SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1172

PROCEDURALNA ANIMACIJA MODELA ČOVJEKA

Marko Prosenjak

Zagreb, veljača 2026.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1172

PROCEDURALNA ANIMACIJA MODELA ČOVJEKA

Marko Prosenjak

Zagreb, veljača 2026.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 6. listopada 2025.

DIPLOMSKI ZADATAK br. 1172

Pristupnik:	**Marko Prosenjak (0036535883)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Proceduralna animacija modela čovjeka**

Opis zadatka:

Proučiti postupak izgradnje modela čovjeka pogodnog za proceduralnu animaciju. Proučiti potrebne pripadne elemente fizikalnog pogona. Razraditi implementaciju fizikalnog pogona za proceduralnu animaciju takozvane simulacije fizike krpene lutke (engl. ragdoll physics) koja odgovara modelu čovjeka. U modelu posebice posvetiti pažnju postavljanju ograničenja pokretljivosti u zglobovima. Obratiti pažnju na optimizaciju implementiranih algoritama, optimizaciju grafičkog i fizikalnog pogona i ostvarivost u stvarnom vremenu. Na nizu primjera ostvariti testiranje dobivenih rezultata. Analizirati i ocijeniti postignute rezultate. Diskutirati upotrebljivost rješenja kao i moguća proširenja. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 13. veljače 2026.

Zahvaljujem se mentorici prof. dr. sc. Željki Mihajlović na stručnom vodstvu, korisnim savjetima i uvođenju u fascinantan svijet računalne grafike. Posebno se zahvaljujem svojoj obitelji na neizmjernoj podršci i strpljenju tijekom cjelokupnog studija. Zahvaljujem i prijateljima koji su mi pomogli i značajno olakšali put kroz studij.

SADRŽAJ

1. Uvod

Animacija je jedna od ključnih tehnika u računalnoj grafici. Služi za "oživljavanje" objekata, te ima moć "ubrizgavanja" emocija u naizgled beživotne objekte [1]. Osim svoje estetske i izražajne vrijednosti, animacija u računalnoj grafici ima i važnu funkcionalnu ulogu. U videoigrama, primjerice, pridonosi uranjanju igrača u virtualni svijet, dok u filmskoj industriji omogućuje prikaz scena, koje bi bilo preskupo, ili čak nemoguće realizirati tradicionalnim metodama snimanja. Zahvaljujući razvijanju novih algoritama, te sve većoj snazi modernog hardvera, koriste se tehnike koje omogućuju realističnije prikaze pokreta. Jedna od takvih tehnika je i tehnika krpene lutke, kojom se ovaj rad bavi. Prednost modernih tehnika nad tradicionalnom tehnikom animiranja je ta što se mogu prilagoditi neočekivanim scenarijima. Samim time olakšavaju posao animatorima, koji ne mogu (niti bi trebali moći) predvidjeti sve moguće scenarije u kojima bi se određeni objekt mogao naći, te za svaki od tih scenarija napraviti posebnu animaciju. Kao primjer takvog problema, dovoljno je navesti problem animacije hodanja. Ta animacija može značajno varirati ovisno o terenu po kojem se lik kreće. Noga treba različito biti postavljena na ravnom i na nakošenom terenu (može hodati nizbrdo i uzbrdo, što dodatno komplicira slučaj). No mogu postojati i značajno kompleksniji scenariji. Primjer je tijelo koje pada, te putem udara u različite prepreke (kako će se ponašati, ovisi o dijelu tijela kojim je udario u prepreku, tome koliko je prepreka na putu, gdje su postavljene...) Srećom, proceduralna animacija, kojom se ovaj rad bavi, uz dovoljno dobro definirane uvjete i ograničenja, može se prilagoditi bilo kakvom scenariju u kojem se objekt može naći.

2. Teorijska osnova

2.1. Proceduralna animacija

2.1.1. Razlika proceduralne i animacije ključnim okvirima

Za razliku od animacije ključnim okvirima (eng. *keyframe animation*), kod koje je potrebno u svakom trenutku animacije definirati gdje će se objekt nalaziti i kamo će biti orijentiran (animacija se uvijek jednako reproducira, neovisno o okolini i uvjetima), kod proceduralne animacije postoji puno više fleksibilnosti. Proceduralna animacija je tip animacije koja se prilagođava okolini i uvjetima, odnosno dinamična je. To je postignuto time što ne koristi isključivo unaprijed definirane okvire, nego prati i definiranu logiku ponašanja, te ograničenja. Može biti u "čista" (eng. *pure*), što znači da se prilikom animiranja koriste isključivo algoritmi, fizičke simulacije i kod, te "hibridna", koja koristi ključne okvire, ali ih modificira da odgovaraju situaciji, kako bi pokreti bili realističniji.

2.1.2. Primjena proceduralne animacije

Hibridna proceduralna animacija najčešće se koristi kod animacije kretanja likova (jer rezultat izgleda fluidnije i realističnije od pokretanja korištenjem isključivo koda). Često se kombinira s inverznom kinematikom, koja služi za ispravljanje pokreta u animaciji dobivenoj iz ključnih okvira (kinematički lanac čine kosti noge kod kretanja, odnosno kosti ruke prilikom dohvaćanja). Čista proceduralna animacija koristi se kod simuliranja tijela na koje djeluju sile (tijelo koje se u potpunosti prepustilo djelovanju sila). Primjer takve animacije je simulacija krpene lutke. Proceduralna animacija koristi se u videoigrama, vizualnim efektima i simulacijama.

2.2. Krpena lutka

Krpena lutka (eng. "ragdoll") u području računalne grafike predstavlja simulirani fizički model promatranog lika (najčešće se radi o humanoidu), koji oponaša ponašanje opuštenog i beživotnog objekta, na kojeg utječu određene sile (poput gravitacije, kolizije i sudara). U računalnim igrama, ova tehnika se najčešće primjenjuje nad "živućim" likovima. Dijeli se na aktivnu (eng. "active ragdoll") i pasivnu (eng. "passive ragdoll"). Kod pasivne instance, u trenutku kada je igrač "oboren", kada pada ili je pogurnut od strane nekog objekta, lik postaje opušten/mlitav (eng. "limp"), odnosno prelazi u stanje krpene lutke. Aktivna instanca se razlikuje po tome što je lik cijelo vrijeme u stanju krpene lutke, no ono radi u kombinaciji s unaprijed napravljenim animacijama. Jednostavnije rečeno, pušta se izvorna animacija, ali je njen izlaz ograničen na ono što fizički pogon dopušta, te na njega utječu sile. Rezultat simulacije krpene lutke može se shvatiti kao određena vrsta animacije, koja za animiranje objekta ne koristi ključne okvire, nego se animiranje provodi uz pomoć fizike. Odnosno, umjesto stvaranja velikog broja animacija za različite vrste scenarija, ova simulacija se oslanja na fizički pogon (eng. "physics engine"), koji izračunava kako bi se dijelovi tijela i zglobovi trebali pomicati i rotirati pri djelovanju sile na tijelo (pazi da ponašanje odgovara fizičkim zakonima svijeta). Prednost ove tehnike je mogućnost produciranja beskonačno mnogo različitih animacija, koje rade u skladu s fizičkim zakonima svijeta. Animatori ne mogu predvidjeti u kojim se svim situacijama tijelo može naći, te ne mogu (niti bi trebali moći) izraditi animaciju za svaki slučaj koji može nastati.

2.2.1. Kosti

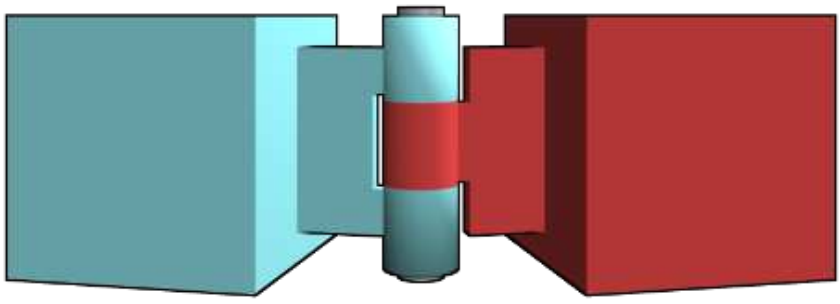
Model krpene lutke sastoji se od većeg broja krutih tijela (eng. "rigid body"), povezanih putem zglobnih ograničenja (eng. "joint constraints") u cjelinu. Svako kruto tijelo omotano je omeđujućim volumenom (koji mu najbolje odgovara po anatomiji tijela - npr. za podlakticu se može koristiti kvadar ili kapsula, dok se za glavu najčešće koristi sfera). Dakle, kruta tijela su oblikovana tako da predstavljaju dijelove anatomije lika i povezana su na način koji oponaša zglobove stvarnog kostura, time omogućujući fizički realističan pad ili reakciju na sile. Za omeđujuće volumene najčešće se koriste grafičke primitive (jer je nad njima jednostavno provesti izračune, a dovoljno precizno omeđuju kosti za naše potrebe) poput kapsule, kvadra i sfere. Svako kruto tijelo, uz omeđujući volumen, također sadrži podatke o masi i inerciji, koji se koriste kod provedbe izračuna u simulaciji. Kruta tijela čine virtualne kosti krpene lutke.

2.2.2. Ograničenja u zglobovima

Za svake dvije neposredno povezane primitive, potrebno je definirati ograničenje u zglobovima (eng. "joint constraint"). Ono definira kako se dvije međusobno povezane kosti mogu pomicati i rotirati relativno jedna u odnosu na drugu. Time se oponaša odnos kosti u stvarnom tijelu. Iznimno je važno pravilno definirati ta ograničenja, kako bi simulacija bila što uvjerljivija i fizički preciznija. U ograničenju su najčešće definirani maksimalan i minimalan dozvoljeni kut između dva zgloba (maksimalan i minimalan kut za koji je moguće saviti jednu kost u odnosu na drugu). U fizičkom pogonu "Bullet Physics" definirano je 13 vrsta ograničenja [2], a neka od njih su:

- "zglobni" (eng. "hinge constraint")
- "stožasti" (eng. "cone twist constraint")
- "ograničenje od točke do točke" (eng. "point to point constraint")
- "6 stupnjeva slobode" (eng. " Six Degrees of Freedom (6DOF) Constraint")
- "klizač" (eng. "slider constraint")

U ovom radu korištena su samo "zglobna" i "stožasta" ograničenja. Zglobna ograničenja dozvoljavaju rotaciju oko samo jedne osi (podsjeća na šarke za vrata). Takvo se ograničenje koristi primjerice za vezu između zglobova nadlaktice i podlaktice. Stožasta ograničenja koriste se za zglobove koji se pomiču unutar volumena stošca, ali se mogu također rotirati oko glavne osi (eng. "twist axis"). Ovakvo ograničenje koristi se za simuliranje prirodnih pokreta zglobova poput kukova i ramena.



Slika 2.1: "Zglobno ograničenje" (izvor: `https://docs.panda3d.org/1.10/_images/bullethinge.png`)

3. Fizički pogon

3.1. Općenito o fizičkom pogonu

Fizički pogon je softverska komponenta koja simulira zakone fizike u virtualnom okruženju, kao što su npr. video igrice. Zadužen je za simuliranje fizičkih interakcija između objekata koji se nalaze u virtualnom fizičkom svijetu. Svaki objekt u svijetu ima definirana fizička svojstva (poput npr. mase), koja utječu na njegovo ponašanje i interakciju s drugim objektima. Fizički pogon koristi matematičke modele i algoritme kako bi izračunao podatke o poziciji, rotaciji, brzini (te mnoštvu drugih veličina) pojedinog objekta. Tijekom provođenja spomenutih izračuna, u obzir se uzimaju sva ograničenja i sile koje djeluju na objekt (poput gravitacije i trenja, te sile koje su nastale kao posljedica sudara s drugim tijelima), kako bi ponašanje bilo što uvjerljivije i fizički ispravno. Primjene fizičkog pogona su razne, a neke od poznatijih su u:

- videoigrama - interakcija objekata, kompleksne animacije likova, destrukcija...
- fizičkim simulacijama - simuliranje kompleksnih fizičkih pojava
- računalnim efektima (eng. *"CGI"*) - stvaranje realističnih specijalnih efekata i pokreta
- robotici

Zašto je koristan:

- realizam - osigurava interakciju objekata koja je u skladu s fizičkim zakonima koje poznajemo
- efikasnost - ušteda vremena koje bi se trebalo utrošiti na predviđanje i izradu animacija za određene fizikalne scenarije i pojave
- "uranjanje" (eng. "immersion") - simuliranjem prirodnih interakcija, stvara se prirodno ponašanje, koje je korisniku poznato i prihvatljivo

3.2. Bullet Physics

Za simulaciju je korišten fizički pogon "Bullet Physics". To je profesionalni, višeplatformski 3D fizički pogon otvorenog koda, koji se koristi za simulaciju dinamike krutih i mekih tijela, te sudara u stvarnom vremenu. Glavne karakteristike:

- dinamika mekih i krutih tijela - Bullet simulira fiziku krutih i deformabilnih tijela, poput tkanine, užadi i ostalih fleksibilnih predmeta
- detekcija sudara - koristi sofisticirane algoritme, čime osigurava preciznu detekciju sudara, te ispravnu informaciju o trenutku, mjestu i obliku sudara
- ograničenja - veliki broj različitih ograničenja, koja služe za stvaranje kompleksnih cjelina (mnoštvo zglobova povezanih u cjelinu, čime nastaje kompleksno kretanje, koje ovisi o svakom tijelu te cjeline)
- različiti omeđujući volumeni - definiran je velik broj omeđujućih volumena, od kojih su neki jednostavni (sfera, kapsula, kvadar...), dok su neki složeniji (konveksna ljuska, eng. *"convex hull"*)
- simulacija u stvarnom vremenu - fizički pogon je optimiziran i dizajniran za provođenje fizičkih izračuna u stvarnom vremenu, uz postizanje visokih performansi

Biblioteka je besplatna za komercijalnu upotrebu (uz navođenje licence *"zlib licence"* - `http://opensource.org/licenses/Zlib`), te se koristi u programskim rješenjima poput Blender-a, Maya-e i "pogonu igre" (eng. *"game engine"*) Godot-u. Biblioteka podržava sve velike platforme: Windows, Linux, Mac OSX, iOS i Android, no autori smatraju da bi trebala raditi na bilo kojoj platformi koja ima C++ prevodioc. Dva posebno bitna algoritma, koja će biti objašnjena u nastavku, su "sekvencijalni rješavač impulsa" (eng. *"Sequential Impulse Solver"*), te GJK algoritam. Radi se dva krovna algoritma korištena "ispod haube" (eng. *"under the hood"*), koja su bitna za ostvarivanje rješenja, kojim se ovaj rad bavi.

3.2.1. Sekvencijalni rješavač impulsa

Sekvencijalni rješavač impulsa je jedna od najčešće korištenih metoda za razrješavanje kolizija, kontakata i ograničenja zglobova u stvarnom vremenu. Konceptualno je povezan s PGS metodom ("Projected Gauss-Seidel"), ali je prilagođen interaktivnim aplikacijama u stvarnom vremenu [3]. Umjesto rješavanja velikih matrica, radi direktno s krutim tijelima: primjenjuje impulse i ažurira brzine, te ponavlja proces nekoliko

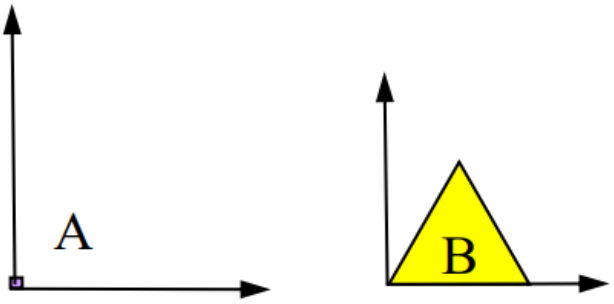
puta. Algoritam funkcionira na sljedećem principu: prolazi kroz sva ograničenja, te za svako ograničenje računa za koliko bi impuls promijenio relativnu brzinu između tijela. Potom ograničava taj impuls unutar definiranog intervala (eng. *clamping*) kako bi se zadovoljile granice. Nakon što je to napravljeno, direktno primjenjuje impuls na brzina oba tijela. Cijeli proces se ponavlja nekoliko puta za svako ograničenje. Nakon nekoliko iteracija, sistem konvergira prema stabilnoj konfiguraciji (tijela se ne probadaju, te se zglobovi drže zajedno). Zbog svoje jednostavnosti i učinkovitosti, koristi se u popularnim fizičkim pogonima poput Bullet Physics-a i Box2D-a. Ova metoda također ima slabosti, koje su vidljive u u sljedećim slučajevima: kada je objekt velike mase postavljen na objekt male mase (npr. tenk na tračnicama), prilikom sudara šarke za vrata (eng. *door hinge*) s objektom velike mase, kod jaza (eng. *gap*) krpene lutke i slično. Tada simulacija postaje nestabilna.

