SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6616

PRIMJENA INVERZNE KINEMATIKE U ANIMACIJI VIRTUALNOG ROBOTA PRI IZVOĐENJU ZADATAKA

Nikola Vugdelija

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6616

PRIMJENA INVERZNE KINEMATIKE U ANIMACIJI VIRTUALNOG ROBOTA PRI IZVOĐENJU ZADATAKA

Nikola Vugdelija

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6616

Pristupnik:	**Nikola Vugdelija (0036509409)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	prof. dr. sc. Maja Matijašević

Zadatak: **Primjena inverzne kinematike u animaciji virtualnog robota pri izvođenju zadataka**

Opis zadatka:

Od brojnih izazova u oblikovanju virtualnih okruženja, tema ovog završnog rada uključuje primjenu znanja iz područja virtualnih okruženja, interaktivne računalne grafike te razvoja programske podrške. Vaš zadatak je proučiti osnove inverzne kinematike pri animaciji te osmisliti i programski izvesti animaciju pokreta virtualnog robota u 3D okruženju u kojem robot provodi određeni zadatak. Kao primjer robotovog zadatka predvidite ličenje jednostavne pravokutne površine (zida) sa zadanim otvorima (prozorima i vratima). Pri izradi programskog rješenja koristite programski alat Unity. Literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Rok za predaju rada: 12. lipnja 2020.

Zahvaljujem se mentorici Maji Matijašević i profesoru Mirki Sužnjeviću na svoj pomoći koju su mi pružili pri pisanju ovog rada. Posebno se zahvaljujem svojim roditeljima, braći i bližnjima na danoj podršci.

Sadržaj

Uvod

Animiranje modela je ključni dio mnogih igara, simulacija i prezentacija. Mnogi alati za pravljenje navedenih programa pružaju potporu za stvaranje unaprijed definiranih animacija, ali problem nastaje kada ih korisnik želi stvarati proceduralno. Proceduralno animiranje se koristi za dobivanje raznovrsnijega skupa animacija koje će se različito izvoditi u ovisnosti o okolini i/ili stanju animiranog modela tijekom izvođenja programa. Primjene proceduralnog animiranja mogu biti:

- funkcionalne (npr. animacije kompleksnih modela, animacije koje su uvjetovane fizikom, stanjem modela, interakcijama s drugim modelima itd.);
- estetske (npr. animacije koje izgledaju prirodnije i/ili zabavnije).

U ovom radu se razmatra proceduralno animiranje robota koji boji zidove različitog izgleda, a bojanje zidova je ostvareno pomoću inverzne kinematike ili preciznije, pomoću algoritma FABRIK. Robota je potrebno animirati pomoću inverzne kinematike zbog kompleksnosti strukture modela robota i zbog prirode zadatka, tj. kako se ne bi trebala definirati nova duga i kompleksna animacija za svaki novi tip zida, nego će se tijekom izvođenja robot „sam“ animirati ovisno o dijelu zida koji trenutno boja.

U prvom poglavlju su opisani alati koji su korišteni u implementaciji. Drugo poglavlje predstavlja osvrt na inverznu kinematiku i algoritam FABRIK. U trećem poglavlju je detaljno opisana implementacija navedenog algoritma, algoritma generiranja sobe te algoritma po kojem robot određuje koji će zid idući bojati, a u zadnjem poglavlju su izneseni rezultate navedene implementacije.

1. Korišteni alati

U ovom poglavlju su navedeni alati koji su korišteni u implementaciji, te je objašnjena svrha tih alata u kontekstu rješenja zadatka. Za implementaciju zadatka su korištena dva programa: Unity Engine (detaljnije opisan u poglavlju 1.1.) i Blender (detaljnije opisan u poglavlju 1.2.).

1.1. Unity Engine

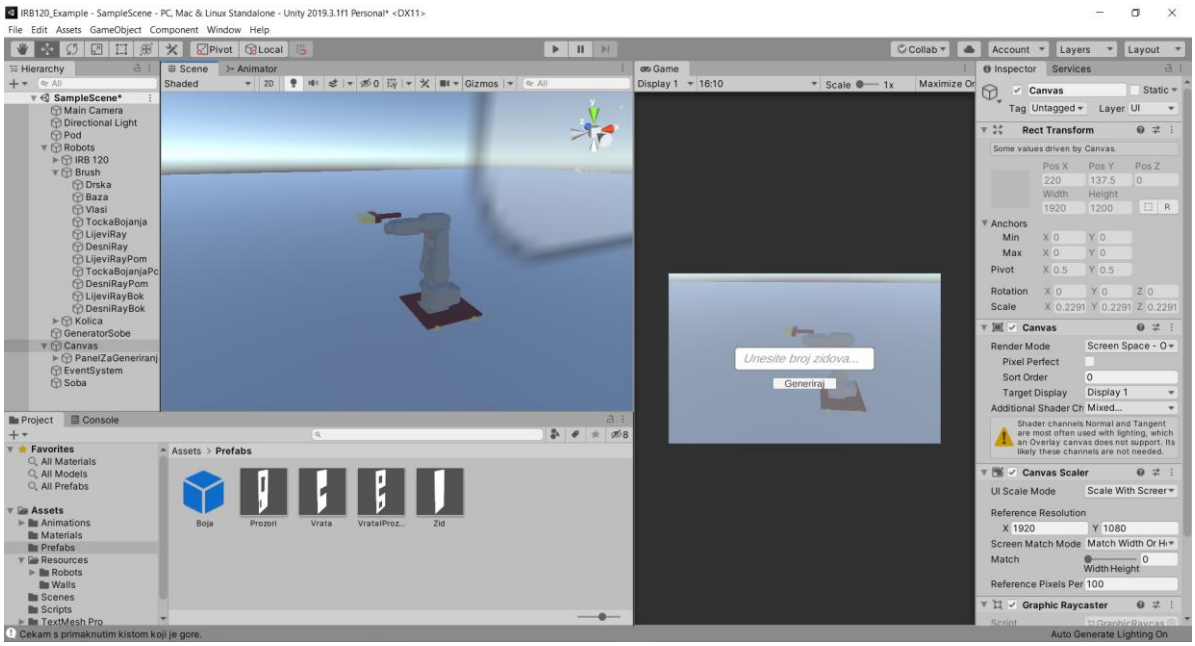
Unity Engine [1] je višeplatformni *game engine* koji je razvila tvrtka Unity Technologies. Prvi put je najavljen u lipnju 2005. godine na konferenciji „Worldwide Developers Conference“ tvrtke Apple Inc., kada je i službeno objavljen kao ekskluzivni *game engine* za Mac OS X. Cilj razvojnog tima bio je „demokratizirati“ područje razvoja igara tako što će napraviti alat koji će biti pristupačan većem broju razvojnih programera. Tijekom godina razvoja Unity je značajno povećao broj podržanih platformi, te se trenutno (stanje u lipnju 2020. godine) igre/simulacije razvijene u njemu mogu pokrenuti na 25 različitih platformi (uz moguće manje izmjene). Najveći napredak prema „demokratizaciji“ tržišta je tvrtka napravila u 2016. godini, kada su promijenili model licenciranja s modela jednokratne kupnje (korisnik kupi *engine* jednom i ima ga zauvijek) na pretplatni model (besplatna i plaćena opcija). Besplatna opcija pretplatnog modela je omogućila da korisnici koji nemaju novca za kupnju Unity-a, a žele praviti igre iz hobija ili za zaradu (sve dok je ta zarada manja 100 tisuća dolara godišnje), to mogu i učiniti bez ikakvih ograničenja u funkcionalnostima samog *engine*-a.

Unity Engine omogućuje stvaranje 2D i 3D igara i ostalih virtualnih iskustava. Iako je primarno razvijen kao *engine* za razvoj 3D igara, s vremenom je ostvarena podrška za 2D igre, ali i za razvoj AR (eng. *Augmented Reality*) i VR (eng. *Virtual Reality*) sadržaja. Kod 3D igara, Unity omogućuje mnoge korisne mogućnosti poput: kompresije tekstura, *mipmap*-a, dinamičkih sjena stvorenih pomoću mapa za sjene (eng. *Shadow maps*), 3D simulacije fizike, 3D svjetala, 3D animacija, itd. Dok „2D Unity“, iako je došao kasnije, podržava: uvoženje *sprite*-ova, korištenje naprednog 2D preglednika svijeta (eng. *2D world renderer*), *tilemap*-e, 2D simulaciju fizike, 2D svjetla itd. U obje dimenzije se može koristiti napredni stog digitalne

obrade (eng. *Post processing stack*) za poboljšanje izgleda igara i virtualnih iskustava pomoću posebnih efekata kao što su okluzija okoline, zamućenje pokreta, vinjeta, riblje oko i sl. Zbog želje za optimizacijom, u skoroj budućnosti se planira prelazak sa objektno orijentiranog dizajna na podatkovno orijentirani dizajn s Unity-evim DOTS-om (eng. *Data-Oriented Technology Stack*).

Primarni programski jezik koji Unity podržava je C#, koji je razvila tvrtka Microsoft, a postoji i *drag and drop* funkcionalnost za korisnike koji ne žele pisati kod. Prije nego što je C# postao primarni programski jezik, *engine* je podržavao programske jezike Boo (nije podržan od Unity verzije 5) i UnityScript (verzija JavaScripta, koja više nije podržana od kolovoza 2017. godine).

Unity Engine je središnji alat ovog zadatka. U ovom radu je korištena verzija 2019.3.1f1. Korišten je za nasumično generiranje sobe, proceduralno animiranje robota, bojanje zidova itd. Ovaj *engine* je odličan izbor za implementaciju ovog zadatka prvenstveno zato što je relativno jednostavan za korištenje, također ima veliku bazu korisnika i opširnu dokumentaciju, a uz sve to je i besplatan.

