Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Vrtačnik

Uporaba inverzne kinematike pri proceduralno generiranih animacijah

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Iztok Lebar Bajec

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Razvijte programski animacijski sklop sposoben predvajanja vnaprej režiranih animacijskih sekvenc skeletnih figur. V nadaljevanju preučite iterativni hevristični pristop inverzne kinematike z imenom FABRIK. Slednjega nato uporabite za proceduralno generiranje sekundarnih akcij, ki s prilagajanjem okolju plemenitijo animacijske sekvence. Rezultate kritično komentirajte.

Zahvaljujem se mentorju izr. prof. dr. Iztoku Lebarju Bajcu, prijateljem in družini, ki so mi pomagali pri izdelavi diplomske naloge.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija za izbrano diplomsko temo 1

1.2 Prispevki diplomske naloge 2

2 Vrste animacij v računalniški grafiki **3**

2.1 Slikovna animacija 3

2.2 Primitivni načini animacije 4

2.3 Kompleksna animacija ali morfne tarče 4

2.4 Skeletna animacija 7

3 Skeletni animacijski sistem **11**

3.1 Priprava 3-dimenzionalnega modela 12

3.2 Izvoz datotek 14

3.3 Konfiguracija animacijskih akcij 15

3.4 Nalaganje animacijskih datotek 17

3.5 Časovnica animacijskih poz 19

3.6 Računanje transformacij posameznih sklepov 20

3.7 Končna vizualizacija 25

4 Inverzna kinematika **27**

4.1 Algoritem FABRIK 27

4.2 Drugi pristopi 32
4.3 Več končnih omejevalnih točk 33
4.4 Uporaba rezultatov za generiranje animacijske poze 39

5 Proceduralna generacija animacijskih poz **43**
5.1 Fizikalna knjižnica 44
5.2 Polproceduralne poze 44
5.3 Fizikalno dinamično krilo 45
5.4 Sklanjanje pred ovirami 47
5.5 Drsanje rok po bližnjih ovirah 48

6 Vizualizacija **49**
6.1 Struktura v ozadju 49
6.2 Risanje 50
6.3 Postprocesiranje 51

7 Rezultati **57**
7.1 Problemi 59

8 Sklepne ugotovitve **61**

Literatura **64**

Seznam uporabljenih kratic

kratica	angleško	slovensko
IK	inverse kinematics	inverzna kinematika
FABRIK	forward and backward reaching inverse kinematics	dvosmerno sežna inverzna kinematika
ASSIMP	open asset import library	odprtokodna knjižnica za uvoz datotek
BVH	Biovision hierarchy	Biovision hierarhija
OpenGL	open graphics library	odprtokodna grafična knjižnica
GLM	OpenGL Mathematics library	OpenGL matematična knjižnica
HDR	High dynamic range	Visok dinamični razpon
LDR	Low dynamic range	Nizek dinamični razpon

Povzetek

Naslov: Uporaba inverzne kinematike pri proceduralno generiranih animacijah

Avtor: Matic Vrtačnik

V diplomski nalogi smo v jeziku C++ s pomočjo algoritma inverzne kinematike FABRIK v kombinaciji s fizikalno knjižnico Bullet Physics implementirali sistem skeletne animacije z dinamično proceduralnim generiranjem animacijskih poz (*angl. animation pose*) v realnem času. Najprej smo se lotili pregleda tipov animacij v računalniški grafiki in opisali njihove prednosti ter slabosti. Temo smo nato poglobili na implementacijo sistema skeletne animacije. Za tem smo povzeli osnove inverzne kinematike in opisali algoritem FABRIK. S pridobljenim znanjem smo se nato lotili še proceduralne generacije animacij s pomočjo fizikalne knjižnice. Na koncu smo govorili še o sami vizualizaciji poz skeletne animacije in poleg zaznanih problemov opisali še možne izboljšave sistema.

Ključne besede: računalniška grafika, OpenGL, skeletna animacija, inverzna kinematika, proceduralna generacija.

Abstract

Title: Using inverse kinematics for procedurally generated animations

Author: Matic Vrtačnik

In this diploma thesis we used the FABRIK inverse kinematics algorithm in combination with physics library Bullet and C++ programming language to create a skeletal animation system with the ability to generate dynamic and procedural animation poses in real time. At the beginning we talked about different types of animations found in computer graphics and their pros and cons. We then focused on the implementation of the skeletal animation system. After that we talked about inverse kinematics and described the FABRIK algorithm. With the acquired knowledge we then proceeded to procedural generation of body poses with the help of the Bullet physics library. Finally, we talked about the visualization of skeletal animation and in addition to the problems encountered, we also described possible improvements to the entire system.

Keywords: computer graphics, OpenGL, skeletal animation, inverse kinematics, procedural generation.

Poglavje 1

Uvod

Diplomska naloga se loti problema pisanja robustnega sistema skeletne animacije, ki je sposoben dinamično spreminjati opise posameznih poz za uporabo pri proceduralni generaciji. Vključuje tudi pristop uporabe fizikalne knjižnice za določanje končnih točk (*angl. end-effectors*) inverzne kinematike, s katero lahko nato glede na kolizijo v okolici sveta proceduralno zgeneriramo iz obstoječih načrtovanih poz nove, razmeroma primerne poze (tu ne upoštevamo omejitev rotacij posameznih sklepov). Z uporabo teh tehnik se lotimo tudi generiranja poz med izvajanjem raznih animacijskih akcij in tako iz njih ustvarimo nove, razširjene akcije. Iz pridobljenih znanj o premikih animirane entitete ustvarimo tudi popolnoma dinamične, fizikalno simulirane dele teles. Naloga opisuje tudi končni vizualizacijski vidik delovanja grafičnega pogona. Rezultati diplomske naloge z navodili za namestitev in uporabo so javno objavljeni na Github repozitoriju.[1]

1.1 Motivacija za izbrano diplomsko temo

Za računalniško grafiko se zanimam že vrsto let in tudi v prostem času delam na svojem grafičnem pogonu. Zaradi manjšega znanja o tematiki animacij in potrebe po solidnem sistemu v osebnem projektu se mi je zdela taka naloga

[1] https://github.com/MaticVrtacnik/ProceduralnaAnimacija

izredno zanimiva, saj vključuje vrsto različnih problemov, ki jih do zdaj še nisem preveč resno obravnaval. Ker po naravi nisem najbolj umetniški tip, sem se moral reševanja problema lotiti bolj s programerskega stališča in kar se da proceduralno.

1.2 Prispevki diplomske naloge

Za razliko od tradicionalnega pristopa shranjevanja poz skeletne animacije smo posamezne animacijske poze izločili iz glavne datoteke s podatki o ogliščih. V glavni datoteki ohranimo vse podatke kot do zdaj, vendar pri izvozu poskrbimo, da ta ne vsebuje nobenih animacijskih poz. Same individualne animacijske poze pa za razliko od standardne implementacije shranimo v ločene datoteke formata BVH. Tu lahko s pomočjo konfiguracijske datoteke večkrat uporabimo isto animacijsko pozo in tako naredimo celoten animacijski sistem zelo učinkovit in fleksibilen. Več o tem si je možno prebrati v poglavju 3.

Dodali smo tudi nov dinamičen pristop za generiranje proceduralne animacije z uporabo fizikalne knjižnice Bullet Physics. Najprej smo z uporabo sil vzmeti usešno zgenerirali rotacijski del transformacij dinamičnega krila. Tako smo z uporabo ročno nastavljenih parametrov vzmeti prišli do želenih rezultatov, vidnih ob premiku in vrtenju telesa. Pri krilu pa nismo imeli želje po realistični simulaciji tkanine, zato smo blago na krilu ohranili togo po dolžini. Več o tem je v nalogi opisano v odseku 5.3. Fizikalno knjižnico smo nato uporabili še za iskanje bližnjih kolizij animacijskega telesa. Na podlagi teh smo nato lahko z uporabo algoritma za inverzno kinematiko FABRIK [21] iz osnovnih poz proceduralno zgenerirali še nekaj novih animacijskih akcij. Za začetek smo animirali sklanjanje ob premikanju med ovirami. Z uporabo kolizijske krogle pa smo ustvarili tudi animacijo dotikanja bližnjih ovir z rokami ob hoji. Več o proceduralni generaciji si lahko preberete v poglavju 5.

Poglavje 2

Vrste animacij v računalniški grafiki

Animacije so velik del računalniške grafike in so tako posledično dodobra raziskane. Zaradi tega jih lahko delimo na več različnih vrst glede na razne optimizirane implementacije in njihove osnovne namene. V začetnih fazah razvoja se odločamo, katera od teh bo igrala v našem projektu vodilno vlogo, lahko pa jih za različne rezultate tudi kombiniramo med seboj. Pri naši diplomski nalogi smo se osredotočili izključno na skeletno animacijo, saj je ta ena izmed najbolj programabilnih obstoječih implementacij.

2.1 Slikovna animacija

Pri 2-dimenzionalnih okoljih za animiranje objektov namesto premika oglišč uporabimo prehode med slikami. Za tak tip animacije se uporablja zaporedje sličic, ki predstavljajo animacijske poze našega objekta. Za shranjevanje teh se pogosto uporabljajo večje teksture, kamor lahko te zaporedno shranimo v velikem številu. V 2-dimenzionalnem svetu lahko s takimi animacijami pridemo do odličnih rezultatov, vendar za naše namene, poleg preobrazb barve posameznih ploskev, niso preveč uporabne.

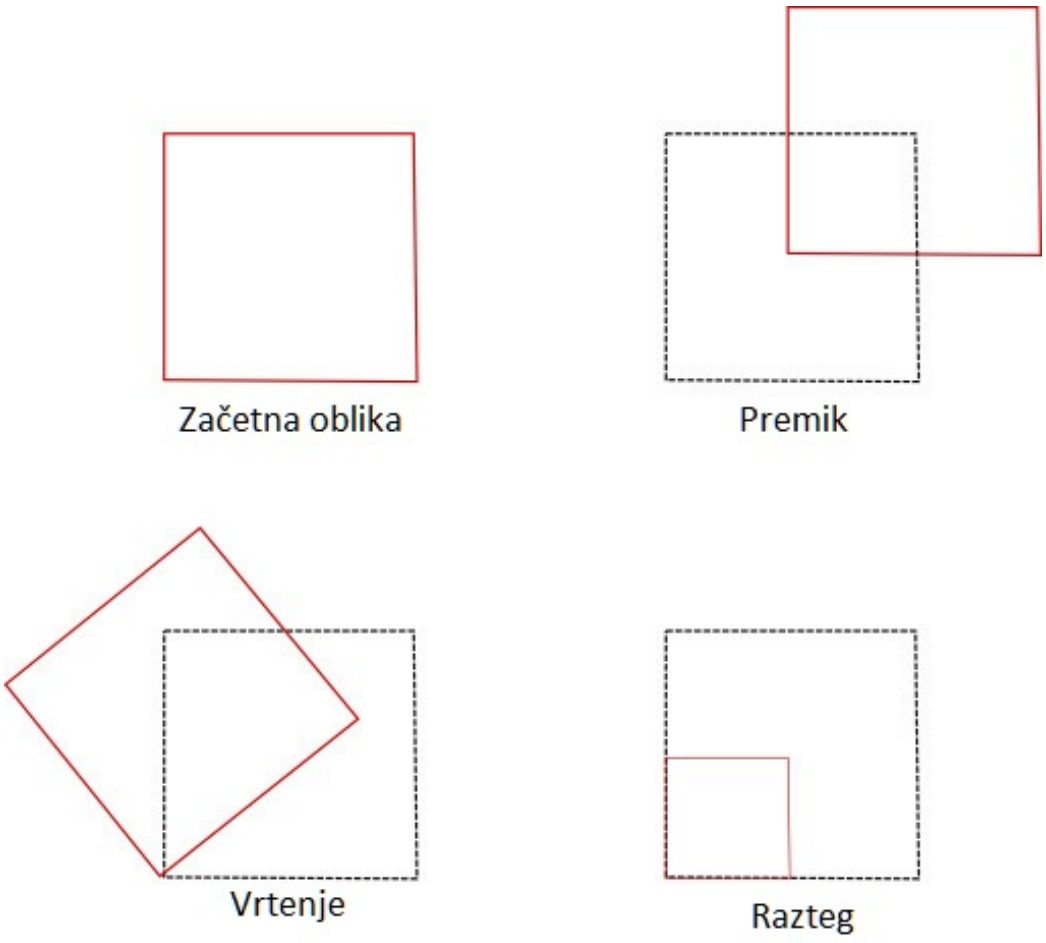
Slika 2.1: Atlas sličic različnih poz za uporabo pri slikovni animaciji.

2.2 Primitivni načini animacije

Ko govorimo o animaciji v računalniški grafiki, lahko začnemo že s preprostimi transformacijami, kot so premik, vrtenje in razteg (slika 2.2). Do teh lahko po uspešni vizualizaciji geometrijskih likov pridemo že s preprostimi spremembami v odvisnosti od časa. Na primer narisanemu trikotniku lahko s časovno spremenljivko spreminjamo lokacijo po eni ali več oseh. S pomočjo osnovnih kotnih funkcij lahko na podoben način točke vrtimo okrog koordinatnega izhodišča. Ena izmed popularnih animacij je tudi razteg oz. povečevanje in zmanjševanje velikosti likov. Poleg teh osnovnih transformacij obstaja še nekaj drugih, manj uporabljenih preobrazb.

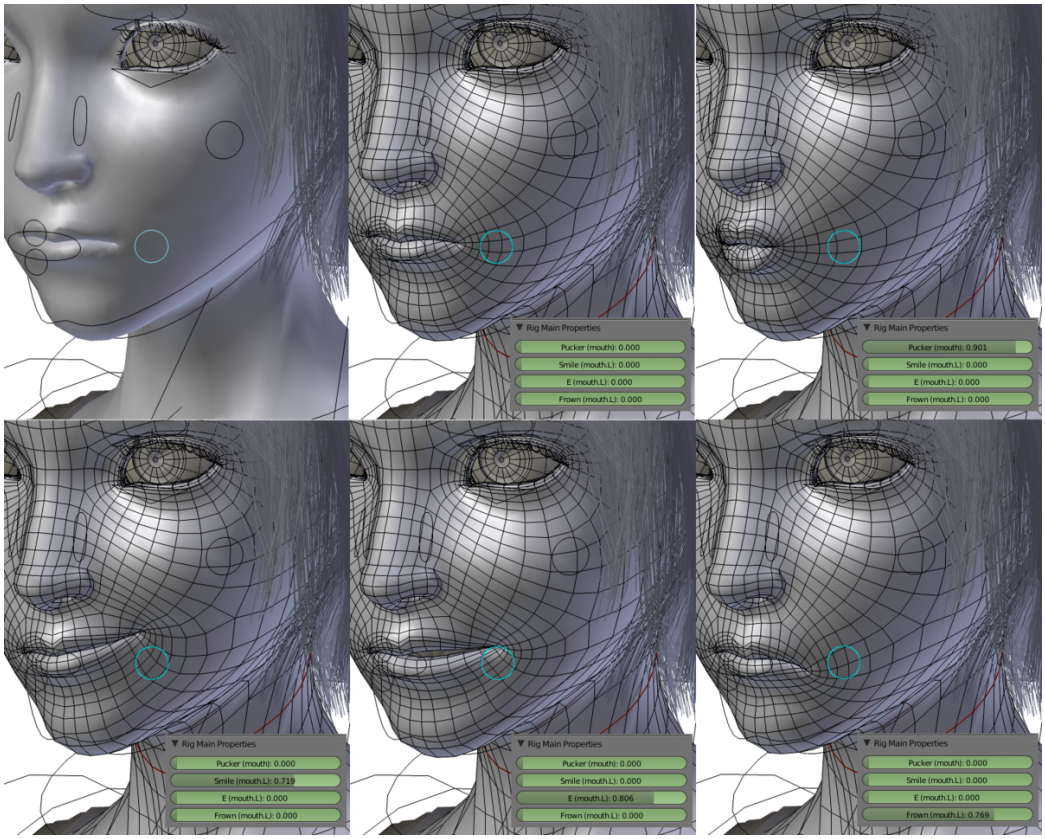
2.3 Kompleksna animacija ali morfne tarče

Ko imamo opravka samo z majhno količino oglišč ali s preobrazbo celotne množice točk hkrati, si lahko v nekaterih primerih še privoščimo ročno programiranje animacij. Ko pa v veliki mreži oglišč želimo z njimi manipulirati bolj natančno, to storimo z uporabo programa za urejanje 3-dimenzionalnih modelov. Tu si lahko shranimo več različnih položajev oglišč kot naše animacijske poze. Ko končamo z urejanjem poz, lahko izvozimo animacijo v zaporedje datotek navadnih 3-dimenzionalnih modelov, le da se lokacije oglišč teh razlikujejo po položajih glede na nastavljene poze. Predvajanje take animacije poteka z menjavanjem med oglišči 3-dimenzionalnega modela. Zaradi prostorske kompleksnosti shranjevanja morfnih tarč si lahko pri predvajanju



Slika 2.2: Osnovni tipi transformacij v računalniški grafiki.

animacije zelo pomagamo z interpolacijo. Pri animiranju s takim pristopom bi shranili samo ekstreme animacij, kjer se oglišča lahko predvidljivo gibljejo proti naslednji animacijski pozi. Najbolj pogosta uporaba take animacije v praksi je prehod med čustvenimi obraznimi izrazi [13], kjer bi zaradi kompleksnosti mišične strukture to zelo težko dosegli s skeletno animacijo. Primer nekaj takih poz lahko vidimo na sliki 2.3. Dokaj pogosto uporabo najdemo tudi pri simulacijah mrež oglišč, za katere obstaja velika možnost za dinamične interakcije. Pod take sodijo na primer površine, narejene iz blaga.



