Razvoj programske knjižnice za kreiranje interaktivnih virtualnih okruženja na web-aplikacijama

Fuček, Matija

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1586

RAZVOJ PROGRAMSKE KNJIŽNICE ZA KREIRANJE INTERAKTIVNIH VIRTUALNIH OKRUŽENJA NA WEB-APLIKACIJAMA

Matija Fuček

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1586

RAZVOJ PROGRAMSKE KNJIŽNICE ZA KREIRANJE INTERAKTIVNIH VIRTUALNIH OKRUŽENJA NA WEB-APLIKACIJAMA

Matija Fuček

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1586

Pristupnik:	**Matija Fuček (0036533249)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Mario Brčić

Zadatak: **Razvoj programske knjižnice za kreiranje interaktivnih virtualnih okruženja na web-aplikacijama**

Opis zadatka:

Cilj ovog završnog rada jest izrada programske knjižnice koja će pojednostaviti i ubrzati izradu različitih tipova virtualnih iskustava u sklopu web-aplikacija, uključujući, ali ne ograničavajući se na video igre, umjetničke instalacije te razne oblike vizualizacija. Biblioteka treba biti izgrađena na temeljima popularnog razvojnog okvira ReactJS, pomoću kojeg treba ostvariti interakciju putem tipkovnice i miša, kao i komunikaciju između virtualne scene i korisničkog sučelja. Virtualnu scenu treba implementirati pomoću WebGL tehnologije koristeći biblioteku ThreeJS. Primarni fokus razvijene biblioteke bit će na efikasnom upravljanju virtualnim scenama te entitetima unutar samih scena.

Rok za predaju rada: 14. lipnja 2024.

Zahvalio bih se svojem mentoru, te kolegama na pomoći pri izradi ovog rada.

Sadržaj

1. Uvod

1.1. Motivacija

Značajan problem tijekom započinjanja novih projekata u području računalne grafike - bilo rada na igrama, stvaranju interaktivnih karti, ili generalno bilo kojeg tipa vizualizacije podataka - leži u postavljanju osnovne funkcionalnosti na temelju kojih bi se dalje gradila kompleksnija ponašanja programa.

Cilj ovog rada je upravo u smanjenju barijere pri započinjanju novih projekata s naglaskom na minimalnu količinu koda tzv. “boilerplate”-a. Osim samih osnovnih funkcionalnosti, u biblioteci koju ovaj rad predlaže i opisuje, cilj mi je bio osigurati postojanje određenih kompleksnijih funkcionalnosti koje se mogu neobavezno uporabljavati.

Vođen osobnim iskustvom, do sada sam se oslanjao na pisanju projekata koristeći objektno-orijentirano programiranje. Međutim, korištenje čistog OOP-a je često dovodilo do izuzetno puno “boilerplate”-a i teže integracije s kompleksnijim sustavima, te je često bilo problematično iz perspektive iskustva korištenja tj. “*developer experience*”-a.

Iz ovih razloga, kao cilj ovog Završnog rada odabrano je kreiranje, implementacija i opis novog radnog okvira za kreiranje interaktivnih virtualnih okruženja na web-aplikacijama koje će biti temeljeno na principima sličnijim funkcionalnom programiranju, odnosno temeljeno na ponašajnim obrascima.

Uspješno kreiranje i implementacija ovakvog okvira predstavlja nekoliko važnih izazova, od kojih su najvažniji: korištenje [1, Typescripta] kao *de-facto* standarda u modernom web developmentu, osiguranje jednostavnosti korištenja te mogućnosti ponovnog korištenja koda (”code reusability”), kao i povezivanja s već postojećim bibliotekama

koje se koriste pri izradi web projekata.

Ostatak ovg rada objasnit će kako su ovi ciljevi postignuti predlaganjem i implementacijom nove biblioteke pod nazivom “Ebon”, te će njene sastavnice i primjeri uporabe biti prikazani na konkretnom primjeru izrade jednostavnog 3D okruženja u kojem korisnik upravlja avatarom u interaktivnoj sceni.

1.2. Odabir jezika

Javascript sam po sebi nema mogućnost dodjeljivanja tipova podataka na varijable, zato je osmišljen jezik Typescript koji se pri kompajliranju (odnosno transpiliranju) pretvara u običan Javascript, kako bi on bio upogonjiv u web okruženjima kao što su web preglednici koji samo znaju tumačiti Javascript.

Biblioteka je, zbog naglaska na sigurnost tipova podataka, stoga pisana u jeziku Typescript, obzirom da se jezik može koristiti za programiranje programa koje se mogu pokretati na mnoštvu okruženja - od web preglednika, nativnih aplikacija za Windows, Mac, Linux, te mobilnim okruženjima. Osim toga, Typescript pruža mogućnost dodjeljivanja tipova podataka varijablama, što olakšava razvoj i održavanje koda.

Osim raširenosti samog Typescripta, jezik je odabran obzirom da nije postojalo slično rješenje za problematiku koju ovaj rad pokušava riješiti, barem ne na razini modularnosti i fleksibilnosti koja je ciljana.

1.3. Ciljevi rada

1.3.1. Jednostavnost korištenja

Bitna značajka svake biblioteke koja je osmišljena za širu uporabu je to da je lagana za korištenje. Kada se ne uporabljava na ciljani način da je IDE (integrirano razvojno okruženje) krajnjeg programera zna upozoriti programera, te predložiti što programer mora ućiniti da postigne ciljano ponašanje programa.

Naravno, dokumentacija biblioteke pomaže krajnjem programeru da točno zna što funkcija koju poziva radi, i koji su mogući načini korištenja. U sklopu rada se pazilo

da brojne bitne funkcije uz sebe imaju tzv. “pragma komentare” koji se prikazuju pri držanju pokazivaća iznad same funkcije.

U lakoći korištenja dosta se oslanjalo na Typescriptove mogućnosti upravljanja tipovima podataka. Primjerice, pri pozivanju određenih metoda, pazilo se da ona može isključivo primati određene parametre, te se u samom izvornom kodu vodila pažnja da se dosljedno prate predani tipovi, bez utvrđivanja tipova na silu ili tzv. “assertion”, što je osiguralo kompletnu sigurnost tipova podataka.

1.3.2. Ponovna uporaba koda

U ovoj biblioteci isto je tako naglasak na mogućnost pisanja atomičnih cjelina koje su namjenjene za višestruku ponovnu uporabu, pa tako smanjenju koliko je koda potrebno napisati kako bi se isprogramirala kompleksnija rješenja.

Jedan od načina kako je ova značajka implementirana uključuje kompostabilnost i ulančavanje funkcija o kojima će više biti rečeno u kasnijim dijelovima rada.

1.3.3. Mogućnost povezivanja s postojećim bibliotekama

S obzirom na to da se Typescript transpilira u Javascript, biblioteku je moguće koristiti i u najjednostavnijim web okruženjima, odnosno bez nužnog korištenja tipova podataka. Osim toga, bitno je napomenuti da je Typescript/Javascript ekosustav velik, te pruža razne treće biblioteke koje bi se mogle koristiti u izradi programa.

2. Osnovne sastavnice biblioteke Ebon

U ovom dijelu rada ćemo se prvenstveno fokusirati na objašnjavanje funkcionalnosti. Zatim ćemo prikazati predložene i implementirane funkcionalnosti biblioteke na primjeru jednostavnog okruženja.

2.1. Vanjske biblioteke

Za početak, važno je napomenuti bitnije vanjske biblioteke koje su korištene i njihovu ulogu.

2.1.1. Three.js

Kako se sama biblioteka bazira na stvaranju 3D virtualnih okruženja, potrebno je bilo ustanoviti kojim će se načinom oblici crtati na zaslonu, po mogućnosti korištenjem grafičke kartice. Već postoje standardni jezici baš za tu svrhu, kao što je OpenGL, međutim pokretanje istih kroz preglednik je teško, osim ako nije riječ o WebGL-u.

[2, WebGL] se pokazao zahtjevan, te bi njegovo direktno korištenje znatno povećao opseg biblioteke. [3, Three.js] je korišten kao apstrakcija WebGL-a koja dozvoljava upotrebom Typescripta definiciju geometrije, materijala, osvjetljenja, kamera, i drugih elemenata potrebnih u stvaranju virtualnih scena.

2.1.2. React i React DOM

Bez korisničkog sučelja, kreirana virtualna okruženja ne bi imala puno koristi, pored naravno umjetničkih instalacija, ili statičkih vizualizacija. Iz ovog su razloga odabrani React i React DOM, kao jedni od popularnijih biblioteka za stvaranje i upravljanje 2D

sučelja na temelju kojih funkcionira veliki dio modernih web stranica.

Sama biblioteka je namjenjena da se koristi u React okruženju, te putem JSX komponente se u biti postavlja na inače praznu stranicu.

Pomoću Reacta, biblioteka omogućuje prikazivanje jednostavnih prozora, smještenih relativno na sam kontejner. No, cilj je bio omogućiti i pozicioniranje prozora relativno na 3D svijet virtualne scene, te da se sami prozori pomiću s obzirom na pomicanje virtualne kamere. Ova funkcionalnost postignuta je komunikacijom biblioteke s trećom vanjskom bibliotekom - Zustand.

2.1.3. Zustand

Biblioteka Zustand je sama po sebi odvojena od ostalih sustava na tipičnoj web stranici, pa tako i od Reacta. Zadaća Zustanda je upravljanje globalnim stanjem web stranice. Posebna mogućnost Zustanda je mogućnost komunikacije s Reactom, i to korištenjem Reactovih "hookova", koji se uštekaju u tzv. "lifecycle" komponenti. Ukratko, kada naša biblioteka promjeni vrijednost globalnog stanja, Zustand zna reći Reactu da osvježi komponentu koja koristi to stanje.

2.2. Ponašajna arhitektura

Osnova biblioteke leži u tzv. "ponašajnoj arhitekturi", koju čine dva glavna objekta: ponašanje (”Behavior”) i entitet (”Entity”). Uz pomoć ponašajne arhitekture, programeri mogu stvoriti kompleksne sustave s mnogo entiteta koji se ponašaju na željeni način.

Ponašajna arhitektura omogućuje modularnost, fleksibilnost i ponovnu upotrebu koda. Također olakšava razumijevanje i nadzor ponašanja entiteta, jer su pravila jasno definirana unutar ponašanja.

Ukratko, ponašajna arhitektura osigurava organizaciju i upravljanje ponašanjem entiteta, što rezultira modularnim i fleksibilnim softverskim sustavom.