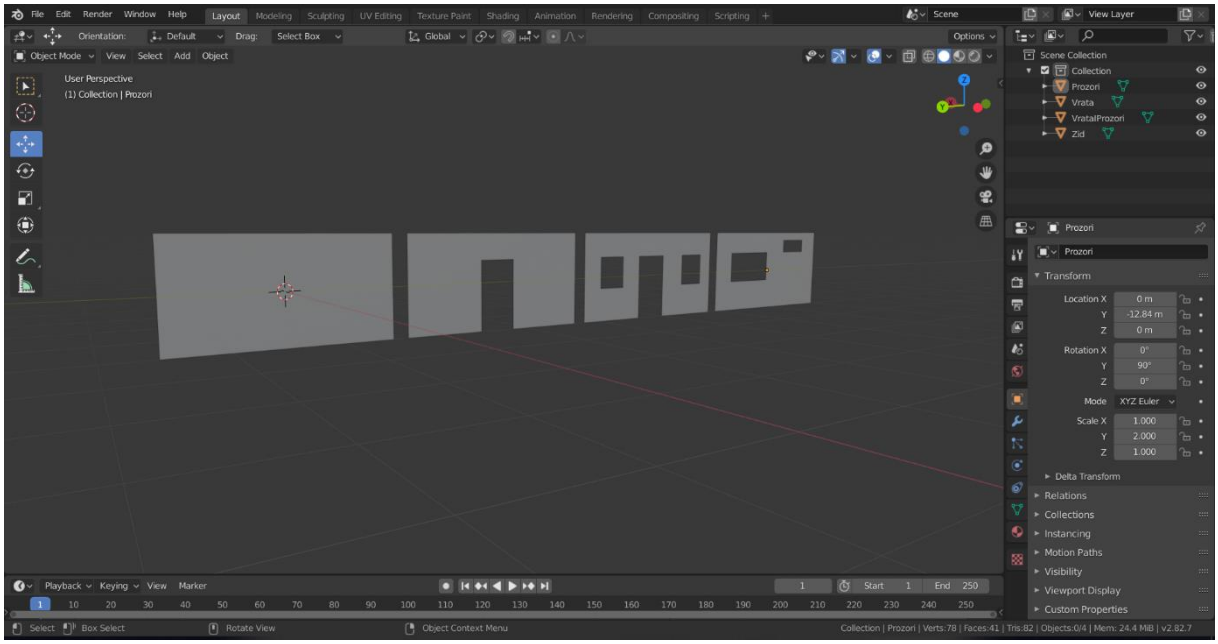


Slika 1.1. Korisničko sučelje Unity Engine-a

1.2. Blender

Blender [2] je besplatni alat za 3D računalnu grafiku otvorenog koda. Koristi se za stvaranje: 3D modela, vizualnih efekata, animiranih filmova (3D, ali i 2D), interaktivnih 3D aplikacija, igara i sl. Objavljen je 1. siječnja 1998. na internetu kao SGI (eng. *Silicon Graphics, Inc*) *freeware*. Jako je popularan kod amaterskih 3D umjetnika kao i kod studija na budžetu jer, iako je besplatan, sa svojim funkcionalnostima pokriva cijeli proces stvaranja 3D modela i efekata. Neke od mnogih Blender-ovih funkcionalnosti su: 3D modeliranje, UV odmatanje, simulacija tekućine i dima, simulacija čestica, simulacija mekih tijela, animiranje modela, stvaranje 2D animacija itd.

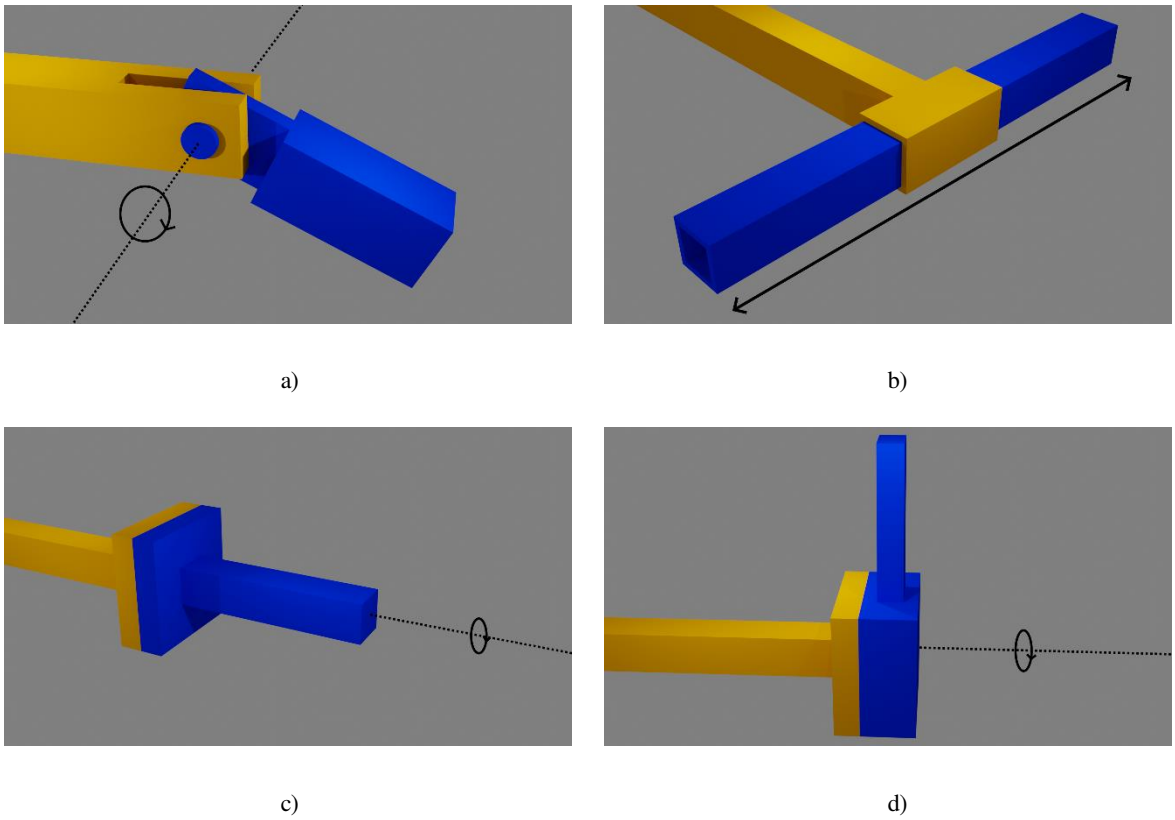
U zadatku je Blender korišten za modeliranje zidova koje robot boja, a verzija koja je korištena u ovom radu je 2.82a. Blender je odabran iz sličnih razloga kao i Unity, a to su: velika baza korisnika, detaljna dokumentacija, te jer je besplatan za razliku od konkurencije (3ds Max i Maya).



Slika 1.2. Modelirani zidovi u Blender-u

2. Inverzna kinematika i algoritam FABRIK

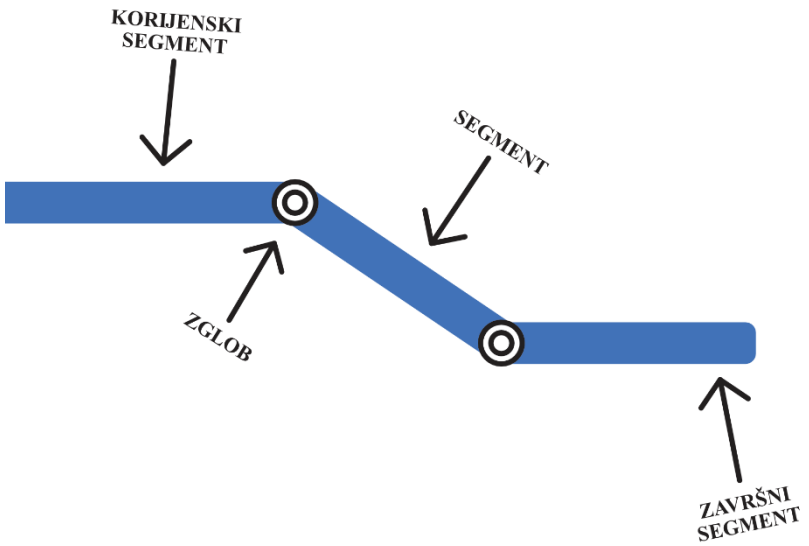
Kinematika je grana mehanike koja proučava pokrete tijela, a da pritom ne uzima u obzir sile pod čijim se djelovanjem to pokretanje odvija. U kontekstu oblikovanja virtualnih okruženja, kinematika je jedan od mnogih pristupa animiranju modela. Dva su osnovna pristupa takvom animiranju – direktna i inverzna kinematika. Oba pristupa počivaju na pretpostavci da su elementi modela koji se animira organizirani u hijerarhijsku strukturu. Takva struktura se sastoji od zglobova i segmenata. Segmenti su stvarni grafički elementi, a zglobovi su poveznice između segmenata čije dimenzije ne uzimamo u obzir. Razlikujemo rotacijske, translacijske, torzijske i revolucijske zglobove (Slike 2.1.) s različitim stupnjevima slobode.



Slike 2.1. Vrste zglobova: a) rotacijski, b) translacijski, c) torzijski, d) revolucijski

Svi elementi strukture čine hijerarhijsko stablo, u kojem će korijenski čvor biti korijenski segment, koji se ovisno o ostalim postavkama može, ali i ne mora micati, a njegova

djeca će biti svi segmenti kojima je vezan sa zglobovima koji su vezani za njega (Slika 2.2.). Slično vrijedi za sve sljedeće segmente i zglobove u hijerarhiji, sve do krajnjeg segmenta/segmenata, ali oni se moraju moći pomaknuti od svoje početne pozicije.