Slika 2.3: Primer morfnih tarč kompleksne animacije za prikaz in prehod čustvenih izrazov obraza.

Glavna dobra lastnost take animacije je natančnost, saj lahko animiramo mrežo oglišč praktično neomejeno. Paziti moramo, da take animacije ne uporabljamo na telesih s prevelikim številom oglišč (lahko jih poskušamo tudi zmanjšati z uporabo orodij za redukcijo). Tukaj poskušamo zaradi prostorske zahtevnosti zmanjšati tudi dolžine svojih animacij oziroma število morfnih tarč, v katere želimo prehajati. Če delamo z animacijo, ki vsebuje več zaporednih poz, med katerimi se premikamo hitro, to poskušamo skrajšati ali pa jo lahko razbijemo na vmesne stopnje in iz njih naredimo krožno predvajanje, ki mu lahko dodamo tudi naključno zaporedje interpolacijskih ciljnih poz, da je animacija na videz daljša ali bolj zapletena, kot je dejansko shranjena na disku.

2.4 Skeletna animacija

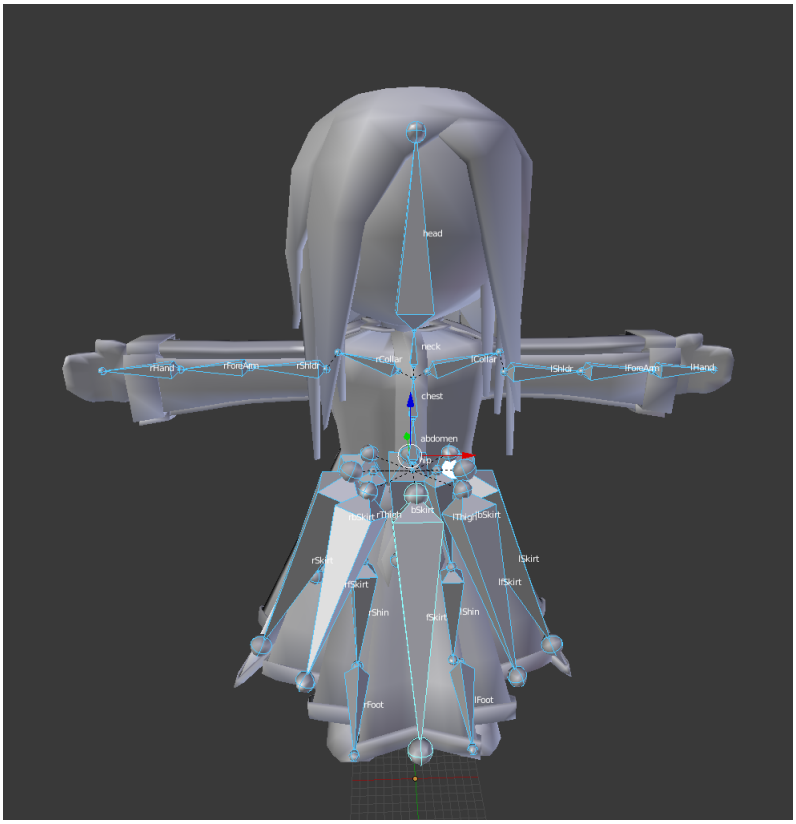
Za bolj kompleksne mreže oglišč, katerih premiki se lahko predstavijo z zbirko transformacijskih sklepov, se uporablja skeletna animacija [16]. Ta je sestavljena iz množice oglišč, ki poleg lokacije vsebujejo tudi animaciji primerne podatke. Med te spada množica parov identifikacijskega števila transformacijskega sklepa in neke uteži, s katero določimo, kakšen vpliv ima položaj sklepa na transformacijo tega oglišča. Za te uteži je priporočljivo, da se skupaj seštejejo v največ število 1, saj v nasprotnem primeru lahko pride do nepredvidljivih in s tem neželenih rezultatov, ko bomo obravnavano mrežo izrisali. Poleg tega dodamo oglišču še podatke, uporabljene za osvetljevanje, in pa odvisno od uporabljenega grafičnega pogona še kaj. Drugi del skeletne animacije je hierarhična struktura sklepov in njim pripadajoče kosti. V primeru telesa po navadi tako drevesno strukturo začnemo pri medenični kosti, nato skelet razvejimo na zgornji trup in vsako izmed nog. Vsak izmed sklepov vsebuje podatke o svojem staršu, ki bo v našem primeru vedno samo eden, in o sklepih otrok, na katere je povezan (primer skeletne hierarhije lahko vidimo na sliki 2.4). Za poznejše računanje transformacij oglišč potrebujejo sklepi zaradi svoje porazdelitve po mreži tudi nek način, kako iz preobrazbe položaja v pozi postavljanja (*angl. rigging*) priti do koordinatnega izhodišča, od koder lahko uporabimo spremembo transformacije kot dejansko preobrazbo oglišča (*angl. skinning*). Primer preproste implementacije skeletne animacije lahko najdemo v viru [17].

V povezavi s skeletno animacijo poznamo dva načina računanja končnih transformacij posameznih sklepov:

- **Direktna kinematika (*angl. forward kinematics*)**
  Z uporabe direktne kinematike s pomočjo množenja zaporednih transformacij posameznih vozlišč v hierarhiji izračunamo končne lokacije njenih listov.

- **Inverzna kinematika (*angl. inverse kinematics*)**
  Za razliko od direktne kinematike pri inverzni namesto računanja končnih

lokacij listov hierarhije izračunamo eno izmed možnih postavitev sklepov verige, katere konec sega do neke želene končne točke v svetu. Torej, kakor že ime pove, deluje popolnoma nasprotno od direktne kinematike.



Slika 2.4: Slika prikazuje hierarhijo kosti skeletne animacije.

Velika prednost skeletne animacije je zadovoljiva natančnost in končen videz za relativno majhno ceno računanja. Zelo dobra je tudi pri shranjevanju velikega števila animacijskih poz na zelo ekonomičen način, saj za vsako izmed njih shranjujemo samo podatke o premiku, vrtenju in po možnosti raztegu posameznih sklepov, katerih število v večini primerov ne presega 100. Če v naši aplikaciji ne potrebujemo popolnega nadzora nad položajem skeleta na vsaki sliki izrisa, lahko večino poz iz datoteke izpustimo in za animacijsko akcijo posnamemo samo tiste poze, ki določajo končna stanja transformacij.

Na primer samo desno in levo pozo hoje telesa (za boljše rezultate seveda lahko uporabimo tudi malo več vmesnih poz). Tu se izkaže skeletna struktura kot zelo uporabna, saj lahko za vmesne transformacije interpoliramo med posameznimi sklepi končnih preobrazb poz na nivoju samega sklepa in tako lahko izračunamo končne transformacije v odvisnosti tudi od položaja starševskih sklepov v hierarhiji.

Skeletna animacija je odlična tudi zaradi hitrosti samega dela animatorjev. Na začetku se sicer porabi nekaj časa, da se dodajo kosti in se te z utežmi prilepi na posamezna oglišča, vendar se po tem procesu delo drastično pohitri in nastavljanje poz lahko postane zelo hitro. Tukaj je vredno omeniti tudi uporabo zajemanja gibanja, ki se zelo pogosto uporablja v industriji filmov in videoiger.

Slabe lastnosti pa se prikažejo pri omejitvah takih transformacij, in sicer pri neželenih raztegih ob slabo uteženi množici oglišč (nekatera izmed oglišč se lahko izmenično premikajo eno čez drugo ali pa celo nimajo uteži in tako kot končni rezultat dobimo točko v sredini koordinatnega izhodišča). Za bolj kompleksne mišične premike potrebujemo tudi ogromno različnih kosti (npr. obrazne izraze). Slabost take animacije je tudi, da se animacijski sistem pri velikem številu sočasno risanih neodvisno animiranih teles ali skeletov z ogromnim številom sklepov opazno upočasni, česar si v primeru naše resnično-časovne aplikacije ne smemo privoščiti.

Poglavje 3

Skeletni animacijski sistem

Preden sploh začnemo govoriti o animacijskem sistemu, moramo razložiti par pojmov, ki jih bomo uporabili. Ko govorimo o animacijskih pozah, imamo v mislih množico transformacij posameznih sklepov skeletne hierarhije. Večkrat omenimo tudi animacijske akcije, ki predstavljajo specifično zaporedje animacijskih poz in trajanja prehodov med njimi (npr. pri animacijski akciji skoka imamo zaporedje poz: priprava na skok, odriv in dejanski skok).

Zaradi učinkovitosti in prilagodljivosti smo se pri našem animacijskem sistemu odločili za skeletno animacijo. Da lahko začnemo s samo implementacijo, moramo najprej poskrbeti za učinkovit in solidno strukturiran animacijski sistem, preko katerega bomo upravljali vse funkcije naših animiranih teles.

Ta mora biti sposoben spreminjati in nadgrajevati transformacije animacijskih poz, ki se že predvajajo, saj samo tako lahko poskrbimo, da bodo te res pravilne glede na vpliv zunanjih dejavnikov na animirano telo. Preko animacijskega sistema želimo predvajati zaporedje poz specifične animacijske akcije in od njega pričakujemo, da bo glede na podatke naloženega 3-dimenzionalnega modela in ključne poze vrnil pravilne transformacije za vsako od kosti v odvisnosti od trenutnega stanja našega skeletnega telesa. Na primer, če trenutno predvajamo animacijo hoje in v trenutku pride do

zahteve po skoku, mora sistem pravilno upoštevati, ali se trenutna animacija lahko prekine ali pa moramo počakati, da se zaporedje prejšnjih akcij dokončno izvede. Ob skoku je tako sistem sposoben akcijo hoje prekiniti in in nemudoma začeti s predvajanjem animacije skoka. Kjer pa bi bilo telo v stanju nepripravljenosti na skok, bi sistem znal tako akcijo zakasniti za toliko časa, da lahko ponovno skočimo.

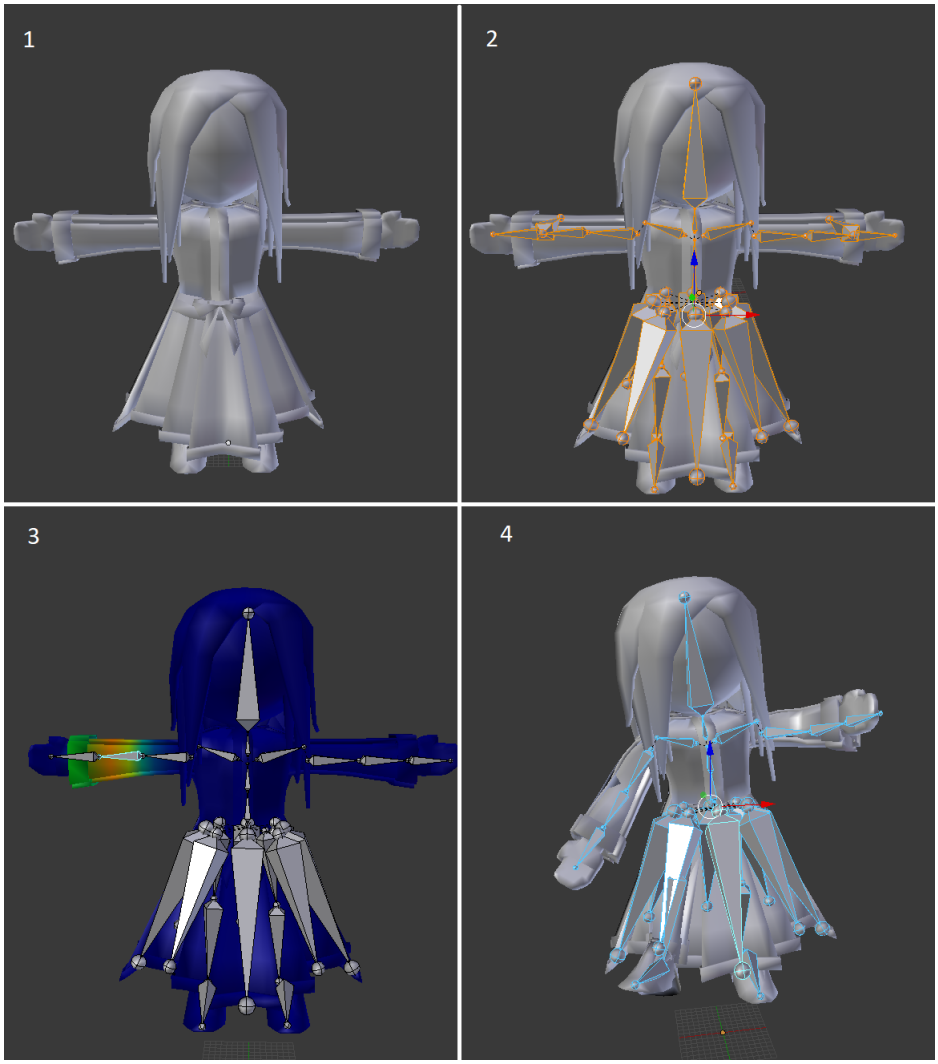
3.1 Priprava 3-dimenzionalnega modela

Z izdelavo modela, ki vsebuje skeletno animacijo, začnemo v programu za grafično 3-dimenzionalno modeliranje. V našem primeru smo uporabili orodje Blender, ki je odprtokodni in eden izmed najbolj priljubljenih programov za izdelovanje 3-dimenzionalnih modelov. Na sliki 3.1 si lahko ogledamo korake izdelave preproste poze z uporabo skeletne animacije.

1. Korak prikazuje naš končani 3-dimenzionalni model, ki vsebuje že vse podatke o ogliščih in ploskvah. Tu lahko vidimo model v začetni pozi s popolnoma iztegnjenimi telesnimi udi. S tako postavitvijo si olajšamo delo dodajanja kosti na primerna mesta in pozneje barvanja uteži oglišč.

2. Korak ponazori proces dodajanja primernih, s kostmi povezanih sklepov, ki se bodo uporabljali za premikanje množic oglišč pri izdelavi animacijske poze (*angl. rigging*). V tem koraku poskušamo naše kosti nadvse približati dejanskim meram, lokacijam in orientacijam 3-dimenzionalnega modela, saj bomo tako mnogo lažje nadaljevali pri naslednjih opisanih korakih.

3. Korak prikazuje uteži oz. območje oglišč, na katera bo vplivala transformacija izbrane kosti (*angl. skinning*). Na ploskve, obarvane čim bolj rdeče, bodo vplivi transformacije kosti bolj opazni, na modrih površinah pa ta ne bo uporabljena. Do teh uteži lahko pridemo z vgrajenimi orodji v programu za 3-dimenzionalno modeliranje, ki nam



Slika 3.1: Slika prikazuje glavne korake izdelave in animiranja skeleta 3-dimenzionalnega modela z uporabo programa za grafično modeliranje Blender.

bodo glede na postavitev kosti relativno na mrežo oglišč poskušala zgenerirati nadvse natančne uteži. Kljub visoki ravni uspešnosti algoritma pa si moramo v večini primerov uteži prilagoditi še ročno, ker se poleg nenatančnosti lahko zgodi tudi, da algoritem kakšno oglišče preprosto izpusti. Tako nanj ne vpliva nobena izmed kosti, kar pomeni, da bo pri računanju končnih lokacij oglišče zmeraj nastavljeno na koor-

dinatno izhodišče, ker nam transformacije zanj ne bo uspelo poiskati. Taka ohlapna lebdeča oglišča moramo v našem urejevalniku najti in jim ročno določiti pravilne pritrditvene kosti, saj se v praksi izkaže, da pri uporabi ničelnih uteži pride do neželjenih rezultatov izrisa.