2.2.1. Ponašanje

Ponašanje, ili 'Behavior', predstavlja skup pravila, funkcionalnosti i akcija koje definiraju kako se entitet ponaša u određenim situacijama.

Ponašanje opisuju četiri ključna aspekta. Prvo, inicijalno stanje koje entitet ima pri instanciranju. Drugo, logika koja se odvija pri samom instanciranju entiteta. Treće, logika koja se događa prilikom osvježavanja stanja entiteta. I četvrto, popis akcija koje se mogu izvana pozvati, a koje utječu na stanje entiteta.

Ova komponenta omogućuje precizno definiranje karakteristika i obrasca ponašanja entiteta u različitim situacijama.

2.2.2. Entitet

Entitet je objekt koji posjeduje određeno ponašanje i ponaša se u skladu s tim pravilima. Entitet nastaje pri instancijaciji odabranog ponašanja i može se opisati kao skup već postojećih ponašanja.

Ponašanja u principu predstavljaju nacrt, koji se može u bilo kojem trenutku instancirati i pretvoriti u živi objekt, odnosno entitet.

3. Izvedba ponašanja

Ponašanje je izvedeno kroz Behavior klasu. Sama klasa sadrži četiri generična tipa podatka: State, Actions, RequiredState, te RequiredActions.

Ponašanje je osmišljeno da se oblikuje nizanjem poziva metoda, nalik "builder" odnosno "middleware" oblikovnom obrascu. Metode korištene u oblikovanju ponašanja čine init, tick, use, require, te action. A za samu instancijaciju ponašanja, odnosno pretvaranja ponašanja u entitet se koristi metoda create.

Kada se klasa Behavior kreira, u sklopu iste se inicijalizira unikatan identifikator koji će kasnije biti od važnosti u prepoznavanju postojanja određenog ponašanja u entitetima.

Pored identifikatora, inicijaliziraju se i dva bitna funkcijska cijevovoda: initCallback, te tickCallback koji predstavljaju funkcije koje pri pozivanju ili generiraju stanje entiteta ili ažuriraju stanje entiteta s novim vrijednostima.

Pri instancijaciji ponašanja se svaka funkcija u danom cijevovodu initCallback poziva redom, te se rezultat izvođenja jedne funkcije proslijeđuje dalje u iduću funkciju. Rezultat izvođenja initCallbacka čini inicijalno stanje entiteta, više o tome u poglavlju vezanom uz entitete 2.2.2.

3.1. Osnovne metode ponašanja

3.1.1. Metoda inicijalizacije stanja

Init metoda klase Behavior prima jedan parametar koji se zove callback te predstavlja anonimnu funkciju koja se poziva pri instancijaciji ponašanja.

Parametar callback, kao funkcija, prima jedan parametar koji predstavlja prethodno

stanje (tipiziran je generikom State), te po mogućnosti vraća objekt koji bi se nadodao na prethodno stanje.

U pozadini, pri pozivanju init metode, proslijeđena anonimna funkcija se stavlja u cijevovod initCallback, te rezultat izvođenja ovog konkretnog callbacka će biti dostupan callbacku idućem po redu, obzirom da je moguće zaredati nekoliko poziva metode init na istom ponašanju.

```
const behavior = new Behavior()
  .init((prevState) => ({
    foo: 'bar'
  }))
  .init((prevState) => ({
    bar: 'foo'
  }))
  .create();
```

Pozivanje init metode vraća novu instancu klase Behavior s State generikom jednakim trenutnom State generiku, ali proširenim tipom vraćene vrijednosti.

Razlog zašto se vraća nova instanca klase "Behavior" je jer Typescriptov sustav tipova nadodan na inaće ne-tipizirani Javascript, i ne postoji provjeravanje tipova podataka tijekom izvođenja programa, stoga se moramo oslanjati na statičku generaciju tipova pri koraku transpiliranja.

Bitno je napomenuti da rezultat izvođenja callbacka mora biti rječnik, obzirom da je stanje u entitetu rječnik. Ukoliko se dogodi da vraćeni rječnik sadržava ključeve već definirane u prethodnim init pozivima, vrijednost i tip podatka vezan uz taj ključ će se prebrisati. Zato je važno obratiti pozornost da se ključevi ne mjenjaju nekompatibilnim tipovima, već da se tip proširuje (primjerice ako je ključ "object" te sadrži rječnik, bitno je proširiti taj rječnik umijesto obrisati već postojeće vrijednosti). Važnost ovoga će se vidjeti u idućem poglavlju.

3.1.2. Metoda ažuriranja stanja

Druga bitna metoda korištena za oblikovanje je "tick". Slično kao "init", ona prima anonimnu callback funkciju koja se nadodaje u cjevovod "tickCallback". Isto kao callback u "init" metodi, "tick" callback prima prethodno stanje kao parametar. Međutim, za razliku od "init" callbacka, rezultat izvođenja "tick" callbacka mora vratiti podskup prethodnog stanja, uzimajući u obzir tipove podataka. Metoda tick ne proširuje generički tip State.

```
const Ponasanje = new Behavior().tick((state) => {
  // state: {}
  console.log('Hello!');
});
```

```
const Ponasanje = new Behavior()
  .init((state) => {
    // state: {}
    return { starost: 1, ime: 'Bob' };
  })
  .tick((state) => {
    // state: {starost: number, ime: string}
    console.log(`Pozdrav, ${state.ime}!`);
    return { starost: state.starost + 1 };
  });
```

3.2. Instanciranje ponašanja i LiveEntity

Sada smo upoznati s osnovnim metodama kreacije i oblikovanja ponašanja. Ponašanje samo po sebi je nalik šabloni; drugim riječima, potrebno je pretvoriti ponašanje (šablonu) u entitet koja će doista imati svoje unutarnje stanje i koja će se moći voditi po pravilima opisanim u pripadajućem ponašanju.

Za instanciranje ponašanja se koristi metoda "create". Metoda "create" prima instancu scene kao parametar (o kojoj ćemo više govoriti u kasnijem poglavlju 4.1.). Također, kao neobavezan dodatak, možemo proslijediti rječnik stanja koji će nadjačati stanje postavljeno putem inicijalnog poziva "initCallback" cjevovoda.

Poziv metode “create” stvara se instanca klase “LiveEntity”. Pri instancijaciji klase “LiveEntity” se stvara stvarno svojstvo “state” koji predstavlja rječnik popunjen po pravilima ponašanja od kojeg je nastala instanca.

U sklopu ove instance se također sabire cijeli “tickCallback” cjevovod, te se otvara mogućnost pozivanja metode “executeTick” za ažuriranje pravog stanja u tzv. “runtime”-u.

3.3. Komunikacija entiteta

Često želimo imati više entiteta prisutnih na istoj sceni, gdje svaki entitet ima svoja svojstva i mogućnost međusobne komunikacije. Ponekad je dovoljna indirektna komunikacija, gdje prisutnost ili vrijednost stanja nekog entiteta utječe na logiku ponašanja. No, ponekad je potrebna i direktnija komunikacija, gdje jedan entitet eksplicitno traži od drugog da nešto učini ili promijeni obrazac ponašanja.

3.3.1. Međukomunikacija entiteta

U svrhu ovog, razrađena je metoda "action" unutar klase "Behavior". Sama metoda "action" prima rječnik lambda funkcija kao parametar. Svaka od lambda funkcija u rječniku prima trenutno stanje kao parametar, kao i proizvoljne dodatne parametre koje će pozivatelj morati unjeti pri pozivu akcije. Svaka lambda funkcija također vraća rječnik koji može, ali i ne mora, sadržavati ključeve "state" i "output". Vrijednost iza ključa "state" se dodaje na postojeće stanje entiteta nakon poziva, dok se vrijednost iza ključa "output" vraća pozivatelju.

```
// Ponašanje s akcijama
const Cave = new Behavior()
  .init(() => ({ numberOfYells: 0 }))
  .action({
    yellInto: (state, message: string) => {
      // Umnožavanje poruke
      const jeka = [message, message, message].join(" ");
      return {
        state: {
          numberOfYells: state.numberOfYells + 1,
        }
        output: jeka;
      }
    },
    whisperInto: (state, message: string) => {
      return {
        output: "tišina..."
      }
    }
  })

// referenca na entitet (instancirano ponašanje)
const caveRef = Cave.create();

// Pozivatelj
const Player = new Behavior()
  .init((state) => {
    const jeka = caveRef.actions.yellInto("Jekaa!");
    // jeka: string (Jekaa! Jekaa! Jekaa!)
  });

Player.create();
```

Popis dostupnih akcija u ponašanju opisan je u generičkom tipu "Actions". Nizanjem više poziva metode "action", taj generički tip se proširuje. Nakon poziva metode "action", poziv metode "init" također mijenja tip "Actions" tako da mijenja tip stanja (koji je povezan s parametrom "state" u callback funkciji) u skladu s novim proširenim tipom stanja.

3.3.2. Čišćenje akcija pri instancijaciji

Pri definiciji akcija vidjeli smo kako akcija, opisana u rječniku proslijeđenom u "action" metodu, prima i varijablu stanje i proizvoljne parametre specifične akciji.

Pri pozivanju akcija, ideja je da se ne mora ručno proslijediti stanje nekog "LiveEntity" objekta, već je poželjno da se to automatski odradi. Isto tako kod povratna vrijednost poziva akcije, nije nužno poželjno dobiti i novo stanje tog entiteta uz "output", osim u izuzetnim situacijama, već nam je interesantna vrijednost same "output" vrijednosti.

Osim mogućnosti inicijalnog postavljanja i opetovanog ažuriranja stvarnog stanja kod instance klase "LiveEntity", definirane akcije ponašanja se čiste i omogućuje se njihov vanjski poziv takav da je potrebno proslijediti samo one proizvoljne parametre, i da rezultat izvođene akcije doista je samo vrijednost iza ključa "output".

```
const SimpleEntity = new Behavior()
  .init(() => ({ lastAdd: 0 }))
  .action({
    add: (state, a: number, b: number) => {
      return {
        state: { lastAdd: a + b }
        output: a + b;
      }
    }
  })
  .create(scene)

// sirova akcija
const resultRaw = SimpleEntity._rawActions.add(
  { lastAdd: 2 },
  1,
  2
);
// resultRaw: { state: (...), output: 3 }

// očišćena akcija
const resultClean = SimpleEntity.actions.add(1, 2);
// resultClean: 3
```

3.3.3. Samokomunikacija entiteta

Osim mogućnosti da jedan entitet pozove akciju pridruženu drugom entitetu, moguće je da entitet sam pozove akciju na sebi. Kako bi entitet bio svjestan samog sebe, potrebno je naslijediti ponašanje SceneReference. Više o nasljeđivanju u sljedećem poglavlju 3.4.

