SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 391

PRIKAZ VELIKIH VIRTUALNIH SVJETOVA

David Čemeljić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 391

PRIKAZ VELIKIH VIRTUALNIH SVJETOVA

David Čemeljić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 391

Pristupnik:	**David Čemeljić (0036523170)**
Studij:	Računarstvo
Profil:	Računalno inženjerstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Prikaz velikih virtualnih svjetova**

Opis zadatka:

Igre s velikim virtualnim otvorenim svijetom (engl. Open-World Game) predstavljaju u tehničkom smislu poseban izazov za implementaciju, posebice zbog veličine svjetova. Proučiti problematiku programske implementacije igara s otvorenim svjetovima, posebice uz korištenje koncepta programske arhitekture ECS (engl. Entity Component System), te mehanizme particioniranja razina svjetova (engl. Level). Razmotriti i razraditi problematiku učinkovitog unaprijednog ostvarivanja prikaza (engl. Forward Rendering) uz tako veliki broj 3D modela objekata u svijetu. Razraditi osnovne komponente fizikalnog pogona u ostvarenom okruženju. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje DirectX 12. Rezultate rada učiniti dostupnima putem weba. Radu priložiti algoritme, izvorni kôd i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

Uvod

Kako industrija video igara svakom godinom raste, zahtjevi igrača također rastu. Igrači traže sve veće, ljepše i kompleksnije igre, dok uprava i dioničari zahtijevaju brzu isporuku proizvoda. U ovakvim uvjetima najviše performance su na dnu liste prioriteta, isporučeni proizvod je pun grešaka i na kraju niti igrači niti dioničari nisu zadovoljni.

Glavni uzrok loših performanci i grešaka je loša podloga odnosno nedovoljno snažan i zastarjeli pokretač video igara (*engl. game engine*). Većina velikih pokretača ima vuče za sobom ogroman tehnički dug (*engl. tech debt*) koji uključuje kompatibilnost s vrlo starim verzijama projekata, mogućnost pokretanja na hardveru koji se ne koristi desetak ili više godina, ali i arhitekturu samog pokretača – jednom dizajniran pokretač teško je izmijeniti kad ga mnogo timova koristi godinama za svoje projekte. Uzmimo za primjer *Unreal Engine 5* koji sam po sebi nema velike arhitekturalne promjene, no velika većina timova još nije nadogradila svoje projekte zbog kompleksnosti nadogradnje i stvaranja niza novih grešaka. Nadogradnja s UE 4.27 na UE 5 za srednje velik projekt traje više od godinu dana, gdje se većina vremena potroši na popravljanje novih grešaka i popravljanje performanci iscrtavanja. S druge strane, moderniji pokretači imaju značajno bolje performance uz odličnu kvalitetu, no izgradnja pokretača traje nekoliko godina, što je većini timova preskupo.

Teško je ostvariti visoke performance ako je pokretač namijenjen za mali broj objekata i male scene – potrebno je promijeniti perspektivu i problemu pristupiti na drukčiji način. Moramo uzeti u obzir da će neka igra u budućnosti zahtijevati iznimno velike scene s iznimno velikim brojem objekata, zahtijevati velik broj sjenčara koje ne možemo sinkrono prevoditi, te ih moramo pažljivo koristiti kako ne bismo imali kratko zastajkivanje kod iscrtavanja (engl. *stutter*).

Cilj ovog rada je umanjiti problem performanci na razini arhitekture.

1. Programski pogon *Infinity Engine*

U sklopu ovog projekta razvijen je moderni pokretač video igara, bez ograničenja na broj entiteta što sa strane procesora, što sa strane grafičke kartice. U nastavku bit će opisani principi efikasne obrade podataka. Potrebno je ostvariti:

- efikasnu obradu velikog broja entiteta na procesoru
- efikasno iscrtavanje velikog broja modela na grafičkoj kartici
- efikasno osvjetljenje koje podržava velik broj svjetala u sceni
- asinkrono učitavanje scene, particioniranje scene, te asinkrono učitavanje ostalih resursa i asinkrono prevođenje sjenčara

Osim opisa same arhitekture bit će objašnjene podatkovne strukture koje su potrebne za ostvarivanje visokih performanci i boljeg korištenja priručne memorije procesora (*engl. cache*).

U okviru projekta ostvaren je i jednostavan sustav fizike koji se koristi za fizikalnu simulaciju i scenovne upite (*engl. scene queries*). Ovaj fizikalni sustav trenutno nije u potpunosti optimiziran, stoga će biti opisane buduće nadogradnje i trenutne limitacije.

Na kraju će biti opisane ostale komponente pokretača koje pojednostavljuju razvoj video igre u pokretaču.

Pokretač je napisan u jeziku C++23, a za iscrtavanje se koristi grafičko programsko sučelje *DirectX 12*.

1.1. Jezgra pokretača *Entity-Component-System* (ECS)

U ovom poglavlju bit će objašnjena jezgra pokretača – obrada velikog broja entiteta na procesoru. Bit će objašnjene mane klasičnog pristupa zbog kojih je vrlo teško ostvariti visoke performance za velik broj objekata.

Entity-Component-System (u ostatku teksta *ECS*) je princip obrade entiteta orijentiran prema podatcima, a ne na objekte, što značajno pojednostavljuje višedretvenu obradu.

1.1.1. Usporedba ECS-a i klasičnog pristupa

Klasični pristup koji koristi većina pokretača zove se *Entity-Component*, skraćeno *EC*. Entitet je objekt, kompozicija komponenata, te enkapsulira i podatke i ponašanje. I objekt i komponente mogu imati i podatke i ponašanje, jedina razlika je u tome što komponente ne mogu postojati same u svijetu, moraju biti u vlasništvu nekog objekta.

Ovakav princip obično je ostvaren baznom klasom objekta – u *Unreal Engineu* to je klasa *Actor* koja ima velik broj virtualnih metoda za nadogradnju osnovnog ponašanja. Klasa *Actor* je sama po sebi velika zato što treba omogućiti implementaciju ponašanja za sve slučajeve, poput serijalizacije i replikacije. U većini igara postoje privremeni objekti koji se neće serijalizirati niti replicirati drugim klijentima kroz mrežu, no programer ne može izbaciti te dijelove koda i tako smanjiti veličinu objekata – to je limitacija arhitekture.

Nastavno na *Unreal Engine*, bazna klasa za sve objekte je *UObject*, koja je sama po sebi velika iz istog razloga kao i *Actor*, *UObject* mora sadržavati metode i podatke za sve slučajeve. *UObject* nema mogućnosti replikacije, ali implementira metode kojima se replikacija dobrim dijelom i ostvaruje.

Veličina objekta je problem koji se većinom zanemaruje, i to je uredu za male projekte i mali broj objekata. Problem s velikim objektima je to što ih je skupo kreirati, pa često treba zaobilaziti rješenja namijenjena od strane pokretača. Drugi problem je to što zauzimaju više radne memorije, što znači da i zauzimaju više priručne memorije procesora i tako smanjuju performance.

Razdvajanje bazne klase na dijelove koji sadržavaju pojedine mogućnosti, npr. osnovnu logiku, logiku koju programer koji razvija igru unutar pokretača (u ostatku teksta *korisnik*) može nadograđivati, serijalizaciju i replikaciju je zahtjevan zadatak te se ne isplati bez drugih arhitekturalnih promjena. Problem koji nastane kod pokušaja razdvajanja: što ako želimo

objekt koji se može i serijalizirati i replicirati kroz mrežu, a i dalje želimo dio ponašanja u komponentama za ponovnu uporabu? Razdvajanje bazne klase zahtijeva dijamantno nasljeđivanje koje sa sobom donosi niz problema, uključujući i razdvajanje objekta na više dijelova u memoriji zbog virtualnog nasljeđivanja. Kod dijamantnog nasljeđivanja imamo dvije ili više instance bazne klase unutar jednog objekta – virtualno nasljeđivanje riješi taj problem tako da dijeljeni dio objekta alocira na drugom mjestu, tako da objekt ima dva dijela i gotovo sigurno zahtijeva barem dva čitanja iz memorije kako bi se našao u priručnoj memoriji. Na ovaj način bismo vjerojatno samo izgubili performance, dakle, nema sretnog rješenja.

Treći i najbitniji problem koji rješava ovaj projekt, je organizacija podataka za višedretvenu obradu. *EC* princip koristi polimorfnu klasu za objekte koji sadržavaju komponente, također instance polimorfne klase. Pitanje je kako više objekata obraditi paralelno. U teoriji, mogli bismo komponente dijelom procesirati u paraleli, uz pažljivu sinkronizaciju komunikacije između komponenata – ponekad jedna komponenta treba promijeniti nešto na drugoj komponenti. No kako sinkronizirati pristup podacima u samom objektu? Više komponenata jednog objekta može pristupati podacima svojeg objekta – moramo sinkronizirati pristup, a sinkronizacija je skupa i nepredvidiva. Sinkronizacija monitorom (engl. *mutex*) je potencijalno iznimno skupa ako dretva čeka na monitor dovoljno dugo zato što će tada dretva biti blokirana. Moderni operacijski sustavi zaključavanje monitora implementiraju kratkim radnim čekanjem, ali i ispitivanje je li moguće zaključati monitor je jeftino. Ako neku dretvu blokira monitor, ostavljena je na milost i nemilost operacijskog sustava i više nemamo konzistentan broj sličica po sekundi. S druge strane, možemo koristiti atomarne operacije za sinkronizaciju, poput mehanizma zaključavanja radnim čekanjem (engl. *spinlock*). Problem kod toga je što ako dvije dretve pokušaju pristupiti objektu, jedna dobije pristup podatcima i započne dugu operaciju, druga će radno čekati cijelo vrijeme – cijena operacije prve dretve je sad dvostruko veća, a nismo ništa postigli, zato što je obrada sekvencijalna, kao da se izvršava na jednoj dretvi. Mogli bismo samo dijelove objekta zaštititi, tako da više dretvi može pristupiti objektu, ali što ako jedna dretva treba pristupiti više dijelova objekta – sada imamo još više sinkronizacije, koristimo značajno više energije (više jezgri) uz malo veće performance.

Višedretvena obrada objekata i komponenata u *EC* sustavu nema sretno rješenje.

U ovom radu koristi se princip *Entity-Component-System*, u daljnjem tekstu *ECS*. Za razliku od *EC* pristupa, *ECS* je orijentiran na podatke, što rješava niz problema, ali zahtijeva prilagodbu programera koji su velikom većinom navikli na objektno orijentirani pristup. Razvijeni ECS je vrlo intuitivno koristiti, te se koristi višedretvenost bez intervencije korisnika.

Kao što ime govori, u *ECS* se sastoji od tri dijela, od entiteta, komponenata i sustava.

1.1.2. Entitet

Entitet (engl. *Entity*) je minimalna struktura, kompozicija komponenata i eventualno identifikator kao cijeli broj. *ECS* princip nije strogo definiran, tako da postoje razne implementacije iste ideje. Entiteti ne smiju implementirati ponašanje, no nitko nas ne sprječava da implementiramo pomoćne funkcije radi jednostavnosti korištenja i čišćeg koda.

1.1.3. Komponenta

Komponenta (engl. *Component*)je također minimalna struktura koja ne smije sadržavati ponašanje, ali ovdje izbjegavamo i pomoćne funkcije. Ideja je da su komponente isključivo podatci koji se obrađuju, a promjena nekih podataka može zahtijevati promjene podataka na drugim mjestima izvan komponente. Više o tome bit će objašnjeno kasnije u tekstu.

1.1.4. Sustav

Sustavi (engl. *System*) jedini smiju imati ponašanje u smislu logike igre. Sustavi imaju najmanje limitacija, mogu biti i polimorfne klase, imati i podatke i ponašanje.

Sustavi obrađuju entitete koji sadržavaju komponente određenog tipa. Odvojili smo ponašanje od podataka.

Sustavi obrađuju podatke u komponentama, no promjene u tim podatcima mogu zahtijevati promjenu nekih „globalnih“ podataka. Ti podatci nisu globalni, samo se tiču određenog dijela scene. Više u poglavlju o particioniranju svjetova. Na primjer, promjena 3D modela na komponenti nekog entiteta zahtijeva kreiranje nove instance u memoriji grafičke kartice. Ovo je glavni razlog zašto komponente generalno nemaju pomoćne funkcije – ako želimo promijeniti 3D model, sustav u kojem se trenutno „nalazimo“ mora signalizirati sustavu koji se bavi 3D modelima da promijeni model i pošalje nove podatke na grafičku karticu.

Limitiranje ponašanja na samo jednu klasu značajno pojednostavljuje rješenja problema *EC* principa. Organizacija podataka u ECS-u rješava problem razdvajanja ponašanja kako bismo dobili manje objekte – ne treba nam višestruko nasljeđivanje kad nemamo objekte u smislu objektno orijentiranog programiranja.

Sustavi obrađuju samo određene entitete koji imaju određene tipove podataka. Ova činjenica drastično pojednostavljuje višedretvenu obradu entiteta. Podaci su već odvojeni na komponente, dok smo u EC pristupu smo trebali štititi dijelove objekta odvojenom sinkronizacijom. Arhitektura pojednostavljuje programerov posao. U EC pristupu smo trebali za svaki dio objekta imati poseban mehanizam sinkronizacije.

1.1.5. Mehanizam arhetipova

Arhetip je način dodavanja identiteta entitetima. Entitet je kompozicija instanci komponenata, arhetip je kompozicija tipova komponenata nekog entiteta.

Arhetip služi za efikasno traženje indeksa pojedinog tipa komponenata, za sinkronizaciju i za sortiranje entiteta u liste. Više o listama u posebnom poglavlju.

Veličina ovog tipa može biti značajno veća od entiteta i komponenata zato što će arhetipova biti zanemarivo malo u odnosu na entitete i komponente. Naravno, ako je arhetip velik, treba ga pažljivije koristiti i spremati, umjesto da se kreira novi.

Moguće je imati dva tipa arhetipa, veliki koji sadržava mapu tip-indeks, mapu ime-indeks i listu tipova komponenata i minimalni koji sadržava samo listu tipova komponenata. Uz posebnu strukturu za liste (poput klase *DArray* opisane u poglavlju o podatkovnim strukturama), kopiranje minimalnog arhetipa nije skupo. U okviru ovog projekta implementiran je samo veliki arhetip uz njegovo pažljivo korištenje, dok je mali arhetip ostavljen za buduće nadogradnje.

Arhetip uz tip komponente sadrži informaciju o vrsti pristupa komponenti – hoćemo li samo čitati iz komponente ili ćemo i čitati i pisati. Ovo je iznimno korisno za sinkronizaciju.

1.1.6. Sinkronizacija

U ECS-u sinkronizacija je praktički trivijalna u odnosu na EC pristup. S obzirom na to da su podatci odvojeni od ponašanja, i podatci odvojeni međusobno, neki sustavi se mogu pokretati paralelno bez međusobne sinkronizacije. Sustavi se mogu pokretati paralelno ako je ispunjen barem jedan od ovih uvjeta:

- obrađuju različite tipove komponenata (nema presjeka)
- čitaju iste tipove komponenata, zapisuju u različite tipove komponenata

Za određivanje uvjeta paralelizacije koriste se arhetipovi. Arhetip zna kojim komponentama će sustav pristupati i na koji način – imamo sve potrebne informacije. Ispitivanjem presjeka arhetipa, te ispitivanjem tipa pristupa pojedinoj komponenti u presjeku, možemo odrediti mogu li se sustavi pokretati paralelno.

Sada kada znamo mogu li se sustavi pokretati paralelno, možemo kreirati usmjereni aciklički graf zadataka. Svaki sustav je jedan zadatak – pokrećemo metodu `Tick(double deltaTime)` nad pojedinim sustavom. Krećemo od korijena, što je početni zadatak koji nema pridijeljen sustav. Kad dodajemo novi zadatak, krećemo se od korijena do završnog zadatka. Nijedan zadatak se ne može pokretati paralelno s korijenskim zadatkom, tako da prvo ispitujemo može li se zadatak pokretati paralelno sa zadacima koji su djeca korijenskog zadatka. Ako je to istina, novom zadatku postavljamo korijen kao povezani zadatak koji se mora odraditi prije, te određujemo zadatke koji ovise o novom zadatku. To su zadatci koji se sigurno nalaze razinu ispod trenutne razine (one u koju smo smjestili novi zadatak). Moramo proći po ostatku grafa i dodati novi zadatak u listu svih zadataka koji se ne mogu pokretati paralelno s njim. Ako takvih zadataka nema, spajamo novi zadatak sa završnim zadatkom.

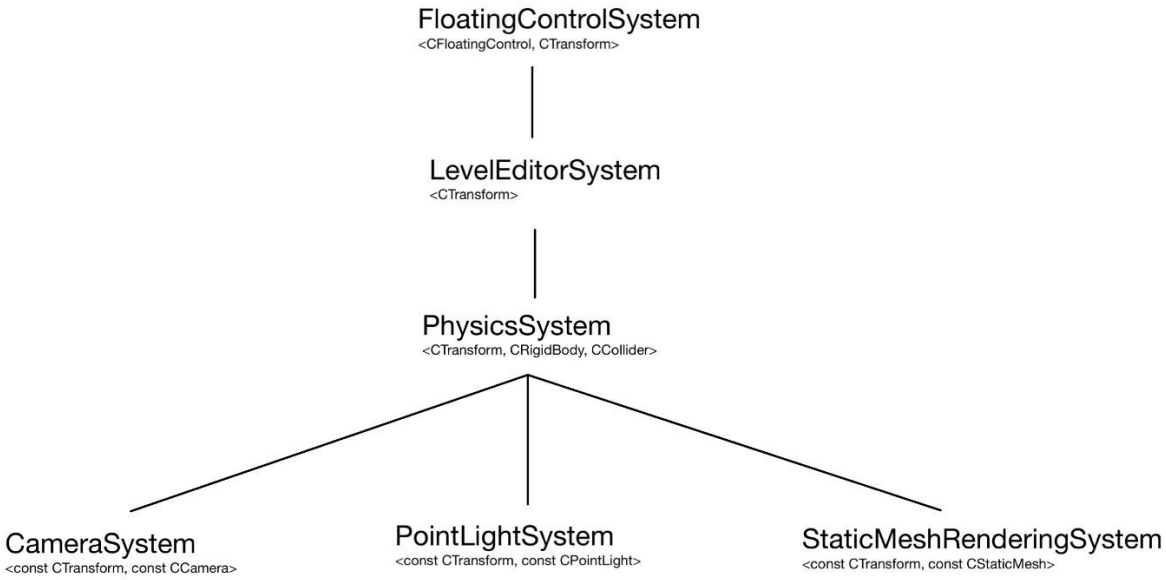
Ovakav graf se mijenja samo kad se u svijet doda novi sustav, ostatak vremena raspored je fiksan.

Kreiranje dretvi je skupo i gubili bismo previše performanci kad bismo kreirali novu dretvu za svaki korak u simulaciji te za svaki zadatak. Zato je implementirana struktura koja jednom kreira određen broj dretvi, daje im zadatke i ponovno ih koristi kad odrade zadatke, odnosno bazen dretvi (engl. *thread pool*). Unutar pokretača postoje dva bazena, glavni bazen za visokoprioritetne zadatke koji sadržava broj dretvi određen brojem jezgri procesora, te bazen za niskoprioritetne zadatke s velikim brojem dretvi koje će većinu vremena biti blokirane zbog npr. čekanja na disk. Za obradu sustava koristi se glavni bazen tako da imamo veću kontrolu nad sinkronizacijom – ne želimo preopteretiti operacijski sustav s dretvama koje se ne moraju paralelno izvršavati zato što će onda operacijski sustav češće mijenjati dretve da ne bi bile izgladnjivane. Mijenjanje konteksta samo po sebi nije besplatno, no postoji često zanemaren utjecaj česte promjene konteksta – dretve obrađuju različite podatke, i priručna memorija nije beskonačna. Ovo znači da jedna dretva „učita“ podatke u priručnu memoriju,

obradi mali dio, bude blokirana zbog izgladnjivanja druge dretve, promijeni se kontekst i druga dretva krene „učitavati“ svoje podatke u priručnu memoriju. Ovako samo gubimo performance.

