SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6515

ISPITIVANJE POVEĆANJA PERFORMANSI UNITY SUSTAVA KROZ PODATKOVNO ORIJENTIRANI TEHNOLOŠKI STOG

Matija Videković

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6515

ISPITIVANJE POVEĆANJA PERFORMANSI UNITY SUSTAVA KROZ PODATKOVNO ORIJENTIRANI TEHNOLOŠKI STOG

Matija Videković

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6515

Pristupnik:	**Matija Videković (0036510812)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	doc. dr. sc. Mirko Sužnjević

Zadatak: **Ispitivanje povećanja performansi Unity sustava kroz podatkovno orijentirani tehnološki stog**

Opis zadatka:

Tržište video igara raste velikom brzinom posljednjih godina te se na tržištu pojavio i veliki broj sustava za razvoj igara. Unity sustav za izradu igara je jedan od trenutno najpopularnijih. Razvojem nove generacije Unity sustava fokusiranog na podatkovno orijentirani tehnološki stog pokušavaju se poboljšati performanse razvijenih igara i omogućiti veće i detaljnije virtualne svjetove. Vaš zadatak je proučiti Unity sustav za izradu igara te u njemu razviti pokaznu aplikaciju za testiranje skalabilnosti rješenja temeljenog na podatkovno orijentiranom tehnološkom stogu i usporedbu s prethodnom verzijom Unity sustava. Razvijena aplikacija trebala bi imati mogućnost višestrukog korištenja određenih predefiniranih objekata, primjerice jato ptica ili riba. Potrebno je definirati metrike na kojima će se usporedba temeljiti, metodologiju ispitivanja te provesti ispitivanja. Skupljene rezultate potrebno je statistički analizirati te vizualizirati. Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Rok za predaju rada: 12. lipnja 2020.

Sadržaj

1. Uvod 1
2. Korišteni alati 3
  2.1. DOTS 3
3. Specifikacija simulacije 5
  3.1. Boid 6
  3.2. Zid 6
  3.3. Životinja 8
4. Implementacija simulacije 9
  4.1. Konfiguracija simulacije 9
    4.1.1. Konfiguracija boida 9
    4.1.2. Konfiguracija sustava prepreka 10
  4.2. OO simulacija 10
  4.3. DOTS Simulacija 13
5. Metodologija ispitivanja 16
  5.1. Mjerenje FPS-a 17
6. Rezultati ispitivanja 19
7. Zaključak 22
8. Literatura 23

1. Uvod

Glavni cilj ovog rada je istražiti značaj Unity DOTS (eng. Data-Oriented Technology Stack) tehnologije za razvoj složenih simulacija koje održavaju složeno stanje u računalnoj memoriji. Ideja je implementirati boide, objekte koji se kreću u prostoru na sličan način na koji to rade ptice ili ribe (simulacija jata).

Danas imamo na raspolaganju mnogo programskih jezika visoke razine koji se temelje na pisanju programskog koda u objektno orijentiranoj (OO) paradigmi (C#, Java, Python…), ako jezik i nudi više paradigmi objektno orijentirana paradigma je gotovo nezaobilazna [2]. Jezici temeljeni na OO paradigmi sigurno dominiraju po popularnosti [11]. Pomoću tih jezika pišemo programski kod koji je organiziran oko ugrađenih podatkovnih tipova (cijeli brojevi, realni brojevi, znakovi, nizovi…). Koristimo dobro poznate strukture podataka (hash tablica, lista, rječnik, set…), te opisujemo objekte koji enkapsuliraju podatke i/ili sadrže logiku za obradu podataka. Objekti se mogu međusobno referencirati, utjecati jedan na drugog (izmjena podataka unutar objekta) te možemo graditi složene hijerarhije objektnog nasljeđivanja. Cilj nam je oblikovati program koji je lagano nadogradiv i jednostavan za održavanje. To postižemo odgovarajućim, dobro poznatim oblikovnim obrascima koji opisuju interakcije između objekata. Želimo visoki stupanj apstrakcije, to znači udaljiti se od implementacijskih detalja raznih sklopovskih arhitektura za koje će naš program biti preveden (naredbeni set procesora, širina sabirnice, količina i brzina priručne memorije…), uz to se ne želimo brinuti ni o alokaciji i de alokaciji računalnih resursa jer nam to stvara dodatne brige te dodatni programski kod (što smanjuje lakoću nadogradnje i održavanja). Želimo moći opisivati program kao složene interakcije između objekata, ne brinemo o tome kako su podatci spremljeni u memoriji i kako im pristupamo. Upravo to nam nudi kombinacija programskih jezika visoke razine (u OO paradigmi) i radnih okvira na kojima oni rade.

Vidi se da objektno orijentirana paradigma ima mnogo primamljivih svojstava. Velika je prednost što nema značajnu ulaznu barijeru za početnike, nudi relativno jednostavno modeliranje odnosa iz stvarnoga svijeta pomoću objekata (pogotovo ako se radi u programskom jeziku visoke razine). Složenost dolazi primarno iz složenih međuovisnosti između objekata.

Programiranje u Unityu je do nedavno bilo isključivo bazirano na pisanju skriptnih komponenti koje se „kače“ na objekt u 3D sceni. Programski jezik za pisanje komponenti je

C# te se koristi OO paradigma. Ovaj način programiranje 3D igre je veoma intuitivan jer se stvara izravna veza između programskih objekata i objekata u 3D sceni. To je jedan od razloga zbog kojega je Unity stekao visoku popularnost među razvijačima 3D simulacijskih rješenja. To je također jedan od razloga zašto Unity nije učinkovit kao što bi mogao biti. Učinkovitost je veća što nam manje vremena treba da obavimo neki posao uz iste uvijete izvođenja, u 3D simulaciji to znači što brže obavljanje iteracije kako bi maksimizirali FPS (broj sličica u sekundi)[3]. Učinkovitost nam je važna jer nam omogućuje složene simulacije uz očuvanje dobrog FPS-a (tako da se ljudskom oku prikaz scene 3D simulacije čini „glatkim", u većini slučajeva ta brojka iznosi 60 FPS-ova). Također nam je bitno učinkovito izvoditi simulaciju kako bi smanjili potrošnju baterije na prijenosnim računalima i mobitelima.

Unity DOTS je uveden kako bi se povećala učinkovitost izvođenja. Jedan od ciljeva ovoga rada je ustanoviti koliko je učinkovitije izvođenje simulacije pomoću DOTS tehnologije u odnosu na izvođenje pomoću klasičnog OO pristupa [1].

Izvorni kod završnog rada je dostupan na adresi: https://gitlab.com/koljo45/boidi. Bilo kakva pitanja vezana uz rad mogu se proslijediti na email adresu: matija.videkovic@fer.hr.

2. Korišteni alati

Prva verzija Unitya je izdana 2005. godine, tada je razvijanje u Unityu bilo podržano isključivo na Mac OS X operacijskom sustavu. Danas je moguće razvijati Unity rješenja na Windows OS-u, macOS-u i Linux OS-u. Unity rješenja mogu biti izrađena za preko 25 jedinstvenih platformi (Windows, macOS, Linux, iOS, Android, PS4, Xbox…). Više od polovice mobilnih video igara je izgrađena pomoću Unitya [12][15] te je sve veći broj video igara koje ciljaju stolna računala i konzole. Posebno je popularno razvijati rješenja bazirana na virtualnoj ili proširenoj stvarnosti u Unityu, takva rješenja mogu ciljati prijenosne i/ili stolne, jače platforme. Da bi ovo sve omogućio Unity integrira mnogo raznovrsnih tehnologija (Apple ARKit, Google ARCore, Vuforia, Visage SDK…). Zbog toga Unity postaje značajno zastupljen na svim platformama. Posebno je zanimljiva nova Dana Oriented Technology Stack tehnologija zbog načina na koji utječe na sve prethodno navedene tehnologije.

2.1. DOTS

DOTS nudi arhitekturu koja omogućava razvoj rješenja temeljenih na podatkovnoj paradigmi. Cilj podatkovne paradigme je blisko grupirati srodne podatke u RAM-u, dati najviše pažnje podatcima i načinu na koji se podatci pohranjuju u memoriji. Ovaj pristup je potaknut željom da se priručna (eng. cache) memorija središnjeg procesora što bolje iskoristi. Ako su podatci pametno grupirani onda i programski kod za obradu podataka može biti grupiran. Sva ova grupiranja omogućavaju da se izvođenje programa odvija sa što manje promašaja pri dohvatu podataka iz priručne memorije [5].

U središtu DOTS-a je Entity Component System (ECS) [14]. Tri glavna dijela ECS-a su:

1. Entiteti (eng. Entities) – stvari koje sačinjavaju simulaciju.
2. Komponente (eng. Components) – podatci koje asociramo sa entitetima.
3. Sistemi (eng. Systems) – logika koja transformira podatke iz trenutnog stanja u iduće stanje.

