Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Martin Plaznik

Primerjava odprtokodnih javanskih fizikalnih pogonov
A comparison of open-source java physics engines

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matija Marolt

Ljubljana 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00077/2013
Datum: 05.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MARTIN PLAZNIK**

Naslov: **PRIMERJAVA ODPRTOKODNIH JAVANSKIH FIZIKALNIH POGONOV**
A COMPARISON OF OPEN-SOURCE JAVA PHYSICS ENGINES

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V diplomski nalogi raziščite odprtokodne fizikalne pogone in algoritme, ki jih uporabljajo za detekcijo trkov. Vse pogone preizkusite v praksi in ocenite kateri se na razvitih testnih primerih najbolje obnese s stališča hitrosti in natančnosti.

Mentor:
doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Martin Plaznik, z vpisno številko **63090112**, sem avtor diplomskega dela z naslovom: Primerjava odprtokodnih javanskih fizikalnih pogonov

Primerjava odprtokodnih javanskih fizikalnih pogonov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne DD. MM 2013 Podpis avtorja:

Na tem mestu se zahvaljujem docentu dr. Matiji Maroltu za vodenje in pomoč pri izdelavi te diplomske naloge. Poleg tega se zahvaljujem tudi ostalim asistentom in profesorjem, ki so mi podali veliko novih znanj iz sveta računalništva.

Kazalo

Slike

Tabele

Povzetek

Z uporabo fizikalnih pogonov v trodimenzionalnih računalniških igrah lahko zelo dobro simuliramo fizikalne pojave iz zunanjega sveta in se tako igralcu še bolj približamo. Ker je na tržišču vedno več odprtokodnih fizikalnih pogonov in sistemov za detekcijo trkov, ki so del le-teh, je programerjeva odločitev glede izbire pravega pogona vedno težja in težja.

Cilj te diplomske naloge je predstaviti nekaj fizikalnih pogonov in ugotoviti, kateri se najbolje obnese v določeni situaciji. Na začetku diplomskega dela si bomo pogledali, kateri so pogosteje uporabljeni pristopi pri implementaciji detekcije trkov v fizikalnih pogonih. Nato bomo preverili, katere izmed prej naštetih algoritmov uporabljajo določeni fizikalni pogoni. Kot primere fizikalnih pogonov bomo predstavili javanske različice pogonov ODE in Bullet ter pogon Jinngine.

Nato bomo naštete pogone tudi preizkusili v praksi. Sprva si bomo pogledali, katere metode pogonov je potrebno poklicati za uspešne inicializacije pogonov in statičnih ter dinamičnih objektov. Na koncu bomo nad pogoni izvedli tudi nekaj testov in s pomočjo rezultatov poizkusili odgovoriti na vprašanje o tem, kateri izmed izbranih fizikalnih pogonov je najboljši.

Abstract

With implementation of physical engines in three-dimensional computer games we can make our environment to appear more real to the player of the game. Because the market presents more and more open source implementations of physical engines and the collision detection systems, which are usually part of engines, programmers decision, about usage of particular physical engine, becomes harder and harder.

The main goal of this diploma assignment is to present few physical engines and to find out, which one of them is better in certain case of usage. At the beginning, we will take a quick overview of most used practices in implementation of collision detection systems in physical engines. Next, we will check, which algorithms are used by selected physical engines. As show cases of engines we will use ODE, Bullet and Jinngine.

After that, we will implement those engines into our project written in Java programming language. At beginning, we will take a look at methods needed in order to set up physical engine and objects that are implemented in those libraries. At the end, we will also run some tests over physics engines and try to answer the question about, which engine is the best.

Poglavje 1

Uvod

Eden glavnih delov sleherne računalniške igre, v kateri je uporabnik oziroma igralec v interakciji z okoljem, je fizikalni pogon. Ta v naše virtualno okolje prinaša fizikalne zakone, ki veljajo v realnem svetu. Namen tega je ustvariti igro, ki bo igralcu delovala čim bolj resnično. Kljub temu so fizikalni pogoni pri tem še vedno omejeni na neko natančnost in zmogljivost računalniškega sistema, na katerem »prebivajo«.

Zaradi omejitev na zmogljivost računalniškega sistema, si pri računalniški grafiki ali bolj natančno, detekciji trkov, pomagamo z različnimi pristopi. Eden teh pristopov je uporaba očrtanih teles. Tu dinamično telo v prostoru, ki je lahko poljubne oblike, aproksimiramo z očrtanim telesom, ki je prav tako poljubne, a bolj enostavne oblike. Pri drugem je izbira očrtanega telesa odvisna od programerja, najpogosteje uporabljene oblike očrtanih teles pa so krogle, z osmi poravnani kvadri, elipsoidi, itd.

Na časovno zahtevnost algoritmov za detekcijo trkov vpliva tudi razdelitev prostora. Tega lahko razdelimo na več načinov z uporabo mrež ali različnih tipov drevesnih struktur, ki si jih bomo podrobneje pogledali v nadaljevanju. Na ta način se lahko izognemo potratnemu testiranju detekcije trkov med objekti po principu vsak z vsakim časovne zahtevnosti $O(n^2)$.

Poleg dveh zgoraj naštetih pristopov je pri detekciji trkov pogosta tudi uporaba hierarhije očrtanih teles. Tu gradimo drevo očrtanih teles, pri tem

pa velja, da dve telesi ne trkata drug ob drugega, če velja, da telesi pripadata različnim vejam drevesa in se očrtani telesi na teh dveh vejah prav tako ne sekata.

V zadnjih letih se je hitro razmahnilo število prosto dostopnih fizikalnih pogonov, pri čemer so nekateri omejeni le na dvo-dimenzionalni svet, spet drugi pa so namenjeni delu v trodimenzionalnem svetu. Pri vse širši ponudbi se programerji težko odločijo, kateri fizikalni pogon bodo uporabili v svoji računalniški simulaciji, igri ali animiranem filmu.

Pri izdelavi računalniških iger se pogosto pojavi potreba po sistemu, ki bo detektiral trke. Prvo vprašanje, ki se nam pri tem porodi, je, kateri pogon izbrati, oziroma kateri pogon se bo v našem primeru najbolje odrezal? Na koncu se moramo vseeno odločiti za eno implementacijo fizikalnega pogona, a vprašanje, ki ostaja, je, ali je bila izbira pravilna? V tem diplomskem delu bomo poizkušali ugotoviti, kateri fizikalni pogon bi bil v primeru izdelave 3D igre najbolj primeren in kateri pogon bi se bolje odrezal v računalniški simulaciji, kjer nas zanima predvsem natančnost pogona. Na primerih implementacije si bomo ogledali delovanje fizikalnih pogonov JBullet, ODE4J in Jinngine, ter nad njimi izvedli tudi nekaj preprostih testov.

Med pisanjem te diplomske naloge sem ugotovil, da je tema detekcije trkov oziroma fizikalnih pogonov zelo široka. Zaradi tega bi lahko samo del te diplomske naloge opisali tako obširno, da bi dosegal dolžino in širino dela, ki je pred vami. Ker bi opis vsega, kar je vredno omeniti pri tej temi in širše, zahteval več prostora in časa, bralcu prepuščam, da si temo podrobneje pogleda v virih, navedenih na koncu diplomske naloge.

Ob zaključku uvoda naj povem še nekaj o sami strukturi diplomskega dela. V poglavju 2 bomo preučili nekaj pristopov pri detekciji trkov. Nato bomo v tretjem poglavju preverili katere izmed teh pristopov uporabljajo implementacije izbranih fizikalnih pogonov. V četrtem poglavju si bomo podrobneje ogledali same implementacije javanskih različic fizikalnih pogonov in v poglavju pet tudi izvedli nekaj testov nad njimi. Ob zaključku, v poglavju 6, bomo poizkušali interpretirati dobljene rezultate in ugotoviti, kateri

izmed pogonov je najboljši v določeni situaciji.

Poglavje 2

Pristopi pri detekciji trkov

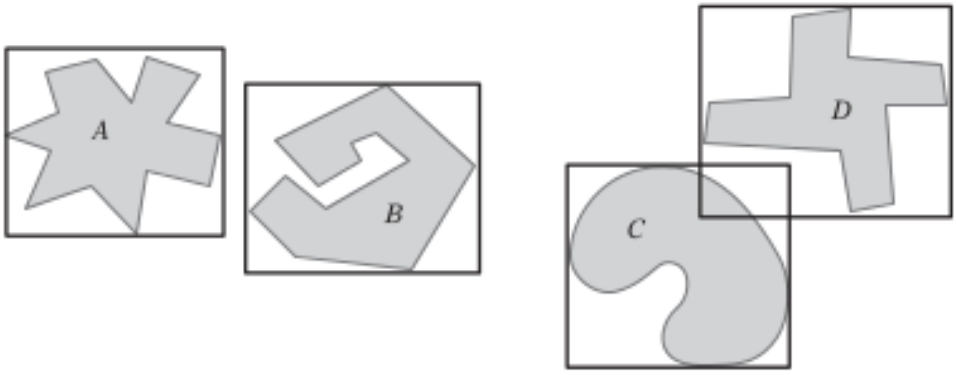
2.1 Očrtana telesa

Ena glavnih nalog fizikalnega pogona je ugotavljanje trka med dvema ali več telesi. Naivni programer bi se verjetno naloge lotil tako, da bi testiral trkanje dveh teles direktno. To bi bilo z vidika časovne zahtevnosti zelo potratno. Čas, ki bi ga fizikalni pogon porabil za preverjanje trka med dvema telesoma, pa bi naraščal s količino poligonov, iz katerih so fizikalni objekti sestavljeni. Da bi skrajšali čas, potreben za računanje trka oziroma zmanjšali število korakov za preverjanje trka med dvem objektoma, se v fizikalnih pogonih uporablja očrtana telesa.

Ideja očrtanih teles je v tem, da za očrtano telo uporabimo telo, za katerega je računanje detekcije manj računsko zahtevno in tako prihranimo čas. Ko zaznamo trk med očrtanimi telesi, pa moramo preveriti tudi trk teles znotraj očrtanega. Seveda je ta drugi korak ponavadi veliko zahtevnejši iz stališča računske zahtevnosti, a v praksi se pokaže, da so le redka telesa dovolj blizu skupaj, da bi prišlo do prekrivanja očrtanih teles.

Časovna zahtevnost računanja trka med poligoni dveh fizikalnih objektov je $\mathrm{O}(n^2)$. Vseeno pa obstajajo določeni algoritmi, s katerimi lahko reduciramo število potrebnih izračunov. Na koncu se lahko lahko tudi sami vprašamo, ali je naš cilj izdelava izredno natančne računalniške simulacije, ali se želimo

zgolj približati fiziki iz realnega sveta. V slednjem primeru lahko računamo le trke med očrtanimi telesi in se tako izognemo zahtevnemu računanju trka med objekti, ki jih očrtana telesa vsebujejo.



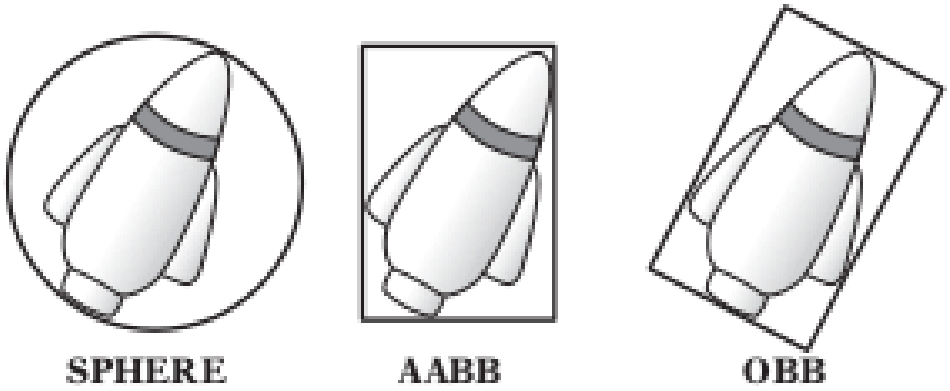
Slika 2.1: Očrtani telesi objektov A in B se ne prekrivata, zato lahko sklepamo, da ni prišlo do trka med objektoma. Nasprotno je pri telesih C in D, kjer se očrtani telesi prekrivata. Zaradi tega moramo tu testirati trk med samima objektoma.

