Primjena programskog jezika C++ u razvoju web aplikacija

Midžić, Noa

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Noa Midžić

PRIMJENA PROGRAMSKOG JEZIKA C++ U RAZVOJU WEB APLIKACIJA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Noa Midžić

Matični broj: 0108082571

Studij: Informacijski i poslovni sustavi

PRIMJENA PROGRAMSKOG JEZIKA C++ U RAZVOJU WEB APLIKACIJA

ZAVRŠNI RAD

Mentor:

doc. dr. sc. Matija Novak

Varaždin, rujan 2023.

Noa Midžić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu će se prikazati tehnologije i jezici za razvoj za poslužiteljsku i klijentsku stranu web aplikacije. Bit će opisane karakteristike C++ jezika te će se istražiti moguće primjene C++ u kontekstu web aplikacija. Prikazat će se uporaba C++ na strani poslužitelja na primjeru web aplikacije koja koristi shell skriptu, te uporaba C++ na strani klijenta preko WebAssemblyja. Obradit će se osnove WebAssemblyja te će biti izrađena web aplikacija koja koristi WebAssembly za implementaciju igre u C++ jeziku sa SDL2 (eng. *Simple DirectMedia Layer 2*) bibliotekom, te ju integrira s drugim web tehnologijama kao što su NodeJS, WebGL i ThreeJS.

Ključne riječi: web, C++, WebAssembly, shell skripta, OpenCV, SDL2, ThreeJS

Sadržaj

1. Uvod

S razvojem Weba je JavaScript ostao većinski programski jezik za web preglednike kojemu su se mogućnosti postepeno proširivale, te mu se prema tome i uloga od jednostavnijih zadaća na klijentskoj strani web aplikacija mijenjala preuzimanjem sve zahtjevnijih zadaća koje su se tradicionalno mogle obavljati samo na poslužiteljskoj strani. Univerzalnost je s jedne strane omogućila brži i jednostavniji razvoj za Web, a proširivanje mogućnosti jezika omogućilo razvoj sve složenijih web aplikacija.

No kako se sve više aplikacija seli na web, što uključuje ne samo stvaranje novih aplikacija već i migraciju postojećih desktop i drugih aplikacija, tako se proširuju domene koje se koriste u kontekstu Weba. U nekima od njih se pokazuju ograničenja JavaScripta kao jedinog univerzalnog jezika koji se može koristiti za Web, primjerice u domenama koje zahtijevaju procesorski zahtjevne zadaće za koje JavaScript nije bio originalno dizajniran, kao što su grafika, obrada slike i videa, igre, zahtjevni izračuni i slično. S jedne strane je ovo djelomično riješeno proširivanjem mogućnosti za JavaScript u preglednicima, kao što je razvoj WebGL-a (eng. *Web Graphics Library*) za omogućavanje naprednije obrade grafike u web aplikacijama. S druge strane se nameće pitanje ponovne iskoristivosti već postojećih tehnologija koje su već optimizirane za te domene i imaju dobro razvijenu dokumentaciju, kao i pitanje isplativosti ponovnog pisanja tih alata za JavaScript u smislu vremenskog i financijskog troška, i potencijalnog gubitka na izvornim performansama radi promjene programskog jezika. S obzirom na širenje kompleksnosti web aplikacija i domena koje se koriste za njih, uvodi se motivacija da se isto tako prošire i opcije za odabir programskih jezika i tehnologija koje se mogu koristiti unutar Weba onkraj samog JavaScripta.

U tom kontekstu će se u daljnjem sadržaju proučiti mogućnosti primjene programskog jezika C++ unutar web aplikacija, domene koje su od interesa za ovu primjenu i načini njegove integracije u web aplikacije na klijentskoj i na poslužiteljskoj strani. Bit će obrađene različite tehnologije i alati koji omogućuju takvu integraciju, te predstavljeni određeni slučajevi upotrebe za web aplikacije koje integriraju C++. Posebno će se obraditi WebAssembly, tehnologija koja uvodi mogućnosti integracije raznih drugih jezika i alata s JavaScriptom unutar web aplikacija i za koju se očekuje da će imati sve veću primjenu u budućnosti, kako se infrastruktura Weba nastavlja mijenjati i integrirati sve više različitih domena. U poglavlju 2 dan je pregled tehnologija i jezika za razvoj web aplikacija. Poglavlje 3 opisuje mogućnosti primjene C++ jezika unutar Weba, dok se poglavlje 4 fokusira na WebAssembly. Poglavlje 5 obrađuje praktični dio rada u obliku aplikacije koja koristi WebAssembly, te je u poglavlju 6 predstavljen zaključak.

2. Tehnologije i jezici za razvoj web aplikacija

U ovom poglavlju će se objasniti pojmovi kao što su Web i Internet, protokoli, web preglednik, web aplikacija, klijent i poslužitelj. Prikazat će se osnovne tehnologije i jezici korišteni za razvoj web aplikacija, te će biti posebno obrađene tehnologije za klijentsku i za poslužiteljsku stranu.

2.1. Što su web aplikacije

Osnovni pojmovi koji se u svakodnevnom govoru često izmjenjuju su Internet i Web, iako nisu sinonimi. Internet je fizička mrežna infrastruktura putem koje se ostvaruje Web, odnosno računalna mreža u kojoj je komunikacija između milijuna međusobno povezanih domaćina ili krajnjih sustava - koji mogu biti stolna ili prijenosna računala, poslužiteljska računala, mobiteli, tableti, bilo kakvi uređaji koji su dio Interneta stvari (eng. *Internet of Things, IoT*) i slično - ostvarena žičanim i bežičnim vodovima, od DSL (eng. *Digital Subscriber Line*) telefonskih linija, kabelskih mreža i radiovalova do satelitskih kanala. Način slanja i primanja poruka putem Interneta, odnosno njihov redoslijed i format, određuje se putem internetskih protokola. Na ovaj način se Internet može promatrati kao infrastruktura koja pruža usluge distribuiranim aplikacijama, to jest, aplikacijama koje se izvršavaju na više domaćina ili krajnjih sustava koji međusobno izmjenjuju podatke, kao što su e-pošta, VoIP (eng. *Voice over Internet Protocol*), pa i Web. Stoga nisu sve internetske aplikacije ujedno i web aplikacije, iako većina jest. (Kurose et al., 2017)

Web je kolekcija međusobno povezanih aplikacija i uz njega vežemo pojmove kao što su web preglednik i web stranica. Web preglednik je aplikacija koja omogućuje pristup Webu, prepoznaje HTML sintaksu i može prikazivati web stranice. Prema tome, web aplikacije su sve aplikacije koje se nalaze na Webu, odnosno softver koji se izvršava unutar web preglednika, generira web stranice koje korisnik može pregledavati i putem kojih je web aplikacija zadužena za logiku prikaza, primanja i spremanja podataka od korisnika.

Kao što Internet funkcionira na arhitekturi klijent-poslužitelj (eng. *client-server*), na isti način možemo podijeliti i arhitekturu Weba, gdje web preglednik predstavlja klijenta, a web poslužitelj poslužitelja. Klijent upućuje zahtjev putem protokola, a poslužitelj osluškuje zahtjeve na mrežnim vratima (eng. *port*) i vraća odgovor klijentu, najčešće u obliku tražene web stranice.

Web aplikacije se mogu svrstavati po različitim kategorijama kao što su LMS (eng. *Learning Management System*), CMS (eng. *Content Management System*), društvene mreže,

e-trgovine i druge, no generalno je njihova glavna primjena obavljanje poslovnih aktivnosti bez potrebe za instalacijom softvera. Mogu biti pisane u različitim programskim jezicima i njihov se razvoj može podijeliti na razvoj sučelja aplikacije (eng. *frontend*) i razvoj pozadinske logike aplikacije (eng. *backend*), te se po tome dijele programski jezici i tehnologije koje se koriste u razvoju web aplikacije. One će biti detaljnije pokrivene u poglavljima 2.2. i 2.3. (Flanagan, 2020; Frisbie, 2020)

2.1.1. Komunikacija web klijenta i web poslužitelja

Prilikom slanja zahtjeva poslužitelju, korisnik unosi adresu željenog web mjesta ili stranice u web preglednik, tako kreirajući poruku zahtjeva koja se prosljeđuje kroz internetske slojeve i pripadajuće protokole te putem internetske mreže putuje do poslužitelja. Protokoli koji se uobičajeno koriste na aplikacijskom sloju su DNS (eng. *Domain Name System*) i HTTP/HTTPS (eng. *HyperText Transfer Protocol (Secure)*). Prilikom predavanja adrese traženog web mjesta, preglednik treba odrediti IP (eng. *Internet Protocol*) adresu odredišnog računala, odnosno poslužitelja kojemu se šalje zahtjev, te putem DNS protokola kreira i šalje zahtjev te prima odgovor o pripadnoj IP adresi za predani naziv. Tada preglednik kreira i šalje zahtjev web poslužitelju na pripadnoj IP adresi putem HTTP protokola, koji definira format poruka koje razmjenjuju web preglednici i web poslužitelji. Poslužitelj osluškuje HTTP zahtjeve na portu 80, obrađuje zahtjev i vraća odgovor klijentu, najčešće u obliku HTML (eng. *HyperText Markup Language*) dokumenta kojeg web preglednik interpretira i prikazuje korisniku, te po potrebi dohvaća slike, CSS (eng. *Cascading Style Sheet*) i JavaScript datoteke i druge datoteke referencirane u dokumentu. Ovakav primjer zahtjeva je GET zahtjev, a osim njega HTTP protokol definira i druge zahtjeve od kojih su najčešće korišteni POST, PUT i PATCH za dodavanje ili izmjenu podataka i DELETE za njihovo brisanje.

Zbog potrebe za sigurnošću prilikom razmjene poruka između web klijenta i poslužitelja se danas umjesto HTTP protokola češće koristi HTTPS protokol, za kojeg je rezerviran port 443. Kako je veza koja koristi HTTP protokol u teoriji vidljiva bilo kome na mreži te kako web poslužitelj s kojim preglednik komunicira ne mora potvrditi svoj identitet klijentu, postoji ranjivost od krađe privatnih podataka koji se šalju preko mreže te ranjivost od komuniciranja s malicioznom stranom koja se predstavlja kao legitimna klijentu. Kod HTTPS protokola (gdje „S“ predstavlja „secure“) se prije razmjene podataka između web klijenta i poslužitelja izvodi rukovanje (eng. *three-way handshake*), odnosno postupak od tri koraka u kojemu klijent i poslužitelj razmjenjuju svoje javne kriptografske ključeve te poslužitelj potvrđuje svoj identitet klijentu putem digitalnih certifikata, i nakon toga su sve daljnje poruke između njih kriptirane.

U sljedećim poglavljima će se posebno obraditi klijentska i posebno poslužiteljska strana. Komunikacijski kanal između obje strane čini API (eng. *Application Programming Interface*) koji je građen s poslužiteljske strane i kojeg klijentska strana koristi za dohvat podataka i njihov prikaz u sučelju web aplikacije. (Kurose et al., 2017)

2.2. Klijentska strana

Sučelje web aplikacije ili *frontend* pokriva dio web aplikacije koji se odnosi na sve što korisnik vidi, odnosno na sučelje web aplikacije vidljivo korisniku u obliku web stranice unutar preglednika. Često se ovaj izraz koristi kao sinonim za klijentsku stranu, iako sučelje web aplikacije i klijentska strana ne moraju nužno biti isti. Klijentska strana se odnosi na sve procese web aplikacije koji se izvršavaju na računalu klijenta, uključujući one procese koji nisu vidljivi klijentu ili dio sučelja. Jedan takav primjer je straničenje ili paginacija tablica koja se izvršava na klijentu, jer iako je rezultat vidljiv na sučelju u obliku podjele podataka po stranicama, sama logika straničenja se izvršava na klijentu što znači da je osim sučelja (*frontend*), i pozadinska logika (*backend*) u ovom slučaju dio klijentske strane aplikacije. Isto tako, pozadinska logika koja provjerava JWT (eng. *JSON Web Token*) token ili na drugi način radi autentifikaciju korisnika prije slanja određenih zahtjeva na poslužiteljsku stranu web aplikacije može biti dio klijentske strane, no nije vidljiva korisniku i time nije dio sučelja aplikacije. Prema tome, klijentska strana sadrži sve što web aplikacija prikazuje korisniku u sučelju odnosno unutar preglednika, od teksta, slika i ostatka UI-ja (eng. *user interface*), zajedno sa svim akcijama koje aplikacija mora izvesti unutar preglednika. Primjer takve akcije je i ako korisnik pozicionira kursor miša iznad slike, te se na to slika poveća i odmakne sve ostale elemente koji je okružuju. Ovakve akcije programski kod unutar web stranice izvodi na klijentskoj strani bez slanja zahtjeva poslužitelju. (Cloudflare, bez dat.)

2.2.1. Tehnologije za klijentsku stranu

Osnovne tehnologije koje se koriste u razvoju klijentske strane su HTML, CSS i JavaScript. Jezike HTML i CSS interpretira preglednik na klijentskoj strani, dok se određeni dio poslovne logike, koji skoro uvijek pisan u JavaScriptu, također može biti optimalnije izvoditi na klijentskoj nego li na poslužiteljskoj strani. Što se tiče pohrane podataka, primarno se koriste baze podataka na poslužiteljskoj strani, no na klijentskoj strani je također omogućena privremena pohrana podataka u pregledniku putem web spremišta (lokalno spremište i spremište sjednice) ili kolačića. Svrha spremanja podataka na klijentskoj strani je smanjenje nepotrebnih zahtjeva prema poslužitelju, poboljšanje korisničkog iskustva, ali i izgradnja

progresivnih web aplikacija putem servisnih radnika, koje ovise o spremanju podataka lokalno kako bi im mogle pristupiti i bez internetske veze.

2.2.1.1. HTML i CSS

HTML je jezik oznaka koji uređuje strukturu web stranice tako da sadrži elemente koje web preglednici mogu interpretirati i na specifičan način prikazati korisniku. Koristi se u HTML dokumentima kao skup oznaka koje se dodaju dijelovima teksta kako bi se oni označili ili kako bi se referencirale vanjske poveznice za druge HTML dokumente, video, slike i slično. Trenutna verzija je HTML 5.

CSS je opisni jezik koji se koristi za stiliziranje web stranica tako da se manipuliraju stilska obilježja selektiranih HTML elemenata. Na taj se način korištenjem CSS-a uz HTML postiže odvajanje sadržaja stranice od njezine prezentacije. Mogućnosti CSS-a obuhvaćaju uređivanje fontova, boja, animacija, drugačiji izgled ovisno o veličini i tipu uređaja poput ekrana mobitela, desktopa ili tableta, pisača i slično.

2.2.1.2. JavaScript na klijentskoj strani

Osim semantike web stranice (HTML) i njezinog stila (CSS), kroz evoluciju prema modernim web stranicama nastala je potreba za prijelazom web stranica iz statičkih, koje su bile iste za sve korisnike, prema dinamičkim, koje mijenjaju sadržaj ovisno o korisniku i njegovoj interakciji sa stranicom. Dijelom se ovo postiže dinamičkim sastavljanjem stranica na strani poslužitelja, no radi smanjenja zahtjeva s klijentske prema poslužiteljskoj strani, a time i latencije te poboljšanja korisničkog iskustva, i klijentska strana je dobila programske mogućnosti preko programskog jezika koji bi, za razliku od opisnih jezika HTML i CSS, mogao obavljati logiku korisničke interakcije. Od takvih jezika, JavaScript je univerzalno podržan u svim preglednicima, dok podrška za izvršavanje skripti pisanih u drugim programskim jezicima ovisi o pojedinom pregledniku. Iz tog razloga se kao programski jezik za klijentsku stranu gotovo uvijek koristi JavaScript, koji je inače prototipski orijentirani jezik s objektno-orijentiranim mogućnostima i čiji je standard ECMAScript.

Na klijentskoj strani JavaScript koristi DOM (engl. *Document Object Model*) i BOM (eng. *Browser Object Model*) kao API-je za rad s HTML dokumentima. Preko DOM-a upravlja HTML elementima na sučelju web stranice s mogućnostima njihovog dodavanja, mijenjanja i brisanja, te s događajima poput pomaka miša ili klika na gumb, dok preko BOM-a upravlja prozorom preglednika, poviješću, URL-om (eng. *Uniform Resource Locator*) i slično. JavaScript se često koristi i za mijenjanje stila određenog elementa ovisno o nekom kontekstu poput npr. korisničkog odabira svijetle ili tamne teme, tako da mijenja nazive klasa dodijeljene HTML elementima i time CSS stil koji se primjenjuje na te elemente. Osim toga, preko AJAX (eng.

Asynchronous JavaScript and XML) zahtjeva, *fetch* API-ja ili drugih alternativa se ostvaruje dohvaćanje podataka od poslužiteljske strane za prikaz na klijentskoj strani. (Flanagan, 2020; Frisbie, 2020)

2.2.1.3. Okviri i biblioteke za klijentsku stranu

Osim navedenih tehnologija, postoje različiti okviri i biblioteke za rad na klijentskoj strani, koje olakšavaju i ubrzavaju proces razvoja klijentske strane za web aplikaciju, ali u slučaju okvira nameću i neka ograničenja. Jedna od najčešće korištenih JavaScript biblioteka je jQuery, koja olakšava rad s DOM-om tako da pojednostavljuje sintaksu za manipulaciju HTML elemenata, rad s događajima i kreiranje AJAX zahtjeva. React se često smatra okvirom, no bilo bi preciznije reći da je također riječ o JavaScript biblioteci koja koristi okvir CRA (eng. *Create React App*) kako bi ju se moglo koristiti kao okvir, i koja se često koristi u sklopu okvira Next.js za izgradnju jednostraničnih (eng. *Single Page Application,* SPA) aplikacija. (Vytenis A., 2022) Od okvira za klijentsku stranu su najpoznatiji Angular i Vue. Vue i ranija verzija Angulara, AngularJS, su okviri pisani u JavaScriptu. S druge strane, Angular koristi TypeScript, dok React nudi kreiranje projekta i u JavaScriptu i u TypeScriptu. TypeScript je verzija JavaScripta koja uvodi strogo tipiziranje i kompilaciju, odnosno prevođenje koda u JavaScript kako bi olakšala pronalaženje grešaka (eng. *debugging*). Osim navedenih, postoje i brojne druge biblioteke i okviri različitih namjena, a njihov izbor ovisi o potrebama pojedine web aplikacije.