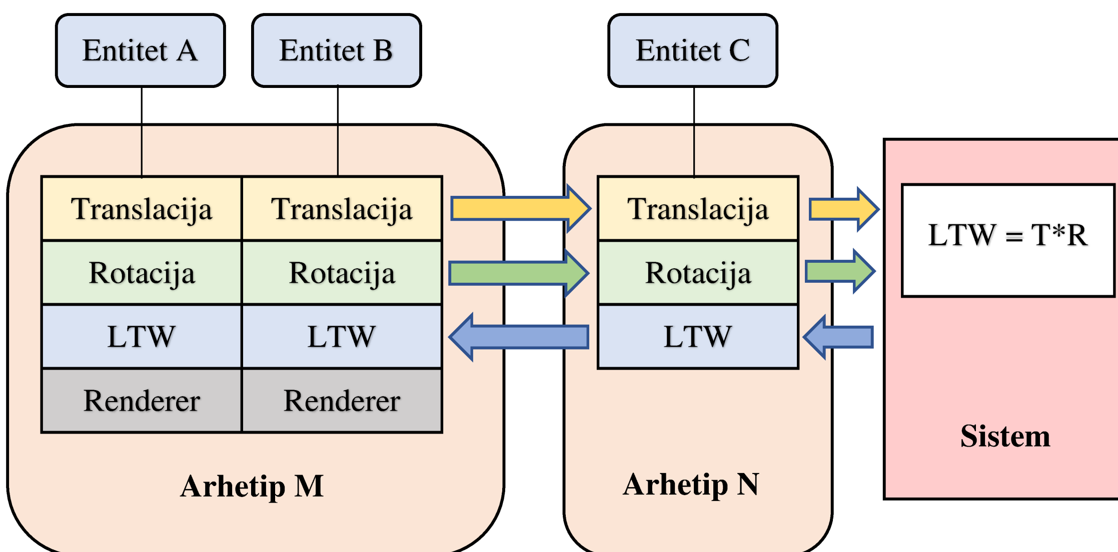
U ovoj simulaciji entiteti će biti većina stvari koje se nalaze u Unity sceni, u ovom trenutku kamera i izvor svijetlosti ne postoje unutar ECS sustava (postoje kao klasični Unity objekti). Na primjer, u simulaciji boida entiteti su: boidi, prepreke, objekti za stvaranje boida te dodatni pomoćni objekti. O entitetu se može razmišljati kao o jedinstvenom identifikatoru koji služi za referenciranje seta komponenti vezanih uz neki objekt.

Komponenta je struktura podataka koja se povezuje sa određenim entitetom. Postoji više vrsta komponenti:

- IComponentData – služi za pohranjivanje podataka koji su jedinstveni za pojedini entitet, također može biti korištena za pohranjivanje podataka na razini komada (eng. chunk).
- IBufferElementData
- ISharedComponentData – kategorizira ili grupira podatke s obzirom na vrijednost unutar arhetipa.
- ISystemStateComponentData
- ISharedSystemStateComponentData

Sistem sadrži logiku koja je potrebna za izmjenu komponentnih podataka. Sistemi se nalaze unutar sistemskih grupa. Sistemske grupe se mogu ugnježđivati te je moguće definirati željeni poredak izvođenja sistema.

Odnosi unutar ECS-a su ilustrirani na **Slika 2-1**. Arhetip je opisan jedinstvenim setom komponenti koja ga sačinjavaju. U ovom primjeru sistem čita komponente translacija i rotacija, množi ih te rezultat sprema u LTW komponentu.



Slika 2-1 Prikaz ECS odnosa.

Činjenica da entiteti A i B imaju renderer komponentu a entitet C nema ne utječe na rad sustava (A i B pripadaju arhetipu M, a C pripada arhetipu N).

ECS alocira memoriju u komadima. Kada se jedan komad memorije popuni entitetima alocira se novi komad memorije. To znači da sa jednim arhetipom može biti povezano više komada memorije. Dodatno, ako arhetip sadrži ISharedComponentData svi entiteti unutar arhetipa se grupiraju s obzirom na vrijednost ISharedComponentData. Svaki entitet će se nalaziti unutar komada memorije gdje svi entiteti dijele istu vrijednost ISharedComponentData komponente. Ako se entitetu promjeni vrijednost ISharedComponentData komponente tada taj entitet biva prebačen u odgovarajući komad memorije (ako je dosada neviđena vrijednost stvara se novi komad memorije u kojeg se premješta entitet).

Rad sistema se izvršava u svakoj iteraciji simulacije što čini glavnu petlju sistema. Glavna petlja svakog sistema se izvodi na istoj, glavnoj dretvi. Zbog načina na koji su organizirani podatci i sistemi paralelizam je znatno olakšan. DOTS nudi C# Job system koji omogućava raspodjele posla na više dretvi (paralelizam).

3. Specifikacija simulacije

Simulacija se odvija u 3D prostoru unutar Unity scene. Scena je popunjena objektima različitih vrsta, neki objekti su dio simulacije dok drugi služe za prikaz simulacije na zaslonu računala. Obvezne vrste objekta u sceni su:

1. Boid
2. Objekt za stvaranje boida (eng. boid spawner)
3. Zid
4. Izvor svijetla
5. Kamera

Postoje dvije scene, jedna je napravljena tako da radi sa OO pristupom, dok je druga scena prilagođena za DOTS pristup. Obje scene imaju dodatne, specifične vrste objekata vezane uz pojedinu izvedbu. Simulacije koje se izvode u pojedinačnoj sceni su identične u smislu izračuna putanje boida, to znači da se boidi ponašaju jednako u obje scene. Dodatno, sve gore navedene vrste objekata će biti identično postavljeni kako bi se ostvarili jednaki uvjeti izvođenja. Za potrebe mjerenja u obje scene je dodan pomoćni objekt koji obavlja mjerenja te sprema rezultate.

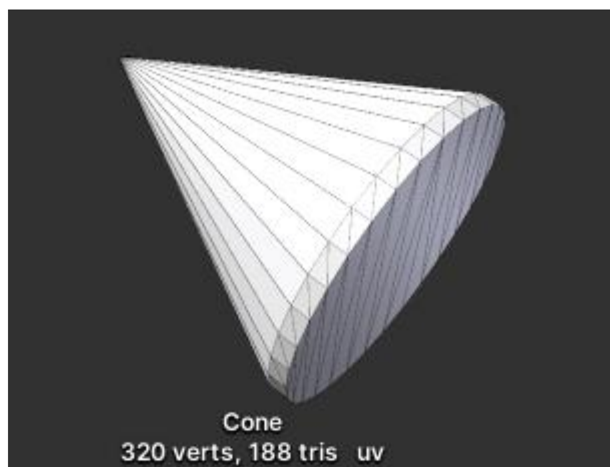
3.1. Boid

Za simulaciju boida korišten je model kretanja gdje boid bira smjer na temelju neposredne okoline u kojoj se nalazi. Sfera predstavlja vidno polje boida. Svi boidi koji su unutar vidnog polja boida su njegovi susjedni boidi. U svakoj iteraciji simulacije boid bira novi smjer kretanja s obzirom na smjer kretanja i poziciju njemu susjednih boida. Postoje tri glavna pravila koja utječu na kretanje boida. Ta pravila su:

1. Izbjegavanje sudara: izbjegavanje sudara sa susjedima
2. Podudaranje smjera: poravnati smjer kretanja sa susjedima
3. Grupno kretanje: kretanje prema susjedima tako da se grupa održi

[6]

Na **Slika *3-1*** vidljiv je žičani prikaz boida. Odabran je jednostavan stožac zbog manjih zahtjeva na sustav za iscrtavanje. U simulaciji vrh stošca je okrenut u smjeru kretanja boida.



Slika 3-1 Žičani prikaz boida.

3.2. Zid

Objekt vrste zid predstavlja prepreku boidu. Zid je ravnina iza koje boid ne smije ići. Ova vrsta prepreke je dodana kako bi se boidi mogli zadržati u ograđenom području. To je bitno za puni prikaz simulacije i interakcije između boida, bez ograde boidi bi se raspršili u beskonačan prostor.

Za modeliranje prepreka postoje dva pristupa. Prvi pristup se temelji na odbojnoj sili koja izvire iz prepreke i odbojno djeluje na boid. Glavna prednost ovog pristupa je jednostavnost implementacije. Odbojna sila se može opisati kao jednostavno polje koje usmjerava boid. Nedostatak ovog pristupa je to što boid nema realistično suočavanje s preprekom. To je

veoma očito kada boid krene izravno prema prepreci (normala prepreke je suprotna kretanju boida), u tom slučaju boid se ne zakreće kako bi izbjegao prepreku nego se usporava pod utjecajem polja prepreke. Boid se također ne može kretati uz prepreku, to je nešto što bi trebalo biti moguće ali nije zbog odbojnog polja prepreke. Ideja je da boid isto kao ptica ili riba uzima u obzir prepreke koje vidi iz daleka te se na temelju toga usmjeri kako bi izbjegao prepreku.

