Izrada igre bazirane na podatkovno orijentiranom tehnološkom stogu

Jerković, Ivan

VELEUČILIŠTE U BJELOVARU
PREDDIPLOMSKI STRUČNI STUDIJ RAČUNARSTVO

IZRADA IGRE BAZIRANE NA PODATKOVNO ORIJENTIRANOM TEHNOLOŠKOM STOGU

Završni rad br. 04/RAČ/2021

Ivan Jerković

Bjelovar, Listopad 2021

obrazac ZR - 001

Veleučilište u Bjelovaru

Trg E. Kvaternika 4, Bjelovar

1. DEFINIRANJE TEME ZAVRŠNOG RADA I POVJERENSTVA

Kandidat: **Jerković Ivan** Datum: 19.07.2021. Matični broj: 001970

JMBAG: 0246059543

Kolegij: **RAZVOJ RAČUNALNIH IGARA**

Naslov rada (tema): **Izrada igre bazirane na podatkovno orijentiranom tehnološkom stogu**

Područje: **Tehničke znanosti** Polje: **Računarstvo**

Grana: **Programsko inženjerstvo**

Mentor: **Ante Javor, struč. spec. ing. comp.** zvanje: **predavač**

Članovi Povjerenstva za ocjenjivanje i obranu završnog rada:

1. **Krunoslav Husak, dipl. ing. rač., predsjednik**
2. **Ante Javor, struč.spec.ing.comp., mentor**
3. **dr.sc. Zoran Vrhovski, član**

2. ZADATAK ZAVRŠNOG RADA BROJ: 04/RAČ/2021

U radu je potrebno izraditi prototip igre konvencionalnim putem koja ima razrađene osnovne mehanike i logiku igre. Na temelju prototipa igre potrebno je kreirati istu igru s istim osnovnim mehanikama i logikom ali baziranu na Unity alatima koji su dio podatkovno orijentiranog tehnološkog stoga. Potrebno je napraviti test performansi obje igre s istim parametrima te usporediti dobivene rezultate. Potrebno je zaključiti prednosti i mane podatkovno orijentiranog tehnološkog stoga naspram klasičnog programiranja.

Zadatak uručen: 19.07.2021.

Mentor: **Ante Javor, struč. spec. ing. comp.**

REPUBLIKA HRVATSKA
BJELOVAR
VELEUČILIŠTE U BJELOVARU

Zahvala

Zahvaljujem se svim profesorima koji su meni i mojim kolegama pomogli pri savladavanju gradiva. Zahvaljujem na ukazanom strpljenju i profesionalnosti. Posebno mom mentoru Anti Javoru koji me mentorirao pri izradi ovog rada i pomogao da se rad dovrši. Osim profesorima želim se zahvaliti i upravi veleučilišta, osoblju i voditeljima studija koji pokazuju brigu prema studentima i entuzijazam prema svome poslu. Zahvaljujem se što su učinili ovo cjelokupno iskustvo studiranja ugodnim. Mogu s ponosom reći da sam sam studirao na Veleučilištu u Bjelovaru.

1. UVOD

Tema završnog rada je izrada igre korištenjem Unity programskog okvira za igre te podatkovno orijentiranog tehnološkog stoga (engl. *data-oriented technology stack*, skraćeno DOTS). Sve većim rastom industrije igara, veći je fokus stavljen na ovu granu ekonomije [1]. Time se više pažnje posvećuje razvojem novih tehnologija za izradu igara. Poseban rast prihoda ostvaruju igre na mobilnim uređajima [1]. Mobilni uređaji za razliku od klasičnih konzola i stolnih računala, zbog svoje prenosivosti i veličine su limitiraniji po pitanju hardvera. Osim limitacija komputacijske mogućnosti postoji i limitacija baterije koja pokreće mobilne uređaje. Slabija komputacijska mogućnost limitira složenost igre a time i brzinu izvršavanja igre. Baterija limitira koliko dugo korisnik može igrati igru na mobilnom uređaju. Pritom će zahtjevnije igre trošiti više resursa i time smanjiti dostupno vrijeme za igru. Unity je osmislio novu tehnologiju kako bi ista igra zahtijevala slabiji hardver. Osim zahtijevanja slabijeg hardvera, prema izvoru [2], igre su i bolje optimizirane za procesore mobilnih uređaja. Rad prikazuje prednosti i mane nove tehnologije kako bi se utvrdilo koliko utjecaja ima na performanse u odnosu na dosadašnju praksu izrade igara.

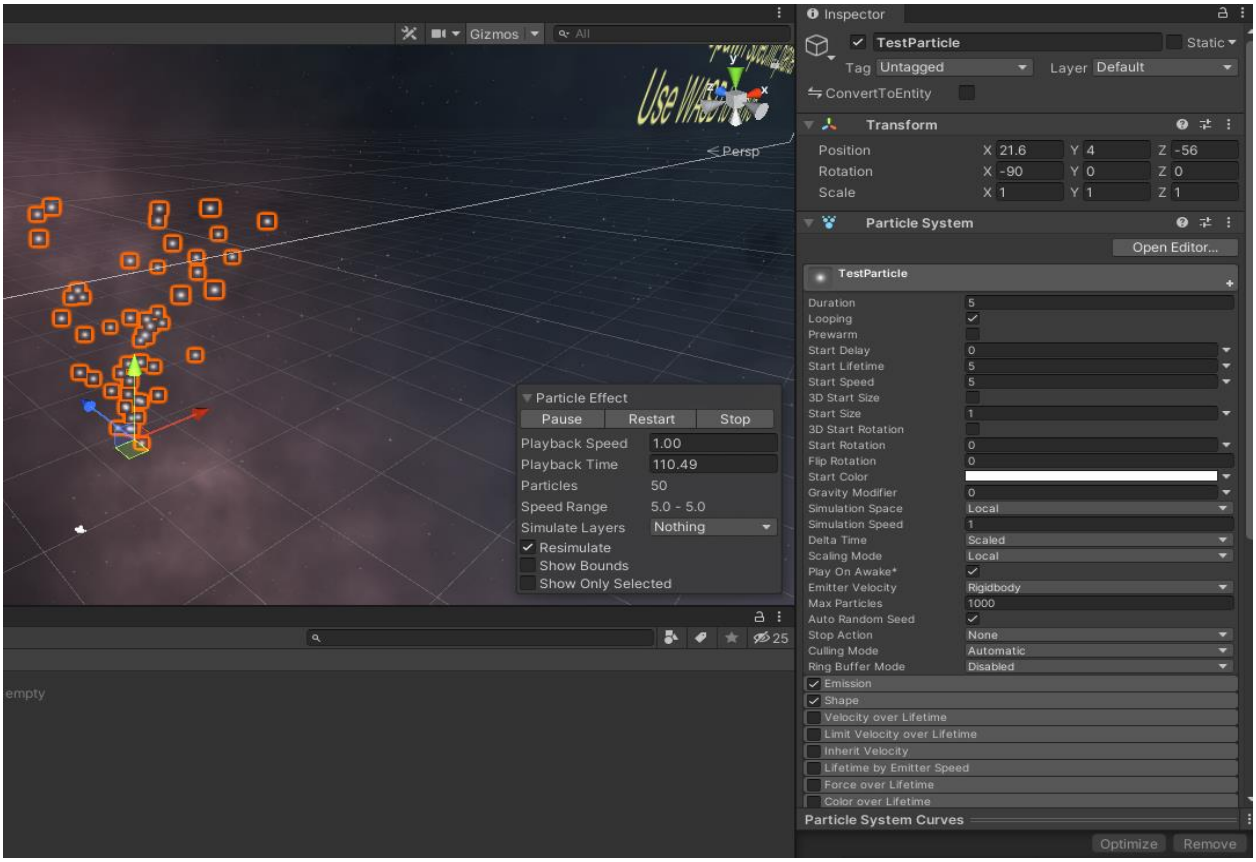
Završni rad prikazuje usporedbu performanse igre kreirane na klasični Unity način s objektima igre (engl. *game object*) te igre kreirane pomoću nove Unity DOTS tehnologije. Klasični Unity način koristi arhitekturu objektno orijentiranog programiranja (engl. *object oriented programming*) dok Unity DOTS koristi podatkovno orijentirani dizajn (engl. *data oriented design*).

Prvo poglavlje je uvodno poglavlje gdje je ukratko opisana tema rada. U drugom poglavlju su opisane tehnologije korištene u radu i objašnjene su osnovne terminologije. Treće poglavlje opisuje igru, koji je cilj igre, arhitektura igre te opis koda igre. Nadalje, opisano je kako se igra ponaša te na koji način je to implementirano. Četvrto poglavlje objašnjava parametre, uvjete, opremu i rezultate testiranja.

2. KORIŠTENE TEHNOLOGIJE

2.1. Programski okvir za igre

Programski okvir za igre (engl. *game engine*) je temelj svake računalne igre. U počecima razvoja industrije računalnih igara nije postojala standardizacija. Svaka igra je rađena paralelno s razvojem svih komponenti igre od početka. Primjerice, komponente za prikaz slike na zaslon bi bile kreirane za svaku igru zasebno zbog velikih limitacija tadašnjeg hardvera [3]. Rastom industrije nastala je potreba za standardizacijom i nastao je programski okvir za igre. Programski okvir za igre je skupina alata koji uvelike olakšavaju izradu igara. Programski okvir za igre čini temelj igre te sadrži univerzalne komponente koje su primjenjive na većinu igara. Brine se o iscrtavanju slike (engl. *image rendering*), osvjetljenju, simuliranju 3D svijeta, primanju kontrola od strane igrača, reproduciranja animacija te ostalih funkcionalnosti koje sadrži svaka igra. Također može sadržavati dodatne alate koji olakšavaju kreiranje specifičnog sadržaja. Primjerice, Unity programski okvir za igre sadrži sustav čestica (engl. *particle system*). Alat u kojem korisnik može brzo kreirati čestice za efekte. Bez alata bi to bilo potrebno raditi izvan programskog okvira za igre te dodati efekte ručno unutar projekta. Slika 2.1 prikazuje uređivanje čestica unutar Unity programskog okvira za igre pomoću alata za čestice.