2.3. Poslužiteljska strana

Slično pojmu klijentske strane, poslužiteljska strana odnosi se na sve procese web aplikacije koji se odvijaju na poslužitelju. Kao što je slučaj s pojmovima „klijentska strana“ i „*frontend*“, poslužiteljska strana se često naziva i *backend,* iako i tu postoji distinkcija. Poslužiteljska strana se odnosi na lokaciju izvršavanja procesa, odnosno na to da je poslužitelj takva lokacija. Pozadinska logika ili *backend* se odnosi na vrstu procesa web aplikacije, odnosno na sve procese koji nisu vidljivi korisniku, dakle nisu dio korisničkog sučelja. Kako je klijentska strana prezentacijski sloj za podatke, poslužiteljska strana je odgovorna za njihovo trajno spremanje korištenjem neke baze podataka. U prošlosti je skoro sva poslovna tj. pozadinska logika bila na poslužiteljskoj strani, poput prikaza dinamičkih web stranica (odnosno onih stranica koje imaju drugačiji sadržaj za različite korisnike), *push* notifikacija, komunikacije s bazom i autentifikacijske logike. Kako bilo koji zahtjev s klijentske strane koji uključuje neku od ovih usluga mora svaki put putovati od klijentskog do poslužiteljskog računala, povećava se latencija, odnosno smanjuju performanse web aplikacije i time i

korisničko iskustvo. Radi toga je danas poželjno premjestiti dio pozadinske logike s poslužiteljske na klijentsku stranu, npr. tako da prikaz dinamičkog sadržaja web stranica izvršavaju skripte u pregledniku, čime se smanjuje broj zahtjeva prema poslužitelju i ubrzava rad aplikacije. (Cloudflare, bez dat.)

2.3.1. Tehnologije za poslužiteljsku stranu

2.3.1.1. Web poslužitelji

Za ostvarivanje poslužiteljske strane potreban je poslužitelj, koji se s gledišta Interneta može definirati kao uvijek aktivno računalo u mreži koje razmjenjuje podatke s drugim povezanim uređajima ili klijentima na njihov zahtjev, a s gledišta Weba kao softver na tom računalu, odnosno HTTP poslužitelj koji interpretira web adrese ili URL-ove te koristi HTTP protokol za komunikaciju s klijentima putem web preglednika. Web poslužitelji se mogu podijeliti na statičke i dinamičke, gdje su statički oni za koje je dovoljno računalo s HTTP poslužiteljem koje pohranjuje HTML i CSS dokumente te JavaScript skripte, te ih nepromijenjene šalje klijentu. Dinamički web poslužitelji dodatno imaju aplikacijski poslužitelj i bazu podataka, gdje je aplikacijski poslužitelj zadužen za poslovnu logiku i obradu sadržaja dokumenata prije nego što ih pošalje klijentu, primjerice sastavljanjem dinamičkih web stranica tako da popuni jedan HTML predložak varijabilnim sadržajem iz baze podataka, umjesto da sadrži više statičkih HTML dokumenata za svaki od mogućih sadržaja. Neki od web poslužitelja u uporabi danas su Nginx, Apache HTTP server, Apache Tomcat, IIS (eng. *Internet Information Services*) i drugi. (MDN, bez dat.)

2.3.1.2. Web servisi

Web servisi predstavljaju način komunikacije s drugim aplikacijama pružanjem određene usluge, odnosno podataka preko neke mrežno dostupne točke (eng. *endpoint*), neovisno o tehnologijama koje koriste aplikacije koje komuniciraju. Najkorišteniji arhitekturni standardi u izgradnji web servisa su REST (eng. *Representational State Transfer*), SOAP (eng. *Simple Object Access Protocol*) i GraphQL, te se koriste u razvoju API-ja poslužiteljske strane prema klijentskoj.

2.3.1.3. Baze podataka

Još jednu važnu komponentu u razvoju poslužiteljske strane čine baze podataka. One služe za perzistenciju podataka, odnosno za njihovu trajnu pohranu te s njima po potrebi komunicira web poslužitelj. Sustavi za upravljanje bazama podataka često korišteni u web aplikacijama su MySQL, PostgreSQL, Oracle, MongoDB, Firebase, Redis i drugi, te se uz

primarnu ili glavnu bazu generalno koristi i sekundarna baza koja služi kao rezervna, kako bi se mogao nastaviti rad web aplikacije u slučaju da je pristup glavnoj bazi onemogućen.

2.3.1.4. Okviri i biblioteke za poslužiteljsku stranu

Za razliku od klijentske strane na kojoj je JavaScript univerzalno podržani jezik, okviri i biblioteke za poslužiteljsku stranu se pronalaze u različitim jezicima. Popularni su PHP okviri Laravel i Symfony, Python okviri Django i Flask, Ruby okvir Ruby on Rails, Java okvir Spring, ASP.NET i drugi. Godine 2009. je Ryan Dahl razvio NodeJS, programsko okruženje koje koristi Googleov V8 JavaScript Engine te na taj način dozvoljava izvršavanje JavaScript koda izvan web preglednika. Na temelju NodeJS-a su razvijeni različiti okviri za poslužiteljsku stranu pisani u JavaScriptu, kao što su Express, Koa i NestJS. Iako se mogu koristiti različite kombinacije tehnologija za klijentsku i za poslužiteljsku stranu u izgradnji web aplikacije, nerijetko će izbor tehnologije jedne strane ovisiti o tehnologiji korištenoj na drugoj strani. Primjerice, sintaksa NestJS-a je dizajnirana po uzoru na Angular, pa se često koristi u kombinaciji s njime.

3. Primjena programskog jezika C++ u razvoju web aplikacija

Programski jezici JavaScript i C++ se razlikuju po nekim ključnim karakteristikama, pa će se u ovom poglavlju napraviti usporedba jezika te razmotriti mogućnosti primjene C++ jezika u web aplikacijama na strani poslužitelja i na strani klijenta.

3.1. Usporedba jezika C++ i JavaScript

JavaScript je razvio Brendan Eich 1995. godine pod originalnim nazivom LiveScript. Skriptni je jezik, što znači da se za njegovo izvršavanje koristi interpreter ili JIT (eng. *Just In Time*) kompilator. U slučaju interpretera se kod izvršava liniju po liniju, dok JIT kompilator analizira kod i prevodi najčešće korištene dijelove koda u *bytecode* te potom u strojni kod. Kako je JavaScript bio originalno dizajniran za izvršavanje u web preglednicima, odnosno unutar JavaScript stroja SpiderMonkey kojeg je također razvio Eich, koristio se interpreter kako bi se datoteke poslane s poslužitelja mogle brzo interpretirati i prikazati unutar preglednika. S vremenom su se performanse pokazale kao problem jer interpreteri ne mogu vršiti optimizaciju koda na isti način kao kompilatori, primjerice pojednostavljivanjem ponavljajućeg koda unutar petlji. Radi toga je uvedeno izvođenje JavaScript koda preko JIT kompilatora, jer JIT kompilator nastoji iskoristiti benefite i interpretera i kompilatora analizom i prevođenjem često ponavljajućih dijelova koda kako bi oni mogli biti optimizirani tijekom izvršavanja, te su na taj način preglednici postali brži.

S druge strane, C++ je kompilatorski jezik kod kojeg kompilator vrši statičku analizu koda prije njegovog prevođenja u strojni kod. Za razliku od interpretera gdje se kod izvršava odmah, kompilatoru je potrebno neko vrijeme za prolazak kroz kod i njegovu analizu, no optimizacije koje napravi generalno rezultiraju boljim performansama prilikom izvođenja. Isto tako, kompilator obavlja proces prevođenja koda prije njegovog izvršavanja, za razliku od JIT kompilatora koji kombinira karakteristike interpretera i kompilatora te prevodi dijelove koda tijekom izvršavanja koda od strane interpretera.

Iduća važna razlika ovih jezika je u tipiziranju. JavaScript je dinamički tipiziran jezik, što znači da se tijekom izvršavanja varijablama dodjeljuju tipovi na temelju vrijednosti koje sadrže, te nije nužno dodijeliti tip varijabli prilikom njezine definicije. Posljedično je moguće mijenjati tip varijable tijekom izvođenja. C++ je statički tipiziran, što znači da tipovi varijabli moraju biti poznati tijekom faze prevođenja od strane kompilatora, te je potrebno dodijeliti tipove

varijablama i oni se ne mogu mijenjati tijekom izvođenja. Glavna prednost statički tipiziranih jezika je da je lakše ranije pronaći mnoge trivijalne greške tijekom faze prevođenja, dok bi takve greške vezane uz tipove mogle proći neopaženo kod dinamički tipiziranih jezika. Ovo je bila i motivacija za razvoj TypeScripta, verzije JavaScripta koja uvodi strogo tipiziranje.

Jezici sa strogim tipiziranjem, odnosno sistemski jezici poput C, C++ i Jave, generalno imaju manji broj instrukcija nižih razina (pa sve do instrukcija strojnog koda) ispod svake vlastite instrukcije, dok je kod jezika sa slabijim tipiziranjem, a to su obično skriptni jezici poput JavaScripta, potrebno izvršiti veći broj strojnih instrukcija za svaku instrukciju napisanu u tom jeziku. (Ousterhout, 1998) Prema tome je JavaScript jezik visoke razine, dok se C++ obično smatra jezikom srednje razine, jer kao i C omogućuje nisku razinu programiranja bližu strojnom jeziku, ali zbog vlastitog objektnog modela omogućuje i visoku razinu programiranja. U svakom slučaju, JavaScript je jezik više razine od C++, što znači da će ekvivalentni kodovi općenito biti kraći u JavaScriptu nego u C++-u, no osim toga ovo igra važnu ulogu i kod usporedbe performansi.

Što se tiče paradigmi jezika, JavaScript spada pod više njih, uključujući funkcionalne programske jezike s obzirom da su funkcije u JavaScriptu entiteti prvog reda, odnosno mogu se predavati funkcijama višeg reda kao argumenti ili biti vraćene kao rezultat takve funkcije, te se mogu pridruživati varijablama na isti način kao i bilo koji primitivni tip. No nije čisti funkcionalni jezik jer ima i osobine objektno-orijentiranog jezika putem modela klasa. U biti je model klasa svojevrstan „sintaktički šećer“ jer je baziran na modelu prototipnog lanca, kako je JavaScript prototipno-orijentiran, a ne objektno-orijentiran. S druge strane, C++ je razvio Bjarne Stroustrup s namjerom da jeziku C doda klase, pa se tako C++ generalno svrstava pod paradigmu objektno-orijentiranih jezika, gdje se objekti instanciraju preko klasa u kojima su objedinjene metode i atributi te se slijede principi enkapsulacije, nasljeđivanja i polimorfizma. Isto tako, kako se u C++ jeziku može sve što se može i u C jeziku, podržava i proceduralnu paradigmu.

Zbog činjenice da je C++ jezik niže razine od JavaScripta i time bliži strojnom jeziku, performanse C++ koda su daleko bolje od JavaScript koda. Ipak, nema previše smisla direktno uspoređivati performanse ovih jezika, kako su jezici različite namjene, razvijeni za drugačije svrhe i u drugačijim kontekstima. JavaScript je prvenstveno razvijen za web, originalno za klijentsku stranu u preglednicima te kasnije i za poslužiteljsku stranu putem NodeJS-a, dok se C++ smatra jezikom opće namjene. Kako C++ dozvoljava upravljanje memorijom na niskoj razini i rad s pokazivačima, najčešće je korišten za operacijske sustave, ugrađene sustave, aplikacije koje zahtijevaju visoke performanse poput procesiranja slika, igre i slično. JavaScript ima automatsko upravljanje memorijom (eng. *garbage collection*) te su za potrebe Weba

performanse JavaScript koda dovoljne, pogotovo u kombinaciji s JIT kompilatorom i raznim optimizacijskim tehnikama prilikom dizajna arhitekture web aplikacija, te je lakše, brže i praktičnije pisati kod u njemu jer je jezik visoke razine i bliži prirodnom jeziku. C++ kod je teži za pisanje, no performanse C++-a su idealne za zahtjevne aplikacije i izračune za koje se uglavnom i koristi. (Scully, 2019)

Oba jezika je moguće koristiti za različite druge svrhe, pogotovo s raznim okvirima i bibliotekama koji su u međuvremenu razvijeni za njih, no primarno se koriste u ranije navedenim područjima. Zanimljivo je da je C++ često korišten i za pisanje kompilatora za druge jezike, uključujući JavaScript stroj (eng. *JavaScript engine*). Primjerice, SpiderMonkey, JavaScript/WebAssembly stroj koji je razvio Eich i koji se i danas koristi u pregledniku Firefox za izvršavanje JavaScripta, dijelom je pisan u C++ jeziku. (SpiderMonkey, bez dat.)

3.2. Mogućnosti primjene C++ jezika u web aplikacijama

Što se tiče igara u preglednicima, JavaScript je dobar izbor za jednostavnije 2D igre koje ne zahtijevaju toliko resursa. Moguće je izraditi i 3D igru korištenjem JavaScript biblioteka kao što je Three.js, no C++ se može koristiti za zahtjevniju igru za koju je potrebno više izračuna u realnom vremenu.

Kao još jedna domena u kojoj C++ može biti koristan radi direktnog pristupa hardverskim resursima i mogućnosti kreiranja malih izvršnih datoteka mogu biti ugrađeni sustavi, koji u pravilu imaju ograničene resurse. Primjer takve primjene prikazan je u kreiranju web aplikacije koja implementira strojno učenje u ugrađene sustave, gdje se iz tih razloga koristi C++ za poslužiteljsku stranu. (Zidek et al., 2017)

C++ se često koristi i u grafičkom programiranju. OpenGL je API, odnosno C++ specifikacija za programiranje grafike koja stvara apstrakciju nad hardverskim resursima i omogućuje izvođenje 2D i 3D grafike na različitim platformama. C++ programeri mogu koristiti OpenGL za upravljanje grafičkom karticom i renderiranje 3D objekata, primjene tekstura, osvjetljavanja itd., najčešće za razvoj igara i grafičkih aplikacija. U kontekstu web aplikacija je moguće integrirati korištenje OpenGL-a preko WebAssemblyja, no to nije zastupljeno jer već postoji WebGL, web verzija OpenGL-a koja omogućuje izvršavanje OpenGL grafike unutar web preglednika. WebGL koristi JavaScript za programiranje, ali u pozadini koristi OpenGL API-je kako bi web aplikacijama omogućio visokokvalitetnu grafiku. (Gonsalves, 2021; OpenGL Overview - The Khronos Group Inc, bez dat.)

C++ može imati značajnu primjenu u web aplikacijama unutar domene računalnog vida i obrade slika, posebno kada se koristi u kombinaciji s popularnom bibliotekom OpenCV, gdje

njezina integracija omogućuje razvoj web aplikacija koje mogu izvoditi napredne operacije na slikama, poput detekcije objekata, praćenja pokreta, segmentacije slika ili prepoznavanja oblika. Integracija je moguća na strani poslužitelja, kao što će se prikazati u poglavlju 3.4., ili na strani klijenta putem WebAssemblyja. (OpenCV: OpenCV Modules, bez dat.; Taheri et al., 2018)

3.3. Primjena C++ jezika na strani poslužitelja

Postoji nekoliko okvira koji omogućuju razvoj web aplikacija u C++. Jedan od njih je Wt, web okvir prvenstveno za razvoj na strani poslužitelja koji pruža značajke za obradu HTTP zahtjeva, obradu podataka, implementaciju poslovne logike i generiranje dinamičkog HTML sadržaja koji će se poslati klijentu. Wt također uključuje skup unaprijed izgrađenih komponenti korisničkog sučelja (eng. *widget*) koji se mogu koristiti za stvaranje sučelja na klijentskoj strani, kao što su gumbi, obrasci, tablice, grafikoni itd., te se mogu kombinirati kako bi se stvorile interaktivne web aplikacije. Ove komponente se renderiraju kao HTML, CSS i JavaScript pomoću okvira i šalju klijentu kao dio odgovora poslužitelja. Klijentska strana se može dodatno proširiti drugim bibliotekama ili okvirima, no moguće je koristiti samo Wt bez potrebe za JavaScriptom na klijentskoj strani, čime se eliminira potreba za prelaskom između različitih programskih jezika za razvoj na poslužiteljskoj i klijentskoj strani. Wt koristi paradigmu programiranja temeljenog na događajima (eng. *event-based programming*), kao što je interakcija korisnika klikom na gumb ili podnošenjem obrasca. Namijenjen je za jednostranične aplikacije i moguće je graditi REST servise preko njega, te može koristiti bilo koje C/C++ biblioteke ovisno o potrebama domene. (Wt, C++ Web Toolkit — Emweb, bez dat.) Iako je Wt poznati C++ web okvir inspiriran Qt-om, postoje i drugi kao što su CppCMS, Pistache i Crow.

Osim web okvira specifično pisanih za razvoj poslužiteljske strane u C++, postoji mogućnost pokretanja programa pisanih u C++ od strane poslužitelja pisanih u drugim jezicima. Ovo bi se moglo primijeniti u slučajevima kada je za neki zadatak potrebno procesiranje u C++ te se odgovarajući parametri predaju programu, a rezultat programa poslužiteljska strana vraća klijentu. Primjer web aplikacije koja koristi ovaj slučaj može se prikazati kod procesiranja slike. Procesiranje slika je područje koje je računalno zahtjevno radi složenih algoritama koji se primjenjuju na slike i velikog broja izračuna za svaki pojedini piksel, te se izvršavanje tog dijela može prepustiti C++ programu, dok poslužitelj pisan u NodeJS-u ili drugom okviru samo prima zahtjeve od klijenta, predaje potrebne parametre programu i od njega dobiva sliku koju vraća klijentu. Za procesiranje slika se u tom slučaju može iskoristiti OpenCV, otvorena biblioteka računalnog vida i obrade slika koja nudi C++ sučelje prema alatima za analizu i manipulaciju slika i videozapisa.

3.4. Shell skripta

U ovom poglavlju će se prikazati uporaba C++ programa pokrenutog kao shell skripta od strane NodeJS skripte. Kao primjer je izrađena web aplikacija u koju korisnik može učitati sliku te odabrati vrstu filtra koji će se primijeniti na sliku. Klijentska strana šalje sliku i informaciju o odabranom filtru na poslužiteljsku stranu, gdje NodeJS skripta pokreće C++ program kao shell skriptu i predaje mu putanju privremene lokacije poslane slike te vrstu filtra. C++ program koristi algoritme iz OpenCV biblioteke za procesiranje slike i sprema ju na poslužiteljskoj strani. Kada je C++ gotov s procesiranjem slike, odnosno shell skripta je izvršena, putem funkcije povratnog poziva NodeJS skripta dohvaća obrađenu sliku s te lokacije i šalje ju na klijentsku stranu, gdje se korisniku ona prikazuje u pregledniku.