3.2.2. GJK algoritam

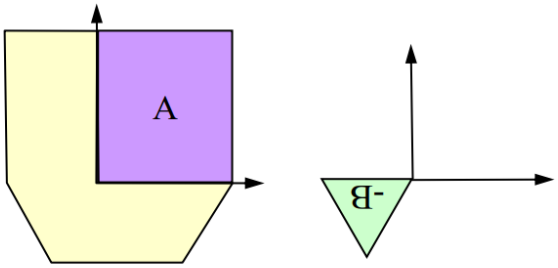
GJK algoritam koristi se za provjeru da li se dva ili više tijela sijeku (jesu li u koliziji). Vrlo je praktičan jer može raditi s bilo kojom kombinacijom konveksnih oblika, te je izrazito brz, što ga čini pogodnim za korištenje u stvarnom vremenu.

Razlika Minkowskog

Prema definiciji, dva se tijela sijeku ako sadrže barem jednu zajedničku točku. S obzirom na to da svako tijelo sadrži beskonačno mnogo točaka, postavlja se pitanje kako za svaku točku provjeriti nalazi li se u oba tijela u prihvatljivom vremenu. Srećom postoji puno elegantnije rješenje u obliku razlike Minkowskog. Razlika Minkowskog je dobivena tako da zrcalimo jedan objekt (koji ćemo označiti s "B") s obzirom na ishodište njegovog koordinatnog sustava, te ga sumiramo s drugim objektom (koji ćemo označiti s "A"). Time je određen prostor u kojem se objekt B može kretati bez da dođe u koliziju s tijelom A [4]. Postavlja se pitanje što smo postigli s generiranjem novog tijela. Ono nam značajno pomaže s provjerom jesu li dva tijela u koliziji, jer je potrebno jedino provjeriti nalazi li se ishodište u razlici Minkowskog, kako bi potvrdili da se zaista sijeku. Razlog tome je taj što će razlika između dviju točaka (od njih beskonačno mnogo) biti 0, što znači da se radi o istoj točki i da ta točka mora biti ishodište [5]. Detekcija kolizije svodi se na pronalaženje trokuta (napravljenog od krajnjih točaka razlike Minkowskog) koji sadrži ishodište. Ako je moguće pronaći takav trokut, postoji kolizija. Taj trokut zovemo "simplex".



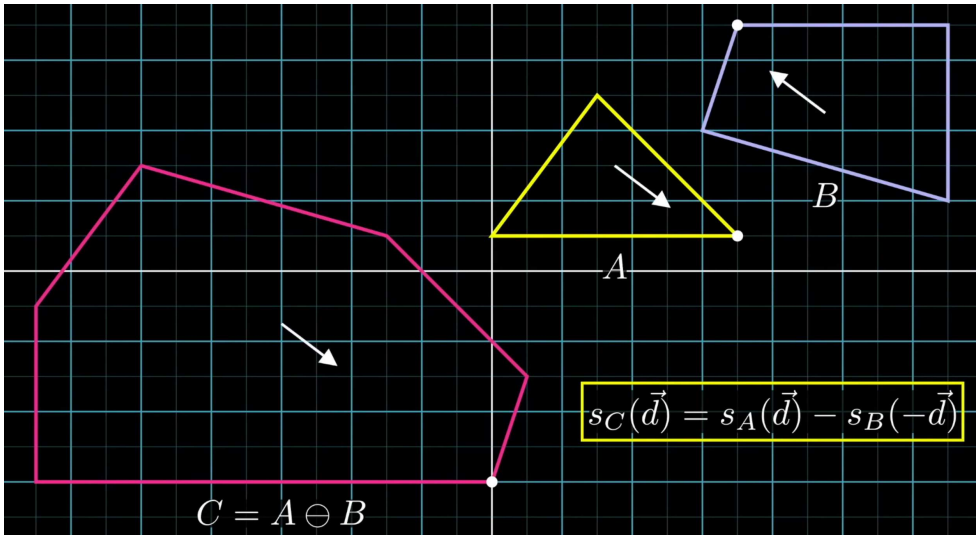
Slika 3.1: Lokalni koordinatni sustavi objekata



Slika 3.2: Razlika Minkowskog

"Support" funkcija

Korisno svojstvo svakog konveksnog geometrijskog lika je to da za svaku krajnju točku na liku vrijedi da postoji vektor smjera u kojem je ta točka najudaljenija. To znači da možemo definirati funkciju, koja uz pomoć vektora smjera, nalazi najudaljeniju točku tog lika. Tu funkciju nazivamo "funkcija podrške" (eng. *"support function"*), a točku (koja je rezultat funkcije) nazivamo "točka podrške" (eng. *"support point"*). Možemo koristiti točke podrške originalnih likova kako bi izračunali točke podrške razlike Minkwoskog. Princip je sljedeći: najprije je potrebno odabrati proizvoljan smjer, naći točku podrške lika "A", korištenjem funkcije podrške za taj lik. Nakon toga treba okrenuti vektor smjera i naći točku podrške lika "B" (korištenjem funkcije podrške za taj lik). Na kraju je potrebno oduzeti točku podrške lika "B" od točke podrške lika "A". Jednostavnije rečeno, nalazimo točke na granici razlike Minkowskog pomoću razlike ekstremnih točaka (u suprotnim smjerovima) likova koji ju čine. Time možemo naći krajnje točke razlike Minkowskog, bez potrebe za izračunom cijele razlike [6].



Slika 3.3: Generiranje krajnjih točaka razlike Minkowskog pomoću funkcija podrške, izvor: `https://www.youtube.com/watch?v=ajv46BSqcK4`

Funkcija podrške vraća točku čiji skalarni umnožak s vektorom smjera daje najveću vrijednost. Ta točka predstavlja najudaljeniju točku na liku u datom smjeru. Ova funkcija je razlog zbog kojeg je moguće provesti algoritam nad bilo koja dva konveksna oblika.

Provođenje algoritma

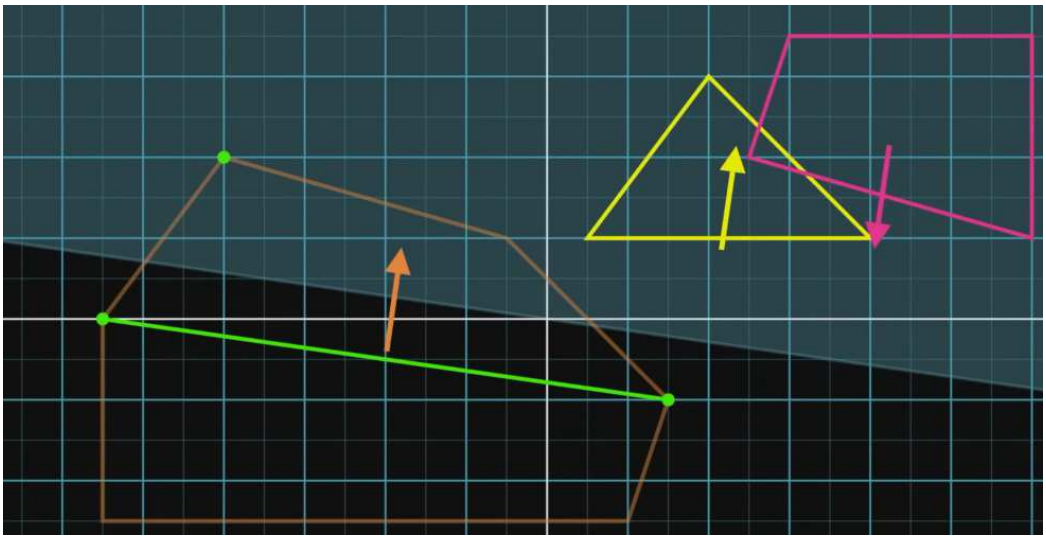
1. Algoritam počinje odabirom proizvoljnog smjera, te određivanjem točke podrške razlike Minkowskog za taj smjer. Time je dobivena prva točka simpleksa.

2. Druga točka simpleksa dobiva se uz pomoć vektora, koji se pruža od prve točke simpleksa prema ishodištu. Razlog odabira tog smjera je taj što je cilj ograditi ishodište simpleksom, čime bi se dokazalo da presjek postoji. Stoga je logičan odabir ekstrema razlike Minkowskog koji se pruža u smjeru od prve točke prema ishodištu (kako bi se druga točka simpleksa nalazila sa suprotne strane ishodišta u odnosu na prvu točku). Stoga je za drugu točku simpleksa logičan odabir krajnje točke razlike Minkowskog, koja je maksimalno udaljena od prve točke simpleksa u smjeru ishodišta.

3. U ovom trenutku potrebno je provjeriti nalazi li se dobivena točka u regiji koja prelazi ishodište. Ako je uvjet zadovoljen, prelazi se na idući korak, inače presjek ne postoji. Razlog tome je taj što je ostvaren maksimalan pomak od prve točke prema ishodištu, no ono svejedno nije premašeno. To znači da se obje točke nalaze s iste strane ishodišta, te ga nije moguće ograditi. Provjera je li točka "prešla" ishodište svodi se na skalarni

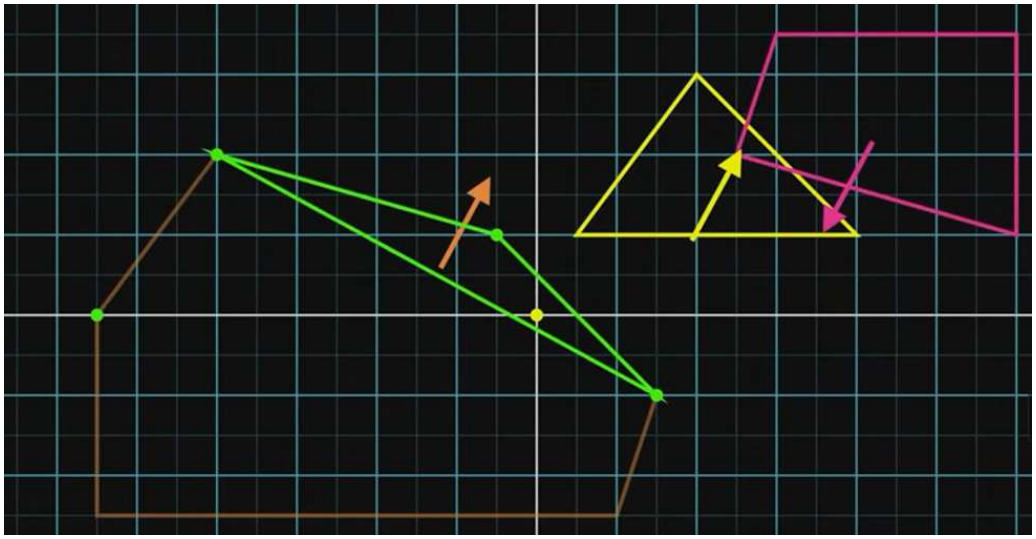
umnožak vektora smjera s vektorom koji se pruža od ishodišta prema promatranoj točki. Ako je taj umnožak manji od 0, točka ne prelazi ishodište.

4. Kao posljednji vektor smjera, uzima se normala nad spojnicom prve dvije točke simpleksa, u smjeru koji gleda prema ishodištu. Koristeći taj vektor smjera, dobivena je treća točka simpleksa. Potrebno je ponovno provesti provjeru iz 3. koraka



Slika 3.4: Odabir treće točke simpleksa, izvor: `https://www.youtube.com/watch?v=ajv46BSqcK4`

5. U ovom koraku je potrebno provjeriti nalazi li se ishodište u dobivenom simpleksu. Ako je to slučaj, presjek sigurno postoji i kolizija je otkrivena, inače je potrebno nastaviti u daljnje iteracije (počevši od koraka 4, s time da spojnica postaje brid simpleksa najbliži ishodištu). Kroz iteracije se prolazi sve dok se ishodište ne nađe unutar simpleksa, ili se ne utvrdi da provjera iz 3. koraka nije zadovoljena.



Slika 3.5: Prikaz slučaja kolizije kod izvođenja GJK algoritma, izvor: `https://www.youtube.com/watch?v=ajv46BSqcK4`

Rad algoritma objašnjen je na 2D primjeru, no glavni koncepti su primjenjivi i u 3D (uz dodatna proširenja).

3.2.3. Fizički svijet

Prvi korak kod izrade simulacije je definirati fizički svijet u kojem će se ta simulacija odvijati. U Bullet Physics-u je taj svijet definiran klasom *btDiscreteDynamicsWorld*. To je objekt koji objedinjuje sve ključne komponente simulacije:

- skup svih krutih tijela koja se u njemu nalaze
- sve definirane zglobne veze i ograničenja
- parametre okoline (poput gravitacije)
- algoritme gibanja i detekcije sudara

te još mnoge druge komponente (koje nisu relevantne za simulaciju krpene lutke). fizički svijet je centralna klasa Bullet Physics biblioteke, jer upravlja cijelom fizičkom simulacijom. Simulacija se pokreće pozivom metode *stepSimulation()*, pri čemu se ažuriraju položaji, rotacije i brzine svih tijela u svijetu. Na taj način *btDiscreteDynamicsWorld* predstavlja osnovu za izradu svake fizičke simulacije.