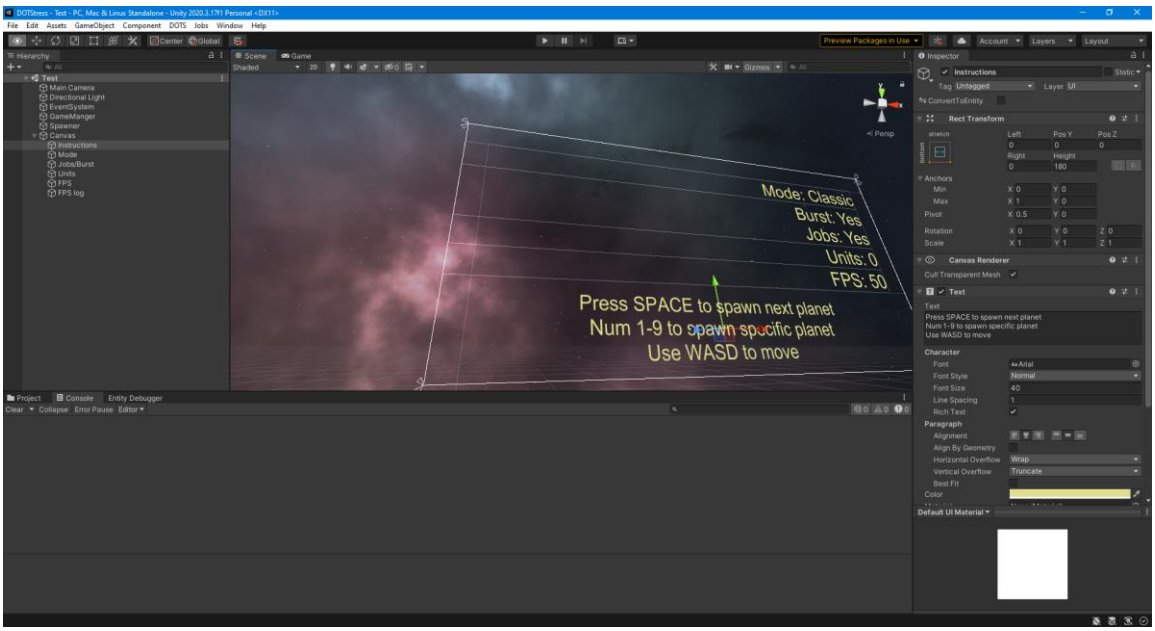


Slika 2.1: Alat za uređivanje čestica

2.2. Unity

Programski okvir za igre korišten za kreiranje programskog dijela ovog rada je Unity. Unity je višeplatformski programski okvir za igre s mogućnosti prilagođavanja sadržaja za iOS, Android, Windows, Linux, PlayStation, XBOX, Nintendo Switch te mnoge druge platforme. Prednost Unity programskog okvira za igre je prilagodba za razne tipove igara, od brzih pucačina iz prvog lica do sporih taktičkih igara. Veliki studiji za razvoj video igara kreiraju svoje programske okvire za igre. Unity se odlučio na poslovni model u kojem kreiraju programski okvir za igre te ga prodaju drugim studijima. Poslovni model koji koriste je također specifičan jer Unity je potpuno besplatan za korištenje i uključuje sve funkcionalnosti programskog okvira za igre. Kreiran je kako bi ga koristilo što više developera. Model naplaćivanja se konstantno mijenja, no trenutni model u rujnu 2021. godine jest da je besplatan do zarade od 100,000 američkih dolara. Nakon navedenog iznosa, zahtjeva kupnju Unity Plus licence do zarade od 200,000 američkih dolara te kupnju Unity Pro ili Unity Enterprise licence iznad zarade od 200,000 američkih dolara. Trenutni poslovni model ne uključuje dijeljenje zarade [4].

Unity sadrži osnovne funkcionalnosti koje svaka igara koristi i te funkcionalnosti su odmah dostupne u svakom projektu. No, Unity koristi i sustav paketa gdje se razne funkcionalnosti i alati mogu naknadno dodati i koristiti, ali nisu nužni za svaki projekt. Sustav paketa također omogućava da korisnici kreiraju svoje alate i funkcionalnosti te ih podijele s drugim korisnicima. Korisnici jednostavno mogu učitati pakete drugih korisnika i imaju istu funkcionalnost unutar svog projekta. Pakete je moguće kontrolirati preko upravitelja paketima (engl. *package manager*). Slika 2.2 prikazuje korisničko sučelje Unity programskog okvira za igre.



Slika 2.2: Prikaz korisničkog sučelja Unity programskog okvira za igre

2.3. Podatkovno orijentirani tehnološki stog

DOTS je skupina paketa tehnologija koje mijenjaju način na koji se kreiraju igre unutar Unity programskog okvira za igre te uvelike poboljšava performanse igara. DOTS je još uvijek u fazi testiranja te nije dio službenog izdanja. Testni paketi koji čine DOTS se mogu preuzeti i koristiti, no Unity tim upozorava da se paketi mogu ponašati nepredvidivo.

Tri osnovna paketa koji čine temelj DOTS su ***Entity Component System*** (skraćeno ECS), ***C# Jobs*** i ***Burst*** **prevoditelj** o kojima će se više pričati u nastavku rada. Bitna značajka DOTS tehnologije jest da omogućuje jednostavno korištenje višedretvenosti kako bi se procesi i podaci paralelizirali te uvelike ubrzali izvođenje igre. DOTS u trenutnom stanju nije samostalan već je samo ekstenzija trenutnog programskog okvira za igre. Nije moguće kreirati potpuni projekt koristeći DOTS, no dijelovi projekta se mogu kreirati pomoću DOTS tehnologije i optimizirati te dijelove projekta. Za budućnost je planiran potpuni prelazak na korištenje DOTS tehnologije, no Unity će paralelno uz DOTS omogućiti i korištenje klasičnog Unity načina za razvoj igre [5].

Unity DOTS je programski okvir za igre koji se razvija iz temelja na DOTS tehnologiji. Sve je kreirano kako bi bilo u što boljoj harmoniji s DOTS tehnologijom i omogućilo što bolje performanse, od izvođenja igara do brzine izrade projekata. Osim gore spomenuta tri temeljna paketa DOTS tehnologije, dodatni paketi poboljšavaju temeljne pakete ili olakšavaju korištenje DOTS tehnologije. Trenutni paketi koji upotpunjuju DOTS tehnologiju su DOTS Editor koji sadrži razne alate i informacije o DOTS komponentama unutar Unity razvojnog okruženja. Hibridni ocrtavač (engl. *hybrid renderer*) je paket koji omogućuje iscrtavanje entiteta. Entiteti u DOTS sustavu rade drugačije jer postojeća *renderer* komponenta ne može iscrtati objekte, no *hybrid renderer* prenosi potrebne informacije postojećim *hybrid renderer* paketima kako bi mogli iscrtati i entitete. Unity fizika je paket nove implementacije fizike koji je kreiran za DOTS tehnologiju kako bi donio efikasnu simulaciju fizike unutar igre. Paket Unity fizike omogućuje korisniku da iskoristi determinističku dinamiku krutih tijela (engl. *deterministic rigid body dynamics*) i sustav prostornih upita (engl. *spatial query system*). Uz Unity fiziku implementirana je i Havok fizika (engl. *Havok physics*). Havok fizika je napredniji fizički sustav i omogućuje kreiranje kompleksnijih fizičkih simulacija. Unity transport je paket koji omogućuje kreiranje servera i klijenta za mrežne projekte. Unity NetCode je također namijenjen razvoju mrežnih projekta te mu je glavni fokus prema kreiranju arhitekture za sinkronizaciju entiteta. Unity matematika je paket koji sadrži brojne matematičke izraze i metode koje su posebno prilagođene za *Burst* prevoditelj [6].

Na Unite LA 2018 Unity konferenciji za kreatore, predstavljen je Unity Megacity demo. Demonstracija mogućnosti Unity DOTS tehnologije u obliku futurističkog grada. Prema izvoru [7] projekt je kreiran u dva mjeseca od strane Unity ECS tima i dva umjetnika. Megacity je sačinjen

od ukupno 4.5 milijuna iscrtanih objekata, 5,000 vozila i 200,000 zasebnih objekata koji sačinjavaju zgrade. Vozila imaju svoju logiku te slijede prometnice, prometna pravila i pri tome se nikad ne sudaraju s drugim vozilima. Megacity također sadrži 100,000 zasebnih izvora zvuka poput zvukova klima uređaja, vozila i neonskih znakova. Sve to je ostvareno bez smanjenja kvalitete zvuka. Slika 2.3 prikazuje prikaz iz demonstativnog projekta Megacity.

Slika 2.3: Isječak iz demonstracije projekta Megacity [7]

2.4. Podatkovno orijentirani dizajn

Podatkovno orijentirani dizajn koda je potpuno drugačija arhitektura i način programiranja u usporedbi s objektno orijentiranim programiranjem. Objektno orijentirano programiranje je dizajnirano oko klasa pa tako u primjeru programskog koda 2.1 je prikazana klasa *Animal* s podacima koje koristi i logikom upravljanja podacima.

Programski kod 2.1: Primjer klase u objektno orijentiranom programiranju

```
class Animal
{
private:
    // Podaci
    Vector3 Position;
    int EnergyPoints;
    bool bIsAlive;

public:
    // Logika
    Animal();
    void Eat(Animal food);
    void Move(float deltaTime);
};
```

Naravno objektno orijentirano programiranje ima i druge funkcionalnosti. Postoje metode kako organizirati kod. Nasljeđivanje uvelike može pomoći u organizaciji hijerarhiji klasa. Primjerice, moguće je uvesti klasu *Organism*. Klasa sadrži podatke i metode koje bi svaki organizam svakako sadržavao. Zatim svaki tip organizma naslijedi klasu *Organism* i dodaje svoje specifične podatke i metode.

Podatkovno orijentirani dizajn zahtjeva drugačiji pristup. Programski kod 2.2 prikazuje identičnu logiku koda kao što je korištena za primjer 2.1 objektno orijentiranog programiranja.

Programski kod 2.2: Primjer u podatkovno orijentiranom dizajnu

```
struct LifeFunctions
{
    // Podaci
    Vector3 Position;
    int EnergyPoints;
    bool bIsAlive;
};

struct Animal
{
    // Podaci
    int legs;
};

class LifeSustain
{
    void OnUpdate()
    {
        Entities.ForEach((ref LifeFunctions lifeFunctions) =>
        {
            lifeFunctions.EnergyPoints = 0;
            bIsAlive = false;
        });
    }
}
```

```
class Moving
{
    void OnUpdate()
    {
        Entities.ForEach((ref LifeFunctions lifeFunctions,
                          in Animal animal) =>
        {
            lifeFunctions.Position = new Vector3(0, 1, 2);
        });
    }
}
```

Iz programskog koda 2.2 je vidljivo pojednostavljeno ponašanje životinje u podatkovno orijentiranom dizajnu. Podatkovno orijentirani dizajn se može i protumačiti kao jedna baza podataka. Iako u pozadini ne postoji konvencionalna baza podataka, podaci su dizajnirani kao relacijske baze. U gornjem primjeru svakom entitetu ili životinji su pridružene strukture *LifeFunctions* i *Animal*. *LifeFunction* i *Animal* su dvije strukture u koje se dodaju entiteti i njihove vrijednosti. *LifeFunction* struktura ima podatke *Position*, *EnergyPoints* i *bIsAlive*. Kada se kreira entitet, *LifeFunction* i *Animal* strukture dobivaju po jedan zapis s vrijednostima koje predstavljaju entitet. Princip pristupa i mijenjanja podataka entiteta odvija se po načelima konvencionalne baze podataka. U primjeru programskog koda 2.3 pretražuju se svi entiteti koji sadrže zapis u *LifeFunctions* strukturi te se nad njima vrše daljnje operacije. Slično kao i u bazi podataka, koriste se uvjeti pretraživanja. Pretraživanje vraća rezultate te se daljnje operacije izvršavaju nad vraćenim rezultatima. Primjer programskog koda 2.3 pretražuje sve organizme dok primjer programskog koda 2.4 pretražuje samo životinje.