Prvo je izrađen OpenCV projekt za potrebe testiranja algoritama i pritom je korišten CMake, alat za izgradnju softvera koji se koristi za automatizaciju procesa kompiliranja i izgradnje projekata. CMake je otvorenog koda i pruža sučelje za izgradnju softvera na različitim platformama i prevoditeljima. Unutar direktorija projekta se u datoteci `CMakeLists.txt` specificira konfiguracija projekta. U konfiguraciji ovog projekta je definirano pronalaženje biblioteka za OpenCV, te njihovo povezivanje s datotekom `main.cpp` unutar direktorija projekta. Konfiguracija generira skripte za izgradnju projekta unutar direktorija `build` koje služe za stvaranje izvršne datoteke C++ programa `opencvtest.exe` unutar direktorija `build/Debug`. (Escriva et al., 2019)

Idući korak je bio stvaranje NodeJS skripte koja će moći pokrenuti izvršnu datoteku `opencvtest.exe` i predati joj putanju do slike te odabrani filter kao argumente, što se postiže uvođenjem modula `shelljs`. Funkcijom iz tog modula `exec` moguće je pokrenuti izvršavanje vanjskog programa iz NodeJS skripte, te se njoj predaje naredba za izvršavanje programa na specificiranoj putanji te argumenti koji se prosljeđuju programu.

U NodeJS je potom uključen modul Express za posluživanje HTML datoteke koja omogućuje korisniku učitavanje slike i prikaz procesiranih slika vraćenih od poslužitelja. Kod prijenosa slike na poslužiteljsku stranu korišten je modul `multer` kao *middleware*, te se preko njega poslana slika sprema u direktorij `uploads` na poslužitelju.

```
const multer = require("multer");
const storage = multer.diskStorage({
    destination: (req, file, callback) => {
      const uploadPath = path.join(__dirname, "uploads");
      fs.mkdirSync(uploadPath, { recursive: true });
      callback(null, uploadPath);
```

```
    },
    filename: (req, file, callback) => {
      callback(null, file.originalname);
    },
});
const upload = multer({ storage });
```

Na putanji `/process` se obrađuje korisnikov POST zahtjev u se kojem *middleware* `upload.single('image')` brine za spremanje slike koju očekuje iz primljenog zahtjeva pod nazivom „image“ jer je pod tim imenom definirana u formi poslanoj s klijenta. Nakon toga se modificirani zahtjev predaje sljedećoj funkciji u obradi zahtjeva i ona pristupa lokaciji slike (`req.file.path`) koja je sada u direktoriju `uploads` te vrijednosti atributa `req.body.filter`, definira putanju do izvršne C++ datoteke i izvršava njezino pokretanje s putanjom slike i filtrom kao predanim argumentima. U funkciji povratnog poziva koja se izvrši nakon što je C++ program završio, dohvaća se obrađena slika s putanje na poslužitelju na koju ju je spremio C++ program i šalje klijentu za prikaz.

```
app.post("/process", upload.single("image"), (req, res) => {
  const scriptPath = path.join(__dirname, ".", "build", "Debug",
"opencvtest.exe");
  const imagePath = req.file.path;
  const filter = req.body.filter;
  const command = `${scriptPath} "${imagePath}" ${filter}`;
  shell.exec(command, (error, stdout, stderr) => {
    if (error) {
      console.error(`Greška u izvođenju C++ programa: ${error}`);
      res.status(500).send(`Greška u izvođenju C++ programa: ${error}`);
      return;
    }
    console.log(`C++ program uspješno izvršen`);
    const processedImagePath = path.join(__dirname, "processed_image.jpg");
    res.sendFile(processedImagePath);
  });
});
```

OpenCV program kojeg pokreće NodeJS skripta provjerava je li primljen traženi broj argumenata, odnosno tri jer je prvi argument argv[0] uvijek ime izvršne datoteke, dok su druga dva argument putanje slike na poslužitelju i vrsta filtra. Pritom se sve poruke zapisuju u datoteku `logs.txt` na poslužitelju, koja se stvara ako ne postoji, te ako postoji otvara se u modu *append* (`ios_base::app`).

```
int main(int argc, char** argv ) {
    ofstream outfile("logs.txt", ios_base::app);
    if (argc < 3) {
        outfile << "Nedostaje argument" << endl;
        return -1;
    }
    const char* imagePath = argv[1];
    const char* filter = argv[2];
```

Zatim se učitava slika sa specificirane lokacije i određuje vrsta odabranog filtra, na temelju koje se izvršava algoritam za procesiranje slike koji vrši transformaciju svakog pojedinačnog piksela. U slučaju *grayscale* filtra, OpenCV funkcija `cvtColor` prima kod konverzije `COLOR_BGR2GRAY` na temelju kojeg se računa težinski prosjek B, G i R vrijednosti piksela za postizanje crno-bijelog efekta, što se može jednostavno obaviti i bez pozivanja funkcije. Za Gaussov filter zamućenja se za svaki piksel uzima u obzir intenzitet susjednih piksela radi postizanja efekta manje izraženih, odnosno izglađenih detalja na slici. Canny algoritam za detekciju rubova je najsloženiji te se sastoji od nekoliko koraka u kojima se primjenjuje više filtara i izračuna da bi se došlo do slike na kojoj su jaki rubovi označeni, a slabiji rubovi i šum suzbijeni. (OpenCV: OpenCV Modules, bez dat.)

```
  Mat image;
  image = imread(imagePath);
  if ( !image.data ) {
        outfile << "Nema podataka za sliku" << endl;
        return -1;
  }
  if (strcmp(filter, "grayscale") == 0) {
        cvtColor(image, image, COLOR_BGR2GRAY);
  } else if (strcmp(filter, "blur") == 0) {
        GaussianBlur(image, image, Size(5, 5), 0);
  } else if (strcmp(filter, "canny") == 0) {
        Mat edges;
        Canny(image, edges, 50, 150);
        image = edges;
  } else {
        outfile << "Filter nije validan" << endl;
        return -1;
  }
```

Nakon procesiranja se obrađena slika sprema na poslužitelju pod nazivom `processed_image` te program bilježi završetak obrade i zatvara datoteku `logs.txt`.

```
  const char* processedImagePath = "processed_image.jpg";
  imwrite(processedImagePath, image);
  outfile << "Slika uspješno procesirana" << endl;
  outfile.close();
  return 0;
}
```

S ovime je završen dio koji se odnosi na poslužiteljsku stranu. Što se tiče strane klijenta, stvoreno je jednostavno sučelje preko HTML, CSS i JavaScripta u datoteci `index.html` koju poslužuje Express na portu 3000. Na stranici se nalaze gumb za učitavanje slike te tri gumba od kojih svaki predstavlja neki filtar („Grayscale“, „Gaussian blur“ ili „Canny Edge Detection“). Slika koju korisnik učita se prikazuje na sučelju, te se klikom na gumb učitana slika šalje poslužitelju i svaka procesirana slika koja se vrati kao odgovor, prikazuje se pokraj prethodne. JavaScript funkcija koja se poziva klikom gumba prima ime odabranog filtra i kreira objekt `FormData` u koji sprema sliku i filter pod imenima „image“ i „filter“, pod kojima će se moći dohvatiti na strani poslužitelja.

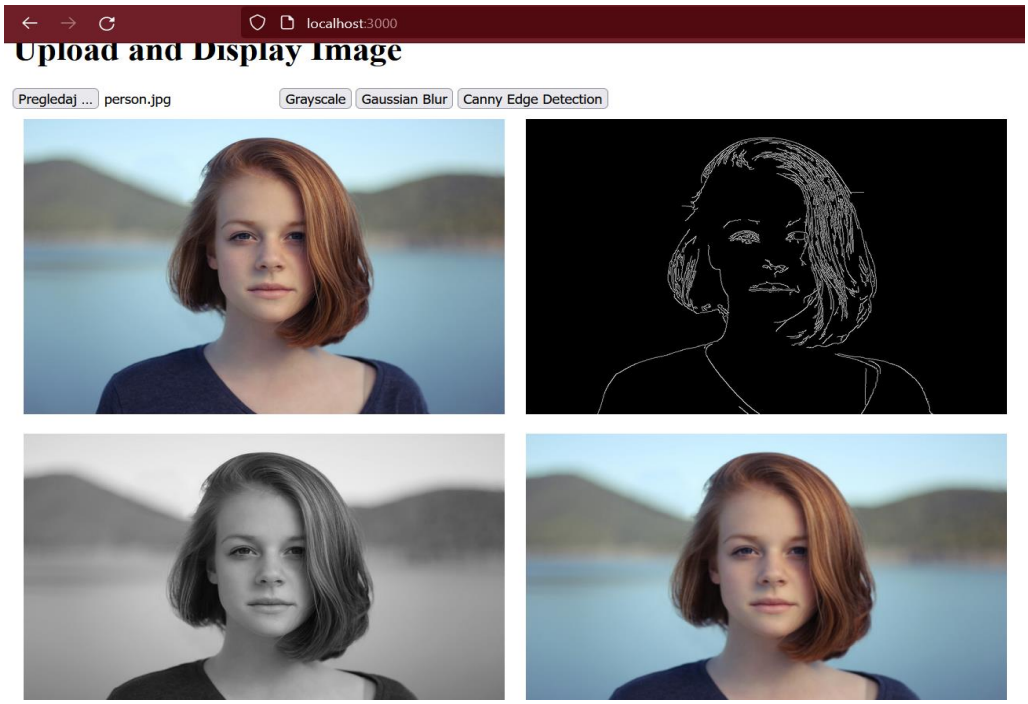
```
function uploadAndProcess(filter) {
  const imageUpload = document.getElementById("imageUpload");
  const imageFile = imageUpload.files[0];
  const formData = new FormData();
  formData.append("image", imageFile);
  formData.append("filter", filter);
```

Potom se kreira POST zahtjev na putanju `/process`. Kako iz `FormData` objekta koji se predaje kao tijelo POST zahtjeva `fetch` API sam prepoznaje da je `content-type` tipa `multipart/form-data`, nije potrebno to posebno specificirati. Kao odgovor na zahtjev se očekuje obrađena slika koja će se prikazati korisniku.

```
fetch("/process", { method: "POST", body: formData })
  .then((response) => response.blob())
  .then((blob) => {
    const imageUrl = URL.createObjectURL(blob);
    const img = document.createElement("img");
    img.src = imageUrl;
    const imageContainer = document.getElementById("imageContainer");
    imageContainer.innerHTML = "";
    imageContainer.appendChild(img);
  })
  .catch((error) => {
    console.error("Greška u procesiranju slike:", error);
```

```
    });
}
```

Rezultat ove aplikacije prikazan je na Slici 1, gdje je prva slika ona koju je korisnik učitao, na drugu je primijenjen Canny algoritam za detekciju rubova, na treću *grayscale* filtar i na četvrtu Gaussov filtar zamućenja.



Slika 1: Prikaz web aplikacije za procesiranje slika

3.5. Primjena C++ jezika na strani klijenta

Kako web preglednici razumiju samo JavaScript, postojala je motivacija za uvođenjem drugih jezika u kojima se mogu pisati web aplikacije i izvršavati u svojoj izvornoj brzini, bilo radi toga da se na Webu mogu izvršavati klijentske aplikacije za koje to nije bilo moguće jer nisu pisane u JavaScriptu, ili da se iskoriste prednosti drugog jezika. Radi toga je 2017. uveden WebAssembly (WASM), binarni izvršni format i standard otvorenog koda podržan od strane svih glavnih web preglednika. WebAssembly omogućuje programerima da napišu aplikacije na različitim programskim jezicima kao što su C, C++ i Rust, koje se mogu pokrenuti u web pregledniku bez potrebe za prevođenjem ili interpretiranjem, što pruža prednost po pitanju performansi. U sljedećem poglavlju će se detaljnije prikazati osnovni koncepti vezani uz uporabu WebAssemblyja te će se prikazati njegova primjena u praktičnom dijelu.

U kontekstu C++ jezika, WebAssembly pruža način za njegovu integraciju u web aplikacije na strani klijenta, te se i ovdje može primijeniti u područjima koja zahtijevaju dosta izračuna kao što su procesiranje slika i grafičko programiranje. Ovdje se također mogu

iskoristiti gotove biblioteke s učinkovitom implementacijom algoritama pisanih u C++ kao što je OpenCV i integrirati u web aplikacije putem WebAssemblyja. Jedan takav primjer prikazan je u radu s kalifornijskog sveučilišta, gdje se upravo radi toga što nema usporedive biblioteke računalnog vida za Web primjenjuje WebAssembly kako bi se OpenCV funkcije prevele u JavaScript kod. (Taheri et al., 2018) U tradicionalnom pristupu arhitekture klijent-poslužitelj gdje je poslužitelj bio zadužen za izvršavanje procesorski zahtjevnijih zadataka, to je bilo izazovnije za izvesti, no danas su preglednici sposobni za složenije zadatke nego prije i mogu iskoristiti prednosti kompilatorskih jezika. Na ovaj način je umjesto razvoja posebnih biblioteka namijenjenih za web u područjima kao što je računalni vid, moguće iskoristiti postojeće biblioteke s razvijenom dokumentacijom i zadržati skoro izvorne performanse. (Taheri et al., 2018)

4. Web Assembly

4.1. Asm.js

Kao što je navedeno u prošlom poglavlju, činjenica da je JavaScript jedini jezik za sve preglednike te potreba za dovođenjem aplikacija pisanih u drugim jezicima na Web i zadržavanja izvornih performansi jezika u kojima su pisane, predstavljala je motivaciju za uvođenje tehnologije koja će to omogućiti. Radi toga je kreator preglednika Firefox, Mozilla, definirala podskup JavaScripta zvan asm.js, koji je služio kao svojevrsna preteča WebAssemblyju. Asm.js se nije trebao pisati izravno, već se logika mogla napisati u C ili C++ te zatim prevesti u JavaScript u procesu transpilacije (pretvorbe koda iz jednog programskog jezika u drugi). Kada god JavaScript stroj preglednika naiđe na poseban *asm pragma* niz `("use asm";)`, on djeluje kao oznaka koja pregledniku omogućuje korištenje operacija niske razine sustava umjesto „skupljih“ JavaScript operacija.

Osim toga, u kod su uključene naznake koji govore JavaScriptu koju vrstu podataka će varijabla sadržavati. Na primjer, `a | 0` se koristi kao naznaka da će varijabla `a` sadržavati vrijednost 32-bitnog cijelog broja. Bitovna OR operacija s nulom ne mijenja izvornu vrijednost, stoga nema sporednih učinaka prilikom ovakve naznake, a ona u ovome slučaju služi tome kako bi „obećala“ JavaScript stroju da ako se varijabla deklarira kao cjelobrojni broj, nikada neće postati npr. string ili neki drugi tip. To znači da JavaScript stroj ne mora pratiti odnosno ispitivati kod da bi saznao kojeg su tipa varijable, već može jednostavno kompajlirati kod onako kako su varijable deklarirane, što kao posljedicu ima brže izvršavanje koda.

Prikazan je primjer JavaScript funkcije gdje *asm pragma* označava početak asm.js koda, a unutar asm.js funkcije `add` se zbrajaju dva broja i varijablama `a, b` te povratnoj vrijednosti `(a + b)` se dodjeljuje naznaka preko bitovne operacije OR (`| 0`) da je riječ o cjelobrojnim tipovima:

```
function AsmModule() {
  "use asm";
  return {
    add: function(a, b) {
      a = a | 0;
      b = b | 0;
      return (a + b) | 0;
    }
  };
```

```
}
```

Iako je uvođenje asm.js-a postiglo željeno poboljšanje performansi, imalo je nekoliko nedostataka. Osim što ovakve naznake za tipove bitno povećavaju same datoteke, problem je što je i dalje potrebna obrada asm.js datoteka od strane JavaScript stroja, što posebno predstavlja izazov na mobilnim uređajima jer ta obrada usporava učitavanje web stranice i troši bateriju. Tu je i činjenica da je ograničeno uvođenje dodatnih značajki jer bi se za to morao mijenjati sam JavaScript od strane proizvođača preglednika. (Gallant, 2019)

4.2. Karakteristike WebAssemblyja

Ovi nedostaci bili su motivacija za uvođenje WebAssemblyja, otvorenog standarda koji se usredotočio na zadržavanje prednosti asm.js-a uz istodobno umanjivanje nedostataka. 2017. godine su Google, Microsoft, Apple i Mozilla uveli podršku za WebAssembly u svoje preglednike, koji su i najkorišteniji – Firefox, Chrome, Safari i Edge. Kao i asm.js, WebAssembly nije dizajniran da se piše ručno i nije namijenjen čitanju od strane ljudi. Sintaksa je bliža asemblerskom jeziku nego li višem programskom jeziku kako bi bila optimiziranija za prevođenje u strojni kod, te bi za ilustrativne svrhe funkcija za zbrajanje dva broja u tekstualnom formatu mogla izgledati ovako:

```
(module
  (func (export "add") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add)
)
```

Međutim, sav kod preveden u WebAssembly zapravo rezultira *bytecodeom* koji se prikazuje u binarnom formatu umjesto tekstualnom, što smanjuje veličinu datoteke i omogućuje brzo prenošenje i preuzimanje.

Implementacijom JIT kompilacije su proizvođači preglednika postigli značajan napredak u poboljšanju performansi JavaScripta. No kako su varijable u JavaScriptu dinamičke i mogu mijenjati tipove, JavaScript stroj može kompilirati JavaScript kod u strojni kod tek nakon što se kod nekoliko puta prati, jer stroj ne može unaprijed znati koje tipove očekivati. S druge strane, WebAssembly kod je statički tipiziran, što znači da su vrste vrijednosti koje će varijable sadržavati poznate unaprijed, i radi toga se WebAssembly kod može pretvoriti u strojni kod od početka bez potrebe za prethodnim praćenjem. Još jedno poboljšanje koje je uvedeno od prvog izdanja WebAssemblyja je tzv. *streaming* kompilacija,

odnosno proces pretvaranja WebAssembly koda u strojni kod dok se datoteka preuzima i prima od strane preglednika. Ona omogućuje inicijalizaciju WebAssembly modula čim završi preuzimanje, što značajno ubrzava vrijeme pokretanja modula.

Osim ovih nadogradnji što se tiče problema učitavanja i obrade datoteka, uklonjena je i potreba za transpilacijom odnosno prevođenjem koda drugih jezika u JavaScript, što je bilo nužno radi komunikacije s web preglednicima, no problematično zbog toga što JavaScript nije zamišljen kao jezik za kompilaciju. Kako se u ovom slučaju kod prevodi iz drugih jezika u WebAssembly koji kao jezik nije vezan uz JavaScript, mogu se dodavati modifikacije u WebAssembly bez potrebe za modificiranjem samog JavaScripta. Što se tiče programskih jezika koji se mogu prevesti u WebAssembly, najveći fokus je pridan jezicima C, C++ i Rust, no postoji eksperimentalna podrška i za mnoge druge jezike: C#/.NET, Java, Python, Go… Što se tiče TypeScripta, kako je riječ o jeziku koji je tipiziran i već se prevodi u JavaScript, za njega postoji AssemblyScript koji služi kao specijalizirano rješenje za kompilaciju koda bliskog TypeScriptu u WebAssembly. Osim pisanja koda u drugim jezicima, postoji i opcija pisanja WebAssembly koda direktno u tekstualnom formatu te njegovog prevođenja u *bytecode,* premda nije preporučljivo.