2.1.1 Karakteristike očrtanih teles

V tem poglavju bom poizkušal razložiti, kakšnim kriterijem mora odgovarjati določeno telo, da ga lahko uporabim kot očrtano telo. V delu [1] avtor uporabi naslednje kriterije:

- Cena računanja preseka oziroma trka med očrtanimi telesi mora biti nizka.
- Očrtano telo se mora ozko prilagajati vsebovanemu objektu.
- Cena računanja telesa mora biti nizka.
- Rotacije in transformacije teles morajo biti lahke oziroma hitre.
- Telo mora uporabiti malo pomnilniškega prostora.

Čeprav obstaja več različnih oblik teles, ki bi jih programer lahko uporabil za implementacijo očrtanih teles, se bom v tej diplomski nalogi dotaknil le nekaterih. Kot primere očrtanih teles bom predstavil krogle, AABB in OBB.



Slika 2.2: Slika prikazuje tri tipe očrtanih teles. Od leve proti desni si sledijo krogla, z osmi poravnan kvader in orientiran kvader.

2.1.2 Z osmi poravnan očrtan kvader (AABB)

AABB je najpogosteje uporabljeno očrtano telo. Pri tem telesu gre za kvadrat v primeru dvodimenzionalnega prostora ali kvader v primeru trodimenzionalnega prostora, ki ima svoje osi ves čas poravnae s koordinatami danega koordinatnega sistem. Glavna prednost tega tipa očrtanega telesa je v tem, da omogoča hitro računanje prekrivanja.

Poznamo tri različne predstavitve AABB-jev in sicer:

1. Z dvema nasprotnima točkama(min, max) v koordinatnem sistemu.

2. Z minimalno kotno točko in premeri.

3. S točko v centru in polmeri.

Kar zadeva karakteristike v prejšnjem poglavju, je glede shranjevanja AABB-ja najbolje uporabiti tretjo različico predstavitve. Ta različica namreč

porabi najmanj prostora, saj je mogoče polmere običajno v spomin zapisati z manj biti kot točko. Poleg tega je pri drugem in tretjem načinu predstavitve, v primerjavi s prvim, ob translaciji potrebno spremeniti le vrednosti koordinat ene točke, medtem ko je pri prvem načinu potrebno spremeniti vrednosti koordinat dveh točk.

Računanje preseka dveh AABB-jev

Pri računanju preseka dveh AABB-jev velja, da se dve telesi prekrivata takrat, ko se prekrivata v obeh dveh, če gre za dvodimenzionalni prostor, oziroma v vseh treh oseh, če gre za trodimenzionalni prostor. Implementacija zgornjega pravila ni draga v smislu računske zahtevnosti in tako imamo še eno potrditev, da je AABB primerno uporabiti za očrtano telo.

Za računanje preseka med dvema AABB očrtanima telesoma se največkrat uporablja »sweep and prune« algoritem. Več o tem algoritmu je navedeno v poglavju o algoritmih.

Posodabljanje AABB-ja

Ko se objekti, ki jih AABB obkroža, premikajo po prostoru, je potrebno skupaj z njimi premakniti tudi pripadajoče očrtano telo. Prav tako je potrebno pogledati tudi rotacijo obkroženega objekta in ustrezno popraviti velikost AABB-ja. Pri tej nalogi imamo ponovno več pristopov.

Težave s popravljanjem velikosti oziroma oblike AABB-ja se lahko lotimo tako, da je AABB že v začetku dovolj velik. Na ta način lahko telo, ki ga AABB vsebuje, obračamo kolikor hočemo in ne bo prebilo meje, ki jo zarisuje očrtano telo. Na podoben način lahko namesto AABB-ja uporabimo kroglo, ki je v tem primeru bolj optimalna izbira.

Drugi način je, da poiščemo zunanje točke telesa, ki ga AABB obkroža in na ta način AABB kar najbolje prilagodimo obkrožajočemu telesu. Do zunanjih točk lahko tu pridemo tako, da se sprehodimo po vseh točkah, ki predstavljajo telo in shranimo le tiste, ki so najbolj oddaljene od vektorja smeri gibanja. Algoritem za iskanje zunanjih točk telesa v tem primeru

lahko izvedemo s časovno zahtevnostjo $O(logn)$, pri čemer je n število točk telesa.

Do zunanjih oziroma ekstremnih točk lahko pridemo tudi tako, da hranimo seznam 6 točk telesa v 6 različnih smereh. Pri tem po vsakem premiku pogledamo, ali je vsaka od teh točk še vedno ekstremna. Če temu ni tako, potem mora biti ekstremna točka sosednja tej. Za iskanje sosednje točke, ki je novi ekstrem, pa ne potrebujemo veliko truda. Težava pri tem načinu je le v tem, da ga lahko uporabimo samo pri konveksnih telesih.

Zadnji pristop pa je naslednji: fizikalni pogon preprosto rotira prejšnji AABB skupaj z obkrožujočim objektom in okrog AABB-ja izriše še en AABB. Ta pristop je najpogosteje uporabljen, hkrati pa ne zagotavlja AABB-ja, ki bi se popolnoma prilegal telesu, ki ga obkroža.

2.1.3 Krogla

Krogla je tako kot AABB ena od pogosteje uporabljenih oblik za očrtano telo, saj prav tako omogoča nezahtevno testiranje preseka. Poleg tega je krogla tudi rotacijsko invariantna. To pomeni, da se objekt v krogli lahko obrača v vse smeri, oblika krogle pa ostaja enaka. Ta lastnost zelo poenostavi premikanje telesa po prostoru, saj nas zanima le translacija.

Krogla je definirana s sredinsko točko in polmerom okrog nje. V 3D prostoru to pomeni, da potrebujemo za zapis krogle le 4 komponente. To so tri koordinate sredinske točke in dolžina radija, kar pomeni, da je krogla najbolj optimalna oblika, kar zadeva spomin. Poleg tega lahko center krogle poravnamo s centrom obkrožujočega telesa in tako moramo shraniti samo radij krogle.

Računanje preseka dveh krogel

Ugotavljanje, ali se dve krogli prekrivata, je zelo enostavno. Vse, kar potrebujemo, je izračunati razdaljo med centralnima točkama krogel in jo primerjati z vsoto njunih radijev. Če je vsota radijev večja od razdalje med točkama, se krogli prekrivata.

Posodabljanje krogle

Izračunavanje optimalne krogle, ki obkroža neko telo, je bolj zahtevno od računanja idealnega AABB-ja. Za to nalogo poznamo več različnih algoritmov.

Eden od načinov, da dobimo kroglo, je, da najprej izračunamo AABB za notranji objekt in center AABB-ja uporabimo za center krogle. Za radij v tem primeru uporabimo razdaljo do najbolj oddaljene točke. Na ta način ne dobimo krogle, ki bi se najbolje prilegala objektu, kljub temu pa je to eden od algoritmov, ki omogoča hitro računanje krogle okrog telesa. Postopek iskanja krogle poteka tako:

1. Na začetku poišči 6 točk za vsako os v koordinatnem sistemu.
2. Izberi dve točki A in B, kjer je razdalja med njima najdaljša.
3. Postavi središče krogle na sredino med A in B.
4. Iteracija po vseh zunanjih točkah telesa.
5. Če je razdalja med središčem in zunanjo točko večja od radija krogle, povečaj radij.

Drugi način je izboljšava prvega. Namesto da izvedemo le eno iteracijo po točkah telesa, izvedemo več iteracij, pri čemer pred vsako iteracijo malo skrčimo, da dobimo malo premajhno očrtano telo, ki ga potem v iteraciji prilagodimo. Algoritem gre torej tako:

1. Na začetku poišči 6 točk za vsako os v koordinatnem sistemu.
2. Izberi dve točki A in B, kjer je razdalja med njima najdaljša.
3. Postavi središče krogle na sredino med A in B.
4. Skrči radij krogle za nekaj odstotkov.
5. Iteracija po vseh zunanjih točkah telesa.

6. Če je razdalja med središčem in zunanjo točko večja od radija krogle, povečaj radij.

7. Vrni se na točko 4.

Ta postopek nam v praksi vrne očrtano telo, ki se bolje prilega telesu. Poleg tega lahko postopek še izboljšamo in sicer tako, da takrat, ko iteriramo po točkah telesa uporabimo naključni vrstni red.

Za računanje krogle obstaja še nekaj algoritmov, ki jih v tej diplomski nalogi ne bom omenjal. Lahko si jih preberete v viru [1].

2.1.4 Oriented bounding box OBB

OBB je očrtano telo, ki je predstavljeno podobno kot AABB, le da ni poravnano s koordinatnimi osmi, temveč gre za lokalno orientacijo. OBB lahko predstavimo na različne načine in sicer:

1. Kot zbirko šestih ploskev.

2. Kot zbirko treh parov vzporednih ploskev.

3. Kot kotno točko s tremi ortagonalnimi vektorji po robovih.

4. Kot centralno točko z rotacijsko matriko in tremi razdaljami do ploskev.

Predstavitev, ki omogoča najcenejše računanje prekrivanja med dvema OBB telesoma, je napisana pod točko tri. Ta predstavitev še vedno zahteva precej prostora za shranjevanje podatkov v primerjavi z AABB in kroglo in zaradi tega ni najbolj optimalna izbira za očrtano telo.

Računanje preseka dveh OBB-jev

Če želimo ugotoviti, ali se dva OBB-ja sekata, moramo uporabiti tako imenovano metodo ločitvenih osi. Ta test je glede na test pri krogli in AABB bolj računsko zahteven in ga bomo podrobneje pregledali v naslednjih poglavjih. Boljšega algoritma, ki bi nam v trodimenzionalnem prostoru dal bolj natančne odgovore o tem, ali se dve telesi prekrivata ali ne, ni.

2.1.5 Ostali tipi očrtanih teles

Čeprav sem v tej diplomski nalogi opisal le nekaj očrtanih teles, ki jih najpogosteje srečamo pri različnih implementacijah fizikalnih pogonov ali pogonov za detekcijo trka, obstaja še veliko več tipov očrtanih teles, ki so v tej diplomski nalogi žal izvzeti. Naj jih naštejem le nekaj:

- elipsoid
- cilinder
- kapsula
- trikotnik ali piramida
- konveksna ovojnica

Pri izbiri tipa očrtanega telesa, ki bo najbolje ustrezal naši igri ali simulaciji moramo tako izbirati med zahtevnostjo implementacije in hitrostjo detekcije trkov. Predstavljajte si, da izdelujemo igro smučarskega krosa, kjer se več igralcev naenkrat spušča po progi. Ob progi so ponavadi postavljeni podolgovati količki, ki so tudi fizični objekt, ki se zamaje v primeru, da se igralec zaleti vanj. Pri tem fizičnem objektu bi v primeru uporabe krogle kot očrtanega telesa, dejanski objekt zasedel odstotkovno zelo majhen del krogle. Tako bi vsakič, ko bi se smučar pripeljal mimo količka, detektirali prekrivanje. Zato bi bilo potrebno kasneje testirati tudi kompleksnejši test trka med telesoma znotraj dveh očrtanih teles, med smučarjem in količkom. Ker to ni optimalno, bi bilo za očrtano telo pri količku bolje uporabiti AABB, cilinder, elipsoid ali drug tip očrtanega telesa, ki bi se bolje prilegal količku.