U drugom pristupu boid aktivno prati prepreke koje mu se nalazi u smjeru kretanja. Kada prepozna prepreku boid pokušava pronaći smjer kretanja koji je najbliži trenutnome smjeru kretanja te dopušta čitavom tijelu boida nesmetani prolazak.

Za simulaciju je korištena kombinacija ova dva pristupa. Prepreke je modelirana kao ravnina iza kojih boid ne smije ići. Ravnina je zapisana u analitičkom obliku, taj oblik je dan u jednadžbi (*5.5*).

$$Ax + By + Cz + D = 0 \tag{3.1}$$

U jednadžbi (*3.4*) (A, B, C) predstavlja vektor normale ravnine, D predstavlja udaljenost ravnine od ishodišta koordinatnog sustava.

Za svaki boid se računa točka x koja se nalazi u smjeru kretanja v, ta točka je udaljena za iznos d od središta boida s. Vektor v je normaliziran. Ovaj odnos je opisan jednadžbom (*3.4*).

$$x = s + d * v \tag{3.2}$$

Idući korak je točku x prebaciti u homogeni prostor. Pomoću izraza (*3.4*) točka x se prebacuje u homogeni prostor.

$$x_h = (x_x, x_y, x_z, 1) \tag{3.3}$$

Sada se pomoću izraza (*3.4*) množi ravnina u analitičkom obliku i točka x u homogenom prostoru.

$$R = A * x_x + B * x_y + C * x_z + D * 1 \tag{3.4}$$

Rezultat R može zadovoljavati jedan od tri uvjeta:

1. R = 0
2. R < 0

3. R > 0

U prvom slučaju točka x je na ravnini, u drugom je ispod ravnine, u trećem je iznad ravnine. U prvom i drugom slučaju boid bi trebao izbjegavati zid. Zid se izbjegava tako da se smjer kretanja boida zakrene prema smjeru normale zida (A, B, C). U trećem slučaju boid je dovoljno daleko od zida te ga ne mora izbjegavati.

Ovaj pristup osigurava da je boid iznad ravnine zida, čak i ako se nekim slučajem dogodi da je boid silom izbijen ispod ravnine zida jako brzo se vraća nazad pod utjecajem sile zida. Boid se također može nesmetano kretati uz zid te ga izbjeći u opasnosti od izravnog sudara.

3.3. Životinja

Za potrebe simulacije je korištena životinja, to je proširenje osnovnog ponašanja boida (životinja je također boid). Uz osnovna tri pravila boida životinja prati još dodatna dva pravila.

1. Izbjegavanje predatora: izbjegavanje životinja koje su označene kao predatori
2. Lov lovine: praćenje životinja koje su označene kao lovina

Sve životinje u simulaciji su smještene u virtualne slojeve. Svaka životinja ima brojčanu oznaku koja govori na kojem se sloju nalazi. Predatori i lovina su organizirani na temelju slojeva. Životinja može biti predator za neki sloj te lovina za neki drugi.

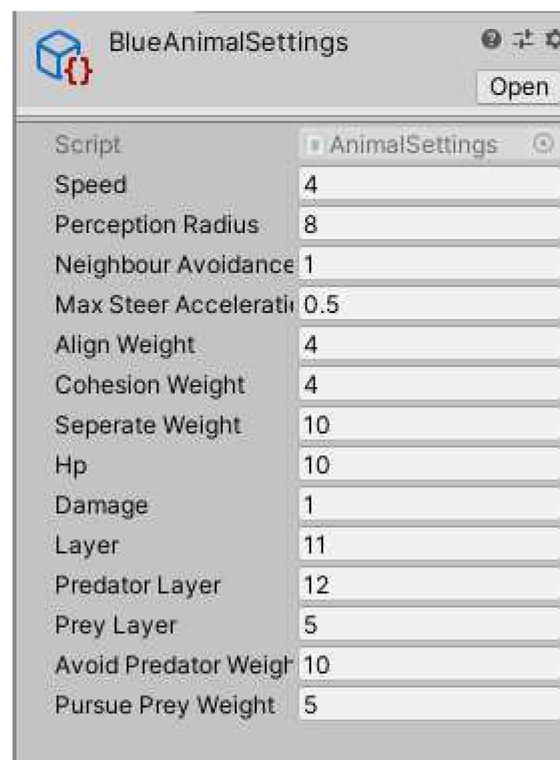
4. Implementacija simulacije

Obje scene se nalaze unutar istog Unity projekta. Scene izgrađene za pojedinačnu simulaciju su jednako strukturirane, u obje scene su korišteni isti 3D objekti za prikaz prepreka i boida. Obje scene koriste iste konfiguracijske objekte koji će sada biti opisani.

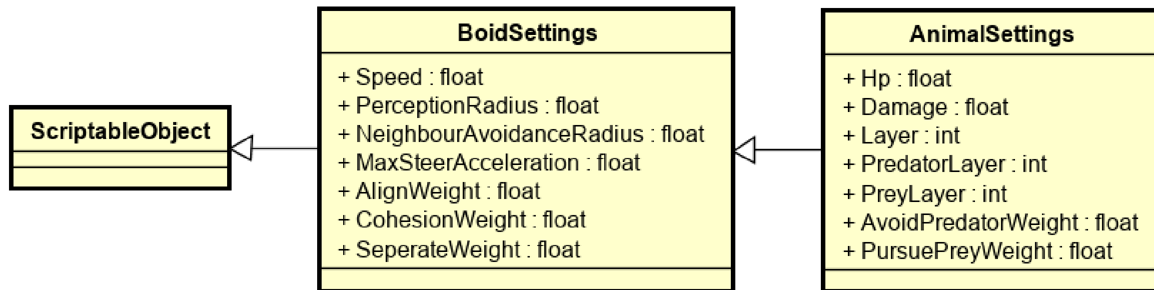
4.1. Konfiguracija simulacije

4.1.1. Konfiguracija boida

Za konfiguraciju boida su korišteni skriptabilni objekti [17]. Skriptabilni objekti služe za pohranjivanje podataka, takvi objekti se inače koriste kao projektni resursi. Jednom kada stvorimo skriptabilni objekt taj objekt može biti referenciran unutar scene. Skriptabilni objekti koji opisuju konfiguraciju boida postoje na razini projekta, to znači da iste objekte koristimo u obje scene. Na **Slika *4-1*** se može vidjeti jedna konfiguracija životinje. Uređenje objekta je opisano na **Slika *4-2***



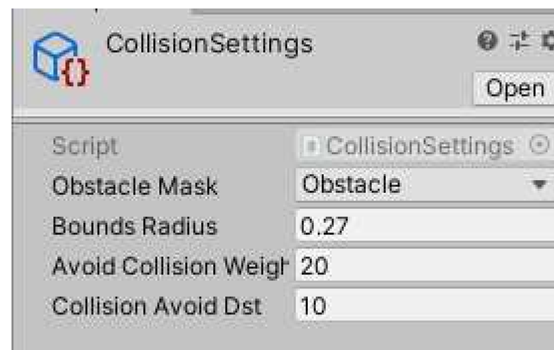
Slika 4-1 Konfiguracija neke životinje



Slika 4-2 Uređenje konfiguracijskog objekta za boide

4.1.2. Konfiguracija sustava prepreka

Za opisivanje interakcije između boida i prepreke također je korišten konfiguracijski skriptabilni objekt. Na **Slika *4-3*** se može vidjeti primjer jedne konfiguracije.