Iako je primarno namijenjen izvođenju u webu, WebAssembly može se izvoditi i izvan web okruženja, npr. na poslužiteljima, uređajima Interneta stvari, te mobilnim i desktop aplikacijama, s time da u tim slučajevima neće biti dostupne neke web značajke. Još jedno okruženje koje podržava WebAssembly je NodeJS, te kao što dozvoljava korištenje JavaScript izvan preglednika odnosno na poslužitelju, na sličan način je ondje moguće koristiti i WebAssembly. Kako je prenosivost jedna od karakteristika s kojima je zamišljen WebAssembly, razvijen je *WebAssembly System Interface* (WASI), standard koji osigurava konzistentno funkcioniranje WebAssemblyja u okruženjima izvan preglednika kao što su poslužitelji te pruža modulima mogućnost interakcije s datotečnim sustavom, mrežom, grafikom i drugim API-jima izvan preglednika.

Bitno je naglasiti da je WebAssembly razvijen kao komplement JavaScriptu i da nije zamišljen da ga u potpunosti zamijeni. U situacijama gdje je dovoljan klasični JavaScript nije ni potrebno koristiti WebAssembly, koji je bolje koristiti da preuzme procesorski zahtjevnije zadatke ili u situacijama kada je potrebno koristiti biblioteke pisane u drugim jezicima bez potrebe da se one ponovno pišu u JavaScriptu. Komplementarnost se vidi i po činjenici da WebAssembly za sada ne može izravno komunicirati s web API-jima, nego samo putem JavaScripta. Kod programiranja za web su dvije glavne komponente JavaScript stroj ili VM (virtualna mašina), u kojoj se izvršava WebAssembly modul, te web API kao što je DOM, WebGL, web radnici i slično. WebAssembly modul može komunicirati s JavaScriptom, što

omogućuje razmjenu podataka, pozivanje JavaScript funkcija iz WebAssembly koda i obrnuto. Ova interakcija omogućuje integraciju WebAssembly aplikacija s postojećim web sustavom. Međutim, WebAssembly modul još uvijek nije sposoban izravno komunicirati s bilo kojim od navedenih web API-ja, već može neizravno komunicirati s njima pozivanjem JavaScript funkcija i prepuštanjem njima da izvrše potrebne radnje u ime modula.

Vrijedno je osvrnuti se i na pitanje sigurnosti kao jedne od karakteristika WebAssemblyja, s obzirom da je to prvi jezik koji dijeli JavaScript stroj, koji je izoliran od izvođenja programa te je prošao godine testiranja sigurnosti. WebAssembly moduli nemaju pristup ničemu čemu JavaScript nema pristup i također poštuju iste sigurnosne politike, uključujući primjenu politike istog podrijetla (eng. *same-origin policy*).

Za razliku od desktop aplikacija, WebAssembly modul nema izravan pristup memoriji uređaja. Umjesto toga, izvršno okruženje prilikom inicijalizacije modulu predaje Array-Buffer koji modul koristi kao linearnu memoriju, a WebAssembly okvir provjerava je li kod u granicama polja. Isto tako, modul nema izravan pristup elementima poput pokazivača na funkcije, koji su pohranjeni u odjeljku Table. Kod traži pristup elementu na temelju njegovog indeksa putem WebAssembly okvira, te okvir pristupa memoriji u ime koda. Za razliku od C++-a, u kojem je preko pokazivača moguće mijenjati izvršni stog i tako greškama programera dovesti do problema kao što su curenje memorije, „viseći“ pokazivači i sigurnosni propusti, u WebAssemblyju je stog odvojen od linearne memorije i nije dostupan kodu. Ovakve sigurnosne provjere kao cijenu imaju lagano smanjenje performansi izvornog koda. (Battagline, 2021; Gallant, 2019)

4.3. Generiranje WebAssembly modula

Glavnu jedinicu WebAssembly programa čini modul, te ga web stranica koja koristi WebAssembly prvo mora učitati. Modul je datoteka u binarnom formatu koja sadrži izvršni kod, funkcije, podatke i ostale resurse koje web aplikacija zahtijeva. On se u pravilu ne stvara ručno, već nastaje kao rezultat prevođenja koda pisanog u C, C++ ili drugim jezicima. Proces prevođenja može se izvršiti preko nekoliko alata od kojih je najkorišteniji Emscripten, niz alata (eng. *toolkit*) za prevođenje primarno C ili C++ koda u WebAssembly *bytecode*. Prvobitno je stvoren za transpilaciju C/C++ koda u asm.js, a tijekom razvoja WebAssemblyja odabran je Emscripten jer koristi LLVM (eng. *Low Level Virtual Machine*) prevoditelj s kojim je radna grupa za WebAssembly već imala iskustvo. Naredba za Emscripten je `emcc` ili `em++` (Windows) odnosno `./emcc` ili `./em++` (Linux/macOS), primjerice za stvaranje `.wasm` datoteke (WebAssembly modula) iz `.c` datoteke na Windowsu: `emcc main.c -o index.wasm`.

Kada se `.wasm` datoteka učita u preglednik, preglednik provjerava je li sve ispravno i ako jest, dovršit će prevođenje WebAssembly modula odnosno njegovog *bytecodea* u strojni kod uređaja. Sam modul ima nekoliko predefiniranih sekcija koje će prilikom prevođenja Emscripten popuniti na temelju C/C++ koda kojeg piše programer. Primjerice, stavit će funkciju `main` u `Start` sekciju modula, gdje se referencirana funkcija automatski poziva prije bilo koje druge.

Kao što je ranije spomenuto, WebAssembly modul dobiva memoriju za korištenje od domaćina u obliku ArrayBuffera koji se ponaša kao stog u C ili C++ jeziku, ali svaki put kad modul vrši interakciju s memorijom, WebAssembly okvir provjerava je li zahtjev unutar granica polja kako bi se spriječio preljev. WebAssembly moduli podržavaju samo četiri vrste podataka: 32-bitni cijeli brojevi, 64-bitni cijeli brojevi, 32-bitni brojevi s pomičnim zarezom i 64-bitni brojevi s pomičnim zarezom. Logičke vrijednosti su predstavljene pomoću 32-bitnog cijelog broja, gdje je 0 lažna, a ne-nulta vrijednost je istinita. Sve druge vrijednosti koje postavlja okruženje domaćina, poput znakovnih nizova, moraju se prikazati u linearnoj memoriji modula.

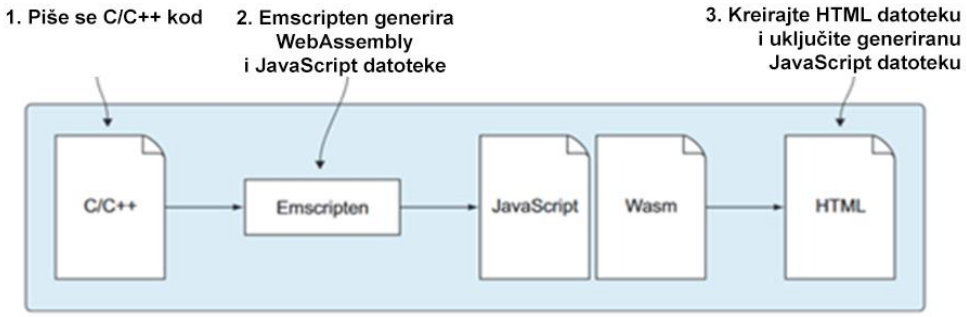
Postoji nekoliko pristupa za generiranje modula preko Emscriptena, ovisno o željenom ishodu. Osim WebAssembly datoteke Emscripten također generira JavaScript datoteku pomoćnih funkcija (eng. *JavaScript plumbing file*), i opcionalno HTML datoteku. JavaScript datoteka pomoćnih funkcija je datoteka koja sadrži kod koji će automatski preuzeti WebAssembly datoteku, prevesti je i instancirati modul u pregledniku, te sadrži pomoćne funkcije za olakšavanje komunikacije između domaćina i modula.

Ako se u naredbi `emcc` ne specificira naziv izlazne `.wasm` odnosno `.js` datoteke preko zastavice `-o`, one se automatski stvaraju pod nazivima `a.out.wasm` i `a.out.js`, inače se obje stvaraju pod istim nazivom koji je definiran za jednu od njih, primjerice, naredba `emcc main.cpp -o index.js` stvara datoteke `index.js` i `index.wasm`. Ako se želi stvoriti i HTML datoteka, ona je jedina koju je potrebno eksplicitno specificirati, primjerice, `emcc main.cpp -o index.html` će stvoriti `index.wasm`, `index.js` i `index.html` datoteke.

Mogu se koristiti tri pristupa za stvaranje modula pomoću Emscriptena. Prvi je da se generiraju WebAssembly modul, JavaScript datoteka pomoćnih funkcija i HTML predložak. Generiranje HTML datoteke od strane Emscriptena nije uobičajeno za produkcijsko okruženje, no omogućuje brzo testiranje C/C++ koda prevedenog u WebAssembly jednostavnim otvaranjem generirane HTML datoteke u pregledniku kako bi se vidjeli rezultati. HTML predložak koji stvara Emscripten usmjerava bilo kakav ispis C/C++ `printf` ili `cout` naredbi iz modula u predefinirano HTML tekstualno polje kako ne bi bilo potrebno otvarati konzolu u razvojnim alatima preglednika, te također uključuje HTML element `canvas` iznad tekstualnog

polja koji omogućuje prikaz WebGL izlaza. Ako se ne kreira HTML predložak na ovaj način, ispis `printf/cout` naredbi se automatski usmjerava u konzolu preglednika.

Drugi pristup je generiranje WebAssembly modula i JavaScript datoteke, što je tipičan pristup za produkcijsko okruženje, jer se generirana JavaScript datoteka može dodati u postojeću HTML stranicu jednostavnim uključivanjem reference na datoteku, a samu stranicu stvara programer. Prilikom učitavanja stranice će uključena JavaScript datoteka učitati i instancirati WebAssembly modul, te je ovo pristup koji će se koristiti prilikom izrade praktičnog dijela rada. I prvi i drugi opisani pristup uključuju sve standardne C biblioteke u modul ako ih C/C++ kod koristi, a alternativno se mogu koristiti zastavice za upute Emscriptenu da uključi one funkcije koje su potrebne.



Slika 2: Generiranje `.js` i `.wasm` datoteka preko Emscriptena te uključivanje `.js` datoteke u postojeću HTML stranicu (Gallant, 2019, str. 35)

Posljednji pristup je da se generira samo WebAssembly modul. Ovaj pristup je namijenjen dinamičkom povezivanju dva ili više modula tijekom izvođenja, a može se koristiti i za stvaranje minimalističkog modula koji ne sadrži podršku standardne C biblioteke ili JavaScript datoteku. U ovom slučaju je potrebno napisati vlastiti JavaScript kod koji se učitati i instancirati WebAssembly modul, što je moguće preko `fetch` API-ja.

Kada se HTML datoteka otvori izravno s lokalnog datotečnog sustava koristeći protokol `file://`, web preglednici je tretiraju drugačije nego datoteke s web URL-ovima (npr. `http://` ili `https://`), pa se primjenjuju strože sigurnosne restrikcije na sadržaj koji se učitava s protokola `file://` i one uključuju ograničenja pristupa vanjskim resursima poput WebAssembly modula. Iz tog razloga je potrebno pokrenuti lokalni server preko kojeg se može pristupiti HTML stranici koja referencira JavaScript datoteku za učitavanje WebAssembly modula. (Battagline, 2021; Gallant, 2019)

4.4. Interakcija WebAssembly modula i JavaScript koda

Kako je pored performansi prednost WebAssemblyja i u ponovnoj iskoristivosti koda, česti slučaj korištenja je prebacivanje postojeće desktop aplikacije pisane u C++ na web, što će se u praktičnom dijelu prikazati na primjeru igre u C++ koja se prebacuje u preglednik. U takvim slučajevima nije potrebno iznova pisati aplikaciju u JavaScriptu već se može zadržati postojeći C++ kod, no potrebno ga je modificirati za prevođenje u WebAssembly modul, te će biti potrebno napisati JavaScript kod koji ima interakciju s njime. Prvi korak je dodavanje zaglavlja za Emscripten u `.cpp` datoteku (`#include <emscripten.h>`) kako bi se mogla prevesti preko Emscriptena.

Kako bi se moglo osigurati da JavaScript kod može pozvati C++ funkcije, potrebno je svaku funkciju za koju želimo da bude dostupna JavaScript kodu staviti u blok `extern "C"`. Razlog tomu je što se kod prevođenja C++ koda zbog preopterećenja funkcija, koje dozvoljava funkcijama da imaju isti naziv ukoliko se razlikuju po potpisu (tipu ili broju argumenata), nazivi funkcija mijenjaju u jedinstvene nazive, što znači da se takve funkcije više neće moći pozvati iz vanjskog koda s obzirom na to da će im nazivi nakon prevođenja biti drugačiji. Blok `extern "C"` osigurava da se nazivi funkcija ne mijenjaju prilikom prevođenja C++ koda. Također se unutar bloka iznad funkcije dodaje deklaracija `EMSCRIPTEN_KEEPALIVE` kako bi se osiguralo da je ta funkcija sačuvana u WebAssembly modulu, odnosno izvezena prema vanjskom kodu, te se ne mora uključivati zastavica `-s EXPORTED_FUNCTIONS` unutar `emcc` naredbe gdje bi inače ručno trebalo navesti imena svih funkcija dostupnih vanjskom kodu. Oba koraka su potrebna kako bi se postigla potpuna interoperabilnost između C++ koda i JavaScripta u WebAssembly okruženju.

Prilikom prevođenja C++ koda preko Emscriptena je potrebno uključiti zastavicu `-s EXPORTED_RUNTIME_METHODS` i navesti funkciju `ccall`, pomoćnu Emscripten funkciju koja će se koristiti prilikom interakcije JavaScript koda s WebAssembly modulom. Preko nje se mogu pozivati C++ funkcije i prenositi argumenti tim funkcijama, te se na taj način pojednostavljuje upravljanje memorijom i konverzija tipova između WebAssemblyja i JavaScripta. Funkciji se pristupa preko objekta `Module`, sučelja za interakciju s WebAssembly modulom kojeg instancira JavaScript datoteka pomoćnih funkcija generirana preko Emscriptena. Prima četiri argumenta: naziv C++ funkcije koju se poziva, povratni tip koji se očekuje iz te funkcije, niz tipova argumenata koji se prenose pozivanoj funkciji, te niz samih argumenata. Primjer korištenja može biti `var rezultat = Module.ccall('moja_funkcija', 'number', ['number'], [9]);` u kojem slučaju se poziva C++ funkcija `moja_funkcija`, prenosi joj se

cjelobrojna vrijednost 9 i očekuje cjelobrojna vrijednost kao rezultat te se ona sprema u JavaScript varijablu `rezultat`.

Što se tiče interakcije modula s JavaScript kodom, jedan način je korištenjem Emscriptenovih makro naredbi kao što su `EM_JS` i `EM_ASM`. Obje naredbe omogućuju ugrađivanje JavaScript koda unutar sebe, a razlika je u tome da se `EM_ASM` koristi za izvršavanje JavaScript koda izravno unutar C/C++ koda bez vraćanja vrijednosti, dok se `EM_JS` koristi za definiranje JavaScript funkcija i pristupanje njihovim povratnim vrijednostima iz C/C++ koda. Alternativni načini interakcije modula s JavaScript kodom su dodavanje vlastitog JavaScript koda u JavaScript datoteku generiranu Emscriptenom koji se može pozvati direktno, ili funkcijski pristup korištenjem funkcija povratnog poziva, odnosno pokazivača preko kojih JavaScript kod prenosi funkciju koju modul treba pozvati. (Battagline, 2021; Gallant, 2019)

5. Opis razvijene web aplikacije koja koristi C++

5.1. Opis aplikacije

Aplikacija razvijena za praktični dio rada je web aplikacija koja omogućuje igranje igre u pregledniku i praćenje rezultata postignutih u igri. Sastoji se od dvije web stranice, jedne za igru (URL: `/game`) i jedne za listu rezultata (URL: `/leaderboard`). Na stranici za igru se nalaze dva HTML elementa `canvas` koji pruža dvostruki pogled na igru s time da je lijevi prikaz u 2D, a desni u 3D. Tematika, odnosno okruženje igre je svemir. Žanr igre je beskonačni trkač (eng. *infinite runner*), što znači da se lik igrača zauvijek kreće kroz prostor igre dok izbjegava prepreke i skuplja bodove, to jest, igra nema uvjet pobjede i završava tek kada igrač izgubi sve živote. U igri igrač dobiva 10 života te gubi život svaki put kada se sudari s neprijateljskim brodovima, a pritom skuplja objekte koji donose bodove čime se povećava rezultat. Igrač također ima opciju pucanja i uništavanja neprijateljskih objekata. Kada igrač dođe na nula života, igra završava te prikazuje bodove i opcije za ponovno igranje i upisivanje korisničkog imena. Web stranica za rezultate prikazuje informacije o korisničkim imenima koje su igrači upisali na kraju igre i postignutim bodovima.

5.1.1. Tehnologije

Za klijentsku stranu korišteni su HTML, CSS i JavaScript, dok su za poslužiteljsku korišteni NodeJS i Express, te SQLite kao baza podataka. Za igru je većina logike izrađena u C++ jeziku što se primarno odnosi na njezin 2D prikaz, za što je korištena biblioteka SDL2 koja omogućuje pristup niske razine multimedijskim mogućnostima poput grafike, tipkovnice, miša i slično. Manju komponentu igre čini 3D prikaz za koje su korištene tehnologije za klijentsku stranu, odnosno HTML, CSS i JavaScript, te JavaScript biblioteka ThreeJS. Integracija logike u C++ jeziku u web aplikaciju je postignuta preko WebAssemblyja, a za prevođenje u WebAssembly format je korišten Emscripten, niz alata za prevođenje koda. Razvojna okolina korištena za projekt je bila Visual Studio Code, a preglednik korišten za testiranje Mozilla Firefox.