2.1.6 Uporabni algoritmi pri detekciji trkov

Test ločitvenih osi

Kot sem zapisal v poglavju o očrtanih telesih tipa OBB, bom na tej točki razložil, kako deluje test ločitvenih osi za ugotavljanje trka med dvema telesoma. Ideja tega testa je, da obstajata dve koveksni telesi A in B, ki sta

sestavljeni iz končne množice točk. Za telesi velja, da se ne prekrivata, če obstaja premica v dvodimenzionalnem ali ploskev v trodimenzionanem prostoru, na katero projeciramo točke teh dveh teles in se intervali projekcij med seboj ne prekrivajo. Torej, če med ti dve telesi potegnemo premico ali ploskev p pravokotno na premico ali ploskev, na katero smo projecirali točke, mora biti interval projekcije telesa A na eni strani premice ali ploskve p in interva projekcije telesa B na drugi strani. Ker bi pri tem testu lahko uporabili neskončno mnogo premic ali ploskev, odvisno od števila dimenzij prostora, bi bilo to časovno zelo potratno. Zato se je bolje omejiti le na osi, ki nam vrnejo pravi rezultat v končnem številu korakov. In to so:

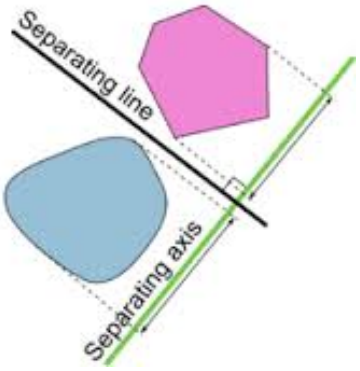
1. Osi, ki so vzporedne z normalami ploskev telesa A
2. Osi, ki so vzporedne z normalami ploskev telesa B
3. Osi, ki so vzporedne vsem vektorjem križnega produkta robov telesa A z robovi telesa B

Algoritem torej poteka tako:

1. Izberemo os iz zgoraj naštete množice.
2. Projeciramo vse točke telesa A na os.
3. Projeciramo vse točke telesa B na os.
4. Preverimo, ali se intervala projekcij točk telesa A in telesa B prekrivata.
5. Če se intervala prekrivata, potem se vrnemo na točko 1., izberemo naslednjo os in ponovimo postopek. Če se intervala ne prekrivata, potem vemo, da se telesi ne sekata in zaključimo s testom.

Kljub temu, da smo iz neskončno mnogo osi, potrebnih za testiranje, prišli le na $2 \times F + E^2$ osi, pri čemer F predstavlja število ploskev telesa A in B, ob predpostavki, da imata obe telesi enako število ploskev, E pa predstavlja število robov teles, test še vedno lahko poenostavimo. Namesto da vedno

začnemo testirati v enakem vrstnem redu, lahko na začetku testiramo os, ki je bila nazadnje pozitivna in upamo, da nam bo tudi v tej iteraciji dala pozitivno vrednost. Tako lahko dobimo rezultate že s testom ene same osi.



Slika 2.3: Slika prikazuje izvajanje algoritma ločitvenih osi.

»Sweep and prune« algoritem

Pri ugotavljanju prekrivanja med AABB objekti lahko uporabimo »algoritem sweep and prune«, ki se izkaže za zelo učinkovitega v tem primeru. Sam algoritem za primer dveh kvadrov v trodimenzionalnem prostoru deluje tako:

- Shranimo vse ekstremne vrednosti točk po osi x, y in z v seznam.
  - Ekstremna vrednost je najmanjša in največja vrednost, ki jo zavzema lik po vsaki osi posebej. Če imamo lika A in B, začetne točke označimo z $A0$ in $B0$, končne pa z $A1$ in $B1$. Za vsako os imamo svoj seznam. Seznam je dolg $2N$, pri čemer je N enak številu likov.
- Pripravimo še 3 sezname za vsako os posebej.
- Sprehodimo se po urejenih seznamih za osi x, y in z. Ko naletimo na prvo točko $A0$, dodamo v seznam, ki pripada osi, ki jo obdelujemo, znak A. Nato lahko naletimo na točko $B0$ ali na točko $A1$. Če naletimo na točko $B0$, dodamo na seznam znak B. V tem primeru imamo

naenkrat na seznamu par AB, kar pomeni, da obstaja možnost, da se lika prekrivata. Če naletimo na točko $A1$, potem znak A vzamemo iz seznama.

- Na koncu preverimo, ali na vseh seznamih obstajajo situacije, ko imamo v nekem trenutku na seznamu hkrati znaka A in B. Takrat vemo, da se lika prekrivata.

Algoritem se ponavadi uporablja za vse objekte v prostoru hkrati. V zgornjem primeru sem predpostavljal, da obstajata le 2 objekta v prostoru.

2.1.7 Povzetek

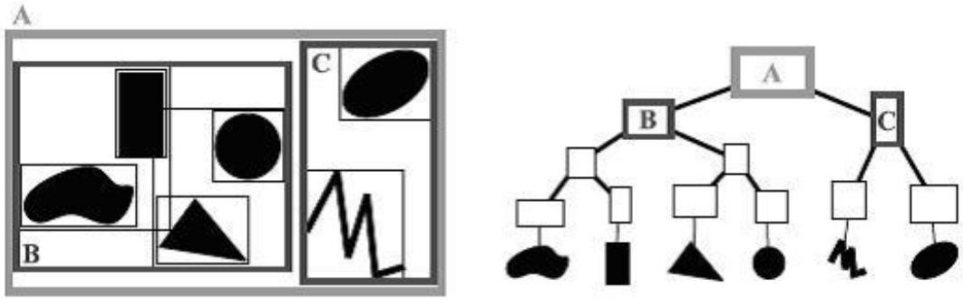
Očrtana telesa so preproste geometrijske oblike, ki obkrožajo kompleksnejša telesa v prostoru. Testiranje prekrivanja med očrtanimi telesi mora biti računsko enostavnejše kot test trka med kompleksnejšimi telesi, ki jih obkrožajo, ker drugače celotna ideja ne bi imela smisla. Najpogostejši tipi očrtanih teles so krogle in škatle. Če želimo, da se očrtano telo bolj prilega kompleksnejšim telesom, pa lahko uporabimo tudi konveksno ovojnico ali kakšen drugi tip. S tem sicer zmanjšamo število »false positive«, po drugi strani pa povečamo kompleksnost preverjanja prekrivanja dveh očrtanih teles.

2.2 Hierarhija očrtanih teles(BVH)

Čeprav z implementacijo očrtanih teles lahko zelo pohitrimo delovanje našega pogona za detekcijo trkov, moramo še vedno testirati telo po principu vsak z vsakim in je zato časovna zahtevnost enaka $O(n^2)$. Z uporabo hierarhije očrtanih teles lahko še hitreje ugotovimo, ali določena objekta trkata drug ob drugega ali ne.

Ideja pri hierarhiji očrtanih teles je v tem, da sestavimo drevo, kjer objekti v prostoru sestavljajo liste tega drevesa. Po drevesu se nato lahko rekurzivno premaknemo eno vozlišče višje in okrog očeta oziroma vozlišča ter njegovih sinov obkrožimo novo očrtano telo. To nato lahko rekurzivno ponavljamo

dokler ne pridemo do vrha drevesa. Pri implementaciji BVH ni vedno nujno, da naredimo očrtano telo okoli očeta, njegovih sinov, sinov sinov in tako dalje vse do listov drevesa. Vseeno pa je na tak način delovanje BVH-ja zelo poenostavljeno. Ko imamo tako zgrajeno drevo se lahko preprosto znebimo velike količine testov trka posameznih objektov med sabo. Na začetku lahko namreč testiramo ali med seboj trkata dve poddrevesi ali veji drevesa. Če ti dve veji ne trkata, potem ni potrebe po testiranju globje.



Slika 2.4: Slika prikazuje gradnjo BVH drevesa.

Glede na vse povedano o BVH, verjetno ni več potrebno veliko razmišljati, da ugotovimo, kakšne so zaželjene karakteristike BVH-ja. Ena od zelo pomembnih karakteristik je ta, da se objeki, ki pripadajo nekemu vozlišču, nahajajo blizu drug drugega v prostoru. Prav tako je pomembno, da liste drevesa predstavljajo minimalni objekti in da se detekcije trka lotimo pri vrhu drevesa, saj lahko hitreje odrežemo velik del drevesa, če ugotovimo, da se dve veji med seboj ne prekrivata.

2.3 Razdelitev prostora

Razdelitev prostora je še eden izmed načinov, kako se lahko znebimo odvečnih testov prekrivanja med očrtanimi telesi. Kot smo že omenili pri poglavju o BVH-jih, je testiranje prekrivanja med očrtanimi telesi po principu vsak z vsakim zelo potratno. Vrnimo se za trenutek k primeru smučarjev, ki se

spuščajo po progi označeni s količki. Na progi imamo lahko več količkov, ki so sicer dinamični objekti, a so pritrjeni v tla, vsak na svojem mestu. Količki torej ne morejo trkati med seboj. Če bi testirali po principu vsak z vsakim, lahko hitro ugotovimo, da bi v vsaki iteraciji poizkušali testirati tudi prekrivanje med očrtanimi telesi količkov, čeprav v resnici vemo, da do tega nikoli ne bo prišlo. Način, da se izognemo nepotrebnim dodatnim testom, je torej razdelitev prostora.

Ideja razdelitve prostora je v tem, da prostor razdelimo na več podprostorov. Sedaj imamo situacijo, kjer je v P podprostorih le N objektov. Vse kar nam sedaj ostane, je to, da testiramo objekte iz P množic oziroma podprostorov po principu vsak z vsakim. Za objekte v drugih podprostorih pa v tej iteraciji vemo, da ne bodo mogli na noben način trkati v objekte iz podprostora, ki ga trenutno obdelujemo.

Za razdelitev prostora na podprostore obstaja več možnosti. Poglejmo si naslednje:

2.3.1 Enotne mreže

Enotna mreža je enostaven način razdelitve prostora. Vse, kar mora program narediti, je, da postavi mrežo čez celotni prostor. Podprostori prostora so v tem primeru enake velikosti. Ker je temu tako, tudi ni težav, pri iskanju celice, ki vsebuje določeno koordinato prostora.

Vsak objekt v tem primeru lahko pripada enemu ali več podprostorom. V primeru, da objekt pripada le enemu podprostoru, lahko enostavno testiramo trk le z objekti v istem prostoru. V primeru, da telo pripada več prostorom, ker leži na meji med dvema ali več prostori, pa moramo testirati objekt z vsemi objekti, ki pripadajo vsem podprostorom, katerim pripada tudi sam.

Velikost celice

Enotna mreža je sicer enostavna za uporabo a, pri velikosti celice se lahko pri tem tipu mreže pojavijo težave. Pozorni moramo biti na naslednje:

- Če je celica premajhna, lahko objekti vedno pripadajo več celicam, ker so za eno preprosto preveliki. Zaradi tega bo objekt asociiran z več celicami, kar pomeni več zasedenega spomina za shranjevanje podatkov o tem, kateri objekti pripadajo določeni celici.

- Če je celica prevelika, zabredemo v težave, ko se večina objektov, ki so v povprečju precej manjši od celice nahaja v eni celici. Na ta način nismo z uvedbo razdelitve prostorov pohitrili skoraj ničesar.

- Če imamo objekte, ki so si po velikosti zelo različni, lahko nastopijo težave že pri izbiri velikosti same celice zaradi zgornjih dveh razlogov.

- Če je objekt preveč kompleksen, celica pa je lahko ravno pravšnja zanj, je včasih vseeno bolje, da je sam objekt s celicami razdeljen, ker tako lahko testiramo objekt po kosih.

2.3.2 Hierarhične mreže

Pri poglavju Enotne mreže smo spoznali, da imamo lahko težave pri izbiri velikosti celice. Do težav pride, če imamo velike celice. Na ta način smo sicer varni pred tem, da bi se veliki objekti nahajali v več celicah naenkrat, po drugi strani pa imamo lahko kar naenkrat veliko majhnih objektov v eni celici, kar nam zelo upočasni delovanje programa. Druga skrajnost je uporaba manjših celic, kjer se lahko en objekt, ki je večji od celice, nahaja v več letih. To nam ponovno dela preglavice, saj moramo v tem primeru testirati trk objekta z objekti v vseh celicah, kjer se nahaja. Težave z velikostmi celic lahko odpravimo tako, da uporabimo mreže hierarhične mreže.

Mreže tega tipa so sestavljene iz več celic različnih velikosti. Med seboj se celice prekrivajo, objekti se lahko premikajo po celicah iste ravni. Celice so urejene po ravneh od najmanjše do največje. Objekti so v celice razdeljeni po velikosti njihovih očrtanih teles. Če imamo na primer celico B, v kateri se nahajata dve manjši celici A in C, in imamo objekt d, ki je malo manjši od C in se nahaja na prostoru, ki ga C zajema, bomo objekt postavili v C.

Nato lahko dodamo še objekt e, ki se prav tako lahko nahaja na prostoru, ki ga zajema C, a je le-ta premajhna zanj. Objekt e bomo v tem primeru dodali v celico B, ki je večja in se nahaja eno raven višje od celic A in C. Na tak način razvrščanja objektov v celice pridemo do situcaije, ko se nek objekt v nekem trenutku lahko nahaja le v največ 4 celicah, če uporabljamo dvodimenzionalni prostor, oziroma v 8 celicah, če uporabljamo 3-dimenzionalni prostor.