Međutim, ako se akcija poziva unutar metode init, zbog redoslijeda izvršenja i inicijalizacije samih akcija pri instancijaciji ponašanja, potrebno je pozvati akciju putem pomoćne funkcije deferAction.

```
const Example = new Behavior()
  .use(SceneReference)
  // state: { scene: (...), this: (...) }
  .init(() => ({ count: 0 }))
  .action({
    increment: (state) => {
      // state: { scene: (...), this: (...), count: number }
      return {
        state: { count: state.count + 1 }
      };
    }
  })
  .init((state) => {
    // u initu je potreban deferAction
    deferAction(() => {
      state.this.actions.increment();
    });
  })
  .tick(() => {
    // radi normalno
    state.this.actions.increment();
  });
```

3.4. Izvedba polimorfizma

Osim ulančavanja init i tick poziva, sama je ponašanja moguće ulančavati, što se postiže pozivanjem metoda use i require.

3.4.1. Metoda nasljeđivanja

Kod korištenja funkcionalnosti jednog ponašanja u drugom, stvarno dolazi do izražaja naglasak na ponovnoj iskoristivosti koda.

Primjer jednog ponašanja može biti za opisivanje položaja u sceni, stoga nebi imalo smisla ponavljati kod za postavljanje položaja u stanju, kada se ista funkcionalnost, odnosno ponašanje, može iskoristit u sklopu drugog ponašanja.

Metodu use možemo iskoristiti da iskoristimo “initCallback” i “tickCallback” od drugog ponašanja, te da proširimo i generičan tip Stanja na temelju vrijednosti generičnog tipa u iskorištenom ponašanju.

```
const Position = new Behavior().init(() => {
  return {
    position: { x: 0, y: 0, z: 0 }
  };
});

const Player = new Behavior().use(Position).tick((state) => {
  // state: { position: { x: number, y: number, z: number } }
  const { x, y, z } = state.position;
  const udaljenost = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2);
  console.log(`Nalazim se ${udaljenost}m od ishodišta!`);
});
```

3.4.2. Metoda postavljanja zavisnosti

Kod ranije spomenutog primjera ponašanja položaja koje bismo iskorištavali u drugim ponašanjima, ponekad želimo opisati sustave koji su zavisni na drugima.

Nadovežimo se na primjer s ponašanjem brzine, gdje želimo opisati brzinu kao vektor, te aplicirati istu na položaj. Require je metoda koju bismo iskoristili da u ponašanju brzine eksplicitno navedemo potrebu za prethodnim postojanjem položaja.

```
const Velocity = new Behavior().require(Position).init((state) => {
  // state: { position: { x: number, y: number, z: number } }
  return {
    velocity: { x: 0, y: 0, z: 0 }
  };
});

const Player = new Behavior()
  .use(Position)
  .use(Velocity)
  .tick((state) => {
    /* state: {
        position: { x: number, y: number, z: number },
        velocity: { x: number, y: number, z: number }
      }
    */
  });
```

Korištenje metode “require” proširuje generičan tip “RequiredState” i “RequiredActions” redom s generičnim tipovima “State” i “Actions” iz ponašanja o kojem zavisimo. Ovo nam omogućuje da pretpostavimo dostupnost podataka u stanju, te dostupnost akcija u izvođenju daljnje logike.

Zahvaljujući Typescriptu, krajnji programer kada koristi jedno ponašanje (metodom “use”) u drugom dobiva povratnu informaciju ukoliko nije prethodno bilo iskorišteno ponašanje o kojem dotično ponašanje ovisi, kao u sljedećem primjeru:

```
const Velocity = new Behavior().require(Position);

const Player = new Behavior()
  .use(Velocity)
    // Ovdje bi se pojavila greška nepodudaranja tipova, obzirom da prethodno
    ↪ nije korišteno ponašanje Position.
  .tick((state) => {
    // state: any
  });
```

3.5. Ugrađena ponašanja

Kako bi se ubrzao proces stvaranja funkcionalnih programa, u sklopu biblioteke “Ebon” se nalazi nekoliko korisnih, već sastavljenih ponašanja koja su spremna za uporabu.

3.5.1. Delta

Ponašanje “Delta” proširuje stanje sa svojstvima “delta” - vrijeme u sekundama proteklo od prošlog ciklusa ažuriranja, te “age” - starost entiteta u sekundama, odnosno vrijeme proteklo od njegova instanciranja.

U ovom primjeru koristimo ponašanje "Delta" kako bismo dobili trenutnu deltu, odnosno starost. Zatim, koristimo taj podatak kako bismo izračunali novu rotaciju entiteta.

```
const Player = new Behavior()
  .use(Delta)
  .init(() => ({
    rotation: 0,
    rotationSpeed: Math.PI // Brzina rotacije u radijanima po sekundi
  }))
  .tick((state) => {
    // state: { age: number, delta: number, (...) }
    const { rotation, rotationSpeed, age, delta } = state;

    // primjer računanja rotacije putem "age" ili "delta":
    const newRotation = rotationSpeed * age;
    const newRotation = rotation + rotationSpeed * delta;



    return {
      rotation: newRotation;
    }
  });
```

Svaki put kad se izvrši tick funkcija, rotacija će se povećati za izračunati kut rotacije na temelju brzine rotacije i proteklog vremena. Time postižemo postupno rotiranje entiteta ovisno o vremenu.

3.5.2. SceneReference

Ponašanje “SceneReference” proširuje stanje sa svojstvima “scene” - referenca na scenu, te “this” - referenca na sam entitet.

Referenca na scenu nam omogućava na primjer da pristupimo listi aktivnih entiteta, pa tako dinamički stvaramo reference na druge entitete, te djelujemo na njih putem akcija ili uvjetujemo ponašanje u ovisnosti o okolini.

```
const Player = new Behavior().use(SceneReference).tick((state) => {
  // state: { scene: (...), this: (...) }
  const entities = state.scene.entitites;

  let nearestEntity: AnyLiveEntity | null = null;
  let nearestDistance = Infinity;

  // pronađi entitet najbliži ishodištu
  for (const entity in entities) {
    if (
      entity.has(Position) &&
      entity.state.position.length() < nearestDistance
    ) {
      nearestDistance = entity.state.position.length();
      nearestEntity = entity;
    }
  }
});
```

Referenca na vlastiti entitet, “this”, dozvoljava entitetima da imaju mogućnost pozivanja akcija nad samim sobom. Primjer samopozivanja akcija moguće je pronaći u poglavlju “Samokomunikacija entiteta” 3.3.3.

3.5.3. Threejs primitivi

S obzirom na to da je srž biblioteke “Ebon” u kreaciji virtualnih svjetova, teško da bi se to moglo postići bez osnovnog pristupa kreaciji objekata koji čine takve svjetove.

Iz tog razloga u sklopu biblioteke “Ebon” postoji nekoliko osnovnih ponašanja koja

inicijaliziraju objekte kompatibilne s Three.js-ovom scenom:

3.5.4. EmptyObject

EmptyObject je ponašanje koje dodaje stavku "object" u stanje entiteta. Prazan Three.js objekt u sebi sadržava sve bitno na temelju čega bi se mogao prikazivati u sceni, pa tako se sam tip stavke "object" može proširiti s raznim Three.js primitivima koji nisu već uključeni u "Ebon" biblioteku.

```
const BlankObject = new Behavior()
  .use(EmptyObject)
  .init((state) => {
    // state: { object: (...) }
    return {
      object: new THREE.AmbientLight( 0x404040 );
    }
  })
```

Zanimljivost u vezi korištenja ponašanja EmptyObject je da kada se koristi, naša scena zna prepoznati da entitet vezan uz to ponašanje je bitan za povezati sa Three.js scenom, što ga čini prikladnim za širok spektar primjena (od različitih vrsta svjetala, složenijih kombinacija geometrija i materijala, do volumetričnih objekata).

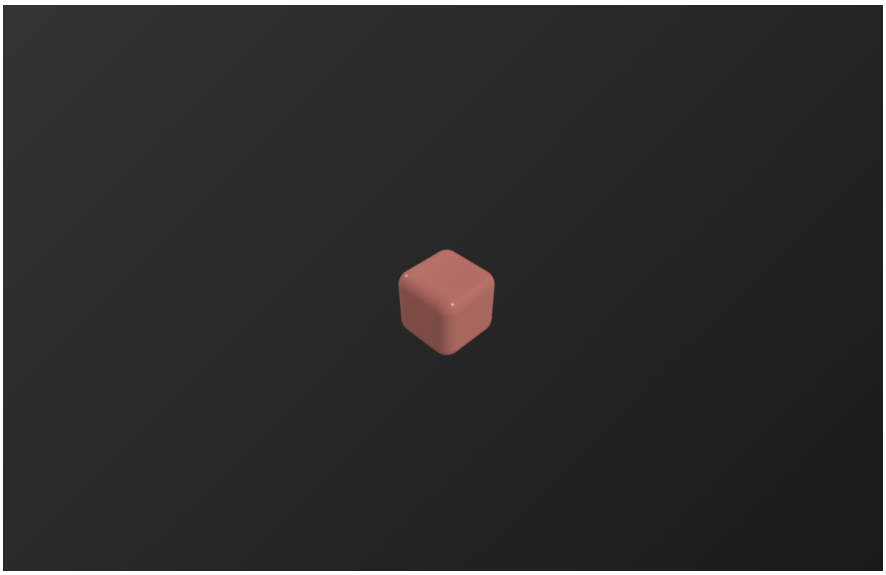
3.5.5. MeshObject

Naravno, u duhu pisanja što manje količine koda, dostupno je i ponašanje "MeshObject" koje je zadano s geometrijom kocke i upečatljivim materijalom kako bi entitet odma mogao krenuti u uporabu.

```
const Player = new Behavior().use(MeshObject);
```

3.5.6. CameraObject

Ponašanje "CameraObject" je još jedno ponašanje naslijeđeno od ponašanja "EmptyObject", i kao što ime govori, pruža nam mogućnost definiranja i kontroliranja kamere.



Slika 3.1. Izgled zadane kocke pi korištenju ponašanja MeshObject

Posebnost ponašanja “CameraObject” je u akcijama dostupnim za pozivanje.

Akcija “focus” dozvoljava kameri da motri neki drugi entitet, te je na programeru da definira što to točno znači u kontekstu željene primjene. Jedna od primjena akcije "focus" bi omogućila kameri da prati položaj fokusiranog entiteta s odmakom, dok bi druga omogućila kameri da zadrži položaj, ali samo gleda u smjeru fokusiranog entiteta.