4. Korak pa prikazuje vpliv animacijske poze skeleta na naš 3-dimenzionalni model. Od tu naprej lahko dodajamo več zaporednih poz in tako s primerno oteženim skeletom ustvarimo različne animacijske akcije. Med spreminjanjem poze skeleta lahko opazimo, da vrtenje starševskega sklepa premakne tudi vse iz njega izhajajoče sklepe (otroke) in tako ohranjamo realistične odnose med sklepi. Za naš animacijski sistem je to pomembno, saj moramo tudi mi poskrbeti, da bodo vse starševske transformacije vplivale na končne preobrazbe otrok.

3.2 Izvoz datotek

Po končanem delu v urejevalnem programu lahko 3-dimenzionalni model izvozimo v primerne datoteke. Za naš animacijski sistem načeloma velja, da so podatki o utežeh oglišč ločeni od samih animacijskih poz, vendar se v grafičnem pogonu lahko uporabijo tudi animacije, izvožene skupaj z datoteko, ki vsebuje podatke o ogliščih. Temu se sicer v našem sistemu želimo izogniti, saj se bodo v takem primeru vse animacijske poze posameznega 3-dimenzionalnega modela zapisale v skupno animacijsko akcijo. Zato je priporočeno, da se pred izvažanjem modela v urejevalniku začasno znebimo animacijskih poz tako, da vse poze izbrišemo iz časovnice in izvozimo datoteko, ki vsebuje samo skelet skupaj z ogliščnimi podatki. Za izvoz je v našem primeru najbolj priporočena uporaba formata FBX (.fbx). Ko smo naš model uspešno izvozili, se lahko lotimo še izvoza posameznih animacijskih poz.

Za posamezne, ločene, animacijske akcije in poze se v animacijskem sistemu uporablja format Biovision Hierarchical Data (.bvh), ki se je razvil z namenom uporabe zajema gibanja človeškega telesa. V ta format shranjujemo samo hierarhično zgradbo našega skeleta in preobrazbene zamike od

koordinatnega izhodišča za vsak sklep ter temu pripadajoče transformacije za eno ali več poz naše animacijske akcije. Po večini tukaj zaradi poznejše boljše prilagodljivosti izvozimo samo posamezne poze ekstremov, ki predstavljajo robne transformacije sklepov, med katerimi lahko še varno interpoliramo, ne da bi popačili animacijo (npr. za hojo izvozimo samo pozi zamahov z desno in levo polovico telesa). Sistem pa omogoča tudi višjo natančnost animacije z izvozom dodatnih zaporednih poz med ekstremi.

3.3 Konfiguracija animacijskih akcij

Preden lahko izvoženo skeletno animacijo uporabimo v sistemu, moramo ustvariti še nekaj konfiguracijskih datotek formata XML. Za nalaganje teh uporabimo odprtokodno knjižnico PugiXML [15]. S konfiguracijsko datoteko "bones.cfg" si pred delom z animiranim telesom nastavimo vzdevke kosti, s katerimi si bomo olajšali delo med programiranjem. Kosti, ki ne bodo imele nastavljenih vzdevkov, bodo interpretirane kot dinamične. Tem kostem pa bo ob nalaganju modela grafični pogon dodelil dinamična fizikalna telesa (trenutno to deluje samo za krilo, ki ima v programu nastavljeno obliko telesa in vedenje tega v odvisnosti od fizikalnih teles drugih kosti) in tako bo tudi premikanje oglišč take kosti odvisno od transformacije kolizijskega telesa.

Za vzdevke kosti našega novo ustvarjenega modela uporabimo konfiguracijsko datoteko z imenom "bones.cfg" s formatom:

```
<Bones>
<Bone name="hip" alias="HIP" />
<Bone name="abdomen" alias="ABDOMEN" />
<Bone name="chest" alias="CHEST" />
</Bones>
```

Najbolj pomembna konfiguracijska datoteka, brez katere naš animacijski sistem sploh ne more delovati, je datoteka "actions.cfg", v kateri so zapisana zaporedja prehodov animacijskih poz in čas prehoda iz prejšnje poze na trenutno v sekundah. Pri konfiguraciji animacijske akcije lahko določimo tudi,

koliko časa traja, da preidemo iz ene animacijske akcije v drugo, in ali je možno trenutno izvajano akcijo prekiniti z novo ali pa je treba počakati na njen konec. Vsaka izmed akcij potrebuje podatek o unikatnem imenu, saj tega v aplikaciji lahko potem uporabimo za določanje, katero od akcij želimo predvajati kot naslednjo. Če kateri izmed akcij ne določimo niti ene animacijske poze, se od nas pričakuje, da nastavimo pot do datoteke, ki vsebuje zaporedje poz celotne akcije. Primer zapisa nekaj akcij, ki jih uporabimo v aplikaciji:

```
<Action name="fancy" path="my_fancy_animation.bvh" cancelable="
   false" />
<Action name="stand" cancelable="true">
       <Keyframe path="stand.bvh" time="0.2" />
</Action>
<Action name="walk" cancelable="true" interpolation_duration="
   0.2">
       <Keyframe path="walk_right.bvh" time="0.35" />
       <Keyframe path="walk_left.bvh" time="0.35" />
</Action>
<Action name="jump" cancelable="false" interpolation_duration="
   0.05">
       <Keyframe path="crouch.bvh" time="0.1" />
       <Keyframe path="stand.bvh" time="0.1" />
</Action>
```

Poleg akcij imamo zraven tudi zapis o animacijskih pozah, ki jih nismo uporabili v nobeni izmed akcij. Te služijo zgolj za pomoč pri polproceduralnem generiranju novih poz obstoječe animacijske akcije glede na potrebe aplikacije. Primer uporabe dodatno definiranih animacijskih poz se lahko opazi pri generiranju poz akcije za hojo z uporabo interpolacije med pozama počasne hoje in hitrega teka v odvisnosti od hitrosti premikanja. Datoteka je videti podobna kot tista, ki vsebuje animacijske akcije, vendar se v tem primeru nastavi samo pot do datoteke z animacijsko pozo, imena pa bodo

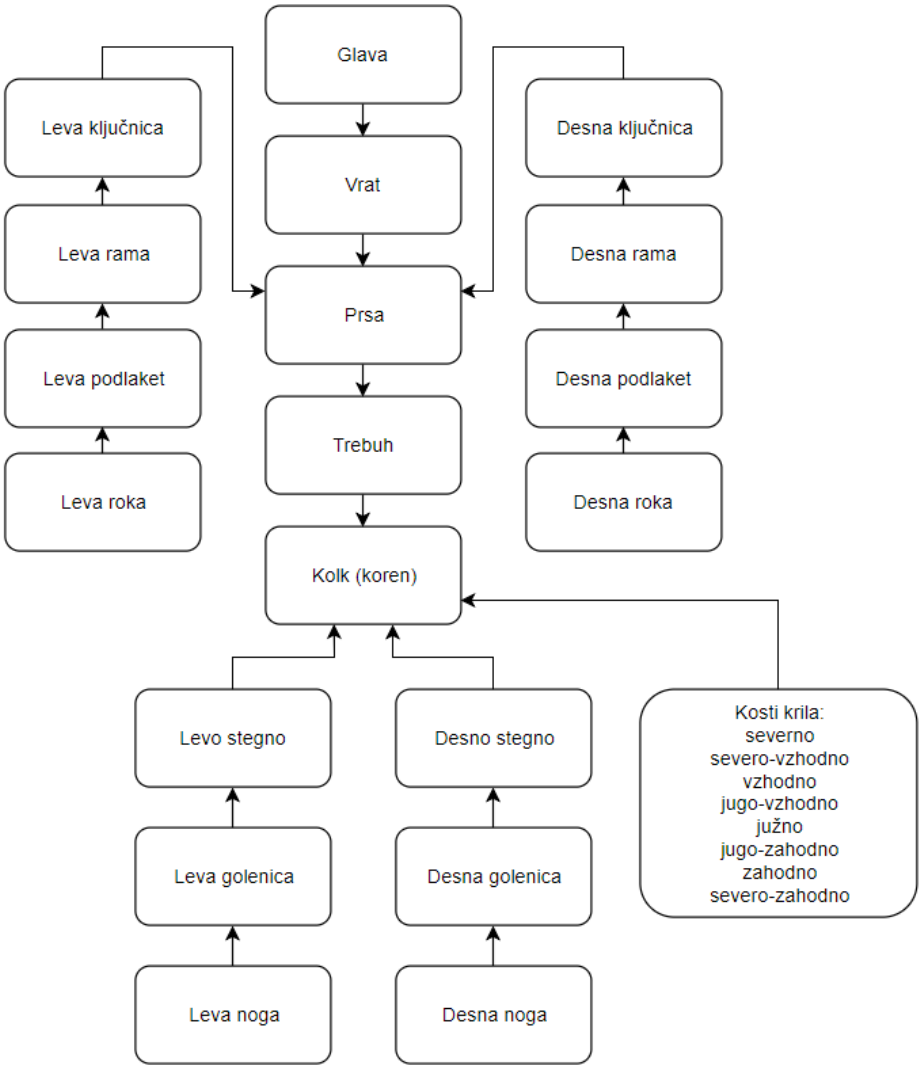
avtomatsko ustvarjena iz naslovov datotek. Primer formata zapisa znotraj konfiguracijske datoteke:

```
<Keyframe path="walk_slowest_right.bvh" time="0.35" />
<Keyframe path="walk_slowest_left.bvh" time="0.35" />
<Keyframe path="run_fastest_right.bvh" time="0.35" />
<Keyframe path="run_fastest_left.bvh" time="0.35" />
```

3.4 Nalaganje animacijskih datotek

Za nalaganje vseh modelov v grafičnem pogonu se uporablja odprtokodna knjižnica ASSIMP [2], s katero lahko naložimo vrsto različnih formatov, med drugimi tudi datoteke z našo skeletno animacijo. Ob klicu funkcije za dodajanje nove entitete v svet grafičnemu pogonu podamo pot do datoteke, ki vsebuje podatke o ogliščih in skeletni hierarhiji. Pogon lahko zatem iz prebrane datoteke ustvari scenski graf, iz katerega rekurzivno pridobi vse podatke o posameznih 3-dimenzionalnih modelih in njihovih materialnih teksturah. Med obravnavanjem scenskega grafa si grafični pogon zapomni tudi globalno transformacijo celotnega modela, saj se lahko zgodi, da ta uporablja drugačen koordinatni sistem kot mi. Zatem končno preveri, ali kakšno vozlišče v scenskem grafu vsebuje podatke o skeletni animaciji. Če najde kakšno od kosti, lahko s pomočjo rekurzije prepiše hierarhično strukturo skeleta iz scenskega grafa v svoj sistem. Pri rekurzivnem prehodu dobi vsa imena kosti, s pomočjo katerih lahko zgradimo slovar unikatnih številskih identifikatorjev, ki ga lahko pozneje uporabljamo za pridobitev vseh podatkov, povezanih z določeno kostjo. Pri pregledovanju scenskega grafa dobimo tudi vse podatke o utežeh vpliva posamezne kosti na oglišča modela. Če smo v datoteki kljub nasvetu, da tega ne počnemo, vseeno vključili zaporedje animacijskih poz, nam bo iz množice animacij scenskega grafa grafični pogon ustvaril novo animacijsko akcijo, v katero bo združil vse animacijske poze, ki jih vsebuje scenski graf za najdeno skeletno strukturo. Iz vseh pridobljenih podatkov scenskega grafa pogon sedaj ustvari novo entiteto, ki jo nato lahko vstavi v

simulacijski svet.



Slika 3.2: Slika prikazuje uporabljeno hierarhijo kosti za skeletno animacijo našega telesa.

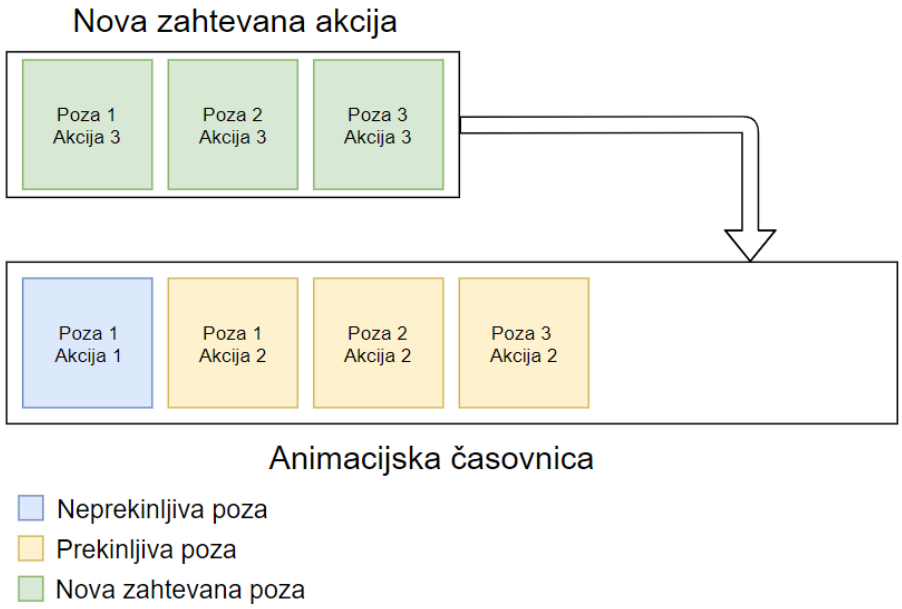
Drugi del nalaganja animacijskih datotek izhaja iz konfiguracijskih datotek, omenjenih v odseku 3.3. Najprej se naložijo vsi vzdevki kosti, ki jih bomo uporabili za delo s podatki individualnih sklepov. Za tem iz datoteke, ki vsebuje konfiguracije animacijskih akcij, preberemo vse poti do posame-

znih datotek z različnimi pozami in iz teh nato ustvarimo dejanske akcije.

3.5 Časovnica animacijskih poz

Za pravilno prehajanje med animacijskimi pozami potrebujemo robusten sistem za predvajanje želenih akcij. Tega problema se lotimo z uvedbo časovnice (*angl. timeline*), ki bo delovala kot vrsta zaporednih animacijskih poz. Ob želji po predvajanju nove animacije bomo vse poze izbrane akcije dodajali na časovnico za pozami zadnje animacijske akcije, ki se je ne da prekiniti (slika 3.3). Če lahko prekinemo vsako od akcij, ki so trenutno na časovnici, bomo časovnico popolnoma izpraznili in jo napolnili s pozami želene nove animacijske akcije ter prehajanje nadaljevali iz trenutno shranjene poze. V nasprotnem primeru pa bomo pobrisali vse prekinljive poze in poze nove animacijske akcije dodali za zadnjo neprekinljivo pozo, kar lahko vidimo iz slik 3.3 in 3.4.

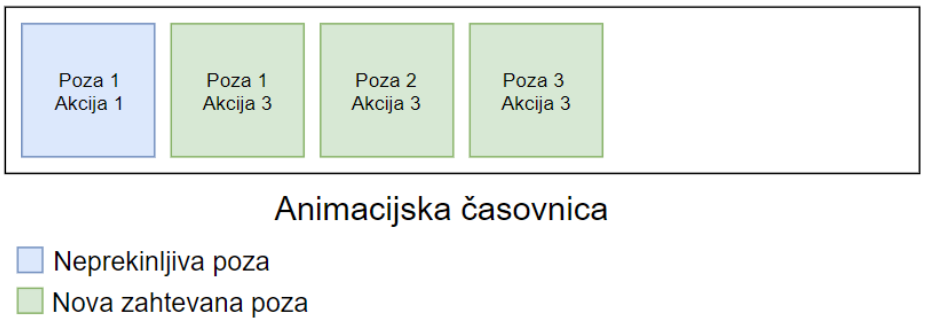
Za vsako od animacijskih poz imamo določeno trajanje prehoda v naslednjo pozo. S pomočjo tega lahko med predvajanjem glede na čas izvajanja trenutnega prehoda med pozama določimo, če smo v naslednjo pozo že prešli. V primeru, da je potekel čas izvajanja trenutnega prehoda, lahko prvo pozo časovnice odstanimo in jo nadomestimo s pozo, v katero smo prešli. Za novo ciljno pozo pa vzamemo naslednjo pozo na časovnici. Primer odstranjevanja zastarele animacijske poze in izračun nove vrednosti trajanja prehoda med pozama lahko vidimo na sliki 3.5. V primeru, da je časovnica po odstranitvi prve poze prazna, bomo prešli v privzeto stoječo pozo. Proces odstranjevanja poz pa ponavljamo, dokler je vrednost časovne spremenljivke trajanja trenutnega prehoda večja od trajanja prehoda prve animacijske poze na časovnici, kar je tudi prikazano na sliki 3.5.