2.3.3 Drevesne strukture

Osmiška drevesa

Pri osmiških drevesih ima vsak oče osem sinov. Vrh drevesa predstavlja kocka, ki jo nato razdelimo na 8 enako velikih podkock po x, y in z osi. Te podkocke nato rekurzivno delimo dalje. Rekurzijo deljenja ponavadi ustavimo takrat, ko dobimo kocke prej nastavljene najmanjše dovoljene velikosti. Tako dobimo osmiško drevo.

Osmiško drevo, ki je popolnoma poravnano, vsebuje $(8^n–1)/(8–1)$ vozlišč, pri čemer nam n predstavlja število nivojev osmiškega drevesa. To pomeni, da imamo pri drevesu tega tipa z 10 nivoji 153391689 vozlišč. Zato je v praksi bolje, če se ustavimo pri 5 ali 6 nivojih.

Pri drevesih te vrste poznamo takšne, ki se zgradijo na začetku iz samih statičnih objektov in takšne, ki jih gradimo dinamično. Pri tem je tako kot pri enotnih mrežah nek objekt lahko prisoten v dveh podprostorih, če se recimo nahaja na meji med njima. V tem primeru bomo objekt dodali v oba sinova. Pri dinamičnih tipih dreves pa imamo poleg omejitve globine tudi omejitev glede na število objektov v določenem podprostoru. Če je nek podprostor poln, ga razdelimo naprej in na ta način dobimo nove liste drevesa. Prav tako imamo minimalno mejo glede na število otrok v podprostoru. Če je otrok premalo, jih premaknemo v nivo višje. Višji nivo nato postane list drevesa.

Štiriška drevesa

Štiriška drevesa delujejo na popolnoma enak način kot osmiška drevesa. Razlika je le v tem, da se štiriška drevesa uporabljajo v dvodimenzionalnem prostoru in tu gradimo drevesa, kjer ima vsak oče v drevesu 4 sinove.

K-D drevesa

Pri osmiških drevesih smo prostor razdelili po treh oseh, ker smo imeli 3D prostor. Pri K-D drevesih pa lahko prostor razdelimo po K dimenzijah. Pri tem K ni nujno enak številu dimenzij v našem prostoru. V vsakem koraku pri tej vrsti dreves prostor razdelimo po natanko eni dimenziji. Kot primer lahko na začetku prostor razdelimo po x osi, nato y, z, x in tako dalje. Prav tako je za K-D drevesa potrebno omeniti tudi to, da se tu gradi binarno drevo, saj se v vsakem koraku prostor razdeli le na 2 dela. Pri osmiških drevesih smo prostor delili na 8 delov, kar pomeni, da je imelo neko vozlišče v osmiškem drevesu 8 sinov.

Podatke o tem, po kateri osi smo razdelili prostor in o ravnini, hranimo v vozlišču drevesa. Poleg tega v drevesu hranimo tudi podatke o telesih, ki se nahajajo v podprostorih, ki jih ravnine delijo. Če ravnina predmet seka, ga razpolovimo in dodamo v oba podprostora.

Gradnja drevesa se lahko ustavi glede na več različnih kriterijev. Pri detekciji trkov nas ponavadi zanima število objektov v vsakem podprostoru, saj želimo zmanjšati število potrebnih testov prekrivanja očrtanih teles. Torej gradnjo ustavimo, ko je v prostoru dovolj majhno število objektov.

BSP drevesa

BSP drevesa so zelo podobna K-D drevesom. Razlika je namreč le v tem, da ravnine, ki razdeljujejo prostor, niso poravnane z osmi prostora. To nam omogoča boljšo razdelitev prostora v primeru objektov, sestavljenih iz poligonov, saj se mu lahko ravnina, ki seka prostor, bolj prilagodi. Cilj je tako pri BSP kot tudi pri K-D drevesih zgraditi čim bolj poravnano drevo.

2.3.4 Povzetek

Ideja razdelitve prostora je v tem, da zmanjšamo število potrebnih kalkulacij prekrivanja očrtanih teles v prostoru. Prostor je tu razdeljen na N podprostorov, ti pa niso nujno istih dimenzij. V vsakem podprostoru imam lahko 0 ali več objektov. Testiranje prekrivanja lahko izvedemo za vsak prostor posebej, pri čemer testiramo le prekrivanje med objekti določenega prostora. Poleg tega nas zanim tudi to, ali se je oziroma se bo določen objekt premaknil iz enega podprostora v drugega.

V zgornjih poglavjih smo spoznali več različnih možnosti razdelitve prostora in sicer mreže s celicami enakih velikosti, hierarhične mreže in drevesne strukture, med katere spadajo osmiška ali štiriška drevesa, K-D drevesa ter BSP drevesa. Več o posameznem načinu razdeljevanja prostora pa si lahko preberete tudi v delih, navedenih v virih.

Poglavje 3

Pregled uporabe pristopov pri detekciji trkov v različnih fizikalnih pogonih

V tem diplomskem delu sem se odločil, da bom primerjal tri različne fizikalne pogone. To so Bullet game physiscs, Open Dynamics Engine oziroma ODE ter Jinngine fizikalni pogon. Povod za izdelavo tega diplomskega dela je bila seminarska naloga, ki smo jo izdelali pri predmetu Računalniška grafika in tehnologija iger, katerega nosilec je bil doc. dr. Matija Marolt. Cilj seminarske naloge je bila izdelava trodimenzionalne računalniške igre. To nam je na koncu tudi uspelo. Zelo pomemben del igre, ki smo jo izdelali, je bil tudi JBullet fizikalni pogon, ki je skrbel za to, da so se objekti v našem 3D prostoru premikali približno tako, kot bi se v realnem svetu. JBulle je pravzaprav različica fizikalnega pogona Bullet napisana v javi. Med samim razvojem seminarske nalogo smo si v skupini večkrat potavili vprašanje ali je bil JBullet fizikalni pogon res najboljša izbira, zato sem se po tem, ko smo seminarsko nalogo uspešno oddali in zagovarjali, odločil, da pridem do dna temu, zelo pogostemu vprašanju, ki si ga postavljajo programerji iz področja računalniških iger. Torej, katera knjižnica oziroma fizikalni pogon je najbolje izbrati pri naši implementaciji igre, simulacije ali česa podobnega? V tem

poglavju je moj namen ugotoviti in predstaviti, kateri so tisti pristopi pri detekciji trkov, ki smo jih preučili v prvem delu diplomske naloge in jih zgoraj našteti fizikalni pogoni uporabljajo.

3.1 Bullet game physics

Fizikalni pogon Bullet je odprtokoden sistem za detekcijo trka. Uporablja se v računalniških igrah in v filmih za dodajanje posebnih učinkov. Nekaj primerov uporabe Bullet fizikalnega modela lahko vidimo v naslednjih računalniških igrah:

- Toy Story 3
- Grand Theft Auto 4, 5
- Red Dead Redemption
- Madagascar Kartz

Prav tako je Bullet prisoten tudi v filmski industriji:

- 2012
- Hancock
- Bolt
- The A-Team
- Sherlock Holmes
- Megamind
- Shrek 4

Pri izdelavi diplomske naloge sem prikazal uporabo vseh opisanih knjižnic fizikalnih pogonov in izdelal nekaj testov. Kot sem že omenil v uvodu v to poglavje, je bil povod za raziskovanje 3D igra, ki smo jo izdelali v javi,

pri čemer smo uporabili javansko različico Bullet fizikalnega pogona JBullet. Programski del moje diplomske naloge je prav tako napisan v javi in zaradi tega sem tudi sam uporabil JBullet. Avtorji obeh knjižnic prav tako zagotavljajo, da se algoritmi, uporabljeni v eni in drugi, ne razlikujejo.

Poglejmo si torej, kakšne alogritme uporablja Bullet fizikalni pogon za detekcijo trkov.

3.1.1 Očrtano telo

Tip očrtanega telesa, ki ga uporablja JBullet pogon, je »z osmi poravnan očrtan kvader« oziroma AABB. Kot smo ugotovili v poglavju o očrtanih telesih in podpoglavju o AABB, imamo za predstavitev očrtanega telesa tega tipa 3 možnosti. To so:

- Predstavitev z minimalnimi in maksimalnimi vrednostmi. V 3D prostoru so to $Xmin$, $Xmax$, $Ymin$, $Ymax$, $Zmin$ in $Zmax$.
- Predstavitev s sredinsko točko in polmeri do stranic.
- Predstavitev s kotno točko in vektorji do sosednjih kotov.

JBullet pogon uporablja prvi tip predstavitve. Konstruktor razreda AABB nam inicializira vektorja min in max, ki vsebujeta minimalne in maksimalne vrednosti po vseh oseh.

```
1 public final Vector3f min = new Vector3f();
2 public final Vector3f max = new Vector3f();
```

Konstruktor prejme 3 vektorje $V1$, $V2$ in $V3$ kot argumente. Ti vektorji predstavljajo točke, iz katerih s pomočjo metod $calc_from_triangle$, BT_MAX3 in BT_MIN3 nato izračunamo minimalne in maksimalne vrednosti po vseh treh oseh. Izvorna koda JBullet pogona, ki izvaja zgoraj opisane operacije, izgledajo tako:

```
1 public AABB(Vector3f V1, Vector3f V2, Vector3f V3) {
2   calc_from_triangle(V1, V2, V3);
```

```
3 }
4 public void calc_from_triangle(Vector3f V1, Vector3f V2, Vector3f
     V3) {
5   min.x = BT_MIN3(V1.x, V2.x, V3.x);
6   min.y = BT_MIN3(V1.y, V2.y, V3.y);
7   min.z = BT_MIN3(V1.z, V2.z, V3.z);
8   max.x = BT_MAX3(V1.x, V2.x, V3.x);
9   max.y = BT_MAX3(V1.y, V2.y, V3.y);
10  max.z = BT_MAX3(V1.z, V2.z, V3.z);
11 }
12 public static float BT_MAX3(float a, float b, float c) {
13   return Math.max(a, Math.max(b, c));
14 }
15 public static float BT_MIN3(float a, float b, float c) {
16   return Math.min(a, Math.min(b, c));
17 }
```

Algoritem za ugotavljanje prekrivanja med dvema očrtanima telesoma tipa AABB, ki ga JBullet implementira, je tako imenovani »sweep and prune« algoritem. Več o tem alogritmu si bralec lahko prebere v poglavju o pristopih pri detekciji trkov. Poleg zgoraj omenjenega algoritma sem pri pregledu izvorne kode JBullet pogona odkril tudi zelo »konzervativen« način ugotavljanja prekrivanja med dvema AABB objektoma naenkrat v vseh treh oseh. Preverjanje tega tipa se v JBullet uporablja v primeru BVH drevesa, ko ugotavljamo, ali se očrtana telesa dveh poddreves prekrivata ali ne. Ker je implementacija zelo preprosto, bralcu prepuščam, da si kodo podrobneje ogleda kar sam.

```
1 public static boolean testAabbAgainstAabb2(Vector3f aabbMin1,
     Vector3f aabbMax1, Vector3f aabbMin2, Vector3f aabbMax2) {
2   boolean overlap = true;
3   overlap = (aabbMin1.x > aabbMax2.x || aabbMax1.x < aabbMin2.x) ?
       false : overlap;
```

```
4   overlap = (aabbMin1.z > aabbMax2.z || aabbMax1.z < aabbMin2.z) ?
        false : overlap;
5   overlap = (aabbMin1.y > aabbMax2.y || aabbMax1.y < aabbMin2.y) ?
        false : overlap;
6   return overlap;
7 }
```

3.1.2 Hierarhija očrtanih teles in razdelitev prostora

Glede na navodila za uporabnike Bullet fizikalnega pogona, nam le ta ponuja nekaj tehnik za zmanjševanje števila potrebnih testov zapletene detekcije trkov. Med drugimi sem že omenil očrtana telesa AABB.

Poleg tega pogon pri detekciji trkov uporablja tudi hierarhijo očrtanih teles. Le to smo bolj podrobno spoznali že v poglavju o splošnih pristopih pri detekciji trkov. Glavna ideja je v tem, da zgradimo drevo, ki v listih vsebuje AABB očrtana telesa. Okrog vsake veje oziroma poddrevesa nato očrtamo še dodatno AABB telo in s tem rekurzivno nadaljujemo do korena. Ko imamo tako očrtano telo, lahko preprosto testiramo, ali se AABB dveh vej med seboj prekrivata. Če temu ni tako, se tudi poddrevesa teh vej ne prekrivajo in lahko ustavimo testiranje.