Programski kod 2.3: Primjer pretraživanja entiteta organizama

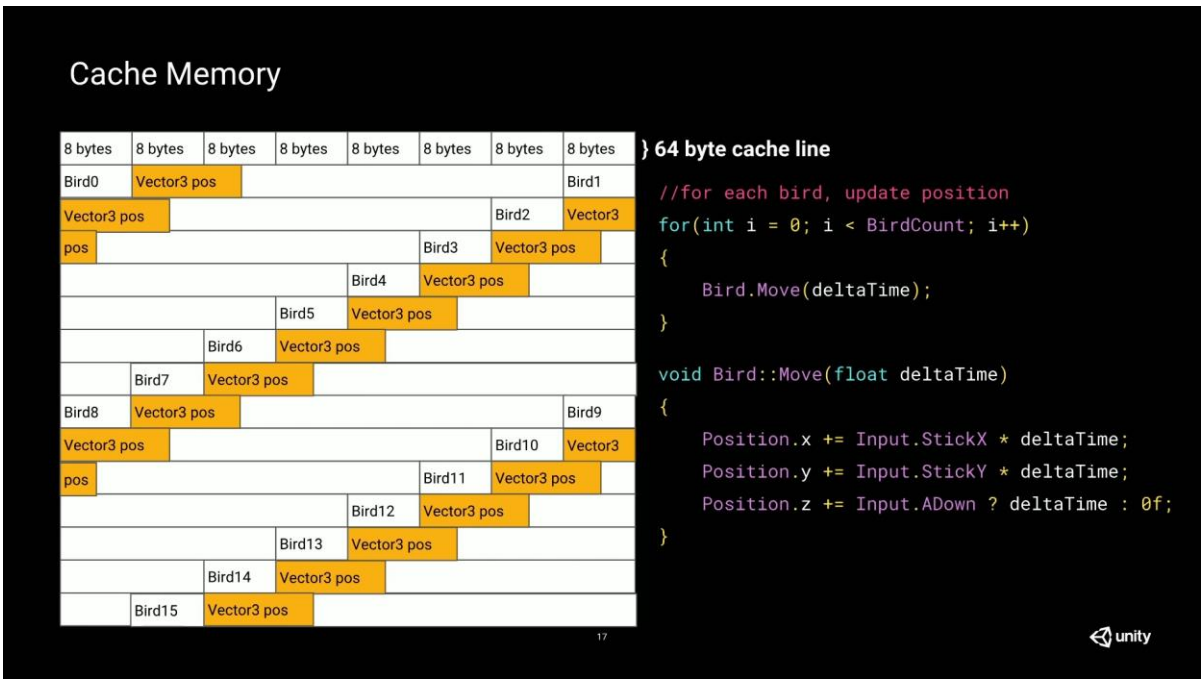
```
class LifeSustain
{
    void OnUpdate()
    {
        Entities.ForEach((ref LifeFunctions lifeFunctions) =>
        {
            lifeFunctions.EnergyPoints = 0;
            bIsAlive = false;
        });
    }
}
```

Programski kod 2.4: Primjer pretraživanja samo entiteta životinja

```
class Moving
{
    void OnUpdate()
    {
        Entities.ForEach((ref LifeFunctions lifeFunctions,
                          in Animal animal) =>
        {
            lifeFunctions.Position = new Vector3(0, 1, 2);
        });
    }
}
```

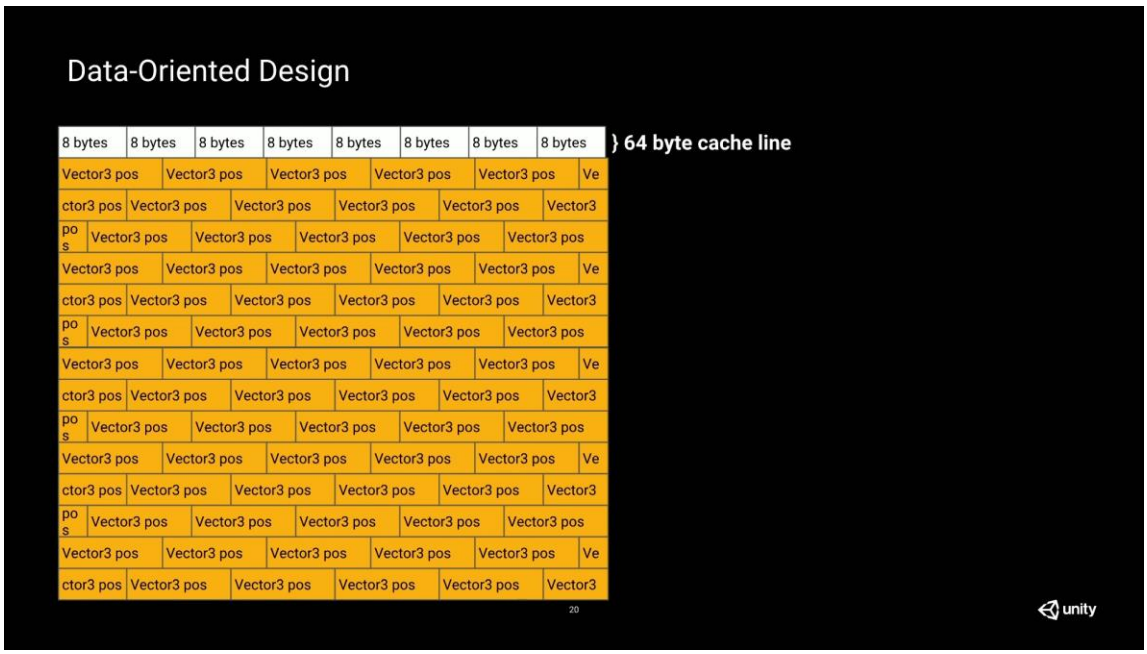
Prethodno navedeni primjeri su Unity implementacija podatkovno orijentiranog dizajna koji je implementiran kroz paket ECS.

Podatkovno orijentirani dizajn nije samo način programiranja već i način iskorištavanja hardvera. Objektno orijentirano programiranje objekte koje koristi sprema unutar procesorske predmemorije (engl. *cache*) za brzo dohvaćanje podataka tog objekta. Procesorska predmemorija je iznimno brza, no također jako ograničena u količini dostupne memorije. Problem s ovim pristupom je da se spremaju podaci koji nisu nužno potrebni te samo zauzimaju ograničenu memoriju. Prednost podatkovno orijentiranog dizajna je da je baziran na principu baze podataka, što omogućuje da se dohvaćaju i koriste samo podaci koji su potrebni. Slika 2.4 prikazuje vizualnu demonstraciju korištenja procesorske predmemorije prilikom objektno orijentiranog programiranja. Za primjer demonstracije korišten je jednostavni program koji mijenja poziciju nekoliko objekata. U navedenom primjeru objekt zauzima 56 bajtova, pri čemu informacija o njegovoj poziciji u svijetu uzima 12 bajtova. Objektno orijentirano programiranje zahtjeva dohvaćanje cijeloga objekta pa tako svaki objekt u primjeru zauzima 44 bajta više nego što je potrebno.



Slika 2.4: Primjer korištenja procesorske predmemorije prilikom korištenja objektno orijentiranog programiranja [8]

To znači da u navedenom primjeru 78.57% procesorske predmemorije stoji neiskorišteno. Podatkovno orijentirani dizajn ne pohranjuje u predmemoriju procesora podatke o cijelom objektu već koristi samo podatke koji su mu potrebni. Slika 2.5 prikazuje vizualnu demonstraciju kako se procesorska predmemorija koristi u podatkovno orijentiranom dizajnu. Dok je primjer sa slike 2.4 uspio učitati 16 objekata u procesorsku predmemoriju, podatkovno orijentirani dizajn je uspio učitati pozicije od 74 objekta igre [8].



Slika 2.5: Primjer korištenja procesorske predmemorije prilikom korištenja podatkovno orijentiranog dizajna [8]

2.5. Entity Component System

Primjena podatkovno orijentiranog dizajna unutar programskog okvira za igre je realizirana kroz sustav entiteta, komponenti i sustava. Podatkovno orijentirani dizajn je osmišljen na temelju baza podataka pa se tako komponente mogu usporediti s tablicama u bazama podataka gdje svaka tablica sadrži redove podataka. Entiteti su objekti igre koji imaju svoje redove unutar komponenti. Sustavi su pozadinske operacije koje upravljaju redovima unutar komponenti te na taj način ujedno kontroliraju i entitete. Svaki objekt unutar DOTS Unity igre je entitet, no svaki entitet nije objekt unutar igre. Entitet također ne sprema nikakve podatke ili komponente. Entitet je samo index koji određuje koji zapis komponente ili komponenti, pripada tom entitetu [8].

Prednost podatkovno orijentiranog dizajna nad objektno orijentiranom programiranju je fragmentacija podataka koja omogućuje uzimanje čim manjeg dijela podataka koji je potreban kako bi se podaci obradili čim brže. Primjer sa slike 2.5 prikazuje učitavanje podatka pozicije objekta unutar svijeta preko varijable *Vector3* što je zapravo struktura sačinjena od 3 varijable *x*, *y* i *z*. Ukoliko se objekt kreće po samo jednoj osi, moguće je dalje fragmentirati i po osima. Kako bi se maksimalno iskoristile prednosti ECS potrebno je fragmentirati podatke u što manje komponente, no zbog čitljivosti koda i lakšeg razvoja igre, potrebno je grupirati podatke u komponente tako da se povezani podaci pohranjuju u iste komponente.

Za razliku od klasičnog pristupa gdje se skripta pridružuje objektu te utječe na taj objekt, sustav DOTS pristupa razvoju igara je drugačiji. Sustavi su metode no odvojene su od podataka i komponente. Mehanika kretanja lika bi u klasičnom pristupu uključivala skripte koja svaku iteraciju (engl. *frame*) igre sluša unose igrača. Ukoliko se detektira unos igrača, mijenja poziciju ili dodaje silu na objekt kojem je skripta dodijeljena. Sustavi su obično fragmentirani u slučaju podatkovno orijentiranog dizajna. Kreira se sustav koji svaku iteraciju prati unos igrača te prema tome mijenja vrijednost podataka unutar komponente koja sadrži igračev unos. Drugi sustav prati podatke unutar komponente unosa igrača te prema tome mijenja poziciju objekata koji su definirani uvjetima. Tim koji je radio na igri *Operation Flashpoint: Dragon Rising* je opisao pokušaje implementacije ECS [9].

2.6. C# Jobs

C# Jobs je DOTS paket koji omogućuje lagano i automatsko kreiranje višedretvenog koda unutar programskog okvira za igre. Višedretvenost zahtjeva praćenje tijeka koda te kontroliranje koji dio koda će se kada izvršiti. Iz toga razloga zahtjeva iskustvo i znanje od programera, no *C# Jobs* paket je sustav kontroliranja izvršavanja višedretvenog koda. *C# Jobs* omogućuje povećanje

performansi igre s manje utrošenog vremena u kontroliranje i dizajniranje koda za višedretvenost. Korisnik kreira poslove koji zatim poslove automatski raspoređuje po dretvama.