4. Postavljanje projekta

4.1. Preduvjeti

- Visual Studio 2022 ili više
- C++ 17 ili više
- OpenlGL 3.3 ili više
- CMake

4.2. Postavljanje projekta i integracija Bullet Physics-a

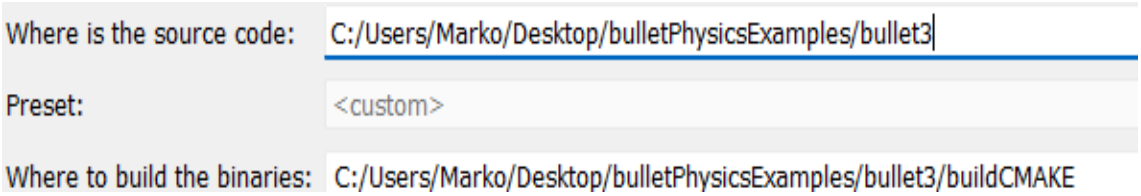
1. Pokretanje cmd.exe
2. Izvođenje naredbi:

```
cd <proizvoljan_direktorij>
git clone https://github.com/NEYMARKO/ModernOpenGL.git
```

3. Izvođenje naredbi:

```
cd <proizvoljan_direktorij_razlicit_onom_iz_drugog_koraka>
git clone https://github.com/bulletphysics/bullet3.git
cd ./bullet3
mkdir buildCMake
```

4. Pokretanje CMake
5. U polje *Where is the source code* staviti putanju do **bullet3** direktorija
   U polje *Where to build binaries* staviti putanju do **buildCMake** direktorija



Slika 4.1: Odabir direktorija u CMake

6. Pritisak na gumb **CONFIGURE**, nakon što su nove vrijednosti prikazane crvenom bojom, potrebno je još jednom pritisnuti na gumb **CONFIGURE**, te zatim na gumb **GENERATE**

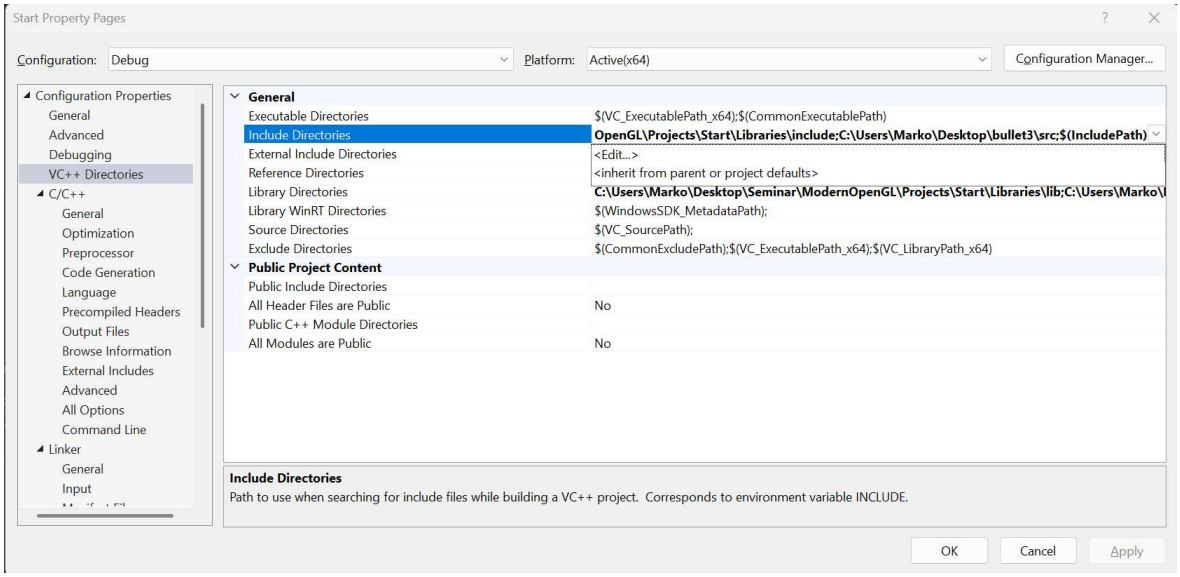
7. Unutar direktorija *bulletCMake* locirati rješenje (eng. *solution*) *BULLET_PHYSICS.sln*, te ga pokrenuti

8. Desni klik na solution -> Build (pobrinuti se da je konfiguracija postavljena na *Debug*, a arhitektura *x64*)

9. Projektima navedenima u 13. koraku potrebno je promijeniti biblioteku vremena izvođenja (eng. *runtime library*):
   - Desni klik na projekt
   - `Properties->C/C++->Code Generation->Runtime Library`
   - Promjena vrijednosti u *"/MDd"*
   - Ponovno izgraditi (eng. *rebuild*) projekt kako bi se izbjegao `missmatch error`

10. Otvaranje rješenja *Engine.sln*, desni klik na rješenje (eng. *solution*) i odabir opcije *Properties*

11. `Configuration Properties->VC++ Directories->` `General->Include Directories` - EDIT (dodavanje novog elementa) - `<puna_putanja>/bullet3/src`

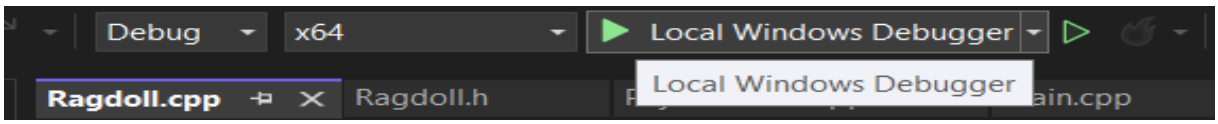


Slika 4.2: Konfiguracija uključujućih direktorija (eng. *Include directories*)

12. `Configuration Properties->VC++ Directories->General->Library Directories` - EDIT (dodavanje novog elementa) - `<puna_putanja>/bullet3/buildCMAKE/lib/Debug`

13. `Linker->Input->Additional Dependencies` - EDIT (dodavanje novih elementa) -
    - *Bullet3Collision_Debug.lib*
    - *Bullet3Common_Debug.lib*
    - *Bullet3Dynamic_Debug.lib*
    - *Bullet3Geometry_Debug.lib*
    - *BulletCollision_Debug.lib*
    - *BulletDynamics_Debug.lib*
    - *LinearMath_Debug.lib*

14. Pokretanje projekta pritiskom na gumb *Local Windows Debugger*



Slika 4.3: Gumb za pokretanje projekta

5. Izrada modela krpene lutke

Kod izrade modela krpene lutke potrebno je definirati kosti (tip: *btRigidBody*), omeđujuće volumene (tip: *btCollisionShape*) i ograničenja u zglobovima (tip: *btTypedConstraint*). Ti su podaci pohranjeni u tri liste, čiji je broj elemenata jednak broju kosti/zglobova (svaki podatak u tim listama odnosi se na jednu od kosti/jedan od zglobova).

```
std::array<btRigidBody*, BONES_COUNT> m_bones;
std::array<btCollisionShape*, BONES_COUNT> m_collisionShapes;
std::array<btTypedConstraint*, JOINTS_COUNT> m_jointConstraints;
```

5.1. Definiranje omeđujućih volumena

Najprije je potrebno definirati omeđujuće volumene za sve kosti koje čine model. Za svaku je kost korišten oblik kapsule (s različitim dimenzijama, ovisno o tipu kosti). Iako se obično za određene kosti (poput zdjelice) koriste primitive poput kvadra, radi jednostavnosti i zadovoljavajuće preciznosti, to nije učinjeno.

```
    m_collisionShapes[BONE_HIPS] = new btCapsuleShape(0.15 * scale,
        0.2 * scale);
    m_collisionShapes[BONE_SPINE] = new btCapsuleShape(0.15 * scale,
         0.28 * scale);
    m_collisionShapes[BONE_HEAD] = new btCapsuleShape(0.1 * scale,
        0.05 * scale);
    m_collisionShapes[BONE_UPPER_LEG_LEFT] = new btCapsuleShape(0.07
         * scale, 0.45 * scale);
    m_collisionShapes[BONE_LOWER_LEG_LEFT] = new btCapsuleShape(0.05
         * scale, 0.37 * scale);
    ...
```

5.2. Definiranje krutih tijela

Nakon što je stvoren omeđujući volumen za svaku kost, potrebno je definirati kruta tijela koja će se simulirati. Tim je tijelima potrebno definirati poziciju u fizičkom svijetu, masu, dodijeliti im odgovarajući omeđujući volumen (koji je stvoren u prošlom koraku), te ih učiniti dijelom fizičkog svijeta, kako bi se mogli simulirati.

```
m_bones[BONE_SPINE] = createRigidBody(1., offset * transform,
    m_collisionShapes[BONE_SPINE]);
```

```
btRigidBody* Ragdoll::createRigidBody(btScalar mass, const
    btTransform& startTransform, btCollisionShape* shape)
{
    bool isDynamic = (mass != 0.0f);

    btVector3 localInertia(0, 0, 0);
    if (isDynamic)
        shape->calculateLocalInertia(mass, localInertia);

    btDefaultMotionState* motionState = new btDefaultMotionState
        (startTransform);

    btRigidBody::btRigidBodyConstructionInfo rbInfo(mass,
        motionState, shape, localInertia);
    btRigidBody* body = new btRigidBody(rbInfo);

    m_physicsWorld->getDynamicsWorld()->addRigidBody(body);
    return body;
}
```

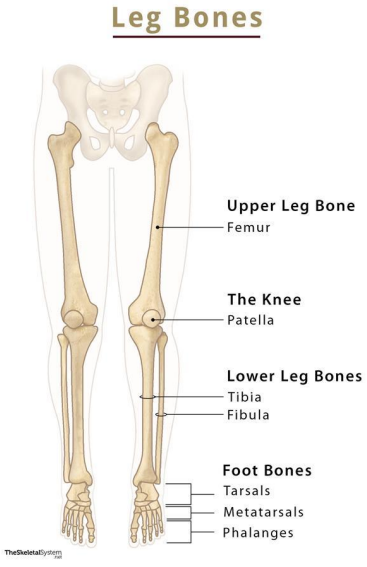
5.3. Definiranje ograničenja

Posljednji korak u izgradnji krpene lutke je definiranje ograničenja koja povezuju pojedina kruta tijela. Kod kreiranja ograničenja, kao argumente konstruktoru, potrebno je predati informacije o krutim tijelima (*btRigidBody*) koja će to ograničenje povezati, te njihovim pripadajućim okvirima (eng. *"frame"* - *btTransform*).

5.3.1. Kreiranje okvira

Okvir je definiran pomoću središnjih točaka tijela (eng. *"pivot point"*), te lokalnih orijentacija (dakle radi se o matrici transformacije). Središnja točka predstavlja točku u

kojoj su dva zgloba povezana, gledano iz lokalnog sustava pojedinog krutog tijela. Jednostavnije rečeno, postavlja se pitanje: "U kojoj točki fizičkog svijeta se ta dva kruta tijela trebaju dodirivati", a odgovor se daje u obliku: "Dodirna točka se u lokalnom sustavu tijela A nalazi <ovdje>, a u lokalnom sustavu tijela B, <ovdje>". Uvjet, koji treba biti zadovoljen, je da te dvije točke moraju odgovarati istoj poziciji u fizičkom svijetu nakon svake iteracije simulacije. To znači da će simulacija na pojedino kruto tijelo primijeniti određene sile, kako bi se taj odnos, definiran ograničenjem, održao. Pozicije središnjih točka definiraju se blizu granica omeđujućih volumena, kako tijela (koja predstavljaju kosti) ne bi preduboko prodirala jedno u drugo. Uzmimo za primjer koljeno: to je zglob koji povezuje bedrenu kost i potkoljenicu (prikazano na slici 5.1). Točka u kojoj su te dvije kosti povezane, nalazi se na donjoj granici omeđujućeg volumena bedrene kosti, te gornjoj granici omeđujućeg volumena potkoljenice. S obzirom na to da se točke definiraju u lokalnom sustavu tijela, korisnik koji stvara ograničenje, treba pripaziti na to da se te točke nalaze što bliže jedna drugoj u sustavu fizičkog svijeta, bez da izađu iz granica pripadajućih omeđujućih volumena. Razlog je taj što će u protivnom doći do primjene impulsa (snage proporcionalne udaljenosti između točaka) kako bi se točke dovele u istu poziciju u fizičkom svijetu. Primjena tih impulsa rezultira ponašanjem koje podsjeća na reakciju tijela na udarac ili bacanje (dolazi do trzanja u zglobu, nad čijim je kostima primijenjen impuls).



Slika 5.1: Anatomija noge, izvor: `https://www.theskeletalsystem.net/wp-content/uploads/2022/08/Leg-Bones.jpg`

Drugi podatak potreban za definiranje okvira je njegova orijentacija. Način na koji se okviri rotiraju, ovisi o tipu ograničenja koje se koristi. Kod *btHingeConstraint* tipa

ograničenja (korišteno za zglobove poput lakta), okvire je potrebno rotirati tako da njihova lokalna z-os leži na pravcu oko kojeg se zglob rotira. Kod ograničenja tipa *btConeTwistConstraint*, koje se koristi za modeliranje zglobova poput ramena, potrebno je orijentirati okvire tako da su njihove lokalne x-osi (*"twist axis"*) poravnate sa smjerom pružanja odgovarajuće kosti (koji je određen pomoću y-osi u lokalnom sustavu krutog tijela). Da bi se osiguralo ispravno ponašanje ograničenja, potrebno je zadovoljiti uvjet da su odgovarajuće osi rotacije poravnate u koordinatnom sustavu svijeta, odnosno: $R_{Rb\_A} * R_{Frame_A} = R_{Rb\_B} * R_{Frame\_B}$ (rezultat umnoška rotacije krutog tijela s rotacijom okvira treba biti jednak za oba kruta tijela). Taj uvjet vrijedi za obje vrste ograničenja. Zadnji korak kod definiranja ograničenja je određivanje koliko se jedno tijelo može rotirati u odnosu na drugo. Granice rotacije definiraju se pomoću funkcije *setLimit*, pri čemu se za svaki tip ograničenja zadaju specifični parametri, koji određuju dozvoljeni raspon pokreta. Obje biblioteke (OpenGL i Bullet Physics) koriste tzv. "sustav desne ruke" (eng. *"right-handed system"*), što znači da *Y* predstavlja vertikalnu, *X* horizontalnu, a *Z* os koja je usmjerena prema promatraču. Kapsule su inicijalno rotirane tako da prate sustav desne ruke.

```
1    btHingeConstraint* hingeConstraint;
2    btConeTwistConstraint* coneConstraint;
3
4    btDynamicsWorld* world = m_physicsWorld->getDynamicsWorld();
5    btTransform localA, localB;
6
7    localA.setIdentity();
8    localB.setIdentity();
9    localA.getBasis().setEulerZYX(0, M_PI_2, 0);
10   localA.setOrigin(m_scale * btVector3(btScalar(0.), btScalar
         (-0.225), btScalar(0.)));
11   localB.getBasis().setEulerZYX(0, M_PI_2, 0);
12   localB.setOrigin(m_scale * btVector3(btScalar(0.), btScalar
         (0.185), btScalar(0.)));
13   hingeConstraint = new btHingeConstraint(*m_bones[
         BONE_UPPER_LEG_LEFT],
14        *m_bones[BONE_LOWER_LEG_LEFT], localA, localB);
15   hingeConstraint->setLimit(btScalar(0), btScalar(M_PI_2));
16   m_jointConstraints[JOINT_LEFT_KNEE] = hingeConstraint;
17   world->addConstraint(m_jointConstraints[JOINT_LEFT_KNEE], true);
```

6. Povezivanje fizičkog i grafičkog pogona

Fizički pogon (eng. physics engine) i grafički pogon (eng. rendering engine) usko surađuju kako bi proizveli uvjerljivu vizualnu simulaciju. Fizički pogon zadužen je za izvođenje svih potrebnih izračuna tijekom simulacije, poput detekcije sudara i ažuriranja položaja tijela u prostoru. Grafički pogon pomoću rezultata, dobivenih od fizičkog pogona, iscrtava scenu (s ažuriranim vrijednostima za sva kruta tijela). Na taj način, fizički pogon osigurava realizam poštujući zakone fizike, dok grafički pogon osigurava vizualnu reprezentaciju tih podataka. Grafički pogon rađen je po uzoru na ECS ("entity component system") arhitekturu, koja odvaja funkcionalnosti u različite komponente, čime dolazi do podjele odgovornosti u smislene cjeline. Konkretno u ovom grafičkom sustavu, entitete predstavljaju objekti u sceni, dok njihove komponente predstavljaju skladišta podataka (eng. *data storages*) koje čuvaju njihove karakteristike. Na primjer:

- RigidBody komponenta - sadrži podatke o masi, omeđujućem volumenu i inerciji objekta
- Transform komponenta - sadrži podatke o poziciji, rotaciji i skali (eng. *scale*), te matrici transformacija objekta
- MeshRenderer komponenta - definira logiku za iscrtavanje objekta, sadrži podatke o svim točkama, bridovima i materijalima od kojih je objekt napravljen

6.1. Stvaranje fizičkih objekata

Da bi na objekt djelovale sile, te da bi bio dio fizičkog svijeta, potrebno mu je dodati `RigidBody` komponentu. Radi se o komponenti koja sadrži podatke poput mase, inercije, reference na sudarač (eng. *collider*). U komponenti su također definirani svi podaci potrebni za izradu krutog tijela koje se koristi u simulaciji:

– `btDefaultMotionState` - sinkronizira fizičko stanje tijela (položaj, rotacija) s grafičkim prikazom

– `btRigidBody::btRigidBodyConstructionInfo` - struktura s parametrima potrebnim da se napravi *btRigidBody*

– `btRigidBody` - stvarno kruto tijelo koje se ubacuje u simulaciju (ono što engine koristi za sve fizičke proračune)