Bullet fizikalni pogon implementira tudi enotske mreže, ki smo jih prav tako že spoznali v enem od prejšnjih razdelkov. Če na kratko obnovimo naše znanje, gre tu za to, da imamo prostor razdeljen z mrežo, ki ima celice enake velikosti. Neko telo lahko pripada eni ali več celic. To je predvsem odvisno od njegove velikosti in trenutne pozicije. Ko imamo tako razdeljen prostor, lahko med seboj testiramo le objekte, ki pripadajo neki celici. Test ponovimo za vse ostale celice.

3.2 Open Dynamics engine

Fizikalni pogon Open Dynamics Engine oziroma ODE je, prav tako kot Bullet, odprtokoden sistem za detekcijo trka in simulacijo dinamike togih teles.

Projekt izdelave knjižnice ODE v $C++$ jeziku se je začel leta 2001. Do sedaj je bil ODE uporabljen v številnih projektih iz sveta zabavne industrije. Nekaj primerov:

- Blood Rayne 2
- Call of Juarez
- S.T.A.L.K.E.R
- Titan Quest
- World of Goo
- X-Moto
- OpenSimulator

Prav tako kot v primeru Bullet pogona, sem tudi tu iz podobnih razlogov uporabil javansko različico pogona. Ta različica se imenuje Open Dynamics Engine for Java oziroma ODE4J. Sam projekt izdelave te knjižnice se je začel leta 2009. Avtorji na uradni strani knjižnice, prav tako kot v primeru JBullet, zagotavljajo, da ni razlik v implementaciji med javansko in $C++$ knjižnico pogona.

3.2.1 Bounding volume

Tako kot pri JBullet fizikalnem pogonu, se tudi v tem primeru uporablja »z osmi poravnan očrtan kvader« oziroma AABB. Tudi tu za predstavitev očrtanega telesa AABB uporabljamo minimalne in maksimalne točke po vseh treh oseh. Vsako telo v prostoru ima svoj pripadajoči AABB. Če nas kadarkoli med simulacijo zanimajo podatki o očrtanem telesu objekta, lahko pokličemo funkcijo dGetGeomAABB, ki nam inicializira AABB podan kot drugi argument.

```
1 public static void dGeomGetAABB (DGeom geom, DAABB aabb) {
2   aabb.set(geom.getAABB());
3 }
```

V implementaciji razreda DAABB torej najprej vidimo deklaracijo in inicializacijo vektorjev oziroma seznamov minimalnih in maksimalnih vrednosti.

```
1   private final DVector3 _min = new DVector3();
2   private final DVector3 _max = new DVector3();
```

V razredu imamo implementirano tudi metodo, ki nastavi vrednosti zgornjima dvema vektorjema, ki predstavljata minimalne in maksimalne točke po oseh. Funkcija setMinMax izgleda tako:

```
1  public void setMinMax(DVector3C v1, DVector3C v2) {
2    if (v1.get0() < v2.get0()) {
3      setMin0( v1.get0() );
4      setMax0( v2.get0() );
5    } else {
6      setMin0( v2.get0() );
7      setMax0( v1.get0() );
8    }
9    if (v1.get1() < v2.get1()) {
10     setMin1( v1.get1() );
11     setMax1( v2.get1() );
12   } else {
13     setMin1( v2.get1() );
14     setMax1( v1.get1() );
15   }
16   if (v1.get2() < v2.get2())
17     setMin2( v1.get2() );
18     setMax2( v2.get2() );
19   } else {
20     setMin2( v2.get2() );
21     setMax2( v1.get2() );
22   }
```

```
23 }
```

Pri ODE4J pogonu poznamo 4 različne strukture prostora, ki jih lahko uporabimo. Na ta način povemo tudi, kako želimo računati kolizijo med očrtanimi telesi. Te strukture so:

- dSimpleSpace
- dQuadTreeSpace
- dHashSpace
- dSAPSpace

Več o tipih prostorov in algoritmih, ki jih uporabljajo, bomo povedali v naslednjem poglavju.

3.2.2 Hierarhija očrtanih teles in razdelitev prostora

Kot sem že omenil, imamo pri ODE4J različne definicije prostora, ki ga inicializiramo ob začetku našega programa. Ti prostori uporabljajo različne tehnike detekcije trkov med telesi. Zaradi tega, se prostori različno dobro odnesejo v različnih situacijah. Tipe definicij prostora sem naštel že v zgornjem poglavju, tu pa si poglejmo, kakšne so njihove značilnosti.

Pri dSimpleSpace definiciji prostora gre za to, da imamo seznam vseh objektov v prostoru. Skozi ta seznam se nato sprehajamo v vgnezdeni zanki in testiramo detekcijo trka po principu vsak z vsakim. Časovna zahtevnost je torej $O(n^2)$, algoritem pa se dobro obnese v situaciji, ko nimamo veliko objektov v prostoru.

Prostor tipa dQuadTreeSpace deluje po principu, kot smo to zapisali v poglavju o osmiških drevesih. Gre za to, da se prostor rekurzivno deli na 4 dele, oziroma 8 delov v primeru osmiškega drevesa, dokler ne pridemo dovolj globoko. Nato detektiramo le trke v posameznem prostoru. Pri detekciji trkov v posameznem prostoru imamo še vedno časovno zahtevnost $O(n^2)$.

Pri prostoru tega tipa je dobro, če so objekti enakomerno porazdeljeni po prostoru, saj imamo tako drevo bolj poravnano.

Pri dHashSpace strukturi imamo hash tabelo vseh AABB očrtanih teles. Ti objekti so nato razdeljeni na več manjših objektov, pri čemer je vsakemu dodeljen prostor v tabeli. V primeru, da imamo objekte, ki pripadajo več AABB očrtanim telesom na isti lokaciji, vemo, da je prišlo do trka.

Tip prostora dSAPSpace uporablja algoritem »sweep and prune« za detekcijo trka. Več o tem algoritmu smo opisali že v poglavju o splošnih pristopih pri detekciji trkov, zato bralca vabim, da si na tistem delu, če misli, da je to potrebno, osveži spomin.

V primeru implementacije tega fizikalnega pogona sem uporabil tip prostora dSAPSpace, ker je algoritem »sweep and prune« tudi največkrat uporabljem algoritem v primeru detekcije trka na prvem nivoju, ko uporabljamo očrtana telesa. Bralca seveda vabim, da preizkusi tudi, kako se obnesejo ostali tipi prostorov oziroma algoritmi.

3.3 Jinngine physics engine

Zadnji fizikalni pogon, ki ga bom poizkušal predstaviti v tej diplomski nalogi, je Jinngine fizikalni pogon. Za razliko od fizikalnih modelov ODE4J in JBullet, je ta fizikalni pogon originalno napisan v programskem jeziku Java, med tem ko sta prejšnja dva prepisana iz $C++$ jezika v Javo. Zadnja verzija Jinngin knjižnice 0.8 je bila izdana 28. maja leta 2010. Trenutno je pogon še vedno v zgodnjih fazah razvoja, saj mu do tega, da bo vseboval vsaj takšno funkcionalnost kot ODE in Bullet, manjka še kar veliko. Zaradi tega sem moral Jinngin v nekaj primerih testov, ki jih bom predstavil v kasnejših poglavjih, tudi izpustiti. Kljub temu, da pogonu manjka še veliko, preden bo dosegel svoj polni potencial, je Jinngin izredno enostaven za uporabo in zato primeren za izdelavo enostavnih 3D iger in učenje o detekciji trkov.

3.3.1 Bounding volume

Ker pri fizikalnem pogonu Jinngine dokumentacije skoraj ni, so glavni viri informacij komentarji v izvorni kodi Jinngine. Bralcu zato predlagam, da si v primeru, da bodo nekoč programerji tega fizikalnega pogona napisali tudi malo bolj obširno dokumentacijo, le to tudi prebere. V tej diplomski nalogi bom Jinngin fizikalni pogon poizkušal predstaviti predvsem s pomočjo izvorne kode.

Jinngin prav tako uporablja »z osmi poravnan očrtan kvader«. Predstavitev, ki jo Jinngin uporablja za AABB, je enaka kot pri ODE4J in JBullet pogonih. To je predstavitev z minimalnimi in maksimalnimi vrednostmi po vseh treh oseh koordinatnega sistema. Vsak razred za neko gemoetrijsko telo, ki ga programerji implementirajo v tej knjižnici, mora vsebovati tudi implementacijo metod, potrebnih za izdelavo »z osmi poravnanega očrtanega kvadra« okoli teh teles.

AABB, ki je očrtan okrog posameznega telesa, je potreben pri poteku tako imenovanega »sweep and prune« algoritma, ki preveri prekrivanje med samimi očrtanimi telesi v prvem delu detekcije trkov. Poleg »sweep and prune« algoritma za detekcijo prekrivanja med AABB telesi, vsebuje Jinngine tudi razred ExhaustiveSearch.java, ki implementira zelo požrešno metodo »overlap« med dvema AABB-jema. Ta metoda za vsak par AABB objektov preveri ali se objekta prekrivata v vseh treh oseh. Če se, potem imamo situacijo, ko se AABB prekrivata. Podobno implementacijo smo opazili že v primeru JBullet pogona. Tam se dotični algoritem uporablja za preverjanje prekrivanja vej v drevesih BVH, tu pa je algoritem sicer na voljo, a se v osnovni konfiguraciji Jinngin pogona ne uporablja. Metoda overlap izgleda tako:

```
1 private static final boolean overlap( BoundingBox i , BoundingBox j
    ) {
2   Vector3 bi = i.getMinBounds();
3   Vector3 ei = i.getMaxBounds();
4   Vector3 bj = j.getMinBounds();
```

```
5   Vector3 ej = j.getMaxBounds();
6   double bix = bi.x; ouble biy = bi.y; double biz = bi.z;
7   double eix = ei.x; double eiy = ei.y; double eiz = ei.z;
8   double bjx = bj.x; double bjy = bj.y; double bjz = bj.z;
9   double ejx = ej.x; double ejy = ej.y; double ejz = ej.z;
10  if( (((bjx < bix) && (bix <= ejx)) || ((bix <= bjx) && (bjx < eix
        ))) &&
11      (((bjy < biy) && (biy <= ejy)) || ((biy <= bjy) && (bjy < eiy
            ))) &&
12      (((bjz < biz) && (biz <= ejz)) || ((biz <= bjz) && (bjz < eiz
            )))) {
13    return true;
14  } else {
15    return false;
16  }
17 }
```

V primeru implementacije Jinngine pogona v moj 3D svet, ki si ga bomo ogledali kasneje, sem uporabil SAP2 algoritem za računanje prekrivanja med AABB telesi. SAP2 je implementacija algoritma »sweep and prune« v pogonu Jinngine. Bralca, ki se bo kasneje tudi sam poigral z napisano kodo, ki pripada tej diplomski nalogi, vabim, da poizkuša uporabiti tudi druge algoritme, ki so implementirani v tem pogonu in ugotovi, kateri mu najbolje služi v dani situaciji.

Poglavje 4

Pregled primerov implementacije fizikalnih pogonov

V tem diplomskem delu sem se odločil, da izdelam primere implementacije treh različnih fizikalnih pogonov in preizkusim njihovo delovanje na praktičnih primerih. Kot sem že omenil v prejšnjem poglavju, sem za to uporabil fizikalne pogone Bullet, ODE in Jinngine. Ker sta fizikalna pogona Bullet in ODE napisana v programskem jeziku $C++$ oziroma C, jaz pa sem pri svojem diplomskem delu uporabil programski jezik Java, sem namesto originalnih verzij uporabil različice, napisane v Javi. Pri javanskih različicah so kljub razliki v jeziku še vedno uporabljeni enaki algoritmi kot v originalih.

Projekt je torej napisan v javi. Pri tem sem poelg knjižnic, ki pripadajo vsakemu fizikalnemu pogonu, posebej uporabil tudi knjižnico lwjgl, ki uporabniku ponuja dostop do zelo zmogljivih knjižnic OpenGL, OpenCL ter OpenAL. Pri svojem delu sem uporabljal OpenGL.