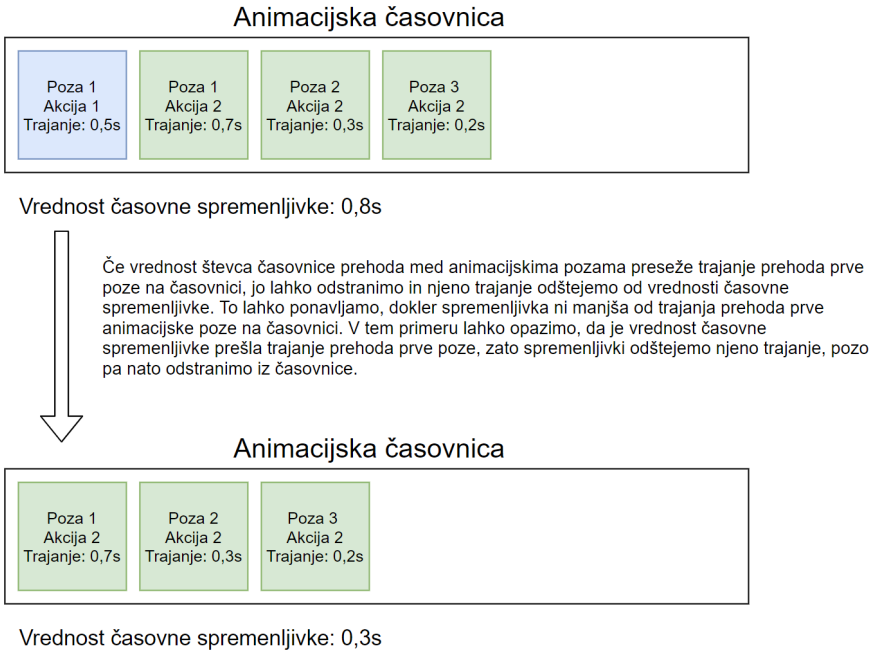
Slika 3.3: Slika prikazuje animacijsko časovnico, v kateri je prikazano zaporedje animacijskih poz, v katere bomo prehajali. Iz legende levo spodaj je razvidno, da je modro obarvana poza del neprekinljive animacijske akcije in vanjo bomo morali nujno preiti, preden lahko nadaljujemo z izvajanjem ostalih animacijskih akcij. Z rumeno so pobarvane animacijske poze, ki so del prekinljivih akcij. Te pa lahko odstranimo iz časovnice in jih nadomestimo z novimi, zeleno obarvanimi animacijskimi pozami.

3.6 Računanje transformacij posameznih sklepov

Ko imamo podatke o tem, med katerima pozama trenutno prehajamo, lahko začnemo z računanjem vmesne poze. Te v hierarhični obliki shranjujejo lokalne transformacije starševskega vozlišča, ki pri končnih transformacijah vplivajo na vse svoje otroke. Z računanjem začnemo pri korenskem vozlišču, nato pa se rekurzivno spuščamo po globini, sproti pa računamo transformacije staršev. Ker korensko vozlišče starša nima, je njegova začetna transfor-



Slika 3.4: Slika prikazuje dodajanje animacijskih poz novo zahtevane akcije za pozami neprekinljivih akcij.



Slika 3.5: Slika prikazuje proces odstranjevanja zastarele animacijske poze iz časovnice in izračun nove vrednosti časovnega števca prehoda med naslednjima pozama.

macija predstavljena z identitetno matriko, ki ne vsebuje nobene transformacije. Ob računanju lokalnih transformacij imamo za vsak sklep obeh poz, med katerima prehajamo, podatka o premiku in vrtenju, predstavljenim s kvaternionom. Pri vrtenju uporabljamo kvaternione, ker omogočajo učinkovitejše shranjevanje podatkov in so bolj preprosti za uporabo kot matrike. Hkrati zagotavljajo tudi bolj pravilen končni rezultat. Vmesne vrednosti med posameznimi transformacijami poz lahko dobimo s pomočjo interpolacije [9]. Preden lahko začnemo z računanjem vmesnih transformacij med pozama, moramo dobiti še faktor, ki določi, kakšen vpliv ima vsaka od poz na končno transformacijo. Glede na željen izgled prehoda med pozama za računanje uporabimo faktorje naslednjih vrst interpolacije:

- **Linearna**

  Pri vsaki od interpolacij najprej začnemo z linearno. Preden se sploh lotimo računanja, moramo trajanje trenutne animacijske poze normalizirati v vrednost intervala $[0, 1)$, kar lahko storimo s pomočjo funkcije:

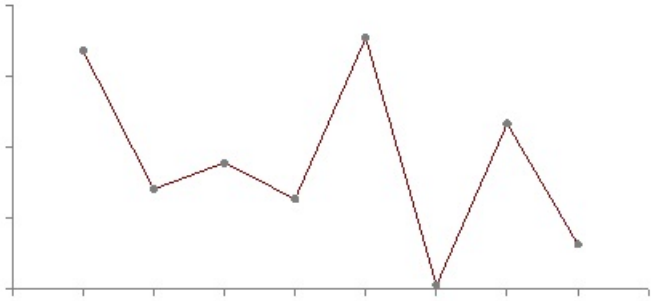
  $$f(t, t_{start}, t_{end}) = \frac{t - t_{start}}{t_{end} - t_{start}}$$

  Za rezultat dobimo interpolacijski faktor, ki ga v našem primeru uporabljamo samo kot osnovo za naslednje interpolacije, saj uporaba faktorja linearne interpolacije pri prehajanju med animacijskimi pozami v praksi očem ni prijazna. Premiki iz dobljenih rezultatov namreč izgledajo predvsem nenaravni, kar lahko vidimo na sliki 3.6.

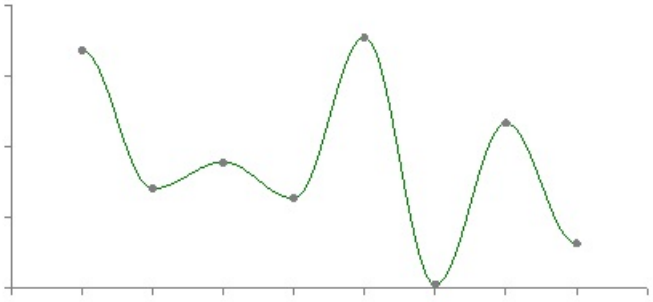
- **Kosinusna**

  Kosinusna interpolacija v primerjavi z linearno drastično zgladi prehode med koncem ene od animacijskih poz do začetka naslednje. To lahko takoj opazimo, če primerjamo grafa na slikah 3.6 in 3.7. Faktor za izračun take interpolacije lahko dosežemo po normalizaciji časa v zgoraj opisanem intervalu $[0, 1)$ z enačbo:

  $$f(t) = \frac{\cos(t\pi) - 1}{-2}$$



Slika 3.6: Graf linearno interpoliranih zaporednih točk.

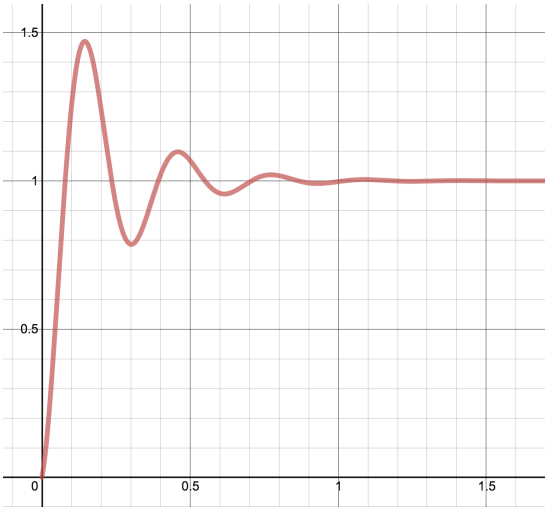


Slika 3.7: Graf kosinusno interpoliranih zaporednih točk.

- **Vzmetna**

  Ko na animacijski časovnici začnemo prehajati na zadnjo pozo predvajanih akcij (ko ni več zahtev po novih animacijah in igralec preide v stanje mirovanja), lahko v primeru poz, kjer pride do nenadnega prenehanja premika, za bolj živ videz mišičnega odziva skeletnega telesa uporabimo faktor vzmetne interpolacije [18]. Ta bo po takojšnjem prenehanju gibanja udom telesa dodal nihanje, ki nastane zaradi razporeditve kinetične energije po mišicah ob spremembi smeri sil. Ker gre za posebno krivuljo, lahko njeno funkcijo zapišemo na več različnih načinov. V našem pogonu smo uporabili krivuljo, vidno na sliki 3.8, ki jo lahko dobimo z uporabo enačbe, kjer spremenljivka a prestavlja amplitudo, w pa časovno frekvenco:

$$f(t) = -[e^{-\frac{t}{a}} \cos(tw)] + 1$$



Slika 3.8: Graf vzmetne interpolacije.

Ko imamo delež mešanja posameznih transformacij vsake od animacijskih poz, lahko glede na izračunan interpolacijski faktor dobimo vmesne transformacije za vsakega od sklepov. Vmesno lokalno lokacijo (lokacija v skeletni hierarhiji, ki še ni bila transformirana s starševsko matriko) posameznih sklepov med obema pozama lahko dobimo s preprosto linearno interpolacijo glede na dobljeni faktor. Če pa na isti način poskusimo tudi pri vrtenju, pride pa pri preveliki razliki med točkama do nenaravnega prehoda, zato se tukaj uporabi metoda sferične linearne interpolacije (Slerp), ki namesto ravne premice za prehod med vrednostma uporabi krivuljo krožnice. Pri računanju teh si pomagamo z matematično knjižnico GLM [7]. Dobljene rezultate lahko za vsak posamezen sklep shranimo v začasno animacijsko pozo, iz katere bomo prehajali pozneje, če bo v trenutku prišlo do prekinitve predvajanja animacijske akcije.

Ko smo izračunali obe od vmesnih transformacij, ju lahko pretvorimo v 4-dimenzionalni matriki in ti v zaporedju vrtenje in nato premik zmnožimo. Rezultat lahko potem pomnožimo še s starševsko matriko in tega uporabimo kot končno preobrazbo trenutnega sklepa, rezultat pa z rekurzivnim klicem pošljemo po hierarhiji navzdol kot starševsko transformacijo otrok vozlišča.

3.7 Končna vizualizacija

Z uporabljenimi izračunanimi transformacijami lahko sedaj skupaj z inverzom odmika od koordinatnega izhodišča posameznih sklepov in globalnega inverza korena scenskega grafa skeletne animacije dobimo končne transformacije. S temi bomo vsako od oglišč najprej preslikali v lokalni koordinatni sistem kosti, iz katerega bomo potem iz izračunanih preobrazb premaknili oglišče nazaj v lokalni prostor poligonske mreže oglišč. Od tu pa lahko popravimo še globalno koordinatno izhodišče s pomočjo inverza transformacijske matrike korena scenskega grafa.

```
glm::mat4 _offset = _playOriginalFile ? _boneTransform.m_originalOffset :
_boneTransform.m_bvhOffset;
m_offsetTransforms[i] = m_skeleton->m_globalInverseTransform * m_transforms[i] *
    _offset;
```

Končne izračunane transformacije lahko zatem pošljemo in uporabimo v ogliščnem senčilniku (*angl. vertex shader*) za izračun končnih lokacij posameznih oglišč glede na uteži vpliva kosti (*angl. skinning*).

```
mat4 boneTransform = u_bones[i_boneIds.x] * i_weights.x;
boneTransform += u_bones[i_boneIds.y] * i_weights.y;
boneTransform += u_bones[i_boneIds.z] * i_weights.z;
boneTransform += u_bones[i_boneIds.w] * i_weights.w;

vec4 skinnedVertex = boneTransform * vec4(i_vertex, 1.0);
OUT.vertex = vec3(u_modelViewMatrix * skinnedVertex);
```

Dobljena oglišča lahko obravnavamo kot končne lokacije namesto vhodnih oglišč poligonske mreže. Pazimo tudi, ker se zaradi novih transformacij oglišč lahko spremenijo tudi kateri drugi vhodni podatki, kot sta na primer normala in tangenta, ki se uporabljata za osvetljavo površine. Ker sta oba od podatkov lokalna glede na oglišče, nad njima uporabimo samo rotacijski del transformacijske matrike.

```
mat3 normalMatrix = transpose(inverse(mat3(u_modelViewMatrix * boneTransform)));
OUT.normal = normalMatrix * i_normal.xyz;
OUT.tangent = normalMatrix * i_tangent.xyz;
```

```
4 OUT.bitangent = normalize(cross(OUT.tangent, OUT.normal));
```

Poleg uporabe v senčilnikih končnega videza entitete s skeletno animacijo lahko enake kostne transformacije uporabimo tudi v senčilniku za računanje senc.

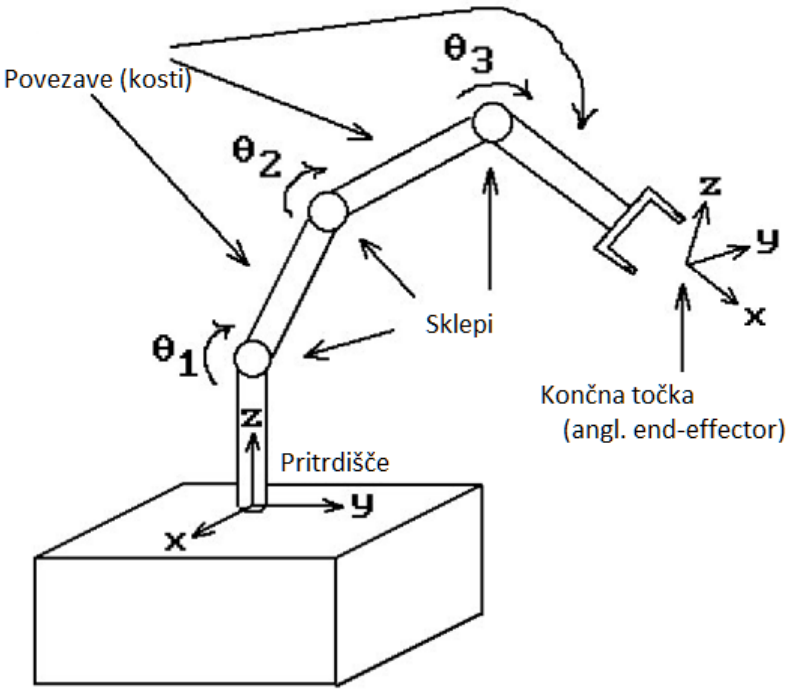
Poglavje 4

Inverzna kinematika

Ko govorimo o inverzni kinematiki na splošno, imamo v mislih pridobivanje lokacije premika nekega predmeta s pomočjo drugih scenskih podatkov. Izredno pogosto uporabo lahko zaznamo v svetu robotike, kjer se za doseg končne točke (*angl. end-effector*) v svetu izračunajo rotacije in premiki posameznih sklepov robotske roke. Na sliki 4.1 lahko vidimo primer modela preproste roke. V industrijski uporabi teh pa brez inverzne kinematike praktično ne bi mogli nič. Zelo pogosta je tudi uporaba v svetu animacij za reševanje problemov pravilnega premikanja skeletnega telesa po razgibanih površinah, ki deluje na podoben način kot omenjena robotska roka.

4.1 Algoritem FABRIK

Odločili smo se za algoritem FABRIK [11] (*angl. Forward And Backward Reaching Inverse Kinematics*), ker je eden izmed najbolj preprostih in najhitrejših obstoječih algoritmov inverzne kinematike. Zaradi narave računanja ga štejemo pod hevristične metode, saj za rezultat ne vrne nujno popolnoma točnih podatkov, ampak to stori v zelo kratkem času, kar je idealno za razmere risanja v realnem času. Za razliko od bolj natančnih algoritmov, kjer se do rotacij sklepov pride z uporabo kompleksne matematike, FABRIK išče samo točke na daljici.



Slika 4.1: Slika prikazuje preprosto robotsko roko s prijemalko na koncu. S pomočjo inverzne kinematike lahko roki izračunamo in nastavimo rotacije vseh sklepov tako, da se s prijemalko, ki je pritrjena na konec verige sklepov, čim bolje približamo želeni končni točki.

Pred začetkom uporabe algoritma moramo najprej najti točko v svetu, kamor bomo poslali enega izmed sklepov našega telesa. Običajno uporabimo za doseganje končne točke konce udov telesa, kot so roke in noge. Ko točko določimo, jo moramo pretvoriti iz koordinatnega sistema sveta v lokalnega obravnavane mreže, kar lahko storimo z uporabo inverza matrike transformacije entitete, ki vsebuje našo skeletno animacijo. Sedaj določimo še sklep, katerega kost bo usmerjena proti končni točki, nato pa lahko s pomočjo rekurzije zgradimo verigo zaporednih sklepov od korena skeleta do izbranega sklepa končnega lista.