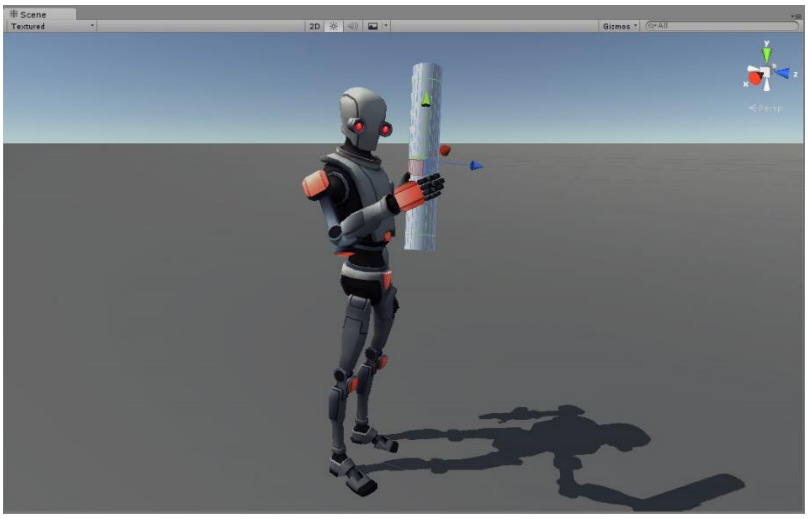


Slika 2.2. Pojednostavljeni prikaz strukture

U nastavku ovog poglavlja opisana je inverzna kinematika (poglavlje 2.1), pošto je s njome animiran robot, te konkretni algoritam ostvarenja inverzne kinematike koji je implementiran, a to je algoritam FABRIK (poglavlje 2.2.).

2.1. Inverzna kinematika

Inverzna kinematika je definirana kao problematika pronalaska skupa prigodnih konfiguracija zglobova za koje će se krajnji segment pomaknuti na željenu poziciju što je tečnije, brže i preciznije moguće. Za razliku od direktne kinematike, koja zadaje položaje i kutove svih zglobova, te na osnovu njih izračunava položaj krajnjeg segmenta, inverzna kinematika, kao što i samo ime kaže, radi obrnuti postupak. Ona nalazi položaje i kutove svih zglobova za koje će krajnji segment biti u poziciji ili što je moguće bliže poziciji zadane točke, pritom uzimajući u obzir zadana ograničenja modela.



Slika 2.3. Primjer modela animiranog inverznom kinematikom iz Unity dokumentacije

Proizvod direktne kinematike je uvijek točno jedno rješenje, dok inverzna kinematika može imati nijedno, jedno, pa čak i više rješenja. Radni prostor je područje oko manipulatora (robotske ruke) koje se sastoji od točaka koje mogu biti rješenje problema inverzne kinematike za zadani manipulator uz zadana ograničenja. Taj prostor još nazivamo i dohvatljivi radni prostor, a njegov podskup, za čije se ciljne točke robotska ruka može kretati sa svakim dozvoljenim stupnjem slobode, se naziva spretno dohvatljiv radni prostor. Kod problema ovakve kompleksnosti očekivano je da ne može postojati samo jedno rješenje, tj. jedan algoritam. Algoritmi se mogu razlikovati po brzini, pristupu (direktan ili iterativan), preciznosti, broju iteracija (kod iterativnih algoritama) i sl. Samo neki od najpoznatijih algoritama su: FABRIK (eng. *Forward And Backward Reaching Inverse Kinematics*), CCD (eng. *Cyclic Coordinate Descent*), *Jacobian DLS* (eng. *Damped Least Squares*), *Jacobian Transpose*, *Simple Inverse Kinematics* i drugi.

2.2. Algoritam FABRIK (novo heurističko rješenje problema inverzne kinematike)

Algoritam FABRIK [3] je novi heuristički pristup rješavanju problema inverzne kinematike. Algoritam je dobio ime FABRIK jer iterativno prelazi unaprijed i unazad preko svih zglobova kako bi što točnije odredio njihovu konfiguraciju. Umjesto da koristi matrice rotacija kako bi pronašao rotaciju svakog zgloba, FABRIK traži njihovu završnu poziciju.

Algoritam je korišten u iznimno kompleksnim sustavima s jednom i s više ciljnih točaka, te s ograničenjima na zglobovima i bez njih. FABRIK je jako učinkovit i kod jednostavnih i kod složenih problema, te proizvodi slične ili čak i bolje pokrete nego neke sofisticiranije metode, a da pritom zahtjeva manje vremena i manje iteracija da dođe do ciljne točke. Još jedna važna prednost ovog algoritma je jednostavnost izmjena za bilo koji problem inverzne kinematike. U nastavku ovog poglavlja je predstavljen spomenuti algoritam (poglavlje 2.2.1.), te ga se uspoređuje s drugim poznatim algoritmima (poglavlje 2.2.2.).

2.2.1. Opis algoritma

Već je spomenuto da algoritam prelazi preko zglobova unaprijed i unazad iterativno kako bi došao do rješenja, ali u nastavku će biti objašnjen redoslijed tih iteracija, te što se u svakoj iteraciji radi. U nastavku se neće koristiti izrazi segmenti i zglobovi, nego će izraz čvor enkapsulirati u sebi segment i zglobove koji su na njega spojeni.

Prije nego što se počne s računanjem, potrebno je odrediti toleranciju na udaljenost završnog čvora od ciljne točke, tj. na kojoj se udaljenosti od ciljne točke završni čvor treba nalaziti da bi algoritam mogao zaključiti da je rješenje pronađeno. Algoritam treba inicijalizirati, nakon čega krajnji čvor treba imati sljedeća saznanja:

- Koji čvorovi se nalaze iznad njega u hijerarhiji?
- Kolika je udaljenost između tih čvorova koji su izravno povezani?

S tim saznanjima, algoritam se može uspješno izvoditi nakon svake promjene pozicije ciljne točke.

Kada dođe do promjene pozicije ciljne točke, algoritam najprije treba provjeriti je li točka unutar radnog prostora modela, te će se, ovisno o onome što korisnik želi od modela, svi čvorovi ispružiti prema ciljnoj točci ili će se izaći iz funkcije i čvorovi će ostati na prijašnjim pozicijama. Ako je točka unutar radnog prostora modela, algoritam može nastaviti s radom tako što će najprije spremiti poziciju korijenskog čvora u posebnu varijablu za buduću usporedbu u iteracijama, te će ući u petlju u kojoj će biti sve dok je udaljenost krajnjeg čvora od ciljne točke veća od zadane tolerancije. U toj petlji će uvijek prvo ići na fazu „prednjeg dostizanja“. U toj fazi se prvo završni čvor postavi na poziciju ciljne točke. Zatim ulazi u petlju u kojoj će raditi sljedeće:

- Dohvatiti roditeljski čvor od čvora iz prijašnje iteracije, a to je inicijalno završni čvor.

- Izračunati vektor od prijašnjeg segmenta do njegovog roditelja.
- Normalizirati taj vektor i pomnožiti ga s prethodno dobivenom udaljenosti između ta dva čvora.
- Postaviti poziciju roditeljskog čvora na poziciju prijašnjeg čvora zbrojenog s dobivenim vektorom.

U fazi „prednjeg dostizanja“ algoritam prolazi kroz sve čvorove i ponavlja navedene radnje. Kada su svi čvorovi premješteni, može se primijetiti da se u većini slučajeva korijenski čvor nalazi izvan svoje početne pozicije. To se lagano popravlja tako da se on vrati na poziciju koja je prethodno spremljena, a s time počinje faza „stražnjeg dostizanja“. To je tzv. „ispravljačka faza“ u kojoj se ponavljaju sve iste radnje kao i u prvoj fazi samo algoritam ide unatrag od korijenskog čvora prema završnom čvoru. Na kraju te faze se opet može primijetiti da u većini slučajeva nakon prve iteracije vanjske petlje, završni čvor neće biti u poziciji ciljne točke, ali to nije bitno jer će biti puno bliže nego je bio kad se vanjska petlja tek počela izvršavati, a kao što je već rečeno, ona će se izvršavati dok završni čvor ne bude dovoljno blizu ciljnoj točki. U nastavku je napisan pseudokod algoritma.

```
1. Ulaz: pozicije čvorova p_i za i=1,…,n, pozicija ciljne točke g,
   udaljenosti između svakog čvora d_i=|p_i+1-p_i| za i=1,…,n-1,tolerancija
   na udaljenost t
2. Izlaz: nove pozicije čvorova p_i za i=1,…,n
3. //Izračunaj udaljenost između korijena i ciljne točke
4. udaljenost = |p_1-g|
5. //Provjeri je li ciljna točka unutar radnog prostora
6. Ako udaljenost>d_1+d_2+…+d_n-1 onda
7.      //Ovisi što želimo s modelom, ali radi jednostavnosti ćemo izaći
8.      Izlaz(p_i za i=1,…,n)
9. Kraj
10.  //Ciljna točka je unutar radnog prostora, spremamo poziciju
   korijena
11.  korijen = p_1
12.  //Ulazimo u vanjsku petlju koja se vrti dok god je udaljenost
   između ciljne točke i završnog čvora veća od tolerancije
13.
14.  Dok |p_n-g| > t činiti
15.       //1. FAZA: PREDNJE DOSTIZANJE
16.       p_n=g
17.       Za i = n-1,…,1 činiti
18.            v = p_i - p_i+1
19.            v /= |v|
20.            v *= d_i
21.            p_i = p_i+1 + v
22.       Kraj
23.       //2.FAZA: STRAŽNJE DOSTIZANJE
24.       p1 = korijen
25.       Za i = 2,…,n činiti
```

```
26.            v = pi-pi-1
27.            v /= |v|
28.            v *= di-1
29.            Pi = pi-1 + v
30.       Kraj
31.  Kraj
32.  Izlaz(pi za i=1,…,n)
```

Pseudokôd 2.1. - Algoritam FABRIK

2.2.2. Usporedba s ostalim algoritmima

Iako je relativno nova metoda, algoritam FABRIK se u praksi pokazao kao veoma brz i pouzdan pristup. U nastavku ga uspoređujemo s nekim od poznatijih algoritama, a to su CCD, *Jacobian DLS* i *Jacobian Transpose*. Usporedbe koje slijede, preuzete iz rada [3], se provode na kinematičkim lancima sastavljenima od 10 zglobova s tri stupnja slobode bez ikakvih ograničenja.. CCD daje brze rezultate, ali zato što razmotava i mota lanac prije nego što dođe do ciljne točke, poze i kretnje koje dobijemo tim algoritmom nisu realistične. Jacobijeve metode daju bolje rezultate, no trošak računanja je iznimno velik, imaju probleme sa singularnošću (konfiguracija kod koje krajnji je krajnji čvor blokiran u određenom smjeru [4]) i proizvode nerealistične pokrete kada je završni čvor značajno udaljen od ciljne točke. FABRIK proizvodi realističnije pokrete nego ostale 3 metode. Također je 10 puta brži od CCD-a i otprilike je 1000 puta brži od Jacobijevih metoda. U tablici 2.1. mogu se vidjeti konkretni rezultati usporedbe provedene u MatLab-u [3].

