 3.000	2.6876	0.9076	39.5486	14.3896	131%	170%
Vlastiti ECS 1.500	3.3697	4.8442	56.3289	46.3609	155%	164%
Vlastiti ECS 2.500	3.7139	5.4285	55.8567	41.4452	149%	155%
Vlastiti ECS 3.000	2.5645	3.4212	49.9223	32.1757	167%	167%
Unity ECS 1.500	3.0216	4.618	48.5185	27.4312	173%	190%
Unity ECS 2.500	2.7133	3.9865	33.2397	22.7062	168%	185%
Unity ECS 3.000	2.5373	3.5917	29.7424	18.90853	205%	232%

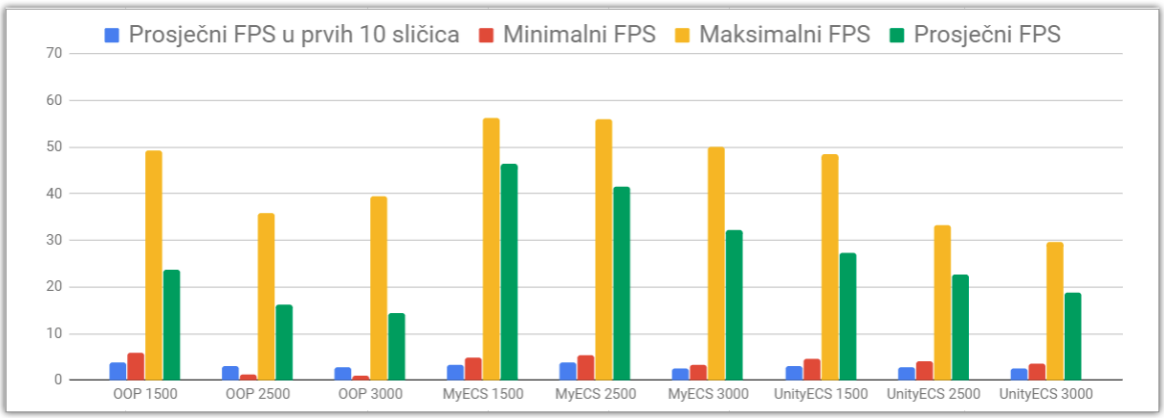
Rezultati su iznenađujuće drugačiji nego na Windows platformi. OOP implementacija prati trendove kao i na Windows platformi, no ostale implementacije imaju neočekivane rezultate. Unity ECS je slabiji od vlastite implementacije ECS-a, što ovisi o krajnjem kôdu koji je generiran. Generirani kôd je optimiziran za Windows i neke ostale samostalne (engl. *standalone*) operacijske sustave, no za Android nije.[17] Uređaj koji se testirao, te većina Android uređaja, koristi ARM procesor[18] dok testirani CPU na Windows platformi je na x64 arhitekturi. Još jedna velika razlika je u setu instrukcija koje ta dva procesora imaju, a to su RISC (engl. *Reduced instruction set computer*, skraćeno RISC) za ARM procesor te CISC (engl. *Complex instruction set computer*, skraćeno CISC) za x64 procesor.

Broj prosječnog FPS-a varira od testa do testa, ali možemo vidjeti da OOP implementacija ima najslabije performanse. Ona ima maksimalni FPS veći od Unity ECS implementacije, ali ako se gleda prosječni FPS, onda ispada lošija. Neočekivano, vlastita implementacija ECS-a ima najbolje performanse u svim testovima. U gotovo svim faktorima koji su se testirali ta implementacija najbolje i najefikasnije iskorištava dane resurse. Razlika između