5.1.2. Arhitektura

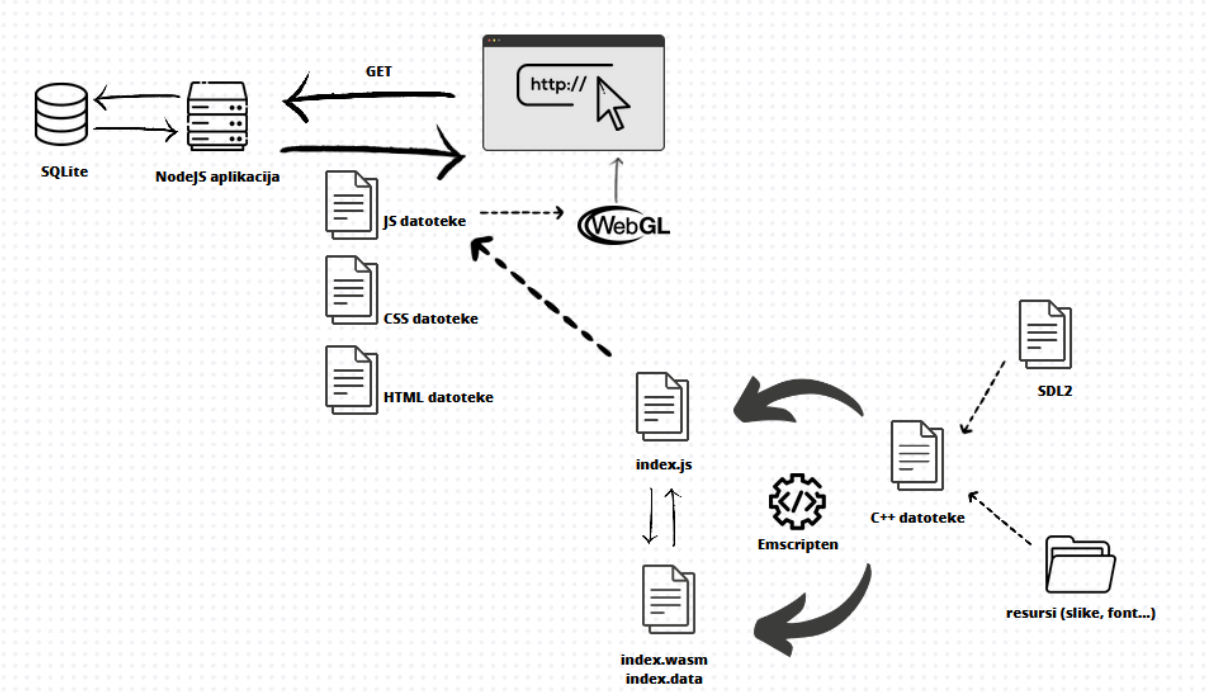
Prevođenje C++ datoteka za igru u WebAssembly putem Emscriptena generira tri datoteke: `index.js`, `index.wasm` i `index.data`. `Index.data` sadrži podatke o resursima koje koristi WebAssembly modul, kao što su slike, fontovi i slično. `Index.wasm` je

WebAssembly modul u binarnom formatu koji sadrži prevedeni strojni kod iz C++ datoteka. `Index.js` sadrži kod koji se ponaša kao sučelje između `index.wasm` i JavaScript koda web aplikacije. Kao takav, `index.js` je ključan za komunikaciju s izvezenim funkcijama i memorijom u WebAssembly modulu te je uključen u HTML stranicu koja sadrži igru.

Sav HTML kod, odnosno sve stranice su dinamički generirane na poslužiteljskoj strani i šalju se kao odgovor na GET zahtjev klijentske strane. Svaka HTML stranica ima pripadnu CSS i JS datoteku koje se također uključuju na strani poslužitelja. Poslužiteljska strana je također odgovorna za kreiranje baze i komunikaciju s njom, te prihvaća GET i POST zahtjeve od klijentske strane za dohvaćanje i dodavanje podataka u bazu. Kako se za bazu koristi SQLite biblioteka uključena unutar NodeJS aplikacije, baza je sadržana unutar jedne datoteke na disku na strani poslužitelja.

Na klijentskoj strani se za dio igre u 3D prikazu koristi ThreeJS, odnosno JavaScript biblioteka koja pruža sučelje za rad s WebGL-om, API-jem niske razine za rad s 3D grafikom temeljenim na OpenGL ES standardu koji omogućuje pristup grafičkoj kartici, čime omogućuje i renderiranje shadera, programa koji se izvode na grafičkoj kartici. Za prikaz 3D pozadine u igri se koristi shader unutar ThreeJS scene unutar HTML elementa `canvas`.

Struktura projekta sastoji se od datoteke `server.js` koja sadrži poslužiteljsku logiku, datoteke za bazu podataka `game_data.db`, direktorija `node_modules` i direktorija `public`, u kojemu su sadržane sve druge datoteke i direktoriji. To su datoteka `main.cpp` koja služi kao ulazna točka za postavljanje C++ logike tijekom prevođenja preko Emscriptena, zatim datoteke generirane tijekom prevođenja od strane Emscriptena `index.js`, `index.wasm` i `index.data`, te direktoriji `src` (sadrži direktorije `include` s datotekama zaglavlja za uključivanje i `lib` s bibliotekama za povezivanje te mu je svrha omogućiti korištenje biblioteka poput SDL2 u C++ kodu igre), `assets` (svi resursi za C++ kod i klijentsku stranu – *favicon*, direktoriji `fonts` za fontove, `images` za slike te `models` za `.obj` datoteke za 3D prikaz), `game` (sve C++ datoteke i datoteke zaglavlja za logiku igre koje su korištene u datoteci `main.cpp`), `html` (HTML datoteke), `css` (CSS datoteke), `js` (JS datoteke) i `shaders` (`.glsl` datoteke za shader program).



Slika 3: Pregled arhitekture i tehnologija izrađene aplikacije

5.1.3. Izgradnja aplikacije

Postupak izgradnje aplikacije uključuje prevođenje C++ koda preko Emscriptena u WebAssembly modul, te se može pokrenuti `server.js` skripta preko koje Express poslužuje aplikaciju. Kako se koristi lokalni poslužitelj na portu 3000, URL za pristup je http://localhost:3000/. S obzirom da projekt sadrži već generirane `index.wasm`, `index.js` i `index.data` datoteke, nije potrebno prevođenje Emscriptenom, već se može odmah pristupiti aplikaciji jednom kad je poslužitelj pokrenut. Prevođenje je potrebno jedino u slučaju kad se rade izmjene u `.cpp` datotekama.

Jednom kada je Emscripten instaliran, na Windowsu se njegovom sučelju pristupa preko naredbenog retka Emscripten Command Prompt, koji je unaprijed konfiguriran s ispravnim putanjama sustava i postavkama za Emscripten alate. Na macOS-u i Linuxu je dovoljno otvoriti običan naredbeni redak. Za prevođenje C++ datoteka potrebno je pozicionirati se u direktorij projekta koji sadrži `main.cpp`, ili u naredbi specificirati punu putanju do `main.cpp` datoteke ako ona nije u trenutnom direktoriju.

Puna naredba korištena za prevođenje C++ koda ove aplikacije u WebAssembly modul preko Emscriptena: `emcc main.cpp game/renderer.cpp game/helper.cpp game/collidable.cpp game/enemy.cpp game/projectile.cpp -o index.js -s "EXPORTED_RUNTIME_METHODS=['ccall']" -s WASM=1 -s USE_SDL=2 -s USE_SDL_TTF=2`

`-s USE_SDL_IMAGE=2 -s SDL2_IMAGE_FORMATS=['jpg','png'] --preload-file assets/images --preload-file assets/fonts -s INITIAL_MEMORY=33554432`

Naredba prevodi sve C++ datoteke potrebne za logiku igre (`main.cpp`, `renderer.cpp`, `helper.cpp`, `collidable.cpp`, `enemy.cpp`, `projectile.cpp`). Zastavicom `-s` se postavljaju različite opcije, kao što je postavljanje izvođenja `ccall` metode koja omogućuje pozivanje C++ funkcija iz JavaScript koda, pomoću `-s "EXPORTED_RUNTIME_METHODS=['ccall']"`. Opcija `-s WASM=1` specificira WebAssembly (umjesto npr. ASM.js-a). Opcije `-s USE_SDL=2, -s USE_SDL_TTF=2, -s USE_SDL_IMAGE=2` omogućavaju korištenje SDL2 biblioteke te dodataka za korištenje fontova i slika. Opcija `-s SDL2_IMAGE_FORMATS=['jpg','png']` postavlja podržane formate za SDL slike. Opcija `-s INITIAL_MEMORY=33554432` postavlja početnu veličinu memorije za program u bajtovima, budući da zbog veličine korištenih resursa poput slika, količina memorije koja se dodjeljuje prema zadanim postavkama nije dovoljna za učitavanje WebAssembly modula u preglednik. Također se dodaju datoteke iz direktorija `assets/images` i `assets/fonts` preko opcije `--preload-file` kako bi bile dostupne tijekom izvođenja igre u pregledniku.

Naredbom `node server.js` se iz korijenskog direktorija projekta za pokretanje skripte `server.js` podiže Express poslužitelj koji poslužuje aplikaciju lokalno na portu 3000 i omogućuje pristup web stranicama.

5.2. Logika igre u C++ jeziku

Ovo poglavlje obrađuje logiku web aplikacije koja je pisana u C++ jeziku, odnosno logiku 2D dijela igre, koja djelomično upravlja i logikom 3D dijela igre, primjerice u sinkronizaciji kretanja lika igrača između 2D i 3D prikaza. Ulaz za C++ kod čini datoteka `main.cpp`, a sve druge C++ datoteke potrebne za igru se nalaze u direktoriju `public/game`, gdje za svaku klasu ili imenski prostor postoje dvije datoteke, datoteka zaglavlja `.h`, koja se po potrebi uključuje u druge datoteke, te pripadna `.cpp` datoteka koja definira funkcije deklarirane u datoteci zaglavlja. Iznimka je datoteka `context.h` koja sadrži definiciju strukture i prema tome samo definira podatke, ali ne i funkcije, pa nema svoju `.cpp` datoteku.

5.2.1. Main.cpp

5.2.1.1. Funkcija main

Prva funkcija koja se poziva je `main` unutar datoteke `main.cpp`, unutar koje se kreira SDL prozor. Varijable koje trebaju biti dostupne kroz sve C++ funkcije i datoteke se pohranjuju

unutar objekta strukture `struct Context` definiranoj u datoteci zaglavlja `context.h` te koja u svakom trenutku sadrži informacije o stanju i postavkama igre. Radi toga se i kreirani SDL prozor pohranjuje u objekt `ctx`, koji se predaje kao parametar svim funkcijama koje čitaju ili mijenjaju podatke o igri.

```
    Context ctx;
    SDL_Init(SDL_INIT_VIDEO);
    ctx.window = SDL_CreateWindow("Space Runner", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, ctx.WINDOW_WIDTH, ctx.WINDOW_HEIGHT, 0);
    ctx.renderer = SDL_CreateRenderer(ctx.window, -1,
SDL_RENDERER_ACCELERATED);
```

SDL prozor će biti prikazan unutar HTML elementa `canvas` na stranici igre zahvaljujući Emscriptenovoj integraciji SDL2 i WebGL-a. Unutar JavaScript koda stranice za igru `play.js` je definirana varijabla `Module.canvas` kojoj se dodjeljuje željeni `canvas` element sa stranice `play.html`. Funkcija `main_loop` iz `main.cpp`-a je petlja igre u kojoj se izvodi kod za renderiranje grafike igre preko SDL2 renderera, te u njoj na kraju svake iteracije pozivom funkcije `SDL_RenderPresent(ctx->renderer)` SDL2 stavlja novu sličicu (eng. *frame*). Kako je Emscripten svjestan varijable `Module.canvas`, renderiranje se u pozadini rješava upućivanjem izlaza SDL2 renderera na specificirani `canvas` element.

Prije pokretanja glavne petlje igre `main_loop`, u funkciji `main` se još vrši učitavanje korištenih slika i fontova, pri čemu se za slike poziva funkcija definirana u imenskom prostoru `Helper` unutar datoteke `helper.cpp` te uključena u `main.cpp` preko datoteke zaglavlja `helper.h`.

```
    ctx.background_image =
Helper::load_image("./assets/images/background.jpg", &ctx);
```

Nakon učitavanja potrebnih resursa iz direktorija `assets`, pozivom funkcije `init_collidables` se proceduralno generiraju objekti klase `Collidable`, koji predstavljaju bodove koje igrač može sakupljati, i klase `Enemy`, koja nasljeđuje klasu `Collidable` radi zadržavanja sposobnosti kolizije s igračem, te koja predstavlja neprijatelje s kojima igrač može doći u koliziju i tako izgubiti život.

```
    init_collidables(&ctx);
```

Objekti se dodaju u polje `collidables` koje je pohranjeno unutar objekta `ctx` i u `main_loopu` će se ažurirati, zajedno s likom igrača i svime drugim što se mijenja od sličice do sličice.

```
    void init_collidables(Context *ctx) {
      for (int i = 0; i < ctx->NUMBER_COLLIDABLES; i++) {
```

```
        Collidable *coin = new Collidable(/* Podaci – dimenzije, itd. */);
        ctx->collidables.push_back(coin);
        Enemy *enemy = new Enemy(/* Podaci – dimenzije, itd. */);
        ctx->collidables.push_back(enemy);
      }
    }
```

Osim inicijalizacije objekata u igri, u funkciji `main` se postavlja slušač na događaj pritiska tipki kojima se kontrolira kretanje lika igrača. Kako se za postavljanje slušača događaja `keydown` koristi JavaScript kod, koristi se makronaredba `EM_ASM` te će se na pritisak tipke preko naredbe `Module.ccall` omogućiti iz JavaScripta pozivanje C++ funkcije koja kontrolira lik igrača. Dodatno se definira varijabla `Module.context` kojoj se predaje objekt `ctx`, kako bi JavaScript skripte mogle po potrebi pristupati informacijama o igri, npr. njezinom početku ili završetku te po tome ažurirati izgled web stranice.

```
    EM_ASM(
      Module.handleEvent = function(event) {
            Module.ccall('handle_input', 'void', ['int', 'int', 'number'],
      [event.keyCode, 1, $0]);
      };
      document.addEventListener('keydown', Module.handleEvent);
      Module.context = $0;
    , &ctx);
```

Nakon što je riješena sva potrebna inicijalizacija za logiku igre, poziva se funkcija `emscripten_set_main_loop_arg` koja je dio Emscriptenovih alata za razvoj softvera, odnosno SDK-a (eng. *Software Development Kit*). Specifično je zamišljena za upravljanje glavnih petlji aplikacija prevedenih u WebAssembly ili asm.js preko Emscriptena. Beskonačna petlja u C++ kodu koja je potrebna svakoj igri, problematična je u web okruženju s obzirom da kontrola nikada nije vraćena pregledniku te se stvaraju problemi kod sinkronizacije s preglednikom, kako su renderiranje u pregledniku i izvođenje JavaScript koda odvojeni od izvođenja WebAssembly koda, kao i s učinkovitošću (visoka upotreba procesora). Radi toga `emscripten_set_main_loop_arg` sinkronizira i optimizira interakcije između glavne petlje C++ koda odnosno igre, i asinkronog izvođenja JavaScript operacija iz web okoline. U stvarnosti se ne ponaša kao prava beskonačna petlja, nego poziva petlju s frekvencijom željenog broja sličica u sekundi te na kraju svake iteracije vraća kontrolu pregledniku za renderiranje trenutnog prikaza stranica, upravljanje događajima i druge potrebne zadatke.

```
    emscripten_set_main_loop_arg(main_loop, &ctx, -1, 1);
```

Funkciji se kao argumenti predaju pokazivač na glavnu petlju `main_loop`, pokazivač na podatke koji se žele prenijeti glavnoj petlji kod svake interakcije (u ovom slučaju to je objekt strukture `Context`, `ctx`, te se njegovom uporabom u ovoj liniji osigurava da svaka iteracija petlje igre ima pristup podacima potrebnim za crtanje trenutnih pozicija objekata, igrača, pozadine i sl.), željeni broj sličica u sekundi po kojem bi se glavna petlja trebala izvoditi ili -1 u slučaju da prepuštamo pregledniku da sam optimizira ovaj broj, te istinita vrijednost 1 za simulaciju beskonačne petlje. Time se osigurava da Emscripten kontinuirano poziva funkciju `main_loop` koja renderira svaku sličicu igre te da brine o tome da je `canvas` stalno ažuriran sadržajem renderiranim preko SDL2 biblioteke, bez ometanja osvježavanja samog preglednika.

5.2.1.2. Petlja igre main_loop

Unutar glavne petlje se prvo dohvaća preneseni kontekst kako bi se moglo pristupati trenutnim informacijama o igri, te se iscrtava početna slika igre preko cijelog SDL prozora. `SDL_Rect` služi za definiranje površine za iscrtavanje gdje nule predstavljaju X i Y koordinatu tj. lijevi gornji kut prozora, a druga dva argumenta su širina i visina površine koja je u ovom slučaju jednaka dimenzijama prozora. `SDL_RenderCopy` prima renderer, sliku, površinu slike koja se iscrtava (u ovom slučaju NULL pokazivač s obzirom da se iscrtava cijela slika) i odredišnu površinu na koju se iscrtava. S obzirom na ograničene mogućnosti SDL2-a kod interakcije s događajima, prikaz START gumba i upravljanje klikom na njega za početak igre je prepušteno JavaScript skripti stranice za igru, `play.js`.

```
  void main_loop(void *arg) {
    Context *ctx = static_cast<Context*>(arg);
    SDL_Rect dest_rect = {0, 0, static_cast<int>(ctx->WINDOW_WIDTH), static_cast<int>(ctx->WINDOW_HEIGHT)};
    SDL_RenderCopy(ctx->renderer, ctx->images[ctx->start_image], nullptr, &dest_rect);
```

Kada je igra u stanju GAMEPLAY, što je vrijednost definirana unutar enumeracije `GameState` u strukturi `Context`, iscrtava se trenutna sličica (pozadina igre, igrač, objekti) i vrši se detekcija sudara igrača s objektima. Iscrtavanje dobiva učinak tek pozivom `SDL_RenderPresent`, a nakon toga se ažuriraju brojčane vrijednosti pozicija za sliku pozadine, kako bi se postigao *looping* efekt i time iluzija da se igrač kreće kroz beskonačni prostor čak i kada stoji na mjestu, te vrijednosti za objekte `collidables` (objekte za sakupljanje i neprijatelje) za upravljanje njihovih kretanja kroz igru. Ta ažuriranja će dobiti učinak u idućoj iteraciji petlje, odnosno s novim pozivom `SDL_RenderPresent`.

```
    if (ctx->game_state == Context::GameState::GAMEPLAY) {
```

```
        render_frame(ctx);
        handle_collisions(ctx);
    }
    SDL_RenderPresent(ctx->renderer);
    update_background_offset(ctx);
    update_collidables(ctx);
```

Nakon iscrtavanja i ažuriranja pozicija se poziva funkcija `monitor_game_over_status`, koja pozivom funkcije iz imenskog prostora `Helper` (datoteke `helper.h` i `helper.cpp`) provjerava je li igra završila i ako jest, makro naredbom `EM_ASM` poziva JavaScript funkcije koje upravljaju prikazom ekrana za kraj igre, koji sadrži polje za upis korisničkog imena i gumbe za ponovno pokretanje igre i slanje korisničkog imena. Isto tako, preko makro naredbe se vrši spremanje postignutog rezultata u igri za koji će na strani JavaScript skripte biti poslan POST zahtjev na poslužitelj radi spremanja u bazu, što će biti prikazano u poglavlju 5.3. koje obrađuje logiku aplikacije orijentiranu na web okruženje.