Celoten javanski projekt je razdeljen na več delov. Med programiranjem sem se poizkušal čim bolj držati osnovne oblike in sicer tako, da sem za vsak fizikalni pogon implementiral razred, kjer inicializiram vse potrebno, da pogon lahko normalno deluje. Poleg tega razreda pa sem implementiral

tudi razrede, ki predstavljajo posamezno geomterijsko obliko objekta. Pri vseh implementacijah geometrijskih objektov sem pazil na to, da so metode, ki izvajajo podobne funkcije, poimenovane enako. Razred Objects vsebuje seznam metod, ki so skupne vsem geometrijskim objektom.

```
1  public abstract class Objects {
2    public float speed;
3    public float prevSpeed;
4    public abstract void render(boolean returnSpeed);
5    public abstract void render();
6    public abstract float getSpeed();
7    public abstract float getPrevSpeed();
8    public abstract void setPrevSpeed(float s);
9    public abstract float getSeconds();
10   public abstract float getPositionY();
11 }
```

Na ta način sem si olajšal delo pri implementaciji razreda BaseWindow.java, kjer se nahaja jedro programa s klici vseh potrebnih metod za normalno delovanje programa in inicializacijo knjižnice lwjgl. Poglejmo si torej primere implementacij.

4.1 JBullet

Integracija JBullet pogona v mojem projektu je sestavljena iz več javanskih razredov, ki kličejo potrebne metode, za uporabo pogona. To so:

- BulletDynamics.java
- BulletBox.java
- BulletCylinder.java
- BulletSphere.java
- BulletTrimesh.java

Prvi razred kliče vse metode, potrebne za začetek programa in dodajanje objektov tipa vseh ostalih razredov v svet. Ostali razredi vsebujejo konstruktorje in ostale metode, potrebne za izdelavo ter interakcijo s fizičnim objektom.

4.1.1 Inicializacija pogona in prostora

Pri inicializaciji pogona na začetku pokličemo konstruktor, ki inicializira pogon.

```
1  public BulletDynamics(){
2    collisionShapes = new ObjectArrayList<CollisionShape>();
3    collisionConfiguration = new DefaultCollisionConfiguration();
4    dispatcher = new CollisionDispatcher(collisionConfiguration);
5    overlappingPairCache = new DbvtBroadphase();
6    constraintSolver = new SequentialImpulseConstraintSolver();
7    dynamicsWorld = new DiscreteDynamicsWorld(dispatcher,
         overlappingPairCache,constraintSolver,collisionConfiguration);
8    dynamicsWorld.getSolverInfo().splitImpulse = true;
9    dynamicsWorld.getSolverInfo().numIterations = 20;
10   dynamicsWorld.getDispatchInfo().useContinuous = true;
11   dynamicsWorld.setGravity(new Vector3f(0,-100f,0));
12   dynamicsWorld.getDispatchInfo().allowedCcdPenetration = 0f;
13 }
```

Iz zgornje kode lahko razberemo, da sem uporabil privzeto konfiguracijo za detekcijo trkov. To je uporaba algoritmov za manipulacijo z AABB objekti ter objekti, ki jih AABB očrta. V razredu »DefaultCollisionConfiguration« se nahajajo vsi potrebni algoritmi, kot so »sweep and prune« ter ostali, ki se uporabljajo pri detekciji trka v tem pogonu. Nato nastavimo informacijo uporavljalca omejitev. Število iteracij nastavimo na 20. To je največja dovoljena vrednost in pomeni, kako natančni bomo pri simulaciji omejitev, kot je naprimer združitev dveh objektov v točki. Na koncu je uporabno nastaviti tudi gravitacijsko silo. V mojem primeru sila kaže navzdol po Y osi.

V glavni zanki programa je potrebno nad objektom tipa BulletDynamic vsak obhod zanke uporabiti tudi spodnjo vrstico:

```
1 dynamics.dynamicsWorld.stepSimulation(0.0065f, 1);
```

S prvim argumentom metode povemo, koliko časa ima JBullet, da opravi število korakov, podanih z drugim argumentom.

4.1.2 Inicializacija statičnega objekta

Primeri statičnih objektov v mojih implemntacijah pogonov so tla, kamor padajo objekti. Statični objekt je v tem primeru tisti, ki ne spreminja pozicije v prostoru, objekti pa se od njega odbijajo oziroma na njem pristajajo. Za statični objekt tal uporabimo kar objekt kockaste oblike.

```
1 Vector3f boxHalfExtents = new Vector3f(50000f, 50f, 50000f);
2 CollisionShape floor = new BoxShape(boxHalfExtents);
```

Z vektorjem »boxHalfExtents« določimo dimenzije tal in izdelamo kocko. Nato inicializiramo vektor pozicije, ki je v mojem primeru kar koordinatno izhodišče in izdelamo geometrijsko obliko škatle, ki jo povežemo s telesom tipa »RigidBody«. Več o tem bomo lahko izvedeli v naslednjem poglavju.

```
1 Vector3f startPos = new Vector3f(0f, 0f, 0f);
2 BulletBox box = new BulletBox(mass, 100, startPos, color1, color2,
     color3);
```

Preden telo lahko postavimo v svet, mu moramo še določiti transformacijsko matriko, ki jo postavimo na vrednost identitete ter ji nastavimo začetno pozicijo, ki smo jo že pripravili.

```
1 Transform m = new Transform();
2 m.setIdentity();
3 m.origin.set(startPos);
```

Na koncu moramo objektu nastaviti še informacije o »RigidBody« telesu kot so masa, oblika ter vektor gibanja. Razlika med dinamičnimi in statičnimi objekti v Bullet pogonu je v tem, da ima masa pri statičnih objektih vrednost

0, med tem, ko ima masa pri dinamičnih objektih vrednost več od 0.

```
1 float mass = 0f;
2 DefaultMotionState motionState = new DefaultMotionState(m);
3 Vector3f localInertia = new Vector3f(0, 0, 0);
4 RigidBodyConstructionInfo rbInfo = new RigidBodyConstructionInfo(
     mass, motionState, floor, localInertia);
5 box.body = new RigidBody(rbInfo);
```

Fizikalnemu telesu objekta lahko nastavimo še dušenje, trenje in ostale fizikalne vrednosti ter ga dodamo v dinamični svet. Objekt moramo dodati še na nek seznam teles, skozi katerega se sprehodimo v vsakem obhodu zanke in pokličemo potrebne metode za izrisovanje in interakcijo z njim.

```
1 box.body.setRestitution(basics.BaseWindow.RESTITUTION);
2 dynamicsWorld.addRigidBody(box.body);
3 basics.BaseWindow.objects.add(box);
```

4.1.3 Inicializacija dinamičnega objekta

Pri inicializaciji dinamičnega objekta uporabimo iste metode, kot so opisane v zgornjem poglavju. Razlika je le v tem, da mora imeti tu objekt maso večjo od 0. V zgornjem primeru smo si pogledali korake, potrebne za kreiranje telesa oblike škatle. Namenoma smo izpustili del, ki se nanaša na izdelavo objekta tipa BulletBox, saj si ga bomo pogledali tu.

V tej diplomski nalogi sem implementiral objekte oblik škatle, krogle, cilindra in konveksne oblike. Ker je postopek izdelave pri vseh zelo podoben, si bomo ogledali le razred BulletBox, bralca pa vabim, da si ostale primere ogleda v priloženem javanskem projektu.

V implementaciji razreda BulletBox so najbolj pomembni konstruktor ter metodi *render*() in *create*(). S konstruktorjem nastavimo maso, barvo, dimenzije ter pozicijo objekta. V metodi *create*() nastavimo podatke, ki se nanašajo na samo pozicijo objekta:

```
1 Transform worldTrans = body.getWorldTransform(new Transform());
```

```
2 worldTrans.origin.set(startPosition);
3 worldTrans.setRotation(new Quat4f(0f, 0f, 0f, 1f));
4 body.setWorldTransform(worldTrans);
```

Tu lahko nastavimo tudi podatke o trenju, gravitaciji, dušenju in ostale značilnosti fizikalnega telesa. V metodi *render*(), ki jo pokličemo nad objektom v vsakem obhodu glavne zanke, posodobimo transformacijsko matriko, iz nje zberemo podatke o poziciji ter dimenzijah objekta in izrišemo želeno obliko.

4.1.4 Omejitve oziroma stikanje objektov

V pogonu Bullet je implementiranih več vrst omejitev, ki nam omogočajo stik med objekti. Mednje štejejo:

- »Point to point« omejitev
- »Hinge« omejitev
- »Slider« omejitev
- »Cone twist« omejitev
- Generična omejitev s 6 dimenzijami svobode

V tem diplomskem delu sem hotel na kratko predstaviti tudi to funkcionalnost, ki jo omogoča pogon. Za preprost prikaz stika dveh objektov sem uporabil omejitev tipa »point to point«. To omejitev bi naprimer lahko uporabili pri implementaciji lutke in sicer za stik med glavo in telesom oziroma vrat. Seveda bi morali dodati še nekaj dodatnih omejitev glede rotacije, saj se pri »point to point« omejitvi dve telesi lahko nemoteno vrtita okrog stičiščne točke, pri človeškem telesu pa glave ne moremo zavrteti za več kot 90 stopinj v levo in desno.

Poglejmo si, kateri koraki so potrebni za dodajanje omejitev med dvema objektoma:

```
1 Vector3f pivotA = new Vector3f(extents.x, extents.y, extents.z);
2 Vector3f pivotB = new Vector3f(radius, height/2, 0);
```

```
3 public void addP2PConstraint(Vector3f pivotA, Vector3f pivotB,
     RigidBody bodyA, RigidBody bodyB){
4   Point2PointConstraint p2p = new Point2PointConstraint(bodyA, bodyB
       , pivotA, pivotB);
5   dynamicsWorld.addConstraint(p2p, false);
6 }
```

Na začetku moramo izdelati vektorja, ki predstavljata točko na vsakem objektu posebej, s katero se bo objekt A pripenjal na objekt B in obratno. Ko imamo ta dva vektorja, moramo samo še izdelati novo omejitev tipa »Point2PointConstraint«. Konstruktor sprejme kot argument telo telesa A in B ter prej izdelana vektorja. Na koncu omejitev še dodamo v svet.

4.2 ODE4J

Tako kot pri integraciji pogona JBullet, imamo tudi tu program razdeljen v več javanskih razredov, ki implementirajo potrebne metode za pravilno delovanje našega programa. Ti razredi so:

- ODEDynamics.java
- ODEBox.java
- ODECylinder.java
- ODESphere.java
- ODETrimesh.java
- ODEComplex.java

V razredu, poimenovanem ODEDynamics, implementiramo konstruktor, ki ob klicu pokliče vse metode, potrebne za inicializacijo pogona ODE4J. Pozoren bralec lahko opazi, da sem v tem primeru pogona izdelal tudi razred ODEComplex. Pri objektih tega tipa gre za to, da so sestavljeni iz več geometrijskih oblik naenkrat. Centralna točka vsake geometrijske oblike se

nahaja v sredini objekta tipa ODEComplex, sam objekt pa je sestavljen iz geometrij škatle, cilindra in krogle. Ostali razredi vsebujejo konstruktorje in ostale metode, potrebne za izdelavo ter interakcijo s fizičnim objektom preprostih oblik.

4.2.1 Inicializacija pogona in prostora

Kot smo že zapisali v zgornjem poglavju, se za inicializacijo dinamičnega pogona uporablja konstruktor v razredu ODEDynamics. S klicem tega konstruktorja na začetku inicializiramo pogon:

```
1 OdeHelper.initODE2(0);
```

Nato izdelamo svet, kjer se bodo naši objekti premikali in inicializiramo tudi množico stičišč objektov, ki ob detekciji trka oziroma dotika dveh teles vsebuje stične točke le teh.

```
1 jgroup = OdeHelper.createJointGroup();
2 world = OdeHelper.createWorld();
```

Naslednja vrstica kode je potrebna, ker tu povemo, kakšen algoritem naj pogon uporablja pri detekciji trkov. V mojem projektu sem uporabil dSAP-Space. Kot smo že omenili v poglavju o algoritmih, ki jih posamezni pogon implementira za detekcijo trkov, prostor tega tipa uporablja algoritem »sweep and prune«.

```
1 space = OdeHelper.createSAPSpace(null);
```

Podobno kot pri JBullet pogonu tudi tu nastavimo gravitacijsko silo v svetu, dušenje ter ostale parametre, za katere ne želimo privzetih vrednosti.

```
1 world.setGravity (0,-100,0);
2 world.setLinearDamping(0.00001);
3 world.setAngularDamping(0.005);
```

Pri moji implementaciji konstruktorja ODEDynamics na koncu pokličemo tudi metodo *addGround*(). Le ta kreira statični objekt tal in jih doda v svet. Več o tem bomo povedali v naslednjem poglavju.