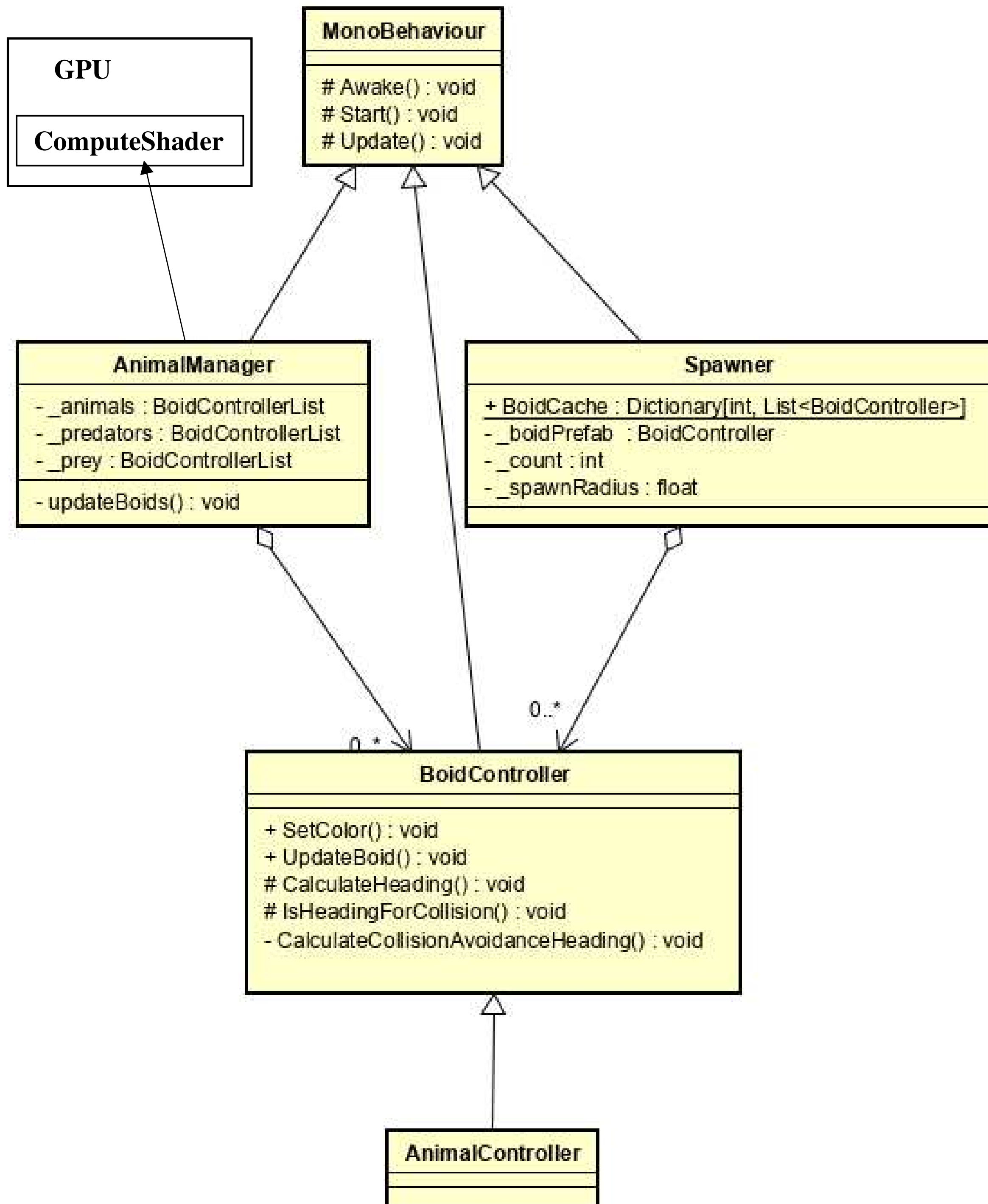


Slika 4-3 Konfiguracija sustava prepreka

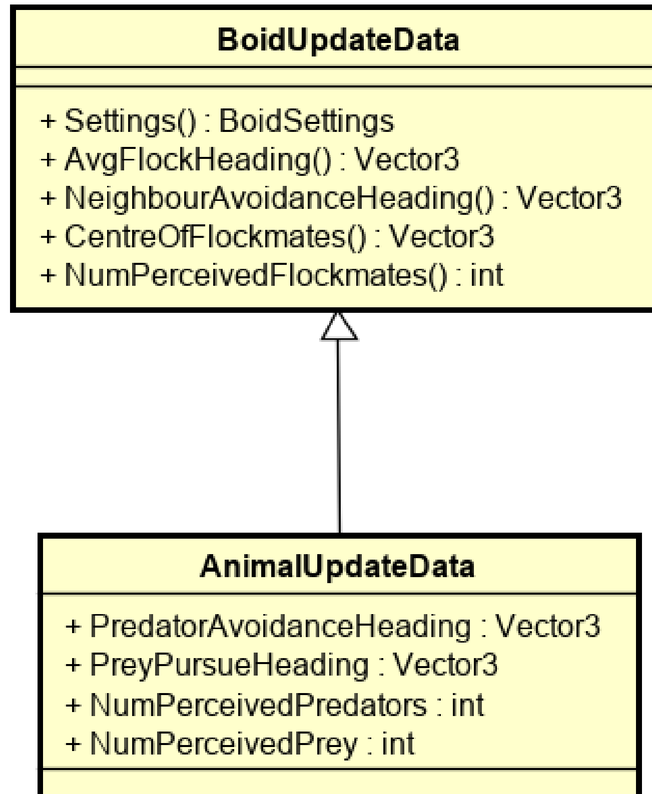
4.2. OO simulacija

U Unityu svaka skriptna komponenta nasljeđuje Unityevu ugrađenu MonoBehaviour klasu [16]. Na svaki objekt u sceni koji je zadužen za stvaranje boida dodajemo Spawner komponentu. Zadaća Spawner komponente je stvoriti zadani broj zadanih objekata na zadanom sloju, također sadržava statičku mapu koja pohranjuje listu stvorenih BoidControllera za svaki sloj radi učinkovitijeg dohvata sloja te smanjuje podatkovni otisak BoidController komponente (ne mora pohranjivati sloj za svaki boid zasebno). Objekte koje Spawner stvara trebali bi imati na sebi BoidController komponentu. Zadaća BoidController komponente je obaviti usmjeravanje boida s obzirom na BoidUpdateData (sadržava informacije o susjednim boidima) koju je primila kao argument metode UpdateBoid. Sadržaj BoidUpdateData je opisan na **Slika *4-5***. AnimalManager u svakoj iteraciji šalje pozicijske podatke svih životinja na sloju za koji je zadužen ComputeShaderu [4]. Zadaća ComputeShadera je učinkovito, koristeći paralelizam izračunati informacije potrebne za

popunjavanje AnimalUpdateData. ComputeShader se izvodi na grafičkoj kartici. Ovi odnosi su prikazani na **Slika *4-4***. BoidController je generalizacija komponente AnimalController, moguće je nasljediti BoidController komponentu za stvaranje neke nove vrste boida, samo se treba osigurati da je virtualna funkcija UpdateBoid() pravilno nadjačana.



Slika 4-4 Dijagram nasljeđivanja Unity komponenti



Slika 4-5 Spremnici informacije potrebni za osvježavanje boida

Na **Isječak koda *4-1*** dana je glavna petlja ComputeShadera. Ova petlja se izvodi za svaki boid. Ovakav izračun inače je kvadratne složenosti jer svaki boid u grupi mora proći kroz svaki drugi boid u grupi. Pošto je korišten ComputeShader svaki boid dobiva svoju dretvu za izvođenje pa se izvodi jedna petlja po dretvi što daje linearnu složenost. Na **Isječak koda *4-2*** dana je petlja kojom se računa smjer bijega od predatora. Životinja ovom petljom mora proći kroz sve životinje koje su na sloju koji je definiran kao predatorski sloj. Slična petlja se izvršava i za izračun smjera lova lovine.

```
for (int indeksZ = 0; indeksZ < brojZ; indeksZ++) {
            Zivotinja zivotinja = zivotinje[indeksZ];
            float3 udaljenost = zivotinja.pozicija - pozicija;
            float sqrDst = udaljenost.x * udaljenost.x + udaljenost.y * udaljenost.y
                        + udaljenost.z * udaljenost.z;

            if (sqrDst < percepcija) {
                  smjerJata += zivotinja.smjer;
                  sredisteJata += zivotinja.pozicija;
                  velicinaJata++;

                  if (sqrDst < udaljenost_izbjegavanja) {
                        if (id.x == indeksZ)
                              continue;
                        smjerIzbjegavanjaSusjeda -= udaljenost / sqrDst;
                  }
            }
      }
```

Isječak koda 4-1 Glavna petlja ComputeShadera.

```
for (int indeksP = 0; indeksP < brojPredatora; indeksP++) {
            Transform predator = predatori[indeksP];
            float3 udaljenost = predator.pozicija - pozicija;
            float sqrDst = udaljenost.x * udaljenost.x + udaljenost.y * udaljenost.y
                         + udaljenost.z * udaljenost.z;

            if (sqrDst < percepcija) {
                  smjerIzbjegavanjaPredatora -= udaljenost / sqrDst;
            }
      }
```

Isječak koda 4-2 ComputeShader petlja za predatore.

4.3. DOTS Simulacija

U simulaciji postoje pet vrsta entiteta, tj. postoje pet vrsta arhetipa na temelju kojih se izgrađuju entiteti. Boidi, prepreke, kolizijske postavke (CollisionSettings) i objekti za stvaranje boida su dodani za potrebe simulacije, WorldTime entitet Unity automatski dodaje. Na **Tablica *1*** je prikazana struktura korištenih entiteta. Većinu komponenti Unity automatski dodaje, te komponente koriste ugrađeni sistemi za prikaz scene.