Do inicijalizacije navedenih članskih varijabli (eng. *member variables*) dolazi kod dodavanja *RigidBody* komponente na *Object* pomoću funkcije `Object::addComponent()`. To je funkcija koja služi za dodavanje komponenti *SceneEntity*-u (kojeg *Object* nasljeđuje). Preduvjet dodavanja *RigidBody* komponente *Object*-u je postojanje *Collider* komponente, jer je ona potrebna kod konstrukcije `btRigidBody::btRigidBodyConstructionInfo`. Prilikom dovršavanja izgradnje (eng. *finalization*), *RigidBody* komponenta preko zajedničkog roditelja (*Object* na kojem se nalazi), dohvaća podatke od bratskih (eng. *sibling*) komponenti *Transform* i *Collider* kako bi mogla izgraditi članske varijable.

```
1  void RigidBody::finalizeRigidBody()
2  {
3      //preventing multiple initializations
4      if (m_rigidBody) return;
5
6      SceneEntity* parentObject = getParentObject();
7
8      if (!parentObject) throw std::runtime_error("Parent object is
           null");
9
10     Transform* t = parentObject->getComponent<Transform>();
11     m_collider = parentObject->getComponent<Collider>();
12
13     if (!t || !m_collider)
14         throw std::runtime_error("Transform or Collider is null");
15
16     m_inertia = calculateInertia();
17     m_motionState =
18         std::make_unique<btDefaultMotionState>
19         (btTransform(t->getBulletQuat(), t->getBulletPosition()));
20     m_rigidBodyCI = btRigidBody::btRigidBodyConstructionInfo(
21         m_mass, m_motionState.get(), m_collider->getCollisionShape()
               , m_inertia);
22     m_rigidBody = std::make_unique<btRigidBody>(m_rigidBodyCI);
```

```
23
24      //allows you to store object pointer in rigidbody - that way you
            can retrive
25      //information about object (if for example rigidbody was hit
            with a ray, you can retrive
26      //information about object to which that rigidbody belongs to)
27      m_rigidBody->setUserPointer(parentObject);
28      setRestitution(m_restitution);
29  }
```

Nakon što je u dovršena izgradnja *RigidBody* komponente, zakazuje joj se dodavanje u fizički svijet pomoću `RigidBodyRegistry::queueRigidBody()` funkcije. Ta funkcija dodaje pokazivač na `btRigidBody` u svoj red čekanja (eng. *queue*), iz kojeg fizički svijet *PhysicsWorld* čita svaki vremenski okvir kako bi utvrdio ima li novih krutih tijela koje je potrebno dodati (ako postoje tijela u redu čekanja, dodaju se u fizički svijet).

```
1   void Object::addComponent(std::unique_ptr<Component> component)
2   {
3       //check if component is null
4       if (!component)
5           throw std::runtime_error("Component is null");
6       //check if component already exists
7       for (auto& existingComponent : m_components)
8       {
9           if (existingComponent.get() == component.get())
10              throw std::runtime_error("Component already exists");
11      }
12
13      m_components.push_back(std::move(component));
14      m_components.back().get()->setParentObject(this);
15
16      if (auto* casted = dynamic_cast<Collider*>(m_components.back().
            get()))
17      {
18          casted->alignBoundsToObject();
19      }
20      //if currently added component is RigidBody, finalize it
21      if (auto* casted = dynamic_cast<RigidBody*>(m_components.back().
            get()))
22      {
23          if (!getComponent<Collider>())
24              throw std::runtime_error("Can't initialize RigidBody
```

```
                without collider");
25          casted->finalizeRigidBody();
26          RigidBodyRegistry::queueRigidBody(casted, 1);
27      }
28  }
```

Kada fizički svijet napokon doda kruto tijelo, objekt će postati fizički objekt, te će na njega djelovati sile, te će biti u interakciji s ostalim fizičkim objektima

6.2. Vizualni prikaz krpene lutke

S obzirom na to da Bullet Physics kao rezultat vraća ažurirane transformacije objekata (a ne informaciju o obliku/mreži objekta koji se simulira), postavlja se pitanje kako iscrtati primitive koje omeđuju objekte (kako bi se moglo promatrati ponašanje onoga što se simulira). Da bi se to postiglo, potrebno je najprije definirati klasu koja implementira `btIDebugDraw` sučelje. Da bi proizvoljna klasa mogla implementirati sučelje, potrebno je "nadjačati/zamijeniti funkcionalnost" (eng. *override*) njenih apstraktnih funkcija. Jedina funkcija u kojoj je potrebno (u kontekstu ovog projekta) definirati funkcionalnost (kako bi se omogućilo iscrtavanje primitiva) je `drawLine()`.

```
1   class MyBulletDebugDrawer : public btIDebugDraw
2   {
3   private:
4       int m_mode;
5       std::vector<glm::vec3> m_linesPoints;
6   public:
7
8       MyBulletDebugDrawer() :
9           m_mode { DBG_DrawWireframe }
10      {
11          m_linesPoints.reserve(200);
12      }
13
14      void drawLine(const btVector3& from, const btVector3& to, const
            btVector3& color) override
15      {
16          m_linesPoints.emplace_back(glm::vec3(from.getX(), from.getY
                (), from.getZ()));
17          m_linesPoints.emplace_back(glm::vec3(to.getX(), to.getY(),
                to.getZ()));
18      }
```

```
19
20      void drawContactPoint(const btVector3& PointOnB, const btVector3
            & normalOnB,
21          btScalar distance, int lifeTime, const btVector3& color)
                override {}
22      void reportErrorWarning(const char* warningString) override {}
23      void draw3dText(const btVector3& location, const char*
            textString) override {}
24      void setDebugMode(int mode) override { m_mode = mode; }
25      int getDebugMode() const override { return m_mode; }
26
27      void resetLines()
28      {
29          m_linesPoints.clear();
30      }
31      std::vector<glm::vec3>* getLinesPoints() { return &m_linesPoints
            ; }
32  };
```

Nakon što je definirana klasa koja implementira sučelje `btIDebugDraw`, izrađena je klasa `BulletGizmos`, koja služi za ažuriranje podataka o točkama koje čine mrežu primitiva, te za iscrtavanje primitiva u žičanoj formi (eng. *wireframe*). Svaki vremenski okvir (eng. *frame*) poziva se funkcija `updateBufferContent()`, koja služi za ažuriranje podataka o točkama primitiva.

```
1   void BulletGizmos::updateBufferContent()
2   {
3       if (!m_physicsWorld)
4           return;
5       //clear info about lines from the previous frame
6       m_bulletDebugDrawer.resetLines();
7       m_physicsWorld->getDynamicsWorld()->debugDrawWorld();
8       const std::vector<glm::vec3>& lines = *m_bulletDebugDrawer.
            getLinesPoints();
9       if (lines.size() == 0) return;
10      //VBO's can't handle store more information - new VBO with more
            capacity has to be initialized
11      if ((m_VBO.m_storageCapacity / sizeof(glm::vec3)) < lines.size()
            )
12      {
13          size_t currentCapacity = m_VBO.m_storageCapacity;
14          m_VBO.Delete();
15
```

```
16          //make another VBO that has increased capacity
17          m_VBO = VBO(1.5 * m_VBO.m_storageCapacity);
18
19          m_VAO.LinkVBO(m_VBO, 0, 3, sizeof(glm::vec3), 0);
20
21          m_VAO.Unbind();
22          m_VBO.Unbind();
23          std::cout << "RESIZED VBO" << '\n';
24      }
25      m_VBO.Bind();
26      void* ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
27      memcpy(ptr, lines.data(), lines.size() * sizeof(glm::vec3));
28      glUnmapBuffer(GL_ARRAY_BUFFER);
29      m_VBO.Unbind();
30  }
```

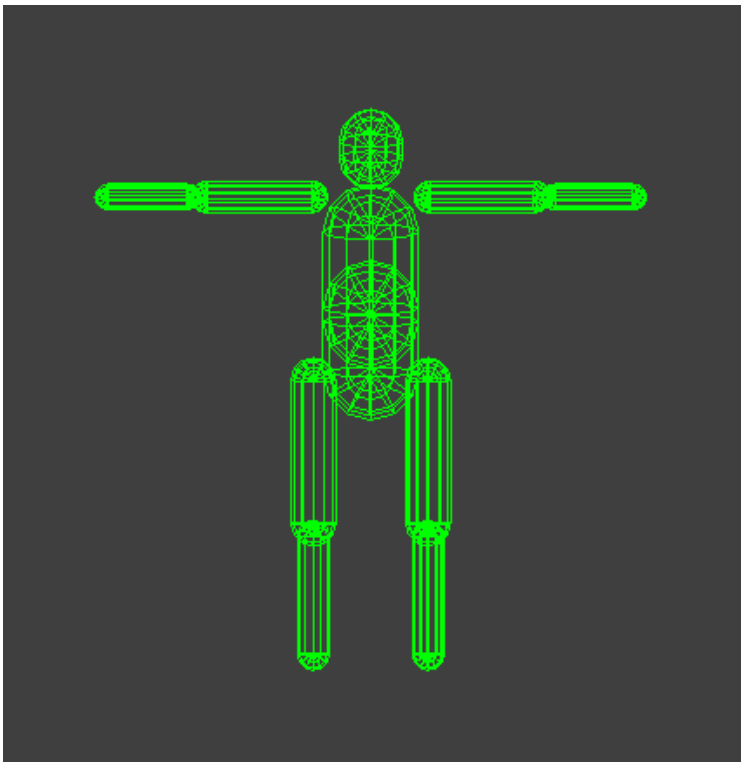
Pozivom funkcije `btDiscreteDynamicsWorld::debugDrawWorld()`, dolazi do iscrtavanja fizičkog svijeta (funkcije za iscrtavanje definirane su u sučelju `btIDebugDraw`, odnosno u klasi koja ga nasljeđuje) [7]. To znači da se poziva funkcija iscrtavanja za svaku primitivu koja je dio fizičkog svijeta (za kvadar će se pozvati funkcija `MyBulletDebugDrawer::drawBox()`, dok će se za kapsulu pozvati funkcija `MyBulletDebugDrawer::drawCapsule()`).

```
1   virtual void drawBox(const btVector3 &bbMin, const btVector3 &bbMax,
        const btVector3 &color)
2   {
3       drawLine(btVector3(bbMin[0], bbMin[1], bbMin[2]), btVector3(
            bbMax[0], bbMin[1], bbMin[2]), color);
4       drawLine(btVector3(bbMax[0], bbMin[1], bbMin[2]), btVector3(
            bbMax[0], bbMax[1], bbMin[2]), color);
5       drawLine(btVector3(bbMax[0], bbMax[1], bbMin[2]), btVector3(
            bbMin[0], bbMax[1], bbMin[2]), color);
6       drawLine(btVector3(bbMin[0], bbMax[1], bbMin[2]), btVector3(
            bbMin[0], bbMin[1], bbMin[2]), color);
7       drawLine(btVector3(bbMin[0], bbMin[1], bbMin[2]), btVector3(
            bbMin[0], bbMin[1], bbMax[2]), color);
8       drawLine(btVector3(bbMax[0], bbMin[1], bbMin[2]), btVector3(
            bbMax[0], bbMin[1], bbMax[2]), color);
9       drawLine(btVector3(bbMax[0], bbMax[1], bbMin[2]), btVector3(
            bbMax[0], bbMax[1], bbMax[2]), color);
10      drawLine(btVector3(bbMin[0], bbMax[1], bbMin[2]), btVector3(
            bbMin[0], bbMax[1], bbMax[2]), color);
```

```
11      drawLine(btVector3(bbMin[0], bbMin[1], bbMax[2]), btVector3(
            bbMax[0], bbMin[1], bbMax[2]), color);
12      drawLine(btVector3(bbMax[0], bbMin[1], bbMax[2]), btVector3(
            bbMax[0], bbMax[1], bbMax[2]), color);
13      drawLine(btVector3(bbMax[0], bbMax[1], bbMax[2]), btVector3(
            bbMin[0], bbMax[1], bbMax[2]), color);
14      drawLine(btVector3(bbMin[0], bbMax[1], bbMax[2]), btVector3(
            bbMin[0], bbMin[1], bbMax[2]), color);
15  }
```

Listing 6.1: Implementacija btIDebugDraw::drawBox preuzeta iz dokumentacije: `https://pybullet.org/Bullet/BulletFull/btIDebugDraw_8h_source.html`

Svaka od tih funkcija u sebi poziva funkciju `drawLine()`, te je sada vidljivo zašto je baš ta funkcija trebala biti "nadjačana". Funkcija `drawLine()` samo uzima početnu i završnu točku brida, te ih sprema u zajedničku strukturu podataka (`std::vector`). Nakon što je funkcija `btDiscreteDynamicsWorld::debugDrawWorld()` gotova s izvršavanjem, u zajedničkoj strukturi podataka će se nalaziti početna i završna točka svih bridova koji čine primitive iz fizičkog svijeta. Ti se podaci najprije pohranjuju u VBO ("vertex buffer object"), te se zatim koriste u funkciji `BulletGizmos::renderColliders()`, koja iscrtava pojedinačne segmente (koristeći primitivu *GL_LINES*)



Slika 6.1: Prikaz iscrtanih primitiva koje čine krpenu lutku

7. Interakcija sa simulacijom

Kako bi simulacija bila što zanimljivija i dinamičnija, te kako bi korisnik bio što više uključen, omogućena je interakcija sa simulacijom. Interakcija je ostvarena pomicanjem krpene lutke. Korisnik može uhvatiti (eng. "*grab*") krpenu lutku za neki od njenih zglobova označavanjem zgloba pritiskom na lijevu tipku miša, te zatim pritiska na tipku "G". Krpena lutka prati poziciju miša sve dok ne dođe do otpuštanja (eng. "*release*").

7.1. Konačni automat

7.1.1. Općenito o konačnom automatu

Konačni automat (eng. *"finite state machine / finite automaton"*) je apstraktna mašina koja u datom trenutku može biti u točno jednom stanju (od njih konačno mnogo). Mogući su prijelazi iz jednog u drugo stanje, potaknuti ulaznim parametrima [8]. Takav prijelaz zove se "tranzicija". Konačni automat definiran je listom stanja, inicijalnim stanjem (iz kojeg sve počinje) i ulaznim parametrima koji pokreću tranzicije. Postoje dvije vrste konačnih automata: deterministička (svaki prijelaz jedinstveno je definiran ulaznim parametrima - nemoguće je napraviti tranziciju između dva stanja, u istom smjeru, koristeći različite ulazne parametre) i nedeterministička (isti prijelaz može biti ostvaren za različite ulazne parametre).