Algoritmi	**Prosječan broj iteracija**	**Vrijeme izvršavanja u Matlab-u [s]**
FABRIK	15.46	0.013
CCD	26.31	0.123
Jacobian Transpose	1311	12.989
Jacobian DLS	998	10.480

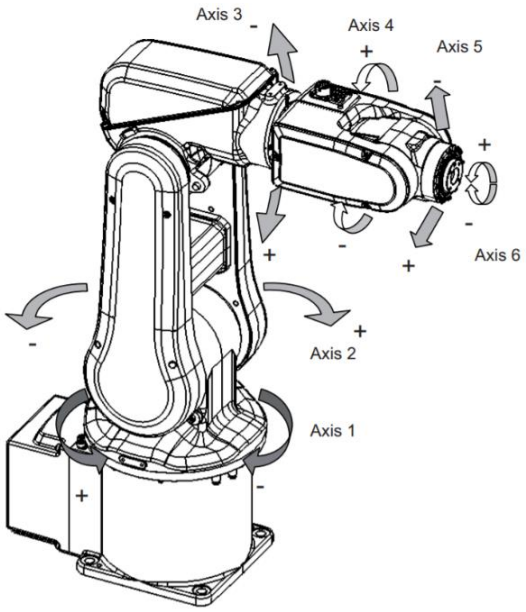
Tablica 2.1. Usporedba algoritama inverzne kinematike [3]

3. Specifikacija zahtjeva

U ovom radu se demonstrira primjena inverzne kinematike za animaciju modela robota kod izvođenja određenog zadatka. Konkretan zadatak za koji je robota potrebno animirati je bojanje zidova. Implementacija tog zadatka može biti podijeljena u više dijelova, tj. više problema. Prvi dio bi bio generiranje zidova, tj. stvaranje okoline koju bi robot trebao obojiti. Drugi dio bi bio osmišljavanje i programsko izvođenje animiranja robota nekim od algoritama inverzne kinematike (u ovome radu odabran je algoritam FABRIK). Pod treći dio zadatka spada smišljanje i implementacija algoritma bojanja zida, a to uključuje smišljanje načina i implementaciju sljedećeg:

- bojanja zida, tj. ostavljanje traga na zidu;
- kretnje robota od jednog kraja zida do drugog, te prepoznavanje kraja zida;
- prepoznavanje rupa u zidu i prigodno reagiranje na njih.

Zadnji tj. četvrti dio zadatka bi bio osmišljavanje i implementiranje preciznog kretanja robota od jednog do drugog zida. Svi navedeni dijelovi zadatka su detaljno opisani u 4. poglavlju. Model robota koji je korišten u zadatku je IRB 120. Robotska ruka IRB 120 je industrijski robot tvrtke ABB. Navedeni model je 6-osni robot, što znači da ima 6 zglobova, a svaki zglob ima jedan stupanj slobode. Slika 3.1. [5] prikazuje izgled robotske ruke.



Slika 3.1. Slika osi robotske ruke [5]

Sadržaj tablice 3.1. je preuzet iz priručnika za korištenje robota IRB 120 [5] i specificira ograničenja u rotaciji oko pojedine osi.

Ime osi	**Vrsta zglobova**	**Raspon pokreta**
Os 1	Torzijski zglob	+165° do -165°
Os 2	Rotacijski zglob	+110° do -110°
Os 3	Rotacijski zglob	+70° do -110°
Os 4	Torzijski zglob	+160° do -160°
Os 5	Rotacijski zglob	+120° do -120°
Os 6	Torzijski zglob	+400° do -400°

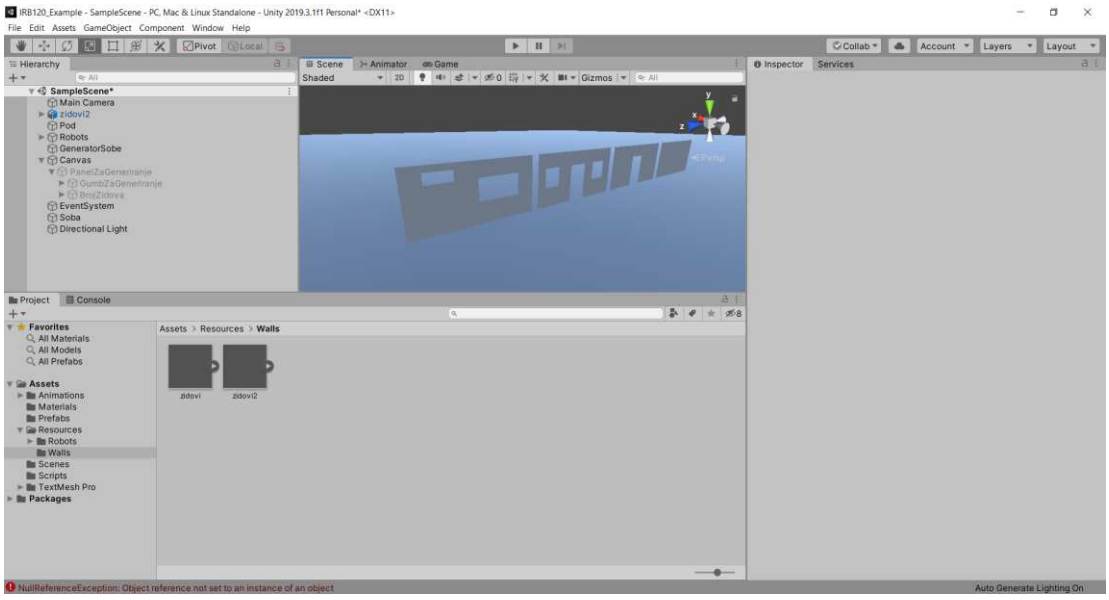
Tablica 3.1. Ograničenja u rotaciji oko pojedinih osi robota [5]

4. Implementacija

U nastavku ovog poglavlja su opisani detalji oko implementacije. Poglavlje 4.1. opisuje algoritam i implementaciju generiranja sobe. Poglavlje 4.2. opisuje implementaciju inverzne kinematike u modelu robotske ruke, a poglavlje 4.3 opisuje algoritam i implementaciju bojanja zida. Zadnje poglavlje je poglavlje 4.4 i u njemu je opisano kretanje robota po generiranoj sobi.

4.1. Generiranje sobe

Kako bi se pokazala općenita primjenjivost ovakvog načina animiranja, u Blender-u su napravljena 4 jednostavna modela zida: prazni zid, zid s vratima, zid s 2 prozora, te zid s vratima i 2 prozora. Zatim su ta 4 modela uvedena u Unity Engine unutar jedne .fbx datoteke. Pošto su sva 4 zida bila u jednoj datoteci, trebalo ih je razdvojiti, postaviti im *Mesh Collider*-e, postaviti im točke okreta (eng. *Pivot points*), te ih spremiti kao predloške (eng. *prefabs*) za nasumično generiranje.



Slika 4.1. Modeli zidova unutar Unity Engine-a

Nakon toga je trebalo odabrati oblik „sobe“ koju će se generirati (pravokutna, ovalna, niz zidova u redu itd.) i na koji način će ona biti generirana. Odlučeno je da će korisnik, do određene mjere, moći odabrati izgled sobe, pri tom demonstrirajući općenitost rješenja. Pošto su svi modelirani zidovi jednake dužine, glavna ideja je bila da korisnik upiše broj zidova koji želi imati u sobi (minimalno 4), a da zatim algoritam stvori sobu u obliku mnogokuta s tim brojem zidova. Vrsta svakog pojedinog zida će biti određena nasumično pri stvaranju, a na kraju stvaranja sobe u njenoj sredini će biti model robota. Ovakvim načinom može se stvoriti i jednostavna kvadratna soba, ali i veća (približno) kružna soba.

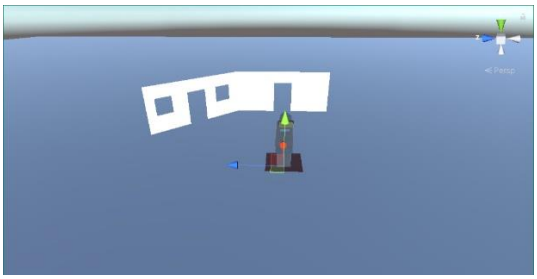
Nakon što je donesena odluka o izgledu sobe, trebalo je smisliti algoritam generiranja sobe. Ideja algoritma je bila da se stvara jedan nasumično odabrani zid iznad robota, dodijeli ga se roditeljskom objektu, a da se tog roditeljskog objekta nakon toga zarotira za konstantnu veličinu kuta α koja će ovisiti o ukupnom broju zidova N. Ta radnja će se ponavljati sve dok nisu stvoreni svi zidovi. Pošto se radi o pravilnom mnogokutu, jednostavno je pronaći vrijednost kuta za koju treba zarotirati roditeljski objekt pomoću izraza (1). Također, svi će zidovi od središta, tj. od robota, u početku biti jednako udaljeni. Tu udaljenost d se može izračunati koristeći izraz (2), za koji vrijedi da je w širina zida.