Kada dođe vrijeme za simulirati jedan korak u vremenu, glavna dretva pokreće početni zadatak koji odmah završava (nema sustava) te predaje zadatke djecu u bazen dretvi. Po završetku zadatka, dretve javljaju podzadacima da su gotove s poslom. Kad zadnji zadatak o kojem ovisi neki podzadatak bude gotov, dretva ubacuje podzadatak u bazen dretvi.

Simulacija koraka završava kad svi zadaci o kojima je završni zadatak ovisan budu završeni. Tu preuzima glavna dretva koja može poslati naredbe za iscrtavanje na grafičku karticu te prikazati trenutni okvir itd. Primjer jednostavnog grafa zadataka prikazan je slikom Slika 1.1.



Slika 1.1 Primjer jednostavnog grafa zadataka

Ovo je vrlo elegantno rješenje dva velika problema – paralelne obrade podataka i intuitivnog korištenja. Korisnik ne mora apsolutno ništa napraviti po pitanju sinkronizacije, ona se događa automatski. Dapače, korisnički kod nema nijedan mehanizam sinkronizacije – nijedan monitor niti eksplicitno radno čekanje. Korisnik samo mora biti svjestan da se sustavi mogu izvršavati paralelno, tako da za komunikaciju između sustava mora koristiti događaje, te ne smije pristupati komponentama kojima sustav nema pristup. Unutar ovog projekta, pristupanje drugim komponentama je zabranjeno metaprogramiranjem – pokušaj dohvaćanja komponente za čitanje ili pisanje unutar sustava koji nije deklarirao da će

pristupati tom tipu komponente na određeni način rezultira greškom u prevođenju. Ovakve značajke pokretača značajno olakšavaju snalaženje korisniku koji je tek počeo koristiti pokretač, ali i iskusnim korisnicima unutar velikih projekata – imamo provjeru za vrijeme prevođenja koja nam ne dozvoljava da radimo sukob pristupa podacima (engl. *data race*). Sukob pristupa podacima je teško uhvatiti za vrijeme izvođenja zbog nedeterminističkog izvođenja unutar našeg programa (skup sustava se može mijenjati za vrijeme izvođenja) i nedeterminističkog rasporeda dretvi unutar operacijskog sustava na koji ne možemo utjecati. Više o provjeri za vrijeme prevođenja u poglavlju o implementaciji.

Korisno je dodati mehanizam zahtjeva za redoslijed sustava unutar ovakvog raspoređivača zadataka. Idealno, želimo da igrač na ekranu vidi najnovije promjene – sustav koji se bavi osvježavanjem podataka na grafičkoj kartici bi se trebao izvoditi nakon svih ostalih sustava koji mijenjaju te podatke – npr. lokaciju entiteta i podatke o 3D modelu koji želimo iscrtati, kao i materijal i podatke instance materijala. Ovaj mehanizam nije komplicirano implementirati, no u okviru ovog projekta ostavljen je za buduće nadogradnje.

1.1.7. Svijet

Četvrta komponenta *ECS*-a koja se ne spominje u samom imenu je svijet (engl. *World*). Svijet sadrži entitete i njihove komponente, te pokreće sustave. Na prvu se ne čini kao bitna komponenta, tako nešto ćemo ionako morati implementirati u *ECS*-u, no svijet u *ECS*-u ima dodatne odgovornosti.

Naime, na svijet možemo gledati kao jednu cjelinu, skup entiteta koji ne interagira s drugim entitetima, što znači da se svjetovi mogu obrađivati paralelno. U prethodnom poglavlju opisano je raspoređivanje sustava na procesoru modeliranjem ovisnosti između sustava. Dobiveni raspored ne mora nužno popuniti sve jezgre procesora – ako imamo nekoliko sustava koji zapisuju u isti skup komponenata, oni se moraju izvoditi sekvencijalno, što znači da ne iskorištavamo cijeli procesor. Particioniranjem scene na više svjetova umanjujemo taj problem, a ako imamo više svjetova nego jezgri procesora, iskoristit ćemo sve jezgre.

Svaki svijet mora imati odvojene podatkovne strukture za spremanje entiteta i komponenata zato što nemamo nikakvih prednosti kod spremanja svih entiteta u jednu strukturu, pristup bi trebalo sinkronizirati na čemu bismo gubili performance. Pristup tehnički možemo raspodijeliti kao kod sustava, no onda ne bismo u potpunosti paralelno izvoditi svjetove. Treća stvar je problem priručne memorije i lažnog dijeljenja (engl. *false sharing*) koji može

poništiti gotovo sav dobitak performanci kod višedretvenosti. Lažno dijeljenje se događa kada dvije ili više dretvi pristupa istoj liniji priručne memorije (engl. *cache line*). Te dretve pristupaju različitim podatcima – pristup ne moramo sinkronizirati, problem je u hardveru odnosno priručnoj memoriji. Kada jedna jezgra zapiše podatak u liniju priručne memorije, sve druge jezgre koje čitaju iz iste linije će trebati ponovno dohvatiti tu liniju iz radne memorije, unatoč tome što se podatak koji one čitaju nije promijenio. Klasični primjer lažnog dijeljenja je zbrajanje niza brojeva s više dretvi, tako da dretve spremaju svoj rezultat u polje. Svaka dretva ima jedan element u polju, dakle pristupa samo jednom podatku onoliko puta koliko joj je dodijeljeno elemenata u ulaznom polju brojeva - pristup ne treba sinkronizirati. Recimo da želimo zbrojiti brojeve od 1 do 10000 i podijelimo to na 10 dretvi, gdje svaka mora zbrojiti redom 1000 brojeva u ulaznom polju. Očekujemo ubrzanje od 10 puta u odnosu na sekvencijalno zbrajanje jednom dretvom, no rezultat mjerenja će nas neugodno iznenaditi – moguće je da ćemo imati i lošije performance. Rješenje ovog problema je da svaka dretva ima lokalnu varijablu koja predstavlja zbroj, te da kad zbroji sve svoje elemente zapiše taj rezultat u polje rezultata.

Isti koncept primjenjujemo u svjetovima, razdvojimo podatke tako da se lažno dijeljenje ne može dogoditi. Svjetovi generalno ne trebaju međusobno komunicirati, jedini slučaj je migracija entiteta. Komunikacija se rješava na isti način kao i između sustava – događajima.

1.1.8. Način spremanja komponenata

Komponente su spremljene u svojim svjetovima. Komponenti se može pristupiti samo iz njezinog svijeta zbog particioniranja i jednostavne sinkronizacije.

S obzirom na to da se komponentama pristupa paralelno, moramo paziti na lažno dijeljenje. Naivan pristup je svaku komponentu zasebno alocirati na gomili, što nije prihvatljivo zato što bi svaki pristup nekoj komponenti bio promašaj priručne memorije i čitanje iz radne memorije. Ovakav pristup ima dva problema, čitanje je sporo, ali i kreiranje novih komponenata je skupo zato što zahtijeva alokaciju odnosno poziv operacijskog sustava.

Rješenje za oba problema je podatkovna struktura koja alocira velike blokove memorije za jedan tip podataka, kad se on popuni, alocira se novi, ali podaci iz prvog bloka se ne kopiraju. Cijena jedne velike alokacije je značajno manja od velikog broja malih alokacija za jednaku količinu memorije. Ovakva struktura amortizira cijenu alokacije, te je kreiranje nove komponente jeftino, kao i uništavanje. Razdvajanjem komponenti po tipu rješavamo i

problem lažnog dijeljenja. Implementacija ovakve strukture je klasa `BucketArray` opisana u poglavlju o podatkovnim strukturama.

Primijenit ćemo ovaj princip za spremanje komponenata. Svijet može imati varijabilan broj tipova komponenata, tako da za svaki tip moramo kreirati novu instancu klase `BucketArray`. Inicijalno, struktura će biti prazna, `BucketArray` ćemo kreirati po potrebi. Prednost ovog pristupa je da ne alociramo previše memorije na početku, što bi uzrokovalo sporo pokretanje igre. Ni u kojem trenutku ne možemo predvidjeti koliko komponenata će nam trebati, zato što je igra zapravo simulacija, jedini način da saznamo kakvo će biti buduće stanje je da simuliramo dovoljno koraka. Nedostatak je to što će zahtjev za kreiranje komponente novog tipa biti skup zbog inicijalne alokacije `BucketArray`-a, što može uzrokovati nestabilan broj iteracija po sekundi (engl. *tick rate*). Cijena inicijalne alokacije `BucketArray`-a se može sakriti izvedbom na pomoćnoj dretvi, kao i cijena naknadnih alokacija blokova.

Implementacija ovakve strukture unutar ovog projekta je klasa `ObjectTypeMap` koja nije nužno vezana za *ECS*, već se može koristiti za svaki reflektirani tip. Refleksija unutar ovog projekta trenutno ima podršku samo za klase izvedene iz klase `Object`, no u budućnosti će biti implementirana generička refleksija, za bilo koju korijensku klasu i za obične strukture. `ObjectTypeMap` mapira tip na `BucketArray`, gdje se tipom smatra klasa `Type` koja sadržava razne informacije o klasi. Više u poglavlju o refleksiji.

1.1.9. Liste entiteta i upiti

Sustavi obrađuju entitete koji sadržavaju određene komponente, odnosno sustav će procesirati entitet ako postoji presjek između arhetipa sustava i arhetipa entiteta.

Entitete želimo spremati tako da im je pristup optimalan, pod uvjetom da će se obrađivati velik broj entiteta pojedinog arhetipa te da entiteti ne trebaju biti sortirani ni po čemu osim po arhetipu.

Najbolje performance imat ćemo ako entitete spremimo slijedno, zato što ćemo po njima iterirati veliku većinu vremena. Ne želimo spremiti sve entitete u jednu listu iz istog razloga zašto ne želimo spremati komponente u jednu listu – lažno dijeljenje, ali i zato što ne znamo unaprijed broj entiteta nekog arhetipa.

Iz tog razloga entitete spremamo u liste, gdje svaka lista ima entitete nekog arhetipa. Liste su implementirane kao `BucketArray`, isto kao i komponente, samo što je organizacija listi značajno kompliciranija. Kod komponenata nas zanima samo podjela po tipu, pa je mapa dovoljna, no kod entiteta imamo veće zahtjeve – želimo pretraživati entitete prema komponentama koje imaju. Naglasak je na tome da entitete većinu vremena nećemo pretraživati po samom *arhetipu*, nego po *podskupu arhetipa*. Na primjer, imamo scenu u kojoj imamo nekoliko entiteta koji imaju komponente `CTransform` i `CStaticMesh`, to su jednostavni entiteti koji se koriste za iscrtavanje 3D modela na nekoj lokaciji, i nekoliko entiteta s komponentama `CTransform`, `CStaticMesh` i `CPathfinding`. Sustav koji osvježava podatke o instancama 3D modela na grafičkoj kartici obrađuje entitete s komponentama `CTransform` i `CStaticMesh`. Upit za taj sustav treba vratiti obje liste, a sustav će kasnije iterirati prvo po jednoj listi, pa po drugoj. U slučaju kad imamo mali broj arhetipova, a time i mali broj lista entiteta, možemo koristiti mapu arhetip-lista, a kod upita iterirati po mapi i ispitivati presjek. Zanemarit ćemo činjenicu da je iteriranje po mapi u najmanju ruku upitno.

Ovaj projekt cilja na velik broj entiteta i velik broj arhetipova, te navedena ideja nije dovoljno dobra.

Kreirat ćemo usmjereni aciklički graf listi entiteta ovisno o njihovom arhetipu. Korijen grafa je prazan čvor, čvorovi djeca kreću od najjednostavnijih arhetipova. Njihova djeca imaju veće arhetipove. Listovi imaju najkompleksnije arhetipove.

Upiti nad ovakvim grafom kreću se od korijena prema listovima, ovisno o presjeku arhetipova. Za razliku od mape, ovdje ćemo ispitati samo liste koje stvarno mogu sadržavati traženi arhetip, kretanjem po grafu biramo samo djecu koja imaju presjek s traženim arhetipom. Kad naiđemo na listu čije nijedno dijete nema presjek, znamo da nema smisla nastaviti ispitivati to podstablo zato što sigurno ne sadržava kompatibilan arhetip.

Ovakav graf nudi maksimalne performance upita, no nije ga trivijalno implementirati. Dodavanje novog arhetipa u mali graf je relativno jeftino, no cijena eksplodira za veliki graf. U idealnom slučaju, dodavanje novih čvorova bi se trebalo izvoditi na pomoćnoj dretvi.

Algoritam dodavanja novih čvorova je sljedeći:

- dok nismo pronašli sve komponente arhetipa
- pronađi čvor s najsličnijim arhetipom ostatku arhetipa koji dodajemo

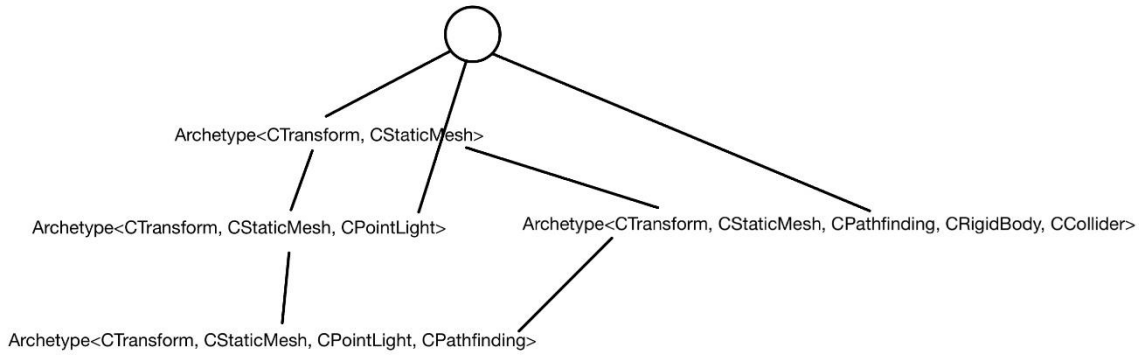
- ako najsličniji čvor ne postoji, spoji novi čvor s korijenskim
- ako je čvorov arhetip nadskup arhetipa novog čvora, novi čvor treba biti iznad najsličnijeg čvora u grafu, odnosno njegov roditelj
    - novom čvoru dodaj najsličniji čvor kao dijete
    - iteriraj po roditeljima najsličnijeg čvora, ako čvor ima presjek s novim, odspoji taj čvor i najsličniji čvor, dodaj novi čvor između njih
- ako je čvorov arhetip podskup arhetipa novog čvora, novi čvor treba biti ispod najsličnijeg čvora u grafu, odnosno njegovo dijete
    - najsličnijem čvoru dodaj novi čvor kao dijete
    - iteriraj po djeci najsličnijeg čvora, ako čvor ima presjek s novim, odspoji taj čvor i najsličniji čvor, dodaj novi čvor između njih

Ovim procesom ćemo kreirati novi čvor i spojiti ga s ostalim čvorovima koji su izravni nadskup i podskup tog arhetipa. Novi čvor ne treba spojiti sa svim čvorovima s kojima ima presjek zato što bismo tako značajno usporili pretraživanje grafa – neka podstabla bismo provjerili više puta.

Iz algoritma je vidljivo da cijena dodavanja eksplodira za velike grafove, zato što novi čvor trebamo spojiti s velikim brojem čvorova. Moramo to učiniti zato što do jednog arhetipa postoji više različitih putova od korijena. Odabir puta ovisi o upitu koji ne možemo predvidjeti. Kad ne bismo spojili novi čvor sa svim izravnim podskupima i nadskupima, maknuli bismo neke putove do tog čvora, te neki upiti ne bi pronašli taj čvor.

Nakon što dodamo novi arhetip, moramo osvježiti upite sustava. Osvježavamo upite samo onih sustava čiji arhetip ima presjek s novim arhetipom. Isto tako, kad dodamo novi sustav u raspored, trebamo mu osvježiti upit.

Implementacija ovakvog grafa je klasa `EntityListGraph`, a upiti su implementirani klasom `ECSQuery`. Slika 1.2 prikazuje jednostavni graf arhetipova. Klasa `EntityListGraph` kreira ovakvu strukturu.



Slika 1.2 Primjer jednostavnog grafa arhetipova

1.1.10. Događaji

Svrha događaja je standardizirati i pojednostaviti komunikaciju između sustava. S obzirom na to da se sustavi mogu izvoditi paralelno, komunikacija mora biti sinkronizirana. Ideja je sakriti sinkronizaciju od korisnika, što će ubrzati razvoj igre, ali i smanjiti broj grešaka. Događaji su zapravo agregacije entiteta nad kojima se dogodila neka operacija. Na primjer, entitet se sudario s nekim drugim entitetom unutar fizikalne simulacije – neke sustave zanima ova informacija, na primjer, sustav koji prati štetu želi nanijeti štetu entitetu kad se sudari, ovisno o brzini sudara i nekim drugim parametrima.

Kad bi se sustavi izvodili sekvencijalno, na jednoj dretvi, mogli bismo koristiti delegate. U okviru ovog projekta implementirani su delegati na koje se može registrirati proizvoljan broj funkcija. Implementacija opisanih delegata nalazi se u klasi `MulticastDelegate`.

Glavna razlika delegata i događaja je to što se delegati izvršavaju odmah na pozivu metode `Broadcast(Args&&…)`, na istoj dretvi, dok se događaji izvršavaju odgođeno, kad sustav odluči procesirati događaje.

S obzirom na to da se događaji mogu događati često, za velik broj entiteta, događaji spremaju entitete u liste, te se koristi mapa za mapiranje arhetip-lista. Nije potrebno kreirati graf zato što nećemo izvršavati upite nad događajem, već iterirati po svim listama. Mapa se koristi samo za indeksiranje, liste su spremljene u listu po kojoj možemo efikasno iterirati.

Implementacija događaja nalazi se u klasi `Event`, parametriziranoj po arhetipu i parametrima specifičnima za situaciju, na primjer, brzina sudara, lokacija sudara, normala itd. Arhetip se koristi za određivanje koje liste entiteta nam trebaju, odnosno sustav, kad mu

se osvježi upit, mora osvježiti liste svojih događaja. Ovo se događa automatski kroz sustav refleksije, sustav zna koje događaje ima i osvježi ih.

Za implementaciju listi ovdje se koristi lista sigurna za korištenje kroz više dretvi, specifično lista bez zaključavanja (engl. *lock-free queue*). U okviru ovog projekta koristi se vanjska implementacija takve liste, odnosno *moodycamel::ConcurrentQueue* [1]. Ovo je MPMC lista, odnosno u nju može pisati više proizvođača i čitati više potrošača istovremeno (engl. *multi-producer multi-consumer queue*). Implementacija struktura bez zaključavanja je jedan od najkompliciranijih zadataka u programiranju i često se ne isplati niti pokušavati zbog raznih problema koji se mogu dogoditi kod sinkronizacije. Osim sinkronizacije, želimo što veće performance dodavanja i micanja elemenata. Navedena implementacija je vrlo popularna s razlogom, ima visoke performance i detaljno je testirana na sukobe pristupa podacima.