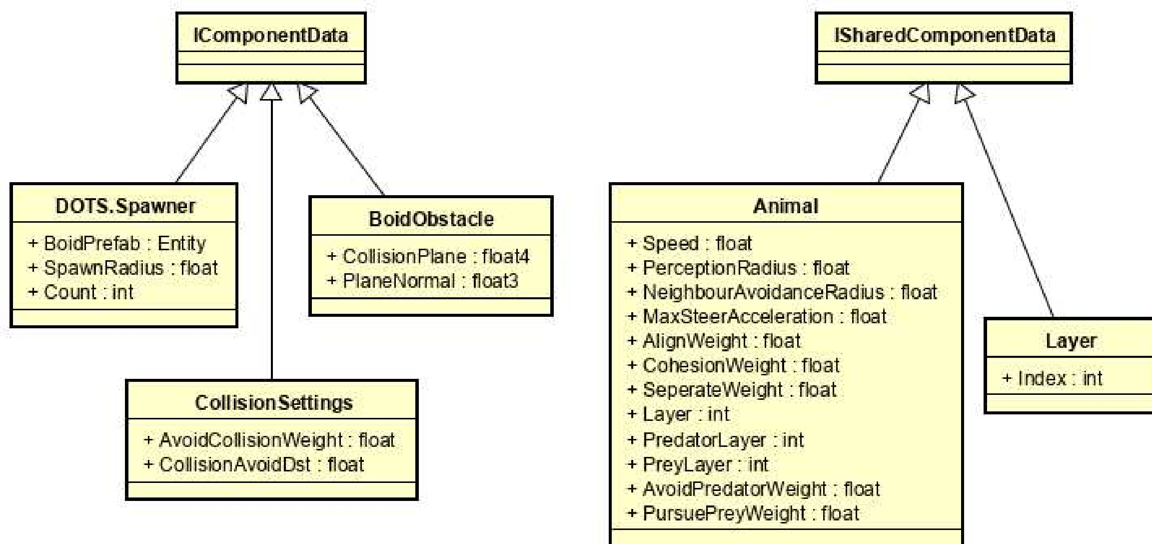
Tablica 1 Prikaz organizacije komponenti za pojedinu vrstu entiteta, * označava automatski dodanu komponentu

<table>
<tr><th>Boid</th><th>Prepreka</th><th>WorldTime</th><th>CollisionSettings</th></tr>
<tr><td>LinkedEntityGroup*
Prefab*
Animal
Layer</td><td>NonUniformScale*
Rotation*
Translation*
BoidObstacle</td><td>WorldTime*
WorldTimeQueue*</td><td>CollisionSettings</td></tr>
<tr><td colspan="2">RenderBounds*
WorldRenderBounds*
LocalToWorld*
PerInstanceCullingTag*
RenderMesh*
ChunkWorldRenderBounds*</td><td></td><td></td></tr>
</table>

Spawner
Spawner LocalToWorld* Rotation* Translation*

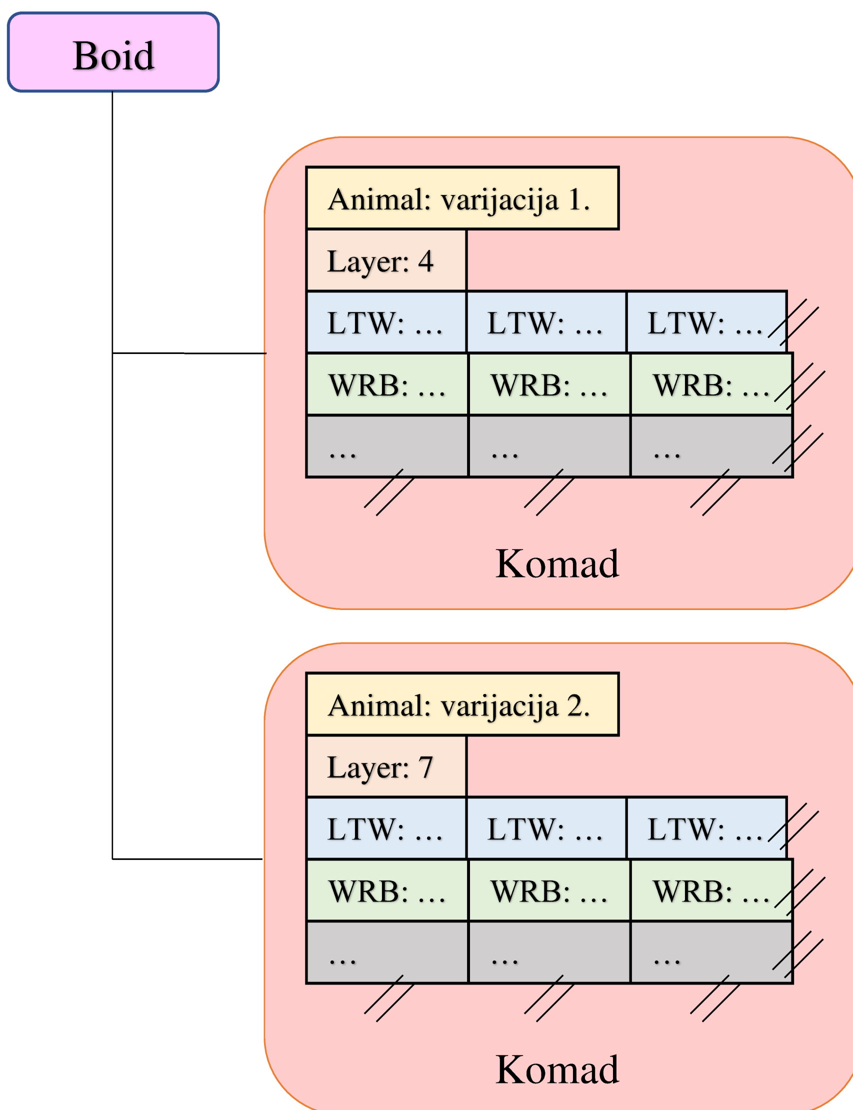
Vrsta korištene komponente i njezin sadržaj je prikazan na **Slika *4-6***. Simulacija koristi dvije komponente vrste ISharedComponentData, to su Animal i Layer komponente. Na **Tablica *1*** je vidljivo da obje komponente opisuju arhetip za izgradnju boid entiteta. To znači da su unutar arhetipa Boid entiteti raspoređeni u komade s obzirom na vrijednosti Animal i Layer

komponente. Takva podjela je korisna jer omogućava dohvat boida koji pripadaju određenom rodu životinje ili su na određenom sloju (ili oboje). Fizička raspodjela Boid arhetipa je opisana na **Slika *4-7***, prikaz je pojednostavljen kako bi primarno prikazao komponente koje su bitne za simulaciju. Arhetip se još dodatno dijeli s obzirom na RenderMesh komponentu koju koristi ugrađeni sistem za iscrtavanje (tijekom testiranja je korišten isti prikaz za sve boide tako da podjele na temelju te komponente ustvari ni nema).



Slika 4-6 Korištene komponente.

Izračun za kretanje boida je gotovo identičan ComputeShader kodu. Razlog tomu je Unity.Mathematics biblioteka koja nudi tipove i funkcije koji sintaksom podsjećaju na programski kod sjenčara. Jedina razlika je u tome što DOTS simulacija izračune provodi na središnjem procesoru, a ComputeShader na grafičkoj kartici. Dodatno, DOTS simulacije ne provodi izračune za sve grupe istovremeno nego slijedno. Razlog tomu je DOTS sigurnosni sustav koji zabranjuje takav izračun iako je izračun sasvim siguran. Tijekom izrade simulacije nije pronađen dobar način za zaobilazak sigurnosnog sustava tako da ovo ograničenje zasada ostaje.



Slika 4-7 Fizička raspodjela Boid arhetipa

Spawner entitet postoji samo u trenutku stvaranja scene. Nakon što su boid entiteti dodani u simulaciju SpawnSystem briše obrađene Spawner entitete.

Za potrebe simulacije su napravljena četiri sistema:

- AnimalSystem – u svakoj iteraciji računa nove pozicije životinja
- CollisionSystem – u svakoj iteraciji usmjerava boide tako da se ne zabiju u zid
- AnimalConversion – priprema primjerak entiteta životinje za potrebe objekta za stvaranje životinja
- SpawnSystem – obrađuje Spawner entitete tako da dodaje boide u simulaciju na temelju Spawner komponente

AnimalSystem i CollisionSystem se izvode u svakoj iteraciji simulacije. AnimalConversion se izvodi samo jednom na početku simulacije kako bi pripremio entitet životinje za stvaranje, to vrijedi tijekom izvođenja simulacije unutar Unitya. Rješenje koje je izgrađeno za odredišnu platformu ne izvodi konverzijske sisteme jer je konverzija obavljena tijekom izgradnje rješenja.