Dalje, akcija "makeActive" jednostavno govori sceni da preuzme ovaj entitet kao aktivnu kameru i prikazuje svijet iz njegove perspektive.

```
const Camera = new Behavior()
  .use(CameraObject)
  .init((state) => {
    // Postavi kameru kao aktivnu
    deferAction(() => {
      state.this.actions.makeActive();
    });
  })
  .tick((state) => {
    const focusPosition = state.focus.position.clone();
    const offset = new Vector3(1, 1, 1);

    // Drži odmak od fokusiranog entiteta
    state.object.position.copy(focusPosition.add(offset));

    // Gledaj u smjeru fokusiranog entiteta
    state.object.lookAt(focusPosition);
  });

const cameraRef = Camera.create(scene);

const Player = new Behavior().use(SceneReference).init((state) => {
  // Fokusiraj kameru na igraća
  cameraRef.focus(state.this);
});
```

3.5.7. Transform

Ponašanje “Transform” jednostavno postavlja u stanje svojstva položaja (odvojenog od Three.js objekta), usmjerene brzine, usmjerene akceleracije, usmjerenog trenja, te quaternion rotacije i vektor skaliranja. Osim samog dodavanja svojstava, ponašanje “Transform” dodaje i logiku primjene akceleracije na brzinu, te brzine na položaj.

Slika 3.2. Perspektiva iz novokreirane kamere

```
const Player = new Behavior().use(Transform).tick((state) => {
  // state: {
  //    position: { x: number, y: number, z: number },
  //    velocity: { x: number, y: number, z: number },
  //    acceleration: { x: number, y: number, z: number },
  //    friction: { x: number, y: number, z: number },
  //    maxVelocity: number,
  //    scale: { x: number, y: number, z: number },
  //    rotation: { x: number, y: number, z: number, w: number }
  // }
});
```

3.5.8. ApplyTransformToObject

Ponašanje “Transform” samo po sebi ne utječe na objekt u 3D prostoru, stoga je potrebno primjeniti vrijednosti računate u ponašanju “Transform”, kao što su translacija, rotacija i skaliranje, koristeći ponašanje “ApplyTransformToObject”, koje ovisi o uporabi ponašanja “Transform” i ponašanja “EmptyObject” (sjetimo se sada korisnosti ponašanja “EmptyObject”)

```
const Player = new Behavior()
  .use(Transform)
  .use(MeshObject)
  .use(ApplyTransformToObject);
```

3.5.9. Keyboard

Ugrađeno ponašanje "Keyboard" omogućuje interakciju s entitetima putem tipkovnice, kao što su pritisak tipke, otpuštanje tipke i držanje tipke.

Samo ponašanje je ipak malo kompleksnije za upogonit, obzirom da se kreira instanca klase “Keyboard” te se na njoj poziva metoda “register” koja tek tada vraća ponašanje spremno za korištenje.

Ukoliko se koristi ponašanje “Keyboard”, ono zahtjeva da se pri instancijaciji klase proslijedi korištena instanca klase “Ebon”, te rječnik u kojem ključevi predstavljaju ime vezane tipke, te samu tipku, primjerice (”up” i slovo na tipkovnici “w”)

Ponašanje u stanje entiteta dodaje objekt pod ključem “keyboard” koji sadrži Booleove vrijednosti za svako ime vezane tipke.

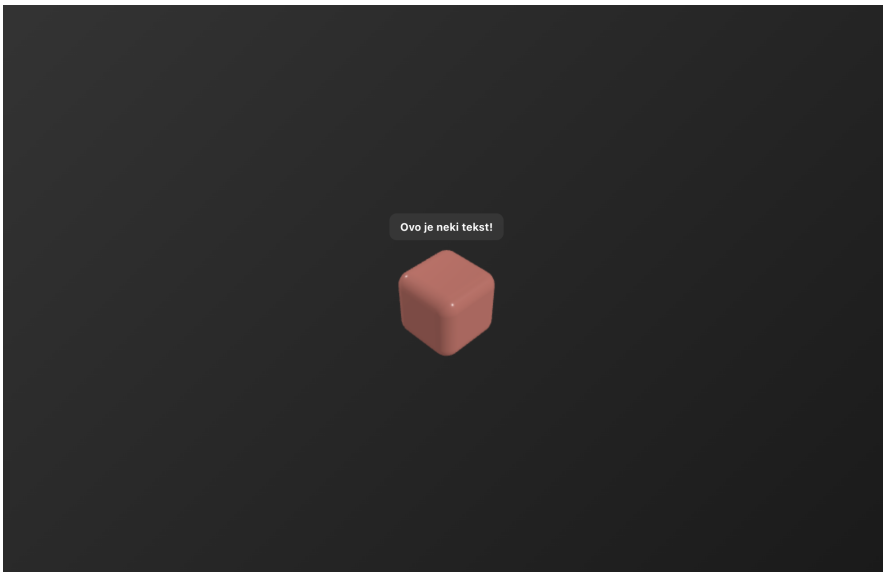
```
const Player = new Behavior()
  .use(Delta)
  .use(
    new Keyboard(ebon, {
      up: 'w',
      down: 's',
      left: 'a',
      right: 'd'
    }).register()
  )
  .tick((state) => {
    // state: { keyboard: {
    //    up: boolean,
    //    down: boolean,
    //    upleft boolean,
    //    right: boolean
    // } }
  });
```

3.5.10. InterfaceAnchored

Konačno, posljednje ugrađeno ponašanje je “AnchoredInterface”. Ono predstavlja za trenutno jedini način za prikazivanje klasičnog korisničkog sučelja u sklopu virtualne scene.

Ponašanje funkcionira na način da se u njega jednostavno proslijedi JSX komponenta, potom će se ista komponenta prikazivati na 2D položaju ekrana koji odgovara projekciji 3D točke iz virtualne scene na aktivnu kameru.

```jsx
const Tooltip: FC<{ text: string }> = ({ text }) => (
  <div
    style={{
      background: '#44444480',
      color: 'white',
      borderRadius: '8px',
      padding: '8px 12px',
      fontSize: '12px',
      fontWeight: 'bold',
      backdropFilter: 'blur(4px)'
    }}>
    {text}
  </div>
);

const Player = new Behavior()
  .use(InterfaceAnchored(<Tooltip text="Ovo je neki tekst!" />))
  .init((state) => {
    // state: {}
  });
```

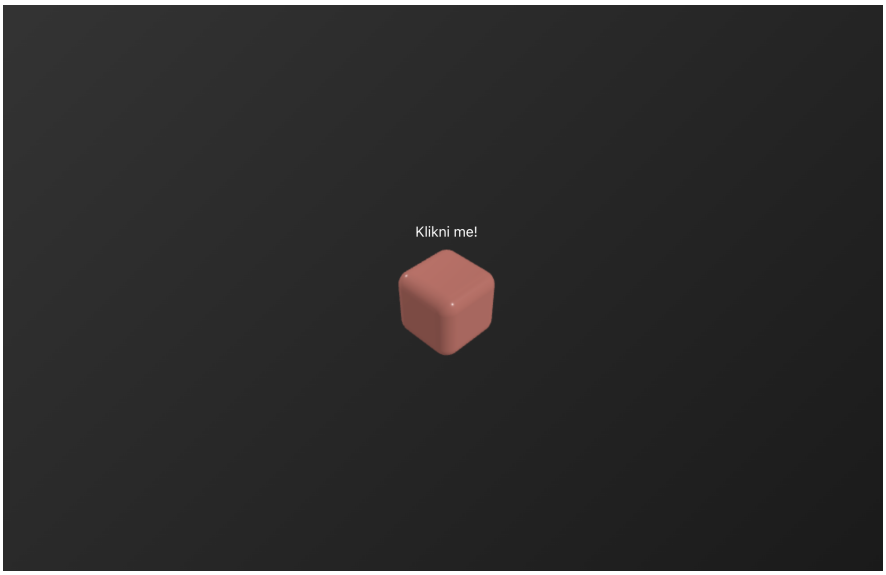


Slika 3.3. Primjer prikaza JSX komponente Tooltip

Pomoću ponašanja "InterfaceAnchored" možete stvoriti proizvoljno korisničko sučelje za interakciju s entitetima, kao što su gumbi, polja za unos i razne druge informacije. Moguće je postići interakciju mišem tako da se JSX komponenti sučelja proslijedi “on-

Click” callback funkcija koja može obavljati pozivanje akcija nad nekim entitetom.

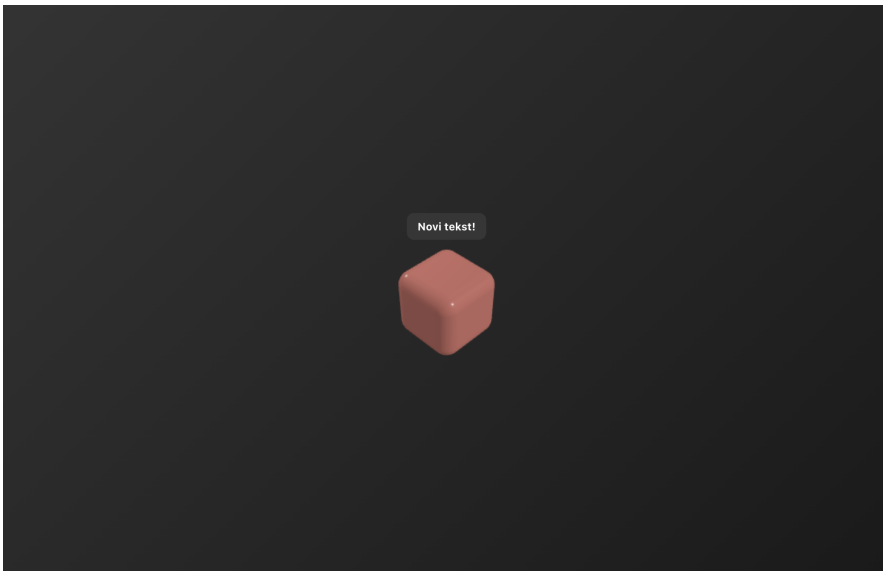
```
const Player = new Behavior()
  .use(
    InterfaceAnchored(
      <div
        style={{ color: 'white' }}
        onClick={() => {
          {
            /* Proizvoljna logika ide ovdje. */
          }
        }}>
        Klikni me!
      </div>
    )
  )
  .init((state) => {
    // state: { interfaceId: string }
  });
```



Slika 3.4. Primjer prikaza JSX komponente koja reagira na klik mišem

Ponašanje također u stanje entiteta ostavlja svojstvo “interfaceId” koje se može proslijediti pomoćnoj funkciji “useEbonInterface” pri pozivu kako bi se dinamički mijenjalo prikazano sučelje.

```
const Player = new Behavior()
  .use(EmptyObject)
  .use(InterfaceAnchored(<Tooltip text="Ovo je neki tekst!" />))
  .init((state) => {
    // state: { interfaceId: string }

    useEbonInterface
      .getState()
      .setElement(interfaceId, <Tooltip text="Novi tekst!" />);
  });
```



Slika 3.5. Primjer promjene prikazane JSX komponente

Samo ponašanje “AnchoredInterface” je implementirano koristeći vanjsku biblioteku Zustand, te pomoću nje se pohranjuje i ažurira položaj registriranog objekta. Vrijednost spremljenu u Zustand-ovom “store”-u React zna pročitat, te na temelju pročitane vrijednosti prikazuje učitane JSX elemente na pripadajućim položajima.