Druga struktura vezana za događaje je `EventDispatcher`. Ta struktura služi za višestruko signaliziranje događaja, odnosno signaliziranje više događaja na različitim sustavima. Na ovu strukturu se registriraju događaji koji se mogu promatrati kao kanali prema drugim sustavima. Kasnije ćemo ubaciti entitet u `EventDispatcher` koji će ga dodati u sve registrirane događaje, i tako će više sustava dobiti signal da se dogodio neki događaj nad specifičnim entitetom. Na primjer, neki sustavi mogu mijenjati lokaciju entiteta. Dohvaćanje komponente `CTransform` će signalizirati sustavu za 3D modele i sustavu za svjetla da se entitet pomaknuo koji će kasnije osvježiti podatke na grafičkoj kartici.

`EventDispatcher` može imati bilo koji sustav, ali i klasa World. Upravo tako je i implementirano signaliziranje promjena lokacije drugim sustavima. Metaprogramiranjem možemo utvrditi trebamo li signalizirati `EventDispatcher`, odnosno postoji standardizirani način za označavanje da komponenta zahtijeva signal za vrijeme prevođenja kako ne bismo trošili cikluse procesora za vrijeme izvođenja za komponente koje ne zahtijevaju takav signal.

1.1.11. Implementacijski detalji

U ovom poglavlju bit će objašnjeni neki bitniji implementacijski detalji.

Komponente su trenutno implementirane kao objekti, odnosno klasa `Component` nasljeđuje klasu `Object`. Razlog tome je što refleksija unutar ovog projekta trenutno

podržava samo klase izvedene iz klase `Object` – u budućnosti, `Component` neće biti klasa nego obična struktura, ali reflektirana. Refleksija nam je potrebna zato što želimo u uređivaču kreirati predloške entiteta koji se mogu stvoriti u svijetu. Imena komponenata bi trebala počinjati velikim slovom „C“. Ovakva nomenklatura rješava problem koje neke jednostavne komponente imaju. Primjer problematične komponente je CTransform koja je omotač oko klase Transform, klase koja predstavlja punu 3D transformaciju, lokaciju, rotaciju, skaliranje te pokazivač na transformaciju roditelja. Dvije klase se ne mogu zvati identično, tako da je standardizacija nomenklature komponenata vrlo elegantno rješenje. Osim toga, programer će odmah primijetiti da je neka klasa komponenta čitanjem samog imena klase. Slika Slika 1.3 prikazuje implementaciju jednostavne bazne klase `Component`.

```
REFLECTED()
class Component : public Object
{
    GENERATED()

public:
    void SetName(Name value, PassKey<World>)
    {
        _componentName = value;
    }

    Name GetName() const
    {
        return _componentName;
    }

private:
    PROPERTY(Edit, Serialize, DisplayName = "Name")
    Name _componentName;
};
```

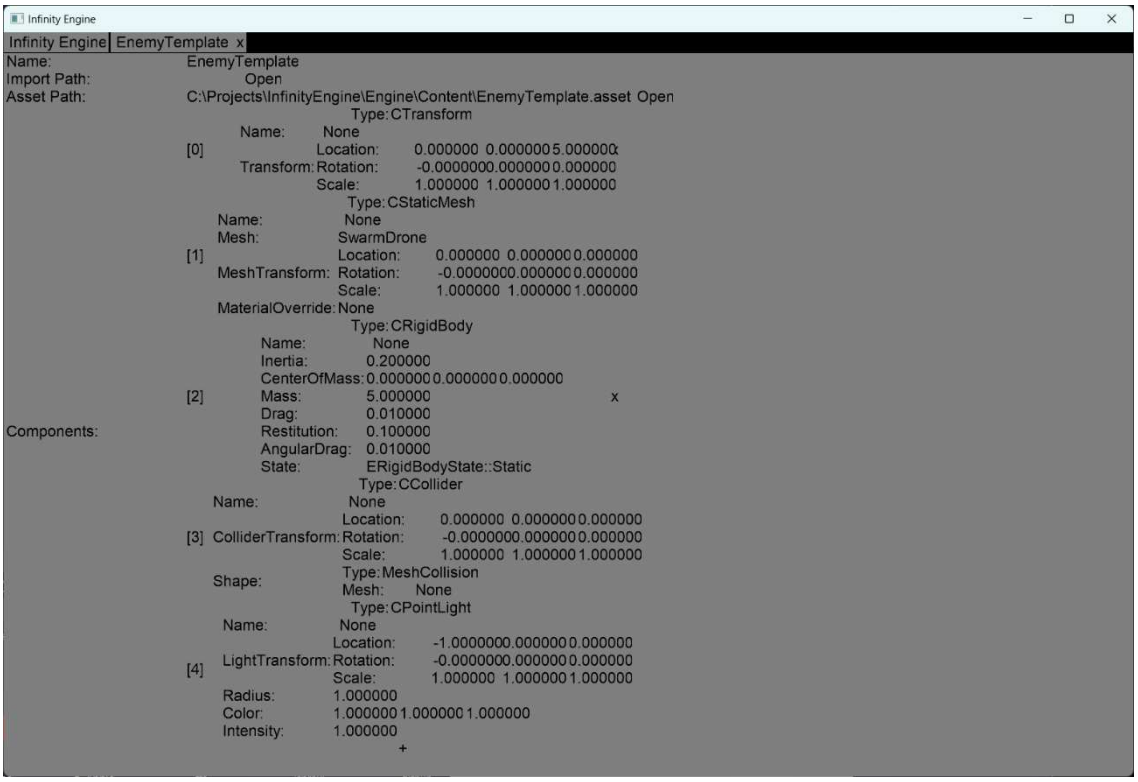
Slika 1.3 Klasa *Component*

Predlošci entiteta implementirani su u klasi `EntityTemplate`. `EntityTemplate` je resurs koji se može serijalizirati te asinkrono učitavati po potrebi. Predložak se može jednostavno kreirati i uređivati u uređivaču. U predlošku je spremljena lista komponenata s postavljenim vrijednostima te arhetip izveden iz navedene liste za jeftinije stvaranje entiteta. Ideja je stvoriti entitet određenog arhetipa te ga inicijalizirati iz predloška s određenim podatcima. Na primjer, želimo stvoriti entitet koji će imati komponente `CTransform`, `CStaticMesh`, `CRigidBody` i `CColider`. Sami entitet kreira se podnošenjem zahtjeva za kreiranje novog entiteta određenog arhetipa određenom svijetu. Navedene komponente imaju prazne pretpostavljene vrijednosti, a mi bismo htjeli stvoriti entitet s određenim 3D

modelom za koji je simulirana fizika, koji ima specifičnu masu i postavke kolizije takve da se poklapaju s 3D modelom. Detaljne postavke je naporno primjenjivati u kodu, a ponekad bismo ih htjeli i promijeniti bez ponovnog prevođenja koda. Zato `EntityTemplate` sadržava cijele instance komponenti s primijenjenim postavkama koje će se kopirati u novonastali entitet. Implementacija ove klase je vrlo jednostavna zato što se koristi sustav refleksije za serijalizaciju i uređivanje kroz korisničko sučelje. Implementacija je prikazana na slici Slika 1.4., dok je na slici Slika 1.5 prikazano korisničko sučelje za uređivanje. Korisničko sučelje je u potpunosti ostvareno refleksijom, mogu se dodati ili maknuti komponente te mijenjati sve njihove vrijednosti.

```
REFLECTED()
class EntityTemplate : public Asset
{
    GENERATED()

public:
    void InitializeEntity(Entity& entity) const;

    const DArray<ObjectEntry<Component>>& GetComponentEntries() const;
    const Archetype& GetArchetype() const;

protected:
    virtual void OnPropertyChanged(Name propertyName) override;

private:
    PROPERTY(Edit, Serialize, DisplayName = "Components")
    DArray<ObjectEntry<Component>> _componentEntries;

    PROPERTY(Serialize)
    Archetype _archetype;
};
```

Slika 1.4 Klasa *EntityTemplate*



Slika 1.5 Korisničko sučelje za uređivanje predložaka entiteta

Klasa `System` implementira mogućnosti sustava opisane u prethodnim poglavljima. `SystemBase` je polimorfna bazna klasa koja implementira osnovne mogućnosti i javno sučelje. System mora biti polimorfna klasa kako zato što raspoređivač ne može znati točan tip sustava – trebamo brisanje tipova (engl. *type erasure*). Ova klasa također implementira inicijalizaciju sustava te osvježavanje upita.

Klasa System je značajno kompliciranija zbog metaprogramiranja. Parametrizirana je po nizu tipova (engl. *variadic template*) komponenata, uz provjeru za vrijeme prevođenja jesu li svi predani tipovi stvarno tipovi komponenata. Provjera je ostvarena konceptom `IsA<Derived, Base>` koji se koristi na mnogo mjesta u projektu, a provjerava je li neki tip izveden iz baznog tipa. Na slici Slika 1.6 vidimo navedeni zahtjev za tipovima i deklaraciju tipova `RequiredComponents`.

```
template <typename... ComponentTypes> requires (IsA<ComponentTypes, Component> && ...)
class System : public SystemBase
{
public:
    using RequiredComponents = TypeSet<ComponentTypes...>;

public:
    System() : SystemBase(Archetype::Create<ComponentTypes...>())
    {
    }
```

Slika 1.6 Prvi dio klase System

`RequiredComponents` je lista tipova `TypeSet`, u potpunosti stvar metaprogramiranja.

`TypeSet` je vrlo korisna struktura koju možemo pitati sadrži li neki tip, koji je indeks tipa (ako tip ne postoji, dobivamo grešku prilikom prevođenja) te je li nadskup neke druge liste tipova. `TypeSet` također pruža jednu jako elegantnu mogućnost – iteriranje po tipovima uz pomoć parametriziranog *lambda* izraza. Ova klasa ne zauzima memoriju, postoji samo za vrijeme prevođenja. Implementacija je prikazana slikom Slika 1.8.

`TypeSet` se koristi kod kreiranja arhetipa kroz kod, takav poziv je vidljiv na slici Slika 1.6, a implementacija te funkcije je prikazana slikom Slika 1.7. U ovom kontekstu `ComponentTypes` je `TypeSet` parametriziran po tipovima komponenata po kojem se iterira za vrijeme prevođenja na pregledan način.

```
template <typename ComponentTypes> requires IsA<ComponentTypes, TypeSetBase>
static Archetype Create()
{
    DArray<QualifiedComponentType> componentTypes;
    ComponentTypes::ForEach([&]<typename T>()
    {
        componentTypes.Add(TypeChecker<T>::CheckConst());
    });

    return CreateFrom(componentTypes);
}
```

Slika 1.7 Implementacija metode Archetype::Create

`RequiredComponents` se koristi kod provjere je li sustav kompatibilan s događajem, odnosno smije li dodati entitet u neki događaj.

```cpp
class TypeSetBase
{
};

template <typename... Types>
class TypeSet : public TypeSetBase
{
public:
    template <typename T>
    static constexpr bool HasType()
    {
        return (std::is_same_v<T, Types> || ...);
    }

    template <typename Other>
    static constexpr bool ContainsAll()
    {
        return (Other::template HasType<Types>() && ...);
    }

    template <typename Other>
    static constexpr bool OwnsAll()
    {
        return (Other::template HasType<std::remove_const_t<Types>>() && ...);
    }

    static constexpr void ForEach(auto&& func)
    {
        (func.template operator()<Types>(), ...);
    }

    template <typename T>
    static constexpr size_t IndexOf()
    {
        return TypeIndex<T, Types...>::value;
    }
};
```

Slika 1.8 Klasa *TypeSet*

Sljedeća bitna stavka sustava je način dohvaćanja komponenata. U poglavlju 1.1.6 spomenuta je provjera za vrijeme prevođenja. Sustav može dohvatiti komponentu za pisanje samo ako je parametriziran po tipu te komponente, dok tip ne smije biti konstantan (ključna riječ `const`). Ako je tip konstantan, sustav može dohvatiti tu komponentu samo za čitanje. Ako korisnik prekrši neko od tih pravila, dobit će grešku prilikom prevođenja – arhitektura ovog *ECS*-a ne dozvoljava sukob pristupa podacima. Implementacija je prikazana slikom Slika 1.9.

```
template <typename ComponentType> requires !std::is_const_v<ComponentType> && (std::is_same_v<ComponentType, ComponentTypes> || ...)
ComponentType& Get(Entity& entity) const
{
    ComponentType& component = entity.Get<ComponentType>(IndexOf<ComponentType>());

    if constexpr (RequiresOnChanged<ComponentType>)
    {
        EventDispatcher<TypeSet<ComponentType>>& event = GetWorld().*ComponentType::OnChanged;
        event.Add(entity, *GetBinding<ComponentType>().ListArchetype, PassKey<System>());
    }

    return component;
}

template <typename ComponentType> requires std::is_const_v<ComponentType> && (IsA<ComponentType, ComponentTypes> || ...)
const ComponentType& Get(Entity& entity) const
{
    return entity.Get<ComponentType>(IndexOf<ComponentType>());
}
```

Slika 1.9 Implementacija metoda za dohvaćanje komponente

U navedenom isječku koristi se metoda `IndexOf<ComponentType>()`. U pozadini postoji struktura `TypeMap<Binding, ComponentTypes…>` koja koristi metaprogramiranje za dohvaćanje indeksa komponente određenog tipa.

Sustav svaki korak u vremenu radi tri operacije: procesiraj generičke događaje, procesiraj sve liste entiteta iz spremljenog upita, procesiraj generičke događaje. Generički događaji mogu biti doslovno bilo što, nema nikakvih ograničenja. Na primjer, neki drugi sustav želi da ovaj sustav na početku obrade promijeni neku od svojih postavki, npr. uključi simulaciju fizike. Implementacija te metode prikazana je slikom Slika 1.10.

```
virtual void Tick(double deltaTime) override
{
    GetEventQueue().ProcessEvents();

    for (EntityList* entityList : GetQuery().GetEntityLists())
    {
        CacheArchetype(entityList->GetArchetype());

        ProcessEntityList([&]*entityList, deltaTime);
    }

    GetEventQueue().ProcessEvents();
}
```

Slika 1.10 Implementacija metode *Tick*

Obrada generičkih događaja se izvršava i na kraju metode kako bismo imali što manje kašnjenje – može se dogoditi da je sustav već započeo s obradom entiteta kad mu je neki

drugi sustav signalizirao svoj generički događaj. Kad ne bismo obrađivali generičke događaje na kraju, trebali bismo čekati sljedeći korak u vremenu.

Generički događaji su drugi način za sinkroniziranu komunikaciju između sustava i namijenjeni su za promjene postavki sustava, ne za obradu entiteta.

Specifični događaji (klasa `Event`) se obrađuju ručno zato što se obrada pojedinog događaja razlikuje od sustava do sustava. U budućnosti će se ovaj proces automatizirati refleksijom i registriranjem funkcija. Trenutno se refleksija koristi za registriranje događaja pojedinog sustava na `EventManager`. Svaki `World` ima jedan `EventManager` koji služi za registriranje događaja kako bi im se kasnije mogli osvježiti upiti.

World ima metode za kreiranje entiteta prema arhetipu i prema predlošku entiteta, na sinkroni ili asinkroni način, može se specificirati broj entiteta, funkcija prije inicijalizacije kroz sustave i funkcija nakon inicijalizacije. Kroz svijet je moguće nekom entitetu dodati ili maknuti komponentu, također na sinkroni ili asinkroni način. Dodavanje ili uništavanje komponenata mijenja arhetip entiteta, tako da se entitet miče iz trenutne liste entiteta u listu s novim arhetipom. Ovo invalidira sve pokazivače na entitet, što je jedan od razloga zašto trebamo ispitati je li entitet validan kod obrade događaja. Obrada listi entiteta uvijek obrađuje validne entitete te je redundantno provjeravati validnost.

Nakon kreiranja entiteta nekog arhetipa, entitet se popunjava podacima iz predloška te se predaje relevantnim sustavima na puni inicijalizaciju – poziv metode `OnEntityCreated`. Relevantni sustavi su oni koji obrađuju barem jednu komponentu iz arhetipa novog entiteta.

Entiteti se uništavaju kroz svijet, koji na isti način javlja sustavima da je entitet uništen, pozivom metode `OnEntityDestroyed`.

Navedeni pozivi su trenutno sinkroni, u budućnosti će se koristiti događaji za ovakve situacije, što će značajno ubrzati kreiranje entiteta, pogotovo velikog broja entiteta.

Sustavi mogu zatražiti kreiranje entiteta pozivom asinkrone verzije funkcije. Entitet će biti kreiran na početku simulacije sljedećeg koraka u vremenu.

Razlog zbog kojeg ne želimo kreirati entitete asinkrono je zato što se može dogoditi da su neki sustavi već završili s obradom kad se entitet kreirao, tako da će entitet na kraju simulacijskog koraka biti djelomično inicijaliziran, što otvara vrata nizu problema. Ovako imamo garanciju da će entitet biti potpuno inicijaliziran.

Strukture u kojima su spremljeni entiteti nisu namijenjene za višedretveni pristup, no zbog prethodno navedenog zahtjeva niti ne trebamo sinkronizirane strukture, samo bismo gubili performance.

Klasa koja implementira raspoređivanje sustava je `SystemScheduler`. S obzirom na to da je algoritam detaljno opisan u poglavlju 1.1.6, ova klasa se neće detaljno razmatrati u ovom poglavlju. Svaki sustav ima svoj raspoređivač, te ima pomoćne metode za pristup njemu za npr. dodavanje novog sustava u raspored.

Pokretač je podijeljen na podsustave, podsustav za upravljanje resursima, podsustav za dohvaćanje korisničkih unosa preko tipkovnice, miša itd., i sustav za logiku igre.

`GameplaySubsystem` ima autoritet nad logikom igre, on kreira nove svjetove. Korisnik treba naslijediti klasu Game te implementirati metodu koja inicijalizira svijet. Opcionalno može implementirati metode koje se pokreću kad se igra pokrene i završi. Korisnik također treba u uređivaču kreirati novi resurs tipa te naslijeđene klase i postaviti ga kao vrijednost varijable „*Game*“ unutar resursa *ProjectSettings*. Podsustav za logiku igre će duplicirati novi resurs i pokrenuti igru.

Inicijalizacija svijeta podrazumijeva dodavanje sustava u njega te odabir razina koje želimo asinkrono učitavati. U tom trenutku je moguće sinkrono kreirati entitete i postavljati sustave.

1.2. Obrada scene za iscrtavanje

U ovom poglavlju bit će objašnjeni principi iscrtavanja implementirani u pokretaču *Infinity Engine* kao i njihove prednosti i mane. *Infinity Engine* je prilagođen za velik broj entiteta, tako da ima lošije performance za jednostavne scene koje ne zahtijevaju nikakvu obradu prije iscrtavanja. U ovom projektu implementirano je iscrtavanje samo statičkih 3D modela (engl. *static mesh*).

Naivni pristup iscrtavanju je za svaki objekt u sceni poslati naredbu za iscrtavanje na grafičku karticu. Svaki poziv za iscrtavanje ima dodanu cijenu (engl. *overhead*) zbog slanja preko *PCIe* sučelja i promjene konteksta na grafičkoj kartici. Promjena konteksta uključuje promjenu spremnika vrhova i indeksa, promjenu sjenčara te promjenu ostalih postavki cjevovoda iscrtavanja.