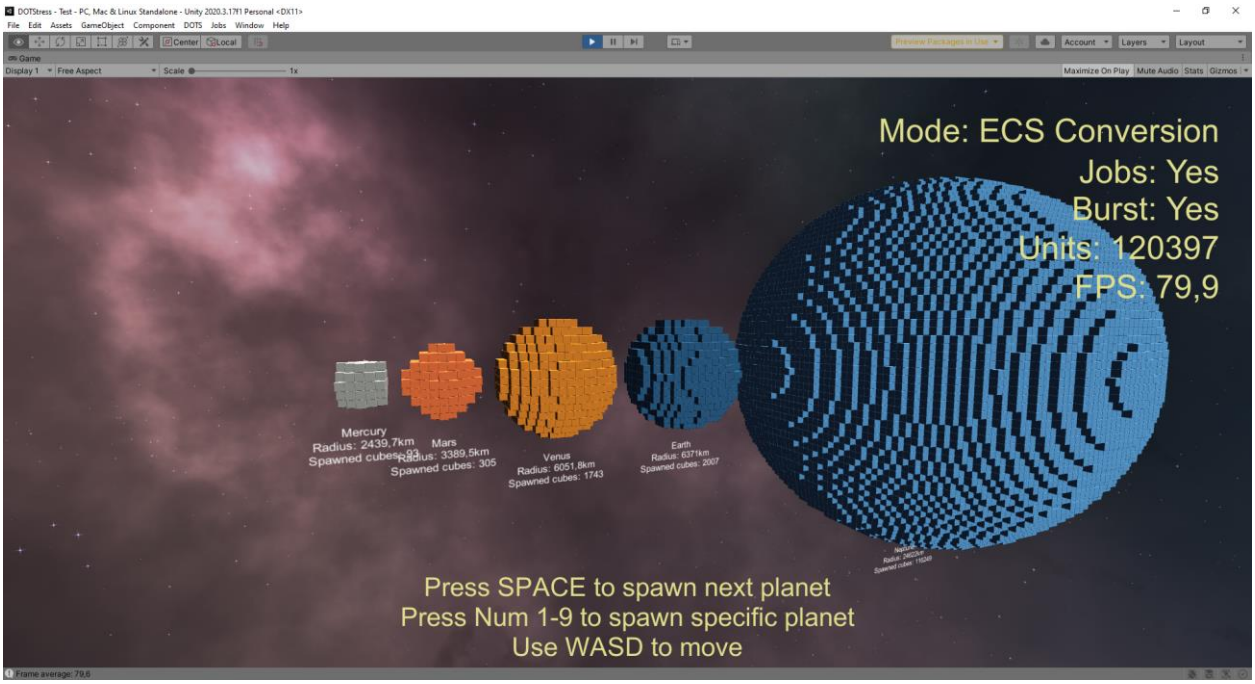
2.7. *Burst* prevoditelj

Burst prevoditelj je zamjena za postojeći Unity prevoditelj. *Burst* je prevoditelj koji je kreiran kako bi maksimalno iskoristio procesore ARM arhitekture, tako da je *Burst* prevoditelj fokusiran na performanse mobilnih uređaja. *Burst* se koristi zajedno s ECS i *C# Jobs* paketima kako bi drastično ubrzao performanse koda koji je zahtjevan za procesor. *Burst* radi tako što visoko performansni C# (engl. *high-performance C#*) kod transformira u LLVM IR posrednički jezik koji koristi LLVM prevoditelj [2].

3. IMPLEMENTACIJA IGRE

3.1. Opis igre

Igra je interaktivna usporedba planeta unutar Sunčevog sustava i Sunca. Veličina planeta je u relativnom omjeru. Smanjenjem jednog planeta, smanjuju se i drugi planeti. Planeti su kreirani od skupa kockica te čine sferu u primarnoj boji planeta. Uključeni planeti su Merkur, Venera, Zemlja, Mars, Jupiter, Saturn, Uran i Neptun te Sunce. Igrač lebdi u svemiru i u mogućnosti se pozicionirati u prostoru. Tipke W, A, S i D omogućuju kretanje dok miš kontrolira smjer kamere i kretanja. Držanjem tipke *shift* igrač se kreće mnogo brže. Pritiskom na tipku razmaknice stvara se sljedeći planet u redoslijedu počevši od Merkura te zatim Mars, Venera, Zemlja, Neptun, Uran, Saturn, Jupiter i Sunce. Igrač može stvoriti i željene planete pomoću numeričke tipkovnice prema spomenutom redoslijedu krećući od broja 1 koji stvara Merkur do broja 9 koji stvara Sunce. Slika 3.1 prikazuje igru tijekom igranja gdje se uspoređuju Merkur, Mars, Venera, Zemlja i Neptun u svojoj relativnoj veličini.



Slika 3.1: Snimka zaslona igre

3.2. Arhitektura igre

3.2.1. Kamera i kretanje

Programskim kodom 3.1 prikazano je kako je realizirano kretanje igrača te kontroliranje rotacije kamere. Dio koda za pozicioniranje u svemiru implementiran je kao objekt igre na klasični Unity način i ne koristi novu DOTS tehnologiju. Metoda *Start* se pokreće kada se objekt kamere

kreira unutar igre te odmah zaključava kursor miša na aplikaciju kako bi igrač mogao bez smetnji okretati se i kontrolirati kameru. Metoda *Update* se poziva svake iteracije te ona poziva druge metode. Točnije, metode za rotaciju i translaciju. Varijabla *mouseSensitivity* kontrolira koliko brzo se kamera okreće, dok *moveSpeed* kontrolira koliko brzo se kamera kreće.

Programski kod 3.1: Inicijalizacija kamere

```
float mouseSensitivity = 3;
float moveSpeed = 20;

private void Start() {
    Cursor.lockState = CursorLockMode.Locked;
}

// Update is called once per frame
void Update() {
    RotateCamera();
    MoveCamera();
}
```

Programski kod 3.2 prikazuje kako je realizirano okretanje kamere. Varijable *mouseX* i *mouseY* sadrže poziciju miša po x i y osi u odnosu na prošli *frame*. Dobivena vrijednost se multiplicira s vrijednošću *mouseSensitivity* kako bi se kontrolirala brzina rotacije kamere. Multiplicirana vrijednost se sprema u varijable *rotAmountX* i *rotAmountY*. Varijable *rotAmountX* i *rotAmountY* se dodaju suprotnim osima kamere dok se x os kamere invertira kako bi iz perspektive igrača pomicanje miša gore rezultiralo pomicanjem kamere prema gore. Slijedi provjera kako rotacija kamere ne bi prešla određene vrijednosti i dozvolila da igrač radi krugove s kamerom. Navedena provjera se radi jer rotiranje po y osi može dezorijentirati igrača pa se postavlja granična vrijednost da igrač ne može rotirati kameru više od 85 stupnjeva prema gore ili dolje. Nulti kut je kada igrač gleda ravno. Nakon provjere granica postavljaju se nove vrijednosti rotacije.

Programski kod 3.2: Rotacija kamere

```
void RotateCamera() {
    float mouseX = Input.GetAxis("Mouse X");
    float mouseY = Input.GetAxis("Mouse Y");

    float rotAmountX = mouseX * mouseSensitivity;
    float rotAmountY = mouseY * mouseSensitivity;

    Vector3 rotCamera = transform.rotation.eulerAngles;

    rotCamera.y += rotAmountX;
    rotCamera.x -= rotAmountY;
    rotCamera.z = 0;

    if (rotCamera.x > 85 && rotCamera.x < 180) {
        rotCamera.x = 85;
```

```
    }
    if (rotCamera.x < 275 && rotCamera.x > 180) {
        rotCamera.x = 275;
    }

    transform.rotation = Quaternion.Euler(rotCamera);
}
```

Programski kod 3.3 prikazuje implementaciju kretnje kamere. Varijabla *speedModifier* je multiplikator brzine te obično iznosi jedan, no držanjem tipke *shift* omogućuje pet puta brže kretanje. Prema igračevom unosu WASD tipki, kamera mijenja poziciju naprijed, nazad, lijevo ili desno. *Transform.forward* i *transform.right* su jedinični vektori za naprijed i desno u Unity koordinatnom sustavu i iznose (0, 0, 1) i (1, 0, 0). U odnosu na unos, jedinični vektor se množi s varijablama *moveSpeed, speedModifier* i *Time.deltaTime*. *Time.deltaTime* predstavlja vrijeme od zadnjeg *frame*. Cijela vrijednost kretanja se množi s *Time.deltaTime* kako bi kretanje bilo konzistentno po sekundi te u ovom slučaju ako *frame* treba više vremena da se procesira, tada će se i igrač pomaknuti za veću vrijednost.

Programski kod 3.3: Kretanje kamere

```
void MoveCamera() {
    //SpeedBoost
    float speedModifier = 1;
    if (Input.GetKey(KeyCode.LeftShift)) {
        speedModifier = 5;
    }

    //Forward/Backwards
    if (Input.GetKey(KeyCode.W)) {
        transform.position += transform.forward * moveSpeed * Time.deltaTime
* speedModifier;
    }
    else if (Input.GetKey(KeyCode.S)) {
        transform.position -= transform.forward * moveSpeed *
Time.deltaTime * speedModifier;
    }

    //Left/Right
    if (Input.GetKey(KeyCode.D)) {
        transform.position += transform.right * moveSpeed * Time.deltaTime *
speedModifier;
    }
    else if (Input.GetKey(KeyCode.A)) {
        transform.position -= transform.right * moveSpeed * Time.deltaTime *
speedModifier;
    }
}
```

3.2.2. Skripta *GameManager*

Inicijalizacija igre prilikom pokretanja se kontrolira preko *GameManager* skripte. *GameManager* je također kreiran bez Unity DOTS tehnologije pa je skripta vezana uz objekt *GameManager*. Programski kod 3.4 prikazuje *GameManager* varijable. *Instance* varijabla omogućuje *singleton* arhitekturu. *Singleton* je tip statičke varijable koja limitira da uvijek postoji samo jedan primjerak *GameManager* objekta sa skriptom. Također omogućuje da *GameManager* skripta uvijek bude dostupana za referenciranje drugim skriptama. Varijable *ui* i *spawner* su reference na skripte koje kontroliraju prikaz korisničkog sučelja i stvaranja novih objekata unutar igre. Varijabla *moveSpeed* kontrolira brzinu kretanja kocki. Varijable *useJobs* i *useBurst* kontroliraju da li će se igra pokrenuti korištenjem *C# Jobs* i s *Burst* prevoditeljom. Varijabla *mode* je ograničeni korisnički određeni skup vrijednosti gdje se određuje da li se koristi obični Unity pristup, čisti ECS (engl. *pure ECS*) ili ECS konverzija Unity objekata igre (engl. *ECS conversion*).

Programski kod 3.4: GameManager varijable

```
//Singleton
public static GameManager Instance;

//References
public UI ui;
public Spawner spawner;

//Stats
float moveSpeed = 0.001f;

//Classic vs ECS
[Header("Settings")]
[SerializeField] public DemoMode mode;
[SerializeField] public bool useJobs;
[SerializeField] public bool useBurst;
```

Programski kod 3.5 prikazuje kako se prilikom pokretanja inicijalizira igra. Programski kod definira u kojem režimu rada će se igra izvršavati. Uključuje potrebne sustave i isključuje nepotrebne. Metoda *Start* se pokreće odmah pri pokretanju igre te ona pokreće *SetSystemMode* metodu. *SetSystemMode* metoda prvo ažurira korisničko sučelje da prikaže u kojem režimu rada se igra pokrenula. Nakon korisničkog sučelja, *SetSystemMode* zatim isključi sve DOTS sustave iz igre kako ne bi bili aktivni i nakon toga ulazi u provjeru koji režim rada je odabran. Ako je postavljen klasični Unity tada se ne uključuju sustavi, u slučaju *Pure ECS* ili *ECS Conversion* uključuju se određeni sustavi. Ako nisu odabrani niti *C# Jobs* niti *Burst* prevoditelj onda se uključuje sustav *MovementSystem*, u slučaju da su odabrani *C# Jobs* dok *Burst* prevoditelj nije onda se uključuje *MovementSystemJobs* a ako su uključeni i *C# Jobs* i *Burst* prevoditelj onda se uključuje *MovementSystemJobsBurst*.