4. Okolina entiteta: Scena i Ebon

4.1. Scena

Osvrnimo se na trenutak kako funkcionira sama scena - kontekst u kojem će entiteti prebivati. Scena je implementirana kao klasa koja sadrži listu aktivnih entiteta.

Lista aktivnih entiteta je dalje opisana klasom "EntityList" koja uključuje iterator za filtriranje po postojanju ponašanja. Također se pruža mogućnost različitih načina iteriranja, kao što je iteriranje po prostornim sektorima - što je korisno za optimizaciju brzine izvođenja programa.

Pozivom metode “tickScene” pokreće se ciklus ažuriranja stanja koji na svakom od aktivnih entiteta poziva metodu “executeTick”. Ovu metodu pokreće klasa “Ebon”, koja će biti spomenuta kasnije u ovom poglavlju 4.2.

4.1.1. Three.js scena

Klasom “Scene” iz Three.js biblioteke se postiže crtanje 3D prikaza, stoga se pri inicijalizaciji scene iz “Ebon” biblioteke istovremeno inicijalizira i Three.js scena.

Međutim, Three.js scena zna tumačiti samo objekte opisane geometrijom i materijalima (tzv. “mesh” objektima), te kamere i objekte osvjetljenja.

S obzirom na to da biblioteka “Ebon” može sadržavati i entitete koji nisu nužno objekti koji se trebaju prikazivati u 3D okruženju, scena biblioteke “Ebon” sama dodaje entitete koji sadrže značajke bitne za prikazivanje u 3D okruženje u Three.js scenu.

4.2. Ebon

Glavna je klasa koja dijeli ime s bibliotekom, “Ebon”, odgovorna za postavljanje “canvas” HTML elementa na koji se crta stanje scene iz perspektive aktivne kamere. Pri kreaciji projekta, prvi je korak instancirati klasu “Ebon”, te instanci pridružiti instancu “Scene”. “Ebon” instanca se dalje, u okruženju Reacta, proslijeđuje ugrađenoj komponenti “EbonContainer” koja je smještena na proizvoljan položaj web stranice.

Instanciranje klase “Ebon” također instancira i druge vanjske sisteme, kao što je sistem motrenja tipkovnice.

4.2.1. Ciklusi ažuriranja

Klasa “Ebon” ima referencu na aktivnu scenu, te na njoj poziva metodu “tickScene” više puta u sekundi pomoću window API-eve “requestAnimationFrame” metode.

Početkom izvršenja ciklusa ažuriranja, zapisuje se trenutno vrijeme, te oduzimanjem trenutnog vremena od vremena prethodnog ciklusa se dobiva diferencijal vremena (tzv. “delta”). Taj diferencijal vremena se proslijeđuje u metodu “tickScene” koja dalje propagira “deltu” na poziv “executeTick” metode na pojedinim entitetima, kako bi se što preciznije izvodile vremenski osjetljive operacije (kao što je računanje akceleracije, utjecaja trenja, te brzine).

4.2.2. Reagiranje na promjenu veličine prozora

Komponenta “Ebon” isto tako vodi računa o dostupnom prostoru na zaslonu, te pri povećanju i smanjenju prozora govori sceni da na aktivnoj kameri ponovno izračuna omjer (”Aspect ratio”), te rezoluciju.

5. Dodatci biblioteci

5.1. Pakiranje biblioteke

Koristeći biblioteku [4, tsup], izvorni kod biblioteke ‘Ebon‘ se pretvara u Javascript datoteke i datoteke Typescript tipova. Ovaj postupak omogućava korištenje biblioteke i u Javascript okruženjima i u okruženjima Typescripta, gdje Typescript može čitati izvedene tipove.

Biblioteka se pakira u navedene datoteke u direktorij ‘/dist‘.

5.2. Objavljivanje paketa u NPM sustavu

Objavljivanje paketa u [5, NPM] sustavu omogućava drugim korisnicima da preuzmu i koriste biblioteku ‘Ebon‘ u svojim projektima. To je korisno jer omogućava jednostavnu distribuciju i dijeljenje koda s drugima. Biblioteka prati semantično verzioniranje, te je proces automatiziran postupkom opisanim u ovom [6, članku]. Sam proces objavljivanja biblioteke je automatiziran te izveden postupkom inspiriranim ovim [6, člankom].

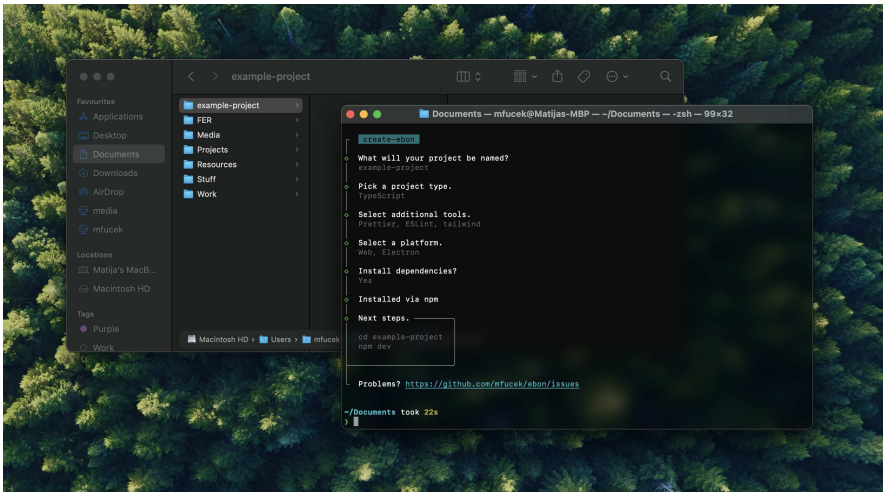
Biblioteka “Ebon” je tim putem objavljena u sustavu NPM, te ju je moguće preuzeti pokretanjem iduće naredbe:

```
npm i ebon@latest
```

5.3. CLI tool

Pored mogućnosti za ručno preuzimanje biblioteke, stvoren je i alat spreman za pokretanje u naredbenom retku koji automatski inicijalizira jednostavan projekt, zatim instalira sve popratne biblioteke, te stvara sve neophodne datoteke.

```
npx create-ebon-app@latest
```



Slika 5.1. Primjer pokretanja CLI alata

Napomenuo bih da alat trenutno inicijalizira identičan projekt neovisno o odabiru opcija. Cijela funkcionalnost će biti implementirana u budućim verzijama biblioteke.

6. Primjena biblioteke

Biblioteka se može koristiti za stvaranje raznolikih virtualnih svjetova, od kreiranja igara do raznih simulacija. U ovom poglavlju ćemo primijeniti koncepte biblioteke i izgraditi jednostavnu virtualnu scenu s avatarom koji se može kontrolirati tipkovnicom.

Kao primjer primjene biblioteke, također ćemo stvoriti entitet koji prati avatara, kao i entitet koji periodično stvara nove entitete koji prate avatara. Na ovaj način ćemo pokazati koliko je biblioteka sposobna u kreiranju jednostavnih scena u maloj količini koda.

6.1. Inicijalizacija projekta

Za početak pokaznog projekta, bitno je osigurati da su instalirane popratna programska podrška za [7, Node] okruženje. U naredbenom retku ćemo upisati naredbu:

```
npx create-ebon-app
```

Dalje, prateći uputstva u naredbenom retku ćemo odabrati ime projekta, te željene postavke za projekt. Alat nam je inicijalizirao sada projekt te, priredio sve bitne datoteke za pokretanje istog.

6.1.1. Pokretanje programa

Za samo pokretanje programa, u naredbenom retku je potrebno pokrenuti sljedeću naredbu:

```
npm run dev
```

Primjetit ćemo da u kreiranom projektu scena isključivo sadržava običnu kocku koja se rotira.

6.1.2. Postavljanje scene

U novokreiranom projektu, navigirat ćemo do datoteke '/src/game.ts' obzirom da se u njoj nalazi početna logika programa, te inicijalna hijerarhija entiteta.

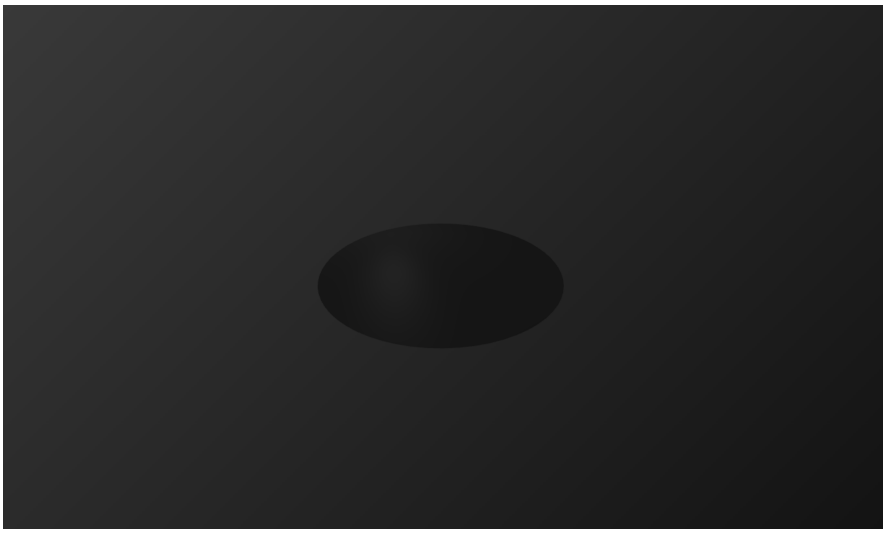
U svrhu kreacije željenog pokaznog primjera, obrisat ćemo ponašanje "Cube" te ćemo krenuti u implementaciju vlastitih ponašanja.