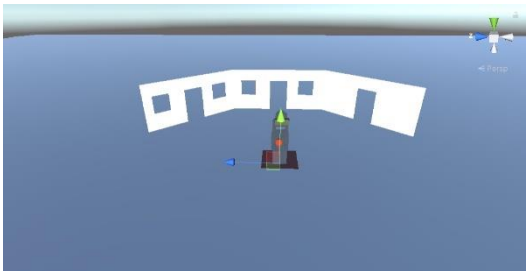
$$\alpha = \frac{360}{N} \tag{1}$$

$$d = \left(\frac{w}{2}\right) * \tan\left(\frac{180-\alpha}{2}\right) \tag{2}$$

Pomoću navedenih formula, postupak se svodi na stvaranje zida na poziciji (0, visina, d), pridjeljivanje roditeljskog objekta tom zidu i rotiranje tog objekta za kut α, sve dok svi zidovi nisu stvoreni, pri čemu se pod varijablom visina misli na vrijednost koja ovisi o karakteristikama modela zida. Na slikama 4.2. je prikazano nekoliko iteracija algoritma.

a)





b) c)

Slike 4.2. Slike algoritma generiranja sobe: a) stvori se prvi zid i postavi mu se roditelj, b) roditelj se zarotira i a) se ponovi za drugi zid, c) slično kao b)

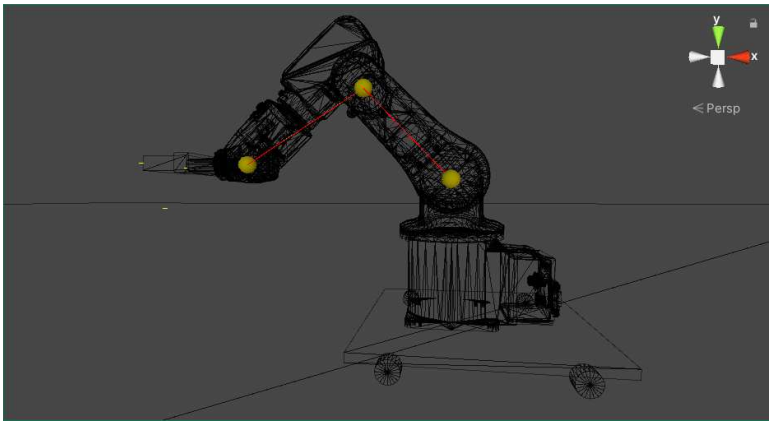
Kada je generator sobe bio implementiran, spojen je na jednostavno korisničko sučelje (Slika 4.3.) u kojem korisnik može upisati broj zidova i pokrenuti generiranje sobe pritiskom na gumb „Generiraj“. Nakon što se soba stvori, robot se okreće prema prvom zidu koji namjerava obojati.

Slika 4.3. Korisničko sučelje u kojem korisnik može definirati broj zidova sobe

4.2. Inverzna kinematika na modelu robota IRB 120

Nakon što je uspješno riješeno generiranje sobe, uslijedilo je implementiranje algoritma FABRIK na modelu robota. Baza implementacije je bio pseudokôd 2.1. uz nekoliko promjena.

Iako bi bilo smisleno zaključiti da 6-osni robot ima 6 čvorova, u ovoj implementaciji to nije slučaj. Svi čvorovi koji sadrže torzijske zglobove su izostavljeni iz algoritma FABRIK, jer njihove rotacije nisu korisne u kontekstu bojanja zida. Čvor na osi 5 je završni čvor, stoga algoritam neće određivati njegovu rotaciju, već samo njegovu poziciju. To je dobro, jer ako bi taj čvor ostao kao jedan od čvorova u nizu, a čvor na osi 6 kao završni čvor, čvor na osi 5 bi se uvijek rotirao prema ciljnoj točki, ali taj čvor treba biti usmjeren prema zidu jer njemu pripada segment robota koji drži kist.



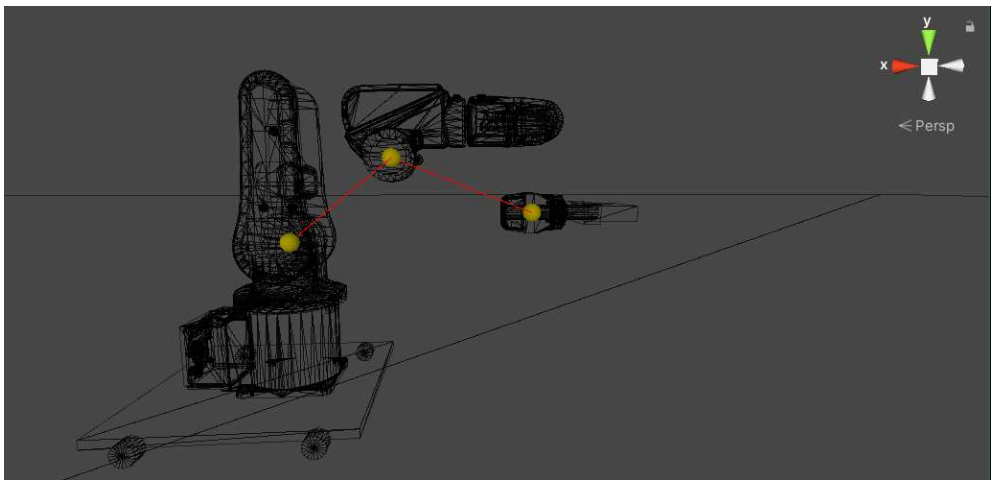
Slika 4.4. Sferama su označeni svi čvorovi robota koji su uzeti u obzir pri računanju FABRIK-a

Skripta koja implementira algoritam FABRIK je komponenta završnog čvora kako bi u inicijalizaciji mogla dohvatiti roditelje svih čvorova u lancu i spremati njihove transformacijske komponente u listu. Osim liste transformacijskih komponenata, u inicijalizacijskoj funkciji se još dohvaća lista udaljenosti između čvorova, dohvaća ukupna duljina robota, te inicijaliziraju početne rotacije na vrijednost (0,0,0).

Nakon što je algoritam uspješno inicijaliziran, komponenta će u svakoj LateUpdate() funkciji moći pozvati funkciju IzracunajFABRIK(), koja će, kao što samo ime kaže, računati nove pozicije i rotacije čvorova primjenom algoritma FABRIK.

Ciljna točka je pozicija kista, a kretnje kista su animirane unutar koda klase Brush. Kist nije animiran u Unity-evom *Animator*-u jer se ovako dobije veća sloboda definiranja kretnji kista, npr. lakše ga je zaustaviti kad dođe do otvora, lakše ga je odmaknuti od zida kad ne treba biti na zidu, lakše je komunicirati s robotom pri kraju poteza itd.

Prvi dio funkcije IzracunajFABRIK() je implementacija pseudokôda 2.1. Jedina razlika između pseudokôda i implementacije je ta da se pozicije izračunate u iteracijama ne spremaju u transformacijske komponente, tj. pozicije samih čvorova se mijenjaju tek na kraju funkcije. Drugi dio funkcije postavlja pozicije čvorova, te izračunava i postavlja rotacije čvorova. Postavljanje pozicija čvorova se obavlja iteriranjem od korijenskog do završnog čvora uz dodjeljivanje izračunatih pozicija transformacijskim komponentama čvorova, ali samo to nije dovoljno kao što se može vidjeti na slici 4.5.



Slika 4.5. Rezultat implementacije FABRIK algoritma koja samo računa pozicije čvorova

Ono što nedostaje su već spomenute rotacije čvorova, a njih se može dobiti koristeći izraz (3), za koji vrijedi da je poz[i-1] trenutna pozicija čvora djeteta, a izPoz[i-1] je izračunata pozicija čvora djeteta.

$$rotacija[i] = 2 * \arcsin\left(\frac{|poz[i-1] - izPoz[i-1]|}{2 * udaljenostCvorova[i-1]}\right) \quad (3)$$

Problem s ovim izrazom je taj što ga se ne može koristiti samostalno jer iz samog izraza neće biti vidljivo je li kut rotacije pozitivan ili negativan, tj. kut će uvijek biti pozitivan jer je

razlomak pod funkcijom arkus sinus uvijek pozitivan. To se može popraviti projiciranjem vektora poz[i-1]-poz[i] i vektora izPoz[i-1] -poz[i-1] na x-y ravninu, te provjeravanjem je li z komponenta vektorskog umnoška novonastalih vektora veća ili manja od 0. Ako je veća, rotacija je pozitivna jer se drugi vektor nalazi „iza“ prvog vektora gledajući u smjeru kazaljke na satu (Unity Engine koristi lijevi koordinatni sustav), a ako je manja, rotacija je negativna i treba je pomnožiti sa -1. Ali ni ovo rješenje samo po sebi nije dovoljno jer se stvaraju problemi kada je robot zaokrenut tako da su x komponente ovih vektora jako male ili jednake 0, pa vektori koji nastaju njihovom projekcijom nisu dobro izračunati. Iz tog razloga, prije nego što ih se vektorski pomnoži, spomenute vektore treba zarotirati za suprotan kut od kuta trenutne rotacije robota, kako bi ti vektori sigurno bili pravilno preneseni na x-y ravninu. Nakon toga se izračunatu rotaciju doda na trenutnu rotaciju čvora, te novo izračunatu rotaciju se spremi u listu rotacija. Zadnja stvar koju algoritam treba izračunati je rotacija završnog čvora jer, kao što je već spomenuto, taj čvor treba uvijek biti usmjeren prema zidu. Njegova rotacija će biti suprotna zbroju rotacija svih čvorova koji su došli prije njega. Konkretna implementacija funkcije u jeziku C# je dana u kôdu 4.1.

```
1. public void IzracunajFABRIK()
2. {
3.     if ((ciljnaTocka.position -
   cvorovi[duljinaLanca].position).magnitude > maxDuljina)
4.         return;
5.
6.     if ((pPozicije[0] - ciljnaTocka.position).magnitude <
   tolerancija)
7.         return;
8.
9.     korijen = cvorovi[duljinaLanca].position;
10.
11.      for (int i = 0; i <= duljinaLanca; ++i)
12.      {
13.          pPozicije[i] = cvorovi[i].position;
14.      }
15.      while ((pPozicije[0] - ciljnaTocka.position).magnitude >
   tolerancija)
16.      {
17.          sPozicije[0] = ciljnaTocka.position;
18.
19.          for (int i = 1; i <= duljinaLanca; ++i)
20.          {
21.              sPozicije[i] = sPozicije[i - 1] + (pPozicije[i] -
   sPozicije[i - 1]).normalized * udaljenostiCvorova[i - 1];
22.          }
23.          pPozicije[duljinaLanca] = korijen;
24.
25.          for (int i = duljinaLanca - 1; i >= 0; --i)
26.          {
27.              pPozicije[i] = pPozicije[i + 1] + (sPozicije[i] -
   pPozicije[i + 1]).normalized * udaljenostiCvorova[i];
```