Programski kod 3.5:Uključivanje potrebnih sustava

```
private void SetSystemMode() {
   ui.SetMode(useJobs, useBurst);


World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystem>().
Enabled = false;

World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystemJobs
>().Enabled = false;

World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystemJobs
Burst>().Enabled = false;

   if (mode != DemoMode.Classic) {
      if (!useJobs) {

World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystem>().
Enabled = true;
         Debug.Log("Enabling MovementSystem");
      }
      else if (useBurst) {

World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystemJobs
Burst>().Enabled = true;
         Debug.Log("Enabling MovementSystemJobsBurst");
      }
      else if (!useBurst) {

World.DefaultGameObjectInjectionWorld.GetOrCreateSystem<MovementSystemJobs
>().Enabled = true;
         Debug.Log("Enabling MovementSystemJobs");
      }

   }
}
```

Programski kod 3.6 prikazuje jednostavnu kontrolu unosa igrača gdje se provjerava koji planet će se stvoriti. Kao što je vidljivo iz programskog koda prilikom pritiska razmaknice stvara se sljedeći planet u redoslijedu. Unosom broja s numeričke tipkovnice stvara se planet povezan s tim brojem.

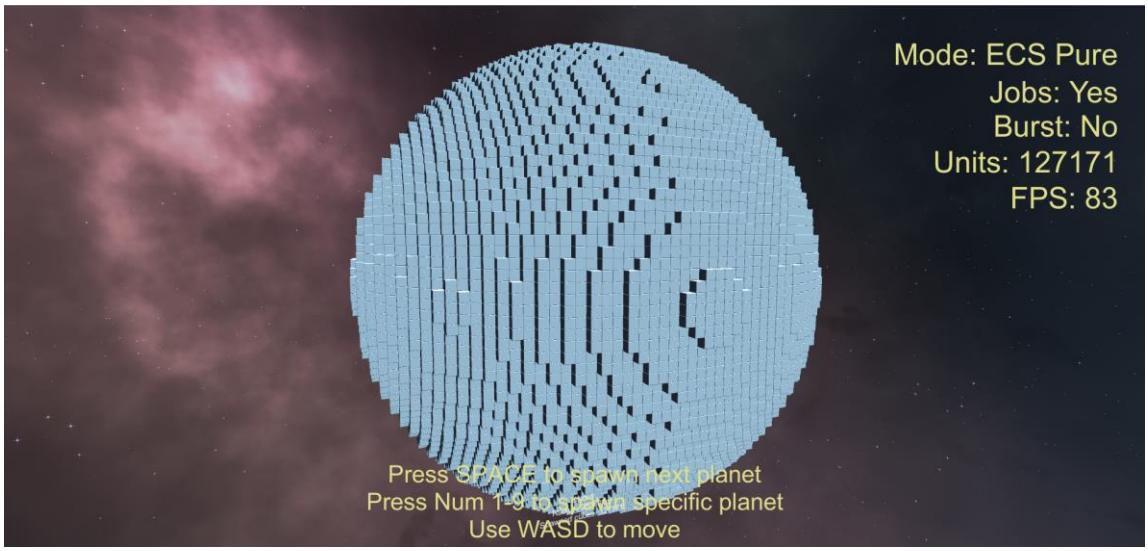
Programski kod 3.6:Kontrola kreiranja planeta

```
private void Update() {
   if (Input.GetKeyDown(KeyCode.Space)) {
      spawner.SpawnUnit(-1);
   }
   else if (Input.GetKeyDown(KeyCode.Keypad1)) {
      spawner.SpawnUnit(0);
   }
   else if (Input.GetKeyDown(KeyCode.Keypad2)) {
      spawner.SpawnUnit(1);
   }
   else if (Input.GetKeyDown(KeyCode.Keypad3)) {
```

```
            spawner.SpawnUnit(2);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad4)) {
            spawner.SpawnUnit(3);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad5)) {
            spawner.SpawnUnit(4);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad6)) {
            spawner.SpawnUnit(5);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad7)) {
            spawner.SpawnUnit(6);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad8)) {
            spawner.SpawnUnit(7);
        }
        else if (Input.GetKeyDown(KeyCode.Keypad9)) {
            spawner.SpawnUnit(8);
        }
    }
```

3.2.3. Korisničko sučelje

Slika 3.2 prikazuje korisničko sučelje igre. Iz slike je vidljivo da se na dnu zaslona u sredini nalaze upute za igru. U gornjem desnom kutu se nalaze informacije o trenutnom režimu rada: da li je uključen sustav s *C# Jobs*, da li je uključen sustav s *Burst* prevoditeljom, koliko je trenutno kreiranih kockica te u koliko sličica po sekundi (engl. *frames per second*, skraćeno FPS) se trenutno igra izvodi.



Slika 3.2: Korisničko sučelje igre

Programski kod 3.7 prikazuje računanje FPS statistike. Za što preciznije FPS podatke uzima se vremenski period koji je potreban za određeni broj iteracija igre koji je definiran varijablom *frameSample*. za testiranje se uzima broj od zadnjih 50 iteracija igre. FPS je definiran varijablom

fps, prvo se podijeli vrijeme potrebno za posljednjih 50 iteracija što je označeno varijablom *frameTimes* s brojem iteracija da bi se dobilo prosječno vrijeme za svaku iteraciju. Zatim se dijeli broj 1 s dobivenim rezultatom tako da se dobije koliki je FPS. Rezultat se množi s 10, pretvara se u cijeli broj gdje se ostatak decimalnog broja odbacuje te se zatim rezultat dijeli s 10 i tako se dobiva prosječni FPS za zadnjih 50 iteracija u obliku decimalnog broja s jednom decimalom. Dalje u kodu se taj broj ispisuje na korisničko sučelje i u konzolu kako bi se podaci analizirali.

Programski kod 3.7:Računanje FPS

```
float fps = frameTimes / frameSample;
fps = 1f / fps;
fps *= 10f;
fps = (int)fps;
fps /= 10f;
FPS.text = "FPS: " + fps;
Debug.Log("Frame average: " + fps);
FPSLog.text = fps + "\n" + FPSLog.text;
```

3.2.4. Kreiranje kocki i planeta

Spawner je zadužen za stvaranje kockica na određenim lokacijama, pravim količinama i tipovima ovisno o trenutnom režimu rada. Programski kod 3.8 prikazuje varijable koje se koriste za kreiranje spomenutih kockica koje sačinjavaju planete. *TotalSpawned* varijabla sprema koliko se kockica ukupno stvorilo, *spawnCounter* varijabla je brojač koji određuje koji je sljedeći planet koji se treba stvoriti pritiskom tipke razmaknice. *MercuryRadius* varijabla određuje koliki je radijus Merkura u simulaciji, to je vrlo bitno jer radijus svakog planeta se određuje kao multiplikator Merkurovog radijusa, na primjer Zemljin radijus je 2.61 puta veći od Merkurovog. *RadiusScale* je niz relativnih veličina planeta naspram Merkura. *RealRadius* je zapis stvarnog radijusa planeta u kilometrima i koristi se za ispisivanje informacija o planetima u tekstu ispod planeta. *PlanetName* je niz imena planeta koji se također koristi za ispisivanje informacija o planetima u tekstu ispod planeta. *PlanetMaterial* je niz varijabli materijala koji se koristi kako bi svaki planet imao svoju unikatnu boju koja je bazirana na primarnoj boji svakog planeta. *Offset* varijabla se koristi da igra zna koliko se mora odmaknuti od pozicije (0,0,0) po x osi kako se novo stvoreni planet ne bi stvorio unutar drugog planeta.

ECSPrefab i *classicPrefab* su varijable koje spremaju objekt kocke kako bi igra znala koju kocku stvoriti. *CubeMesh* sprema trodimenzionalni objekt kocke i zajedno s varijablom *cubeMaterial* moguće je stvoriti kocku bez referenciranja već stvorenog objekta koji se klonira kao s *ECSPrefab* i *classicPrefab*. Ovaj način se koristi kod *Pure ECS* režima rada. *TextDescription*

je varijabla koja sprema objekt teksta koji se stvara ispod svakog planeta kako bi dao informacije o imenu planeta, stvarnom radijusu u kilometrima i broju kocki stvorenih kreiranjem toga planeta.

DefaultWorld, *entityManager*, *archetype* i *unitEntityPrefab* su varijable specifične za DOTS tehnologiju. *DefaultWorld* varijabla se referencira na svijet igre koji sadrži entitete i sustave [10]. *EntityManager* je kontroler koji je dio ECS paketa i kontrolira entitete, *entityManager* se referencira na zadani *EntityManager* unutar svijeta [11]. *Archetype* je pomoćna varijabla koja grupira zadane komponente [12]. U projektu *archetype* se koristi kako bi se spremio plan za kreiranje kocke u *Pure ECS* režimu rada. *UnitEntityPrefab* je varijabla koja sprema entitet verziju objekta igre kocke.

Programski kod 3.8:Varijable kreatora

```
//Stats
private int totalSpawned = 0;   //Total number of cubes spawned so far
private int spawnCounter = 0;   //Which planet needs to be spawned
private float mercuryRadius = 3f;   //Base radius of smallest planet
private float[] radiusScale =
{ 1f, 1.39f, 2.48f, 2.61f, 10.09f, 10.4f, 23.87f, 28.66f, 285.42f};
//Relative size of planets to base radius
private float[] realRadius =
{ 2439.7f, 3389.5f, 6051.8f, 6371f, 24622f, 25362f, 58232f, 69911f,
969340f };   //Real radius of planets in km
private string[] planetName =
{ "Mercury", "Mars", "Venus", "Earth", "Neptune", "Uranus", "Saturn",
"Jupiter", "Sun" };   //Name of planets
[SerializeField] private Material[] planetMaterial;
// Mercury, Mars, Venus, Earth, Neptune, Uranus, Saturn, Jupiter, Sun

private float offset = 0;   //Offset, how much next planet needs to be
offset to not overlap with old (x plane)

//Prefabs
[SerializeField] private GameObject ECSPrefab;
[SerializeField] private GameObject classicPrefab;
[SerializeField] private Mesh cubeMesh;
[SerializeField] private Material cubeMaterial;
[SerializeField] private GameObject textDescription;

//Variables
private World defaultWorld;
private EntityManager entityManager;
private EntityArchetype archetype;
private Entity unitEntityPrefab;
```

Programski kod 3.9 prikazuje metodu *Start* koja se poziva prilikom pokretanju igre i može pozvati metode *SetupECSConversion* i *SetupECSPure*. Ovisno o režimu rada se zatim poziva *SpawnUnit* metoda koja je zadužena za određivanje koja kocka će se kreirati. *SetupECSPure* metoda referencira svijet u *defaultWorld* varijablu i *EntityManager* u *entityManager* varijablu. Ovdje se kreira *archetype* varijabla i spremaju se komponente koje će kocka sadržavati kada se

kreira preko koda u *Pure ECS* režimu rada. *SetupECSConversion* metoda osim referenciranja svijeta i *EntityManagera* također sprema referencu na entitet koji kreira od objekta igre koji je spremljen u varijabli *ECSPrefab*, *unitEntityPrefab* se kasnije koristi kao referenca za stvaranje kocki u *ECS Conversion* načinu rada.