Za početak, kreirat ćemo ponašanje podloge, te ćemo ju dodati na scenu.

```
// /src/entities/floor.ts

const Floor = Entity.init(({ object }) => {
  const plane = new THREE.Mesh(
    new THREE.CircleGeometry(4, 64),
    new THREE.MeshPhongMaterial({
      color: '#222222'
    })
  );
  plane.receiveShadow = true;
  plane.castShadow = false;
  return { object: plane };
});

// /src/game.ts

Floor.create(scene);
```

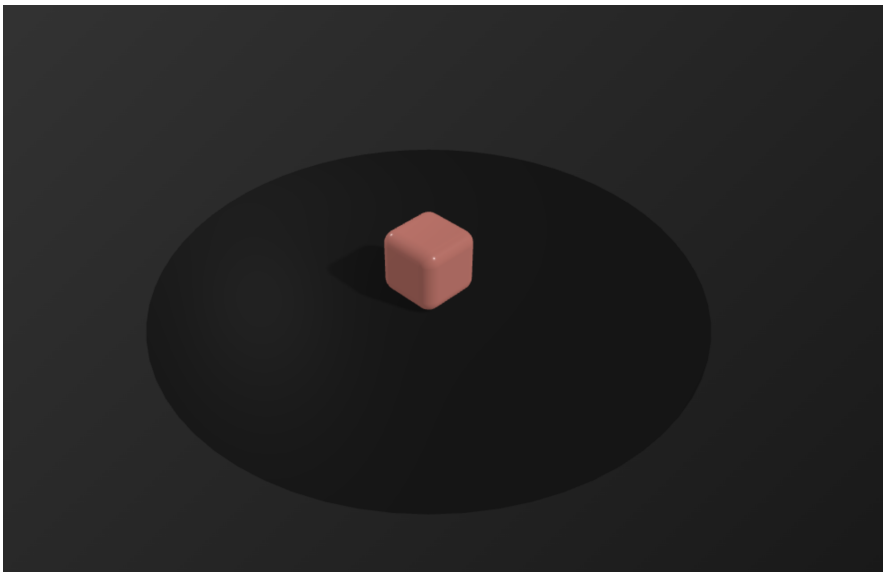
6.2. Avatar

Iduće, stvorit ćemo ponašanje avatara, te ćemo ga dodati na scenu.



Slika 6.1. Izgled kreirane podloge

```
const Player = new Behavior()
  .use(Delta)
  .use(SceneReference)
  .use(MeshObject)
  .use(Transform);

Player.create(scene);
```

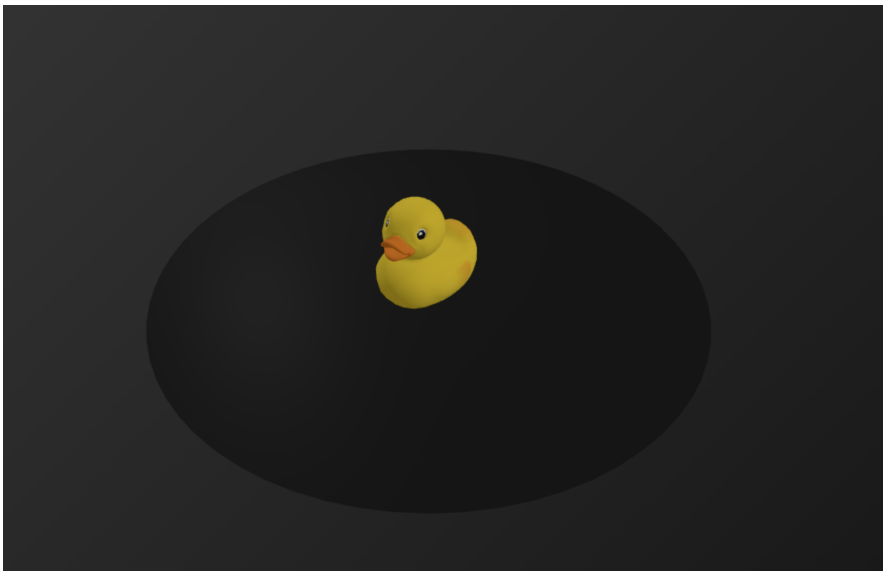


Slika 6.2. Avatar kao zadana kocka

6.2.1. Izgled avatara

S obzirom na to da je korišteno ponašanje “MeshObject” avatar izgleda kao crvena kocka. Promjenit ćemo izgled avatara učitavanjem vanjskog modela na lokaciji ‘/models/duck/Duck.gltf‘ uporabom “GLTFLoader” i “DRACOLoader” alata iz biblioteke Three.js

```
const Player = new Behavior()
  // ...
  .init((state) => {
    const loader = new GLTFLoader();
    const dracoLoader = new DRACOLoader();

    dracoLoader.setDecoderPath('/examples/jsm/libs/draco/');
    loader.setDRACOLoader(dracoLoader);

    loader.load('models/duck/Duck.gltf', (gltf) => {
      const obj = gltf.scene.children[0].children[0] as THREE.Mesh;

      // korekcija orijentacije i skale učitanog modela
      obj.geometry.applyMatrix4(
        new THREE.Matrix4().makeScale(0.01, 0.01, 0.01)
      );
      obj.geometry.applyMatrix4(new THREE.Matrix4().makeRotationX(Math.PI /
      ↪  2));

      // primjena učitanog modela na objekt
      state.object.copy(obj as THREE.Mesh);
    });

    // Postavljanje parametara koje ponašanje "Transform" koristi
    state.friction.set(25, 25, 0);
    return {
      maxVelocity: 10
    };
  });
```



Slika 6.3. Avatar kao učitani model patke

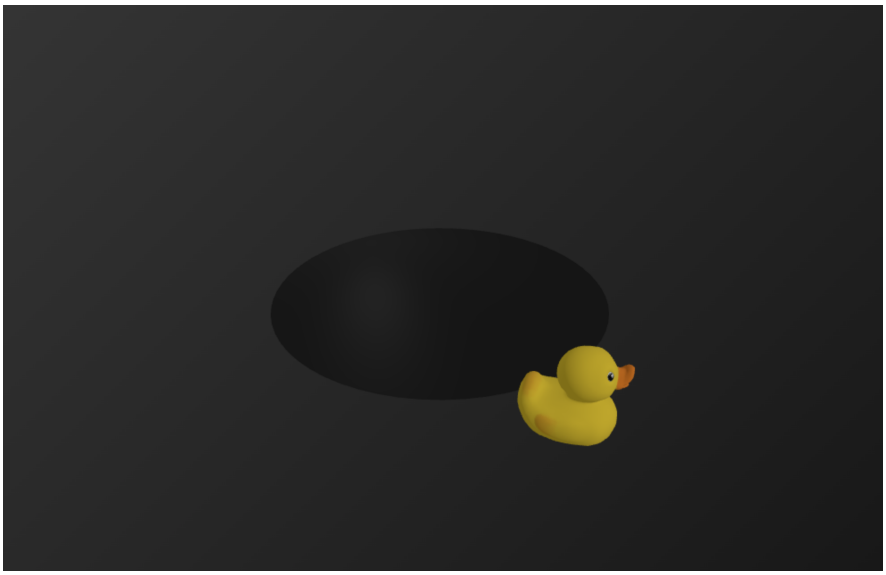
6.2.2. Kretanje avatara

Na ponašanje avatara ćemo dodati ponašanje “Keyboard”, te povezati potrebne tipke za kretanje

```
const Player = new Behavior()
  // ...
  .use(
    new Keyboard(ebon, {
      up: 'w',
      down: 's',
      left: 'a',
      right: 'd',
      jump: ' '
    }).register()
  );
```

Dalje, implementirat ćemo samu logiku za kretanje i rotaciju entiteta pri kretanju.

```
const Player = new Behavior()
  // ...
  .tick(({ keyboard, acceleration, position, velocity, object }) => {
    const isGrounded = position.z <= 0;

    // resetiranje akceleracije u slučaju da nije pritisnuta tipka
    acceleration.set(0, 0, 0);

    if (isGrounded) {
      // osiguravanje da avatar ne propadne kroz podlogu
      velocity.z = 0;
      position.z = 0;
    } else {
      // ako avatar nije na podlozi, neka djeluje gravitacija
      acceleration.z = -50;
    }

    // primjena pritiska tipke za skok na iznos usmjerene brzine
    if (keyboard.jump && isGrounded) velocity.z = 15;

    // primjena pritiska tipki za kretanje na iznos usmjerenog ubrzanja
    if (keyboard.left) acceleration.x = 100;
    if (keyboard.right) acceleration.x = -100;
    if (keyboard.up) acceleration.y = -100;
    if (keyboard.down) acceleration.y = 100;

    // jedinični vektor brzine bez Z komponente
    const movingDirection = velocity.clone();
    movingDirection.z = 0;
    movingDirection.normalize();

    // rotiraj entitet u smjeru kretanja
    const angle = Math.atan2(velocity.y, velocity.x);
    object.rotation.z = angle;
  });

// /src/game.ts
export const playerRef = Player.create(scene);
```