5.2.2. Renderiranje u C++ kodu

5.2.2.1. Imenski prostor Renderer

Funkcija `render_frame` koja se poziva u svakoj iteraciji glavne petlje za vrijeme trajanja igre u biti obavlja pozive funkcija iz vlastitog definiranog imenskog prostora Renderer koje koriste SDL2 biblioteku za iscrtavanje trenutnog izgleda pozadine, trenutnih pozicija igrača i objekata, te trenutnog rezultata, odnosno dijela ekrana za kraj igre (za cijeli prikaz ekrana nakon završetka igre se odgovornost dijeli između C++ funkcije `Renderer::draw_game_over` i JavaScript skripte `play.js`).

```
void render_frame(Context *ctx) {
  Renderer::draw_background(ctx);
  Renderer::draw_player(ctx);
  Renderer::draw_collidables(ctx);
  if (Helper::is_game_over(ctx))
      Renderer::draw_game_over(ctx);
  else
      Renderer::draw_score(ctx);
}
```

Sve funkcije iz imenskog prostora Renderer deklarirane su u datoteci `renderer.h`, koja je uključena u main.cpp. Definicije, odnosno implementacije funkcija nalaze se u datoteci `renderer.cpp`.

5.2.2.2. Crtanje pozadine igre

Pozadinu 2D prikaza igre čini slika svemira `background.jpg`, koja se poput drugih resursa učitava iz direktorija `assets` prilikom inicijalizacije igre u funkciji `main`. Kako je žanr igre beskonačni trkač, pozadina se ostvaruje tako da se počinje iscrtavati slika od lijevog ruba i pomiče sliku prema lijevo sve dok ne dođe do desnog ruba slike, te se proces ponavlja. Tako se postiže dojam kao da se lik igrača cijelo vrijeme kreće slijeva prema desno, čak i kada korisnik ne kontrolira kretanje lika putem tipki. Slika je modificirana tako da je njezin desni dio identičan lijevom dijelu slike unutar onih dimenzija koje odgovaraju širini SDL prozora, pa se na taj način postiže privid da nema prekida kada tijekom renderiranja slika dosegne svoj desni rub i počne se ponovno iscrtavati od lijevog ruba, što rezultira efektom kao da se igrač kreće kontinuiranim prostorom.

U funkciji `Renderer::draw_background` se zaustavlja kretanje pozadine u slučaju da je igra gotova, postavljanjem varijable `scroll_speed`, kojom se kontrolira brzina pomicanja pozadine, na nulu.

```
  if (Helper::is_game_over(ctx)) ctx->scroll_speed = 0.0f;
```

Potom se već viđenom funkcijom `SDL_RenderCopy` iscrtava slika, bez obzira kreće li se pozadina ili ne. X koordinata površine na koju se slika iscrtava je postavljena na vrijednost:

```
-static_cast<int>(ctx->background_offset) % ctx->background_image_width
```

Što znači da se pozadina „pomiče“ prema lijevo kako X koordinata, koja predstavlja gornji lijevi kut od kojeg počinje iscrtavanje slike, sve više povećava svoju negativnu vrijednost sve dok se ponovno ne postavi na 0, što se događa kada `background_offset` dosegne vrijednost širine slike te ostatak u tom slučaju daje nulu.

Ažuriranje samog kretanja pozadine kontrolira se pozivom funkcije `update_background_offset` u petlji igre, gdje se povećava `background_offset` za vrijednost varijable `scroll_speed`. Ako `background_offset` premaši širinu pozadinske slike umanjenu za širinu prozora, odnosno zadnji dio slike koji se može iscrtati bez da se prođe desni rub slike i počne iscrtavati nepostojeći prostor, vraća se na nulu tj. na lijevi rub ili početak slike.

```
  ctx->background_offset += ctx->scroll_speed;
  if (ctx->background_offset >= ctx->background_image_width - ctx->
WINDOW_WIDTH) ctx->background_offset = 0.0f;
```

5.2.2.3. Crtanje ostalih objekata

Što se tiče objekata za sakupljanje i neprijatelje, kako se pozadina „pomiče“ od lijeva prema desno, inicijalno im se pozicija postavlja pored desnog ruba ekrana gdje još nisu vidljivi unutar SDL prozora i pomiču se prema lijevom rubu prozora za stvaranje efekta kao da se kreću ususret igraču, prisiljavajući igrača da ih ili pokupi ili izbjegne. S obzirom da `Renderer::draw_collidables(ctx)` samo poziva metodu `draw` za svaki `Collidable` objekt, njihovo iscrtavanje će se obraditi u poglavlju 5.2.3. koje predstavlja tu klasu.

```
for (Collidable *collidable : ctx->collidables) collidable->draw(ctx);
```

Funkcije `Renderer::draw_game_over` i `Renderer::draw_score` obje koriste funkciju `Renderer::draw_text`, u kojoj se pozivaju SDL metode za iscrtavanje željenog teksta, u ovim slučajevima tekst za kraj igre, odnosno za trenutni rezultat i broj preostalih života u gornjem lijevom kutu prozora tijekom trajanja igre.

Funkcija `Renderer::draw_player` za iscrtavanje igračevog lika koristi trenutnu poziciju lika iz varijable `ctx->player_position` s X i Y koordinatama, a ova varijabla se ažurira svaki put prilikom pritiska WASD tipki ili strelica, kako je u funkciji `main` postavljeno da se na svaki `keydown` događaj pozove C++ funkcija `handle_input`. Kod ažuriranja pozicija igrača ovisno o smjeru kretanja, pomoću funkcija `min` i `max` se ograničava kretanje igrača na granice prozora, primjerice, pritiskom na tipku W ili gornju strelicu se omogućuje kretanje samo do gornje granice SDL prozora:

```
ctx->player_position.y = max(ctx->player_position.y - ctx->player_speed,
    ctx->player_size / 2);
```

Osim pozicije lika igrača, `Renderer::draw_player` također mijenja boju lika igrača ovisno o modelu broda odabranom u 3D prikazu igre, koji je pod kontrolom JavaScript koda klijentske strane aplikacije.

5.2.3. Klase objekata u igri i detekcija kolizije

Objekti u igri koji pored samog igrača imaju mogućnost kolizije, bilo da su to objekti koje treba pokupiti za bodove, neprijatelji s kojima treba izbjeći koliziju ili projektili koji mogu uništiti neprijatelje, objekti su klase Collidable ili jedne od klasa koje nasljeđuju od nje. Klase Enemy i Projectile obje nasljeđuju klasu Collidable i u njima se nadjačavanjem metoda iz bazne klase mogu iscrtavati neprijatelji ili projektili na specifičan način ili im se dati druge mogućnosti. Za objekte koji se skupljaju za bodove i koji nemaju druge specifičnosti, dovoljna je klasa Collidable. Sučelja svih klasa definirana su u datotekama zaglavlja, a implementacije metoda u istoimenim `.cpp` datotekama.

U datoteci `collidable.h`, javni članovi klase Collidable su `x` i `y` koji predstavljaju 2D koordinate gornjeg lijevog kuta objekta, `width` i `height` za dimenzije objekta, te `collided` koja označava da li je sudar bio već detektiran, što će igrati ulogu u sprečavanju beskonačne detekcije sudara tijekom perioda dok su lik igrača i Collidable objekt u kontaktu. Također, klasa ima konstruktor koji inicijalizira te varijable, metodu `reset_position` koja vraća poziciju objekta na nasumičnu poziciju desno od desnog ruba prozora, metodu `update_position` za ažuriranje vrijednosti pozicije, te virtualne metode `draw` za renderiranje objekta, i `handle_collision` za određivanje toga što se događa kod sudara s likom igrača.

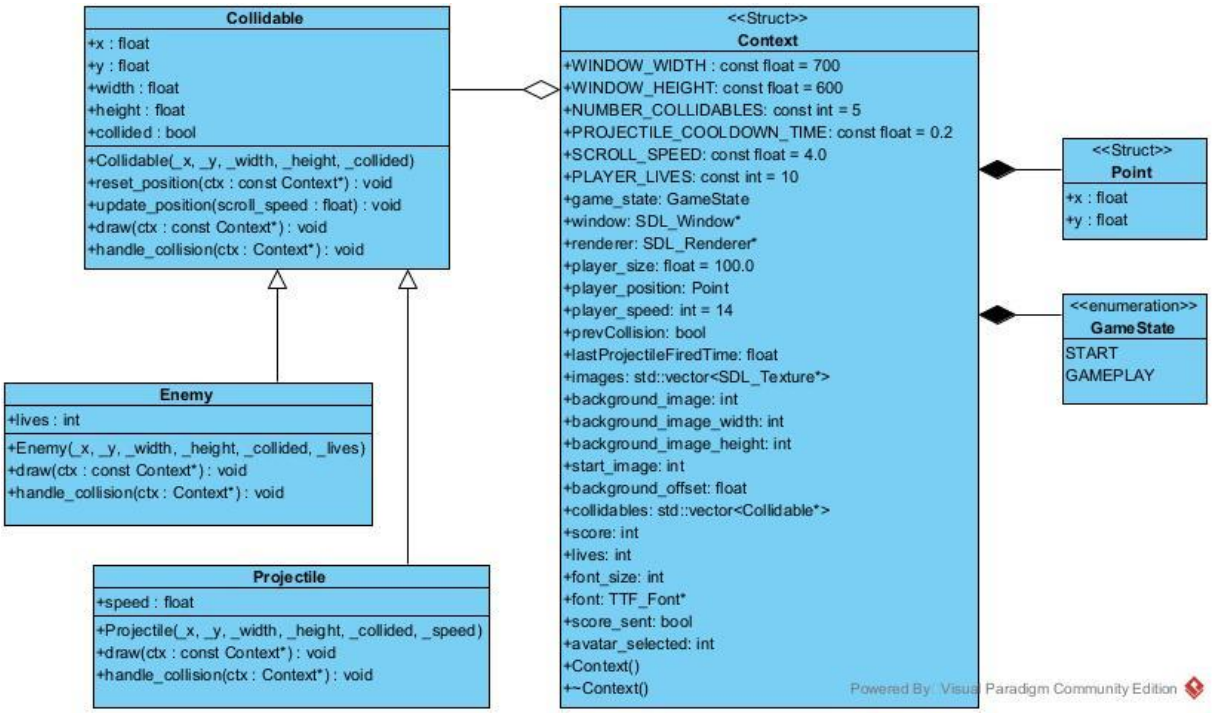
```
class Collidable {
 public:
   float x, y, width, height;
   bool collided;
   Collidable(float _x, float _y, float _width, float _height, bool
_collided);
   void reset_position(const Context *ctx);
   void update_position(float scroll_speed);
   virtual void draw(const Context *ctx);
   virtual void handle_collision(Context *ctx);
 };
```

Klasa Enemy nasljeđuje Collidable, a time i ove atribute i metode. Atributima jedino dodaje broj života koji neprijateljski objekt može imati, a koji se gubi svaki puta kada ga pogodi igračev projektil. Od metoda, nadjačava naslijeđene metode `draw` i `handle_collision`, kako su iscrtavanje neprijatelja i učinak kod sudara s igračem drugačiji nego kod objekata koji se skupljaju za bodove. Radi toga se tim metodama dodaje ključna riječ `override`.

```
 class Enemy : public Collidable {
   public:
       int lives;
       Enemy(float _x, float _y, float _width, float _height, float
_collided, int _lives);
       virtual void draw(const Context *ctx) override;
       virtual void handle_collision(Context *ctx) override;
 };
```

Klasa Projectile ima sličnu definiciju kao klasa Enemy, jedino umjesto atributa `lives` dodaje atribut `speed` za određivanje brzine projektila, i također nadjačava ove dvije metode vlastitom implementacijom. Na Slici 4 je prikazan odnos između ovih klasa preko nasljeđivanja, kao i njihova povezanost sa strukturom `Context` preko veze agregacije, s obzirom da `Context` sadrži vektor Collidable objekata, `std::vector<Collidable *> collidables`. Dodatno su

prikazane i struktura `Point` te enumeracija `GameState` povezane sa strukturom `Context` preko veze kompozicije, kako su i `Point` i `GameState` definirani u njoj te ne mogu postojati ako ne postoji niti `Context`.



Slika 4: UML dijagram klasa s prikazom C++ klasa, struktura i enumeracije za logiku igre

Unutar glavne petlje igre `main_loop` se pozivaju funkcije `handle_collisions` i `update_collidables`, te obje iteriraju kroz `ctx->collidables`, koja je u objektu `ctx` definirana kao `std::vector<Collidable *> collidables`.

```
for (Collidable *collidable : ctx->collidables)
    collidable->handle_collision(ctx);
```

U poglavlju 5.2.1.1. prikazana je `init_collidables` funkcija, gdje se u vektor `collidables` dodaju pokazivači na stvorene objekte, bez obzira jesu li objekti klase Collidable ili naslijeđene klase Enemy. Isto tako, prilikom pritiska na tipku za razmak (Spacebar) igrač može pucati na neprijateljske objekte, te u tom slučaju funkcija `handle_input` (koja se poziva iz JavaScript koda na svaki `keydown` događaj, kao što je ranije prikazano, te upravlja i kretanjem lika igrača) poziva `init_projectiles(ctx)` gdje se stvoreni objekti klase Projectile također dodaju u vektor `collidables`.

Svaki element u vektoru `collidables` je pokazivač na objekt. Kada se pozove `collidable->handle_collision(ctx)`, pozvat će se pravilna metoda zahvaljujući mehanizmu kasnog povezivanja, odnosno polimorfizmu, pri čemu se „pravilna metoda“ odnosi ili na implementaciju iz bazne klase Collidable ili na nadjačanu verziju iz klasa Enemy ili

Projectile, ovisno o stvarnom tipu objekta na koji pokazuje svaki pokazivač. Na ovaj način svaki objekt „zna“ crtati sam sebe i proizvesti odgovarajući efekt pri sudaru s igračem. Taj efekt će u slučaju Collidable objekta biti da samo nestane i poveća bodove igrača za jedan jer se smatra „pokupljenim“. U slučaju sudara igrača s Enemy objektom, objekt neće nestati i oduzet će jedan život igraču.

U slučaju Projectile objekta se ne provjerava sudar s igračem već sa svim objektima tipa Enemy, kako bi se oduzeo život tom neprijateljskom objektu, odnosno uklonio neprijateljski objekt u slučaju da mu kod sudara s Projectile objektom broj života padne na nulu. Isto tako se pri svakom sudaru s neprijateljem i Projectile objekt uklanja iz vektora `collidables` (čime se više niti ne renderira u prozoru), inače se uklanja tek kada prođe desni rub prozora, u kojem slučaju više nema potrebe da se renderira, odnosno drži u memoriji.

U svakom slučaju, zahvaljujući polimorfizmu se svi objekti mogu držati unutar jedne podatkovne strukture unutar strukture `Context` te se u `for` petlji prikazanoj iznad, različite implementacije metode pozivaju za svaki objekt bez potrebe da se ručno provjerava njegov tip. Ista petlja se koristi i u funkciji `update_collidables` koja u svakoj iteraciji glavne petlje igre `main_loop` ažurira poziciju objekta za njegovo iscrtavanje, samo što je metoda koja se poziva ovaj puta `update_position` koja je ista za sve objekte bez obzira na njihov stvarni tip.

```
for (Collidable *collidable : ctx->collidables) {
     collidable->update_position(ctx->scroll_speed);
     if (Helper::is_outside_window_bounds(collidable))
         collidable->reset_position(ctx);
 }
```

Što se tiče „nestajanja“ objekata, u ovom isječku koda se može uočiti da se u biti njihova pozicija samo resetira, što je slučaj ne samo kad se nađu izvan granica prozora, već i kod rješavanja sudara gdje neki objekt nestaje, npr. objekt za sakupljanje u sudaru s igračem. Umjesto da se briše iz vektora `collidables` i time iz renderiranja, takav objekt se samo ponovno premješta pokraj desnog ruba prozora s nasumičnim vrijednostima te „ulazi“ u prozor, kako se funkcija `update_position` i dalje primjenjuje na njega jer je još uvijek dio vektora `collidables`. Tako je zapravo cijelo vrijeme broj objekata za skupljanje i neprijatelje u vektoru konstantan tj. isti broj objekata koji se kreirao prilikom inicijalizacije, i uvijek se radi s istim objektima. Iznimka su objekti klase Projectile koji se zaista brišu iz vektora, pošto njihovo kreiranje ovisi o odluci igrača i ne može se predvidjeti. Alternativni pristup bi bio dinamičko dodavanje novih i uklanjanje starih objekata iz vektora za sve objekte, pogotovo ako se želi povećavati ili smanjivati gustoća neprijatelja ili objekata za sakupljanje između različitih sličica.

Osim navedenih klasa i imenskih prostora, vrijedi spomenuti još i imenski prostor Helper u kojem su grupirane razne vlastite pomoćne funkcije poput `load_image` za učitavanje slika iz direktorija `assets`, Booleovih funkcija provjere za kraj igre `is_game_over` ili ranije viđene `is_outside_window_bounds`, funkcije `check_collision` za matematički izračun jesu li dva objekta koji se predaju kao argumenti u doticaju tj. sudaru, i slično.

5.3. Web aplikacija

5.3.1. Poslužiteljska strana i baza podataka

Skripta `server.js` čini logiku poslužiteljske strane aplikacije, u kojoj se kreira baza podataka te definiraju rute za posluživanje HTML stranica i API zahtjeva. Ona omogućuje postavljanje Express poslužitelja, spremanje korisničkih imena i rezultata igrača u bazu podataka te dinamičko generiranje HTML stranica za prikazivanje sadržaja igre i ljestvice najboljih rezultata. Stranica s ljestvicom rezultata sadrži tablicu za koju se straničenje obavlja na poslužitelju, odnosno umjesto da se pri prvom učitavanju stranice dohvaćaju svi rezultati iz baze i potom straničenje obavlja na klijentu, za svaki klik na drugu stranicu se šalje novi zahtjev za rezultate te stranice tablice prema poslužitelju.