Pred začetkom izvajanja lahko najprej preverimo, ali je veriga s svojo dolžino sploh zmožna priti do končne točke. V primeru, da je končna točka

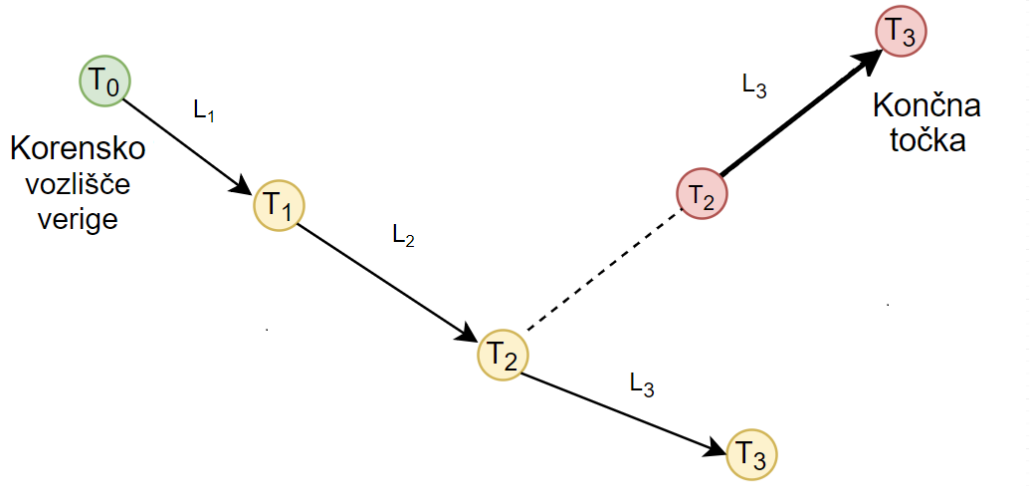
od korena oddaljena za daljšo razdaljo kot meri celotna, iztegnjena veriga, lahko za rezultat to popolnoma iztegnemo v smeri končne točke in predčasno vrnemo izračunan rezultat. V nasprotnem primeru pa se moramo lotiti dejanskega računanja.

Izvajanje algoritma bomo, kakor že njegovo ime pove, razdelili na dva koraka, in sicer seganje naprej in nazaj. Kot prvi del algoritma (seganje naprej) za začetek vzamemo zadnje vozlišče (sklep, ki je najbolj oddaljen od korena) v verigi in ga premaknemo na želeno, končno točko (sklep T3 na sliki 4.2). Iz slike 4.2 lahko vidimo, da zatem točko T2 pomaknemo na premico med vozliščema T2 in T3, oddaljeno od točke T3 za dolžino kosti med vozliščema v smeri lokacije točke T2.

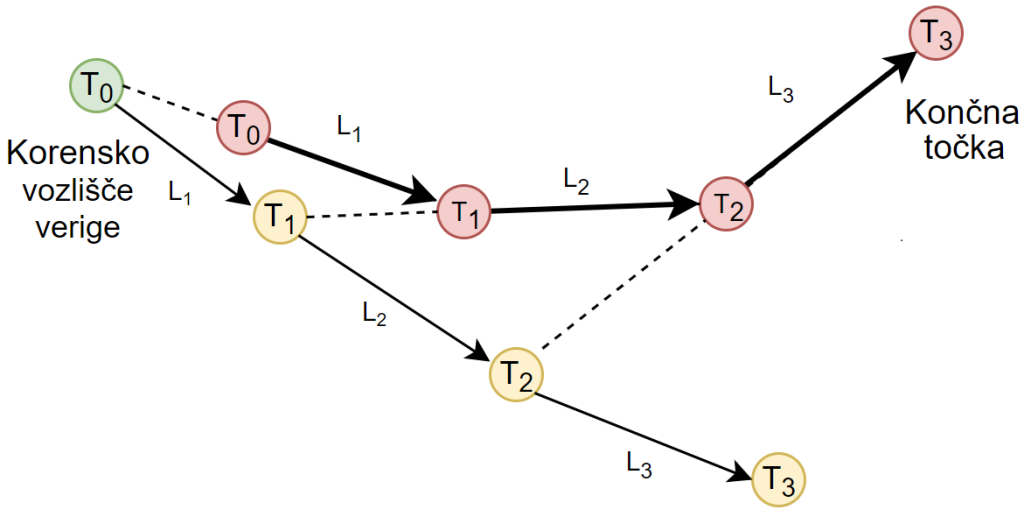
Sedaj se lahko pomaknemo po verigi navzgor (bližje korenu) in postopek ponovimo tako, da si vzamemo za novo končno točko lokacijo prejšnjega obravnavanega sklepa (T2). S to predpostavko lahko sedaj postavimo, podobno kot zgoraj, točko T1 na premico med točkama T1 in T2, oddaljeno od vozlišča T2 za dolžino kosti med njima. To lahko, kakor je vidno na sliki 4.3, ponavljamo do korenskega vozlišča verige. Za rezultat dobimo mnogo bolj približane transformacije končni točki, vendar sedaj korensko vozlišče v verigi ni več na začetnem položaju. To bomo popravili v drugi, nazaj-sežni, fazi algoritma.

V drugi fazi algoritma začnemo tokrat s korenskim sklepom verige, ki ga najprej premaknemo na prvotno začetno lokacijo verige (pred izvajanjem algoritma). Sedaj nadaljujemo podobno kot pri naprej-sežni fazi algoritma, vendar vozlišče pritrjujemo v obratnem vrstnem redu. Namesto od zadnjega vozlišča proti prvemu sedaj iteriramo od prvega vozlišča proti zadnjemu. Na sliki 4.4 lahko vidimo enak proces. Točko T1 pritrdimo na premico med vozliščema T0 in T1, oddaljeno za dolžino kosti med njima v smeri vozlišča T1. To lahko ponavljamo enako kakor pri naprej-sežni fazi, dokler ne dosežemo konca verige, kar prikazuje slika 4.5. Za rezultat dobimo mnogo bolj približano končno kost verige želeni točki kakor pred izvajanjem algoritma.

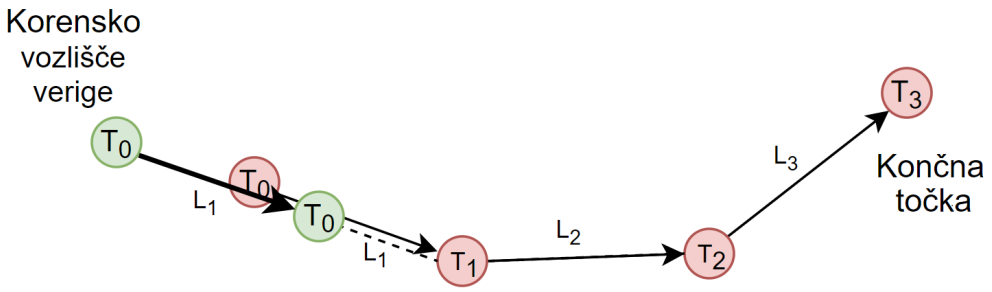
V teh fazah algoritma v nekaterih implementacijah, kot je na primer



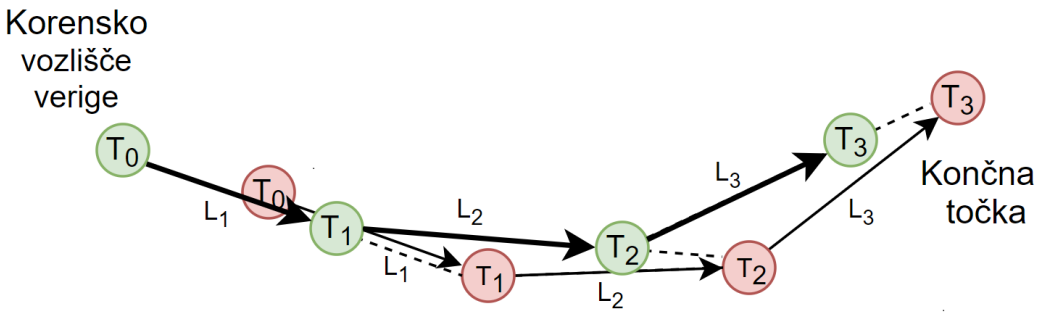
Slika 4.2: Slika prikazuje eno iteracijo naprej-sežnega dela FABRIK algoritma.



Slika 4.3: Slika prikazuje rezultat transformacije sklepov po izvedbi naprej-sežnega dela FABRIK algoritma na celotni verigi.



Slika 4.4: Slika prikazuje eno iteracijo nazaj-sežnega dela FABRIK algoritma.



Slika 4.5: Slika prikazuje rezultat transformacije sklepov po izvedbi nazaj-sežnega dela FABRIK algoritma na celotni verigi.

knjižnica inverzne kinematike Caliko [4], lokacije sklepov posplošimo na začeten in končen položaj kosti. S tem omogočimo presledke med sklepi in tako večjo prilagodljivost implementacije. Za potrebe našega telesa, kjer so sklepi v vseh primerih združeni skupaj, lahko tako posplošitev izpustimo.

V zelo redkih primerih nam uspe zadovoljive rezultate pridobiti že po eni iteraciji seganja naprej in nazaj. V nasprotnih primerih pa zaradi želje po

natančnosti rezultate algoritma zaporedno rešujemo večkrat, kjer za začetne lokacije uporabimo rezultate prejšnje iteracije. Tu se po navadi omejimo na maksimalno število ponovitev ali pa predčasno prenehamo z izvajanjem, ko dosežemo zadovoljivo majhno oddaljenost konca zadnje kosti verige od želene končne točke. Več o originalni implementaciji algoritma si lahko preberete tudi v viru [21].

4.2 Drugi pristopi

Poleg algoritma FABRIK poznamo tudi mnogo drugih pristopov, ki pa so po veliki večini bistveno počasnejši, a obenem ponujajo večjo natančnost. V našem primeru takih zahtev ne potrebujemo, zato je opisani algoritem za nas najbolj primeren. Pod ostale, bolj popularne pristope, pa štejemo:

- **Jacobijeve inverze**
  Njegova uporaba se je začela na področju robotike kot eden izmed prvotnih pristopov k reševanju problema inverzne kinematike. Implementacija je relativno preprosta in zelo učinkovita, vendar pa je zaradi velikega števila iteracij za doseg zadovoljivega rezultata pogosto počasna. Več o implementaciji takega algoritma inverzne kinematike si lahko preberete v virih [12, 10, 11].

- **Hevristične metode**
  Poleg metod, kjer za pridobitev rezultata uporabljamo Jacobijeve inverze, med bolj poznane štejemo tudi hevristične metode. Te s pomočjo preprostih, a zelo hitrih iterativnih sprememb, pridejo do zadovoljivega približka želeni končni točki. Poleg uporabljenega algoritma FABRIK [21] med zelo popularne štejemo tudi CCD [22].

V primerjavi z drugimi metodami, kar je razvidno iz slike 4.6, sodi algoritem FABRIK med ene izmed najbolj hitrih.

	Reachable Target				Unreachable Target	
	Number of Iterations	Matlab exe. time (*sec*)	Time per iteration (in *msec*)	Iterations per second	Number of Iterations	Matlab exe. time (*sec*)
FABRIK	15.461	0.01328	0.86	1164	67.564	0.06207
CCD	26.308	0.12356	4.69	213	390.135	3.92869
Jacobian Transpose	1311.190	12.98947	9.90	101	6549.000	33.90473
Jacobian DLS	998.648	10.48051	10.49	95	2881.667	14.87918
Jacobian SVD-DLS	808.797	9.29652	11.50	87	2808.452	15.97591
FTL	21.125	0.02045	0.97	1033	22.325	0.02526
Triangulation	1.000	0.05747	57.47	21	1.000	0.06993

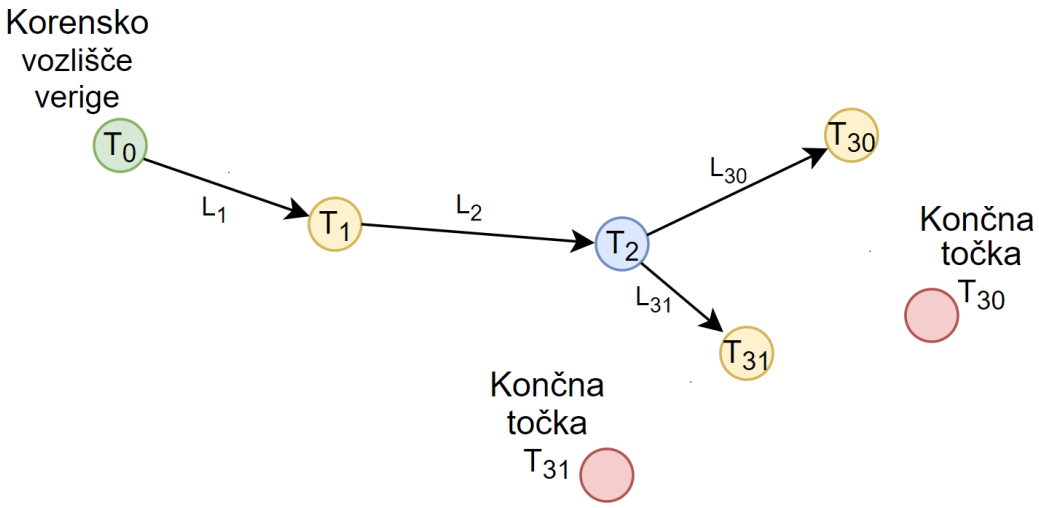
Slika 4.6: Primerjava hitrosti algoritmov inverzne kinematike v programskem jeziku MATLAB. Povzeto iz [21].

4.3 Več končnih omejevalnih točk

Ko imamo pri računanju inverzne kinematike opravka s samo eno končno točko, se lahko zadovoljimo že s preprosto rešitvijo algoritma. Ko pa imamo teh več, kar se v uporabi pri računalniški grafiki pogosto zgodi (npr. pri računanju transformacij za doseg končnih točk posameznih nog in rok sočasno), nočemo, da se ob računanju naslednjih transformacij sklepov pokvarijo prejšnji rezultati. Za to si moramo omisliti nov pristop in ustrezno nadgraditi osnovni algoritem.

Problema se lotimo tako, da s pomočjo seznama parov sklepov in pripadajočih želenih končnih točk rekurzivno obhodimo hierarhijo skeletnega telesa. Ob tem iščemo starševska vozlišča, ki vsebujejo verigo otrok, katerih konce kosti želimo premakniti na lokacije končne točke. Hierarhijo preiskujemo najprej v globino, nato pa se premikamo po nivojih navzgor. Ko najdemo prvo vozlišče, ki vsebuje v svojem poddrevesu otroke, ki bi jih radi pritrdili na končne točke, lahko začnemo z izvajanjem algoritma. Primer stanja ob uspešno najdenem skupnem vozlišču lahko vidimo na sliki 4.7.

Za pravilno izvajanje algoritma ni treba poleg iskanja skupnih vozlišč dodajati nobenih novih funkcionalnosti. Začnemo z najglobljim skupnim vozliščem v hierarhiji, ki ima 2 ali več poddreves in vsebujejo otroke, ki jim je nastavljena želena končna točka. Seznam parov sklepov in njihovih končnih točk mora nastaviti uporabnik grafičnega pogona ob klicu funkcije



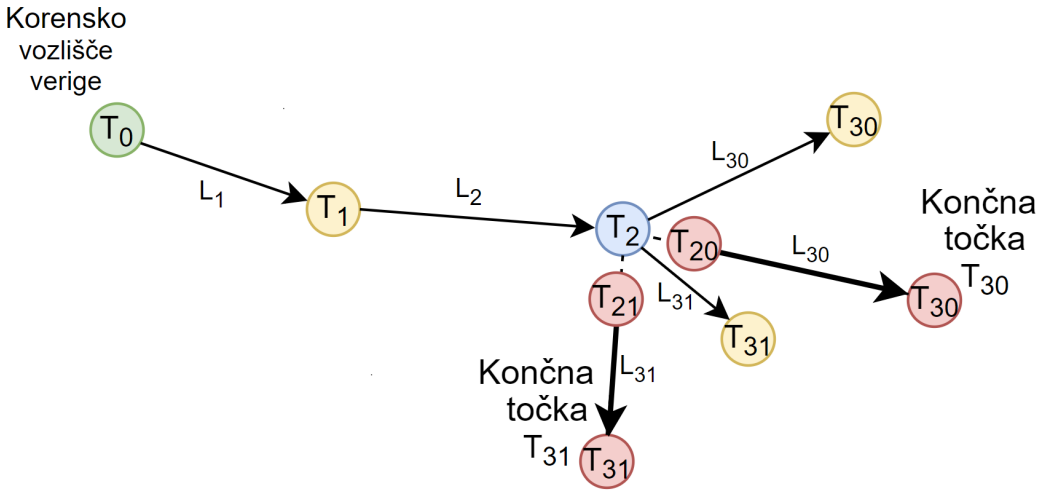
Slika 4.7: Slika prikazuje začetno stanje hierarhije ob uspešno najdenem skupnem staršu poddreves otrok (obarvano z modro), ki želita doseči končni točki.

za izračun transformacij inverzne kinematike. Na primer, če imamo končni točki nastavljeni za desno in levo roko, bomo, kakor je razvidno iz slike 3.2, kot tako skupno vozlišče našli prsno kost.