5. Metodologija ispitivanja

Prepoznato je da simulacijska rješenja mogu biti objektivno ocijenjena s obzirom na njihovu učinkovitost. Učinkovitija simulacija obavi isti posao u kraćem vremenu, pod uvjetom da se izvršna okolina ne mijenja. U programiranju se to postiže optimizacijom koda i izbjegavanjem skupih operacija kao što je dohvat podataka iz RAM-a (pametnim korištenjem priručne memorije) [10]. Interaktivne simulacije su posebno izazovne jer moraju izgledati dobro korisniku te uz to moraju biti i responzivne. Ovaj problem nam je dodatno otežan ako je simulacija prikazana korisniku preko VR naočala [7]. Povećanje učinkovitosti povlači smanjenu potrebu za procesorskim resursima te štedi bateriju na prijenosnim uređajima, ovo je posebno bitno jer je Unity najpopularnija tehnologija za izradu mobilnih video igara.

Za responzivnost sustava se može reći da je tim veća što mu manje vremena treba da prikaže rezultat korisnikove interakcije sa simulacijom koja se odvija unutar sustava. Na primjer, okretanje kamere u video igrama pomoću miša, što manje vremena treba da se na zaslonu prikaže nova slika koja je rezultat pomjeranja miša to je responzivnost veća. Na responzivnost utječe više faktora, ne samo simulacija [8]. Unosne jedinice (miš, tipkovnica itd.) primaju korisnikov unos, operacijski sustav obrađuje unos te prosljeđuje unos simulaciji. Simulacija modificira svoje stanje s obzirom na unos te se onda prikaz simulacije šalje grafičkoj kartici koja prikaz prosljeđuje na zaslon gdje je onda vidljiva promjena. Svakako se može zaključiti da će povećanje učinkovitosti simulacije povoljno utjecati na responzivnost (u slučaju da usko grlo ne stvara neki drugi faktor).

Za ispitivanje pojedinog rješenja (simulacije) korišten je iznos prosječnog FPS-a. FPS je broj sličica u sekundi koje je naš sustav sposoban prikazati. FPS izravno ovisi o duljini koraka simulacije. Što je korak simulacije manji to je FPS veći. Dodatno je analiziran korak simulacije kako bi se odredila raspodjela računalnih resursa.

Konfiguracija na kojoj su izvedene mjerenja simulacije je sljedeća:

- CPU: Intel Core i5-3470
- RAM: 8 GB DDR3 RAM (1600 MHz)
- GPU: AMD Radeon R9 380 Series
- 60Hz monitor

Za mjerenje ukupne potrošnje resursa korišten je Windows Resource Monitor.

Za analizu koraka simulacije korišten je Unity Profiler.

5.1.Mjerenje FPS-a

Za mjerenje FPS-a u OO simulaciji korištena je MonoBehaviour skripta koja je dodana na objekt za mjerenje. Nakon pokretanja simulacije skripta čeka određeno vrijeme prije nego krene sa skupljanjem uzoraka, u ovim mjerenjima vrijeme čekanja je uvijek 10 sekundi. Vrijeme čekanja je uvedeno kako inicijalizacija simulacije ne bi utjecala na mjerenja, 10 sekundi je odabrano jer nakon tog vremena inicijalizacija je sigurno prošla.

Nakon vremena čekanja skripta kreće sa skupljanjem određenog broja uzoraka. Trenutni uzorak se mjeri u sekundama, kao vrijeme izvršavanja prethodnog koraka simulacije. U ovim mjerenjima broj uzoraka u svakom mjerenju je bio 10000.

Nakon što je skupljen određeni broj uzoraka skripta računa prosjek svih uzoraka kao što je prikazana u izrazu (*5.5*).

$$\overline{uzorak} = \frac{\sum_{i=1}^{N=1000} uzorak_i}{N} \tag{5.1}$$

Zatim izračun prosječnog FPS-a ide po izrazu (*5.5*).

$$\overline{FPS} = \frac{1}{\overline{uzorak}} \tag{5.2}$$

Standardna devijacija uzoraka se računa po izrazu (*5.5*).

$$\sigma_{uzorak} = \sqrt{\frac{1}{N} \sum_{i=1}^{N=1000} (x_i - \bar{x})^2} \tag{5.3}$$

Za izračun standardne devijacije FPS-a se koristi izraz (*5.5*).

$$\sigma_{FPS} = \frac{1}{\overline{FPS}} - \frac{1}{\overline{FPS} + \sigma_{uzorak}} \tag{5.4}$$

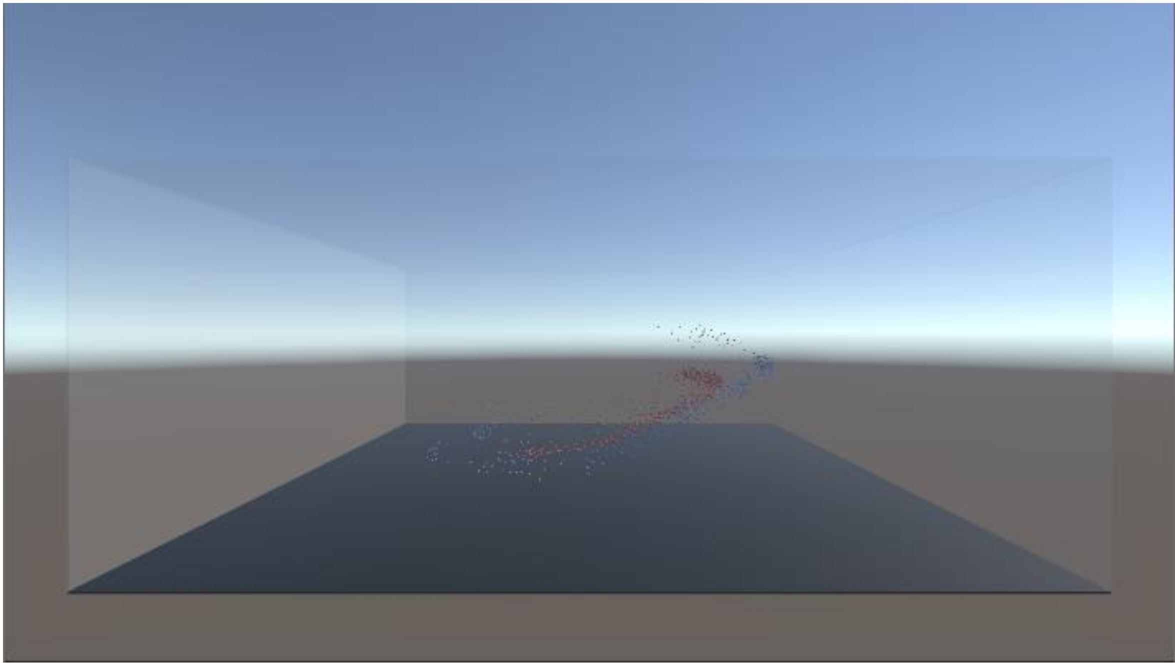
Sada FPS može biti prikazan na način (*5.5*).

$$FPS = \overline{FPS} \pm \sigma_{FPS}\ s^{-1} \tag{5.5}$$

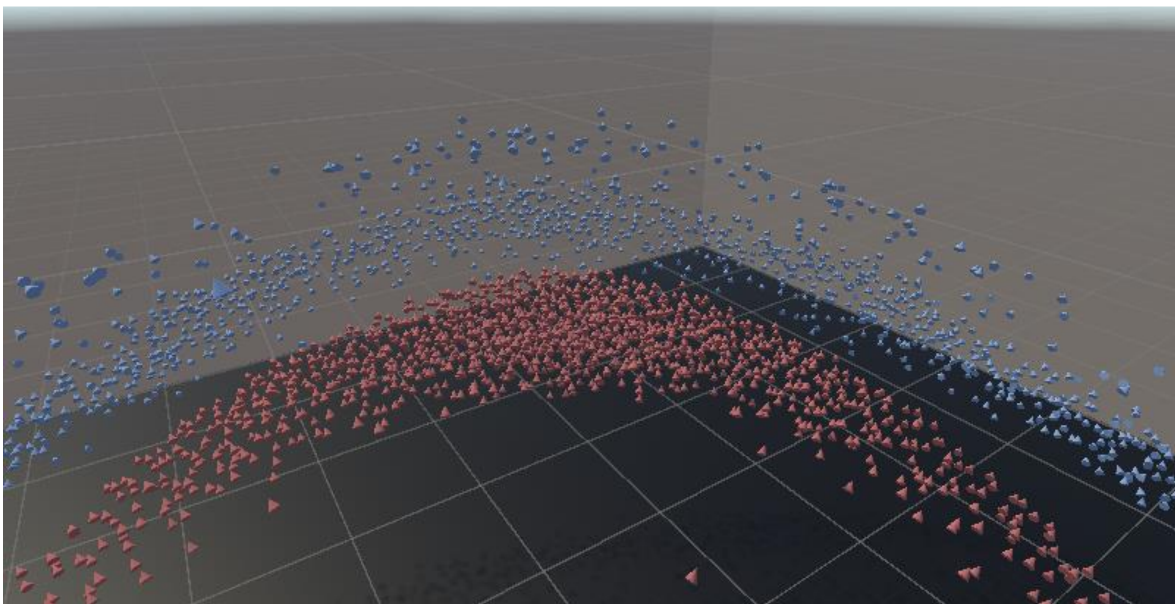
Nakon što je mjerenje završeno skripta sprema izračune u tekstualnu datoteku na disku.