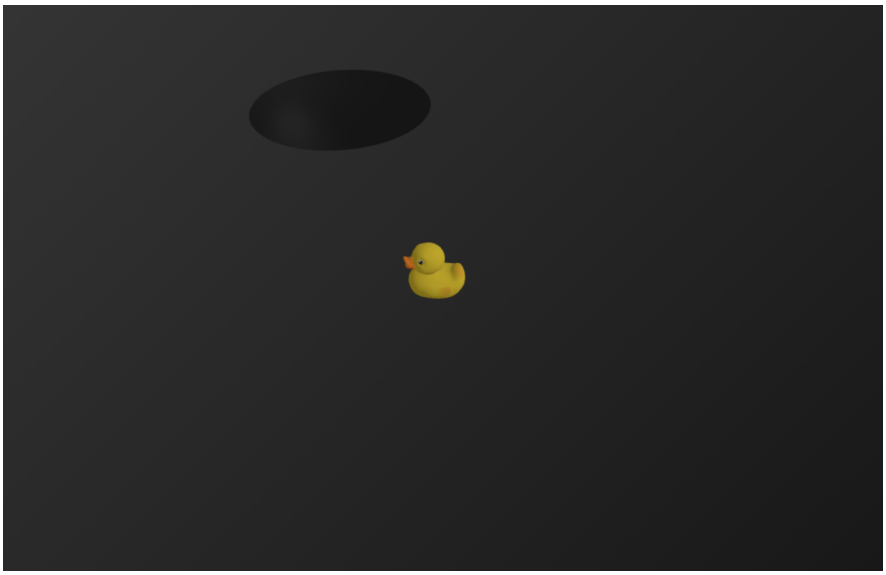


Slika 6.4. Demonstracija kretanja avatara

6.3. Implementacija kamere

S obzirom na to da se avatar može slobodno kretati po sceni, nedostaje nam mogućnost da kamera prati njegov položaj. Kako bismo prebrisali zadanu kameru, implementirat ćemo našu, koja prilagođava svoj položaj u ovisnosti o položaju avatara, te sa dodanim odmakom.

```
// /src/entities/mainCamera.ts

import { playerRef } from '@/src/game';

// odmak kamere od fokusiranog entiteta
const cameraOffset = new THREE.Vector3(0, 5, 5).multiplyScalar(2);

export const MainCamera = new Behavior()
  .use(Delta)
  .use(SceneReference)
  .use(CameraObject)
  .use(Transform)
  .use(ApplyTransformToObject)
  .init(({ position, object, ...state }) => {
    // postavljanje MainCamera entiteta kao aktivnom kamerom
    deferAction(() => {
      state.this.actions.makeActive();
    });

    // zadan položaj bez fokusiranog entiteta
    position.copy(cameraOffset);
    object.position.copy(cameraOffset);
    object.lookAt(0, 0, 0);
    return { focus: playerRef };
  })
  .tick(({ position, object, age, focus }) => {
    if (!focus) {
      return;
    }

    // prilagodba položaja u ovisnosti o položaju fokusiranog entiteta
    position.lerpVectors(
      position.clone(),
      focus.state.position.clone().add(cameraOffset),
      0.2
    );
  });

// /src/game.ts

MainCamera.spawn(scene);
```



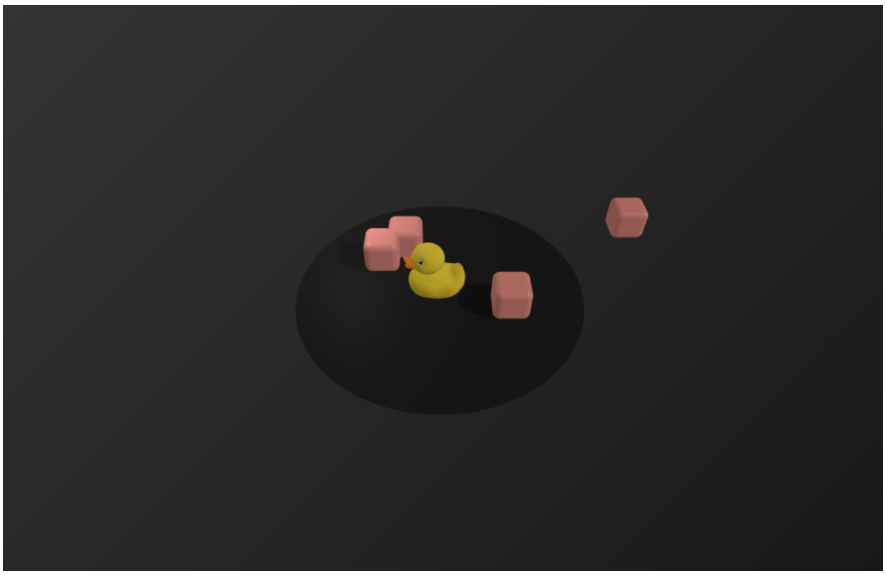
Slika 6.5. Demonstracija kamere koja prati položaj avatara

6.4. Ostali entiteti

6.4.1. Praćenje avatara

Sličnim principom kojim kamera prati ponožaj igraća ćemo primjeniti na druge entitete, koji će pratititi položaj avatara, te će mu se približavati. Osim samog približavanja, u ovisnosti o blizini entiteta od avatara ćemo zakretati vektor kretanja, pa time postižemo izgled orbitiranja entiteta oko avatara.

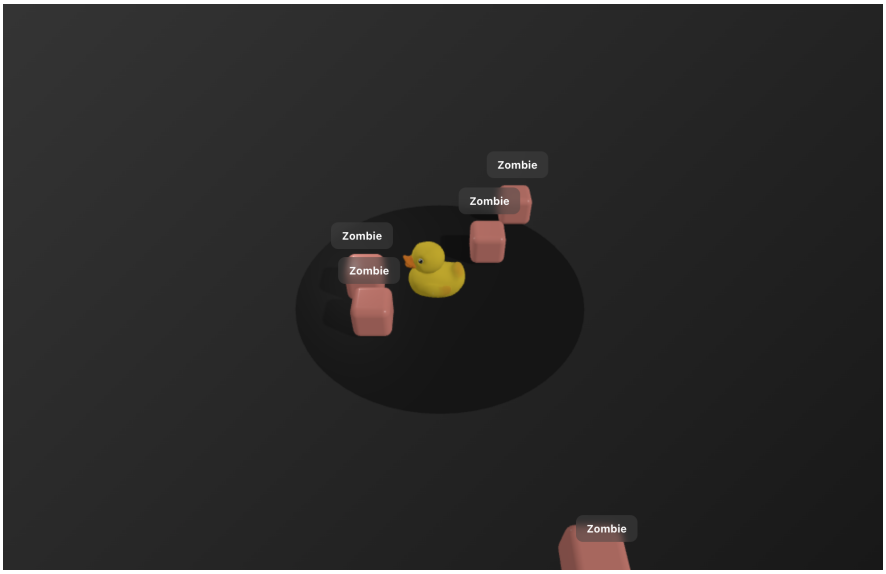
```
// /src/entities/enemy.ts

import { playerRef } from '@/src/game';
const spawnRange = 10;

export const Enemy = new Behavior()
  .use(Delta)
  .use(Transform)
  .use(MeshObject)
  .use(ApplyTransformToObject)
  .init(({ position }) => {
    const playerPosition = playerRef.state.position.clone();
    playerPosition.z = 0;

    // nasumični 2d vektor radijusa spawnRange oko avatara
    const angle = Math.random() * Math.PI * 2;
    const randomOffsetVector = new THREE.Vector3().set(
      spawnRange * Math.cos(angle),
      spawnRange * Math.sin(angle),
      0
    );
    position.copy(playerPosition.add(randomOffsetVector));

    // definiramo brzinu kretanja koja blago varira od entiteta do entiteta
    return {
      speed: Math.random() * 2 + 4
    };
  })
  .tick(({ position, velocity, speed }) => {
    // računanje smjera kretanja prema avataru
    const playerPosition = playerRef.state.position.clone();
    let direction = new THREE.Vector3(
      playerPosition.x - position.x,
      playerPosition.y - position.y,
      0
    );

    // zakretanje smjera kretanja u ovisnosti o blizini
    direction = rotateVectorZ(direction, Math.PI / direction.length());
    velocity.copy(direction.clone().normalize().multiplyScalar(speed));
  });
```



Slika 6.6. Nasumično postavljanje entiteta koji prate avatara

Konačno, na entitete koji prate avatara postavljamo natpis na kojem piše “Zombie”.

```
// /src/entities/enemy.ts

export const Enemy = new Behavior()
  // ...
  .use(InterfaceAnchored(<Tooltip text="Zombie" />));
```

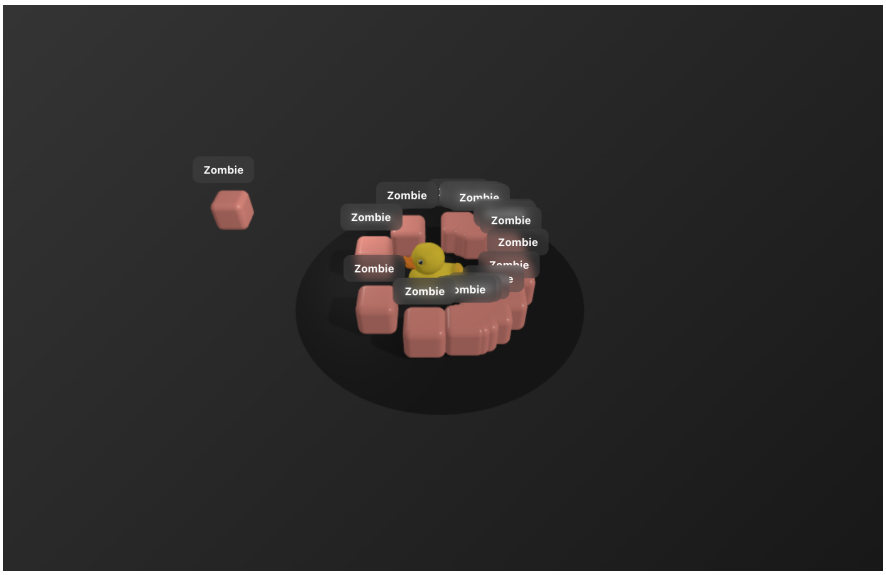


Slika 6.7. Prikaza JSX komponente na entitetima koji prate avatara

6.4.2. Implementacija “Spawnera”

Sistem periodičnog stvaranja entiteta koji prate igraća implementirat ćemo kroz ponašanje “Spawner”.

```
// /src/entities/spawner.ts

import { Enemy } from '@/src/entities/enemy';

export const Spawner = new Behavior() //
  .use(Delta)
  .use(SceneReference)
  .init(() => {
    return {
      spawnInterval: 5000,
      lastSpawn: 0
    };
  })
  .tick(({ scene, spawnInterval, lastSpawn, age }) => {
    if (age > lastSpawn + spawnInterval) {
      Enemy.create(scene);
      return { lastSpawn: age };
    }
  });

// /src/game.ts

Spawner.create(scene);
```



Slika 6.8. Demonstracija periodičnog stvaranja entiteta koji prate avatara

7. Rezultati i rasprava

Biblioteka se pokazala kao izuzetno učinkovit način za stvaranje jednostavnih interaktivnih iskustava zbog svoje sposobnosti da olakša proces programiranja i pruži korisnicima širok spektar mogućnosti. Osim toga, korištenje biblioteke omogućuje programerima da brzo prototipiraju i testiraju različite ideje, čime se poboljšava kreativni proces i omogućuje inovativnost.

Međutim, biblioteka također ima neke nedostatke. Na primjer, nedostaje podrška za određene složenije funkcionalnosti i efekte. Također, dokumentacija biblioteke skroz razrađena te nedostaje primjera i uputa za rješavanje uobičajenih problema.