7.1.2. Implementacija determinističkog konačnog automata

Ponašanje grafičkog sustava određeno je stanjem u kojem se nalazi. Broj tih stanja je konačan, a neka od njih su:

- *SELECTED (stanje u kojem postoji odabran (na koji je korisnik kliknuo) objekt)*
- *GRAB (stanje za pomicanje odabranog objekta)*
- *ROTATE (stanje za rotiranje odabranog objekta)*

– *CAMERA_MOVE (stanje u kojem se pomiče kamera)*

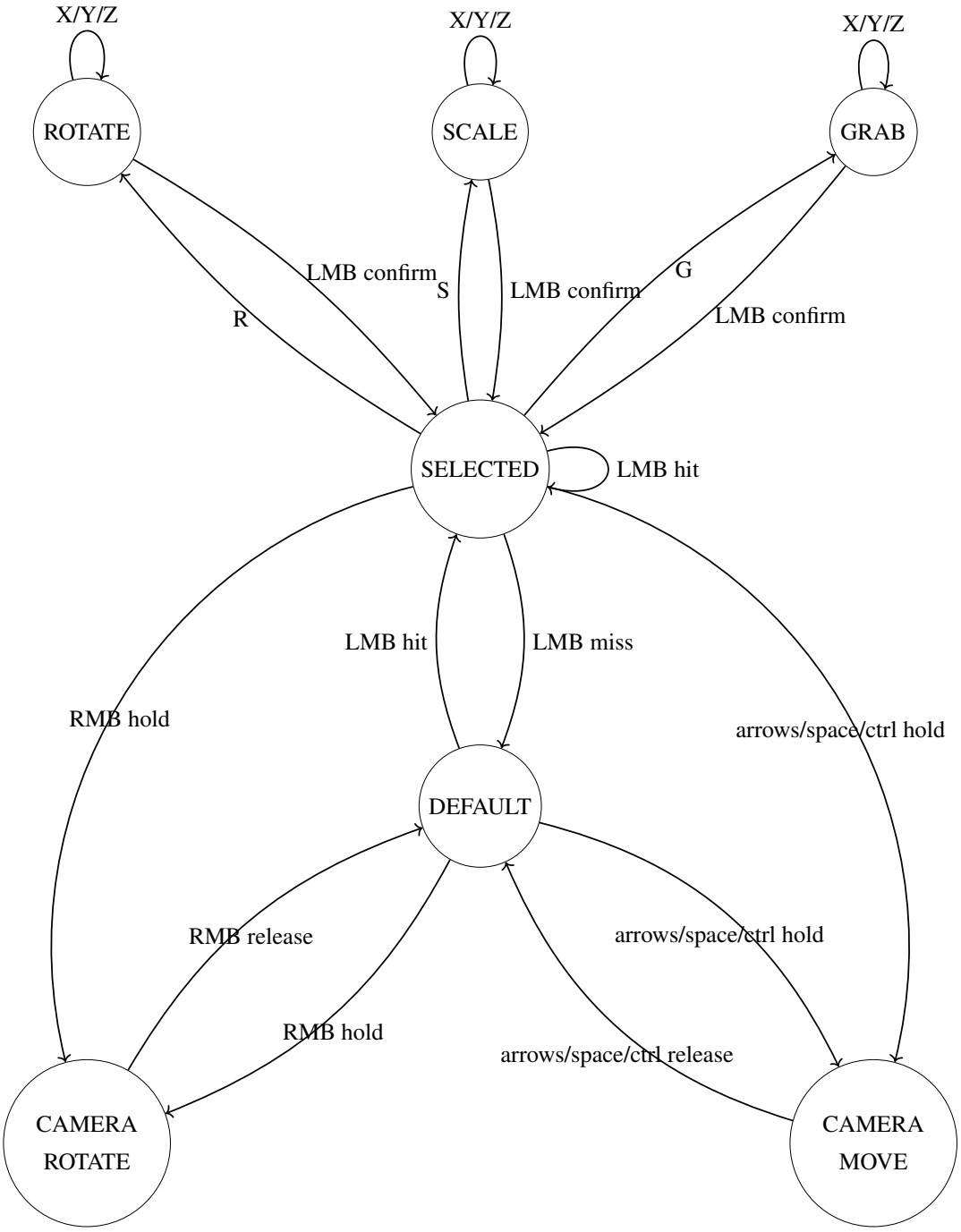
Svako stanje nasljeđuje klasu *State*, koja sadrži podatak o tranzitnom stanju (stanje u koje je potrebno prijeći - *NO_TRANSITION* ako je potrebno ostati u istom stanju) i funkcijama koje se "odazivaju" (eng. "*callback*") na određene događaje. Događaji su generirani od strane tipkovnice i miša, a radi se o događajima za detekciju promjene pozicije miša, pritisak tipke miša, te pritisak tipke na tipkovnici. Od svakog stanja se očekuje da će nadjačati (eng. *"override"*) funkcije definirane u baznoj klasi (*State*) i prilagoditi ih svojim potrebama. Korištenjem navedenog pristupa moguće je ostvariti različito ponašanje za iste događaje (eng. *"event"*), ovisno o stanju u kojem se aplikacija nalazi. Na primjer: pritisak na tipku "G" će se ignorirati ako se aplikacija ne nalazi u stanju *SELECTED*, no ako se u njemu nalazi, to će označiti potrebu za prelazak u stanje *GRAB*, te će doći do provođenja izračuna parametara ravnine u kojoj se kamera nalazi. Stanjima upravlja klasa *StateMachine*, koja predstavlja jednostavni deterministički konačni automat. Odgovorna je za promjenu stanja i upravljanje aktivnim stanjem. Ova klasa prima podatke o generiranim događajima, te na temelju tih informacija, poziva odgovarajuću funkciju iz aktivnog stanja. Također se brine za to da svako stanje "počisti iza sebe", odnosno poziva pripadajuću funkciju *exit()*, koja je odgovorna za obavljanje logike prilikom izlaska iz stanja. Isto tako, nakon što je došlo do promjene stanja, poziva funkciju *enter()*, koja služi za postavljanje stanja. Izgled automata je prikazan pomoću grafa 7.1

```
enum States
{
    NO_TRANSITION,
    DEFAULT,
    IDLE,
    SELECTED,
    GRAB,
    ROTATE,
    SCALE,
    CAMERA_MOVE,
    CAMERA_ROTATE,
    ADD,
    DELETE
};

class State
{
protected:
    States m_transitionState = States::NO_TRANSITION;
```

```
20      StateMachine* m_stateMachine;
21  public:
22      State(StateMachine* stateMachine, bool convertTo3D = false) :
23          m_stateMachine{ stateMachine }, m_convertMouseTo3D {
                convertTo3D }
24      {
25      }
26      bool m_convertMouseTo3D{ false };
27      bool m_activeRigid{ false };
28      /*bool m_movingCamera{ false };*/
29      virtual void enter() {};
30      virtual void exit() {};
31      virtual void update() {};
32      virtual void onMouseMove(double x, double y) {};
33      virtual void onMouseMove(const glm::vec3& mouseStartWorld, const
            glm::vec3& mouseDirectionWorld) {};
34      virtual void onMouseClick(const glm::vec3& mouseStartWorld,
            const glm::vec3& mouseDirectionWorld,
35          int button, int action);
36      virtual void onKeyboardPress(int key, int action);
37      //Sorts objects using 2 conditions: by layer (starting from
            those that have higher priority layer),
38      //and by distance (those closer to ray start have advantage)
39      void sortObjects(std::vector<Hit>& hits, const glm::vec3& start)
            ;
40      void updateSelection(const std::vector<Hit>& hits, btRigidBody*
            rb);
41      States getTransitionState() { return m_transitionState; }
42      virtual ~State() {};
43  };
```

Listing 7.1: State.h datoteka



Slika 7.1: Prikaz konačnog automata. Legenda: LMB = Left Mouse Button, RMB = Right Mouse Button

7.2. Interakcija sa zrakom

Ako se konačni automat nalazi u stanju *DEFAULT*, znači da ne postoji odabrani (eng. *"selected"*) objekt, te da je klasa *State* (klasa koju ostale klase stanja nasljeđuju) aktivna

klasa. Pritiskom na lijevu tipku miša poziva se njena verzija *onMouseClick* funkcije. U funkciji se inicijalizira vektor *Hit*-ova te točka presjeka. *Hit* je struktura koja sadrži informaciju o pokazivaču na *SceneEntity* objekt koji zraka siječe, te točka u kojoj zraka siječe objekt.

7.2.1. Određivanje presjeka s objektima iz fizičkog svijeta

Nakon inicijalizacije vektora i točke presjeka, poziva se funkcija *PhysicsWorld::getIntersection*, kako bi se pronašao presjek zrake s objektima u fizičkom svijetu (ako postoji). Funkcija *PhysicsWorld::getIntersection* kao parametre prima točke početka i završetka zrake. Početak zrake dobiven je pretvaranjem 2D koordinata klika miša u 3D koordinate svijeta, a završetak zrake dobiven je dodavanjem vektora smjera kamere, pomnoženog s duljinom zrake, na početak zrake. Ta dva parametra, se zajedno s pozivajućom funkcijom (eng. *"callback"*) *btDiscreteDynamicsWorld::ClosestRayResultCallback*, predaju Bullet-ovoj ugrađenoj funkciji *rayTest*, koja provodi provjeru presjeka zrake nad svim objektima u *btCollisionWorld*. Nakon poziva funkcije *rayTest* potrebno je provjeriti da li je došlo do presjeka koristeći funkciju *RayResultCallback::hasHit*. U slučaju da je došlo do presjeka, dohvaća se informacija o kolizijskom objektu koji je pogođen, te se nad tim objektom provodi pretvorba u nadklasu (eng. "upcasting"), kako bi se dobio pokazivač na objekt tipa *btRigidBody*. Na kraju je potrebno izvući podatke o točki presjeka, i te dvije informacije zapakirati u strukturu *BulletRayHit*, te ih vratiti kao rezultat poziva funkcije *PhysicsWorld::getIntersection*.

```
1  BulletRayHit PhysicsWorld::getIntersection(const glm::vec3& start,
       const glm::vec3& end)
2  {
3      btCollisionWorld::ClosestRayResultCallback rayCallback(rayFrom,
           rayTo);
4      mDynamicsWorld->rayTest(rayFrom, rayTo, rayCallback);
5      if (rayCallback.hasHit())
6      {
7          btRigidBody* hit = (btRigidBody*)btRigidBody::upcast(
               rayCallback.m_collisionObject);
8          glm::vec3 hitPoint = glm::vec3(rayCallback.m_hitPointWorld.
               getX(),
9              rayCallback.m_hitPointWorld.getY(), rayCallback.
                   m_hitPointWorld.getZ());
10         return BulletRayHit{ hit, hitPoint };
11     }
12     else return BulletRayHit{ nullptr, glm::vec3(0.0f, 0.0f, 0.0f)
```

```
        };
13  }
```

Listing 7.2: Funkcija za određivanje presjeka nad objektima iz fizičkog svijeta

7.2.2. Određivanje presjeka s objektima koji nisu dio fizičkog svijeta

Nakon što je proveden postupak za pronalaženje presjeka objekata iz fizičkog svijeta sa zrakom, potrebno je istu stvar provjeriti za objekte koji nisu dio tog svijeta. Kako bi se obavila ta provjera, potrebno je nad svakim takvim objektom pozvati funkciju *EditorCollider::intersects*. Svaki nefizički objekt posjeduje *EditorCollider* komponentu, u kojoj je definiran OBB omeđujući volumen tog objekta. Nad tim omeđujućim volumenima se provodi provjera presjeka zrake. Mali nedostatak ovog pristupa je taj što ti volumeni često imaju dosta "praznog prostora" (u odnosu na konveksnu ljusku ili hijerarhiju više omeđujućih volumena pomoću kojih se objekt aproksimira), odnosno postoji prostor unutar volumena u kojem se ne nalazi dio tijela objekta (volumen široko obuhvaća objekt). OBB volumen se koristi unatoč ovom nedostatku radi jednostavnosti, te dovoljno dobre preciznosti za trenutačne zahtjeve. Za detekciju presjeka omeđujućeg volumena sa zrakom koristi se tzv. "metoda plohe" (eng. *"slab method"*). S obzirom na to da taj algoritam radi s omeđujućim volumenima koji nemaju rotaciju (AABB), potrebno je zraku prebaciti u lokalni sustav omeđujućeg volumena. To je postignuto množenjem početne točke zrake i njenog smjera s inverznom matricom transformacije objekta. Nakon provođenja te transformacije, moguće je koristiti originalnu implementaciju "slab" metode, bez potrebe za daljnjim modifikacijama. Prije povratka iz funkcije potrebno je točku presjeka prebaciti iz lokalnog koordinatnog sustava u sustav svijeta množenjem s matricom transformacije objekta.

Metoda plohe

Metoda plohe se u računalnoj grafici koristi za rješavanje problema presjeka zrake s kvadrom (eng. "ray-box intersection") za slučaj kada je omeđujući volumen kvadra poravnat s osima (eng. *"axis aligned bounding box" - AABB*). Zbog svoje učinkovitosti, metoda se vrlo često primjenjuje u raznim grafičkim aplikacijama. Ideja algoritma je "odrezati" zraku pomoću 6 stranica koje definiraju omeđujući kvadar [9]. Svaki par paralelnih stranica definira "slab", a volumen unutar kvadra dobiven je kao presjek sva 3 "slab"-a. Trodimenzionalni AABB se može predstaviti pomoću dvije

trojke: $l = (l_0, l_1, l_2)$ i $h = (h_0, h_1, h_2)$, odnosno minimalnih i maksimalnih vrijednosti omeđujućeg volumena u sve tri dimenzije. Proizvoljna točka na zraci s početkom u točki $o = (o_0, o_1, o_2)$ i vektorom smjera $r = (r_0, r_1, r_2)$, može se zapisati u obliku $p(t) = o + t * r$ (*t* je skalar). Rješavanjem jednadžbe, dobiven je sljedeći izraz:

$$t = \frac{p - o}{r}$$

Ta jednadžba može se dalje razložiti na sljedeća dva izraza:

$$t_i^{low} = \frac{l_i - o_i}{r_i}$$

$$t_i^{high} = \frac{h_i - o_i}{r_i}$$

koji predstavljaju presjeke zrake s plohama koje su okomite na i-tu os (x, y ili z). Time su dobivene vrijednosti koeficijenata t, koje nam govore gdje zraka siječe ravnine definirane pomoću 6 stranica kvadra. Bliži i daljnji ekstrem segmenta zrake unutar i-tog "slab"-a, definiran je izrazom:

$$t_i^{close} = min\{t_i^{low}, t_i^{high}\}$$

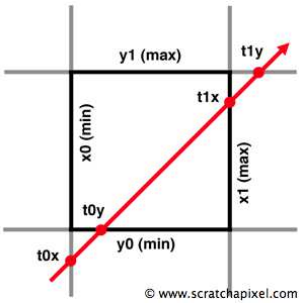
$$t_i^{far} = max\{t_i^{low}, t_i^{high}\}$$

To što zraka siječe ravninu, ne znači nužno da je u presjeku s kvadrom [10] (prikazano na 2D primjeru na slici 7.3). Kako bi se prostor oko zrake suzio što je više moguće, odabire se maksimalna dobivena vrijednost koeficijenta *t* za bliži ekstrem, te minimalna dobivena vrijednost koeficijenta *t* za daljnji ekstrem segmenta zrake. Time je definiran presjek svih segmenata zrake (prostire se između točaka dobivenih uvrštavanjem t^{close} i t^{far} u jednadžbu $p(t) = o + r * t$):

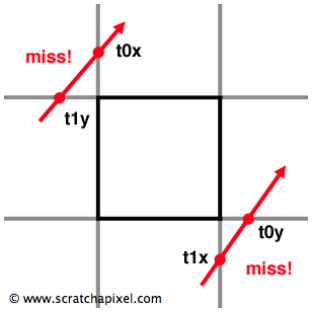
$$t^{close} = \max_i \{t_i^{close}\}$$

$$t^{far} = \min_i \{t_i^{far}\}$$

Presjek između kvadra i zrake postoji samo u slučaju kada vrijedi: $t^{close} \leq t^{far}$.