Programski kod 3.9:Inicijaliziranje kreatora

```
void Start() {
    if (GameManager.Instance.mode == DemoMode.ECSConversion) {
        SetupECSConversion();
    }

    if (GameManager.Instance.mode == DemoMode.ECSPure) {
        SetupECSPure();
    }

    SpawnUnit(-2);
}

private void SetupECSPure() {
    if (GameManager.Instance.mode == DemoMode.ECSPure) {
        defaultWorld = World.DefaultGameObjectInjectionWorld;
        entityManager = defaultWorld.EntityManager;
        archetype = entityManager.CreateArchetype
        (
            typeof(Translation),
            typeof(Rotation),
            typeof(RenderMesh),
            typeof(LocalToWorld),
            typeof(CubeData),
            typeof(Scale),
            typeof(RenderBounds)
        );
    }
}

private void SetupECSConversion() {
    if (GameManager.Instance.mode == DemoMode.ECSConversion) {
        defaultWorld = World.DefaultGameObjectInjectionWorld;
        entityManager = defaultWorld.EntityManager;
        GameObjectConversionSettings settings =
GameObjectConversionSettings.FromWorld(defaultWorld, null);
        unitEntityPrefab =
GameObjectConversionUtility.ConvertGameObjectHierarchy(ECSPrefab,
settings);
    }
}
```

Programski kod 3.10 prikazuje stvaranje planeta pomoću kocki u klasičnom režimu rada. Varijabla *radius* određuje koliki će biti radijus stvorenog planeta. *RadiusInt* je cjelobrojni rezultat računanja radijusa dok *spawnedNow* je brojač koji bilježi koliko se kocki stvorilo prilikom stvaranja planeta. *Offset* se računa tako da se doda 120% radijusa planeta tako da je planet odmaknut od drugog relativno prema svojoj veličini umjesto fiksno određenog razmaka. *LowestY* bilježi najmanju y poziciju kocke unutar planeta kako bi se znala odrediti visina planeta, koristi se

za određivanja gdje stvoriti tekst ispod planeta. Kreiranje planeta je implementirano pomoću 3 ugniježđene for petlje. Provjerava se udaljenost svake kocke od središta. Ako je radijus planeta manji od udaljenosti kocke od središta onda se kocka ne stvara. Stvorene kocke čine kuglu. Ako se odredi da se kocka treba stvoriti onda se dodaje u brojač stvorenih kocki, dodjeljuje joj se materijal planeta, pomiče se po x osi određeno prema *offset* varijabli i gleda se da li je to najniža točka planete. Nakon stvaranja svih kocki stvara se objekt tekstualnih informacija o planetu. Na kraju se broj stvorenih kocki dodaje u ukupan zbroj i to se prikazuje na korisničkom sučelju.

Programski kod 3.10:Stvaranje planeta pomoću kocki

```
//Prepare
Vector3 center = Vector3.zero;

int planetIndex = spawnCounter;
if (index != -1) {
   planetIndex = index;
}

float radius = mercuryRadius * radiusScale[planetIndex];
int radiusInt = (int)radius + 1;
int spawnedNow = 0;

//To not overlap
offset += radius * 1.2f;
float lowestY = 0;   //Stores lowest y position of cube, so text goes
underneath

for (int i = -radiusInt; i <= radiusInt; i++) {
   for (int j = -radiusInt; j <= radiusInt; j++) {
      for (int k = -radiusInt; k <= radiusInt; k++) {
         Vector3 position = new Vector3(i, j, k);
         if (Vector3.Distance(center, position) < radius) {
            spawnedNow++;
            GameObject instance = Instantiate(classicPrefab);
            instance.GetComponent<MeshRenderer>().material =
planetMaterial[planetIndex];
            Vector3 offsetPosition = new Vector3(position.x + offset,
position.y, position.z);
            instance.transform.position = offsetPosition;

            if(position.y < lowestY) {
               lowestY = position.y;
            }
         }
      }
   }
}

//Planet info
GameObject info = Instantiate(textDescription);
Vector3 textPosition = new Vector3(0f + offset, lowestY - 5f, 0f);
info.transform.position = textPosition;
info.GetComponent<TextMesh>().text = planetName[planetIndex] + "\nRadius:
" + realRadius[planetIndex] + "km\nSpawned cubes: " + spawnedNow;

if(index == -1) {
   spawnCounter++;
```

```
}
offset += radius * 1.2f;
totalSpawned += spawnedNow;
GameManager.Instance.ui.SetTotal(totalSpawned);
```

Programski kod 3.11 prikazuje kreiranje planeta koristeći DOTS tehnologiju uz *Pure ECS* i *ECS Conversion* režim rada. Kreiranje planeta koristeći DOTS tehnologiju je identično kao i u klasičnom Unity načinu. Jedina razlika je koji objekti se stvaraju dok je proces stvaranja identičan. Klasični Unity, *Pure ECS* i *ECS Conversion* svaki ima svoje objekte koje stvara. Kada se odredi da je kocka unutar radijusa, određuje se pomak po x osi kako se ne bi preklapalo s drugim planetima. Nakon pomaka kreira se entitet, u *Pure ECS* režimu rada se kreira prema varijabli *archetype* dok se u *ECS Conversion* kreira iz varijable *unitEntityPrefab* koja sprema konvertirani Unity objekt igre u entitet. Nakon kreiranja entiteta, mijenja mu se pozicija u svijetu i postavlja se na skalu veličine 1 kako bi objekt bio pravilne veličine. Entitetu se dodaje i trodimenzionalni oblik te materijal kako bi planet dobio primarnu boju stvarnoga planeta. Na kraju se entitetu dodaje komponenta *CubeData* kako bi sustavi znali pronaći sve kocke planeta.

Programski kod 3.11:DOTS kreiranje kocki

```
if (Vector3.Distance(center, position) < radius) {
    spawnedNow++;
    float3 offsetPosition = new float3(position.x + offset, position.y,
position.z);

    Entity myEntity = entityManager.CreateEntity(archetype);

    entityManager.AddComponentData(myEntity, new Translation { Value =
offsetPosition });
    entityManager.AddComponentData(myEntity, new Scale { Value = 1 });

    entityManager.AddSharedComponentData(myEntity, new RenderMesh {
        mesh = cubeMesh,
        material = planetMaterial[planetIndex]
    });

    entityManager.AddComponentData(myEntity, new CubeData { speed =
GameManager.Instance.GetMoveSpeed() });

    if (position.y < lowestY)
    {
        lowestY = position.y;
    }
}
```

3.2.5. Upravljanje kockom

Programski kod 3.12 prikazuje skriptu koja je pridružena kocki. Programski kod prikazuje kretanje po x osi tako da se svaka kocka kreće zasebno klasičnim Unity načinom. Iako se kocke

kreću identično, svaka ima samostalnu logiku pomicanja. Programski kod 3.12 uzima trenutnu poziciju kocke, izračunava koliko bi se kocka trebala pomaknuti u vremenu od zadnje iteracije prema brzini definiranoj varijablom *moveSpeed* unutar skripte *GameManager*, udaljenost se dodaje poziciji kocke po x osi te se nova vrijednost pozicije postavlja kao nova pozicija kocke.

Programski kod 3.12:Logika kretanja kocke klasičnim Unity načinom

```
public class CubeMove : MonoBehaviour {
   void Update()
   {
      Vector3 pos = transform.position;
      pos += transform.right * GameManager.Instance.GetMoveSpeed() *
Time.deltaTime;

      transform.position = pos;
   }
}
```

Programski kod 3.12 prikazuje kako je to odrađeno na klasični Unity način s objektima igre. DOTS način je malo drugačiji te u programskom kodu 3.13 se vidi komponenta koja se pridružuje entitetu. Logika kocke je vrlo jednostavna te od podataka treba samo varijablu *speed*, no to je dovoljno da se prikaže podatkovno orijentirani dizajn.

Programski kod 3.13:Komponenta entiteta kocke

```
[GenerateAuthoringComponent]
public struct CubeData : IComponentData {
    public float speed;
}
```

Programski kod 3.14 prikazuje sustave koji pomiču kocke. Logika kretanja kocki je identična logici kretanja klasičnom Unity načinu. Programski kod 3.14 prikazuje tri zasebna sustava koji obavljaju istu stvar, no na drugačiji način. Prvi sustav je *MovementSystem* bez *C# Jobs* i *Burst* prevoditelja. Drugi sustav je *MovementSystemJobs* s *C# Jobs* i bez *Burst* prevoditelja. Treći i zadnji sustav je *MovementSystemJobsBurst* s *C# Jobs* i s *Burst* prevoditeljom. *MovementSystem* sustav kao i druga dva sustava, pomiče sve entitete kocki po x osi kao i klasični Unity način, samo mijenja poziciju entiteta u svijetu. *MovementSystem* pretražuje sve entitete koji sadrže *Translation* komponentu koju sadrže svi objekti i *CubeData* komponentu koju sadrže samo kocke stvorene na DOTS način te nad njima obavlja procese. *MovementSystemJobs* obavlja istu logiku, no ključne naredbe su *WithoutBurst* koja isključuje korištenje *Burst* prevoditelja koji se koristi u trećem sustavu i *Schedule* komanda koja stvara *JobHandle* za logiku. *OnUpdate* metoda koja se izvršava svaku iteraciju, vraća *JobHandle* objekt što omogućuje da pomoću *C# Jobs* paketa Unity iskoristi

višedretvenost DOTS tehnologije. *MovementSystemJobsBurst* sustav je identičan *MovementSystemJobs* sustavu, no s korištenjem *Burst* prevoditelja.