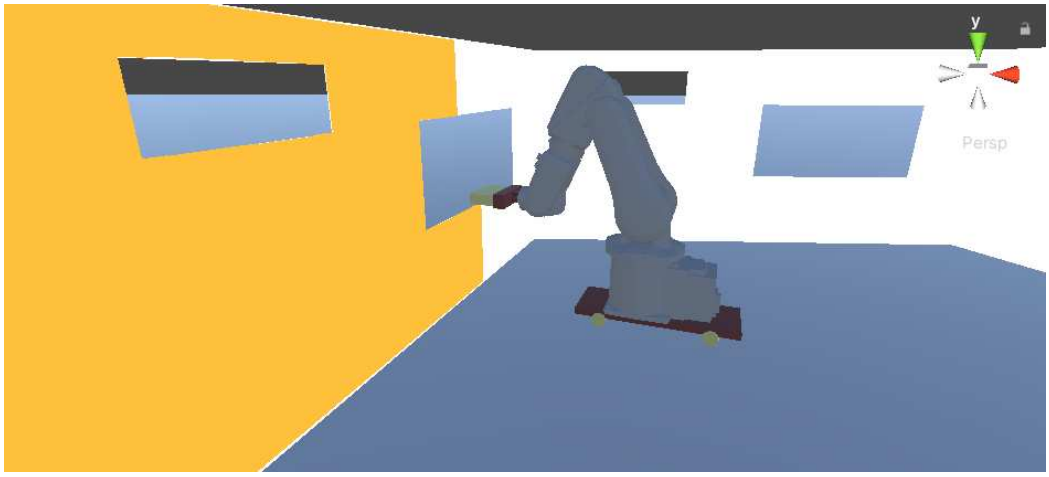
```
28.           }
29.       }
30.       float rotacija1 = 0;
31.       for (int i = duljinaLanca; i > 0; --i)
32.       {
33.           ParentController parent =
   cvorovi[i].gameObject.GetComponent<ParentController>();
34.           float rotacija = Mathf.Asin(((-pPozicije[i - 1] + cvorovi[i
   - 1].position).magnitude / 2) / udaljenostiCvorova[i - 1]) *
   Mathf.Rad2Deg * 2;
35.           Vector3 vec1 = cvorovi[i - 1].position -
   cvorovi[i].position;
36.           Vector3 vec2 = pPozicije[i - 1] - cvorovi[i].position;
37.           float rotirajZa = robot.transform.rotation.eulerAngles.y;
38.           if (rotirajZa > 180)
39.               rotirajZa -= 360;
40.
41.           vec1 = Quaternion.Euler(0, -rotirajZa, 0) * vec1;
42.           vec2 = Quaternion.Euler(0, -rotirajZa, 0) * vec2;
43.           if (Vector3.Cross(vec1, vec2).z < 0)
44.           {
45.               rotacija = -rotacija;
46.           }
47.           rotacija += rotacije[i][(int)parent.os];
48.           rotacija = OgraniciRotaciju(rotacija, parent.minRot,
   parent.maxRot);
49.           Vector3 rotacijskiVektor = new Vector3(0, 0, 0);
50.           rotacijskiVektor[(int)parent.os] = rotacija;
51.           cvorovi[i].localEulerAngles = rotacijskiVektor;
52.           rotacije[i] = cvorovi[i].localRotation.eulerAngles;
53.           rotacija1 -= rotacija;
54.       }
55.
56.       rotacija1 = OgraniciRotaciju(rotacija1, minRot, maxRot);
57.       cvorovi[0].localRotation = Quaternion.Euler(0, 0, rotacija1);
58.       rotacije[0] = cvorovi[0].localRotation.eulerAngles;
59.   }
```

Kôd 4.1. Funkcija IzracunajFABRIK()

Svaki čvor rotira oko svoje lokalne osi koja može biti x, y ili z, a os rotacije pojedinog čvora se može namjestiti kroz komponentu ParentController pomoću varijable os koja ima 3 moguće vrijednosti, a to su x, y i z. Pomoćna funkcija OgraniciRotaciju služi za stavljanje izračunate rotacije u interval između -180° i 180°, te potom ograničavanje te rotacije na intervale navedene u tablici 3.1.

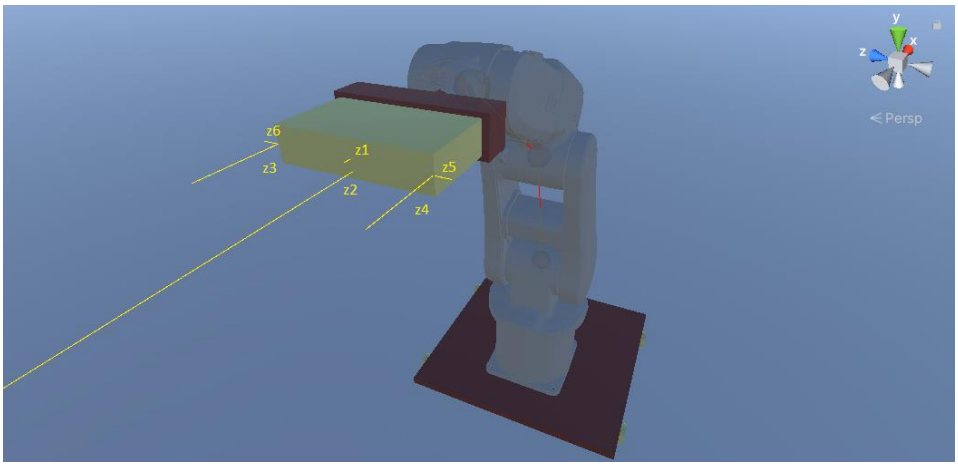
4.3. Bojanje zida

Kao što je spomenuto u trećem poglavlju, problem bojanja zida se može „razbiti“ na 3 manja problema, prvi od kojih je problem ostavljanja traga na zidu. Kako rješenje ovog problema nema nikakav utjecaj na funkcionalnost programa, odabrani su obični crtači linija (eng. *Line Renderer*) za crtanje poteza kista jer je takav način crtanja jednostavan za implementirati, a ne utječe značajno na performanse. Kad robot počne bojati zid, tj. kada spusti kist na površinu zida, u prvu točku linije se sprema položaj kista u tom trenutku, a druga točka se ažurira sa svakim mijenjanjem položaja kista. Kada robot prepozna da je naišao na otvor u zidu ili na vertikalni kraj zida, druga točka linije se postavlja na trenutnu poziciju kista i linija tog crtača se više ne mijenja (Slika 4.6.).



Slika 4.6. Druga točka linije je postavljena na zadnju poziciju na zidu kada robot podigne kist

Drugi dio problema je kretanje robota uz bojanje i prepoznavanje horizontalnih krajeva zida. Kako bi se simulirao računalni vid robota, korišteni su Unity-evi *Raycast*-i, koji su bili postavljeni oko kista, a sve zrake *Raycast*-a koje se koriste prikazane su na slici 4.7.



Slika 4.7. Sve korištene zrake i njihove oznake

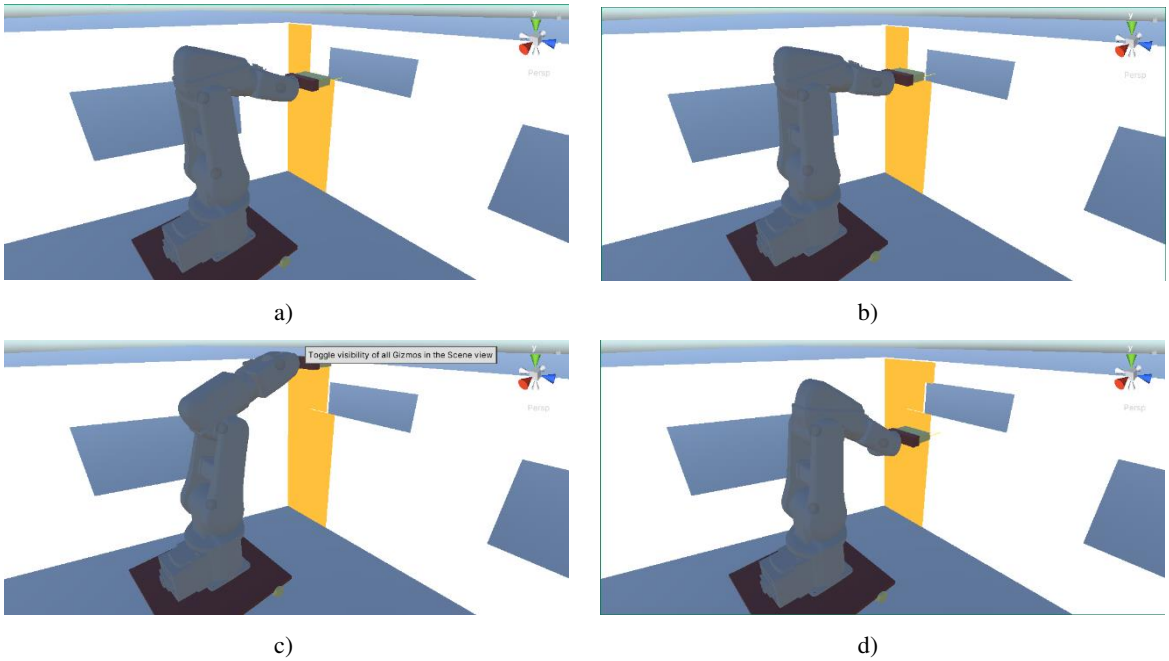
Zraka koja je postavljena nasred kista, kada je kratka se koristi za detekciju blizine zida (z1), a kada je duga, koristi se za pronalazak zida za bojanje (z2). Dvije zrake se nalaze sa strana kista i usmjerene su prema zidu. Njihova svrha je detekcija otvora u zidu (z3 i z4). Zadnje dvije zrake se isto nalaze sa strana kista, ali su okomite na prethodne dvije zrake i veoma su kratke, a služe za detekciju lijevog i desnog ruba zida (z5 i z6).