Slika 7.2: Prikaz zrake koja presijeca kvadrat, izvor: `https://www.scratchapixel.com/images/ray-simple-shapes/2dfig.png`



Slika 7.3: Prikaz zraka koje nisu u presjeku s kvadratom, izvor: https://www.scratchapixel.com/images/ray-simple-shapes/2dfigmiss.png

```
bool EditorCollider::intersects(const glm::vec3& start, const glm::
    vec3& direction, glm::vec3& intersectionOut)
{
    if (!m_parent)
        return false;

    glm::mat4 transform = (&m_parent->m_transform)->getModelMatrix()
        ;
    glm::vec3 _start = glm::vec3(glm::inverse(transform) * glm::vec4
        (start, 1.0f));
    //2nd argument in vec4 has to be 0.0f because _dir is line, not
        a point
    glm::vec3 _dir = glm::vec3(glm::inverse(transform) * glm::vec4(
        direction,0.0f));

    float t_close[3]{ 0.0f };
    float t_far[3]{ 0.0f };

    for (int i = 0; i < 3; i++)
    {
        if (_dir[i] == 0)
            //if the current dimension of direction vector is
                parallel to slab
            //we need to check whether origin of ray is outside of
                box - if it is,
            //it will never intersect with the box
        {
            if (_start[i] < m_min[i] || _start[i] > m_max[i])
                return false;
        }
        //avoid calculating if direction[i] == 0 to avoid division
            by 0
```

```
25          else
26          {
27              float t_i_low = (m_min[i] - _start[i]) / _dir[i];
28              float t_i_high = (m_max[i] - _start[i]) / _dir[i];
29              t_close[i] = fmin(t_i_low, t_i_high);
30              t_far[i] = fmax(t_i_low, t_i_high);
31          }
32      }
33
34      float t_close_float = *std::max_element(std::begin(t_close), std
            ::end(t_close));
35      float t_far_float = *std::min_element(std::begin(t_far), std::
            end(t_far));
36
37      //Transform intersection point back to world space (_start and
            _dir will give intersection in local
38      //space of the object)
39      intersectionOut = glm::vec3(transform * glm::vec4(_start +
            t_close_float * _dir, 1.0f));
40      return t_close_float <= t_far_float && t_far_float > 0;
41
42  }
```

Listing 7.3: Detekcija presjeka omeđujućeg volumena sa zrakom pomoću algoritma plohe

7.3. Odabir aktivnog objekta

Svaki nefizički objekt, koji je u presjeku sa zrakom, dodaje se u vektor *hits*. Vektor sadrži strukture tipa *Hit*, koja čuva podatak o pokazivaču na objekt koji je u presjeku sa zrakom, te točki presjeka. Potom se elementi vektora *hits*, pomoću funkcije *std::sort*, sortiraju prema udaljenosti od kamere te sloju (eng. *"layer"*) u kojem se nalaze (sloj ima veći prioritet). Na kraju je potrebno usporediti rezultate obje provjere (rezultantne nefizičke i fizičke objekte), te kao aktivan objekt (nad kojim se izvode transformacije) postaviti onaj koji se nalazi bliže kameri.

7.4. Pomicanje objekta i "stanje hvata"

Objekt je moguće pomaknuti jedino ako se konačni automat nalazi u stanju hvata (eng. *"grab state"*). U to je stanje moguće prijeći ako je zadovoljen sljedeći uvjet: tipka

"G" je pritisnuta dok se automat nalazi u stanju *State::SELECTED*. Kod inicijalizacije stanja hvata, poziva se konstruktor klase *TransformState*, koju nasljeđuju ostale klase transformacije: *GrabState*, *RotateState* i *ScaleState*. U njenom konstruktoru se inicijalizira ravnina u kojoj leži aktivni objekt. Za definiranje normale ravnine koristi se vektor suprotan vektoru smjera kamere, a za točku koja se nalazi u ravnini, koristi se pozicija aktivnog objekta (postavlja se kao ishodište ravnine). Ti podaci se uvrštavaju u jednadžbu ravnine:

$$Ax + By + Cz + D = 0$$

$$\vec{n} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$T = \begin{pmatrix} A, & B, & C \end{pmatrix}$$

Pomoću tih informacija moguće je izračunati koeficijent D, koji je bio jedina nepoznanica. Dok je konačni automat u stanju hvata, objekt prati poziciju miša (koju je potrebno konstantno pretvarati iz 2D koordinata zaslona u 3D koordinate svijeta). Da bi se očuvala dubina (z-koordinata) pri pomicanju objekta, potrebno je pronaći presjek zrake, ispaljene iz 3D koordinate miša (u smjeru u kojem kamera gleda) i ravnine u kojoj se objekt nalazi. Budući da je ravnina orijentirana prema kameri, presjek zrake s ravninom daje točku koja je uvijek na istoj dubini relativno na kameru kao i originalan objekt. Ovime je osigurano da neće dolaziti do "propadanja objekta" dublje u scenu, niti će objekt prilaziti bliže od očekivanog - osigurano je intuitivno ponašanje pri pomicanju objekta. Objekt je također moguće pomicati isključivo po jednoj od tri globalne osi (x, y i z os u sustavu svijeta, a ne u lokalnom sustavu). Ideja je najprije projicirati globalnu os na ravninu u kojoj se objekt nalazi. Nakon toga je potrebno projicirati 3D poziciju miša na prethodno projiciranu ravninu, te odrediti udaljenost ishodišta ravnine i trenutačno projicirane točke. To je udaljenost za koju je potrebno pomaknuti tijelo po u smjeru zadane globalne osi (u obzir treba uzeti i smjer u kojem se miš pomaknuo u odnosu na ishodište ravnine). Time bi se ostvarilo sljedeće ponašanje: Zamislimo slučaj kada kamera gleda u smjeru Z-osi, te njezin "up" vektor gleda u smjeru pozitivne Y-osi, a korisnik želi pomicati objekt po globalnoj X-osi. Poželjno je da kada korisnik miče miš gore-dolje, nema pomaka, a kada ga miče u stranu, da se objekt pomiče. Inspiracija za pomicanje po osima je preuzeta iz grafičkog alata Blender, te je pomoću obrnutog inženjerstva (eng. *"reverse engineering"*) ostvarena. Prvi korak u implementaciji je određivanje osi po kojoj će se objekt pomicati. To je ostvarivo pritiskom na neku od tipki: "X", "Y" ili "Z", dok se konačni automat nalazi u stanju koje

nasljeđuje *TransformState*. Tada se u vektore *m_projectedAxis* i *m_infiniteAxis* pohranjuje vrijednost odabrane globalne osi: $\vec{x} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $\vec{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ ili $\vec{z} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$. Potom se pomoću funkcije *TransformPlane::projectVectorToPlane* vektor *m_projectedAxis* projicira na ravninu. Da bi se pomoću tog vektora dobila os, potrebno ga je "rastegnuti u oba smjera", odnosno napraviti novi vektor, koji će predstavljati os. "Os" je definirana sljedećim izrazom:

$$B = m\_transformPlane.m\_origin + m\_projectedAxis * length$$

$$A = m\_transformPlane.m\_origin - m\_projectedAxis * length$$

$$\vec{AB} = B - A$$

, *m_transformPlane.m_origin* predstavlja ishodište ravnine, a *length* je proizvoljna duljina za koju će vektor biti "razvučen" u oba smjera. Sada kada je globalna os preslikana na ravninu, potrebno je samo preslikati točku na tu os, odrediti udaljenost te točke od ishodišta ravnine, te pomaknuti objekt za tu udaljenost u smjeru globalne osi. Pritom je potrebno paziti na predznak, odnosno smjer u kojem je potrebno pomaknuti tijelo. Predznak se određuje pomoću skalarnog umnoška vektora smjera preslikane osi, te vektora koji se pruža od ishodišta ravnine prema projiciranoj točki.

7.4.1. Postavljanje ograničenja "od točke do točke" (p2p)

Budući da pozicijom fizičkih objekata u potpunosti upravlja fizički pogon, potrebno je dodati par koraka, kako bi pristup opisan u prošlom poglavlju funkcionirao. Radi toga je potrebno iskoristiti takozvano ograničenje "od točke do točke" (eng. *"point to point constraint"*). Ograničenje od točke do točke osigurava da se središnje točke (eng. *"pivot point"*) nalaze na istoj poziciji u sustavu fizičkog svijeta. U simulaciji se primjenjuju različiti impulsi na tijela povezana tim ograničenjem, kako bi se njihove središnje točke dovele u istu (zajedničku) točku. To je svojstvo pogodno za primjenu kod pomicanja fizičkih objekata praćenjem pozicije miša (eng. *"mouse picking"*). Proces počinje u funkciji *TransformState::enter*. Tu dolazi do inicijalizacije podataka potrebnih za stvaranje ograničenja, kao što su pokazivač na fizički svijet i fizički objekt, te točka presjeka zrake s fizičkim objektom (*StateMachine::m_rbPickPos*). Te su varijable definirane u klasi *RigidBodyPicker*, u kojoj je implementirana logika za dodavanjem i micanjem ograničenja, te pomicanjem fizičkog objekta. Nakon inicijalizacije varijabli, poziva se funkcija *RigidBodyPicker::pickBody*, pomoću koje se definira ograničenje od

točke do točke. Koristi se varijanta p2p ograničenja kojoj je potrebna samo informacija o pokazivaču i središnjoj točki krutog tijela nad kojim će se raditi transformacije. To tijelo se pomoću ograničenja ne veže za drugo kruto tijelo, već za fiksnu točku u fizičkom svijetu. Zahvaljujući ograničenju, odabrano kruto tijelo prati poziciju središnje točke *PivotB*, te je stoga dovoljno toj točki zadati novu poziciju kako bi se kruto tijelo, a samim time i cijela krpena lutka (pošto su kruta tijela koja ju čine međusobno povezana ograničenjima), pomaknula. Logika za pomicanje krutog tijela definirana je u funkciji *RigidBodyPicker::movePickedBody*. Sve što je potrebno za određivanje nove pozicije krutog tijela su podaci o inicijalnoj udaljenosti presjeka (krutog tijela sa zrakom) od ishodišta zrake (podatak je pohranjen u varijablu *RigidBodyPicker::m_oldPickingDist* tijekom poziva funkcije *RigidBodyPicker::pickBody()*) i trenutačni smjer zrake. Nova pozicija krutog tijela dobivena je pomakom duljine *m_oldPickingDist* od ishodišta zrake u smjeru zrake (linija 12). Time je osigurano da će se kruto tijelo uvijek nalaziti na istoj dubini relativno na kameru. Budući da bi pomicanje ili rotiranje kamere promijenilo aktivno stanje konačnog automata (te samim time izazvalo izlazak iz stanja hvata - završetak pomicanja krutog tijela), nije potrebno brinuti o tome da će vektor pomaka biti prekratak ili predug. Konačno, potrebno je omogućiti otpuštanje "uhvaćenog" krutog tijela (u svrhu hvatanja novog). To je ostvareno micanjem p2p ograničenja (funkcija *RigidBodyPicker::removePickingConstraint()*).

```
1  void movePickedBody(const btVector3& rayFromWorld, const btVector3&
       rayToWorld)
2   {
3       if (m_pickedBody && m_pickedConstraint)
4       {
5           btPoint2PointConstraint* pickCon = static_cast<
              btPoint2PointConstraint*>(m_pickedConstraint);
6           if (pickCon)
7           {
8               btVector3 newPivotB;
9               btVector3 dir = rayToWorld - rayFromWorld;
10              dir.normalize();
11              dir *= m_oldPickingDist;
12              newPivotB = rayFromWorld + dir;
13              pickCon->setPivotB(newPivotB);
14          }
15      }
16  }
```

Listing 7.4: Funkcija za pomicanje krutog tijela

8. Mjerenja i testiranje rezultata

8.1. Skripta za prikupljanje rezultata

Za potrebe mjerenja performansi prilikom simuliranja različitog broja krpenih lutaka, stvorena je skripta *automation.py*. Njena je zadaća pročitati sadržaj skripte u kojoj je definirana glavna petlja simulacije (*Main.cpp*), pronaći redak u kojem se definira broj krpenih lutaka koji će se instancirati, te ga modificirati. Skripta za automatizaciju najprije pročita sav sadržaj *Main.cpp* dokumenta, te locira redak u kojem se definira broj krpenih lutaka koji će se generirati. Vrijednosti se pohranjuju u varijable *source_file_content* i *idx*, kako bi se mogle ponovno upotrijebiti, bez potrebe za ponavljanjem ovih koraka. Zatim započinje izvođenje petlje, koja u svakoj iteraciji prepiše sadržaj *Main.cpp* dokumenta, sadržajem koji se razlikuje samo u liniji na poziciji *idx* (linija u kojoj je definiran broj simuliranih krpenih lutaka). Ta je linija izmijenjena pomoću funkcije *modify_line*. Zadaća te funkcije je zamijeniti postojeći broj, koji dolazi iza znaka jednakosti, brojem krpenih lutaka koje je potrebno simulirati.

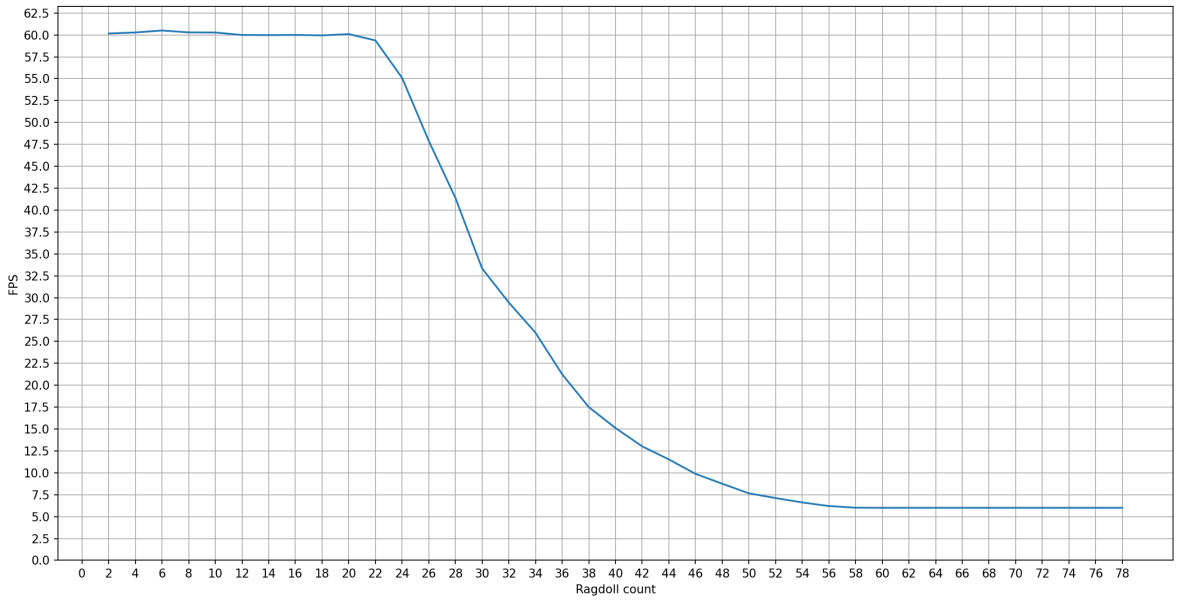
```
1 def modify_line(original_line : str, new_value : str) -> str:
2     modified_line = ""
3     print(f"{original_line=}")
4     original_value = re.search(r"(?<=\=)\s*([^;]+)\s*(?=;)",
          original_line).group(1)
5     start = original_line.find(original_value)
6     modified_line = original_line[:start] + new_value + ";\n"
7     print(f"{modified_line=}")
8     return modified_line
```

Listing 8.1: Funkcija za izmjenu linije koja definira broj krpenih lutaka u simulaciji

Nakon što je odgovarajuća linija izmijenjena, te je sadržaj *Main.cpp* dokumenta prepisan, potrebno je "izgraditi datoteku" (eng. "build a file"), te pokrenuti program. Kako bi se omogućilo izvršavanje paralelnih procesa (koji će izgraditi datoteku i pokrenuti izvršavanje simulacije), potrebno je stvoriti "procese potomke" (eng. "child proce-

sses"). U svakoj iteraciji se stvara jedan potomak koji će izgraditi datoteku, te nakon njega potomak koji će pokrenuti simulaciju. Proces koji je pokrenuo simulaciju, također osluškuje "standardni izlaz" (eng. "standard output") pokrenute aplikacije, jer će se u njega (za svaki okvir tijekom trajanja simulacije) ispisivati broj sličica u sekundi. Nakon *WAIT_TIME* sekundi (unaprijed definirano trajanje simulacije), dolazi do stvaranja *quit.flag* dokumenta, što označava potrebu za terminiranjem simulacije (glavna petlja u *Main.cpp* se izvršava sve dok se ne aktivira *glfwWindowShouldClose()* ili dok se ne pojavi dokument *quit.flag*). Na kraju iteracije je potrebno naći aritmetičku sredinu zbroja svih sličica, te ju upisati u rječnik (eng. "dictionary") konačnih rezultata (ključ je broj simuliranih krpenih lutaka, a vrijednost je aritmetička sredina broja sličica u sekundi). Na početku iduće iteracije dolazi do brisanja *quit.flag* dokumenta, kako bi, i u sljedećoj iteraciji, izvođenje simulacije trajalo *WAIT_TIME* sekundi.
Da bi *automation.py* skripta ispravno radila, potrebno ju je pokrenuti putem *Developer Command Prompt for VS <verzija>*, što omogućuje korištenje naredbi poput *devenv* i *\Build*, koje su korištene za izgradnju datoteke.