Programski kod 3.14:DOTS sustavi

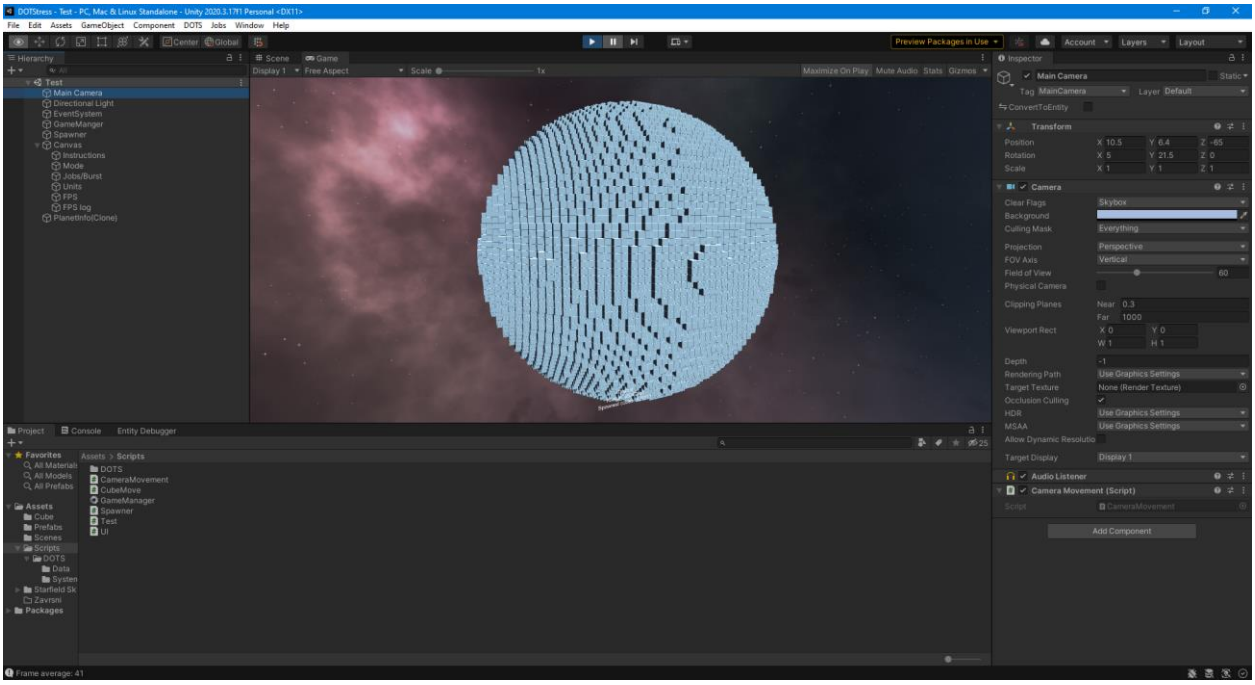
```
public class MovementSystem : ComponentSystem {
    protected override void OnUpdate()
    {
        Entities.ForEach((ref Translation trans, ref CubeData moveForward)
=>
        {
            float deltaTime = Time.DeltaTime;

            trans.Value += new float3(moveForward.speed * deltaTime, 0f, 0f);
        });
    }
}


public class MovementSystemJobs : JobComponentSystem {
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        float deltaTime = Time.DeltaTime;

        JobHandle jobHandle = Entities.
            WithoutBurst().
            ForEach((ref Translation trans, ref CubeData moveForward) =>
            {
                trans.Value += new float3(moveForward.speed * deltaTime, 0f,
0f);
            }).Schedule(inputDeps);

        return jobHandle;
    }
}

public class MovementSystemJobsBurst : JobComponentSystem {
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        float deltaTime = Time.DeltaTime;

        JobHandle jobHandle = Entities.
            ForEach((ref Translation trans, ref CubeData moveForward) =>
            {
                trans.Value += new float3(moveForward.speed * deltaTime, 0f,
0f);
            }).Schedule(inputDeps);

        return jobHandle;
    }
}
```

4. TESTIRANJE

4.1. Postavke

Iako je u pitanju igra, testiranje je izvršeno u kontroliranim uvjetima gdje su neke funkcionalnosti onemogućene kako bi pri svakom testu bili što sličniji uvijeti. Kontrolirani uvjeti omogućuju precizniji uvid u rezultate testiranja i točnije zaključke. Testiranje je izvršeno u obliku stres testa gdje je cilj opteretiti sustav do granice te bilježiti ponašanje igre.

Svako testiranje je izvršeno stvaranjem Urana i samo Urana koji je stvoren svaki put na istoj poziciji u svijetu. Bazni radijus Merkura koji je korišten kao referenca za skaliranje drugih planeta je 3. S baznim radijusom 3 prilikom stvaranja Urana stvoreno je 127171 kocki koje su sačinjavale planet. Kontrole za upravljanje kamerom su tijekom testiranja bile isključene tako da pomicanje kamere nije bilo moguće. Pozicija i rotacija kamere je vrlo bitna za testiranje jer ako objekti nisu unutar kadra kamere onda se niti ne iscrtavaju [13] što može drastično utjecati na performanse. Kamera je postavljena tako da bi u kadru kamere bio vidljiv cijeli planet Uran sa svim kockicama kao i informativni tekst ispod planeta. Slika 4.1 prikazuje kadar kamere tijekom testiranja.



Slika 4.1: Korisničko sučelje testiranja

Testiranje je izvršeno izvođenjem preko samostalnog izgrađenog pokretača igre (engl. *standalone build*). Testiranje je obavljeno s kockama koje se kreću i kockama koje se ne kreću kako bi se usporedilo koliko utjecaja bi jednostavna logika utjecala na performanse igre.

4.2. Hardver

Testiranje je provedeno na tri različita uređaja kako bi se u obzir uzelo više različitih hardvera. Prvo računalo je ASUS laptop s procesorom Intel i7-6700HQ s 2.60GHz frekvencijom, 16GB radne memorije i nVidia GeForce GTX950M grafičkom karticom s 4GB video memorije s nasumičnim pristupom (engl. *video random access memory,* skraćeno *VRAM*). Drugo računalo je stolno računalo s AMD Ryzen 5 3600 s 3.60GHz procesorom, 16GB radne memorije i nVidia GeForce RTX 3060ti grafičkom karticom od 8GB VRAM. Treće računalo je ACER laptop s AMD Ryzen 5 3500U procesorom frekvencije 2.10GHz i s integriranom grafičkom karticom Radeon Vega Mobile 8 i 8GB radne memorije. Hardverom se može zaključiti da stolno računalo ima najviše snage dok ACER laptop ima najmanje snage.

4.3. Plan testiranja

Testiranje se izvodi s sedam različitih postavki igre. Prvo pokretanje u svakom režimu rada. **Test 1** je klasični Unity način s objektima igre i arhitekturom objektno orijentiranog programiranja. **Test 2** je u *Pure ECS* režimu rada bez *C# Jobs* i bez *Burst* prevoditelja. **Test 3** je s *C# Jobs* i bez *Burst* prevoditelja. **Test 4** je s *C# Jobs* i s *Burst* prevoditeljom. **Test 5** je *ECS Conversion* bez *C# Jobs* i bez *Burst* prevoditelja. **Test 6** je s *C# Jobs* i bez *Burst* prevoditelja. **Test 7** je s *C# Jobs* i s *Burst* prevoditeljom. Svaki test se izvodi na svakom računalu i mjeri se FPS, veći FPS znači bolje performanse igre ili računala. Kako bi se dobili što točniji rezultati, FPS je rezultat srednje vrijednosti od zadnjih pet uzoraka testiranja. Svaki FPS uzorak se računa kao prosječni FPS zadnjih 50 iteracija, za testiranje to znači da FPS unutar tablica je prosječni FPS za posljednjih 250 iteracija.

4.4. Rezultati testiranja

4.4.1. Laptop 1

Tablica 4.1: Rezultati testiranja laptop 1 računala

Kretanje kocki		Da	Ne
	Test	FPS	FPS
Klasični Unity	Test 1	4,5	5,8
ECS Pure	Test 2	29,9	37,3
ECS Pure	Test 3	44,9	45,8
ECS Pure	Test 4	44,6	46
ECS Conversion	Test 5	29,2	35,1
ECS Conversion	Test 6	44,7	45
ECS Conversion	Test 7	44,9	45

4.4.2. Stolno računalo

Tablica 4.2: Rezultati testiranja stolnog računala

Kretanje kocki		Da	Ne
	Test	FPS	FPS
Klasični Unity	Test 1	8,7	9,5
ECS Pure	Test 2	48,5	71,1
ECS Pure	Test 3	82,2	81,8
ECS Pure	Test 4	80,9	82,2
ECS Conversion	Test 5	45,9	67,1
ECS Conversion	Test 6	79,9	77,3
ECS Conversion	Test 7	77,7	79,7

4.4.3. Laptop 2

Tablica 4.3: Rezultati testiranja laptop 2 računala

Kretanje kocki		Da	Ne
	Test	FPS	FPS
Klasični Unity	Test 1	4,4	4,9
ECS Pure	Test 2	28,3	34,7
ECS Pure	Test 3	45,5	46,6
ECS Pure	Test 4	45,8	46,4
ECS Conversion	Test 5	27,4	34,1
ECS Conversion	Test 6	44,4	45,2
ECS Conversion	Test 7	45,1	45,6

4.4.4. Zaključak testiranja

Nakon testiranja jasno su vidljive razlike u performansama. Tablica 4.1 prikazuje rezultate laptop 1 računala, tablica 4.2 prikazuje rezultate stolnog računala te tablica 4.3 prikazuje rezultate laptop 2 računala. Prvi red opisuje da li su kocke sadržavale logiku kretanja prilikom testiranja ili ne. Prvi i drugi stupac opisuju u kojem režimu rada je test odrađen.

Igre na konzolama su često zaključane na 30 FPS ili 60 FPS, obično u igrama fokusiranim na mrežno igranje [14]. Igre na stolnim računalima vrlo rijetko imaju zaključan FPS. Iako FPS nije zaključan, limit je postavljen hardverom to jest monitorom. Prema izvoru [15] moderni monitori su u mogućnosti prikazati i FPS od 360. Filmovi imaju nešto sporiji FPS jer su mediji koji se samo gleda bez interakcije te je bitno da oko ne primjeti zasebne sličice. FPS u filmovima je obično 24 prema izvoru [16]. To znači da se od igara očekuje da se izvode u barem 30 FPS kako ne bi umarale igrača.

Gledajući rezultate s kretanjem kocki, prema tablicama je vidljivo kako klasični Unity ili test 1, postiže jako niske FPS rezultate. FPS rezultati laptop 1 i laptop 2 računala su 4,5 i 4,4 FPS. Iako stolno računalo ima skoro duplo bolji rezultat, rezultat je dosta ispod praga od 30 FPS što cijelu igru čini neigrivom.

Test 2 koristi *Pure ECS* režim rada. Koristi DOTS tehnologiju no bez višedretvenosti i bez *Burst* prevoditelja. Rezultati su znatno bolji od klasičnog Unity načina. Test 2 pokazuje 564,44% bolje performanse za laptop 1, 543,18% bolje performanse za laptop 2 i 457,47% bolje

performanse na stolnom računalu. To je ogromno povećanje performansi samo prelaskom na podatkovno orijentirani dizajn, bez višedretvenosti i bez *Burst* prevoditelja. Laptop 2 u testu 2 postiže FPS od 28,3 što je jako blizu 30 FPS za ugodno igranje. Laptop 1 postiže FPS od 29,9 što je skoro 30 FPS i omogućuje ugodno igranje. Test 5 je sličan testu 2 no u *ECS Conversion* režimu rada i postiže neznatno slabije rezultate na svakom računalu.

Test 3 uključuje *C# Jobs* sustav za *Pure ECS* režim rada što znači da igra koristi višedretvenost. Uključivanjem višedretvenosti postiže se još veće povećanje performansi od 69,48% na stolnom računalu naspram testa 2 ili povećanje od 844,83% naspram testa 1. Laptop 1 bilježi povećanje od 50,17% i laptop 2 računalo povećanje od 60,78%. Test 6 je identični test u *ECS Conversion* režimu rada te kao test 2 i test 5, postiže neznatno slabije rezultate.

Test 4 uključuje i *Burst* prevoditelj pored *C# Jobs* sustava za *Pure ECS* režim rada. *Burst* prevoditelj ne donosi povećanje performansi, zapravo postiže slabije rezultate naspram testa 3. Laptop 1 bilježi smanjenje performansi od 0,67%, stolno računalo smanjenje od 1,58% dok laptop 2 postiže zapravo povećanje performansi od 0,66%. Razlike koje variraju su neznatne na svim računalima. *Burst* prevoditelj je optimiziran za ARM procesore što može biti razlog neznatnih promjena u performansama jer navedena računala ne koriste ARM procesore. Test 7 za *ECS Conversion* režim rada također bilježi neznatne promjene u FPS rezultatima naspram testa 6.