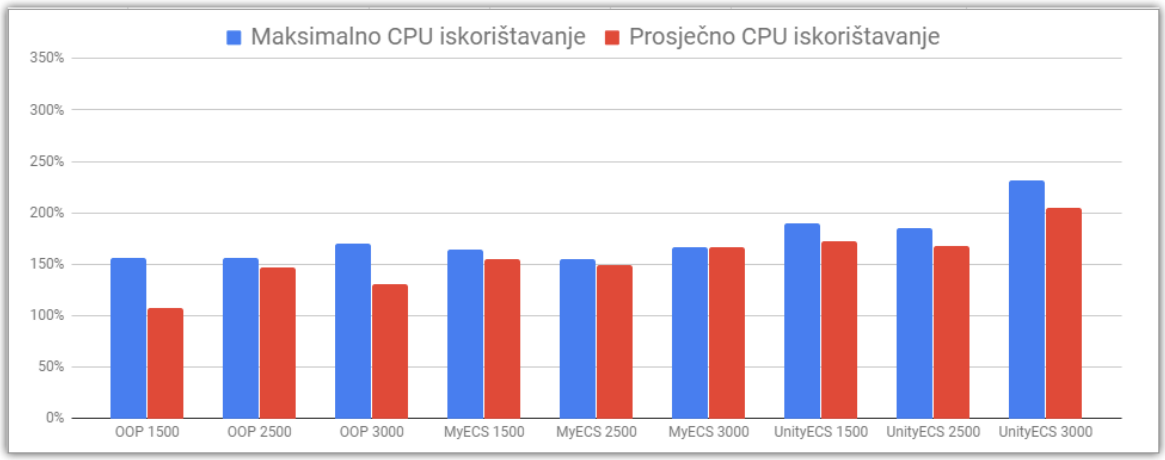
OOP i vlastite ECS implementacije, može biti u tome što na slabijem uređaju OOP teže izvršava svoje instrukcije te je razlika očitija.

Trend iskorištenosti CPU-a je također interesantan. Potrošnja CPU-a na vlastitoj ECS arhitekturi i OOP igra je gotovo identična, s malo boljim performansama na OOP implementaciji. Nasuprot dobrim brojevima na testovima Windows platforme, Unity ECS na Androidu iskorištava CPU puno više nego bilo koji drugi test. Na testu s 3.000 neprijatelja možemo vidjeti koliko je Unity ECS neefikasan kada pogledamo prosječan FPS od 18.9 sličica po sekundi i prosječnu iskoristivost CPU-a od 205%.

Kao i u prošlom testiranju, kreirani su grafički prikazi rezultata u slikama 5.3 i 5.4 radi lakšeg pregleda brojeva.



Slika 5.3 Grafički prikaz FPS-a na Android platformi



Slika 5.4 Grafički prikaz iskorištavanja CPU-a na Android platformi

Zaključak

U ovom radu prikazan je rad različitih arhitektura. U programerskom svijetu postoji pozamašan broj različitih arhitektura, a one se, na dnevnoj bazi, unapređuju i inoviraju. Ovo područje struke je vrlo dinamično i kompleksno za savladati.

Teorija i praksa je do sada pokazala da je OOP jedan od najrasprostranjenijih arhitektura zbog jednostavnosti poimanja programskih problema, mogućnosti proširivanja kôda, ponovnog upotrebljavanja kôda i slično. Svim tim postupcima, kôd je postajao lakši programeru, a teži procesoru. Programeri su primijetili da bi mogli drugačije pristupiti rješavanju problema performansa te su na arhitekturnoj razini počeli primjenjivati ECS.

Unity je jedan od prvih platformi koje je počelo gurati ECS prema široj publici, lagano mijenjajući način pisanja kôda u njihovom okruženju. U vrijeme pisanja ovog rada, ECS okvir u Unityju još je u razvoju. Zbog te se činjenice očekuju poneke greške u radu. Neovisno o tome, sustav je zreo za prve dojmove i testiranja.

Da bi se moglo usporediti performanse OOP arhitekture s ECS-om, morala se stvoriti ECS implementacija koja nema koristi od prevoditelja koliko ima Unityjeva implementacija ECS-a. Vlastiti se ECS sustav napisao s performansama u vidu te je vrlo optimiziran.

Testovi u ovom radu pokazali su djelomičnu točnost izjave da su programi postali kompleksniji za izvođenje u OOP-u. Ukoliko je ECS kôd optimiziran od strane prevoditelja biti će performantniji te efikasniji od bilo koje druge implementacije. Ukoliko nije, razlike u performansama između OOP i ECS-a vrlo su male u korist ECS-a. Moderni prevoditelji optimiziraju generirani kôd te pisanje OOP-a postaje sve ugodnije, kako programeru tako i procesoru.