U idućem dijelu poglavlja, pogledat ćemo i neke veće arhitekturalne probleme, te ograničenja biblioteke.

Pored navedenih mogućnosti i primjena biblioteke, važno je napomenuti da se susrećemo i s nekim nedostacima i ograničenjima.

7.1. Nedostaci

7.1.1. Dinamično dodavanje ili micanje ponašanja

Jedan od trenutno uočenih nedostataka postojeće implementacije je nemogućnosti dodavanja i uklanjanja ponašanja tijekom izvođenja programa. U trenutnoj implementaciji, ovaj se nedostatak može zaobići korištenjem ponašanja s dodanom zastavicom aktivnosti, te upravljanjem vrijednosti te zastavice kako bismo uključili ili isključili određenu logiku.

7.1.2. Relativno pozicioniranje

Osim toga, implementacija ponašanja “Transform” funkcionira samo na principu apsolutne pozicije, i to ponašanje samo po sebi nije dovoljno za postizanje ugnježđenog pozicioniranja entiteta, primjerice izazovno je postići da igračev avatar nosi virtualni predmet.

Ovaj je izazov trenutno moguće riješiti dodavanjem ponašanja koje transformira poziciju objekta iz relativnih koordinata u odnosu na “parent” objekt u apsolutne koordinate parent objekta, ali ovakvo ponašanje nije već uključeno u biblioteku.

7.2. Ograničenja

7.2.1. Typescript ograničenje dubine

Postoje neka ograničenja vezana za dubinu ugniježđenosti tipova u TypeScriptu, što može predstavljati izazov prilikom proširivanja generičkih tipova ponašanja.

Način na koji se ponašanja proširuju prilikom pozivanja metoda (init, tick, use, itd.) podrazumijeva proširenje generičke klase "Behavior". Kada se generičko stanje proširuje drugim stanjem, vrijednosti iza ključeva prvog stanja zamjenjuju se vrijednostima iza ključeva drugog stanja.

Problem leži u prirodi izvođenja ovog prebrisavanja vrijednosti ključeva kroz funkcionalno ugniježđivanje tipova.

```
type Primjer = Omit<
    Omit<
      Omit<
        StateA,
        keyof StateB
      >
      & StateB,
      keyof StateC
    >
    & StateC,
    keyof StateD
  >
  & StateD;
```

Problem nastaje kada ugnježđenost pređe dubinu od 50 poziva u kojoj Typescript dalje ne razlučuje tipove te baca pogrešku ‘Type instantiation is excessively deep and possibly infinite. TS(2589)‘. Ovo je arhitekturalni problem koji sprječava programera da kreira kompleksnija ponašanja.

7.2.2. Skalabilnost

Konačno, još jedan važan aspekt koji treba uzeti u obzir je upravljanje memorijom i tzv. ‘garbage collection‘, posebno ako se radi o velikom istovremenom broju entiteta u sceni.

Entitet pri inicijalizaciji je u biti instanca samo jednog ponašanja. Ponašanje koje je načinilo entitet može biti naravno skup više drugih ponašanja, ali po prirodi implementacije kranje ponašanje koje se koristi u entitetu ima jedinstvenu i to lokalno definiranu funkciju za ažuriranje stanja. Drugim riječima, funkcije za ažuriranje stanja nisu definirani na globalnoj razini, već se pozivaju dubinski u sklopu drugih funkcija.

Efektivno, za svaku kombinaciju ponašanja koja načine novo ponašanje, postoji nova funkcija za ažuriranje stanja, što čini paralelizaciju tih procesa vrlo teškim problemom za riješit. Stoga, nije iznenađujuće da će programi biti niskih performansi pri velikom broju entiteta na sceni.

Unatoč tim nedostacima, ‘Ebon‘ biblioteka i dalje pruža korisnicima moćan temelj

za izgradnju manje zahtjevnih virtualnih svjetova i interaktivnih iskustava. Sa daljnjim razvojem i poboljšanjem, ova biblioteka ima potencijal postati vrlo korisna za programere koji se bave 3D grafikom i interaktivnim simulacijama.

7.3. Sličnost modela s ECS arhitekturom

Biblioteka "Ebon" je građena od početka na temelju ideje poboljšavanja sintakse. Kroz iteracije razvoja biblioteke, rješavanjem jednog po jednog problema, lagano se istitralo prema obrascima već implementiranim u drugim arhitekturama upravljanja stanjima entiteta.

Jedan od takvih arhitektura je upravo [8, ECS], ili tzv. 'Entity Component System' arhitektura, koja prati vrlo slične principe. Ono što je u ovom radu opisano kao "ponašanje" podrazumijeva neku specifikaciju tipa podatka stanja - ono što je u ECS arhitekturi komponenta - te funkcionalnost ažuriranja tog stanja - ono što je u ECS arhitekturi sistem.

Entiteti u ECS arhitekturi su predstavljeni samo kao 32-bitni brojevi, i sami po sebi nemaju nikakvo stanje, već to stanje nastaje pridruživanjem određene komponente (opet, slično kao pridruživanjem ponašanja).

Razlika između ECS arhitekture i "ponašajne" arhitekture baš leži u opsegu u kojem se nalaze sistemi za ažuriranje. U ECS-u su globalni, i pozivaju se na specifične entitete s određenim komponentama, što čini paralelizaciju laganom za izvest. Dok, kod "ponašajne" arhitekture ti sistemi su ugnježđeni i teško da se mogu paralelizirati.

8. Zaključak

U sklopu ovog Završnog rada predložena je i implementirana biblioteka za kreiranje interaktivnih virtualnih okruženja na web-aplikacijama koja se temelji na načelima ponašajnih obrazaca.

Implementacija ove biblioteke napravljena je u jeziku Typescript, uz korištenje obrazaca funkcionalnog programiranja te na način da je biblioteku moguće lako povezati s ostalim bibliotekama za razvoj web-aplikacija.

Unatoč određenim opisanim izazovima koje ovakav oblikovni obrazac uzrokuje, implementirana biblioteka zanimljiv je i važan eksperiment u smjeru razvoja ekspresivnih funkcionalnih interaktivnih virtualnih okruženja.

Zaključak ovog rada na biblioteci za upravljanje stanjem entiteta je da ponekad nije najbolja ideja izmišljati toplu vodu. Tijekom razvoja biblioteke ‘Ebon‘ naučio sam da postoje već postojeće arhitekture poput ECS (Entity Component System) koje već nude rješenja za slične probleme.

Unatoč tome, razvoj ove biblioteke bio je koristan jer sam stekao dublje razumijevanje principa upravljanja stanjem entiteta i sintakse Typescripta. Također sam otkrio neke nedostatke i ograničenja u implementaciji, kao što su nemogućnost dinamičkog dodavanja ili uklanjanja ponašanja te izazovi u paralelizaciji procesa ažuriranja stanja.

U budućnosti, sigurno ću nastaviti istraživanje postojećih arhitektura kao što je ECS i razmotrit ću implementaciju istih pri razvoju sličnih projekata. Također bih volio nastaviti proširivati svoje znanje o upravljanju stanjem entiteta i 3D grafici te primijeniti ta znanja u budućim projektima.

Kroz ovaj rad, shvatio sam važnost korištenja postojećih arhitektura i alata kako bi

se olakšao razvoj i postigla bolja skalabilnost. Uz to, važno je uzeti u obzir performanse i upravljanje memorijom prilikom izrade interaktivnih iskustava.

U konačnici, biblioteka ‘Ebon‘ je dovršena, i može se stabilno koristiti unutar granica ograničenja, te pruža korisnicima temelj za izgradnju manje zahtjevnih virtualnih svjetova i interaktivnih iskustava. Sa daljnjim razvojem i poboljšanjem, ova biblioteka ima potencijal postati vrlo korisna za programere koji se bave 3D grafikom i interaktivnim simulacijama.

Literatura

[1] Microsoft, “The starting point for learning TypeScript”, [mrežno; stranica posjećena: 2023-11-03].

[2] M. Corporation, “Webgl: 2d and 3d graphics for the web - Web APIs”, [mrežno; stranica posjećena: 2024-01-18].

[3] mrdoob, “three.js docs”, [mrežno; stranica posjećena: 2024-01-26].

[4] egoist, “Tsup documentation”, https://tsup.egoist.dev/, [mrežno; stranica posjećena: 2024-01-26].

[5] I. NPM, “Npm Homepage”, [mrežno; stranica posjećena: 2024-12-20].

[6] A. Stagi, “Publish to NPM using GitHub Actions”, https://dev.to/astagi/publish-to-npm-using-github-actions-23fn, aug 25 2021., [mrežno; stranica posjećena: 2024-01-03].

[7] O. Foundation, “Node.js”, [mrežno; stranica posjećena: 2023-10-24].

[8] R. Foundation, “bevy_ecs Documentation”, [mrežno; stranica posjećena: 2024-01-26].

Sažetak

Radni okvir za kreaciju interaktivnih virtualnih iskustava na webu

Matija Fućek

Ovaj rad pruža pregled biblioteke ‘Ebon‘ za upravljanje stanjem entiteta u 3D grafici. Biblioteka pojednostavljuje proces stvaranja i upravljanja entitetima s različitim ponašanjima. Naglašava fleksibilnost i proširivost biblioteke, omogućavajući programerima definiranje prilagođenih ponašanja. Rad objašnjava osnovne koncepte poput ponašanja, stanja i akcija te prikazuje primjere koda. Također, prikazuje primjer projekta za stvaranje 3D interaktivnog igračkog okruženja. Biblioteka ‘Ebon‘ omogućuje programerima stvaranje uzbudljivih virtualnih svjetova.

Ključne riječi: biblioteka; računalna grafika; 3D; sučelje; ponašanje; entitet; stanje; akcija; igra; virtualna stvarnost; interaktivnost; Typescript; Three.js; WebGL

Abstract

Framework for creating interactive virtual experiences on the web

Matija Fućek

This document provides an overview of the ‘Ebon‘ library, which is designed for managing entities in 3D interactive experiences. The library simplifies the process of creating and manipulating entities with different behaviors, offering flexibility and extensibility. It explains the core concepts of behaviors, states, and actions, along with code examples. The document also features a detailed example project showcasing the library’s capabilities, including avatar control, camera implementation, enemy entities, and entity spawning. With its intuitive syntax and powerful features, the ‘Ebon‘ library empowers developers to create immersive virtual worlds. The example project serves as a practical demonstration of the library’s capabilities.

Keywords: library; computer graphics; 3D; interface; behavior; entity; state; action; game; virtual reality; interactivity; Typescript; Three.js; WebGL