Usporedbom testova 2, 3 i 4 naspram testova 5, 6 i 7, *ECS Conversion* režim rada konstantno postiže slabije rezultate naspram *Pure ECS* režima rada. Iako su razlike neznatne, između 4% i 0 %, *ECS Conversion* konstantno postiže slabije rezultate.

Isključivanjem kretanja kocki, klasični Unity postiže porast performansi od 28,88% na laptopu 1, 9.2% porast na stolnom računalu i 11,36% na laptopu 2. Iako su performanse porasle, FPS je svejedno daleko ispod granice igrivog na 30 FPS.

Test 2 bilježi značajni rast performansi te omogućava igrivost. Laptop 1 bilježi rast od 543,1%, stolno računalo rast od 648,42% i laptop 2 bilježi rast performansi od 608,16%. Rast je veći nego s pokretnim kockama no ne mnogo veći rast. Test 5 postiže slične rezultate no neznatno manji FPS naspram testa 2.

Test 3 postiže nešto manji porast nego što je to bio slučaj s pokretnim kockama. Stolno računalo bilježi rast od 15.05% što je dosta manji rast nego rast od 69,48% s pokretnim kockama. laptop 1 bilježi rast od 22,79% naspram 50,17% i laptop 2 rast od 34,29% naspram 60,78%. No FPS razlika između pokretnih i nepokretnih kocki za test 3 je neznatna. Test 6 pokazuje slične rezultate no neznatno manji FPS nego test 3

Test 4 kao i za pokretne kocke, bilježi neznatni rast i test 7 postiže neznatno slabije rezultate.

Za testove 3, 4, 6 i 7 igra ima slične performanse s pokretnim i bez pokretnih kocki. Porast s testa 2 na test 3 je popriličan na pokretnim kockama i dosta manji na nepokretnim.

Uspoređivanjem testa 2 naspram testa 1 je veći s nepokretnim kockama nego s pokretnim kockama. Testovi stoga prikazuju da podatkovno orijentirani dizajn znatno utječe na performanse igre, 500% do 600%. Prelazak na podatkovno orijentirani dizajn ima manji porast sa pokretnim kockama jer pokretne kocke zahtijevaju više komputacijske snage zbog računanja pozicija kocki. Zbog toga rast kada se uključi *C# Jobs* sustav je mnogo veći sa pokretnim kockama, jer *C# Jobs* omogućuje višedretvenost. No za testove 3, 4, 6 i 7 FPS rezultati su identični jer se opet stvara usko grlo (engl. *bottleneck*) pri iscrtavanju kocki umjesto izračuna pozicija.

5. ZAKLJUČAK

Kao što je vidljivo iz rezultata testiranja, podatkovno orijentirani dizajn definitivno donosi prednosti naspram objektno orijentiranog programiranja. Podatkovno orijentirani dizajn nije nova tehnologija i poznata je od prije gdje je nekoliko igara implementiralo ili barem pokušalo koristiti podatkovno orijentirani dizajn. DOTS definitivno ima prednosti no također svoje mane iako ih je sve manje.

Podatkovno orijentirani dizajn nije još popularan zbog svoje strme početne krivulje učenja. Teško je za shvatiti i vizualizirati cijelu ideju naspram objektno orijentiranog programiranja. Drugi razlog je primjena objektno orijentiranog programiranja unutar drugih grana programiranja gdje prednosti performansi podatkovno orijentiranog dizajna nisu toliko izražene. Negativne strane ne opravdavaju dodatne troškove educiranja programera i povećano vrijeme razvoja. Podatkovno orijentirani dizajn nije nužno teži oblik programiranja ali jest drugačiji što programerima koji imaju iskustva s objektno orijentiranim programiranjem čini prelazak teži. Podatkovno orijentirani dizajn obvezuje pisanje ispravnim metodama programiranja što također programeru može stvarati osjećaj usporavanja, no prilikom izrade većih projekata ispravno strukturiran kod uvelike može ubrzati razvoj u kasnijim fazama izrade. Izrazito bolje performanse su značajne te posebno izražene pri velikom broju objekata, no prihvaćanje podatkovno orijentiranog dizajna će potrajati ne samo zbog privikavanje na drugačiji oblik strukturiranja koda već i podrške alata za razvoj. Iako je DOTS funkcionalan, mnoge funkcionalnosti koje programski okvir za igre već ima integrirane, nisu još dostupne za korištenje uz DOTS.

Unity s svojom DOTS tehnologijom čini podatkovno orijentirani dizajn dostupan masama što znači da će mnogo više programera testirati igre u podatkovno orijentiranom dizajnu te prihvatiti nedostatke za bolje performanse. Što više funkcionalnosti se prenese na DOTS, više ljudi će se odvažiti na eksperimentiranje s DOTS tehnologijom. Do 2030. godine možemo očekivati velike razlike u načinu izradi igara. Ne samo načinu izradi igara već i vrste igara, gdje DOTS omogućuje mnogo više objekata i kompleksnije svjetove.

6. LITERATURA

[1] Global Games Market to Generate $175.8 Billion in 2021; Despite a Slight Decline, the Market Is on Track to Surpass $200 Billion in 2023 [Online] Dostupno na: https://newzoo.com/insights/articles/global-games-market-to-generate-175-8-billion-in-2021-despite-a-slight-decline-the-market-is-on-track-to-surpass-200-billion-in-2023/ (14.10.2021.)

[2] Enhancing mobile performance with the Burst compiler [Online] Dostupno na: https://blog.unity.com/technology/enhancing-mobile-performance-with-the-burst-compiler (6.10.2021.)

[3] Retrochallenge 2018/04 (Now in COLOR) Refraction for the Atari 2600 [Online] Dostupno na: https://www.masswerk.at/rc2018/04/01.html (8.10.2021.)

[4] Unity pricing options [Online] Dostupno na: https://store.unity.com/ (6.10.2021.)

[5] What is DOTS [Online] Dostupno na: https://learn.unity.com/tutorial/what-is-dots-and-why-is-it-important (6.10.2021.)

[6] DOTS packages [Online] Dostupno na: https://unity.com/dots/packages#getting-started-dots (6.10.2021.)

[7] Megacity [Online] Dostupno na: https://unity.com/megacity (6.10.2021.)

[8] Understanding data-oriented design for entity component system [Online] Dostupno na: https://www.youtube.com/watch?v=0_Byw9UMn9g (6.10.2021.)

[9] Entity Systems are the future of MMOG development - Part 1 [Online] Dostupno na: http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (6.10.2021.)

[10] Unity manual - World [Online] Dostupno na: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/world.html (6.10.2021.)

[11] Unity manual - Entity manager [Online] Dostupno na: https://docs.unity3d.com/Packages/com.unity.entities@0.17/api/Unity.Entities.EntityManager.html (6.10.2021.)

[12] Unity manual - Archetype [Online] Dostupno na: https://docs.unity3d.com/Packages/com.unity.entities@0.17/api/Unity.Entities.EntityArchetype.html?q=entityar (6.10.2021.)

[13] Unity manual - Camera [Online] Dostupno na: https://docs.unity3d.com/Manual/class-Camera.html (6.10.2021.)

[14] PS4 vs. Xbox One Native Resolutions and Framerates [Online] Dostupno na: https://www.ign.com/wikis/xbox-one/PS4_vs._Xbox_One_Native_Resolutions_and_Framerates (13.10.2021.)

[15] What Is The Best FPS for Gaming? [Online] Dostupno na: https://www.gamingscan.com/best-fps-gaming/ (13.10.2021.)
[16] Smooth movies: Are high-frame rate films a good idea? [Online] Dostupno na: https://www.cnet.com/tech/home-entertainment/smooth-movies-are-high-frame-rate-films-a-good-idea/#:~:text=In%20the%20case%20of%20nearly,30%2C%2060%20or%20even%20120fps. (13.10.2021.)

7. OZNAKE I KRATICE

DOTS - Data-oriented technology stack

ECS - Entity component system

VRAM - video random access memory

FPS - frames per second

8. SAŽETAK

Rad je kreiran u obliku interaktivne igre gdje igrač ima mogućnost uspoređivanja planeta sunčevog sustava u trodimenzionalnom obliku. Igra je kreirana na klasični Unity način te uz nove Unity DOTS alate. Testiranje razlika između klasičnog Unity načina i DOTS tehnologije je učinjeno u obliku stres testa. Cilj je opteretiti igru objektima i pratiti utjecaj na performanse. Za potrebe testiranja kontrole su isključene kako bi testovi bili u kontroliranim uvjetima. Testovi su odrađeni s raznim postavkama i raznim uvjetima. Korištena su i tri različita uređaja za testiranje. Stres testom je utvrđeno kako se performanse igre značajno poboljšavaju korištenjem DOTS tehnologije. Prednosti su dosta izraženije kada je velika količina objekata u kadru kamere. DOTS tehnologija ima i nedostatke kao strma početna krivulja učenja te nedostatak funkcionalnosti naspram klasičnog Unity programskog okvira za igre.

Ključne riječi: unity, dots, podatkovno orijentirani dizajn, ecs, višedretvenost

9. ABSTRACT

The thesis is created in a form of an interactive video game where the player can compare the size of solar system planets in a three-dimensional representation. The game is created using the classic Unity way of creating games and using new Unity DOTS tools. Testing the differences between classic Unity and DOTS technology is done in a form of a stress test. The goal of the stress test is to use the system to the limits and observe how the system behaves. For the test, player controls have been disabled to allow a controlled testing environment. Tests are completed under various settings and conditions. Three different machines were used for conducting tests. The stress test showed how game performance increased drastically with new Unity DOTS technology. The performance increase is even more pronounced when there are lots of objects in the camera view. DOTS has some negative sides like a steep learning curve and lack of features compared to classic Unity.

Keywords: unity, dots, data-oriented, design, ecs, multithreading

IZJAVA O AUTORSTVU ZAVRŠNOG RADA

Pod punom odgovornošću izjavljujem da sam ovaj rad izradio/la samostalno, poštujući načela akademske čestitosti, pravila struke te pravila i norme standardnog hrvatskog jezika. Rad je moje autorsko djelo i svi su preuzeti citati i parafraze u njemu primjereno označeni.

Mjesto i datum	Ime i prezime studenta/ice	Potpis studenta/ice
U Bjelovaru, 13.10.2021.	Ivan Jerković	Jerković Ivan

Prema Odluci Veleučilišta u Bjelovaru, a u skladu sa Zakonom o znanstvenoj djelatnosti i visokom obrazovanju, elektroničke inačice završnih radova studenata Veleučilišta u Bjelovaru bit će pohranjene i javno dostupne u internetskoj bazi Nacionalne i sveučilišne knjižnice u Zagrebu. Ukoliko ste suglasni da tekst Vašeg završnog rada u cijelosti bude javno objavljen, molimo Vas da to potvrdite potpisom.

Suglasnost za objavljivanje elektroničke inačice završnog rada u javno dostupnom nacionalnom repozitoriju

Ivan Jerković

ime i prezime studenta/ice

Dajem suglasnost da se radi promicanja otvorenog i slobodnog pristupa znanju i informacijama cjeloviti tekst mojeg završnog rada pohrani u repozitorij Nacionalne i sveučilišne knjižnice u Zagrebu i time učini javno dostupnim.

Svojim potpisom potvrđujem istovjetnost tiskane i elektroničke inačice završnog rada.

U Bjelovaru, 13.10.2021.

Jerković Ivan

potpis studenta/ice