Detekcija vertikalnog ruba se ne provodi jer ga se ne može razlikovati od ruba vrata ili prozora. Pod sobe se također ne može detektirati, tako da se za duljinu poteza koji robot još mora povući koristi brojač. Na samom početku, kada se soba stvori i robot inicijalizira, robot pomoću zrake z2 pronađe zid koji treba bojati, ispravno se zarotira i krene prema njemu, pritom konstantno provjeravajući je li zraka z1 pogodila taj zid. Kada zraka z1 pogodi taj zid, robot se počinje translatirati ulijevo i stalno provjerava je li se zraka z5 sudarila s nekim zidom, jer ako jest, robot se nalazi na početku tog zida i može početi s bojanjem. Dok je u stanju bojanja zida, robot boja vertikalnim potezima, pritom koristi već spomenuti brojač da bi znao koliko još treba bojati. Kada brojač dođe do nule, robot se translatira udesno za duljinu kista, postavlja brojač na početnu vrijednost, te nastavlja bojati u suprotnom smjeru nego je to radio u prošlom potezu. Ako se tijekom translacije udesno zraka z6 sudari s nekim zidom, robot translaciju prekida, povlači liniju tu gdje je stao i prelazi na bojanje sljedećeg zida na način na koji je to objašnjeno u poglavlju 4.4.

Treći dio problema je prepoznavanje otvora u zidu, npr. vrata ili prozora, te prigodno reagiranje na njih. Kao što je već spomenuto, robot će otvore u zidu prepoznavati pomoću zraka z3 i z4, a kada se bilo koja od te dvije zrake ne sudari sa zidom robot obavlja sljedeći niz akcija:

1. Zaustavi kist.
2. Provjeri koja se zraka nije sudarila sa zidom.
3. Ako se desna traka nije sudarila sa zidom, robot se vraća ulijevo, ako se lijeva traka nije sudarila sa zidom robot se vraća udesno, inače samo preskače otvor i sljedeći koraci nisu potrebni jer se ništa ne treba ispravljati.
4. Robot putuje odnosno, translatira se paralelno sa zidom sve dok se obje zrake ne sudare sa zidom, te usput zbraja koliku je udaljenost proputovao.
5. Kada se obje zrake sudare sa zidom, robot nastavlja bojanje odakle je stao, pa sve do kraja zida.
6. Kada završi potez popravljanja, robot se vraća za udaljenost koju je izračunao u točki 4 u obrnutom smjeru od onog iz kojeg je došao.
7. Kada se vrati na prijašnji položaj, prelazi cijeli potez od početka do kraja i nastavlja dalje s bojanjem zida.

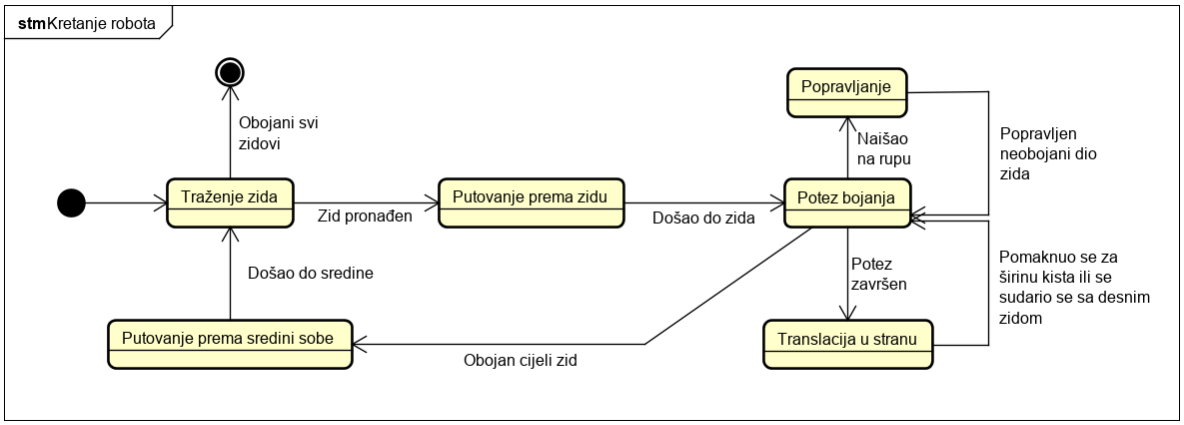
Navedeni algoritam će raditi kod svih zidova na kojima su susjedni otvori horizontalno udaljene jedne od drugih, ali i od zida, barem za širinu kista.



Slike 4.8. a) kist se zaustavlja jer z3 više ne pogađa zid, b) robot se translatira u lijevo dok z3 ne pogodi zid, c) robot povuče „popravni“ potez do kraja, d) robot se vrati na prijašnji potez i završi ga

4.4. Kretanje robota

Nakon što je implementacija svega potrebnog za bojanje jednog zida bila gotova, ostao je samo problem kretanja robota od zida do zida. Kako bi se riješio taj problem, robota se modeliralo kao automat stanja čiji je dijagram stanja prikazan na slici 4.9.

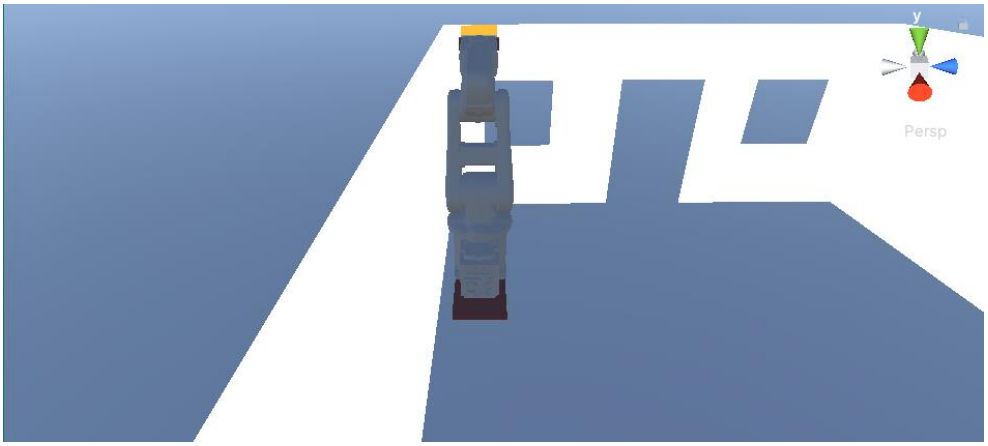


Slika 4.9. Dijagram stanja robota

Kada se cijela soba generira i FABRIK inicijalizira, robot prelazi u stanje traženja zida. U tom stanju će se okrenuti prema prvom zidu na koji zraka z2 naiđe i krenut će prema njemu. Kada robot dođe do zida, translatirat će se u lijevo sve dok ne dođe do lijevog susjednog zida, te će onda započeti s bojanjem. Nakon svakog poteza, translatirat će se u desno za širinu kista, a ako usred poteza naleti na otvor u zidu ispravit će neobojane dijelove zida na način koji je opisan u poglavlju 4.3. Ako tijekom translacije udesno robot naleti na desni susjedni zid, on se zaustavlja, povlači zadnji potez, diže kist gore, ako je završio dolje, te se vraća se na sredinu sobe. Ako nije obojao cijelu sobu, robot će se zarotirati za kut koji je dobiven formulom (1), te se ponovno vratiti u stanje traženje zida i postupak se ponavlja, inače robot ostaje na sredini i soba je obojana.

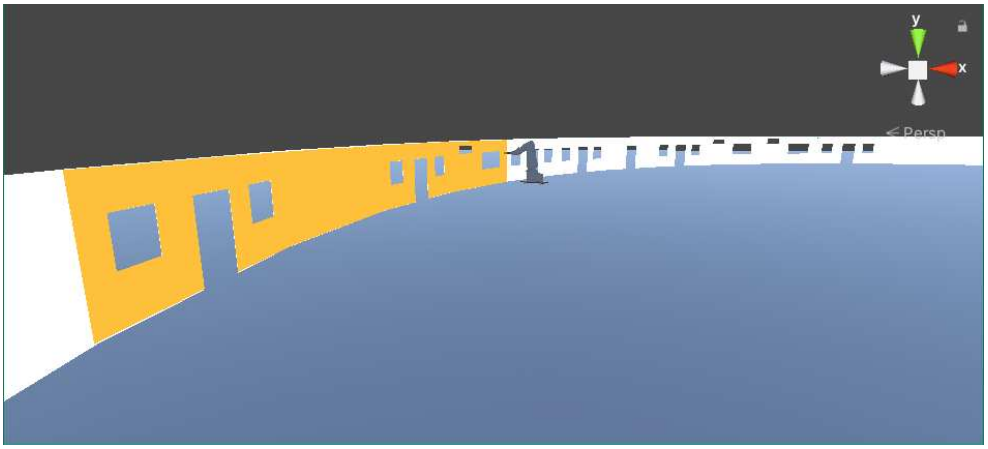
5. Rezultati

Navedeno rješenje radi na širokom rasponu broja zidova. Manji problemi se javljaju kod zidova pod pravim kutom, tj. kada su generirane sobe s četiri zida, jer robot fizički ne može obojati dio zida koji je bliži kutovima, kao što se može vidjeti na slici 5.1.