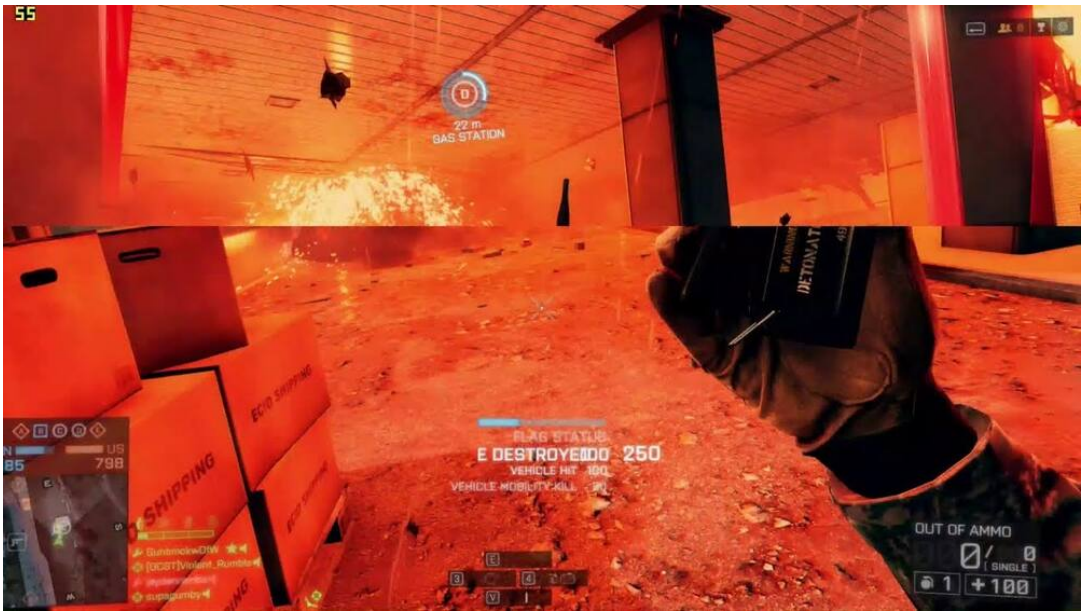
8.2. Rezultati mjerenja



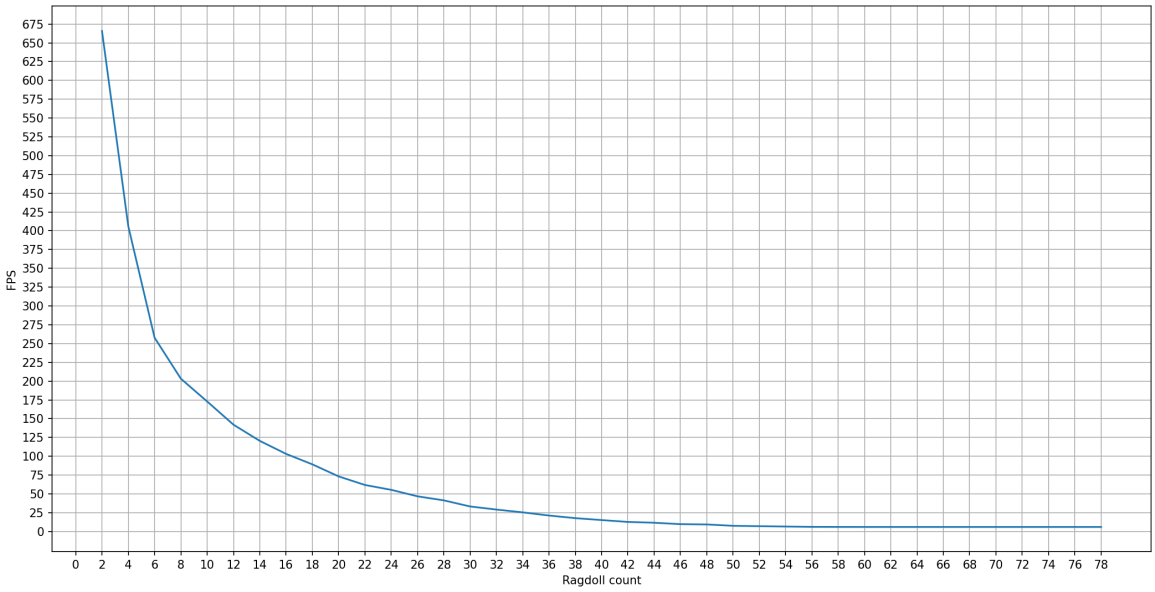
Slika 8.1: Prikaz grafa ovisnosti broja sličica u sekundi (eng. "frames per second" / "fps") o broju simuliranih krpenih lutaka - VSync uključen

Da bi se mjerenja odnosila isključivo na vrijeme trajanja simulacije, odnosno kako ne bi bila ograničena specifikacijama monitora na kojem se simulacija prikazuje, isklju-

čen je *VSync*. *VSync* je grafička postavka koja usklađuje / sinkronizira (eng. "syncs") brzinu kojom grafička kartica generira okvire za prikaz, s brzinom osvježavanja monitora (eng. "refresh rate"). Koristi se kako bi se spriječio problem "trganja ekrana" (eng. "screen tearing") [11]. Problem nastaje kada grafička kartica pošalje novi okvir/sliku, prije nego što je monitor do kraja iscrtao trenutačni okvir. Budući da monitor ažurira piksele red po red, a ne sve odjednom, može doći do istovremenog prikaza sadržaja 2 (ili više okvira). Preciznije, pretpostavimo da monitor osvježava okvir F_1 i da je došao do reda x. Ako u tom trenutku grafička kartica generira okvir F_2 i rezultat zapiše u memorijski međuspremnik (eng. *frame buffer*) iz kojeg monitor čita, sadržaj okvira F_1 bit će prepisan sadržajem okvira F_2. Budući da će monitor samo nastaviti čitati iz međuspremnika, za ostatak redova će koristiti sadržaj okvira F_2. To će rezultirati time da se u redovima $[1, x]$ nalazi sadržaj iz okvira F_1, a u redovima $[x + 1, n]$ (n je zadnji red piksela) sadržaj iz okvira F_2. Rezultat je vizualni artefakt prikazan na slici 8.2. *VSync* postavka ne dozvoljava grafičkoj kartici slanje novih okvira sve dok monitor nije u potpunosti iscrtao trenutačni okvir.



Slika 8.2: Prikaz efekta "trganja ekrana", izvor: `https://www.cyberpowerpc.com/blog/w/wp-content/uploads/2025/08/ScreenTearing.jpg`



Slika 8.3: Prikaz grafa ovisnosti broja sličica u sekundi (eng. "frames per second" / "fps") o broju simuliranih krpenih lutaka - VSync isključen

Slika 8.3 prikazuje mjerenja koja se odnose na slučaj kada je postavka *VSync* isključena.

Na oba grafa je vidljivo da se vrijednosti ne spuštaju ispod 6 FPS-a. Razlog tome je taj što je vrijednost *delta_time* varijable, koja predstavlja vremensku razliku između iscrtavanja dva okvira, "zaključana" na vrijednosti 0.167. To znači da se u slučaju prevelike vremenske razlike (sve što je veće od 0.167s) neće iskoristiti stvarna vrijednost *delta_time*, već će se postaviti u maksimalnu dozvoljenu vrijednost (0.167s). Razlog tog rješenja je taj što bi u slučaju vizualno zahtjevne scene, vrijeme između iscrtavanja dvaju okvira moglo rezultirati velikim vrijednostima *delta_time* varijable. Taj scenarij je problematičan zbog toga što se vrijednost te varijable koristi u simulaciji - njome je određen broj koraka koji će se simulirati u jednom pozivu. Korištenjem velike vrijednosti *delta_time* varijable, simulacija više nije kontinuirana (doima se kao da se objekti teleportiraju jer grafički pogon treba čekati da se simulira veliki broj koraka, prije nego što ponovno dođe red na iscrtavanje). Rezultat takvog pristupa je tzv. "zaostajanje okvira" (eng. "frame lag"). "Zaključavanjem" vrijednosti *delta_link* nije riješen problem smanjenja broja sličica u sekundi ("fps drop"), ali je osigurana "dosljednost" simulacije. Drugim riječima, grafički pogon može dovoljno često iscrtavati scenu, a fizički pogon obavlja simulaciju u malim koracima (međusobno se prate i ne zaostaju jedan za drugim), te samim time nema "skakanja" i "teleportiranja" fizičkih objekata između dva iscrtana okvira.

8.3. Analiza rezultata

Konfiguracija računala na kojem su mjereni rezultati:

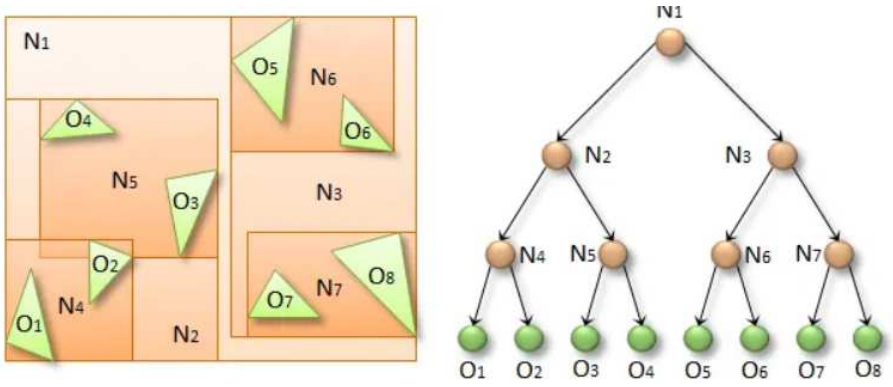
- *CPU*: AMD Ryzen 5 5500
- *GPU*: RTX 2060 6GB
- *RAM*: 16GB RAM (3600MHz)
- *SSD*: Samsung SSD 980 500GB

Rezultati mjerenja pokazuju značajan pad performansi s povećanjem broja simuliranih objekata. Između dvije i četiri lutke pad iznosi 260 fps, dok je pad između četiri i šest lutaka nešto manji (no još uvijek značajan) - oko 150 fps. Pad performansi je nelinearan, te podsjeća na graf recipročne funkcije s asimptotom jednakoj 6. Pri simulaciji približno 50 lutaka sustav pada na 6 fps, što predstavlja donju granicu ispod koje broj okvira u sekundi nikada neće pasti (razlog tom ograničenju je objašnjen u prethodnom poglavlju). Za glatku simulaciju u stvarnom vremenu potrebno je održavati minimalno 60 okvira u sekundi. Mjerenja pokazuju da sustav postiže prag od 60 fps-a pri simuliranju 24 lutke. To znači da je to maksimalan broj lutaka koji možemo simulirati u stvarnom vremenu prije nego počnemo primjećivati zastajkivanje i nedovoljnu brzinu simulacije.

8.4. Optimizacija

Biblioteka Bullet Physics prema zadanima postavkama (eng. *default settings*), kao i većina fizičkih pogona, koristi tehnike optimizacije poput prostorne particije i sustava spavanja/buđenja. Prostorna particija je postignuta korištenjem tzv. BVH stabla [12]. BVH stablo je podatkovna struktura koja se koristi za ubrzanje algoritama detekcije sudara i algoritma praćenja zrake (eng. *ray tracing*). Organizirana je na sljedeći način: u listovima stabla se nalaze omeđujući volumeni objekata. Njihovi roditelji su čvorovi, koji predstavljaju omeđujući volumen koji obujmljuje prostorno bliske (čije su pozicije na međusobno maloj udaljenosti u prostoru) omeđujuće volumene u razini ispod njih samih. Ti se čvorovi rekurzivno grupiraju pod veće omeđujuće volumene, sve dok ne ostane samo jedan čvor, koji obuhvaća volumene svih grupa omeđujućih volumena (a samim time i omeđujuće volumene svih objekata). Sustav spavanja/buđenja omogućava prekid simuliranja dinamičkih krutih tijela za koje je određeno da su u stanju spavanja. Tijelo prelazi u stanje spavanja ako se, tijekom određenog vremena, nije pomaknuo ili se pomiče jako sporo. Da bi se simulacija ponovno počela provoditi

nad tim tijelom, potrebno ga je "probuditi". Do buđenja automatski dolazi u situacijama kada kruto tijelo koje nije u stanju spavanja (eng. *non-sleeping rigid body*) dođe u kontakt s "uspavanim" krutim tijelom.



Slika 8.4: Prikaz BVH stabla, izvor: `https://miro.medium.com/v2/resize:fit:786/format:webp/1*23gT465NHfR1pAvIDiDMCg.png`

8.4.1. Mogućnosti optimizacije

Iako je biblioteka podnijela većinu optimizacijskih mjera, postoje još neka rješenja koja bi se mogla iskoristiti za dodatno ubrzavanje rada simulacije. Neka od tih rješenja su:

- ručno deaktiviranje fizike za objekte koji se nalaze na velikoj udaljenosti od kamere (te ponovno aktiviranje kada se nađu dovoljno blizu kamere)
- dinamičko smanjenje broja iteracija rješavača kada fps pada
- korištenje kolizijskih maski za filtriranje kolizija po slojevima u kojima su kruta tijela definirana
- korištenje paralelizacije - paralelna detekcija kolizije i paralelno izvođenje rješavača ograničenja

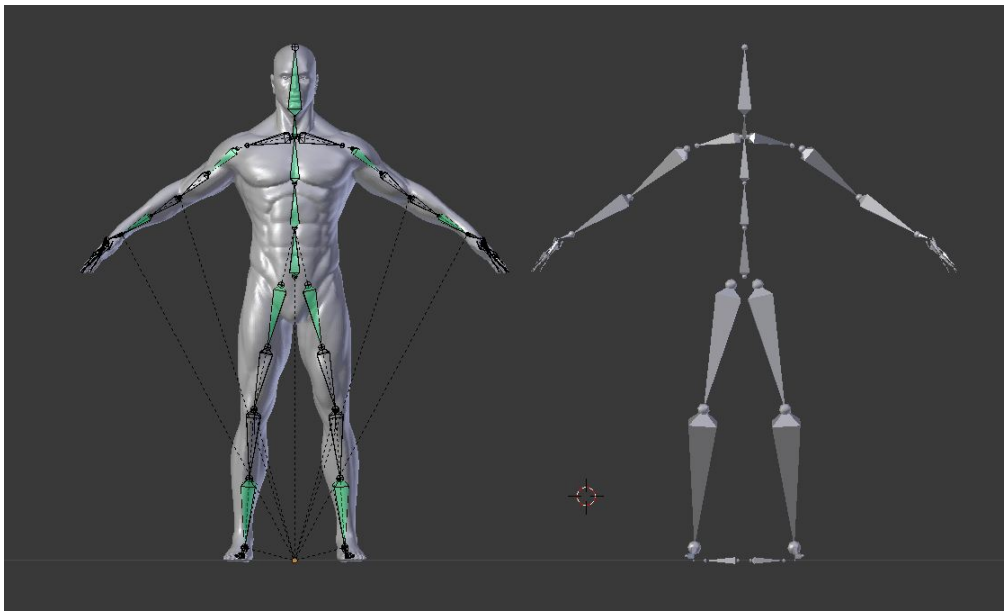
9. Budući rad i nadogradnje

Trenutačna implementacija sustava predstavlja funkcionalnu osnovu za simulaciju fizike krpene lutke, no postoji niz mogućih nadogradnji koje bi unaprijedile vizualnu kvalitetu i korisničko iskustvo.