Bolji pristup je snimati naredbe u listu naredbi (engl. *command list*). Liste naredbi može puniti više dretvi, što povećava performance na procesoru. Spremanje naredbe u listu je značajno jeftinije od izravnog slanja na grafičku karticu. Tek kada spremimo sve naredbe koje želimo, šaljemo listu na grafičku karticu na izvođenje. Ovaj projekt koristi *DirectX 12*, tako da je potrebno ručno sinkronizirati CPU i GPU čekanjem na memorijske barijere (engl. *memory barrier*), te je potrebno koristiti sinkronizacijske mehanizme samih resursa. Za sinkronizaciju resursa koriste se tranzicije stanja, na primjer, prije osvježavanja spremnika na grafičkoj kartici, potrebno ga je prebaciti iz stanja za generalnu uporabu u `D3D12_RESOURCE_STATE_COMMON` u stanje za kopiranje `D3D12_RESOURCE_STATE_COPY_DEST`, te nakon toga vratiti u stanje za generalnu uporabu. Ovo je primjer za spremnik za generalnu uporabu, npr. za konstantne spremnike koji se koriste za ulazne podatke za sjenčare. Konstantni spremnici se tako zovu zato što se njihov sadržaj ne može mijenjati na grafičkoj kartici, specifično unutar računskih sjenčara (engl. *compute shader*) zato što grafički sjenčari ionako ne mogu mijenjati nikakve podatke [2]. Spremnici koji se koriste u računskim sjenčarima, specifično oni u koje se može zapisivati imaju dozvoljen pristup s grafičke kartice (engl. *unordered access*), sinkroniziraju se na malo drukčiji način, no ideja je ista.

DirectX 12 omogućava programerima značajno višu razinu kontrole svih dijelova cjevovoda, sinkronizacije, kopiranja podataka iz procesorske memorije u grafičku memoriju i obrnuto. Viša kontrola znači da programer može ostvariti više performance u implementaciji specifičnih slučajeva.

Ovaj pristup smanjuje cijenu iscrtavanja na procesoru, no možemo i bolje. Brži pristup je korištenje GPU instanciranja (engl. *GPU instancing*). Instancirani poziv iscrtavanja je zapravo obični poziv iscrtavanja koji se izvodi nekoliko puta. Klasični pristup instanciranju je definiranje podataka o instancama unutar definicije tipa vrha (u *DirectX 12* to se zove *input layout*). Za potrebe ovog projekta ovakav pristup ne omogućuje dovoljno visok stupanj kontrole, tako da se koristi kompliciranije rješenje.

1.2.1. GPU instanciranje

U sklopu ovog projekta razvijeno je robusno rješenje za GPU instanciranje. Podaci o specifičnoj instanci su spremljeni u jedan velik spremnik koji procesor osvježava po potrebi. Svaki svijet ima sustav `StaticMeshRenderingSystem` koji sadržava opisani spremnik i osvježava specifične instance kojima se promijenila lokacija (odnosno bilo koji podatak o transformaciji). Na slici Slika 1.11 prikazana je definicija strukture instance.

```
struct SMInstance
{
    Matrix World;
    uint32 MeshID;
    uint32 LOD;
    uint32 MaterialID;
    uint32 MaterialIndex;
    uint32 Count;
};
```

Slika 1.11 Struktura s podacima o instanci

Struktura sadrži matricu transformacije instance, odnosno matricu kojom množimo vrhove modela kako bismo dobili njihove lokacije u globalnom koordinatnom sustavu. U *DirectX 12*, specifično u HLSL-u, svi elementi strukture moraju biti poravnati (engl. *aligned*) na 4 bajta zato što su podatci spremljeni kao `float4`, odnosno vektor od 4 32-bitne vrijednosti. Poravnavanje zahtijeva posebnu pažnju, tako da se struktura u C++-u poklapa sa strukturom u HLSL-u, inače ćemo u kodu zapisivati podatke na krivo mjesto.

Struktura također sadrži podatke o resursu 3D modela, razini detalja, resursu materijala i broju instanci. Broj instanci se trenutno koristi kao povratna informacija procesoru te će biti maknuta u budućim verzijama.

Varijabla `MaterialIndex` predstavlja indeks instance u spremniku specifičnog materijala. `StaticMeshRenderingSystem` također održava mapu materijala i spremnika instanci tog materijala. Svaki materijal ima svoj spremnik instanci materijala koji sadrži podatke specifične za taj materijal.

Podaci koje instanca treba sadržavati određuju se iz sjenčara. Prilikom prevođenja sjenčara koristi se HLSL refleksija kojom se dobiju parametri sjenčara. Parametri se povežu s refleksijom pokretača u strukturu `MaterialParameterMap` i njezinu *DirectX 12* verziju `DX12MaterialParameterMap`. Iscrtavanje je apstrahirano kako bismo budućnosti mogli jednostavnije dodati podršku za druge biblioteke za iscrtavanje.

Ovakav pristup omogućava nam instancirano iscrtavanje 3D modela bilo kojeg materijala, i taj materijal može imati bilo kakve parametre, bez ograničenja.

1.2.2. Obrada scene na grafičkoj kartici

S obzirom na to da projekt cilja na velik broj instanci, koristit ćemo procesor namijenjen za paralelnu obradu velikog broja malih podataka, odnosno GPU. Ovakav princip zove se *GPU-driven rendering*. Cilj je gotovo sve što se tiče iscrtavanja obaviti na grafičkoj kartici. U poglavlju 1.2.1 navedeno je da instance svih 3D modela različitih materijala i razina detalja spremamo u isti spremnik.

Ako želimo koristiti instancirano iscrtavanje, spremnik mora biti sortiran. Sortiranje je moguće obaviti na procesoru, odnosno sortirati spremnik u radnoj memoriji te ga kopirati u memoriju grafičkog procesora. Očekujemo visok broj entiteta, što znači da bismo trebali sortirati spremnik uz pomoć više dretvi. Drugi problem je što bi spremnik mogao biti velik, tako da bi i kopiranje bilo skupo.

Pametnije rješenje je sortirati spremnik na grafičkoj kartici uz pomoć računskog sjenčara. Na procesoru nas ionako ne zanima točan redoslijed instanci, tako da ne trebamo kopirati spremnik. Računski sjenčar će sortirati spremnik instanci na grafičkoj kartici koristeći međuspremnik iste veličine.

Bitna okolnost je da za sortiranje koristimo GPU, odnosno procesor s vrlo velikim brojem jezgri koje su same po sebi sporije od CPU jezgri. To znači da za sortiranje moramo koristiti nekoliko dretvi – treba nam višedretveni algoritam sortiranja koji dobro radi na grafičkom procesoru.

U sklopu ovog projekta za sortiranje koristi se algoritam bitonskog sortiranja (engl. *bitonic sort*). Algoritam je namijenjen za paralelno sortiranje velikog broja elemenata, što nam i treba. Bitonsko sortiranje podvrsta algoritma sortiranja spajanjem (engl. *merge sort*).

Bitonsko sortiranje operira nad spremnicima čija je veličina potencija broja 2 zato što algoritam ima nekoliko razina. Sortiranje se izvršava u blokovima, gradi se mreža uspoređivanja koja izgleda piramidalno. Blokovi su prvo veličine 2 elementa, pa 4, pa 8 itd. Za računanje indeksa elemenata koje uspoređujemo koriste se binarne operacije (engl. *bitwise*) uz masku izračunatu iz trenutne razine.

Implementacija algoritma za pokretanje na grafičkoj kartici nije trivijalna. Specifično, implementacija samog algoritma odnosno računskog sjenčara nije osobito komplicirana, no priprema sa strane procesora zahtijeva značajne napore.

Računski sjenčar treba pozvati za svaku razinu dvaput, uz pomoćni računski sjenčar za „transponiranje“ spremnika (drugi korak algoritma). Kao podloga za implementaciju sortiranja koristi se implementacija algoritma u *DirectX 11* [3], iako značajno izmijenjena kako bi podržavala potrebe ovog pokretača. Preuzeti kod je također bio izmijenjen kako bi se mogao pokretati u kontekstu *DirectX 12*, odnosno dodana je sinkronizacija resursa i drukčiji prijenos parametara. Zbog kompleksnosti implementacije, detalji neće biti razmatrani u ovom radu.

Za potrebe sortirana razvijeno je generalno rješenje za računske sjenčare, kao i za obične grafičke sjenčare. Svi sjenčari su resursi koji se mogu spremati i učitavati, a prema potrebi i ponovno prevesti. S obzirom na to da su sjenčari spremljeni, nema potrebe ih prevoditi, što ubrzava učitavanje programa, ali i asinkrono učitavanje razina u svijetu. Jednostavno je dodati novi sjenčar bilo kojeg tipa te ga koristiti u ostatku programa.

Na slici Slika 1.11 prikazane su tri varijable po kojima trebamo sortirati spremnik: `MaterialID`, `MeshID` i `LOD`. Sortiramo tim redoslijedom zbog cijene promjene sjenčara, odnosno promjene objekta stanja cjevovoda (engl. *pipeline state object*, *PSO*). Promjena 3D objekta je jeftina, samo treba povezati drugi resurs na sjenčar prilikom poziva iscrtavanja. Na ovaj način prvo ćemo iscrtati sve objekte koji koriste jedan sjenčar, i to prvo objekte koji koriste jedan 3D model najviše razine detalja (najniži *LOD*).

Poziv za iscrtavanjem scene unutar pokretača ima nekoliko koraka:

- Skupi vidljive instance

- Odredi razinu detalja pojedine instance
- Skupi vidljiva svjetla
- Sortiraj vidljive instance
- Kompresiraj vidljive instance u podatke za instancirani poziv
- Pokreni pozive za iscrtavanje s dobivenim podacima i svjetlima

Skupljanje vidljivih instanci obavljeno je pomoću grafa poslova u DirectX 12 (engl. *work graph*). Grafovi poslova su najnoviji dodatak u DirectX 12 te omogućavaju efikasnije pozive računskih sjenčara. Trenutno ne mogu izdavati pozive za iscrtavanje, no u budućnosti vjerojatno hoće.

Ulaz u graf poslova su instance koje se filtriraju računskim sjenčarom. Ako je instanca izvan radijusa iscrtavanja zapisanog u strukturi scene, neće se iscrtavati (engl. distance culling). Drugi uvjet je da kamera vidi instancu, odnosno da se instanca nalazi unutar frustuma kamere (engl. frustum culling). Dodatni uvjet je da je instanca stvarno vidljiva na ekranu, odnosno nije sakrivena iza nekog zida (engl. *occlusion culling*). Ovaj uvjet nije implementiran u sklopu ovog projekta te neće biti detaljno razmatran. Najčešći pristup je iscrtati na pomoćni spremnik okvira (engl. *frame buffer*) dovoljno velike objekte koji bi mogli sakrivati manje, a nakon toga pokušati iscrtati granični okvir malog objekta te provjeriti je li stvarno vidljiv. Ovakav pristup zahtijeva dodatni posao s procesorske strane, tako da će u budućnosti ta provjera biti ostvarena kroz graf poslova na drukčiji način.

U ovom koraku određujemo razinu detalja modela ovisno o udaljenosti od kamere. Formula za računanje razine detalja prikazana je izrazom (1). Razina detalja je najčešći i najjednostavniji pristup optimizaciji igre - ako objekt ne zauzima značajan postotak ekrana, nema razloga iscrtavati verziju s najviše detalja. Klasa StaticMesh podržava više spremnika vrhova i indeksa, po jedan za svaku razinu detalja. Ako želimo koristiti ovu funkcionalnost, izvezeni model iz programa za uređivanje 3D modela mora imati specifičnu nomenklaturu – naziv objekta mora završavati s „_LODX“ gdje X predstavlja razinu detalja. Ako objekt ne nazovemo tako, pretpostavlja se razina detalja 0, odnosno najdetaljnija razina. Pokretač podržava 10 razina detalja, no ta brojka je izabrana arbitrarno te se može jednostavno promijeniti.

Graf poslova dodaje vidljive instance u međuspremnik za daljnju obradu.

$$LOD = e^{\frac{\frac{distance * e}{drawDistance}}{lodInterval}} - 1 \quad (1)$$

Nažalost, unutar ovog projekta ne iskorištava se puni potencijal grafova poslova, tako da će u budućnosti ovaj dio koda biti izmijenjen i procesor će stvarno raditi minimalnu količinu posla. Trenutna implementacija je prilično neefikasna i gubimo performance.

Skupljanje vidljivih svjetala bit će objašnjeno u poglavlju 1.2.3.

Za sortiranje vidljivih instanci koristimo prethodno opisani algoritam.

Kompresiranje vidljivih instanci koristi posebni računalni sjenčar koji pokreće 32 dretve po grupi te onoliko grupa koliko je potrebno da se pokrije cijeli ulazni spremnik. Svaka dretva broji identične instance te prvoj instanci tog tipa povećava varijablu `Count`.

Pomoću te varijable na procesoru znamo koliko instanci kojeg tipa trebamo iscrtati. Prilikom instanciranog poziva na cjevovod povezujemo spremnik instanci, spremnik instanci trenutnog materijala i spremnik svjetala.

1.2.3. Blokovski model osvjetljenja

Tri glavna modela sjenčanja su unaprijedno, napredno unaprijedno i odgođeno sjenčanje.

Unaprijedno sjenčanje (engl. *Forward shading*) je najjednostavniji model sjenčanja. Sva svjetla u sceni su pohranjena u jedan spremnik, te iscrtavamo 3D modele njihovim sjenčarima koji iteriraju po svim svjetlima u sceni i računaju boju piksela. Kod velikih količina svjetala u sceni, obično unaprijedno osvjetljenje ima vrlo loše performance.

Odgođeno sjenčanje (engl. *deferred shading*) je kompletno drukčiji pristup sjenčanju. Prvo jednostavnim sjenčarom iscrtamo vidljive modele u pomoćni spremnik okvira. Potreban nam je samo sjenčar vrhova, što smanjuje cijenu iscrtavanja upola. U pomoćni spremnik zapisujemo podatke o pravom sjenčaru i njegove parametre. Nakon toga prolazimo po svim pikselima i osvjetljavamo ih sjenčarima koji su zapisani u pomoćnom spremniku. Ovaj korak zahtijeva pisanje velikog sjenčara koji poziva specifične sjenčare (engl. *uber-shader*). Tu se kompliciraju stvari, taj sjenčar treba znati za svaki mogući sjenčar u sceni, što znači da kad napišemo novi sjenčar, moramo ga dodati u veliki sjenčar. Tu se podrazumijeva da specifični sjenčari imaju identične parametre. Ovo je veliko ograničenje, specifični sjenčari više ne mogu imati posebne parametre. Drugi problem kod odgođenog osvjetljenja je kako efikasno odrediti koje piksele trebamo iscrtati kojim sjenčarom – idealno želimo jednim pozivom sjenčara iscrtati sve piksele koje on treba iscrtati.

Blokovsko osvjetljenje (engl. *tiled shading*) je jezgra naprednog unaprijednog osvjetljenja (engl. *Forward+ shading*). Ideja je ekran podijeliti na blokove koji sadržavaju svjetla relevantna za taj dio ekrana i tako smanjiti cijenu računanja osvjetljenja. Ovaj model ostvaruje veće performance za velik broj svjetala u odnosu na odgođeno sjenčanje i nema ograničenje na parametre [4].

U ovom projektu materijali mogu imati bilo kakve parametre – svaki materijal ima spremnik svojih instanci u grafičkoj memoriji.

Ideja je iterirati po svim svjetlima u sceni na sljedeći način: pozvat ćemo računski sjenčar s nekoliko grupa dretvi veličine 16x16. Svaka grupa predstavlja jedan blok svjetala. Svaka dretva obrađuje svako 16x16=256. svjetlo Tako svaka grupa obradi svako svjetlo jednom. Provjera vidljivosti svjetla je provjera sjecišta sfere i frustuma. Ako je svjetlo vidljivo, dodaje se u blok te grupe.

Kod inicijalizacije prozora i kod svake promjene veličine prozora trebamo inicijalizirati pomoćni spremnik – spremnik u kojem se nalaze frustumi za pojedinu dretvu. Računanje pojedinog frustuma je relativno jednostavno, uključuje kreiranje 2D kvadrata u koordinatnom sustavu ekrana te inverznu projekciju u koordinatni sustav kamere (engl. *view space*).

1.2.4. Upravljanje svjetlima

Pokretač trenutno podržava samo točkasta svjetla (engl. *point light*). Svaki svijet ima sustav `PointLightSystem` koji radi na sličan način kao `StaticMeshRenderingSystem` – registrira pomake entiteta preko događaja. Kada se entitet koji ima svjetlo na sebi pomakne, ovaj sustav će na sljedećem pozivu iscrtavanja osvježiti lokaciju svjetla na grafičkoj kartici.

1.2.5. Računanje osvjetljenja

Osnovni sjenčar koristi navedene podatke o svjetlima za računanje osvjetljenja pojedinog piksela. Osvjetljenje se računa koristeći Cook-Torranceov model uz GGX funkciju normalne distribucije. Izraz (2) prikazuje Cook-Torranceov [5] model sjenčanja za jedno svjetlo.

$$f(l,v) = \frac{D(h)F(l,h)G(l,v,h)}{4(n\cdot l)(n\cdot v)} \quad (2)$$

- D – funkcija normalne distribucije
- F – Fresnelova funkcija

- G – geometrijska funkcija

U ovom sjenčaru koristi se GGX funkcija normalne distribucije, prikazana izrazom (3).

$$D(m) = \frac{\chi^+(n \cdot m)}{\pi \alpha_x \alpha_y \left(\frac{(t \cdot m)^2}{\alpha_x^2} + \frac{(b \cdot m)^2}{\alpha_y^2} + (n \cdot m)^2 \right)^2} \quad (3)$$

Vrijednost α predstavlja funkciju hrapavosti (engl. *roughness*). Postoji više funkcija od kojih se u ovom projektu koristi najintuitivnija, prikazana izrazom (4).

$$\alpha = roughness^2 \quad (4)$$

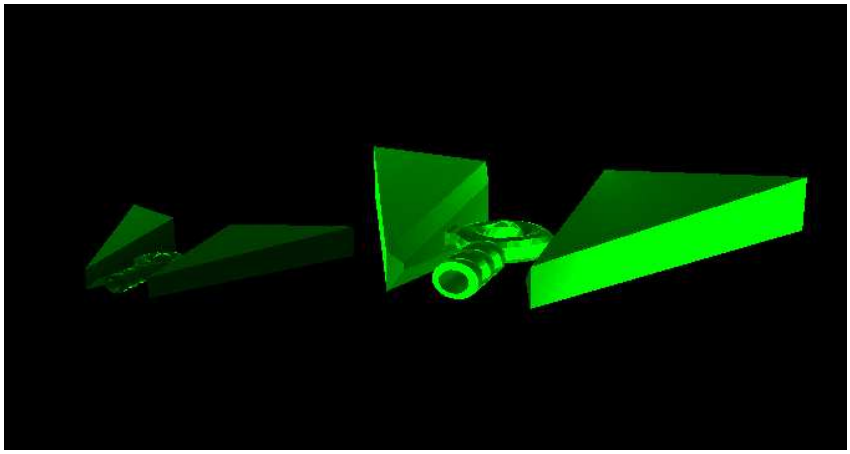
Za Fresnelovu funkciju koristi se Schlickova aproksimacija prikazana izrazom (5).

$$F(n, l) = F_0 + (1 + F_0)(1 - (n \cdot l)^+)^5 \quad (5)$$

U ovom sjenčaru, koristi se Karisova aproksimacija Smithove G_1 funkcije za GGX, prikazana izrazom (6).

$$G_1 \approx \frac{2(n \cdot s)}{(n \cdot s)(2 - \alpha) + \alpha} \quad (6)$$

Koristeći izraz (2) računamo doprinos svakog relevantnog svjetla. Doprinose jednostavno sumiramo te na njih dodamo ambijentno osvjetljenje. Relevantna svjetla se određuju iz bloka unutar kojeg se nalazi trenutni piksel. Blok može imati do 256 svjetala, no ta brojka se može jednostavno promijeniti. Slika 1.12 prikazuje rezultate iscrtavanja opisanim modelom.