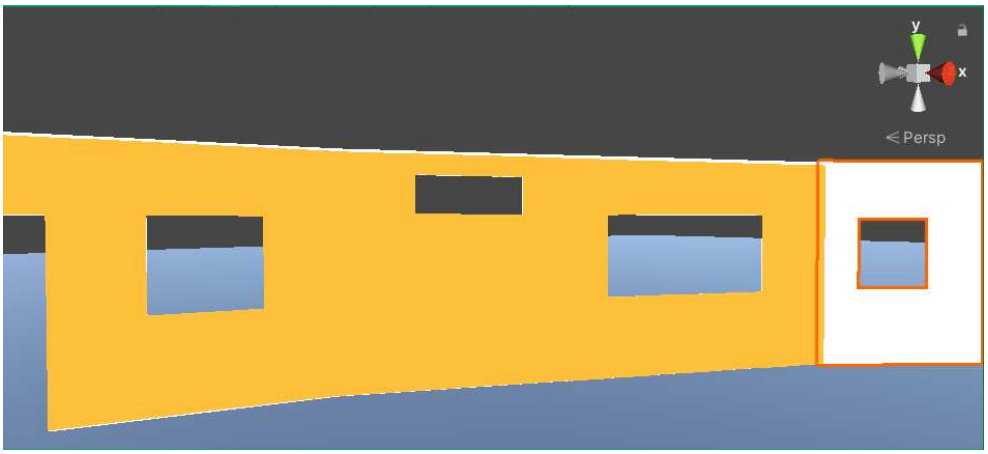


Slika 5.1. Robot je sudaren sa zidom i ne može bojati dalje ulijevo

Robot dosta dobro boja (približno) okrugle sobe (Slika 5.2.), ali kod takvih soba s većim brojem zidova dolazi do otežane detekcije krajnjih rubova zida. To je očekivano jer s većim brojem zidova raste i kut između njih, pa je onda sljedeći zid sve teže razlikovati od nastavka prethodnog, što je vidljivo na slici 5.3.

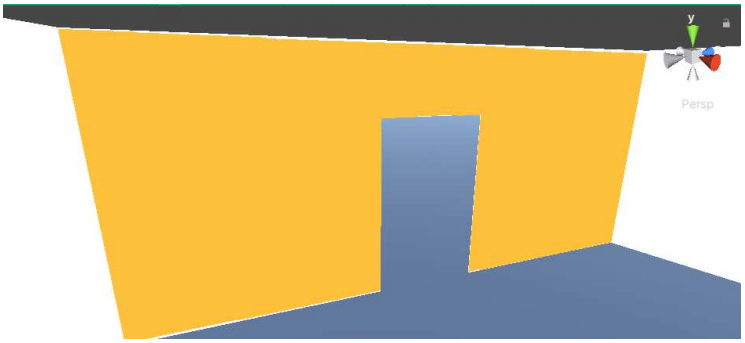


Slika 5.2. Bojanje okrugle sobe



Slika 5.3. Pogreške pri bojanju okrugle sobe s velikim brojem zidova

Valja napomenuti i da zbog neujednačenog odbrojavanja brojača kista, robot zna povući neravnomjerno duge poteze, ali to se može popraviti smanjenjem brzine bojanja, kako bi se kist preciznije micao i preciznije smanjivao brojač. Iz toga možemo zaključiti da je urednost bojanja obrnuto proporcionalna brzini poteza kistom, što je i vidljivo na slikama 5.4.

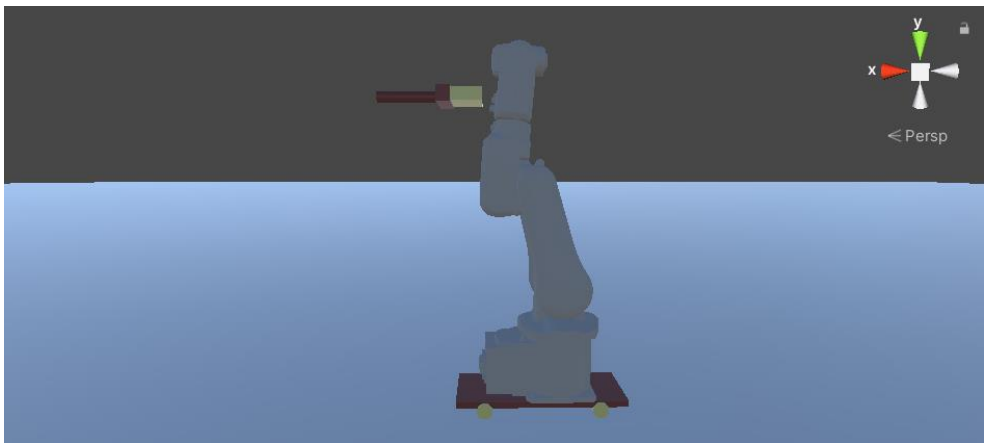


a)

b)

Slike 5.4. a) Zid obojan brzinom poteza v, b) Zid obojan brzinom poteza 4*v

Što se tiče inverzne kinematike, algoritam FABRIK funkcionira uspješno za veliki dio poteza kista, no kod nekih dijelova poteza dolazi do manjih trzaja koji se pojavljuju zbog toga što algoritam ne traži točnu poziciju nego poziciju koja je odmaknuta od ciljne za veličinu tolerancije. Ovaj problem se može popraviti dodatnim smanjenjem tolerancije ili promjenom algoritma inverzne kinematike koji se koristi. Valja napomenuti da odluka da kist predstavlja ciljnu točku FABRIK algoritma uzrokuje nepoželjna ponašanja u sveukupnom sustavu simulacije. Kada se ciljna točka nalazi u području koji robot ne može dohvatiti (zbog rotacijskih ograničenja ili duljine robota) kist se odvoji od robotske ruke (Slika 5.5.), no to ne uzrokuje velike probleme pri bojanju zida jer se cijeli potez bojanja nalazi unutar radnog prostora robota i ne krši nijedno rotacijsko ograničenje.



Slika 5.5. Kada je kist postavljen u položaj koji završni čvor ne može dosegnuti, on se odvoji od ruke

Zaključak

Inverzna kinematika je matematički proces računanja parametara zglobova koji su potrebni da bi se postavio kraj kinematičkog lanca (npr. kostur animiranog lika ili robotsku ruku) na neku zadanu poziciju, a da bi se pri tome očuvali svi prijašnji odnosi između zglobova (njihova udaljenost i/ili ograničenja rotacija).

U okviru ovog rada je model robota IRB 120 animiran metodom inverzne kinematike kako bi obojao nasumično generiranu sobu. Glavni problem rada je bila implementacija inverzne kinematike na modelu robotske ruke. Iz rezultata se može vidjeti da je korištenje algoritma FABRIK za rješavanje takvog problema vrlo jednostavno, te nije računski zahtjevno, no iterativna priroda tog algoritma nekad vraća konačne pozicije završnog čvora koje nisu sasvim točne. Sporedni problemi su bili generiranje i bojanje sobe. Algoritam generiranja sobe stvara prostorije koje izvrsno mogu testirati uspješnost algoritma bojanja, ali općenitost testiranja će uvelike ovisiti o modelima zidova koje se generatoru ponude. Algoritam bojanja sobe uspješno radi na velikom broju problema, ali ima manje nedostatke s točnošću detekcije ruba zidova koji se mogu popraviti usporavanjem kretnje robota, te usporavanjem kretnje kista.

Literatura

[1] Unity Home Page: https://unity.com; pristupljeno 12.6.2020.

[2] Blender Home page: https://www.blender.org; pristupljeno 12.6.2020.

[3] Andreas Aristidou, Joan Lasenby: FABRIK: A fast, iterative solver for the Inverse Kinematics problem; Graphical Models, 73. svezak, 5. izdanje, rujan 2011., str. 243-260, http://www.andreasaristidou.com/publications/papers/FABRIK.pdf; pristupljeno 12.6.2020.

[4] Mecademic: „What are singularities in a six-axis robot arm?“, https://www.mecademic.com/resources/Singularities/Robot-singularities; pristupljeno 12.6.2020.

[5] ABB: Product specification IRB 120, Document ID: 3HAC035960-001, 16.9.2019., https://library.e.abb.com/public/7139d7f4f2cb4d0da9b7fac6541e91d1/3HAC035960%20PS%20IRB%20120-en.pdf; pristupljeno 12.6.2020.

[6] Andreas Aristidou, Yiorgos Chrysanthou, Joan Lasenby: Extending FABRIK with model constraints; 27. svezak, 1. izdanje, veljača 2016., str. 35-57, https://www.researchgate.net/publication/271771862_Extending_FABRIK_with_model_constraints; pristupljeno 12.6.2020.

[7] Željka Mihajlović: „Unaprijedna i inverzna kinematika", Fakultet elektrotehnike i računarstva u Zagrebu: http://www.zemris.fer.hr/predmeti/ra/predavanja/; pristupljeno 12.6.2020.

Primjena inverzne kinematike u animaciji virtualnog robota pri izvođenju zadataka

Sažetak

Animiranje modela je ključan dio oblikovanja virtualnih okruženja. Mnogi modeli u današnjim virtualnim iskustvima su animirani pomoću unaprijed definiranih statičnih animacija, koje nekad ne daju željene rezultate. Proceduralno se koristi za dobivanje raznovrsnijega skupa animacija koje će se različito izvoditi u ovisnosti o okolini i stanju animiranog modela tijekom izvođenja programa. Postoji mnogo pristupa ostvarivanju takvih animacija, a jedan od njih je inverzna kinematika. U okviru rada, razmatraju se osnove inverzne kinematike, te primjena algoritma FABRIK kod proceduralnog animiranja robota pri izvođenju zadataka. Kao primjer primjene osmišljena je i unutar Unity Engine-a programski izvedena simulacija bojanja zidova u 3D virtualnom okruženju.

Ključne riječi: proceduralno animiranje, inverzna kinematika, algoritam FABRIK, Unity Engine

Application of inverse kinematics in animating a task-performing virtual robot

Abstract

Model animation is an essential part of designing virtual environments. Many models in today's virtual experiences are animated using predefined static animations, which sometimes don't necessarily produce wanted results. Procedural animation is used for creating a more diverse set of animations that will behave differently depending on the environment and state of the animated model. There are many methods of creating such animations, and one of those is using inverse kinematics. Within this work, the fundamentals of inverse kinematics, and application of FABRIK algorithm in procedural animation of a task-performing robot are discussed. As an example of application, a programmatically performed simulation of wall painting in a 3D virtual environment was designed within Unity Engine.

Keywords: procedural animation, inverse kinematics, FABRIK algorithm, Unity Engine