Poleg glavnega konstruktorja razreda imamo v tem javanskem razredu implementirano tudi zelo pomembno metodo *step*(), ki jo pokličemo v vsakem obhodu glavne zanke programa. Metoda izgleda tako:

```
1 public void step(){
2   space.collide(null,nearCallback);
3   world.quickStep(0.0065);
4   jgroup.empty();
5   for(int i=0;i<basics.BaseWindow.objects.size();i++){
6     basics.BaseWindow.speedMonitor(basics.BaseWindow.objects.get(i))
        ;
7   }
8 }
```

Kot lahko razberemo iz kode, v tej metodi na začetku pokličemo metodo *collide*(), ki sproži detekcijo trkov med razredi. Nato nastavimo, koliko časa naj traja korak, v katerem pogon opravi vse potrebno za izračun stanj objektov. Pomembno je tudi, da izpraznemo množico stičnih točk ob detekciji trka in pokličemo potrebne metode za izrisovanje in manipulacijo s fizikalnimi objekti.

4.2.2 Inicializacija statičnega objekta

Tudi v tem pogonu bomo za primer statičnega objekta pogledali kar tla. V svet jih dodamo ob klicu konstruktorja ali kasneje s klicom metode *addGround*(). Poglejmo si, kaj se zgodi ob klicu te metode.

Na začetku pripravimo dva vektorja. Prvi bo služil za shranjevanje podatkov o dimenziji kocke oziroma tal, drugi pa bo predstavljal pozicijo telesa. Po tem inicializiramo maso in ji nastavimo obliko telesa, ki bo imelo to maso. V našem primeru je to kocka oziroma »Box«. Zadnji trije argumenti metode *setBox*() predstavljajo dolžine stranic kocke.

```
1       m = OdeHelper.createMass();
2       m.setBox(5.0, sides[0], sides[1], sides[2]);
```

Preden nadaljujem z razlago kode, naj omenim, da imamo pri pogonu ODE4J objekte, sestavljene iz geometrij in fizičnega telesa. Objekt je lahko sestavljen iz več geometrij, pripada pa mu samo eno telo. Tak primer imamo v razredu ODEComplex, kjer je objekt sestavljen iz treh geometrij. Zaradi nekolikšne poenostavitve razumevanja sestave objekta sem uporabil dobro prakso in združil geometrijo in telo v en razred. Enako strukturo boste lahko opazili tudi v primerih, ki se nahajajo ob izvorni kodi pogona.

```
1     static class MyObject {
2       DBody body = null;
3       DGeom geom;
4     }
```

Ko imamo tako strukturo, lahko inicializiramo objekt tipa DGeom, ki predstavlja geometrijo. Geometriji nato nastavimo tudi pozicijo in rotacijsko matriko, ki je v mojem primeru identiteta.

```
1     obj.geom = OdeHelper.createBox(BaseWindow.odeDynamics.space,
          sides[0],sides[1],sides[2]);
2     obj.geom.setPosition(pos.x, pos.y, pos.z);
3     DMatrix3 R = new DMatrix3();
4     R.setIdentity();
5     obj.geom.setRotation(R);
```

Bralec lahko opazi, da ima struktura MyObject spremenljivko *body* postavljeno na vrednost *null*. Takšna implementacija v tem primeru ni naključje. Pri pogonu ODE4J je namreč glavna razlika med statičnim in dinamičnim objektom v tem, da statičnemu objektu ne dodamo fizikalnega telesa. Torej je to telo sestavljeno samo iz geometrije. Metode, ki so potrebne za uspešno kreiranje fizikalnega telesa, ki objekt spremeni v dinamičnega, si bomo zato pogledali v naslednjem poglavju.

4.2.3 Inicializacija dinamičnega objekta

Ker smo večino metod, potrebnih za uspešno kreiranje statičnega objekta pri pogonu ODE4J že razložili v prejšnjem poglavju, bom tu razložil le klice, potrebne za dinamični objekt.

```
1  public ODEBox(Vector3f pos, Vector3f ext){
2    obj.body = OdeHelper.createBody(BaseWindow.odeDynamics.world);
3    DMatrix3 R = new DMatrix3();
4    R.setIdentity();
5    obj.body.setPosition(pos.x,pos.y,pos.z);
6    obj.body.setRotation(R);
7    double[] sides= {ext.x, ext.y, ext.z};
8    m = OdeHelper.createMass();
9    m.setBox(5.0, sides[0], sides[1], sides[2]);
10   obj.geom = OdeHelper.createBox(BaseWindow.odeDynamics.space,
         sides[0],sides[1],sides[2]);
11   obj.geom.setBody(obj.body);
12   obj.body.setMass(m);
13 }
```

Kot lahko opazimo, smo za primer vzeli kar objekt tipa ODEBox. Bralcu predlagam, da si ostale tipe pogleda v sami izvorni kodi diplomske naloge. Torej, v primeru dinamičnega objekta najprej izdelamo fizikalno telo. To telo nato po uspešni inicializaciji geometrije »prilepimo« nanjo s pomočjo metode *setBody*(), telesu pa dodamo tudi podatek o masi. Na tak način izdelamo dinamični objekt.

V tem poglavju bom omenil tudi metodo, ki prav tako pripada implementaciji razreda in je za uporabnika zelo pomembna. To je metoda *render*(), ki izrisuje geometrije in jo moramo poklicati ob vsakem obhodu zanke našega programa.

```
1  public void render(){
2    DVector3C pos = obj.geom.getPosition();
3    DMatrix3C R = obj.geom.getRotation();
```

```
4     DVector3C sides = ((DBox)obj.geom).getLengths();
5     dsDrawBox(pos, R, sides);
6   }
```

Na začetku metode shranimo podatke o poziciji, rotacijski matriki in dimenzijah objekta. Metoda *dsDrawBox*() nam s pomočjo le-teh podatkov nato izriše objekt. Bralca vabim, da si metodo *dsDrawBox*() in metode, ki jih le-ta kliče, ogleda bolj podrobno, saj je razumevanje delovanja tega procesa zelo uporabno pri programiranju računalniške grafike in iger.

4.2.4 Omejitve oziroma stikanje objektov

Tako kot pri JBullet pogonu imamo tudi tu več vrst omejitev. Naj omenim, da je ODE4J pri implementaciji omejitev malo bolj skop in zato v tem pogonu najdemo le:

- »Ball and socket« omejitev
- »Hinge« omejitev
- »Slider« omejitev

Pri tem pogonu sem kot primer implementacije omejitev uporabil »Ball and socket« stik. Ta stik deluje na enak način, kot »point to point« v primeru pogona JBullet. Implementacija v mojem primeru pa izgleda tako:

```
1   public void addP2PConstraint(DBody b1, DBody b2)
2
3   {
4     DBallJoint jointBall = OdeHelper.createBallJoint(world, null);
5     jointBall.attach (b1, b2);
6     jointBall.setAnchor(b1.getPosition().get0()-25, b1.getPosition()
          .get1()-25, b1.getPosition().get2()-25);
7   }
```

Na začetku izdelamo stik tipa DBallJoint in ga dodamo v naš svet. S klicem metode attach() k omejitvi dodamo telesi naših objektov. Na koncu moramo

pogonu povedati še, okrog katere točke se bosta objekta vrtela. To naredimo s klicem metode setAnchor(). Pri pogonu ODE4J je določanje točke, okrog katere se dve telesi vrtita, lahko malo nerodno, saj če želimo, da se točka, ki jo nastavimo z metodo setAnchor, nahaja na objektih, moramo objektom pred dodajanjem omejitev ustrezno nastaviti pozicijo.

4.3 Jinngine

Pri uporabi Jinngine pogona sem imel nekoliko več problemov s samo implementacijo, zaradi skope dokumentacije pogona, ki praktično ne obstaja. Poleg tega je Jinngine eden tistih pogonov, ki ni širše poznan in uporabljen, kot je to v primeru JBullet in ODE4J pogonov. A vendar moramo na tej točki omeniti, da je uporaba te knjižnice zelo enostavna, ko si enkrat pogledamo osnovne primere, podane skupaj z izvorno kodo. Vseeno sem se poizkusil držati strukture, ki sem jo zastavil pri ostalih simulacijah. V to strukturo implementacij razredov spadajo:

- JinngineDynamics.java
- JinngineBox.java
- JinngineSphere.java
- JinngineTrimesh.java

Prvi razred na seznamu služi inicializaciji pogona. Hkrati vsebuje tudi metode za interakcijo in dodajanje novih objektov. Ostali trije razredi predstavljajo vsak svojo geometrijsko obliko.

Hitro lahko opazimo, da v zgornji zbirki razredov manjka implementacija razreda, ki predstavlja geometrijsko obliko valja oziroma cilindra. Kot sem že omenil, je Jinngine še vedno na začetku razvoja. Na pravilnost te trditve lahko sklepamo tudi zato, ker ima najnovejša verzija v času pisanja te diplomske naloge oznako 0.8. Trenutno nam torej ta pogon ponuja le implementacijo geometrijskih oblik Box, Sphere in Trimesh. Ostale oblike pa so, glede na forume o tem pogonu, v fazi izdelave.

4.3.1 Inicializacija pogona in prostora

V tem poglavju si bomo pogledali, kaj se zgodi ob inicializaciji pogona Jinngine. Za pričetek tega procesa moramo poklicati konstruktor JinngineDynamics(), ki nam izdela nov objekt istoimenskega tipa. Ker je koda konstruktorja dolga le nekaj vrstic, si jo bomo v celoti razložili naenkrat.

```
1  public JinngineDynamics(){
2    scene = new DefaultScene(new SAP2(), new
         NonsmoothNonlinearConjugateGradient(44), new
         DefaultDeactivationPolicy());
3    scene.setTimestep(0.0065);
4    addGround();
5  }
```

V prvi vrstici konstruktorja inicializiramo spremenljivko tipa Scene, kjer pogonu povemo, da bomo uporabljali privzeto konfiguracijo s »sweep and prune« algoritmom za detekcijo trkov. Naslednja dva parametra nista dobro razložena v dokumentaciji, a po kodi sodeč bi lahko rekli, da je njuna funkcija preračunavanje sil na dinamične objekte.

Za razliko od prejšnjih dveh pogonov lahko v naslednji vrstici vidimo, da se dolžina časa, ki jo ima na voljo pogon do naslednje zanke, nastavi kar na začetku ob inicializaciji pogona in samo tu.

Zadnji klic v konstruktorju je dodajanje statičnega objekta, ki v našem primeru predstavlja tla. Ker je to že tema naslednjega poglavja, si bomo to pogledali tam.