Slika 1.12 Implementirani model osvjetljenja

1.2.6. Podrška za grafove poslova

Grafovi poslova su podržani na grafičkim karticama od serije *NVIDIA RTX 30*, pokretač trenutno nije moguće pokrenuti na računalima sa starijim grafičkim karticama.

Operacijski sustav Windows 10/11 dolazi s ugrađenom podrškom za *DirectX 12*, no verzija biblioteke i prevoditelja je stara nekoliko godina. Od 2021. godine postoji *Agility SDK*, biblioteka koja implementira najnovije standarde *DirectX 12* i sadrži novi prevoditelj. Ako želimo podršku za grafove poslova ili sjenčare mreža (engl. *mesh shader*), moramo koristiti novu verziju biblioteke *Agility SDK*. Također, novi prevoditelj je značajno brži, podrška za prevođenje je robusnija.

Ovaj projekt je inicijalno koristio biblioteke koje dolaze s operacijskim sustavom, tako da je bilo potrebno integrirati biblioteku *Agility SDK*, odnosno zamijeniti postojeći kod prevođenja sjenčara i serijalizacije s novim. Skripta za instalaciju također skida ovu biblioteku i dodaje u projekt.

Korištenje ove biblioteke podrazumijeva pakiranje dinamičke biblioteke (engl. *dynamic link library*, *.dll*) DirectX-a uz naš program te povezivanje s njom kod pokretanja programa.

Nadogradnja na *Agility SDK* nam omogućava da u budućnosti iscrtavanje zamijenimo novim cjevovodom, sjenčarima mreža.

1.2.7. Buduće nadogradnje

Pokretač trenutno nema implementiranu simulaciju sjena niti globalnog osvjetljenja. Klasični pristup simulaciji sjena je mapiranje sjena (engl. *cascaded shadow maps*, *CSM*) koje zahtijevaju iscrtavanje dubine iz perspektive svakog svjetla u sceni. *CSM* ima visoku cijenu sa strane procesora, ali i sa strane grafičke kartice, a kvaliteta sjena je osrednja. Paralelno uz *CSM* korisno je računati sjene u prostoru ekrana (engl. *screen space shadows*) koje dodaju veliku količinu detalja, pogotovo za male i tanke objekte.

Pametniji pristup je spremanje informacija o trenutnom osvjetljenju, tzv. *radiance caching*. Ovo nije novi koncept, no tek unazad nekoliko godina se počeo spominjati za osvjetljenje u stvarnom vremenu. Glavni razlog tome je bio nedovoljno sposoban hardver. U ovom pokretaču, u budućnosti će biti implementirano spremanje informacija o osvjetljenju u kombinaciji s praćenjem zraka. Postoji ideja o korištenju neuralnih mreža za spremanje informacija o osvjetljenju kako bi se ostvarile više performance. S obzirom na to da nove generacije grafičkih kartica imaju poseban hardver za umjetnu inteligenciju (tenzor jezgre) i poseban hardver za praćenje zraka (*RT* jezgre), trebali bismo iskoristiti puni potencijal hardvera.

Budućnost osvjetljenja je kombinacija spremanja informacija i praćenje zraka uz korištenje neuralnih mreža. Puno praćenje zraka (engl. *path tracing*) zahtijeva iznimno jak hardver koji neće biti dostupan nekoliko generacija. Neuralne mreže značajno smanjuju cijenu ovakvih algoritama grube sile, bez njih i obično praćenje zrake (engl. *ray tracing*), čak i uz posebno namijenjen hardver, ostvaruje vrlo nizak broj sličica u sekundi. Praćenje zrake postaje moguće tek uz iscrtavanje na niskoj rezoluciji pa povećanje rezolucije slike neuralnim mrežama poput *DLSS*-a i *XeSS*-a ili eventualno algoritmima bez korištenja neuralnih mreža, poput algoritma *FSR*, koji ostvaruju nižu kvalitetu slike.

1.3. Fizikalna simulacija

U ovom projektu ostvarena je jednostavna funkcionalnost fizikalne simulacije entiteta. Cilj nije bio ostvariti maksimalne performance već samo osnovnu funkcionalnost dovoljnu za ostvarivanje uređivača razina.

`PhysicsSystem` je sustav koji ima mogućnost fizikalne simulacije entiteta – simulacija kretanja i sudara, te izvršavanje upita nad scenom. Trenutno je implementiran samo presjek linije i scene (engl. *raycast*, *line trace*).

Komponente koje su nam potrebne za simulaciju su `CTransform`, `CRigidBody` i `CCollider`. `CCollider` trenutno podržava oblike `BoundingBox` (specifično tzv. *axis aligned bounding box*) i `MeshCollision`. Korisnik može oblik odabrati u uređivaču, kao i popuniti podatke specifične za taj oblik.

Svijet je podijeljen u 3D uniformnu mrežu gdje svaka ćelija sadržava pokazivač na fizikalno tijelo. Kada kreiramo novi entitet s navedenim komponentama, entitet se dodaje u sve ćelije koje se preklapaju s njegovim graničnim okvirom. Ćelije se kreiraju po potrebi kako bi inicijalizacija sustava bila brža.

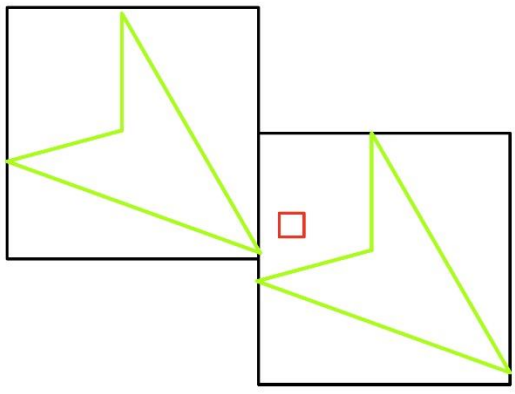
Sustav ima nedovršenu podršku za preklapanja (engl. *overlap*) koja nam trenutno nije potrebna. Sustav sluša na događaje promjene transformacije entiteta. Nakon pomaka, sustav pomiče entitet u nove ćelije, provjerava je li entitet ostao u preklapanju s entitetima otprije (želimo znati kad je entitet ušao u preklapanje s drugim entitetima i kad je izašao iz njega) te računa presjeke u novim ćelijama. Radi optimizacije, prvo se ispituje presjek graničnih okvira, ako on postoji, računa se pravi presjek oblika zapisanih u komponenti `CCollider`.

Drugi dio sustava je fizikalna simulacija. Komponenta `CRigidBody` ima informaciju o stanju entiteta u simulaciji, odnosno je li entitet statički, dinamički ili je u hibernaciji. Statički entiteti se nikad ne miču (odnosno simulacija ih ne miče), entiteti u hibernaciji se mogu probuditi kad se dinamički entiteti sudare s njima.

Simulacija obrađuje samo dinamičke entitete. Entiteti u ćelijama su sortirani prema stanju radi optimizacije – u svakoj ćeliji želimo pomaknuti samo dinamičke elemente, zato ćemo dinamičke entitete smjestiti na kraj polja, kako bi se lako mogli dodavati i micati iz ćelija.

Za svaki dinamički entitet izračunat ćemo novu lokaciju ovisno o njegovoj trenutnoj brzini zapisanoj u `CRigidBody`. Ne možemo samo pomaknuti entitet na tu lokaciju i testirati presjeke, zato što taj pristup funkcionira samo za entitete s malom brzinom. Drugi problem su mali entiteti. Brzi entitet bi mogao proći kroz nekoliko entiteta prije nego što dođe na ciljanu lokaciju.

U ovom projektu pomak se radi u koracima, entitet se pomiče u intervalima te se svaki put računaju presjeci. Interval ovisi o veličini graničnog okvira entiteta i smjeru kretanja Čim nađemo presjek, zaustavljamo iteraciju i obrađujemo sudar. Ovakav pristup ne rješava problem u cijelosti, i dalje je moguće da veći entitet prođe kroz mali entitet bez detekcije sudara. To će se dogoditi kad se mali entitet nađe između dva koraka. Bit će pokriven graničnim okvirom velikog entiteta, ali prilikom računanja presjeka, presjek nećemo pronaći zato što se dogodio između koraka. Potrebno je pronaći preciznije rješenje. Slika 1.13 prikazuje jednu takvu situaciju – zeleni objekt prolazi kroz crveni.



Slika 1.13 Primjer prolaska velikog objekta kroz malen objekt

1.3.1. Detekcija presjeka

Kada pomaknemo entitet, moramo provjeriti je li entitet u presjeku s drugim entitetom, što radi simulacije sudara, što radi simulacije preklapanja.

Za detekciju presjeka koristi se par algoritama, Gilbert-Johnson-Keerthi (*GJK*) [6] i algoritam širenja politopa (engl. *expanding polytope algorithm*, *EPA*) [7]. Rezultat algoritma *GJK* je simpleks i dubina presjeka, bez detalja o lokaciji sudara i normali. Ako tijela imaju presjek, moramo izračunati ostale informacije. Za to se koristi algoritam *EPA* koji kao ulaz prima simpleks koji smo dobili kao rezultat algoritma *GJK*.

Generalna ideja kod traženja presjeka je izračunati Minkowskijevu razliku dva objekta. Minkowskijeva razlika se računa oduzimanjem vrhova drugog 3D modela od prvog, specifično, najdaljih vrhova u nekom smjeru. Ako dobiveni oblik sadrži ishodište, postoji presjek, no ne znamo ništa više od toga. Izračun cijele razlike nije jeftin, ali i provjera je li oblik sadrži ishodište nije jeftina za kompleksniji oblik.

GJK računa samo onaj dio razlike koji nam je dovoljan da zaključimo postoji li presjek. Kako bismo saznali koji vrh je najdalji u nekom smjeru, implementirat ćemo pomoćnu funkciju, tzv. *support function*. Algoritam započinje biranjem smjera, obično je to os X, te izračun pozicija najdaljih vrhova u tom smjeru pomoćnom funkcijom. Iz Minkowskijeve razlike i najdaljih vrhova kreira se simpleks. Ako simpleks sadrži ishodište, algoritam je gotov. Inače tražimo novi simpleks i smjer, tako da uzmemo trenutni simpleks i iz njega odredimo smjer prema ishodištu te nove vrhove simpleksa. Ako je simpleks linija, novi simpleks će sadržavati samo jedan vrh, a smjer će ovisiti o tome je li ishodište ispod ili iznad linije. Slučaj kad je simpleks trokut obrađuje se na sličan način, trokut se sastoji od tri linije. Treći mogući slučaj je simpleks u obliku tetraedra koji se sastoji od četiri trokuta, tako da možemo ponovno iskoristiti funkciju za slučaj trokuta. Proces se ponavlja dok ne pronađemo simpleks koji sadrži ishodište ili dok kut između Minkowskijeve razlike i trenutnog smjera ne postane veći od 90 stupnjeva, kada znamo da presjeka nema.

Ako postoji presjek, s pomoću algoritma *EPA* računamo točku kontakta i normalu. *EPA* je kompleksniji algoritam koji generira 3D mrežu, tzv. politop, koji se nadograđuje novim trokutima dok ne pronađemo trokut najbliži ishodištu. Trokute politopa držimo u polju sortiranom prema udaljenosti trokuta od ishodišta.

Algoritam se inicijalizira iz simpleksa, odnosno iz trokuta simpleksa oblika tetraedra (ako simpleks nije tetraedar, potrebno ga je nadopuniti). Pratimo trenutnu udaljenost od ishodišta, te ako je trenutni trokut koji obrađujemo bliži ishodištu od te udaljenosti, moramo nadograditi politop. Trenutni trokut brišemo te pratimo njegove susjedne trokute i njihove susjede – tražimo siluetu odnosno sve trokute koji vide lokaciju Minkowskijeve razlike. Razlika se računa u smjeru najbliže točke na trokutu do ishodišta, gdje točka ne mora nužno biti unutar trokuta, samo mora biti na istoj plohi. Trokuti iz kojih se vidi razlika se brišu.

Iz dobivene siluete kreiramo nove trokute te ih povezujemo međusobno i sa susjedima trokuta koje smo izbrisali u procesu računanja siluete. Ako novi trokut sadrži najbližu točku ishodištu na svojoj plohi, dodajemo ga u sortiranu listu. Proces se prekida kad u listi naiđemo na trokut čija je udaljenost dalja od trenutne najkraće udaljenosti.

Iz podataka o najbližem trokutu možemo izračunati normalu (to je normala najbližeg trokuta), dubinu penetracije – udaljenost od ishodišta, te lokaciju presjeka iz baricentričnih koordinata najbliže točke na trokutu.

Niti *GJK* niti *EPA* ne podržavaju konkavne objekte. Ako želimo konkavne kolizije, moramo ih podijeliti u nekoliko konveksnih. `PhysicsSystem` trenutno ne podržava više kolizija po entitetu, no u budućnosti hoće.

1.3.2. Simulacija sudara

Nakon što izračunamo parametre sudara, moramo simulirati sudar između dva entiteta. Impuls se računa iz relativne brzine, normale sudara, koeficijenta restitucije i momenta inercije. Simuliramo promjenu i linearne i kutne brzine oba entiteta, naravno, ako su dinamički. Nova brzina prvog entiteta je razlika trenutne brzine i impulsa podijeljenog masom prvog entiteta. Kutna brzina ovisi o vektoru centar mase-lokacija sudara i momentu inercije. Nova kutna brzina je razlika trenutne kutne brzine i vektorskog produkta navedenog vektora i vektora impulsa dijeljenog s momentom inercije.

Nakon sudara trebamo pomaknuti entitet u smjeru normale sudara ovisno o dubini penetracije.

Ako se sudar nije dogodio, entitet pomičemo za $v * \Delta t$, te primjenjujemo novu rotaciju iz $\omega * \Delta t$.

1.3.3. Test presjeka s linijom

Postojeći sustav za detekciju sudara moguće je iskoristiti za izračun presjeka s linijom (engl. *raycast*, *line trace*). Algoritam ne zahtijeva promjene, pomoćna funkcija za izračun najdalje točke na dužini je trivijalna.

Trebamo efikasan način za pronalaskom ćelija kroz koje prolazi linija, nešto slično kao Bresenhamov postupak u 3D.

U ovom projektu koristi se efikasni algoritam za prolazak kroz voksele [8]. Prvo izračunamo indekse početne i završne ćelije te korake u smjeru sve tri osi, tako da je izračun sljedećeg indeksa jednostavniji. Algoritam putuje po ćelijama tako da se prvo pomiče maksimalno po X osi, pa Y osi, pa Z osi – dio algoritma je osvježavanje lokalnih maksimuma. Razlog tome je optimizacija – manje operacija, te u potpunosti izbjegavamo operaciju dijeljenja.

Prilikom ulaska u svaku ćeliju računamo presjek sa svim tijelima u njoj. Ako pronađemo presjek, zaustavljamo proces i vraćamo rezultat.

1.3.4. Buduće nadogradnje

Potrebno je dovršiti implementaciju detekcije preklapanja. Ova značajka je bitna za logiku igre, npr. želimo saznati kad igrač uđe u neku prostoriju.

Nedostaje filtriranje, želimo specificirati s kojim tipom tijela se neko drugo tijelo može sudariti ili preklapati. Filter je inače implementiran kao 32-bitni cijeli broj, tijela se sudaraju ako binarna operacija „i“ vrati rezultat različit od 0 – oba tijela moraju imati uključenu opciju sudara za tip drugog tijela. Naravno, ovako se ograničavamo na 32 tipa tijela.

Ovaj sustav nema nikakve optimizacije, implementacija višedretvenosti nije osobito komplicirana. Svijet se može podijeliti između nekoliko dretvi, gdje svaka dretva ima blok ćelija. Posebnu pažnju treba pridodati tijelima koja će prijeći granice između blokova, ta tijela se ne smiju obrađivati dok neka druga dretva obrađuje blok u koji ulaze.

1.4. Particioniranje svjetova

Kod velikih scena, trenutna implementacija *ECS*-a neće ostvarivati maksimalne performance - ako korisnik ne podijeli logiku na dovoljan broj sustava, nećemo koristiti sve jezgre procesora. Ako scena ima velik broj entiteta, svaki sustav će morati slijedno obraditi puno entiteta – na jednoj dretvi. Naravno, korisnik može sustav implementirati tako da koristi više dretvi, no pokretač trenutno nema pomoćne funkcije koje bi to olakšale, poput funkcije *ParallelFor* koja je dio svake biblioteke za višedretvenost, poput *Intel Thread Building Blocks* (*TBB*). Jednostavno i jeftino rješenje je stvoriti više svjetova koji ionako neće često međusobno komunicirati.

Pokretač nema ograničenja na broj svjetova, oni se kreiraju po potrebi. Svaki svijet se obrađuje zasebno, u odvojenim dretvama, a svjetovi međusobno komuniciraju generičkim događajima. To znači da ako jedan svijet ne iskorištava sve jezgre, više takvih svjetova hoće.

Svjetovi su trenutno oblika kocke kako bi se mogli smjestiti u 3D uniformnu mrežu i kako bi se lako mogli računati susjedi.

Postoji još jedan bitan razlog zašto bismo trebali particionirati scenu. U većini igara za koordinate se koriste 32-bitni realni brojevi zbog performanci matematičkih operacija nad njima. Danas 64-bitni brojevi imaju gotovo zanemarivo veću cijenu operacija, no to ovdje nije razlog. U ovom pokretaču, kao i u mnogim drugima, koriste se *SIMD* matematičke operacije. SIMD operacije koriste posebne registre, dok veličina tih registara ovisi o mikroarhitekturi procesora. Korištenje 32-bitnih brojeva za vektore znači da jedan 4-komponentni vektor zauzima 128 bitova, kad bismo koristili 64-bitne brojeve, trebali bismo 256 bitova za jedan vektor. Problem je u tome što su procesori tek unazad nekoliko godina počeli dobivati podršku za AVX-256 i AVX-512 instrukcije, dok bi pokretač trebao podržavati i starije procesore. Iz istog razloga nisu niti matematičke biblioteke nadograđene.

Jedno od rješenja problema numeričke preciznosti je pomicanje ishodišta koordinatnog sustava (engl. *world origin rebasing*). Naravno, ne možemo pomaknuti samo ishodište, moramo pomaknuti sve objekte u sceni, što je skupo, tako da to ne možemo raditi često. Ako novo ishodište zaokružimo na najbliže jedinice, kod normalnog korištenja nećemo gubiti numeričku preciznost. Kad bismo gubili preciznost, nakon nekoliko sati scena više ne bi bila na istom mjestu kao kod pokretanja igre, što bi uzrokovalo mnoštvo problema.

Razdvajanjem scene u posebne svjetove dijelimo je na više koordinatnih sustava, gdje svaki ima svoje ishodište – postigli smo „pomicanje ishodišta“ bez pomicanja svakog objekta u sceni, no zakomplicirali smo granične slučajeve – dva entiteta koji su svaki u svojem svijetu, blizu granica, imaju vrlo različite koordinate, a zapravo su vrlo blizu.