Prvo se učitavaju potrebni moduli `express`, `sqlite3` i `fs/promises` za kreiranje Express poslužitelja te za rad s bazom i datotekama.

```
const express = require("express");
const sqlite3 = require("sqlite3");
const fs = require("fs/promises");
```

Stvara se nova Express aplikacija i postavlja se na port 3000 za posluživanje HTTP zahtjeva. Omogućuje se posluživanje statičkih datoteka iz direktorija `public`, što znači da će datoteke koje se nalaze unutar tog direktorija biti dostupne klijentima tj. web preglednicima preko HTTP zahtjeva. Ovo je korišteno za posluživanje HTML, CSS i JavaScript datoteka, `.glsl` datoteka za shadere, `.obj` datoteka za 3D modele, kao i resursa iz direktorija `assets` poput fonta, favicona i slično.

```
const app = express();
const port = 3000;
app.use(express.static("public"));
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
```

Preko Express *middleware* funkcije `urlencoded` je opcija `extended` postavljena na `true` kako bi omogućila obradu složenijih podataka poput nizova i objekata. Funkcija `json` je još jedna ugrađena *middleware* funkcija koja analizira zahtjeve s tijelom u JSON formatu. Obje je funkcije potrebno koristiti radi pravilnog analiziranja POST zahtjeva.

Nakon što je konfiguriran Express poslužitelj, kreira se baza i definiraju rute na koje dolaze API zahtjevi. Na kraju skripte se pokreće poslužitelj i čeka zahtjeve na navedenom portu.

```
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

5.3.1.1. Baza podataka

Na početku skripte se linijom `const db = new sqlite3.Database("game_data.db")` kreira nova instanca klase `Database` iz modula `sqlite` i povezuje se s datotekom baze podataka kojoj se dodjeljuje ime `game_data.db`. Ako datoteka ne postoji, stvorit će se, a ako postoji, uspostavit će se veza s tom postojećom bazom. Potom se SQLite naredbi `db.run` predaje SQL upit za stvaranje tablice `players` (ako ona već ne postoji) koja sadrži stupce `id` (`INTEGER`), `username` (`TEXT`) i `score` (`INTEGER`) koji predstavljaju identifikator igrača, korisničko ime koje igrač unosi na kraju odigrane igre i rezultat postignut u odigranoj igri.

Na neke API zahtjeve se izvršavaju funkcije koje komuniciraju s bazom, a to su `async` funkcije `getPlayers`, `getPlayersPaginated` i `savePlayer`. Prve dvije funkcije koriste SQLite naredbu `db.all` za SELECT upit koji u slučaju `getPlayers` vraća sve igrače iz tablice, a u slučaju `getPlayersPaginated` samo one igrače koje trenutno prikazuje odabrana stranica tablice na klijentskoj strani. Funkcija `savePlayer` prima korisničko ime igrača i postignuti rezultat kao parametre i sprema ih u tablicu igrača zajedno s automatski generiranim ID-om. Za spremanje u bazu se koristi SQLite naredba `db.run`, koja je namijenjena za izvođenje SQL upita koji ne očekuju rezultate vraćene iz baze, kao što je naredba INSERT.

5.3.1.2. API zahtjevi

Pri pristupanju korijenskoj ruti (eng. *root path*) web aplikacije, stranica se automatski preusmjerava na rutu `/play`, kako bi se odmah učitala stranica za igru kao početna stranica aplikacije.

```
app.get("/", (req, res) => {
  res.redirect("/play");
});
```

Radi toga je učinak isti kao da se slao GET zahtjev na rutu `/play`, za koji je definirana funkcija povratnog poziva u kojoj se preko funkcije `loadPage` učitava HTML za stranicu igre i šalje kao odgovor klijentu.

```
app.get("/play", async (req, res) => {
  const playHtml = await loadPage("play");
  res.send(playHtml);
});
```

Funkcija `loadPage` preko funkcije `loadHTML`, koja koristi modul `fs` za učitavanje datoteka, učitava tri HTML dokumenta – željenu stranicu (`play.html` ili `leaderboard.html`, ovisno o nazivu rute na koju je poslan GET zahtjev), `nav.html` za definiranje izgleda navigacijske trake koja je jednaka na svim stranicama, i `head.html` za uključivanje značajki zajedničkih svim stranicama poput favicona, skripte `style.css` i slično.

```
function loadHTML(path) {
  return fs.readFile(__dirname + "/public/html/" + path + ".html",
    "UTF-8");
}
async function loadPage(pageTitle) {
  let pages = [loadHTML(pageTitle), loadHTML("nav"), loadHTML("head")];
  let [page, nav, head] = await Promise.all(pages);
  page = page.replace("#nav#", nav);
```

Funkcija također dinamički dodjeljuje pripadnu CSS skriptu svakoj stranici, kao i klasu `active-page` onoj poveznici na navigacijskoj traci koja predstavlja stranicu na kojoj se korisnik trenutno nalazi, bila to stranica za igru ili za rezultate, kako bi se za tu poveznicu primijenio CSS stil za njezino isticanje.

```
  head += `<link rel="stylesheet" type="text/css"
href="../css/${pageTitle}.css" />`;
  page = page.replace("#head#", head);
  const linkToReplace = `href="/${pageTitle}"`;
  const markPageActive = `${linkToReplace} class="active-page"`;
  page = page.replace(linkToReplace, markPageActive);
  return page;
}
```

Za GET zahtjev na rutu `/leaderboard` se također šalje odgovor u obliku dinamički generiranog HTML dokumenta, s time da je sadržaj stranice generiran u obliku tablice s podacima o igračima i njihovim rezultatima iz baze. Tablica je straničena na strani poslužitelja, pa se u nju ne stavljaju svi rezultati, već samo rezultati za onu stranicu tablice koja se trenutno

prikazuje na strani klijenta. Za to je zadužena funkcija `getPlayersPaginated` koja će biti detaljnije obrađena u idućoj sekciji.

```
app.get("/leaderboard", async (req, res) => {
  const players = await getPlayersPaginated(1, 10);
  const playersTable = generatePlayersTable(players);
  let leaderboardHtml = await loadPage("leaderboard");
  leaderboardHtml = leaderboardHtml.replace("#content#", playersTable);
  res.send(leaderboardHtml);
});
```

Konačno, definirani su GET i POST zahtjevi za rute `api/players`, gdje se na GET zahtjev vraća JSON odgovor sa svim igračima iz baze (funkcija `getPlayers`) ili samo s igračima za određenu stranicu tablice (funkcija `getPlayersPaginated`). Na POST zahtjev se izvršava funkcija `savePlayer` za spremanje podatka o igraču u bazu i kao odgovor se šalje poruka uspjeha ili greške.

5.3.1.3. Straničenje

S klijentske strane se kod učitavanja stranice s rezultatima ili promjene stranice tablice šalje GET zahtjev na rutu `api/players` s parametrima upita `itemsPerPage` i `page`, npr. na rutu `api/players?itemsPerPage=5&page=3` za treću stranicu tablice koja prikazuje pet igrača po stranici. Parametri se iz objekta zahtjeva predaju funkciji `getPlayersPaginated`, koja definira indeks početka za stranicu tablice, odnosno koliko elemenata treba preskočiti pri dohvatu podataka iz baze kako bi se došlo do početka željene stranice.

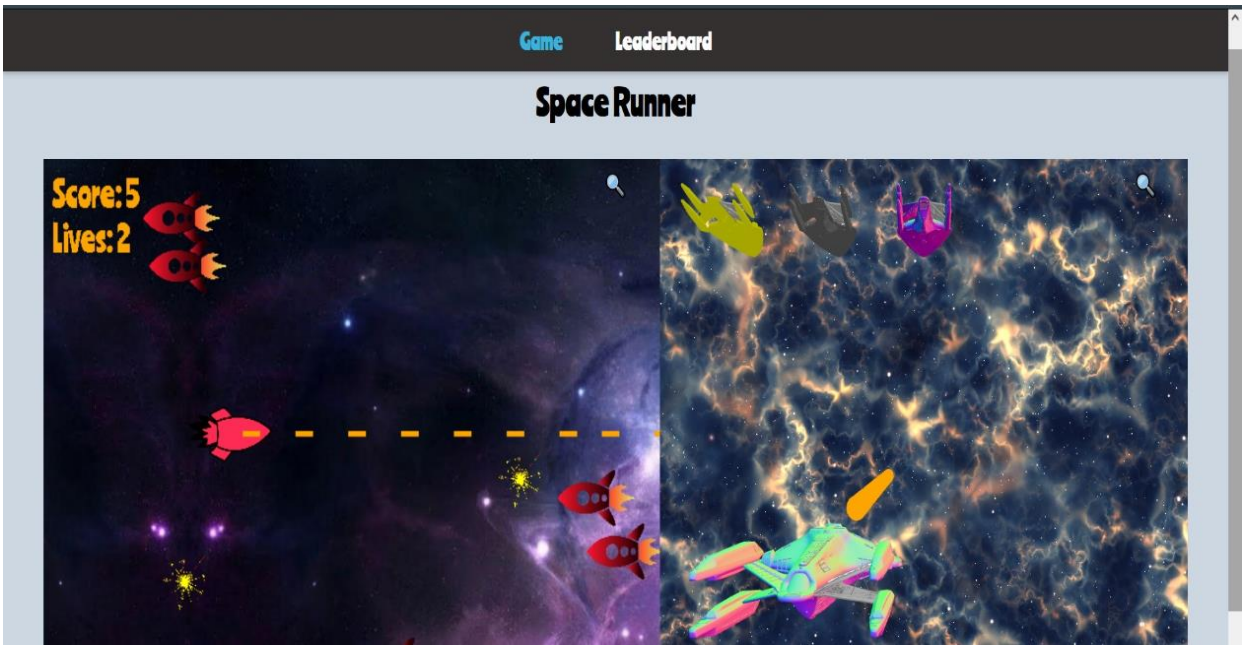
```
const offset = (page - 1) * itemsPerPage;
```

Potom se naredbi `db.all` predaje SQL upit `SELECT * FROM players ORDER BY id DESC LIMIT ? OFFSET ?`, gdje klauzula LIMIT dobiva vrijednost varijable `itemsPerPage`, a OFFSET vrijednost varijable `offset`. Pritom se koristi `ORDER BY id DESC` kako bi prikaz igrača u tablici počeo od posljednjeg igrača koji je odigrao igru.

5.3.2. Klijentska strana

5.3.2.1. Stranica za igru

HTML kod stranice za igru `play.html` se generira na strani poslužitelja na ranije opisan način, i u nju se uključuje skripta `play.css` za primjenu pripadnog CSS stila, te skripta `play.js` za logiku stranice na klijentskoj strani.



Slika 5: Stranica za igru s 2D prikazom kojim upravlja C++ i 3D prikazom kojim upravlja JavaScript

Skripta `play.js` ima nekoliko odgovornosti: učitavanje skripte `index.js` koja je zadužena za komunikaciju s WebAssembly modulom, prikaz sučelja za početak igre, prikaz dijela sučelja za kraj igre koji sadrži omogućuje upis korisničkog imena i ponovno pokretanje igre, komunikaciju sa skriptom `avatarSelection.js` koja upravlja 3D prikazom igre te upravljanje prikazom punog zaslona za 2D i 3D prikaze igre.

Najvažnija odgovornost skripte `play.js` je dinamičko učitavanje skripte `index.js`. Prvo se provjerava je li `Modul.canvas` već definiran, što će biti slučaj ako se po drugi put posjećuje stranica za igru prebacivanjem s neke druge stranice kao što je stranica s rezultatima i u tom slučaju bi pokušaj redefiniranja doveo do konflikta s WebAssembly modulom.

```
if (!Module.canvas) Module.canvas = document.getElementById("canvas");
```

Definicija ove varijable omogućuje povezivanje WebAssembly modula s odgovarajućim HTML elementom `canvas` koji ima istoimeni `id`. U slučaju stranice `play.html` koja ima dva HTML elementa `canvas`, takav `id` ima lijevi prozor budući da se u njemu treba prikazati rezultat C++ logike upravljanja i iscrtavanja 2D prikaza igre. Desnim prozorom za 3D prikaz igre upravlja čisti JavaScript kod pomoću biblioteke ThreeJS te radi toga on nije potreban WebAssembly modulu.

Nakon definiranja ove varijable, poziva se `loadIndexJS` za učitavanje `index.js`, gdje pri svakom novom učitavanju stranice uklanja stari `index.js` ako on postoji i uključuje novi u `head` element, kako bi se izbjegli konflikti s WebAssembly modulom.

```
function loadIndexJS() {
  let existingScript = document.querySelector("script[src='/index.js']");
  if (existingScript) existingScript.remove();
  let script = document.createElement("script");
  script.src = "/index.js";
  document.head.appendChild(script);
}
```

Nakon toga se u objektu `Module` definiraju varijable koje sadrže funkcije povratnog poziva koje se trebaju izvršiti svaki put kada se dogodi određeni događaj u igri kojom upravlja C++ logika. Primjerice, `Module.onGameLoaded` kao parametre prima visinu i širinu SDL prozora koje su inicijalizirane unutar C++ koda pri prvom učitavanju igre, te ih koristi za postavljanje korisničkog sučelja početnog zaslona ili zaslona za ponovno pokretanje igre na iste dimenzije, kako bi bili usklađeni s C++ prikazom prozora. Isto tako, `Module.onGameStarted` i `Module.onGameEnded` postavljaju globalnu varijablu `playing` na istinitu ili lažnu, kako bi bila pravovremeno ažurirana za skripte `play.js`, te `avatarSelection.js` koja upravlja 3D prikazom igre. Ove funkcije povratnog poziva su preko objekta `Module` dostupne C++ funkcijama, te će ih one kod odgovarajućeg događaja (učitavanje, početak ili kraj igre) u igri pozvati i na taj način omogućiti JavaScript kodu praćenje statusa igre te prilagođavanje korisničkog sučelja prema tom statusu, kao što je prikaz ili sakrivanje sučelja za početak ili ponovno pokretanje igre.

Inicijalno je `div` kontejner koji sadrži gumb START postavljen kao vidljiv, a na ovaj gumb je dodan slušač događaja koji na klik poziva C++ funkciju `start_game`, sakriva JavaScript sučelje za početak igre i postavlja varijablu `playing` na `true`.

```
startButton.addEventListener("click", function () {
  Module.ccall("start_game", "void", ["number"], [Module.context]);
  document.getElementById("start-container").style.display = "none";
  playing = true;
});
```

Analogno se na događaj klika na gumb RESTART, uz sakrivanje JavaScript sučelja s tim gumbom, poziva C++ funkcija `restart_game` koja, osim resetiranja informacija za igru, preko `EM_ASM` bloka sakriva sučelje za ponovno pokretanje igre, što uključuje i polje za upisivanje korisničkog imena igrača i gumb za njegovo slanje.

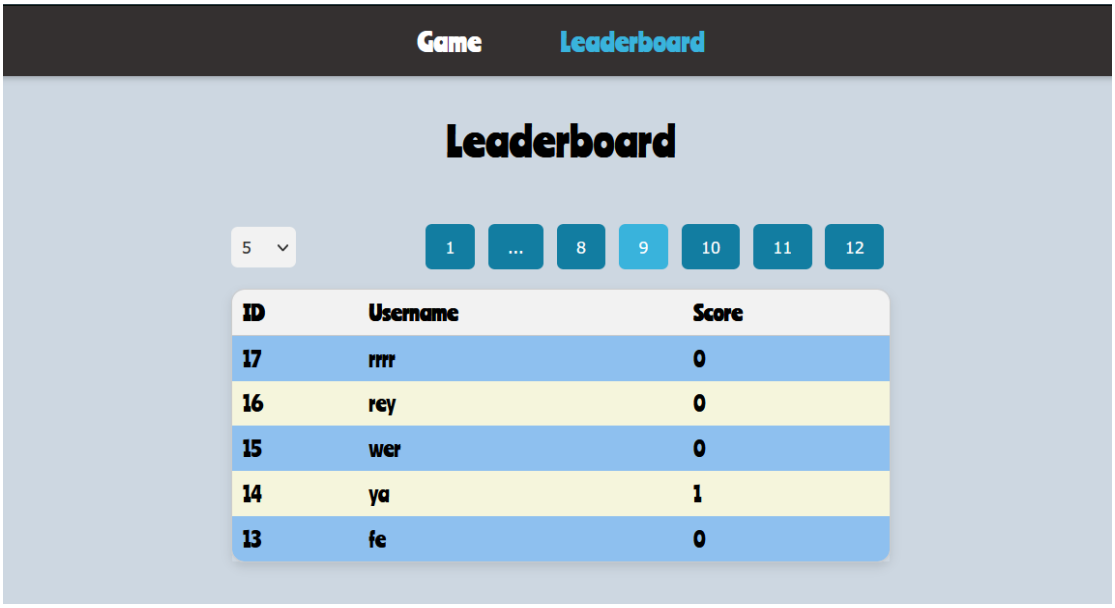
Na klik gumba za slanje korisničkog imena se šalje POST zahtjev prema poslužitelju gdje su u tijelu zahtjeva sadržani uneseno ime i rezultat odigrane igre, koji se dobiva od C++ koda preko varijable `Module.playerScore`. Nakon što se dobije odgovor od poslužitelja,

sakrivaju se polje za unos i gumb za slanje korisničkog imena, no igra se ne aktivira ponovno prije nego što korisnik klikne gumb RESTART.

Dodatno, skripta `play.js` upravlja i prikazom prozora igre u punom zaslonu, i za 2D i 3D prozor. Kako su na stranici prisutna dva elementa `canvas`, svaki prozor se može samo zasebno prikazati u punom zaslonu klikom na ikonicu povećala koja se nalazi u gornjem desnom kutu svakog prozora.

5.3.2.2. Stranica za rezultate

Stranica za rezultate prikazuje tablicu s podacima o igraču kao što su ID, korisničko ime uneseno na kraju odigrane igre i rezultat postignut u toj igri. U HTML stranicu generiranu na strani poslužitelja `leaderboard.html` uključene su `leaderboard.css` za primjenu CSS stila i `leaderboard.js` za logiku prikaza tablice. Tablica ima opciju promjene broja rezultata koji se prikazuju po stranici.



Slika 6: Stranica s tablicom s podacima o igračima i njihovim rezultatima

U skripti leaderboard.js, varijabla `selectedItemsPerPage` označava broj rezultata prikazanih po stranici, dok se `currentPage` postavlja na početnu stranicu, koja je inicijalno 1. Nakon što je korisnički odabir broja rezultata po stranici postavljen, skripta prvo poziva funkciju `getPagesTotal()`, koja asinkrono dohvaća ukupan broj rezultata i računa ukupan broj stranica na temelju odabranog broja rezultata po stranici. Nakon toga, funkcija `showPage(1)` se poziva kako bi inicijalno prikazala prvu stranicu rezultata.

Kada korisnik mijenja broj rezultata po stranici, vrijednost `selectedItemsPerPage` se ažurira i pozivaju se funkcije `getPagesTotal()` i `showPage(1)` kako bi se osvježili podaci i

prikazala nova prva stranica. Funkcija `showPage(page)` je odgovorna za dohvaćanje podataka o rezultatima s odgovarajuće stranice putem GET zahtjeva. Nakon što dobije podatke za tu stranicu, generira HTML kod za prikaz tablice rezultata te poziva funkciju `createPagination` kako bi stvorila navigaciju tj. prikaz gumba za tablicu.

Kako je broj raspoloživih gumba u navigaciji uvijek ograničen na 7, ako je broj stranica manji ili jednak tome, generiraju se gumbi s brojevima za sve dostupne stranice. U slučaju da je ukupan broj stranica veći od 7, dinamički se određuju gumbi na kojima će se prikazivati tri točkice umjesto broja ako nije moguće prikazati sve brojeve stranica između prve i trenutne stranice. Za određivanje rasporeda gumba u ovom slučaju se koristi funkcija `getRangeAroundNumber`, koja je odgovorna za izračunavanje raspona brojeva oko zadane vrijednosti tj. trenutnog broja stranice (parametar `number`), uzimajući u obzir ograničenje definiranog raspona, koji čine prva i zadnja stranica koje se predaju kao polje za parametar `constraintRange`.