Zdaj lahko iz skupnega vozlišča sestavimo nove verige. Kot najbolj pomembno bomo za začetek ustvarili verigo, ki sega od prvega starša, ki ima v svojih poddrevesih več otrok, katerim želimo nastaviti končne točke (ob iskanju ne upoštevamo trenutne skupne točke). Če takega starša v hierarhiji ni, potem namesto tega uporabimo koren celotne hierarhije. Nato sestavimo še verige, ki od skupnega vozlišča segajo do vsakega od otrok, katerim želimo nastaviti lokacije vozlišč kot lokacije končnih točk (te verige vsebujejo lahko tudi več vozlišč).

Po ustvarjenih verigah lahko sedaj nad vsemi verigami poddreves otrok izvedemo naprej-sežni del algoritma FABRIK tako, da pritrdimo vsako od verig na ustrezno končno točko, kar lahko vidimo na sliki 4.8. Koreni teh verig sedaj niso pritrjeni na skupnega starša. Iz novih lokacij vsakega od

korenov verig nato izračunamo njihovo povprečno lokacijo (centroid). Primer centroida korenov vozlišč lahko vidimo na sliki 4.9.

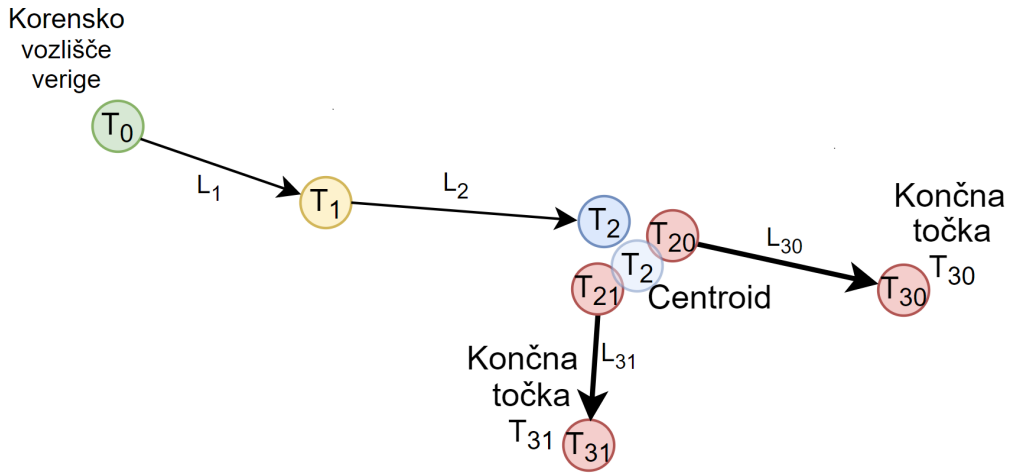


Slika 4.8: Slika prikazuje uporabo naprej-sežnega dela FABRIK algoritma za pritrditev verig končnih točk na skupnega starša.

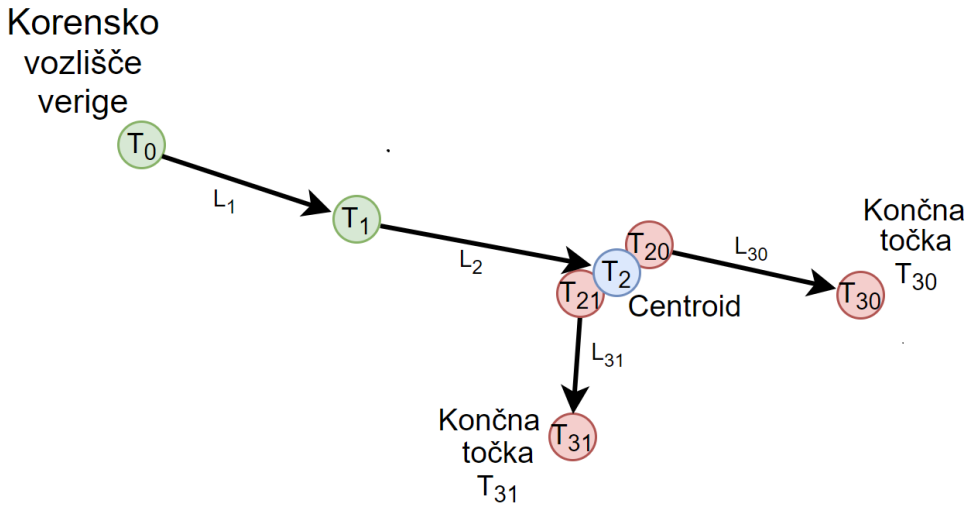
Dobljeno centroidno točko lahko sedaj uporabimo kot končno točko reševanja obeh delov FABRIK algoritma starševske verige od korena celotne hierarhije (oz. najdenega vozlišča skupnega starša otrok končnih točk) do skupnega vozlišča (slika 4.10).

Centroid lahko sedaj uporabimo tudi kot lokacijo korenskega vozlišča izvajanja nazaj-sežnega dela algoritma FABRIK pri vseh verigah otrok s končnimi točkami. Rezultat računanja algoritma pa lahko vidimo na sliki 4.11. Ko smo uspešno obdelali eno izmed skupnih vozlišč, lahko nadaljujemo preiskovanje za nova skupna vozlišča staršev (upoštevamo tudi skupno vozlišče, ki smo ga pravkar obdelali). Če najdemo novo skupno vozlišče, ga obdelamo na isti način, kot smo opisali zgoraj. V nasprotnem primeru pa bomo prišli do korena celotne hierarhije in tako iteracijo algoritma zaključili.

Iz slike 4.11 je razvidno, da smo se že po eni iteraciji algoritma zelo približali željeni rešitvi. Podobno kot tudi pri implementaciji z eno končno

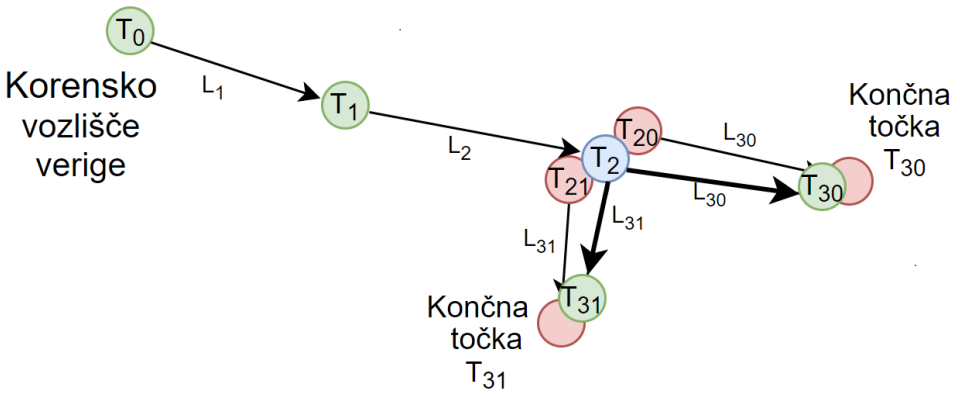


Slika 4.9: Slika prikazuje novo lokacijo skupnega centroida (povprečje lokacij korenov verig, ki segajo do otrok končnih točk po naprej-sežni iteraciji FABRIK algoritma) med izvajanjem FABRIK algoritma za vozlišča z več končnimi točkami.



Slika 4.10: Slika prikazuje izvajanja obeh delov algoritma FABRIK za verigo od korenskega do skupnega vozlišča.

točko celoten algoritem, da bi dobili boljše rezultate, raje izvedemo večkrat.



Slika 4.11: Slika prikazuje zadnji del algoritma FABRIK za povezovanje verig več končnih točk na novo izračunano skupno lokacijo vozlišča.

Ko smo prišli z uporabo algoritma inverzne kinematike FABRIK do končnih lokacij posameznih sklepov v skeletni hierarhiji, lahko sedaj dobljene rezultate uporabimo še za določanje pravilnih rotacijskih transformacij. Če bi računanje novih rotacij izpustili, bi ob velikih spremembah lokacij vozlišč dobili popolnoma napačne končne transformacije (slika 4.12).

Računanje transformacij opravimo rekurzivno od korena do listov po verigah, ki smo jih spremenili med računanjem transformacij inverzne kinematike. Za posamezno vozlišče najprej pridobimo vektor, ki je usmerjen iz trenutnega vozlišča proti svojemu otroku. Sedaj lahko enak vektor izračunamo še med soležnima vozliščema v pozi postavljanja, ki ne vsebuje nobenih rotacijskih transformacij, ampak samo premike in predstavlja začetno stanje skeleta. Oba vektorja normaliziramo in s pomočjo matematične knjižnice GLM izračunamo kvaternion rotacije med njima, kar lahko vidimo na sliki 4.13.

Računanja vmesnega kvaterniona se lotimo najprej z iskanjem pravokotnice na ploskev, ki jo normalizirana vektorja omejujeta. Tak vektor lahko

Slika 4.12: Slika prikazuje izris animiranega modela brez popravkov vrtenja posameznih sklepov.

najdemo s pomočjo vektorskega produkta. Pri računanju tega pazimo na vrstni red množenja vektorjev, saj njuna zamenjava obrne smer vrtenja. Dobljeni vektor predstavlja rotacijsko os.

$$\vec{a} = \vec{v_1} \times \vec{v_2} \tag{4.1}$$

S pomočjo skalarnega produkta dobimo tudi kot med vektorjema. Ob računanju tega pazimo, da sta vektorja normalizirana.

$$\begin{aligned} \cos(\theta) &= \frac{\vec{v_1} \cdot \vec{v_2}}{|\vec{v_1}||\vec{v_2}|} \\ \theta &= \cos^{-1}\left(\frac{\vec{v_1} \cdot \vec{v_2}}{|\vec{v_1}||\vec{v_2}|}\right) \end{aligned} \tag{4.2}$$

Sedaj lahko iz osi vrtenja in kota med vektorjema s pomočjo knjižnice GLM ustvarimo nov kvaternion, ki se po komponentah izračuna na sledeč način:

$$
\begin{aligned}
Q_x &= \vec{a_x} \sin(\frac{\theta}{2}) \\
Q_y &= \vec{a_y} \sin(\frac{\theta}{2}) \\
Q_z &= \vec{a_z} \sin(\frac{\theta}{2}) \\
Q_w &= \cos(\frac{\theta}{2})
\end{aligned}
\tag{4.3}
$$

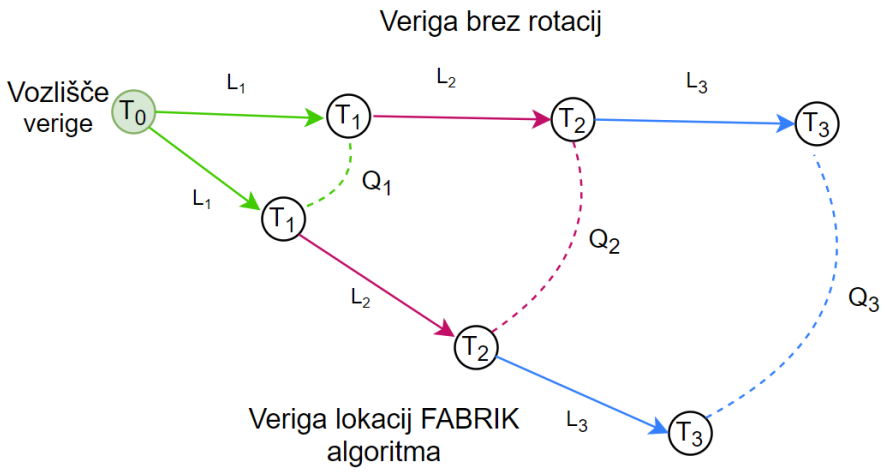
Postopek lahko ponavljamo vse do konca verig, kjer pa rotacij listov na ta način ne bomo mogli dobiti. Take rotacije lahko na koncu ročno nastavimo glede na želene rezultate. Na primer pri dotikanju bližnjih ovir v poglavju 5.5 smo predpostavili, da bomo roko ob dotiku ovire položili pravokotno na njeno površino.

4.4 Uporaba rezultatov za generiranje animacijske poze

Dobljene kvaternione posameznih vozlišč lahko sedaj združimo z rezultati algoritma FABRIK v skupne matrike, ki že predstavljajo končne transformacije sklepov in ne več lokalnih preobrazb. Za uporabo končnih transformacij v animacijski časovnici pa jih moramo spremeniti še v animacijsko pozo. To storimo s pretvorbo posameznih končnih transformacij nazaj v lokalni prostor sklepa.

Generiranja poze se lotimo podobno kakor v odseku 3.6 tako, da se po skeletni hierarhiji spustimo v globino in sproti računamo transformacije. Za razliko od metode v odseku 3.6 pa namesto računanja starševskih transformacij računamo njihove inverze, ki jih potem lahko pošljemo vsem otrokom vozlišča. V slednjem nato iz končne transformacije rezultat inverzne kinematike pomnožimo z inverzom starševske matrike. Za rezultat dobimo lokalno preobrazbo sklepa za obravnavano vozlišče.

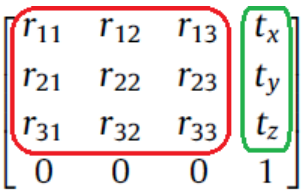
Ker v naših transformacijskih matrikah ne uporabljamo raztega, lahko na zelo učinkovit in preprost način iz dobljene lokalne transformacijske matrike



Slika 4.13: Slika prikazuje kvaternione vrtenja med vektorji sklepov usmerjenih proti svojim otrokom iz vozlišč poze postavljanja (poz brez rotacij) proti vozliščem rezultata FABRIK algoritma. Dobljeni kvaternioni predstavljajo že rotacije končnih transformacijskih matrik posameznih kosti (zaporedne rotacije od T0 do obravnavanega vozlišča) in ne lokalnih rotacij za vsak sklep. Dobljene rezultate moramo nato za vsak sklep z uporabo inverza starševske transformacije preslikati v lokalne rotacije.

sklepa izluščimo vrtenje in premik. Za matriko vrtenja lahko uporabimo kar sredinsko matriko, tako da iz 4-dimenzionalne matrike odstranimo zadnjo vrstico in stolpec. Za premik pa lahko uporabimo prve tri komponente zadnjega stolpca matrike, kar je razvidno iz slike 4.14.

Dobljeni lokalni premik lahko zdaj preprosto shranimo v sklep nove animacijske poze. Matriko vrtenja pa za lažje shranjevanje in poznejšo uporabo s pomočjo matematične knjižnice GLM pretvorimo v kvaternion. Tako smo uspešno z uporabo inverzne kinematike zgenerirali in shranili novo animacijsko pozo. Celoten potek generiranja poze iz končnih transformacij vozlišč si lahko ogledate v sledečem algoritmu:



Slika 4.14: Slika prikazuje transformacijsko matriko posameznega vozlišča skeletne hierarhije. Z rdečo obarvano lahko vidimo matriko vrtenja, z zeleno pa vektor premika.

`GeneratePose`(*node*, *finalTransforms*, *pose*, *inverseParentTransform*)

```
1  boneId = node.boneId
2  localTransform = inverseParentTransform * finalTransforms[boneId]
3
4  pose[boneId].location = Vector3(localTransform[3])
5  pose[boneId].rotation = Quaternion(Matrix3x3(localTransform))
6
7  Matrix4x4 inverseTransform = MatrixInverse(localTransform) *
     inverseParentTransform
8
9  for child in node.children do
10     GeneratePose(child, finalTransforms, pose, inverseTransform)
```

Poglavje 5

Proceduralna generacija animacijskih poz

Kot proceduralno generacijo štejemo vse, kar smo zgenerirali z nekim algoritmom in ne ustvarili na roke. Prednosti take metode so manjša poraba prostora na disku, velika možnost za naključen in s tem unikaten videz ter izredno veliko število različnih zadovoljivih rezultatov. Slabosti take metode pa se opazijo že iz samih prednosti, saj v nekaterih primerih namesto naključnosti ali deterministočnega generiranja iz predpostavljenega stanja želimo popolnoma prilagodljive rezultate, ki bi jih želeli ročno sami spreminjati. Eno izmed večjih težav predstavlja tudi investicija časa in denarja v razvoj algoritmov za proceduralno generiranje. V praksi se proceduralno generacijo najbolj pogosto uporablja pri ustvarjanju tekstur in 3-dimenzionalnih modelov, med temi pa je zelo pogosto uporabljena za generiranje popolnoma naključnega terena in s tem v določenih primerih tudi unikatne igralne izkušnje.