Popis kratica

OOP	*Object oriented programming*	Objektno orijentirano programiranje
DDP	*Data Driven Programming*	Podatkovno orijentirano programiranje
ECS	*Entity Component System*	Entitet komponenta sustav
FPS	*Frames Per Second*	Sličica po sekundi
UI	*User Interface*	Korisničko sučelje
SoC	*System on a Chip*	Sustav na čipu
API	*Application Programming Interface*	Aplikacijsko programsko sučelje
SRP	*Single Responsebility Principle*	
SDK	*Software Development Kit*	
CPU	*Central Processing Unit*	
RISC	*Reduced instruction set computer*	
CISC	*Complex instruction set computer*	
GC	*Garbage Collector*	

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] University of Minnesota Duluth, Object-Oriented Principles https://www.d.umn.edu/~gshute/softeng/presentations/oo-principles.xhtml, listopad 2014

[2] ECKEL, B. Thinking in C++, Volume 1, 2nd Edition, 2003

[3] MONO PROJECT, Compiling LLVM support, https://www.mono-project.com/docs/advanced/mono-llvm/, prosinac 2018

[4] UNIVERSITY OF RHODE ISLAND, DATA-DRIVEN PROGRAMMING https://homepage.cs.uri.edu/~thenry/resources/unix_art/ch09s01.html, travanj. 2005.

[5] COFFEBRAINGAMES, HOW UNITY'S ECS EXPANDS YOUR OPTIMIZATION SPACE, https://coffeebraingames.wordpress.com/2018/08/19/how-unitys-ecs-expands-your-optimization-space, kolovoz 2018.

[6] ORACLE, WHAT IS AN OBJECT?. https://docs.oracle.com/javase/tutorial/java/concepts/object.html, 2017

[7] UNITY TECHNOLOGIES, ENTITYCOMPONENTSYSTEMSAMPLES https://github.com/Unity-Technologies/EntityComponentSystemSamples/tree/master/Samples/Assets/TwoStickShooter/Hybrid, 2018.

[8] Unity TECHNOLOGIES, EXECUTION ORDER OF EVENT FUNCTIONS, https://docs.unity3d.com/Manual/ExecutionOrder.html, 2018

[9] Simonov V, 10000 Update() calls, https://blogs.unity3d.com/2015/12/23/1k-update-calls/, 2015

[10] Ferreira C, Get Started with the Unity Entity Component System (ECS), C# Job System, and Burst Compiler https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler, svibanj 2018

[11] Unity TECHNOLOGIES,COMPONENTDATA https://github.com/Unity-Technologies/EntityComponentSystemSamples/blob/1d41ad0ab87331ca5b1814eac60795e2e47deb24/Documentation/reference/component_data.md, 2018

[12] Unity TECHNOLOGIES, SHARED COMPONENTDATA https://github.com/Unity-Technologies/EntityComponentSystemSamples/blob/1d41ad0ab87331ca5b1814eac60795e2e47deb24/Documentation/reference/shared_component_data.md, 2018

[13] Unity TECHNOLOGIES, BURST USER GUIDE https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html, 2018

[14] Sschmid, Entitas-CSharp https://github.com/sschmid/Entitas-CSharp, 2018

[15] Leopotam, ecs
https://github.com/Leopotam/ecs, 2019

[16] Oodesign, Single Responsibility Principle
https://www.oodesign.com/single-responsibility-principle.html

[17] Ferreira C, The Burst Compiler,
https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler, svibanj 2018

[18] Unity TECHNOLOGIES, MOBILE (ANDROID) HARDWARE STATS
https://web.archive.org/web/20170808222202/http://hwstats.unity3d.com:80/mobile/cpu-android.html, 2017

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, datum.

Ime Prezime

Algebra
visoka škola za
primijenjeno računarstvo

Primjena ECS arhitekture u izradi računalnih igara

Pristupnik: Jurica Adamek, 0321003231

Mentor: dr. sc. Goran Đambić, v. pred.