Rađeno je 10 mjerenja, 5 za svaku simulaciju. Mjerenja su spojena pomoću alata [9].

Prvi set mjerenja simulira dvije grupe životinja, svaka po 500 životinja. Drugi set mjerenja simulira grupe po 1500 životinja. Životinja prikazana crvenim boidom je predator koji lovi životinju prikazanu plavim boidom. Prikaz simulacije je vidljiv na **Slika *5-1***.



Slika 5-1 Prikaz simulacije životinja.



Slika 5-2 Prikaz boida unutar simulacije.

6. Rezultati ispitivanja

Tablica 2 Izmjereni FPS za grupe od 500 životinja.

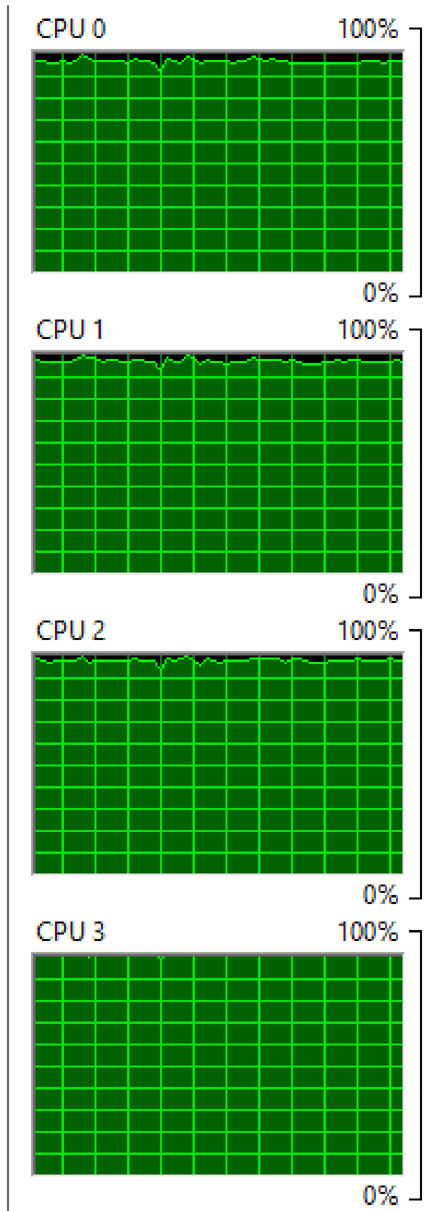
Br.	OO$[s^{-1}]$	DOTS$[s^{-1}]$
1.	96.8444 ± 3.8626	347.6117 ± 33.1061
2.	95.3843 ± 4.3243	346.4811 ± 29.2691
3.	97.0560 ± 3.8275	350.1290 ± 21.8360
4.	96.3090 ± 3.7656	350.3417 ± 22.0906
5.	96.0009 ± 4.8612	352.2478 ± 24.7095
∑	96.3189 ± 4.1921	349.3623 ± 26.6421

Na **Tablica *2*** su prikazani rezultati mjerenja za grupe od 500 životinja. Obje simulacije su izgledale jednako glatko. Razlog tomu je korišteni ekran koji ima frekvenciju osvježivanja od 60Hz.

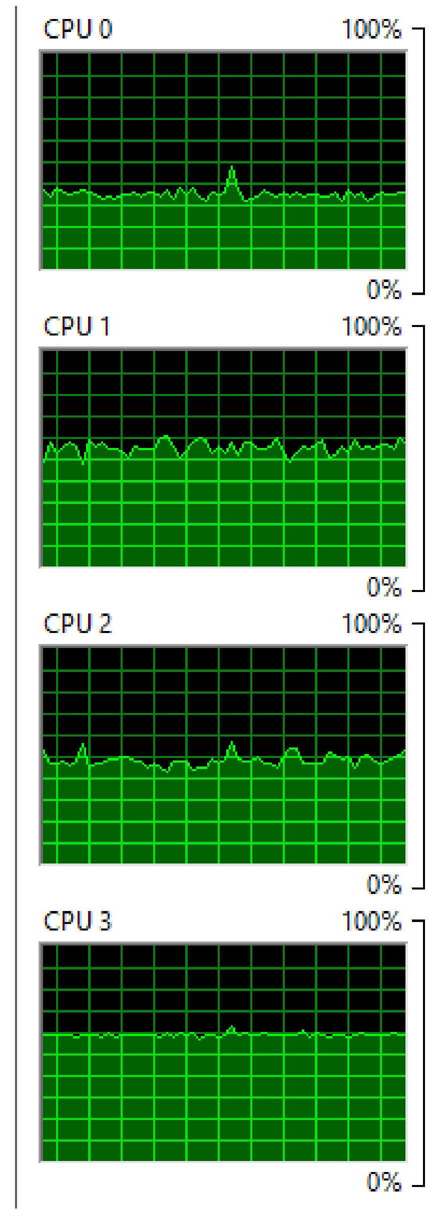
Tablica 3 Izmjereni FPS za grupe od 1500 životinja.

Br.	OO$[s^{-1}]$	DOTS$[s^{-1}]$
1.	35.5233 ± 0.9739	83.7904 ± 9.6207
2.	35.6395 ± 0.6622	86.7020 ± 11.9898
3.	35.6104 ± 0.5505	86.3743 ± 11.8025
4.	34.7032 ± 0.6776	85.1012 ± 11.0681
5.	35.6123 ± 0.7459	85.3103 ± 10.9214
∑	35.4177 ± 0.8187	85.4556 ± 11.1594

Na **Tablica *3*** su prikazani rezultati mjerenja za grupe od 1500 životinja. DOTS simulacija je zamjetno bolje radila.



Slika 6-2 Iskorištenost središnjeg procesora tijekom DOTS simulacije.



Slika 6-1 Iskorištenost središnjeg procesora tijekom OO simulacije.

Na **Slika 6-1** i **Slika 6-2** su prikazane iskorištenosti središnjeg procesora tijekom pojedine simulacije.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	73.6%	0.4%	2	111.6 KB	13.93	0.08
▼ Update.ScriptRunBehaviourUpdate	39.7%	0.0%	1	111.2 KB	7.53	0.00
▼ BehaviourUpdate	39.7%	0.0%	1	111.2 KB	7.53	0.00
▼ AnimalManager.Update()	39.7%	3.4%	2	111.2 KB	7.51	0.65
Updating Boids	20.1%	20.1%	2	0 B	3.81	3.81
▶ Gfx.GetComputeBufferData_Request	12.2%	0.0%	2	0 B	2.32	0.00
▶ LoadAnimalComputeBuffer	3.7%	3.4%	2	86.5 KB	0.71	0.65
Compute.Dispatch	0.0%	0.0%	2	0 B	0.00	0.00
GC.Alloc	0.0%	0.0%	16	24.7 KB	0.00	0.00
FPSDisplay.Update()	0.0%	0.0%	1	0 B	0.00	0.00
▶ Camera.Render	28.7%	0.2%	1	0 B	5.43	0.04

Slika 6-3 Raspodjela računalnih resursa tijekom OO simulacije.

Overview	Total	Self	Calls	GC Alloc	Time ms
▼ PlayerLoop	56.0%	0.8%	2	2.2 KB	5.23
▶ SimulationSystemGroup	26.4%	0.0%	1	192 B	2.46
▶ Camera.Render	12.3%	0.3%	1	384 B	1.15
▶ PresentationSystemGroup	8.7%	0.0%	1	256 B	0.81
▶ InitializationSystemGroup	5.2%	0.0%	1	1.4 KB	0.49

Slika 6-4 Raspodjela računalnih resursa tijekom DOTS simulacije.

Na **Slika *6-3*** je vidljivo koliko je vremena potrebno za izvođenje pojedinog dijela OO simulacije.