V tem poglavju bomo govorili predvsem o generiranju transformacijskih poz iz kolizijskih teles in uporabi fizikalne knjižnice v kombinaciji z inverzno kinematiko. Transformacije kolizijskih teles lahko s pretvorbo koordinatnih sistemov uporabimo za spreminjanje aktivnih animacijskih poz v našem sistemu za predvajanje skeletne animacije. Z uporabo proceduralne generacije

pa bomo zagotovili pravilen odziv našega skeletnega telesa na kolizijsko okolico.

5.1 Fizikalna knjižnica

Osrednji del proceduralne generacije v našem primeru predstavlja fizikalna knjižnica Bullet Physics [3]. Vanjo ob dodajanju entitet vstavljamo kolizijska telesa, generirana na podlagi omejujočih škatel (*angl. bounding box*) ali posplošenih konveksnih oblik njihovih poligonskih mrež. Tako ustvarimo nadvse natančen kolizijski svet za poznejše računanje končnih točk pri uporabi inverzne kinematike. Poleg kolizij knjižnica skrbi tudi za realistično simulacijo sil, tako da poskrbi za vse, kar potrebujemo za uspešno generiranje dinamičnih animacijskih poz.

5.2 Polproceduralne poze

Poleg popolnoma proceduralnih poz lahko brez pomoči fizikalne knjižnice z uporabo interpolacije ustvarimo vmesne poze med dvema ekstremoma premika (npr. popolnoma sklonjena poza in stoječa poza). Takemu generiranju bomo rekli kar polproceduralno, saj izhajamo iz popolnoma ročno izdelanih poz in jih ne spreminjamo algoritmično. Dober primer v našem animacijskem sistemu je interpolacija med počasno hojo in hitrim tekom ob pospeševanju hitrosti igralca. Sistem glede na trenutno hitrost premika med začetno in najvišjo hitrostjo izračuna interpolacijski faktor, s katerim potem lahko iz omenjenih poz izračuna vmesno pozo za levi in desen premik ob hoji. Zanimiva uporaba se vidi tudi pri sklanjanju telesa za ublažitev sil pri padcu. Tu s pomočjo popolnoma sklonjene in stoječe poze ustvarimo vmesno, interpolirano, pozo glede na hitrost in s tem silo padca. Tako se pri nizkih padcih telo komaj odzove na sile, pri visokih pa se najprej skloni popolnoma do tal, preden lahko nadaljuje s premikanjem.

Eden od bolj uporabnih primerov, ki bi ga lahko implementirali v igrah,

je prilagajanje animacijske poze za nošenje različno velikih predmetov. Tu bi lahko iz poze, kjer nosimo z rokami zelo majhno škatlo, interpolirali proti pozi, kjer igralec nosi zelo velik predmet. Temu bi lahko primešali tudi pozo nagibanja nazaj glede na težo nošenega objekta. Za kaj takega pa bi morali v sistem skeletne animacije dodati še mešanje različnih poz.

5.3 Fizikalno dinamično krilo

Ob nalaganju konfiguracij animacijskega skeleta smo že v odseku 3.3 omenili, da se kosti brez nastavljenih vzdevkov obravnavajo kot dinamične. Za te se navadne preobrazbe iz animacijskih poz ne računajo, ampak se dinamično nastavijo glede na transformacijo njihovega kolizijskega telesa, ki je pritrjeno na telo starševske kosti.

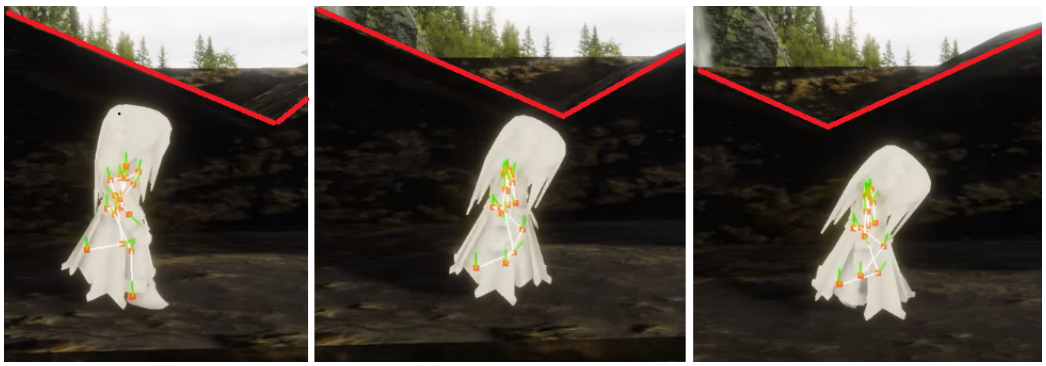
Po naloženih utežeh skeletne animacije grafični pogon iz seznama oglišč poligonske mreže ustvari omejujočo škatlo za območje vpliva vsake od kosti. Iz teh se nato ustvarijo kolizijska telesa našega krila, ki se pritrdijo na njihove starše v skeletni hierarhiji z uporabo vzmetne omejitve. Za te pa skrbi fizikalna knjižnica Bullet Physics, ki samodejno konfigurira vzmetne omejitve iz prednastavljenih parametrov.

Med delovanjem aplikacije se bodo po vsaki spremembi simulacije kolizijskega sveta iz rotacij kolizijskih teles krila nastavile nove transformacije sklepov. Tu pazimo, da poleg rotacijske transormacije ne uporabimo nobene od drugih transformacij, saj se lahko zgodi, da pri simulaciji omejitev vzmeti pride do nepričakovanih zaostankov premika, kar bi ob uporabi lokacije kolizijskih teles pri transformacijah kosti pripeljalo do neželjenega raztega krila, kar pa bi pokvarilo celotno animacijsko pozo. Nekaj fizikalno generiranih transformacij pa lahko vidimo na sliki 5.1.

Slika 5.1: Slike prikazujejo nekaj različnih transformacij dinamičnega krila med premikom telesa. Zgornji dve prikazujeta krilo ob padanju in pristanku na tleh. Spodaj levo lahko vidimo krilo med rotacijo telesa, zadnja slika pa prikazuje odkrivanje kolizije med krilom in bližnjo površino.

5.4 Sklanjanje pred ovirami

Z uspešno implementiranim algoritmom za inverzno kinematiko lahko sedaj začnemo s pravim generiranjem proceduralnih animacijskih poz. Z uporabo fizikalne knjižnice lahko iz lokacije glave ob najnižjem možnem sklonjenem položaju pošljemo žarek naravnost navzgor v kolizisjki svet. Ta bo dolg največ za razliko zgornjega dela glave med najnižjo sklonjeno in najvišjo stoječo animacijsko pozo. Če bo ta v kolizijskem svetu naletel na katerega od teles, nam bo vrnil točko v svetu, kjer je prišlo do preseka med žarkom in ploskvijo. V primeru, da ta ne bo našel kolizije, vemo, da nove animacijske poze ni treba generirati, in lahko uporabimo tisto, ki smo jo ustvarili ročno. Če pa bo prišlo do kolizije z žarkom, pa bomo dobljeno točko lahko uporabili kot končno točko pri računanju inverzne kinematike za glavo telesa. Na sliki 5.2 lahko vidimo zaporedne slike končnega rezultata ob sklanjanju pred prihajajočo oviro.



Slika 5.2: Slika postopoma prikazuje sklanjanje pred prihajajočo oviro med hojo proti njej. Rdeče črte niso del končne slike in prikazujejo zgolj rob ovire za boljšo vidljivost.

Tega problema bi se zaradi spodnje in zgornje omejitve končne višine sklonjenega telesa lahko lotili tudi polproceduralno. Tu bi med animacijskima pozama popolnoma sklonjene in stoječe hoje interpolirali novo, delno sklonjeno pozo s pomočjo odmika stropa od višine stoječe poze.

5.5 Drsanje rok po bližnjih ovirah

Z uporabo fizikalne knjižnice lahko zaznamo s pomočjo kroglastega kolizijskega telesa vse površine, ki so v našem dosegu. Tako najdemo vse bližnje ovire, ki se jih lahko dotaknemo z rokama. Za vsako od rok moramo poiskati tisto, ki je najbolj smiselna in ni izven omejitev dosega telesa, drugače pa tako točko zavržemo. Ker kolizijska krogla ne bo zaznala dveh točk, ki se razlikujeta za večjo širino kot je naš razpon rok, s tem ne bomo imeli težav. Ko si izberemo najbolj primerne točke, jih lahko pretvorimo iz koordinatnega sistema sveta v lokalni koordinatni sistem telesa in jih pošljemo algoritmu za inverzno kinematiko, ki nam bo vrnil končne transformacije animacijske poze. Če pa za katero od rok ne najdemo nobene točke, pa algoritma sploh ne kličemo. Ko nam algoritem inverzne kinematike vrne končen rezultat, lahko temu dodamo tudi rotacije dlani, da so te vzporedne z dotaknjeno površino. Na sliki 5.3 lahko vidimo dva primera dotikanja bližnih površin.

Slika 5.3: Sliki prikazujeta različni animacijski pozi ob dotikanju bližnjih kolizijskih teles. Leva slika prikazuje pozo po zaznanih veljavnih točkah za obe roki, desna pa samo za eno.

Poglavje 6

Vizualizacija

Celotna vizualizacija scene našega animiranega telesa poteka z uporabo programskega vmesnika za računalniško grafiko OpenGL [14]. V ozadju se skriva entitetna arhitektura [6], ki skrbi za urejenost in ločenost delov sistema. Z moderno metodo zakasnjenega senčenja [5] (*angl. deferred shading*) najprej zberemo vse ploskovne podatke scenske geometrije v zaporedje tekstur. Iz teh lahko po računanju osvetlitve dodatno izračunamo še več različnih, post-procesiranih (*angl. post-processing*) grafičnih učinkov. Za konec še vse rezultate smiselno združimo in končno teksturo prikažemo poleg grafičnega vmesnika na zaslonu uporabnika.

6.1 Struktura v ozadju

Za začetek z uporabo knjižnice ASSIMP [2] naložimo iz datotek scenske grafe, iz katerih lahko v našem svetu ustvarimo entitete, v katere vstavimo vse zahtevane komponente glede na njihov tip in uporabo. V kontekstu naše diplomske naloge v grafičnem pogonu uporabljamo naslednje komponente:

- **Matrična komponenta**
  Skrbi za globalno transformacijo posamezne entitete, ki jo prestavi v koordinatni sistem igralca. V to so vključeni premik, vrtenje in razteg.

- **Kolizijska komponenta**
  Predstavlja seznam kolizijskih teles in povezav med njimi za vsako entiteto. Tukaj so shranjena tudi kolizijska telesa dinamičnega krila.

- **Materialna komponenta**
  Shranjuje vse materiale in njim pripadajoče sezname tekstur, ki jih model v entiteti uporablja v senčilnikih (*angl. shaders*) za izračun končnih barv geometrijskih površin na zaslonu.

- **Poligonsko-mrežna komponenta**
  Vsebuje vse podatke o ogliščih, ki so potrebni za končen izris. Ne vsebuje pa uteži in indeksov skeletne animacije.

- **Skeletno-animacijska komponenta**
  Poligonsko-mrežni komponenti doda še za vsako od oglišč podatke o utežeh in indeksih kosti, ki nanj vplivajo. Komponenta vsebuje tudi animacijsko časovnico, preko katere se potem lahko predvajajo prehodi med posameznimi pozami skeletnega telesa.

- **Igralčeva komponenta**
  Uporabljena je za nadzor akcij premikov igralca. S te komponente pridejo praktično vse zahteve po predvajanju novih animacijskih akcij skeletnega sistema.

6.2 Risanje

Med sistemi, ki upravljajo s skupinami komponent, imamo po principu zakasnjenega senčenja skupne ciljne teksture. Vanje ob vsaki osvežitvi slike vnašamo ločene podatke geometrijskih površin narisanih likov za poznejšo uporabo pri računanju osvetljevanja in postprocesiranje. Čez vsakega od sistemov iteriramo trikrat. V prvi od teh najprej počistimo delovne teksture in se pripravimo na risanje. V drugi pa izvedemo dejansko risanje v posamezne teksture, kar lahko vidimo kot primer na sliki 6.1. Ko končamo z navadnim

risanjem, pride še tretja iteracija, kjer narišemo še sence s perspektiv luči za dinamične objekte. Na koncu zadnje iteracije se v zadnjem od sistemov iz dobljenih podatkov izračuna še končna osvetljava in pa faktorji senc. Ko imamo vse rezultate, lahko začnemo s korakom postprocesiranja.

Slika 6.1: Primer zbirke tekstur pri osvetljevanju z zakasnjenim senčenjem. Levo zgoraj lahko vidimo popolnoma svetlo barvo, tekstura desno zgoraj prikazuje normale površin, levo spodaj lahko vidimo globino izrisa, zadnja slika pa prikazuje končni rezultat po izračunu osvetljave in vseh grafičnih učinkov.

6.3 Postprocesiranje

Poleg izračunane osvetljene scene želimo običajno v bolj naprednih grafičnih pogonih na končno sliko dodati tudi nekaj učinkov. Za to skrbi sistem postprocesiranja, ki iz podatkov zakasnjenega senčenja in končnih barv osvetljene scene po korakih izračuna končne barve posameznih učinkov. Ta deluje s pomočjo risanja kvadrata, raztegnjenega čez celoten viden prostor kamere našega pogleda, tako da prekrije zaslon. S pomočjo senčilnika fragmen-

tov (*angl. fragment shader*) lahko nato pridemo do končnega rezultata. V našem grafičnem pogonu se po izračunu osvetljave scene po vrsti izračunajo in združijo še naslednji učinki:

- **Svetlobni sijaj (*angl. bloom*)**
  S tem učinkom povečamo intenzivnost in premer sijaja najbolj svetlih delov slik [8]. Najprej moramo take regije na končni sliki s pomočjo neke robne vrednosti najti. Ko imamo črno-belo sliko, jo lahko z uporabo Gaussovega megljenja spremenimo tako, da bodo svetle regije imele večji obseg. Ta postopek megljenja lahko večkrat ponovimo, da dobimo boljši rezultat. Končni rezultat s takim učinkom je viden na sliki 6.2.

Slika 6.2: Primerjava slike brez dodatnega sijaja (levo) in z njim (desno).

- **Meglitev gibanja (*angl. motion blur*)**
  S tem učinkom [23] lahko simuliramo zameglitev barv, ki jih ob slikanju hitro gibljivih predmetov ustvarijo fotoaparati, ko se predmet premakne med časom osvetlitve, ki ga določa hitrost zaklopa zaslonke

fotoaparata. Pred računanjem zameglitve v senčilnikih moramo shranjevati vse prejšnje transformacije predmetov, ki jih želimo zamegliti ob premiku. Pri skeletni animaciji moramo pa za tak učinek shraniti celotno animacijsko pozo prejšnje sličice. V senčilnikih fragmentov lahko nato zgeneriramo teksturo, ki predstavlja smer megljenja glede na razliko med prejšnjo in trenutno transformacijo na zaslonu. To pa na koncu uporabimo za zameglitev celotne osvetljene slike. Primerjavo videza lahko vidimo na sliki 6.3.

Slika 6.3: Leva slika prikazuje animacijo skoka brez zameglitve gibanja, desna pa z njo, povečano za faktor 10.

- **Volumetrični žarki**

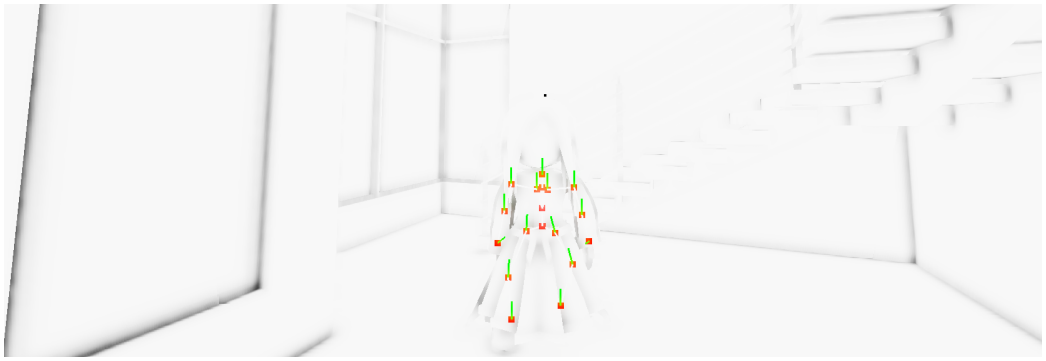
  Kot eden izmed bolj zanimivih učinkov je ta zelo podoben svetlobnemu sijaju [20], vendar namesto da bi vplival na svetleče površine celotne scene, ta vpliva samo na oddajnik svetlobe. V našem primeru je to sonce. Z uporabo radialne zameglitve se na vidna mesta, ki jih določimo s pomočjo globinske teksture, iz središča vira svetlobe izrišejo propagirani žarki. Na sliki 6.4 lahko vidimo končni rezultat za primer

žarke sonca.