Naravno, razdvajanje komplicira i druge rubne slučajeve, npr. želimo pronaći sve entitete unutar sfere koja je na rubu svijeta – operacija bi trebala vratiti i entitete iz drugog svijeta. Ova situacija je vrlo problematična, zato što prvi svijet nema vlasništvo nad entitetima drugog svijeta, odnosno ne može osigurati da će tuđi entiteti postojati cijelo vrijeme dok ih njegovi sustavi obrađuju. Korisnik ovakve probleme mora zaobići mehanizmom generičkih događaja i ne smije spremati pokazivače na tuđe entitete. Tehnički, moguće je spremiti pokazivače na entitete, ali moramo spremiti i njihove identifikatore. Prije svakog korištenja takvog pokazivača moramo provjeriti dvije stvari: je li entitet validan (je li obrisan) i je li to stvarno entitet koji nas zanima – moramo provjeriti poklapa li se identifikator entiteta sa spremljenim. Ako se ne poklapa, to znači da je naš entitet uništen i da je već novi entitet na njegovom mjestu, ili da je entitetu dodana ili maknuta komponenta, što znači da je premješten i više nemamo pokazivač na njega, niti ga možemo dobiti. Uglavnom, spremanje pokazivača na entitete generalno nije preporučeno, čak i ako pokazuju na entitete ovog svijeta.

1.4.1. Particioniranje razina

Bitna stavka kad razmišljamo o performancama video igara je postepeno učitavanje resursa, samo onih koji nam trebaju (engl. *streaming*). Razine u igrama otvorenog svijeta su vrlo velike, a igrač želi da se igra što prije učita – učitat ćemo samo onaj dio koji je potreban te po potrebi učitavati više.

U ovom projektu razine su predstavljene klasom `Level`. Razine su podijeljene na dijelove (engl. *chunk*) unutar 3D uniformne mreže. Mogu se mogu učitati u cijelosti (metoda `LoadAllChunks`) ili po dijelovima. Učitavanje po dijelovima je asinkrono, potrebno je predati lokaciju, radijus i funkciju povratnog poziva (engl. *callback function*). Bit će učitani svi dijelovi unutar predane sfere, te će se za svaki od upravo učitanih dijelova pozvati funkcija povratnog poziva. Metodu `Stream` dozvoljeno je pozivati iz različitih dretvi u isto vrijeme i više puta – dijelovi će se učitati samo jednom.

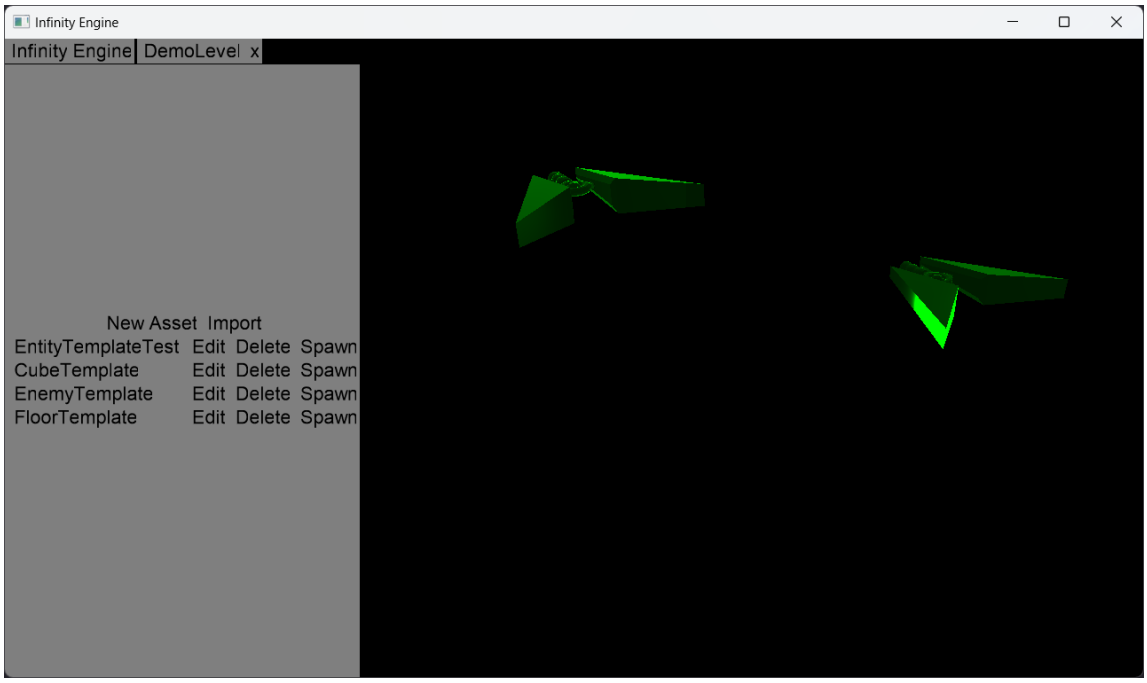
Za asinkrono učitavanje razina koristi se sustav `LevelStreamingSystem` koji svaki korak u vremenu pokuša učitati razinu na lokaciji svih entiteta koji imaju `LevelStreamingInvoker` komponentu. Komponenta propisuje radijus koji se koristi za učitavanje razine. Sljedeći korak u vremenu nakon što dijelovi razine budu učitani, ovaj sustav kreira nove entitete sa spremljenim podacima. Trenutno je moguće spremiti samo lokaciju i predložak, što je dosta veliko ograničenje koje će biti riješeno u budućim verzijama sa serijalizacijom entiteta u smislu logike igre.

Za uređivanje razina kreiran je jednostavni uređivač u kojem je moguće kreirati nove entitete i transformirati postojeće – pomicati, rotirati i skalirati. Korisničko sučelje nudi samo predloške entiteta radi lakšeg snalaženja. Klikom na entitet preuzimamo kontrolu nad njim. Operaciju biramo brojevima 1, 2 i 3, koristeći tipke WASD i QE specificiramo osi, odnosno tipka W će pomaknuti entitet u smjeru u kojem je okrenut. Uređivač je trenutno vrlo jednostavan i bit će nadograđen u budućnosti raznim vizualizacijama, mogućnosti stvaranja entiteta iz 3D modela i svjetala (trenutno je moguće stvoriti samo predložak), kontrolama na mišu, podrškom za označavanje više entiteta itd.

Za uređivanje razina koristi se sustav `LevelEditorSystem`. Ovaj sustav je povezan s korisničkim sučeljem delegatima – kad korisnik klikne gumb „*Spawn*“, sustav stvori novi entitet i doda mu komponente `CRigidBody`, `CCollider` i `CLevelEdit` ako ih nema. Prve dvije komponente su nam potrebne kako bi `PhysicsSystem` mogao obratiti te entitete – entiteti moraju biti registrirani u tom sustavu kako bismo ih mogli pogoditi zrakom ispod miša i označiti ih. Komponenta `CLevelEdit` sadrži interni identifikator entiteta u razini koji nam je nužan zato što želimo moći pomicati i uništavati entitete u razini.

Uređivanje razine koristi posebnu klasu `LevelEditorGame` koja inicijalizira svjetove s potrebnim sustavima. U ovom kontekstu `PhysicsSystem` ima ugašenu simulaciju fizike zato što želimo samo koristiti scenovne upite za odabir entiteta mišem. Trenutno je podržano uređivanje samo jedne razine, odnosno nije moguće uređivati više razina u isto vrijeme. Sučelje uređivača razine prikazano je slikom Slika 1.14.

U budućnosti ovaj problem će biti riješen klasom `Universe` koja sadrži neograničen broj razina – nešto kao velika razina manjih razina. Ovako više nećemo morati učitavati razine kroz kod, već samo specificirati resurs tipa `Universe` unutar našeg resursa tipa `Game`.

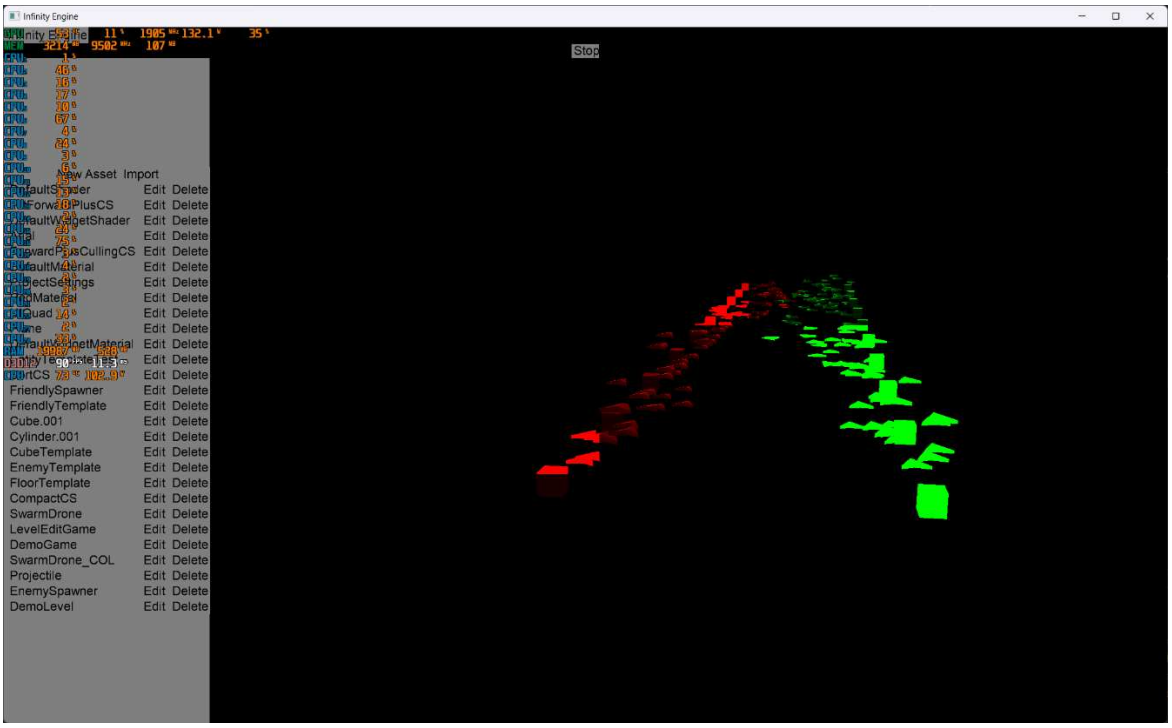


Slika 1.14 Sučelje uređivača razine

1.5. Rezultati i performance

Kod većeg broja entiteta, pokretač ostvaruje zadovoljavajuće performance. U ovoj specifičnoj sceni, traženje sljedeće mete je skupo te uzrokuje pad performanci, no u realnom slučaju, traženje mete bi bilo optimizirano. Iscrtavanje ima odlične performance, koristeći svega 11% grafičkog procesora. Kad bi traženje mete bilo optimizirano, iskorištenost grafičke kartice bi bila veća, no ne značajno. U ovoj sceni svaki entitet ima svoje svjetlo što ne uzrokuje značajan pad performanci. Slika 1.15 prikazuje scenu koja se koristi za demonstraciju. Sa svake strane postoji 10 entiteta koji periodički stvaraju entitete. Stvoreni entiteti traže protivničke entitete te se kreću prema njima. Kada im se dovoljno približe, stvaraju nove entitete, projektile, te oštećuju metu. Nakon što unište metu, traže novu. Performance su mjerene alatom *MSI Afterburner*. Na slici se također vidi kako pokretač raspoređuje poslove na više procesorskih jezgri. Za demonstraciju korišten je procesor *AMD Ryzen 9 5900X* (12 jezgri), 32 GB *DDR4* 3600 MHz radne memorije te grafička kartica *NVIDIA RTX 3080*. Broj sličica po sekundi zaključan je na maksimalno 116.

Problematični sustavi koji trenutno nisu optimizirani mogu se interno podijeliti na više dretvi. U budućim verzijama pokretača bit će jednostavno implementirati takve sustave.



Slika 1.15 Demonstracija

1.6. Upravljanje resursima

Ovaj pokretač ima implementiran uvoz resursa i kreaciju novih resursa kroz korisničko sučelje. Resursi su predstavljeni baznom klasom `Asset` koja je izvedena iz tipa `Object`, tako da podržava refleksiju i prikaz u uređivaču.

Resursima upravlja instanca klase `AssetManager` kroz koju je moguće pronaći resurs prema njegovom identifikatoru. Ova klasa je također zadužena za kreiranje novih resursa i u budućnosti će biti zadužena za pakiranje resursa za distribuciju igračima.

Prilikom pokretanja programa, `AssetManager` učitava opise resursa te ih kreira, ali ih ne učitava automatski. Ovako možemo bilo kada dohvatiti resurs i provjeriti je li učitan.

Ako želimo uvoziti resurse nekog tipa, moramo implementirati jednostavnu klasu tipa `Importer` koja služi za pohranu postavki o uvozu, npr. `StaticMeshImporter` ima podatke o putanji do 3D modela. Navedena klasa je prikazana slikom Slika 1.16.

```
REFLECTED()
class StaticMeshImporter : public Importer
{
    GENERATED()

public:
    PROPERTY(Edit)
    std::filesystem::path Path;

public:
    StaticMeshImporter() = default;
};
```

Slika 1.16 Klasa za uvoz 3D objekata

Klase za uvoz se također koriste u uređivaču. Kada kliknemo gumb „*Import*“, prikažu se sve klase izvedene iz tipa `Asset` koje imaju pridijeljen tip objekta za uvoz. Nakon odabira tipa resursa prikazuju se postavke uvoza – zapravo se pokazuje UI element za uređivanje objekta za uvoz, bit će prikazane sve varijable koje su označene s *PROPERTY* i *Edit* ili *Visible*. Kasnije u implementaciji virtualne funkcije `Import` dobivamo taj objekt za uvoz te iz njega čitamo postavke.

Resursi se mogu učitavati asinkrono, no pokretač trenutno nema elegantnu podršku za to, te je potrebno posvetiti posebnu pažnju kod sinkronizacije s ostatkom pokretača.

Za uvoz 3D modela koristi se vanjska biblioteka Assimp [9] koja omogućava jednostavan uvoz 3D modela raznih formata. U sklopu ovog projekta koristi se format *Autodesk FBX*, no kod pokretača ne razlikuje formate, za to se pobrinula navedena biblioteka.

Serijalizacija i deserijalizacija implementirana je pomoću metoda `Serialize(MemoryWriter& writer)` i `Deserialize(MemoryReader& reader)`. Primjer implementacije funkcije za serijalizaciju je na slici Slika 1.17. Implementacija funkcije za deserijalizaciju je vrlo slična, varijable moramo učitavati istim redoslijedom.

```cpp
bool StaticMesh::Serialize(MemoryWriter& writer) const
{
    if (!Asset::Serialize([&]writer))
    {
        return false;
    }

    writer << _lods;

    if (_material == nullptr)
    {
        writer << static_cast<uint64>(0u);
    }
    else
    {
        writer << _material->GetAssetID();
    }

    writer << _boundingBox;

    return true;
}
```

Slika 1.17 Metoda StaticMesh::Serialize

Klasa `MemoryWriter` ima implementiran operator „<<” kojim se zapisuje binarna reprezentacija strukture u memoriju koja će kasnije biti zapisana u datoteku. Na sličan način je implementirana i klasa `MemoryReader`.

Ako želimo ovako elegantnu serijalizaciju za neki novi tip, moramo implementirati operatore za serijalizaciju. Primjer implementacije prikazan je slikom Slika 1.18. Također, ovi operatori moraju biti implementirani ako želimo koristiti generiranu serijalizaciju i deserijalizaciju pomoću refleksije. *PROPERTY(Serialize)* se neće prevesti bez navedenih operatora.

```cpp
MemoryWriter& operator<<(MemoryWriter& writer, const BoundingBox& boundingBox)
{
    writer << boundingBox.GetMin();
    writer << boundingBox.GetMax();

    return writer;
}

MemoryReader& operator>>(MemoryReader& reader, BoundingBox& boundingBox)
{
    reader >> boundingBox._min;
    reader >> boundingBox._max;

    return reader;
}
```

Slika 1.18 Implementacija operatora za serijalizaciju

Ovakav princip serijalizacije ima veliku manu – podrazumijeva da je struktura klase finalna, odnosno da više nećemo dodavati varijable koje želimo serijalizirati, ili micati postojeće. Ako to učinimo, krivo ćemo učitavati stare resurse.

Pametnija ideja je kompliciranа. Za svaki tip resursa moramo kreirati listu verzija. Verzija ovdje predstavlja objekt s listom serijaliziranih varijabli. Kada dodamo neku novu varijablu koju želimo serijalizirati, prilikom sljedećeg pokretanja refleksije, stvorit će se nova verzija. Svaki serijalizirani objekt ima identifikator verzije – kada ga želimo deserijalizirati, pročitamo identifikator verzije, pronađemo verziju i znamo koje varijable su stvarno spremljene u datoteku.

Ovakav način serijalizacije kompliciра sustav za upravljanje izdanjima (engl. *version control system*, *VCS*). Prije nego korisnik pošalje svoje promjene na server, mora povući promjene koje nema lokalno. Moguće je da je neki drugi korisnik paralelno s njim uređivao neki resurs te ga poslao prije njega. Sada, kada korisnik povuče promjene, mora ponovno spremiti taj resurs kako bi bio spremljen s najnovijom verzijom. Ovaj proces bi trebao biti automatiziran, odnosno trebao bi se dogoditi prilikom pokretanja uređivača uz poruku korisniku. Idealno, proces bi trebao biti automatiziran i bez pokretanja uređivača, tako da ako korisnik pokuša poslati promjene bez ponovnog spremanja, trebao bi ili dobiti grešku, ili još bolje, proces bi se trebao automatski izvršiti. Unutar uređivača imali bismo meni za upravljanje verzijama resursa. Tamo bismo mogli maknuti verzije koje se više ne koriste, tako da ne povećavaju veličinu izvršne datoteke. Ovo se može dogoditi kada jedan korisnik radi na jednom tipu resursa i više puta ga izmijeni i pokrene refleksiju – bit će stvorene verzije koje se nikad neće koristiti.

Resursi se mogu uređivati kroz korisničko sučelje. Mogu se kreirati UI elementi za uređivanje i za pregled resursa. Ti elementi se kreiraju kroz mehanizam refleksije, odnosno kreira se UI element tablica varijabla - UI element. Više u poglavlju o uređivaču.

1.7. Podatkovne strukture

Za potrebe projekta, implementirano je nekoliko specifičnih podatkovnih struktura koje osiguravaju više performance od postojećih struktura u standardnoj biblioteci.

1.7.1. Klasa *DArray*

Klasa `DArray` predstavlja polje dinamičke veličine. Podaci mogu biti zapisani slijedno u jednom polju, ili u dva polja, gdje je jedno polje dio samog objekta tipa `DArray`. U prvom slučaju, klasa se ponaša kao `std::vector` s mnoštvom pomoćnih metoda za upravljanje nad podacima, poput intuitivnog brisanja elemenata, traženja indeksa elementa, brisanja elementa i zamjena sa zadnjim što drastično smanjuje cijenu brisanja elemenata, pogotovo u slučaju velikog polja. Klasično brisanje elementa iz dinamičkog polja zahtijeva pomicanje svih elemenata nakon njega ulijevo za jedno mjesto što je skupo. Također, ova klasa ima metode za traženje elementa s pomoću predikata, te brisanje svih elemenata koji zadovoljavaju predikat.

Ova klasa ima dvije varijante, varijantu koja ima jedno dinamički alocirano polje i drugu varijantu koja ima određen broj elemenata u samom objektu, a po potrebi se alocira polje za ostatak elemenata. Ovakav koncept zove se optimizacija za male veličine (engl. *small size optimization*, *SSO*). Elementi unutar *SSO* polja su jedan skok bliže od dinamički alociranih, što znači da ako u priručnoj memoriji imamo učitan objekt `DArray`, vrlo vjerojatno ćemo imati učitane i elemente iz *SSO* polja, naravno, ako ono nije preveliko za priručnu memoriju. Ako je `DArray` na stogu, čitanje *SSO* elemenata je praktički besplatno. U ovakvoj situaciji možemo se naći kod implementacije nekih algoritama, `DArray` nam daje besplatne performance. Deklaracija objekta `DArray` sa *SSO* poljem piše se ovako: `DArray<T, N>`, gdje je `T` tip objekata koje želimo spremati, a `N` kapacitet *SSO* polja.