- Updating Boids – pozivanje metode UpdateBoid() svake životinje s AnimalUpdateData argumentom koji je popunjen rezultatima ComputeShadera
- Gfx.GetComputeBufferData_Request – čekanje rezultata iz ComputeShadera
- LoadAnimalComputeBuffer – priprema podataka za ComputeBuffer, uključuje iteriranje kroz sve životinje
- Camera.Render – prikaz simulacije na zaslonu

Osvježavanje boida najviše košta jer se izračun novog smjera svakog boida izvodi na jednoj, zajedničkoj drevi. OO model zahtjeva mogućnost nasljeđivanja BoidControllera gdje se onda može nadjačati UpdateBoid() metoda. Zato je potrebno pozivati tu metodu na svakom objektu. Taj dio koda nije jednostavno paralelizirati jer Unity ne podržava višedretveno izvođenje ugrađenih funkcija koje je potrebno pozivati u UpdateBoid(). Iscrtavanje obavlja Unity (Camera.Render).

Sličan ispis za DOTS simulaciju je dan na **Slika *6-4***. Osvježavanje boida je značajno brže. Dodatno, iscrtavanje je brže jer Unity u pozadini koristi sisteme za iscrtavanje [13].

7. Zaključak

Simulacije napravljene u OO i DOTS pristupu se drastično razlikuju, ne samo po performansama koje nude nego po načinu na koji se izrađuju.

OO simulaciju je bilo mnogo lakše izgraditi. U središtu pozornosti su razredi koje sam već znao dobro modelirati. Objekte je bilo lagano modelirati jer se jednostavno preslikavaju iz Unity scene u programski kod. Na primjer, na objektima za stvaranje boida je jednostavna skripta koja na početku izvođenja stvori definirani broj boid objekata te nakon toga prestaje sa radom. Rad sa ComputeShaderom je također bio relativno jednostavan gdje je trebalo napisati sjenčar te kod za učitavanje podataka u sjenčar. Nije bilo potrebe za alociranje i de alociranjem podataka osim kod učitavanja podataka u sjenčar.

Pri izradi DOTS simulacije prvo je trebalo odrediti kakve podatke simulacija koristi. U ovom djelu se definiraju vrste komponenti i tipovi podataka koje spremaju. Komponente su bitne jer se sustavi pišu s obzirom na komponente, ako promijenimo strukturu komponenata i sustavi se mijenjaju. Određivanje komponenti je relativno jednostavno. Za boide su stavljeni Layer i Animal komponente kao ISharedComponentData jer segmentiraju podatke na prigodan način. Težak dio je pisanje raznih sustava. Objekt za stvaranje boida zahtjeva puno više posla nego što je to bilo potrebno u OO pristupu. Bilo je potrebno napraviti sustav za konverziju primjerka boida u entitet kako bi objekt za stvaranje boida imao dostupan entitet za stvaranje. Pisanje sustava za simulaciju boida je također bilo izazovno jer zahtjeva složenu pripremu podataka prije izračuna te niz zavisnosti kako bi se dobilo sigurno izvođenje. DOTS tehnologija je još uvijek u ranoj fazi razvoja, često izlaze nove verzije koje značajno mijenjaju mogućnosti te način izvedbe rješenja. U budućnosti bi trebalo postati lakše razvijati rješenja pomoću DOTS-a kada bude bolje integriran u Unity.

OO pristup će uvijek imati prednost fleksibilnosti. Lakše je uvesti promjene jer imamo moć proizvoljnog povezivanja objekata, te ne moramo brinuti o mnogim stvarima o kojima moramo u DOTS-u. To znači da je lakše izgrađivati prototipe i proširivati postojeća rješenja. Još jedna velika prednost OO pristupa je njegova popularnost, povijest i podrška (ne samu u Unityu nego općenito). Prednost DOTS-a, tj. podatkovne paradigme je u tome što nam omogućava pisanje koda koji je maksimalno prilagođen radu računala. To mora biti ključna stavka pri izradi rješenja. Unity je prepoznao koliko je bitno imati učinkovita rješenja te je zbog toga ponudio DOTS. Prije izrade rješenja bitno je analizirati podatke i odlučiti o pravilnom pristupu za izradu rješenja.

8. Literatura

[1] Acton, M. (2018). *Unity at GDC - A Data Oriented Approach to Using Component Systems*. Preuzeto 1.6.2020. iz youtube: https://www.youtube.com/watch?v=p65Yt20pw0g&feature=youtu.be&t=1446

[2] *Comparison of multi-paradigm programming languages*. (2020). Preuzeto 1.6.2020. iz wikipedia: https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

[3] *Frame rate*. (2020). Preuzeto 1.6.2020. iz wikipedia: https://en.wikipedia.org/wiki/Frame_rate

[4] Khronos Group. (2020). *Compute Shader*. Preuzeto 1.6.2020. iz khronos: https://www.khronos.org/opengl/wiki/Compute_Shader

[5] Llopis, N. (2009). *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)*. Preuzeto 1.6.2020. iz gamesfromwithin: http://gamesfromwithin.com/data-oriented-design

[6] Reynolds, C. W. (1987). *Flocks, Herds, and Schools: A Distributed Behavioral Model.* Preuzeto 1.6.2020. iz red3d: https://www.red3d.com/cwr/papers/1987/SIGGRAPH87.pdf

[7] Road to VR. (2019). *How to Tell if Your PC is VR Ready*. Preuzeto 1.6.2020. iz roadtovr: https://www.roadtovr.com/how-to-tell-pc-virtual-reality-vr-oculus-rift-htc-vive-steam-vr-compatibility-tool/

[8] Soomro, A. (2015). *Reduce Input Lag in PC Games: The Definitive Guide*. Preuzeto 1.6.2020. iz displaylag: https://displaylag.com/reduce-input-lag-in-pc-games-the-definitive-guide/

[9] StatsToDo. (2014). *StatsToDo : Combine Means and SDs Into One Group Program*. Preuzeto 1.6.2020. iz statstodo: https://www.statstodo.com/CombineMeansSDs_Pgm.php

[10] Tamasi, T. (2019). *Why Does High FPS Matter For Esports?* Preuzeto 1.6.2020. iz nvidia: https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/

[11] TIOBE. (2020). *TIOBE Index*. Preuzeto 1.6.2020. iz TIOBE: https://www.tiobe.com/tiobe-index/

[12] *Unity (game engine)*. (2020). Preuzeto 1.6.2020. iz wikipedia: https://en.wikipedia.org/wiki/Unity_(game_engine)

[13] Unity. (2020). *DOTS Hybrid Renderer*. (unity3d) Preuzeto 1.6.2020. iz unity3d: https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.5/manual/index.html

[14] Unity. (2020). *Entity Component System*. Preuzeto 1.6.2020. iz unity3d: https://docs.unity3d.com/Packages/com.unity.entities@0.11/manual/index.html

[15] Unity. (2020). *Games made in Unity*. Preuzeto 1.6.2020. iz unity3d: https://unity3d.com/games-made-with-unity

[16] Unity. (2020). *MonoBehaviour*. Preuzeto 1.6.2020. iz unity3d: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[17] Unity. (2020). *ScriptableObject*. Preuzeto 1.6.2020. iz unity3d: https://docs.unity3d.com/Manual/class-ScriptableObject.html

Ispitivanje povećanja performansi Unity sustava kroz podatkovno orijentirani tehnološki stog

Sažetak

Izrada složene simulacije predstavlja veliki izazov, pogotovo ako je ta simulacija interaktivna. Unity već dugi niz godina nudi alate za izradu interaktivnih simulacija te se najčešće koristi za izradu video igara. Kako se zahtjevi tržišta video igara mijenjaju s dolaskom novih tehnologija tako se stvara potreba za novim pristupom za izradu video igara. Uređaji za prikaz virtualne stvarnosti i prijenosni uređaji zahtijevaju učinkovita rješenja koja štede bateriju i procesorske cikluse. U svrhu rješavanja ovih problema Unity je ponudi novu DOTS tehnologiju. Cilj ovog rada je proučiti razliku između novog, DOTS pristupa koji je vođen podatkovno orijentiranom paradigmom te klasičnog pristupa koji je vođen objektno orijentiranom programskom paradigmom. Za potrebe usporedbe izgrađene su dvije simulacije boida, jedna koristi klasični pristup, druga koristi DOTS.

Ključne riječi: simulacija, boidi, Unity, DOTS, objektno orijentirana paradigma, podatkovno orijentirana paradigma, učinkovitost

Testing performance gain of Unity systems by using the Dana-Oriented Technology Stack

Abstract

Designing a complex simulation represent a great challenge, especially if the simulation is interactive. Unity, for quite some time now, has been offering tools for creating interactive simulations and has been mostly used for creating video games. With new, emerging technologies needs of the video game industry are changing. Virtual reality headsets and mobile devices require solutions that save battery power and processor cycles. Trying to solve these issues Unity has offered their new DOTS technology. The goal of this thesis is to compare the new, data-oriented DOTS design with the classical, object-oriented design. For this test two simulations were constructed, one using the classical approach, the other using DOTS.

Keywords: simulation, boids, Unity, DOTS, object-oriented paradigm, data-oriented paradigm, efficiency