```
function getRangeAroundNumber(number, constraintRange) {
  const start = Math.max(number - 2, constraintRange[0]);
  const end = Math.min(number + 2, constraintRange[1]);
  const result = Array.from(
    { length: end - start + 1 },
    (_, index) => start + index
  );
  let diff = 1;
  while (result.length < 5) {
    if (end == constraintRange[1]) result.unshift(result[0] - diff);
    else if (start == constraintRange[0])
      result.push(result[result.length - 1] + diff);
  }
  return result; }
```

Ova funkcija osigurava da brojevi stranica prikazani prije i poslije trenutne stranice u navigaciji budu relevantni za tu stranicu.

5.3.3. Implementacija ThreeJS biblioteke

Skripta `avatarSelection.js` se uključuje u stranicu igre `play.html` i koristi biblioteku ThreeJS za prikaz interaktivne 3D scene u desnom prozoru koja omogućava odabir avatara za igru i primjenu shadera na pozadini. Skripta također omogućava manipulaciju avatara putem tipkovnice i stvaranje metaka klikom miša, kako bi se igračeve akcije u 2D prikazu igre slagale s 3D prikazom.

Prilikom učitavanja stranice za igru, 2D prikaz igre je postavljen na početnu stranicu s gumbom START, dok 3D prikaz služi kao izbornik igračevog avatara na kojem se inicijalno prikazuju tri modela svemirskih brodova između kojih korisnik može birati, svaki od kojih ima drugačiji materijal i izgled. Pri kliku na START gumb u 2D prikazu igre, C++ kod obavještava skripte `play.js` i `avatarSelection.js` promjenom Booleove varijable `playing`, te se prikaz 3D scene mijenja tako da se odabrani model postavlja u sredinu ThreeJS scene. U scenu je postavljena ravnina koja služi kao pozadina i na nju je primijenjen shader s fraktalnim uzorkom koji simulira svemir. Ažurira se sa svakom sličicom kako bi se postigao efekt pokreta te dojam kao da se statički model broda kreće kroz dubinu. U gornji lijevi kut su tijekom igranja postavljena sva tri modela brodova, te se korisnik može u bilo kojem trenutku prebacivati između njih, što mijenja igračev avatar u 2D i 3D scenama.

5.3.3.1. Upravljanje ThreeJS scenom

Za učitavanje modula potrebnih za korištenje ThreeJS-a se koristi ImportMap konfiguracija, koja koristi objekt `imports` za mapiranje imena modula na njihove stvarne putanje, odnosno URL-ove gdje se ti moduli mogu pronaći i učitati. Ona je definirana unutar HTML koda za `head.html`:

```
<script type="importmap">
  {
    "imports": {
      "three": "https://unpkg.com/three@v0.155.0/build/three.module.js",
      "three/addons/": "https://unpkg.com/three@v0.155.0/examples/jsm/"
    }
  }
</script>
```

U ovom slučaju, modul `three` mapiran je na URL glavnog modula ThreeJS (`three.module.js`), a poddirektorij `three/addons/` mapiran je na direktorij `jsm/` u okviru Three.js primjera. U skripti `avatarSelection.js`, linija `import * as THREE from "three"` uvozi sve module iz Three.js pod imenom THREE, dok linija `import { OBJLoader } from "three/addons/loaders/OBJLoader.js"` uvozi modul `OBJLoader` (koji će biti potreban za učitavanje 3D modela iz `.obj` datoteka) iz poddirektorija `addons/loaders/` unutar Three.js primjera.

Unutar skripte `avatarSelection.js` se prvo preko `Promise.all` učitavaju shaderi iz vanjskih datoteka, a zatim poziva funkcija `renderScene` za prikaz scene.

```
Promise.all([
  loadShader("./shaders/vertexShader.glsl"),
```

```
    loadShader("./shaders/fragmentShader.glsl"),
  ])
  .then(([vertexShaderCode, fragmentShaderCode]) => {
    renderScene(avatarContainer, vertexShaderCode, fragmentShaderCode);
  })
```

Za učitavanje shadera iz vanjskih `.glsl` datoteka smještenih u direktoriju `shaders` se koristi funkcija `fetch`.

```
const loadShader = async (shaderPath) => (await fetch(shaderPath)).text();
```

Funkcija `renderScene` poziva nekoliko funkcija preko kojih inicijalizira scenu i osnovno osvjetljenje za nju kako bi se mogli vidjeti materijali modela (`initScene`), postavlja shader definiranjem pozadinske plohe na kojoj ga iscrtava (`renderShader`), učitava 3D modele brodova iz vanjskih datoteka korištenjem modula `OBJLoader` (`loadModels`), i postavlja slušače na događaje pomicanja i klika miša te pritiska na tipkovnicu. Na kraju poziva funkciju `render`, koja se predaje funkciji `requestAnimationFrame` kako bi ažurirala prikaz scene i animacije.

Funkcija `requestAnimationFrame` je metoda koja prihvaća funkciju kao argument i govori pregledniku da je potrebno izvršiti tu funkciju prije nego što se prikaže sljedeća sličica na zaslonu. Kako ona samo jednom stavlja funkciju koju želimo izvršiti u red čekanja za izvršenje pri sljedećem osvježavanju zaslona, da bi se postigla neprekidna animacija i redovito ažuriranje scene, mora se pozvati `requestAnimationFrame` unutar same funkcije koja joj je prethodno predana. Tako se osigurava da se željena funkcija ponovno pozove pri svakom sljedećem osvježavanju zaslona. Na ovaj način `render` koristi `requestAnimationFrame` kako bi se omogućila beskonačna petlja za prikazivanje scene i animacije.

Unutar funkcije `render` se izvršava provjera uvjeta. Ako je igra u stanju igranja, provjerava se model svemirskog broda odabran za igračevog avatara i postavlja na sredinu scene, dok se u gornji lijevi kut postavljaju sva tri umanjena modela koji i dalje služe kao izbornik te igrač može kliknuti na bilo koji od njih da promijeni model u sredini. Svakom promjenom modela se mijenja i model u 2D prikazu, a ta komunikacija se odvija pozivom C++ funkcije `set_avatar` preko metode `Module.ccall` iz JavaScript koda, te `set_avatar` pri svakoj obavijesti o promjeni avatara mijenja SDL2 prikaz boje svemirskog broda u 2D igri preko C++ koda.

S druge strane, ako igra trenutno prikazuje START ili GAME OVER zaslon, `render` uklanja postojeći avatar (ako on postoji), te se resetiraju položaji i rotacije modela kako bi se vratili na stanje izbornika kakav je bio pri prvom učitavanju igre, odnosno tri uvećana modela u sredini scene koji se rotiraju po Y osi. Isto tako, u svakoj iteraciji petlje se ažurira varijabla

`iTime` kako bi se osiguralo da se shaderi mogu ažurirati tj. mijenjati svoj prikaz u svakoj sličici radi postizanja efekta pozadine koja se mijenja kao da se brod kreće kroz svemir u 3D prikazu.

Posljednji dio funkcije `render` odnosi se na upravljanje mecima, koji se kao i u 2D prikazu, kreiraju u 3D prikazu pritiskom na tipku razmaka. Ovdje se u petlji kroz postojeće metke (pohranjene u polju `bullets`) ažuriraju njihovi položaji. Meci se pomiču po osi Z i povećavaju svoju poziciju po osi Y, kako bi se postigao efekt kao da 3D svemirski brod ispucava u dubinu. Ako metak udalji dovoljno od broda, uklanja se iz scene i briše iz polja `bullets`. Konačno, `renderer.render(scene, camera)` se koristi za prikazivanje trenutnog stanja scene.

Ostatak skripte definira pomoćne metode, kao što su `setModelsSelection` koja postavlja modele svemirskih brodova na scenu unutar izbornika, `setAvatarModel` koja postavlja odabrani avatar na scenu, funkcije za manipulaciju modelima (pozicija, rotacija, veličina) i stvaranje metaka, `createShipModels` koja učitava modele brodova, funkciju za primjenu materijala na modele, funkcije za interakciju s mišem i tipkovnicom, kao i za promjenu avatara.

5.3.3.2. Implementacija shadera

Shaderi su programi koji se izvode na grafičkoj kartici, za razliku od većine programskih jezika i aplikacija koje su namijenjene izvođenju na procesoru. Time omogućuju veliku kontrolu nad procesom renderiranja, uključujući osvjetljenje, boje, teksture i druge vizualne efekte. U kontekstu ove web aplikacije, shaderi se koriste unutar ThreeJS scene za generiranje dinamičke pozadine svemira kako bi se kreirala iluzija kretanja kroz prostor i u 3D prikazu igre. Njihovo korištenje omogućuje WebGL koji pruža izravan rad s grafičkom karticom, dok biblioteka ThreeJS pojednostavljuje rad s WebGL-om. WebGL koristi dva osnovna tipa shadera: vertex shader i fragment shader. Vertex shader obrađuje svaku točku geometrije, dok fragment shader obrađuje svaki piksel prikaza. Radi toga se koriste dvije datoteke za svaku vrstu shadera, `vertexShader.glsl` i `fragmentShader.glsl`. Oba shadera su pisana u GLSL-u (eng. *OpenGL Shading Language*), jeziku za programiranje grafike u 3D aplikacijama. Njihove datoteke su smještene u direktoriju `shaders` i obje se učitavaju unutar skripte `avatarSelection.js` pri učitavanju stranice za igru, na način prikazan u prethodnoj sekciji.

Vertex shader odgovoran je za izračun transformacija svake točke objekta unutar prostora. Svrha ovog shadera je postaviti stvarne koordinate točaka u odnosu na kameru i osvjetljenje, te stvoriti promjenjive varijable koje će se prenijeti fragment shaderu za daljnje izračune. Fragment shader obavlja glavni dio izračuna za stvaranje vizualnog efekta, odnosno

fraktalnih uzoraka, te obrađuje svaki piksel na ekranu i određuje njegovu boju i izgled. Kako bi se rezultat shadera mogao računati za svaku sličicu zaslona radi dinamičkog prikaza, u funkciji `render` se ažurira varijabla vremena `iTime` koja se koristi u izračunima fragment shadera.

5.4. Prednosti i nedostaci korištenja C++ jezika u razvoju web aplikacije

Kao jednu od potencijalnih prednosti uvođenja C++ koda u web aplikacije se u literaturi navode performanse po pitanju brzine izvođenja koda, no bez mjerenja performansi ove web aplikacije i njezine igre, te bez usporedbe s igrama u pregledniku koje koriste samo JavaScript kod i biblioteke, za sada se ne može procijeniti je li došlo do poboljšanja performansi zbog pisanja glavne logike igre u C++. Zahvaljujući Emscripten podršci za većinu standardnih C++ funkcionalnosti i biblioteka, omogućena je uporaba biblioteke SDL2 za pristup niske razine multimedijskim mogućnostima poput tipkovnice, miša, i općenito za renderiranje 2D grafike. Ipak, postoje mnoge JavaScript biblioteke i okviri koje se mogu koristiti umjesto SDL2 za iste svrhe, kao što su Phaser, PixiJS, PhantomJS i slično, tako da mogućnost korištenja SDL2 preko C++ koda ne predstavlja prednost sama po sebi.

Isto tako je otvoreno i pitanje je li C++ u kombinaciji s WebAssemblyjem po pitanju performansi i lakoće integracije bolji odabir za 3D grafiku, s obzirom da uvođenje OpenGL-a u C++ kod, iako moguće, nije uvijek jednostavno, pogotovo s kombiniranjem shadera i drugih vrsta renderiranja putem biblioteka kao što je SDL2 jer je u tom slučaju potrebno pravovremeno izmjenjivati između WebGL i SDL konteksta unutar istog prozora. U ovom slučaju sam WebGL vjerojatno pruža bolju alternativu po pitanju jednostavnosti korištenja, iako ne podržava sve mogućnosti OpenGL-a poput drugih vrsta shadera osim vertex i fragment shadera.

WebAssembly je omogućio da se mogu koristiti web tehnologije (HTML, CSS, JavaScript, NodeJS) samo za uređivanje korisničkog sučelja i posluživanje datoteka ako je to sve za što ih se želi koristiti, a poslovna logika aplikacije, odnosno igre, može se pisati u jeziku po izboru, u ovom slučaju u C++. Tako se mogu koristiti strukture specifične za taj jezik koje ne funkcioniraju na isti način u JavaScriptu, kao što su upravljanje memorijom i pokazivači, što kod ove aplikacije najviše dolazi do izražaja u logici dijeljenja informacija o stanju igre između različitih dijelova C++ koda putem strukture `Context` čijim se članovima pristupa putem pokazivača. Na taj način se može iskoristiti sloboda da se biraju programski jezici po prikladnosti za poslovnu logiku web aplikacije i po vlastitim preferencijama.

6. Zaključak

WebAssembly je još uvijek relativno nova tehnologija i ostaju pitanja za koja je potrebno više istraživanja, poput toga je li smisleno koristiti ga za OpenGL ako već postoji WebGL kao način integracije 3D grafike u web aplikacije, i kakva vrsta poslovne logike nosi najveće uštede u performansama kada se on koristi, u usporedbi s tradicionalnim web tehnologijama. Također se postavlja pitanje koji su programski jezici najpopularniji za prevođenje u WebAssembly, što pored samog jezika ovisi i o stupnju razvijenosti tehnologija koje podržavaju integraciju s WebAssemblyjem i web preglednicima. Primjerice, pored C/C++, Rust postaje sve popularniji odabir za takve projekte.

Iako i dalje ima prostora za napredak po pitanju zrelosti alata, WebAssembly i Emscripten kao tehnologije predstavljaju još jedan korak u uklanjanju prepreka između tradicionalnih desktop i web aplikacija, te između ostaloga ovaj trend dovodi i do lakše primjene C++ jezika u web aplikacijama. I pored novijih jezika koji mogu služiti kao alternative C++ jeziku, kao što su Rust, Go i drugi, C++ se održao do danas zahvaljujući učinkovitosti i raširenosti u različitim domenama, pogotovo u području operacijskih sustava, grafike i igara. Radi toga postoji motivacija za širenjem njegove primjene i unutar web aplikacija kako se Web nastavlja razvijati izvan svojih tradicionalnih tehnologija. Ipak, potrebno je za svaku web aplikaciju procijeniti u kojim će domenama ta primjena donijeti značajne prednosti u smislu proširivanja mogućnosti aplikacije, njezine učinkovitosti i prenosivosti.

Popis literature

Battagline, R. (2021). *The Art of WebAssembly*. No Starch Press, Inc.

Cloudflare. (bez dat.). *What do client side and server side mean? | Client side vs. server side*. https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/. Pristupano 22.4.2023.

Escriva, D., Laganiere, R. (2019). *OpenCV 4 Computer Vision Application Programming Cookbook - Fourth Edition*. Packt Publishing Ltd.

Flanagan, D. (2020). *JavaScript: The Definitive Guide, 7th Edition*. O'Reilly Media, Inc.

Frisbie, M. (2020). *Professional JavaScript® for web developers*. John Wiley & Sons, Inc.

Gallant, G. (2019). *WebAssembly in action: with examples using C++ and Emscripten*. Manning Publications Co.

Gonsalves, L. (2021). *How To Get The Most Out Of OpenGL With C++ And WASM*. https://hackernoon.com/how-to-get-the-most-out-of-opengl-with-c-and-wasm-eop33sb. Pristupano 21.5.2023.

Kurose, J. F., Ross, K.W. (2017). *Computer Networking A Top-Down Approach Seventh Edition*. Pearson Education Limited

MDN (bez dat.). *What is a web server? - Learn web development.* https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server. Pristupano 22.4.2023.

OpenCV: OpenCV modules. (bez dat.). https://docs.opencv.org/4.x/. Pristupano 24.4.2023.

OpenGL Overview - The Khronos Group Inc. (bez dat.). https://www.khronos.org/opengl/. Pristupano 10.6.2023.

Ousterhout, J. K. (1998). *Scripting: Higher-level programming for the 21st century.*

Scully, E. (2019). *JavaScript vs C++: Differences and Similarities*. https://careerkarma.com/blog/javascript-vs-cplusplus/. Pristupano 20.4.2023.

SpiderMonkey JavaScript/WebAssembly Engine. (bez dat.). https://spidermonkey.dev/docs/. Pristupano 24.4.2023.

Taheri, S., Vedienbaum, A., Nicolau, A., Hu, N., & Haghighat, M. R. (2018). *OpenCV.js: Computer vision processing for the openWeb platform.*

Vytenis A. (2022). *Why is React a Library and Next.js a Framework?* https://blog.bitsrc.io/why-is-react-a-library-and-next-js-a-framework-and-which-is-better-cee342bdfe8c. Pristupano 21.4.2023.

Wt, C++ Web Toolkit — Emweb. (bez dat.). https://www.webtoolkit.eu/wt/documentation. Pristupano 20.6.2023.

Zidek, K., Piteľ, J., & Hošovský, A. (2017). *Machine learning algorithms implementation into embedded systems with web application user interface.*

Popis slika

Prilozi

Prilog 1: ZIP datoteka s programskim kodom