SSO polje je implementirano kao polje bajtova poravnato kao sami tip T. Implementacija je prikazana slikom Slika 1.19. Poravnavanje (engl. *alignment*) je vrlo bitno kod performanci čitanja i pisanja u memoriju – operacije u memoriju nad neporavnatim objektom koštaju više ciklusa. Na primjer, kada bismo jedan 64-bitni objekt poravnali na 4 bajta, on se može naći na granici riječi u memoriji – prva polovica objekta će biti zapisana u drugoj polovici prve riječi, a druga polovica u prvoj polovici druge riječi. Za čitanje objekta sada nam trebaju dva ciklusa zato što trebamo pročitati dvije riječi.

```
using SSOBufferType = std::conditional_t<SSO_SIZE == 0, std::monostate, std::byte[sizeof(ValueType) * SSO_SIZE]>;

alignas(ValueType) SSOBufferType _ssoBuffer{};
```

Slika 1.19 Implementacija *SSO* polja

Postoje situacije kada će *SSO* smanjivati performance. Na primjer, imamo objekt `DArray<T, 16>` koji sadrži 32 elementa, prvih 16 elemenata je smješteno unutar samog objekta, a ostalih 16 je dinamički alocirano. Performance iteriranja bit će lošije nego kod iteriranja po objektu klase `std::vector` zato što na pola polja gubimo priručnu memoriju, ali i zato što je iterator za `DArray` sa *SSO* poljem kompliciraniji – izračun pokazivača na sljedeći element je skuplji zbog provjere je li indeks unutar *SSO* polja ili unutar dinamički alociranog polja. Također, iterator mora imati pokazivač na oba polja, odnosno zauzima dvostruko više memorije te ga je skuplje kopirati i uspoređivati.

Može se zaključiti da bi se *SSO* trebao koristiti samo za polja po kojima će se rijetko iterirati kad sadrže više elemenata nego što stane u *SSO* polje – ako želimo koristiti ovu optimizaciju, trebamo pažljivo odrediti veličinu *SSO* polja tako da u njega stanu svi elementi.

1.7.2. Klasa *BucketArray*

Klasa `BucketArray` je najbitnija struktura u ovom pokretaču. Filozofija pokretača je ne ograničavati korisnika, čak i ako će on učiniti nešto što će smanjiti performance. Iz tog razloga nije nadjačan globalni operator `new` koji bi sve alokacije radio na određenom objektu `BucketArray`. Ova klasa se koristi kada želimo alocirati velik broj objekata i osigurati da su pokazivači na njih uvijek validni. Cijena alokacije je sama po sebi značajno jeftinija, ali i amortizirana je. Svi objekti unutar pokretača alocirani su na svojim poljima, odnosno svaka klasa koja je izvedena iz klase `Object` ima svoju instancu klase `BucketArray`. Objekti se kreiraju pozivom funkcije `NewObject<T>` ili, ako ne znamo konkretni tip, korištenjem refleksije: `type->NewObject()`.

Ideja ove klase je alocirati velik blok (engl. *bucket*) memorije u koji stane N elemenata. Nakon što se taj blok popuni, alociramo novi blok. Kada izbrišemo element, označavamo njegov indeks kao slobodan. Novi objekt bit će kreiran na prvom slobodnom indeksu.

Ovakav način brisanja elemenata uzrokuje fragmentaciju koja postaje velik problem kod iteracije po svim elementima. Problem je što ne znamo je li objekt na nekom indeksu validan ili je bio izbrisan. `BucketArray` očekuje da tip `T` implementira sučelje `IValidateable` kojim možemo provjeriti validnost objekta i postavljati mu validnost direktno. Kod iteracije,

ako objekt nije validan, neće se obrađivati. Sučelje `IValidateable` je statičko sučelje, odnosno koristi princip *CRTP* za izbjegavanje virtualnih funkcija. Deklaracija ovog sučelja prikazana je slikom Slika 1.20. Naziv „statičko“ dolazi od suprotne riječi „dinamičko“, dinamičko je nešto poznato samo za vrijeme izvođenja, a statičko je nešto poznato i za vrijeme prevođenja. Prevoditelj zna točno koje metode mora pozvati i te pozive može optimizirati, dok kod dinamičkog sučelja prevoditelj bi znao samo da će objekt određene polimorfne klase imati pokazivač u tablici virtualnih funkcija. Poziv te metode za vrijeme izvođenja imao bi dodatnu cijenu, praćenje pokazivača na funkciju koja vjerojatno nije u priručnoj memoriji.

```
class IValidateable
{
public:
    template <typename Self>
    void SetValid(this Self&& self, bool value)
    {
        self.SetValidImplementation(value);
    }

    template <typename Self>
    bool IsValid(this Self&& self)
    {
        return self.IsValidImplementation();
    }
};
```

Slika 1.20 Deklaracija sučelja *IValidateable*

Sučelje koristi mogućnost standarda *C++23* za eksplicitnu deklaraciju parametra `this` u metodama (engl. *deducing this*). Ovo pojednostavljuje implementaciju principa *CRTP*.

Primjer implementacije ovog sučelja je sama klasa `Object`, čiji isječak je prikazan slikom Slika 1.21. Navedene metode jednostavno postavljaju i čitaju privatnu varijablu `_isValid`.

```
    bool _isValid = false;

    // IValidateable
private:
    void SetValidImplementation(bool value);
    bool IsValidImplementation() const;
};
```

Slika 1.21 Primjer implementacije sučelja *IValidateable*

Ova klasa se također koristi za liste entiteta i za spremanje komponenata.

Klasa `ObjectTypeMap` povezuje refleksiju i liste objekata, gdje su liste instance klase `BucketArray`. Upravo ta klasa koristi se za spremanje komponenata. Komponente ne želimo spremati u globalne strukture zbog lažnog dijeljenja i potrebne sinkronizacije kod dodavanja i brisanja komponenata. Svaki svijet ima instancu ove klase za spremanje komponenata kako bi se izbjegli navedeni problemi. Naravno, cijena ovog rješenja je povećana potreba za memorijom koja možda neće biti iskorištena.

1.7.3. Ostale strukture

U pokretaču postoji još nekoliko pomoćnih struktura za spremanje objekata.

Klasa `SparseArray` koristi se za spremanje objekata s pridodanim indeksima te je očekivano da će indeksi biti veliki, a broj aktivnih objekata malen. Problem kod korištenja slijednih polja u ovom slučaju je vrlo visoka cijena inicijalizacije. Recimo da imamo objekt s indeksom 10 000, trebali bismo alocirati polje koje može sadržavati barem toliko elemenata, što znači velika dinamička alokacija memorije, a polje će ionako imati mali broj elemenata, tako da bi iskorištenost memorije bila vrlo mala.

Klasa `SparseUniformGrid3D` je 3D varijanta klase `SparseArray` te se koristi za kreiranje mreža s vrlo velikim brojem ćelija. Inicijalizacija 3D polja s velikim brojem ćelija je vrlo skupa i uzrokuje vidljivo zastajkivanje, a znamo da velika većina polja niti ne treba postojati zato što se neće koristiti.

Klasa `LockFreeQueue` je *MPMC* red koji se koristi kod implementacije događaja, ali i bilo gdje kod jednostavne sinkronizacije više dretvi. S obzirom na to da je detaljno opisana u poglavlju 1.1.10, neće se razmatrati u ovom poglavlju.

1.8. C++ refleksija

U sklopu ovog projekta implementirana je refleksija jezika *C++*. Ako želimo koristiti refleksiju za neku klasu, moramo tu klasu označiti atributom *REFLECTED*, a varijable koje želimo reflektirati označiti atributom *PROPERTY*. Slika 1.22 prikazuje navedeno označavanje klase `CStaticMesh` koja predstavlja komponentu u ECS sustavu.

```
#include "CStaticMesh.reflection.h"

REFLECTED()
class CStaticMesh : public Component
{
    GENERATED()

public:
    PROPERTY(Edit, Serialize)
    AssetPtr<StaticMesh> Mesh;

    PROPERTY(Edit, Serialize)
    AssetPtr<Material> MaterialOverride;

    PROPERTY(Edit, Serialize)
    Transform MeshTransform;

    uint32 InstanceID = 0;
};
```

Slika 1.22 Deklaracija komponente *CStaticMesh*

Zadnje zaglavlje koje uključujemo mora biti zaglavlje refleksije za trenutno zaglavlje. Navedeno zaglavlje je generirano od strane sustava refleksije. Primjer generiranog zaglavlja za klasu `CStaticMesh` prikazan je slikom Slika 1.23. Uključeno je cijelo zaglavlje radi snalaženja čitatelja ovog rada.

Prva linija kod deklaracije klase mora sadržavati makro *GENERATED* koji sadrži generirani kod.

Generirani kod sadrži implementacije nekih virtualnih metoda, popunjavanje strukture `Type` i dodatne deklaracije koje pojednostavljuju generirani kod, a mogu ih koristiti i korisnici. Deklaracija `Super<T>` među roditeljskim klasama i strukturama traži onaj tip koji je izveden iz `T`. Na primjer, kod nadjačavanja virtualne metode moramo pozvati istu metodu od klase roditelja. Korisnik može ručno napisati tip roditelja koji implementira tu metodu, no ako kasnije u promijeni deklaraciju trenutne klase tako da je naslijeđena iz neke druge klase, npr. klase naslijeđene iz prethodnog roditelja, kod će se i dalje prevoditi ali zvat će krivu metodu – preskočit ćemo poziv jedne metode i uzrokovati grešku koja sama po sebi nije očita. *Unreal Engine* ima slično rješenje ovog problema, generirano zaglavlje sadrži deklaraciju Super tipa, no to podrazumijeva da će klasa nasljeđivati samo jedan tip. Ideja implementirana u ovom pokretaču funkcionira i za višestruko nasljeđivanje sve dok nemamo

dijamantno nasljeđivanje. Unatoč tome što bismo dijamantno nasljeđivanje trebali izbjegavati zbog razloga navedenog u poglavlju 1.1.1, filozofija pokretača ostaje ista, bez ograničavanja korisnika. Dijamantno nasljeđivanje i dalje ima svoju svrhu, ako korisnik želi koristiti dijamantno nasljeđivanje, pretpostavimo da zna što radi. U budućnosti će biti razvijeno rješenje koje funkcionira i za dijamantno nasljeđivanje.

Varijable označene atributom *PROPERTY* mogu imati više oznaka:

- `Visible` - varijabla vidljiva u uređivaču
- `Edit` - varijabla vidljiva u uređivaču, može se uređivati
- `DisplayName` = "MyName" – mijenja naziv varijable u uređivaču
- `Serialize` – varijabla je automatski serijalizirana i deserijalizirana
- `Load` – učitaj resurs nakon učitavanja ovog resursa

Ako ne želimo koristiti automatsku serijalizaciju i deserijalizaciju, moramo staviti oznaku `CustomSerialization` unutar atributa *REFLECTED*. Metode `Serialize` i `Deserialize` neće biti generirane.

Podržane su i korisničke oznake. Korisnik može napisati bilo što unutar atributa *PROPERTY* i to kasnije dohvatiti u kodu.

Refleksija metoda bit će podržana u budućoj verziji. Najveće ograničenje trenutne implementacije je to što se može koristiti isključivo za klase naslijeđene iz klase Object. U budućnosti bilo koji tip će se moći reflektirati, uključujući i jednostavne strukture.

Sustav refleksije podržava enumeracije, moguće je ispisati naziv enumeracije i njezinog pojedinog elementa.

Trenutno nije moguće imati deklaraciju reflektirane klase unutar neke druge klase samo zato što implementacija te funkcionalnosti nije dovršena. Bit će dovršena u budućoj verziji.

```
#pragma once

#undef FILENAME
#define FILENAME CStaticMesh

#define GENERATED_CStaticMesh_12() \
    using ClassType = CStaticMesh; \
    template <typename Interface> \
    using Super = typename FindSuperOfType<Interface, Component>::type; \
public: \
    friend struct ObjectDeleter<ClassType>; \
public: \
    static Type* StaticType() \
    { \
        static Type* staticType = TypeRegistry::Get().CreateType<CStaticMesh, Component>() \
                ->WithPropertyMap(std::move(PropertyMap() \
                  .WithProperty("Mesh", TypeOf<UnderlyingType<AssetPtr < StaticMesh >>>(), &CStaticMesh::Mesh, {{"Edit", ""}, {"Serialize", ""}}) \
                  .WithProperty("MaterialOverride", TypeOf<UnderlyingType<AssetPtr < Material >>>(), &CStaticMesh::MaterialOverride, {{"Edit", ""}, {"Serialize", ""}}) \
                  .WithProperty("MeshTransform", TypeOf<UnderlyingType<Transform>>(), &CStaticMesh::MeshTransform, {{"Edit", ""}, {"Serialize", ""}}) \
        )); \
        return staticType; \
    } \
    \
    virtual Type* GetType() const override \
    { \
        return StaticType(); \
    } \
    template <typename... Args> \
    static SharedObjectPtr<ClassType> New(Args&&... args) \
    { \
        return SharedObjectPtr<ClassType>(ClassBucketArray.Emplace(std::forward<Args>(args)...), ObjectDeleter<ClassType>()); \
    } \
    \
    template <typename... Args> \
    static SharedObjectPtr<ClassType> New(BucketArray<ClassType>& bucketArray, Args... args) \
    { \
        return SharedObjectPtr<ClassType>(bucketArray.Emplace(std::forward<Args>(args)...), ObjectDeleter<ClassType>()); \
    } \
    virtual std::shared_ptr<Object> Duplicate() const override \
    { \
        auto newObject = std::shared_ptr<ClassType>(ClassBucketArray.Add(*this), ObjectDeleter<ClassType>()); \
        ConditionalCopyAssignUnchecked(*newObject.get(), *this); \
       \
        return newObject;\
    } \
    \
    virtual Object* DuplicateAt(void* ptr) const override \
    { \
        return new(ptr) ClassType(*this); \
    } \
    virtual void Copy(const Object& other) override \
    { \
        ConditionalCopyAssign(*this, dynamic_cast<const ClassType&>(other)); \
    } \
    \
    virtual bool Serialize(MemoryWriter& writer) const override \
    { \
        if (!Super<ISerializeable>::Serialize(writer)) \
        { \
            return false; \
        } \
        writer << Mesh; \
        writer << MaterialOverride; \
        writer << MeshTransform; \
        \
        return true; \
    } \
    \
    virtual bool Deserialize(MemoryReader& reader) override \
    { \
        if (!Super<ISerializeable>::Deserialize(reader)) \
        { \
            return false; \
        } \
        reader >> Mesh; \
        reader >> MaterialOverride; \
        reader >> MeshTransform; \
        \
        return true; \
    } \
    \
    std::shared_ptr<ClassType> SharedFromThis() \
    { \
        return std::static_pointer_cast<ClassType>(shared_from_this()); \
    } \
    \
    std::shared_ptr<const ClassType> SharedFromThis() const \
    { \
        return std::static_pointer_cast<const ClassType>(shared_from_this()); \
    } \
    \
private: \
    inline static BucketArray<ClassType> ClassBucketArray; \
private:
```

Slika 1.23 Primjer generiranog zaglavlja refleksije

Proces refleksije generira instancu klase `Type` koji sadrži informacije o reflektiranim varijablama i neke informacije o samom tipu, poput poravnanja i veličine. Navedena klasa ima mogućnost kreiranja nove instance svojeg tipa, dohvaćanje glavne instance (engl. *class default object*, *CDO*) te iteriranje po varijablama. Primjer iteriranja po varijablama prikazan je slikom Slika 1.24, gdje tražimo sve varijable tipa `AssetPtr<T>` i učitavamo ih. Ovako je implementirano automatsko učitavanje potrebnih resursa o kojima ovisi resurs koji trenutno učitavamo. U budućnosti bit će implementirana struktura `SoftAssetPtr<T>` koja neće automatski učitavati navedeni resurs.

```
GetType()->ForEachProperty( callback: [this](PropertyBase* property) ->bool
{
    const SharedObjectPtr<Asset> valueRef = dynamic_cast<Property<Asset, AssetPtrBase>*>(property)->GetRef( object: this);
    if (valueRef == nullptr)
    {
        return true;
    }

    valueRef->Load();

    return true;
});
```

Slika 1.24 Primjer iteriranja po varijablama

Kreiranje objekata (instanci klasa izvedenih iz klase `Object`) generalno vraća `SharedObjectPtr<T>` što je trenutno alias za `std::shared_ptr`, no u budućnosti će biti zamijenjen implementacijom specifičnom za ovaj pokretač. Svaka funkcija koja kreira novi objekt vraća dijeljeni pokazivač sa specifičnim objektom za brisanje (engl. *deleter*) koji briše objekt iz instance klase `BucketArray` njegovog tipa.

Klasa `Type` ima identifikator tipa i puni naziv tipa i njegovih roditelja. Moguće je iterirati po stablu nasljeđivanja – korisno za prikaz u korisničkom sučelju. Ako serijaliziramo identifikator, kod deserijalizacije možemo pronaći tip s navedenim identifikatorom te kreirati novi objekt.

Proces refleksije implementiran je kao poseban program koji se prevodi i izvodi prije glavnog programa. Program parsira promijenjena zaglavlja te generira zaglavlja refleksije te glavnu datoteku s izvornim kodom koji inicijalizira refleksiju kod pokretanja glavnog programa.

U budućnosti će biti podržana i statička refleksija, tako da je moguće iterirati po varijablama za vrijeme prevođenja.

1.9. Korisničko sučelje

Cilj prijašnjeg projekta *Korisničko sučelje za uređivač pokretača video igara u DirectX 12* bio je kreirati razvojni okvir za korisničko sučelje unutar ovog pokretača.

Implementirani su mnogi UI elementi, neki od njih su i nadograđeni u sklopu ovog projekta. Također, interakcije s UI elementima se sada mogu preciznije kontrolirati, odnosno moguće je specificirati želimo li moći kliknuti neki element. Implementiran je UI element `ScrollBox` koji omogućava pomicanje elementa djeteta kotačićem miša.

Klasa `Widget` je bazna klasa za sve UI elemente. Generalno, klase vezane uz UI elemente bi trebale biti reflektirane zato što će u budućnosti biti implementirano korisničko sučelje unutar uređivača za kreiranje novih UI elemenata.

Ideja je sljedeća: gradimo stablo UI elemenata kroz odnose roditelj-dijete. U prvom prolazu putujemo po stablu od listova do korijena i računamo željenu veličinu UI elementa. U drugom prolazu po stablu putujemo u suprotnom smjeru te osvježavamo raspored UI elemenata sada kada znamo koliko prostora nam je dostupno.

Navedena funkcionalnost implementira se metodama `UpdateDesiredSizeInternal` i `RebuildLayoutInternal` – korisnik jednostavno može kreirati novi tip UI elementa s drukčijim rasporedom.

UI elementi ekstenzivno koriste delegate za propagiranje interakcija kako bismo u ostatku koda mogli saznati npr. kada je igrač kliknuo gumb.

Elementi se iscrtavaju algoritmom BFS kako bismo zadržali redoslijed iscrtavanja. Model pravokutnika ne stvara probleme kod iscrtavanja – možemo koristiti spremnik dubine. Problem je tekst koji se u trenutnoj implementaciji ne može iscrtavati u spremnik dubine, tako da moramo paziti na redoslijed iscrtavanja, inače će tekst biti iscrtan ispred svih elemenata.