Slika 6.4: Leva slika prikazuje končni prikaz samostojnega sonca, na desni pa lahko vidimo tega združenega s celotno sceno, ko ga zastira eden od bližnjih objektov.

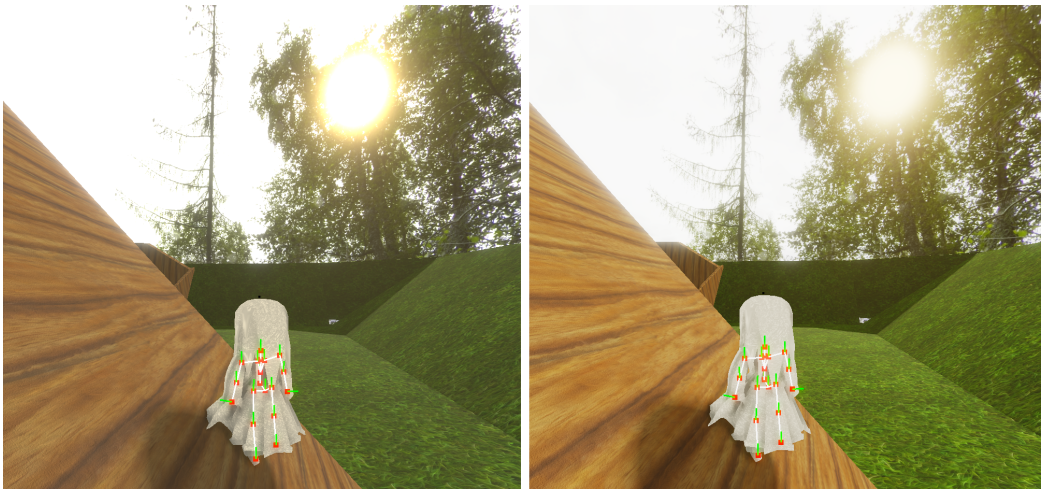
- **Ambientalna okluzija**
  Za simuliranje zatemnjenosti manjših špranj geometrije, kjer se ujame svetloba, se uporablja v kombinaciji s sencami, ki jim v takih primerih tudi zmanjka natančnosti, algoritem za ambientalno okluzijo [24]. Ta s pomočjo podatkov okoliških točk izračuna, kako zatemnjena je trenutna točka teksture, kar lahko vidimo na sliki 6.5. Ker v našem primeru model skeletne animacije ne uporablja posebne teksture, je ta učinek še posebej opazen.

- **Združevanje učinkov**
  Ko končamo z obdelavo vseh učinkov, ki jih želimo prikazati, jih moramo združiti s prvotno, osvetljeno, sliko. Te glede na intenzivnost posameznih učinkov primerno prištejemo ali jih zmnožimo z barvami končne slike.



Slika 6.5: Primer izrisa samo ambientalne okluzije.

- **Pretvorba HDR v LDR**
  Ker se pri računanju osvetljevanja velikokrat lahko zgodi, da kontrast barv preseže hranilno kapaciteto 8 bitov, moramo za oči prijazne rezultate uporabiti večjo barvno globino. Razliko lahko takoj opazimo, če na sliki 6.6 gledamo barvo in sij sonca. V našem primeru uporabimo za barvno globino vsakega barvnega kanala osvetljene slike 16 bitov. Tako omogočimo možnost izrisa zelo svetlih in tudi temnih površin hkrati. Ker pa naš zaslon ni sposoben prikazati take barvne globine, moramo na koncu to zmanjšati nazaj na 8 bitov, da sliko lahko pravilno prikažemo. Omenjeno pretvorbo opravimo v tem postprocesirnem koraku. Tu nastavimo tudi, katerim barvam damo več poudarka za spremembo kontrasta ali odtenka, in določimo, kako svetla bo končna slika z uporabo faktorja izpostavitve [1].



Slika 6.6: Leva slika prikazuje omejeno barvo na LDR, desna pa neomejene, malenkost nasičene, končne HDR barve, preslikane v LDR.

Poglavje 7

Rezultati

Rezultat diplomske naloge je prikazani animirani model, ki smo ga najprej ustvarili v programu za grafično 3-dimenzionalno modeliranje Blender. Robusten animacijski sistem uporabniku zmanjša delo pri animiranju tako, da skrbi za gladko interpolacijo med posameznimi pozami in s tem zagotovi veliko prilagodljivost pri manipulaciji in generiranju proceduralnih transformacij kosti. Z uporabo konfiguracijske datoteke za definiranje animacijskih akcij in popolnoma ločenim shranjevanjem poz lahko dosežemo solidne animacije v zelo kratkem času. Prav tako pa lahko delo pohitrimo z uporabo ene od animacijskih poz v različnih akcijah.

Z uporabo inverzne kinematike v kombinaciji z različnimi načini odkrivanja kolizij naše fizikalne knjižnice smo uspeli zgenerirati proceduralne animacije, odvisne od okolice v realnem času. Z algoritmom inverzne kinematike FABRIK nam je uspelo obravnavati tudi več omejevalnih končnih točk skeleta in tako smo rezultat lahko uporabili za naravno prilagajanje animacijskih poz bližnji okolici, kar lahko opazimo na sliki 7.1. Tako smo lahko za predstavitev delovanja izdelali več različnih uporabnih scenarijev. Za začetek nam je z uporabo sledenja žarkov v fizikalni knjižnici uspelo zgenerirati poze za dinamično sklanjanje pred prihajajočimi ovirami. Potem smo za večji izziv uporabili kolizijsko sfero, ki je zaznala bližnja telesa, te pa smo glede na orientacijo uporabili kot končne točke algoritma inverzne kinematike in tako se

je animacijsko telo ob hoji blizu kolizijskih predmetov z rokami na rahlo dotikalo zaznanih površin. Pri tem zaradi pomanjkanja omejitev rotacij sklepov najboljših rezultatov nismo uspeli doseči, saj predvajane animacijske poze niso prilagojene za nagibanje telesa med hojo.

Slika 7.1: Končni izris skeletno animiranega telesa s popravljenimi transformacijami nog z uporabo algoritma inverzne kinematike FABRIK.

Z uporabo knjižnice za fiziko nam je s pomočjo sil vzmeti uspelo iz transformacij določenih kolizijskih teles simulirati dinamično gibanje togega krila. Prav tako smo s pomočjo odkrivanja kolizij zagotovili, da krilo ne bo presekalo katerega od drugih objektov. Poleg dinamično generiranih poz nam je z uporabo začasnih transformacij iz dveh animacijskih poz uspelo glede na spremenljivko izračunati in uporabiti transformacije vmesnih interpoliranih

vrednosti (npr. prehod iz hoje v tek).

7.1 Problemi

Na večje probleme ob izdelavi naloge nisem naletel. Nekaj težav je bilo sicer z raztegom celotnega modela, saj so poleg mreže oglišč drugačnih velikosti tudi vsi podatki povezani s skeletno animacijo. Tako so se pri kakšnem zapletenem zaporedju množenja matrik sklepi pogosto preveč oddaljili med sabo in prišlo je do nepravilnega izrisa telesa. Primer napačnih velikosti v matriki je viden na sliki 7.2.

Na zanimivo težavo sem naletel tudi, ko je ob napaki v animacijskem sistemu nekje prišlo do deljenja z 0, kar je pokvarilo celotno množico transformacijskih matrik in aplikacijo drastično upočasnilo, kar je sčasoma pripeljalo do popolnega sesutja. Nekoliko zapleteno je bilo dobiti tudi posamezne preobrazbe kosti iz končnih matrik rezultata inverzne kinematike, ki se uporabljajo pri spremembi animacijskih poz na časovnici glede na bližino kolizijskih teles.

Absolutno največji izziv pa so zaradi nepoznavanja različnih formatov predstavljali prehodi med koordinatnimi sistemi različnih datotek in knjižnic. Največ časa smo porabili za problem prehoda med odmikom sklepov od koordinatnega izhodišča iz glavne animacijske datoteke, kjer so shranjena oglišča 3-dimenzionalnega modela in pripadajoči podatki, vključno z utežmi kosti, v odmik posameznih datotek animacijskih poz. Na koncu pa se je izkazalo, da pretvorba sploh ni bila potrebna, in tako smo lahko odmike, pridobljene iz datoteke s skeletno strukturo, brez problema nadomestili z odmiki, ki smo jih prebrali iz scenskega grafa datotek animacijskih poz.

Slika 7.2: Problem z vizualizacijo napačno skaliranih transformacij kosti.

Poglavje 8

Sklepne ugotovitve

V diplomski nalogi smo uspešno implementirali animacijski sistem za gladek prehod med pozami skeletne animacije. Ta za predvajanje želenih animacijskih akcij deluje solidno, vendar ne podpira mešanja več različnih poz (*angl. animation blending*), kar pomeni izvajanje večjega števila poz hkrati (npr. mahanje z rokami ob hoji). Zaradi abstrakcije podatkovnih struktur pri predstavitvi animacijskih poz pa bi se po mojem mnenju trenutni sistem relativno hitro dalo nadgraditi, da bi podpiral tudi tako vrsto animacije, saj bi po mešanju želenih poz dobljene rezultate brez problema nadomestili z obstoječo pozo na animacijski časovnici, v katero trenutno prehajamo. Poleg tega pa bi lahko dodali še kakšen parameter prehodu med pozami, kot na primer interpolacijsko funkcijo, ki bi jo sistem uporabil. V takem primeru bi lahko do izraza prišla vzmetna interpolacija, ki bi nakazala na neko ustavitev akcije in vpliv različnih sil na telo.

Projektu bi lahko dodali tudi morfne tarče oziroma kompleksno animacijo, opisano v poglavju 2.3. Te bi lahko pri bolj natančnih obraznih potezah služile kot izražanje čustev na obrazu. Za naše potrebe te niso najbolj pomembne, vendar bi lahko v poštev prišle pri katerih drugih animacijah. Naše dinamično krilo, za transformacije katerega skrbi knjižnica za fiziko z uporabo omejene vzmeti, bi lahko spremenili v mrežo ogjišč, ki bi jih obravnavali kot mehka telesa (*angl. soft body*) in tako ustvarili animacijo višje kakovo-

sti, vendar bi zaradi te izgubili trenutno stabilnost in hitrost procesiranja. Pri takem problemu sicer ne gre neposredno za morfno animacijo, ker moramo oglišča spreminjati nemudoma ob odkritju trka, vendar to vseeno lahko uporabimo za podobne, bolj statične animacije mreže oglišč.

Implementirali smo tudi preprost, a zelo hiter algoritem inverzne kinematike, imenovan FABRIK. Slednjemu pa nismo dodali nobenih omejitev vrtenja. če pa bi jih uspešno implementirali, bi lahko te omejitve upoštevali ob računanju inverzne kinematike in tako za rezultat dobili na pogled veliko bolj naravno gibanje našega animiranega telesa. Take omejitve so za proceduralno generirane poz še posebej pomembne, saj se v večini primerov zgodi, da samo katera od kosti v naši hierarhiji prevzame večinsko vlogo vrtenja končnih transformacij rezultata inverzne kinematike. Tak problem se lahko občasno opazi pri hoji telesa, ko se ob oviri zavrti samo spodnji del noge (kolenski sklep), zgornji pa ostane relativno nespremenjen.

Z uporabo fizikalne knjižnice smo uspešno prišli tudi do proceduralnih transformacij krila kot dela oblačila skeletnega telesa. Ker je pri taki animaciji celoten končen videz odvisen od nastavitev omejitev sklepa med glavno kostjo in posameznimi kostmi krila, bi si za izboljšavo izgleda lahko vzeli še malo več časa za spreminjanje parametrov fizikalnih teles in njihovih omejenih sklepov. Poleg krila smo fizikalno knjižnico uporabili tudi za nekaj popolnoma proceduralno generiranih poz z uporabo sledenja žarkom in iskanja bližinskih trkov. Z dobljenimi rezultati smo lahko z uporabo algoritma za inverzno kinematiko izračunali pravilne transformacije sklepov, vendar kakor sem omenil že zgoraj, bi se videz teh lahko še dodatno izboljšal z uporabo naravnih telesnih omejitev. Za boljši izgled takih animacij bi našemu skeletu lahko dodali tudi konfiguracijsko datoteko s težami posameznih udov, ki bi jih lahko potem pri proceduralni generaciji uporabili za popravke naših animacij. Na primer, če bi imel naš junak v roki nek težek predmet, bi se to v animaciji odražalo s poskusom uravnoteženja telesnega gravitacijskega središča (nagib v nasprotno smer) ali pa bi se telo v sproščeni pozi nagnilo bolj proti smeri, kamor ga na novo nastavljena sila vleče.

Literatura

[1] Aces HDR tone mapping. Dosegljivo: https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/. [Dostopano: 12. 11. 2018].

[2] Assimp importer. Dosegljivo: http://assimp.sourceforge.net/lib_html/class_assimp_1_1_importer.html. [Dostopano: 20. 12. 2018].

[3] Bullet physics examples. Dosegljivo: https://github.com/bulletphysics/bullet3/tree/master/examples. [Dostopano: 1. 1. 2019].

[4] Caliko library. Dosegljivo: https://openresearchsoftware.metajnl.com/articles/10.5334/jors.116/. [Dostopano: 3. 1. 2019].

[5] Deferred shading. Dosegljivo: http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf. [Dostopano: 30. 1. 2019].

[6] Entity component system. Dosegljivo: https://en.wikipedia.org/wiki/Entity%E2%80%93component%E2%80%93system. [Dostopano: 22. 12. 2018].

[7] GL math. Dosegljivo: https://glm.g-truc.net/0.9.9/index.html. [Dostopano: 18. 12. 2018].

[8] Good bloom tutorial. Dosegljivo: http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/. [Dostopano: 11. 11. 2018].

[9] Interpolation types. Dosegljivo: http://paulbourke.net/miscellaneous/interpolation/. [Dostopano: 1. 1. 2019].

[10] Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. Dosegljivo: https://groups.csail.mit.edu/drl/journal_club/papers/033005/buss-2004.pdf. [Dostopano: 31. 1. 2019].

[11] Inverse kinematics. Dosegljivo: https://en.wikipedia.org/wiki/Inverse_kinematics. [Dostopano: 17. 12. 2018].

[12] Jacobian inverse IK overview. Dosegljivo: https://medium.com/unity3danimation/overview-of-jacobian-ik-a33939639ab2. [Dostopano: 3. 1. 2019].

[13] Morph target animation. Dosegljivo: https://en.wikipedia.org/wiki/Morph_target_animation. [Dostopano: 16. 12. 2018].

[14] OpenGL. Dosegljivo: https://opengl.org/sdk/docs/books/SuperBible/. [Dostopano: 31. 1. 2019].

[15] PugiXML. Dosegljivo: https://pugixml.org/. [Dostopano: 22. 12. 2018].

[16] Skeletal animation. Dosegljivo: https://en.wikipedia.org/wiki/Skeletal_animation. [Dostopano: 26. 12. 2018].

[17] Skeletal animation tutorial. Dosegljivo: http://ogldev.atspace.co.uk/www/tutorial38/tutorial38.html. [Dostopano: 26. 12. 2018].

[18] Spring interpolation. Dosegljivo: https://evgenii.com/blog/spring-button-animation-on-android/. [Dostopano: 3. 12. 2018].

[19] Tutorial 17 : Rotations. Dosegljivo: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/. [Dostopano: 4. 2. 2019].

[20] Volumetric light scattering. Dosegljivo: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch13.html. [Dostopano: 12. 11. 2018].

[21] Andreas Aristidou and Joan Lasenby. Fabrik: A fast, iterative solver for the inverse kinematics problem. *Graphical Models*, 73(5):243–260, 2011.

[22] Ben Kenwright. Inverse kinematics – cyclic coordinate descent (CCD). *Journal of Graphics Tools*, 16(4):177–217, 2012.

[23] Morgan McGuire, Padraic Hennessy, Michael Bukowski, and Brian Osman. A reconstruction filter for plausible motion blur. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games 2012 (I3D'12)*, February 2012. Proceedings of the ACM Symposium on Interactive 3D Graphics and Games.

[24] Morgan McGuire, Michael Mara, and David Luebke. Scalable ambient obscurance. *Proceedings of ACM SIGGRAPH / Eurographics High-Performance Graphics 2012 (HPG '12)*, June 2012. High-Performance Graphics 2012.