9.0.1. Povezivanje krpene lutke s 3D modelom

Krpena lutka je u trenutačnoj implementaciji prikazana nizom kapsula, odnosno geometrijskih primitiva koje aproksimiraju oblik tijela. Budući da se krpena lutka koristi kako bi se postojeći humanoidni modeli "oživjeli", te da bi se postiglo da uvjerljivije reagiraju na različite podražaje i sile iz okoline, logično bi bilo simulaciju povezati sa statičnim modelom, te ga tako učiniti dinamičnim. U toj namjeri bi veliku ulogu igrala biblioteka *Assimp*, koja bi poslužila za učitavanje modela i kosti koje su definirane za taj model. Svaka kost sadrži informaciju o količini utjecaja (težina - eng. *weight*) koju ima nad određenom točkom modela. Težine se kreću u rasponu [0,1], a označavaju mjeru u kojoj je potrebno transformirati točku, u ovisnosti o transformaciji kosti: primjerice, ako je težina za određenu točku 0.5, to znači da će se prilikom pomaka kosti za 2m u smjeru x, točka u istom smjeru pomaknuti za 1m. Ovaj primjer opisuje slučaj kada samo jedna kost ima utjecaj nad točkom. Najčešće više kosti ima utjecaj nad nekom točkom, te je stoga potrebno kombinirati transformacije svih kosti da bi se dobila konačna transformacija točke. Da bi došlo do vizualne "deformacije" 3D modela, potrebno je povezati fizičke zglobove krpene lutke s kostima modela. Ovaj proces zahtijeva "mapiranje" svakog zgloba na odgovarajuću kost: primjerice, fizički zglob koji predstavlja lakat povezuje se s kosti nadlaktice u modelu. Mapiranje podrazumijeva primjenu transformacija zglobova (rotacija i translacija), dobivenih iz fizikalne simulacije, na pripadajuće kosti modela, uz poštivanje hijerarhijske strukture (uz primjenu transformacije pripadnog zgloba, potrebno je primijeniti transformaciju svih roditeljskih zglobova). Sustav za *skinning* zatim automatski ažurira pozicije vrhova modela prema težinama kostiju, što rezultira realističnim prikazom deformacije tijela tijekom

simulacije.



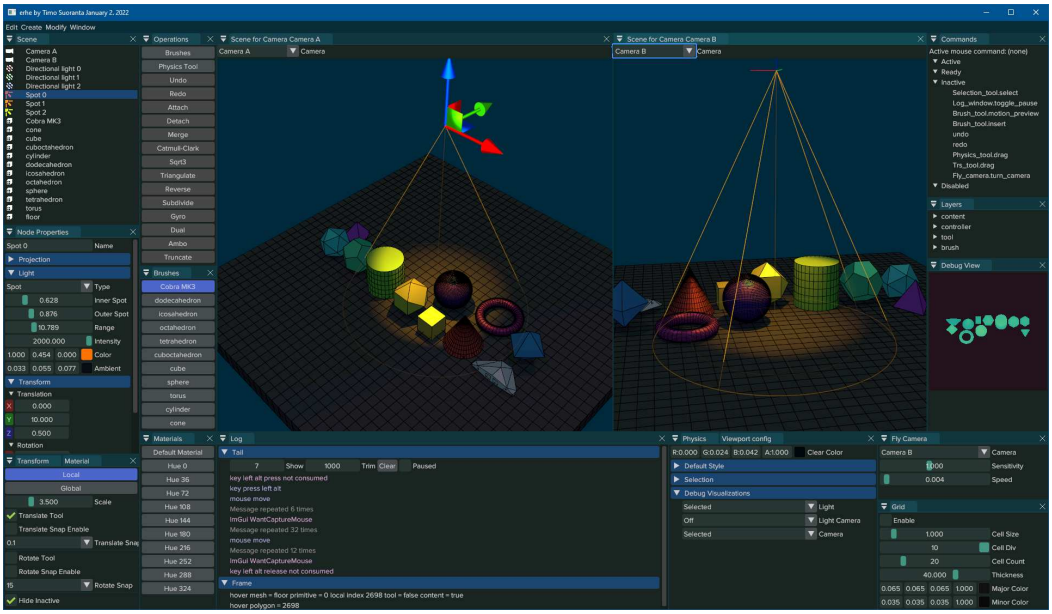
Slika 9.1: Prikaz kosti modela u grafičkom alatu Blender, izvor: `https://blenderartists.org/uploads/default/original/3X/1/3/13f89bebb4d35f3cd478bba3833bc1d47f1cb157.JPG`

9.0.2. Korisničko sučelje

Implementacija grafičkog korisničkog sučelja pomoću biblioteke Dear ImGui omogućila bi interaktivnu kontrolu parametara simulacije, te cijelog grafičkog pogona u stvarnom vremenu. Korisnik bi mogao:

- Dodavati i micati entitete (Object, Lighting, ...) u/iz scenu/e
- Objektima dodavati i micati komponente (Transform, MeshRenderer, RigidBody)
- Kontrolirati parametre komponenti
- Kontrolirati parametre kamere i scenskog svijetla
- Prikazati hijerarhiju svih entiteta u sceni
- Prikazati hijerarhiju kosti
- Povezati kosti modela sa zglobovima krpene lutke
- Pratiti performanse sustava (fps, broj objekata, memorijsku potrošnju)
- ...

Dear ImGui je biblioteka otvorenog koda prikladna za 3D aplikacije koje se izvršavaju u stvarnom vremenu [13]. Dizajnirana je kako bi omogućila brze iteracije, te olakšala programerima stvaranje alata za vizualizaciju i detektiranje pogrešaka (eng. *debug*). Biblioteka je brza, neovisna o aplikacijskom prikazivaču (eng. *renderer*), te ne ovisi o vanjskim komponentama (eng. *external dependency*), kao što su npr. dodatne biblioteke ili resursi.



Slika 9.2: Prikaz grafičkog pogona koji koristi Dear ImGui biblioteku, izvor: `https://user-images.githubusercontent.com/994606/147875067-a848991e-2ad2-4fd3-bf71-4aeb8a547bcf.png`

9.0.3. Unaprjeđenje iscrtavanja

Vizualna kvaliteta scene mogla bi se značajno poboljšati implementacijom:

- **Materijalnog sustava** - podrška za različite površinske materijale s kontrolom parametara poput sjaja, hrapavosti i reflektivnosti
- **Naprednijeg osvjetljenja** - implementacija physically-based rendering (PBR) modela s višestrukim izvorima svjetla i sjenama
- **Skybox-a** - kubna mapa okoline za realističniji dojam prostora i ambijentalno osvjetljenje

10. Zaključak

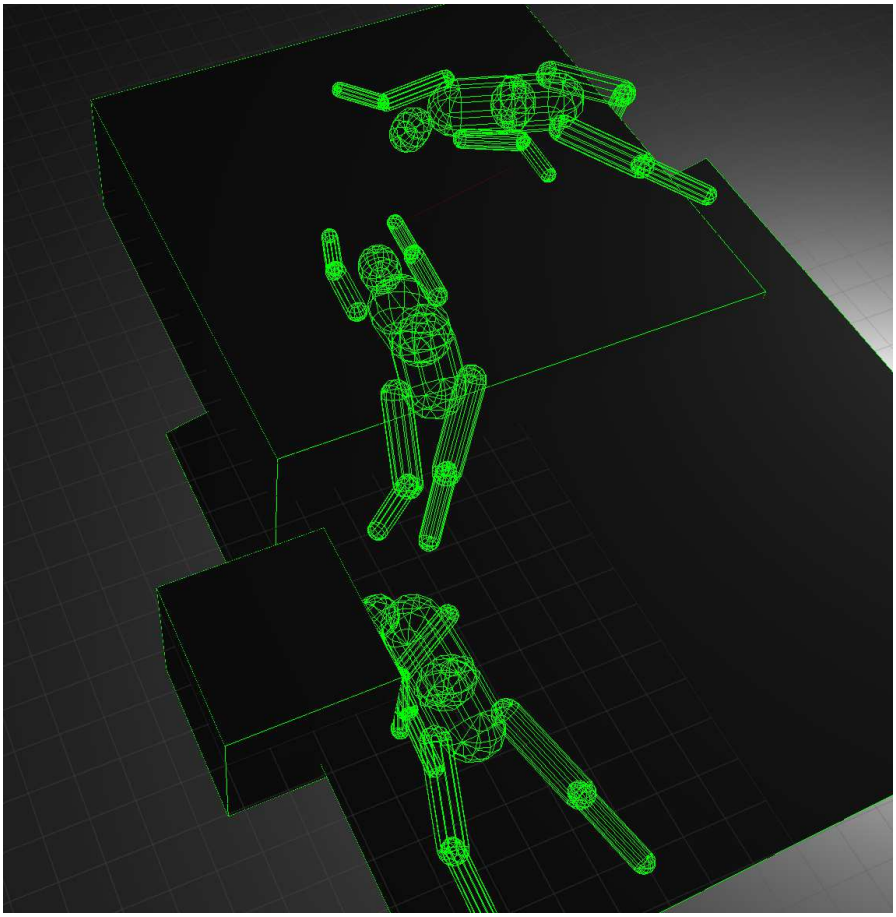
Cilj ovog rada bio je implementirati sustav za simulaciju fizike krpene lutke u stvarnom vremenu, te izmjeriti performanse rješenja ovisno o broju simuliranih objekata. Implementacija je provedena korištenjem Bullet Physics biblioteke, koja je uspješno povezana s postojećim OpenGL grafičkim pogonom. Proces implementacije uključuje izradu fizičkog modela krpene lutke pomoću definiranja omeđujućih volumena, krutih tijela, te njihovih međusobnih ograničenja. Rezultat je model koji realistično reagira na sile i impulse iz okoline.

Poseban naglasak stavljen je na povezivanje fizičkog i grafičkog pogona, što omogućava vizualnu povratnu informaciju. Time je olakšano razumijevanje ponašanja simulacije, te međusobnog odnosa fizičkih objekata u sceni. Povezivanjem fizičkog i grafičkog pogona omogućene su transformacije 3D modela (modeliranih u nekom 3D grafičkom alatu) uzrokovane silama: npr. padanje modela zbog utjecaja gravitacije.

Implementacijom konačnog automata omogućena je interakcija korisnika sa sustavom, a time i veća kontrola nad simulacijom. Konačni automat ovisno o korisnikovom unosu (putem tipkovnice ili miša) prelazi u različita stanja, od kojih svako ima definirano jedinstveno ponašanje. Jedno od tih stanja je stanje pomicanja, koje je od posebne važnosti za simulaciju, jer omogućava interaktivno pomicanje krpene lutke definiranjem *point to point* ograničenja.

Provedena mjerenja performansi pokazala su kako se broj okvira u sekundi smanjuje s porastom broja simuliranih krpenih lutaka. Mjerenja su provedena uz uključenu i isključenu opciju *VSync*. Rezultati ukazuju na to da je implementirani sustav primjenjiv u stvarnom vremenu, te da se 60 fps postiže pri simuliranju 24 krpene lutke, uz prostor za daljnje optimizacije.

Sustav predstavlja funkcionalnu osnovu za daljnje proširenje. Neka od proširenja o kojima se raspravlja su vizualno realniji prikaz pomoću povezivanja kostiju 3D modela i krutih tijela krpene lutke, implementacija grafičkog sučelja za interaktivnu kontrolu parametara, te različite tehnike optimizacije, koje bi omogućile simuliranje još većeg broja objekata.



Slika 10.1: Rezultat simuliranja krpenih lutaka

LITERATURA

[1] tutorialspoint *Computer Graphics - Animation, 2016.* https://www.tutorialspoint.com/computer_graphics/computer_animation.htm pristupljeno: rujan 2025.

[2] , Yunfei Bai, Erwin Coumans. *Bullet Collision Detection & Physics Library, 2006.* https://pybullet.org/Bullet/BulletFull/classbtTypedConstraint.html, pristupljeno: rujan 2025.

[3] Erwin Coumans *Exploring MLCP solvers and Featherstone, 2014.* https://media.gdcvault.com/GDC2014/Presentations/Coumans_Erwin_Physics_for_Game.pdf

[4] prof. dr. sc. Željka Mihajlović. *Detekcija sudara, 2024.* https://www.zemris.fer.hr/predmeti/ra/predavanja/6_collision.pdf, pristupljeno: rujan 2025.

[5] Jeffrey Potasky, Fentahun Reta, Nicholas Rollock, Jianhao Zhou. *Gilbert-Johnson-Keerthi Distance Algorithm, 2018.* https://cse442-17f.github.io/Gilbert-Johnson-Keerthi-Distance-Algorithm/, pristupljeno: rujan 2025.

[6] Reducible. *A Strange But Elegant Approach to a Surprisingly Hard Problem (GJK Algorithm), 2021.* https://www.youtube.com/watch?v=ajv46BSqcK4&t=84s, pristupljeno: rujan 2025.

[7] Tim. *How to draw/render a Bullet Physics collision body/shape?, 2012* https://stackoverflow.com/questions/11985204/how-to-draw-render-a-bullet-physics-collision-body-shape, pristupljeno: rujan 2025.

[8] Jochen Burghardt, DuncanHill, Seraphimblade, Tule-hog. *Finite-state machine, 2025.* https://en.wikipedia.org/wiki/Finite-state_machine, pristupljeno: studeni 2025.

[9] Bruce1ee, PlasticWonder, Tavianator, Tino. *Slab method, 2025.* https://en.wikipedia.org/wiki/Slab_method, pristupljeno: studeni 2025.

[10] Jean-Colas Prunier. *A Minimal Ray-Tracer, 2015.* https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection.html, pristupljeno: studeni 2025.

[11] Tulie Finley-Moise. *VSync: What It Is and When to Use It for Better Gaming, 2024.* https://www.hp.com/us-en/shop/tech-takes/vsync-should-i-turn-it-on-or-off, pristupljeno: siječanj 2026.

[12] Mike Seymour. *Bullet open source physics engine, 2011.* https://www.fxguide.com/fxfeatured/bullet_open_source_physics_engine/#:~:text=%E2%80%9CThe%20two%20main%20stages%20in,broadphase%20implementations%20for%20different%20purposes., pristupljeno: siječanj 2026.

[13] ocornut. *Dear ImGui, 2026.* https://github.com/ocornut/imgui/blob/master/docs/README.md, pristupljeno: siječanj 2026.

Proceduralna animacija modela čovjeka

Sažetak

U ovom radu je implementiran sustav za simulaciju fizike krpene lutke u realnom vremenu korištenjem Bullet Physics biblioteke i OpenGL-a. Rad započinje motivacijom problema i teorijskim osnovama proceduralne simulacije. Detaljno su obrađene teorijske osnove algoritma detekcije kolizija GJK, korištenog u Bullet Physics biblioteci. Opisana je arhitektura sustava, uključujući postavljanje projekta, integraciju Bullet Physics biblioteke i povezivanje grafičkog pogona s fizikalnim pogonom. Razrađen je proces stvaranja krpene lutke, te implementacija konačnog automata, korištenog za upravljanje stanjima sustava. Opisana je interakcija korisnika sa simulacijom. Provedena su mjerenja performansi sustava u stvarnom vremenu, te su analizirane postojeće i dodatne optimizacijske tehnike. Naposljetku su navedena moguća proširenja sustava.

Ključne riječi: proceduralna animacija, krpena lutka, fizički pogon, Bullet Physics, Assimp, Dear ImGUI, OpenGL, GJK algoritam, razlika Minkowskog, sekvencijalni rješavač impulsa, AABB, OBB, metoda plohe, BVH stablo, ECS

Procedural animation of human model

Abstract

In this paper, a real-time ragdoll physics simulation system was implemented using the Bullet Physics library and OpenGL. The paper begins with the motivation of the problem and the theoretical foundations of procedural simulation. The theoretical foundations of the GJK collision detection algorithm used in the Bullet Physics library are discussed in detail. The system architecture is described, including project setup, integration of the Bullet Physics library, and connection of the graphics engine with the physics engine. The process of creating a ragdoll model is presented, along with the implementation of a finite state machine used for managing system states. User interaction with the simulation is described. Real-time system performance system measurements are performed, and existing as well as additional optimization techniques are analyzed. Finally, possible extensions of the system are discussed.

Keywords: procedural animation, ragdoll, physics engine, Bullet Physics, Assimp, Dear ImGUI, OpenGL, GJK algorithm, Minkowski difference, Sequential Impulse Solver, AABB, OBB, slab method, BVH tree, ECS