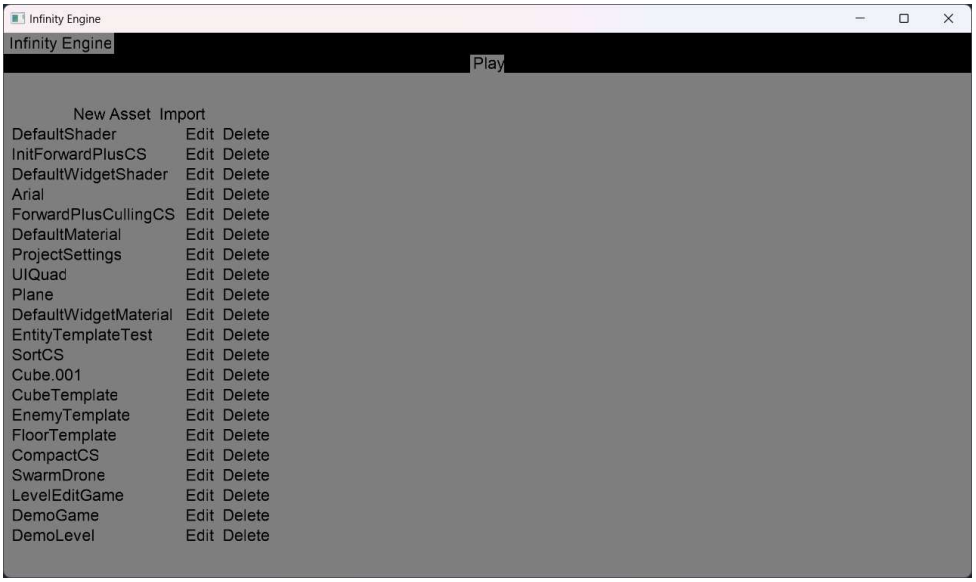
Za iscrtavanje teksta koristi se vanjska biblioteka *DirectXTK12*, specifično *bitmap* fontovi. Iz ove biblioteke se također koristi omotač oko biblioteke *DirectXMath*, SIMD matematičke biblioteke za operacije nad vektorima i matricama. *DirectXMath* je dio samog *DirectX*-a

Iscrtavanje UI elemenata nije optimizirano, svaki UI element zahtijeva jedan poziv za iscrtavanjem što je skupo i bit će optimizirano u budućnosti.

1.10. Uređivač

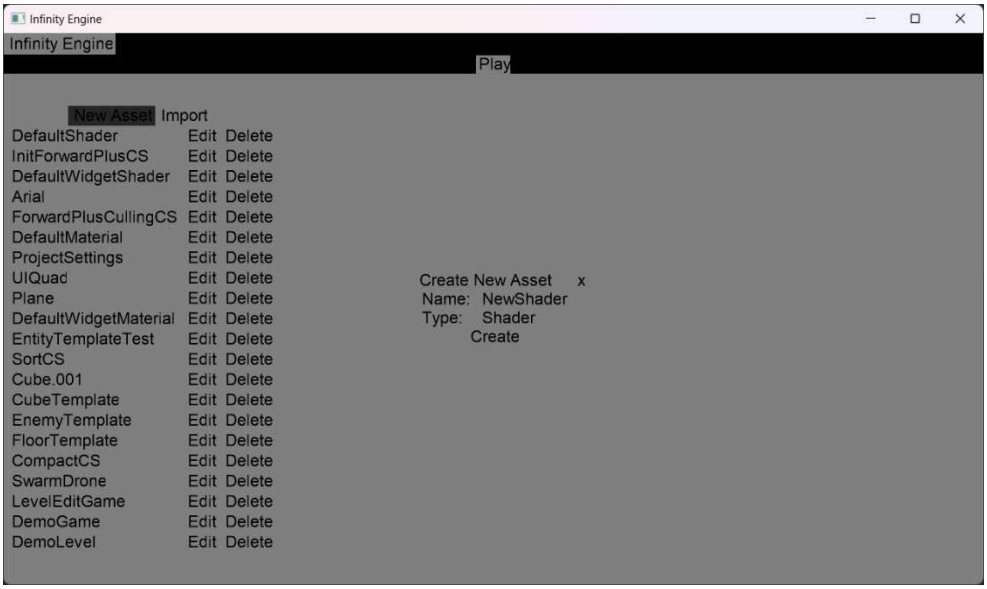
Za potrebe pokretača razvijen je uređivač za upravljanje resursima i uređivanje razina. Uređivač je kreiran korištenjem razvojnog okvira opisanog u poglavlju 1.9.

Prilikom pokretanja projekta, otvara se uređivač sa sučeljem za upravljanje resursima, prikazanim na slici Slika 1.25.



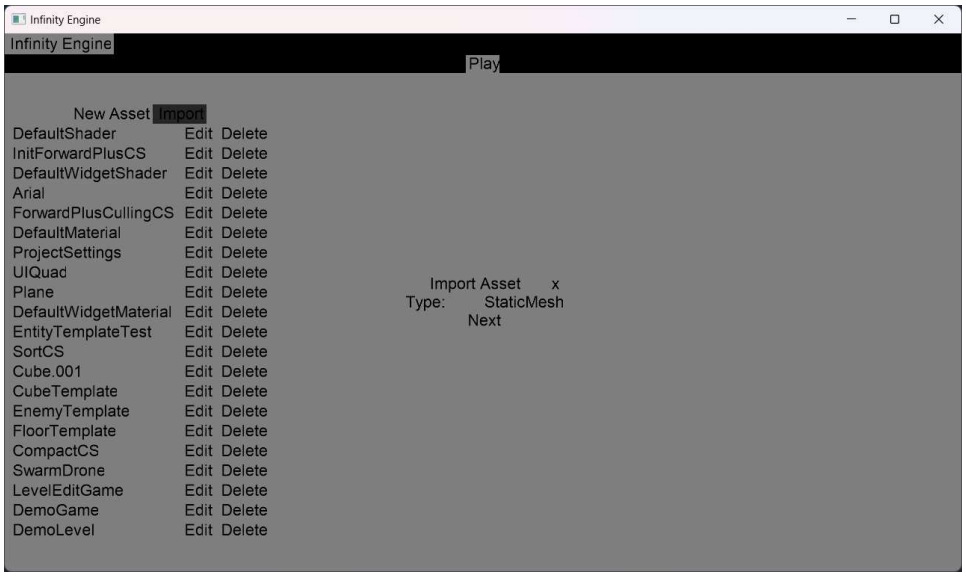
Slika 1.25 Uređivač resursa

Klikom na gumb „*New Asset*“ otvara se meni za kreiranje resursa. Potrebno je odabrati tip resursa i unijeti naziv. Ponuđeni tipovi dobiveni su refleksijom, iteriranjem po stablu nasljeđivanja tipa `Asset`. Sučelje je prikazano slikom Slika 1.26.



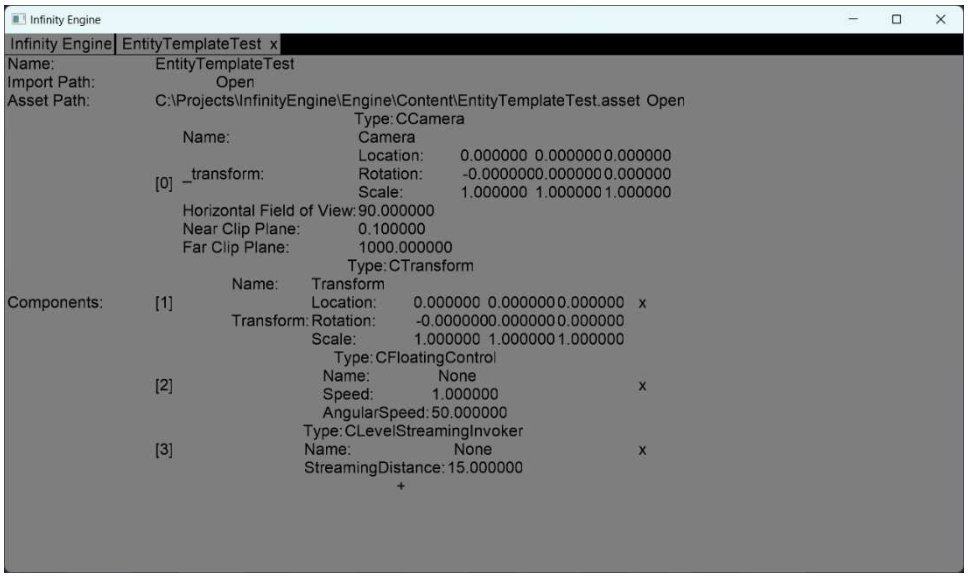
Slika 1.26 Sučelje za kreiranje novih resursa

Klikom gumba „*Import*“ moguće je uvesti resurse onih klasa koje specificiraju tip objekta za uvoz. Trenutno podržani resursi su `DX12Shader`, `StaticMesh` i `Font`. Sučelje za uvoz resursa prikazano je slikom Slika 1.27.



Slika 1.27 Sučelje za uvoz resursa.

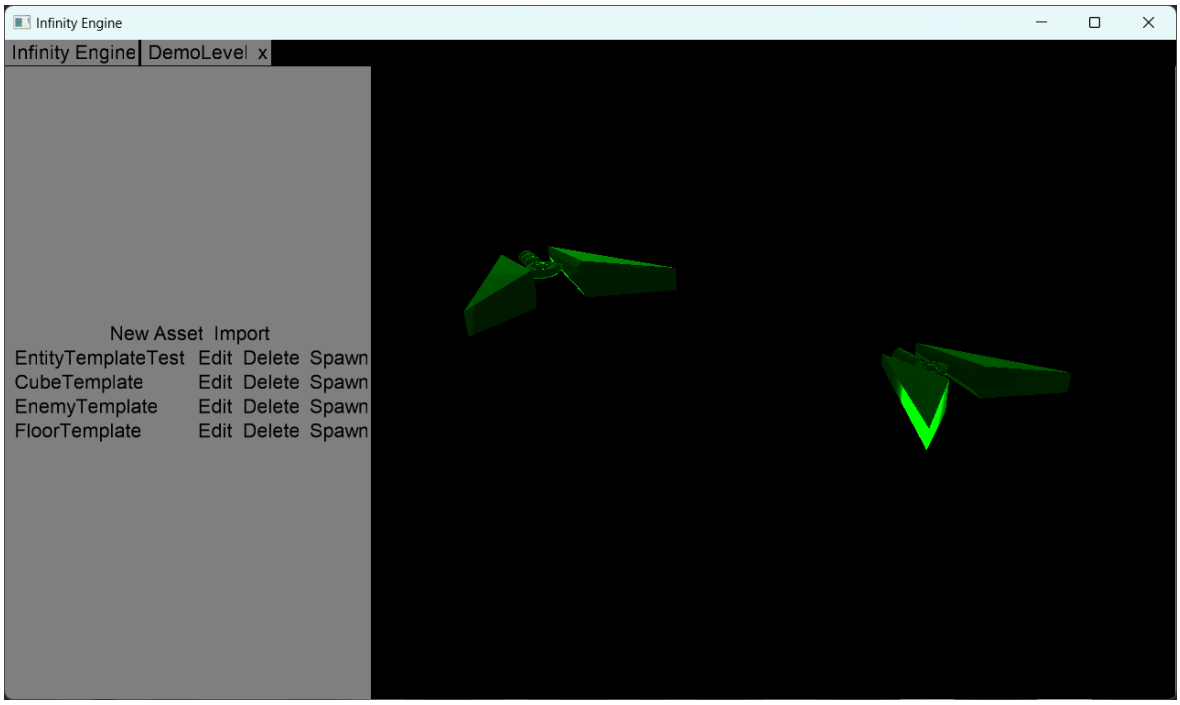
Resursi se mogu uređivati klikom gumba „*Edit*“. U novoj kartici otvara se sučelje kreirano refleksijom. Sve varijable označene s `Visible` ili `Edit` bit će prikazane u ovom sučelju. Slika 1.28 prikazuje sučelje za uređivanje predložaka entiteta.



Slika 1.28 Sučelje za uređivanje predložaka entiteta

Predložak entiteta ima polje komponenata implementirano koristeći klasu `DArray`. Za `DArray` je implementiran UI element kroz koji se može dodavati i brisati elemente. Klikom na gumb „+“ moguće je dodati novi element, u ovom slučaju komponentu. Nakon što izaberemo tip komponente, sučelje se osvježava s novim UI elementom za tu specifičnu komponentu.

Uređivač razina je implementiran kao poseban UI element koji se kreira klikom gumba „*Edit*“ za resurs tipa `Level`. Metoda `Asset::CreateEditWidget` je virtualna, tako da je korisnik može nadjačati u novim tipovima resursa. Slika 1.29 prikazuje razinu DemoLevel otvorenu u uređivaču razina. S lijeve strane vidimo listu predložaka koje možemo stvoriti. Za prikaz te liste koristi se isti UI element kao za prikaz svih resursa na glavnom ekranu, samo što pojedini element u listi ima dodatni gumb „*Spawn*“. Ova funkcionalnost implementirana je kreiranjem nove klase elemenata izvedene iz bazne. Listi je postavljen tip te klase dobiven iz sustava refleksije kao tip elementa.



Slika 1.29 Uređivač razina

S obzirom na to da je uređivač razina objašnjen u poglavlju 1.4.1, ovdje neće biti detaljnije razmatran.

Zaključak

Za ovaj rad pokrenut je velik projekt *Infinity Engine*, pokretač video igara bez ograničenja namijenjen za simulaciju vrlo velikog broja objekata u sceni. Razvijena su efikasna rješenja nekoliko velikih problema, poput performanci obrade velikog broja objekata na strani procesora, performanci obrade i iscrtavanja velikih scena, asinkrono učitavanje velikih razina te su riješeni mnogi sporedni problemi koji su se našli na putu. Razvijeno je korisničko sučelje za kreiranje i uređivanje resursa, kao i sustav refleksije koji povezuje serijalizirane podatke, uređivač i sami kod.

Objašnjene su prednosti i mane razvijenih rješenja, te su predložena bolja rješenja za neke probleme koja će biti implementirana u budućim verzijama pokretača.

Infinity Engine je velik projekt u koji je uloženo mnogo truda, no prema rezultatima, trud se isplatio, te će se rad na projektu nastaviti i nakon ovog rada.

Literatura

[1] Cameron, *moodycamel::ConcurrentQueue*. Poveznica: https://github.com/cameron314/concurrentqueue; pristupljeno 11. svibnja 2024.

[2] Luna, F.D. *Introduction to 3D Game Programming with DirectX 12*. 1. izdanje. Mercury Learning and Information, 2016

[3] Walbourn, C., *Compute Shader Sort*. Poveznica oveznica: https://github.com/walbourn/directx-sdk-samples/tree/main/ComputeShaderSort11; pristupljeno 27. travnja 2024.

[4] Jeremiah, *Forward vs Deferred vs Forward+ Rendering with DirectX 11*. Poveznica: https://www.3dgep.com/forward-plus; pristupljeno 1. svibnja 2024.

[5] Akenine-Möller, T; Haines, E; Hoffman, N; Pesce, A; Iwanicki, A; Sébastien, Hillaire, S. *Real-Time Rendering*. 4. izdanje. Boca Raton, FL, USA: A K Peters/CRC Press, 2018.

[6] Potasky, J. *Gilbert-Johnson-Keerthi Distance Algorithm*. Poveznica: https://cse442-17f.github.io/Gilbert-Johnson-Keerthi-Distance-Algorithm; pristupljeno 18. svibnja 2024.

[7] Olvång, L. *Real-time Collision Detection with Implicit Objects*. Poveznica: http://uu.diva-portal.org/smash/get/diva2:343820/FULLTEXT01; pristupljeno 27. svibnja 2024.

[8] Amanatides J., Woo. A. *A Fast Voxel Traversal Algorithm for Ray Tracing*. Poveznica: http://www.cse.yorku.ca/%7Eamana/research/grid.pdf; pristupljeno 27. svibnja 2024.

[9] Assimp. *Open Asset Import Library*. Poveznica: https://github.com/assimp/assimp; pristupljeno 5. listopada 2023.

Sažetak

Naslov: Prikaz velikih virtualnih svjetova

Ključne riječi: *ECS*, instanciranje, višedretvenost, asinkrono učitavanje

U sklopu ovog projekta razvijena su rješenja za nekoliko ključnih problema kod izrade pokretača video igara.

Demonstrirano je efikasno i intuitivno rješenje za simulaciju velikog broja objekata na procesoru koristeći princip *Entity-Component-System*. Implementirani sustav je robustan te koristi višedretvenost koju korisnik neće niti primijetiti.

Implementirana je efikasna obrada velikih scena na grafičkoj kartici koristeći novu funkcionalnost *DirectX*-a *12*, grafove poslova, i računske sjenčare. Obrada scene rezultira podacima za minimalan broj poziva iscrtavanja – koriste se instancirani pozivi za iscrtavanje.

Implementiran je jednostavan sustav za simulaciju fizike koji podržava simulaciju sudara, traženja presjeka zrake i scene, te ima djelomičnu podršku za preklapanja.

Implementirano je asinkrono učitavanje velikih razina te uređivač za razine, kao i mnoge pomoćne strukture i sustavi koji povezuju sve navedeno u cjelinu.

Summary

Title: Simulation of large virtual worlds

Keywords: *ECS*, instancing, multithreading, asynchronous loading

This project contains solutions for several key problems that arise when developing a game engine.

Project demonstrates efficient and intuitive solution for simulating a large number of objects on the processor using Entity-Component-System principle. Implemented solution is robust and utilizes multithreading in a way that user will not even notice.

A solution for efficient preprocessing of large scenes was developed using GPU-driven rendering. Solution utilizes a new feature of *DirectX 12*, work graphs, compute shaders and GPU instancing. Result of preprocessing is data for minimum number of instanced draw calls.

Project contains a simple solution for physics simulation, which supports collision simulation, ray cast tests and partially supports overlaps.

Project contains an implementation of level streaming, a way to asynchronously load parts of large scenes.

Many other problems were solved along the way and many other structures and systems were implemented that bind all parts together.

Skraćenice

ECS	*Entity-Component-System*	tip arhitekture pokretača video igara
LOD	*Level of detail*	razina vizualnih detalja
HLSL	*High Level Shading Language*	programski jezik za pisanje sjenčara
CSM	*Cascaded Shadow Maps*	metoda izračuna sjena
GJK	*Gilbert-Johnson-Keerthi*	algoritam za detekciju preklapanja
EPA	*Expanding Polytope Algorithm*	algoritam za izračun parametara sudara
CRTP	*Curiously Occuring Template Pattern*	oblikovni obrazac u jeziku C++
CDO	*Class Default Object*	glavni objekt neke klase
UI	*User Interface*	korisničko sučelje

Privitak

Instalacija programske podrške

Izvorni kod dostupan je na poveznici: https://github.com/WingmanD/InfinityEngine

Potrebno je klonirati ili skinuti projekt.

Prije instalacije potrebno je instalirati:

CMake 3.12 ili noviji

Conan 1.59 ili noviji verzije 1.X (Conan 2.0+ nije podržan)

Visual Studio 2022 ili noviji (ili neki drugi sustav po želji, potrebna je izmjena skripte za generiranje projekta)

Nakon toga potrebno je pokrenuti skriptu *GenerateProjectFiles.ps1* koja će generirati direktorij *Build*. Unutar tog direktorija nalazit će se datoteka *InfinityEngine.sln* koju je moguće otvoriti kroz *Visual Studio* ili *Rider*.

Nakon otvaranja navedene datoteke, potrebno je postaviti projekt *Engine* kao projekt za pokretanje (*Set as Startup Project* u *Visual Studiju*) te pokrenuti prevođenje.