Poleg konstruktorja imamo v tem razredu implementirano še eno omembe vredno metodo in sicer *step*(). Ob klicu te metode pokličemo metodo *tick*() nad objektom tipa Scene. Klic te metode je pomemben, saj na začetku pokliče metodo *run*() nad objektom, ki mu nastavimo vrednost na $SAP2$ ob klicu iz konstruktorja in tako sprožimo izvajanje izbranega algoritma za detekcijo trkov na nivoju očrtanih teles:

```
1  scene = new DefaultScene(new SAP2(), new
```

```
2 NonsmoothNonlinearConjugateGradient(44), new
     DefaultDeactivationPolicy());
```

4.3.2 Inicializacija statičnega objekta

Pri inicializaciji statičnega objekta si bomo zopet izposodili statični objekt oblike kocka, ki predstavlja tla v našem svetu. Na začetku kreiranja moramo pripraviti dva vektorja. Prvi bo predstavljal pozicijo telesa, drugi pa bo shranjeval dimenzije telesa.

```
1 Vector3 pos = new Vector3(0f, 0f, 0f);
2 Vector3f ext = new Vector3f(10000f, 50f, 10000f);
```

Tako kot pri pogonu ODE4J, imamo tudi tu razdeljen objekt na geometrijo in telo. Pri ODE4J smo statičnemu objektu inicializirali le geometrijo. Temu tu ni tako in zato moramo v vsakem primeru inicializirati obe spremenljivki. Najprej izdelamo geometrijo, ki ji kot argument podamo vse tri komponente vektorja, ki predstavlja dimenzije objekta:

```
1 boxgeom = new Box(extents.x, extents.y, extents.z);
```

Nato moramo izdelati telo, ki mu kot argument podamo neko oznako, ki je javanskega tipa String, ter geometrijo, ki smo jo kreirali. Za tem moramo telesu določiti tudi pozicijo. Pri pregledovanju izvorne kode nisem zaznal funkcije prvega argumenta pri kreiranju telesa.

```
1 body = new Body(identifier, boxgeom);
2 body.setPosition(pos);
```

Sedaj pride na vrsto razlika med statičnim in dinamičnim objektom. Če malo osvežimo spomin iz prejšnjih poglavij, ugotovimo, da smo pri JBullet pogonu morali maso telesa nastaviti na 0, pri ODE4J pa statični objekt ni imel telesa »prilepljenega« na geometrijo. V Jinngine pogonu imamo za to funkcionalnost namenjeno metodo setFixed(), ki jo pokličemo nad telesom in ji kot argument podamo spremenljivko tipa boolean. Ta spremenljivka ima vrednost false, če želimo, da je telo dinamično oziroma true, če želimo

povedati, da je telo statično. Ko smo uspešno poklicali vse zgoraj naštete metode, moramo telo samo še dodati v naš prostor.

```
1 scene.addBody(body);
```

4.3.3 Inicializacija dinamičnega objekta

Kreiranje statičnega objekta se pri pogonu Jinngine ne razlikuje dosti od kreiranja dinamičnega objekta. Edina razlika je v tem, da pri klicu metode setFixed() spremenljivko tipa boolean, ki predstavlja argument funkcije, postavimo na vrednost false. Nad dinamični objekt lahko obesimo tudi gravitacijsko silo. To storimo tako, da v svet »scene« dodamo novo silo tipa GravityForce, ki kot prvi argument v konstruktorju sprejme telo objekta, kot drugi argument pa vektor, ki nakazuje smer in velikost sile.

```
1 Vector3 gravity = new Vector3(0, -10, 0);
2 scene.addForce( new GravityForce(box.body, gravity));
```

Objektu oziroma geometriji ali telesu lahko nastavimo še nekaj drugih značilnosti, kot je recimo »restitution«, ki je neke vrste koeficient odbojnosti.

```
1 box.boxgeom.setRestitution(basics.BaseWindow.RESTITUTION);
```

Poleg tega je pametno objekte, ki jih kreiramo, hraniti v nekem seznamu, skozi katerega iteriramo v vsakem obhodu glavne zanke in izrišemo objekte na novih pozicijah.

4.3.4 Omejitve oziroma stikanje objektov

Kot smo že omenili, je pogon Jinngine trenutno še v fazi razvoja in zato implementira veliko manj funkcionalnosti kot ODE ali Bullet. Med funkcionalnosti pogona, ki trenutno še niso implementirane, štejejo tudi omejtive, ki bi omogočale stike med objekti. Na SVN repozitoriju pogona, ki je trenutno na voljo samo za branje, lahko že opazimo nekaj razredov, ki bodo služili implementacijam omejitev. Na žalost ti razredi še niso vključeni v zadnjo stabilno verzijo pogona.

Poglavje 5

Izvajanje testov nad fizikalnimi pogoni

V zadnjem poglavju tega diplomskega dela si bomo pogledali nekaj testov, ki sem jih izvedel nad zgoraj naštetimi in opisanimi fizikalnimi pogoni. Cilj tega poglavja je ugotoviti, kateri izmed izbranih pogonov je bolj primeren za uporabo pri računalniških igrah, kjer nas bolj kot natančnost zanimata predvsem igralčeva izkušnja ter »gladkost« delovanja pogona, in kateri od fizikalnih pogonov je bolj primeren za izdelavo računalniških simulacij, kjer nam je bolj od izgleda in uporabniške izkušnje pomembna pravilna simulacija realnega sveta, torej natančnost.

Kot primere testov za implementacijo sem izbral test prostega pada, test z velikim številom dinamičnih objektov v prostoru, test odbojnosti objektov glede na spremenljivko vračanja objekta in test hitrosti, pri kateri dinamični objekt prebije steno statičnega. Več o ozadju testov si bomo pogledali v naslednjih poglavjih.

5.1 Test prostega pada

5.1.1 Ideja testa

Pri tem testu bomo preverili, kako natančni so fizikalni pogoni pri računanju pozicije objekta v določenem trenutku, če nanj deluje le ena sila. Smer, v katero kaže sila, bo vzporedna z osjo Y navzdol. Torej bomo simulirali nekakšno gravitacijsko silo v našem svetu. Test bo potekal tako, da bomo dinamični objekt za posamezni pogon kreirali na določeni višini in nanj »obesili« gravitacijsko silo. Po določenem času bomo test ustavili in podatke zapisali v datoteko.

5.1.2 Implementacija testa

Kot je že omenjeno v poglavju »Ideja testa«, bomo na začetku testa kreirali objekt. Test bo potekal za vse implementirane geometrijske oblike. Objekti posameznih oblik bodo v vseh treh pogonih enake velikosti in sicer približno 50 enot v prostoru, ki smo ga izdelali s knjižnico lwjgl v razredu BaseWindow. Ko bo naš program zaznal prvi objekt v svetu, se bo test pričel.

Ob vsakem obhodu zanke bomo nad objektom poklicali metodo $speedTask()$. Poglejmo si implementacijo te metoda za lažjo razlago njene funkcije.

```
1  public static void speedTask(Objects b){
2    b.render(true);
3    if(b.getSpeed() != b.getPrevSpeed()){
4      speeds.add(b.getSpeed());
5      b.setPrevSpeed(b.getSpeed());
6    }
7    if(b.getSeconds() == 50){
8      b = null;
9      objects = new ArrayList<Objects>();
10     writeSpeeds();
11   }
12 }
```

Kot lahko vidimo v zgornjem delu kode, pokličemo metodo $speedTask()$ nad objektom tipa Objects. Ta razred je vmesnik, ki ga implementirajo vse implementacije geometrijskih oblik vseh objektov v vseh pogonih in vsebuje metode, kot je $getPrevSpeed()$ in $getSpeed()$. Implementacij teh metod si ne bomo podrobno pogledali, saj njuni imeni dovolj nazorno predstavita, čemu sta namenjeni. Prva torej vrne hitrost v prejšnjem merjenju, druga pa vrne povprečno hitrost objekta v zadnji sekundi. Pri metodi $speedTask()$ vidimo, da na začetku pokličemo metodo $render()$ nad objektom, s katero izrišemo objekt na trenutni poziciji s trenutno rotacijo. Nato preverimo, ali je trenutna hitrost različna od prejšnje oziroma ali je že pretekla sekunda od zadnjega računanja povprečne hitrosti. Če je temu tako, dodamo povprečno hitrost v tabelo hitrosti in nadaljujemo z izvajanjem testa. Test v trenutni iteraciji se konča, ko preteče 50 sekund. Takrat izpraznemo tabelo objektov, skozi katero se sprehodimo v vsakem obhodu zanke in pokličemo $speedTask()$ nad njimi ter pokličemo metodo $writeSpeeds()$. V tej metodi pripravimo vse potrebno za kreiranje naslednjega objekta na vrsti za testiranje ter ga inicializiramo. Hkrati tudi izračunamo, kakšna so odstopanja merjene hitrosti od hitrosti, ki jo izračunamo po enačbi $v = a \times x$ ter ta podatek skupaj z normiranimi povprečnimi hitrostmi za vsako sekundo zapišemo v razpredelnico.

5.1.3 Težave pri implementaciji

Pri implementaciji tega testa sem naletel na dve večji težavi. Prva težava je v tem, da dokumentacije posameznih pogonov ne povedo kaj dosti o tem, kaj pomeni velikost posamezne komponente vektorja, ki ga podamo kot vektor sile. Težava je v tem, da v primeru, če nastavimo vektor s komponentami $x = 0$,$z = 0$ in $y = 100$, pri tem vemo le to, da se nek objekt, na katerega deluje sila s temi komponentami, ne bo premikal pospešeno v smeri x in z. Kar nas ponavadi pri programiranju računalniških simulacij zanima, pa je, kaj pomeni vrednost 100 pri komponenti y. Na žalost iz nobene od dokumentacij nisem uspel izvedeti, s kakšnim pospeškom v enotah prostora na določeno časovno enoto naj bi se premikal določen objekt z neko silo. Po-

leg tega sem pri hkratnem spustu treh kock iz vseh treh pogonov z enakim vektorjem za gravitacijsko silo opazil, da se njihova hitrost povečuje z zelo različnimi pospeški. Kljub temu je za to težavo obstajala rešitev. To je bila normalizacija podatkov.

Druga težava, ki sem jo opazil šele po prvem naivnem testiranju, je ta, da so podatki lahko zelo različni zaradi vplivov drugih procesov, ki so zagnani na sistemu, kjer so se testi izvajali. To težavo sem deloma odpravil tako, da za posamezen objekt poženem več iteracij testov in na koncu izračunam povprečje odklonov od enačbe $v = a \times x$.

5.1.4 Rezultati testa

Fizikalni pogon/Tip objekta	`Kvader`	`Krogla`	`Trimesh`	`Cilinder`
`JBullet`	23,9	28,9	26,5	25,6
`ODE4J`	42,5	39	43,6	37
`Jinngine`	24,4	28,8	26,4	N/A

Tabela 5.1: Prikaz natančnosti pogonov pri računanju pozicije dinamičnih objektov pri prostem padu.

V zgornji tabeli so prikazani odmiki pogonov od enačbe za prosti pad pri računanju pozicije dinamičnih objektov za zgoraj opisan test. Odstopanje od realne krivulje smo izračunali tako, da smo na začetku normalizirali podatke in njihovo vrednost odšteli od realne vrednosti. Absolutna vrednost dobljenega rezultata je bila nato zapisana v tabelo. Po desetih merjenjih dobimo povprečno vrednost odstopanj od realne krivulje za vsak objekt in pogon posebaj.

Kot lahko vidimo v zgornji tabeli, sta pogona JBullet in Jinngine po seštevku prvih treh tipov objektov skoraj izenačena, pri čemer JBullet nekoliko prednjači. Pogon ODE4J pa se je pri tem testu zelo slabo izkazal.

5.2 Test z velikim številom dinamičnih objektov v prostoru

5.2.1 Ideja testa

Ideja tega testa je v tem, da preverimo, kako »hitri« so zgoraj opisani fizikalni pogoni pri veliki množici objektov v prostoru.

5.2.2 Implementacija testa

Implementacija tega testa ni zelo zapletena. Več časa nam vzame samo poganjanje samega testa. Pri preverjanju, kako se določeni fizikalni pogoni obnesejo v okolju z velikim številom objektov, moramo ob vsakem obhodu glavne zanke poklicati metodo $loopTimeTest(N)$. Ker le-ta zaseda kar precej prostora v smislu vrstic kode, bomo razložili le nekaj delčkov kode, ostalo pa predajam bralcu, da si nariše veliko sliko problema s pomočjo izvorne kode projekta.

V prej omenjeni metodi uporabljamo globalno spremenljivko tipa int, $loopCounter$. Ta spremenljivka nam pove, koliko obhodov zanke je že preteklo odkar smo jo zadnjič ponastavili. Ko je spremenljivka enaka kot parameter, podan v $loopTimeTest(1000)$ metodi, lahko skočimo v glavni del metode. Tu izračunamo čas, ki je pretekel od zadnje ponastavitve $loopCounter$ spremenljivke na 0 in ga delimo s parametrom, podanim v $loopTimeTest$. Na ta način dobimo povprečen čas obhoda zanke za zadnjih N obhodov.

```
1       createObjects(engine, 10);
2       times[objectsCounter] = loopTime;
3       numObjects-=10;
```

Zgoraj so nato našteti naslednji koraki testa. Na tem mestu moramo kreirati naslednjih 10 objektov s pomočjo metode »createObjects«. Ta metoda kot argument prejme ime fizikalnega pogona in število objektov, ki jih želimo kreirati. Nato kreira objekte naključnih, v naprej implementiranih oblik in jih postavi na naključno mesto v prostor. V naslednjem koraku shranimo pov-

prečen čas obhoda zanke v zadnjih N obhodih in zmanjšamo števec objektov za število objektov, ki smo jih kreirali. Na začetku testa smo spremenljivko *numObjects* postavili na vrednost 500.

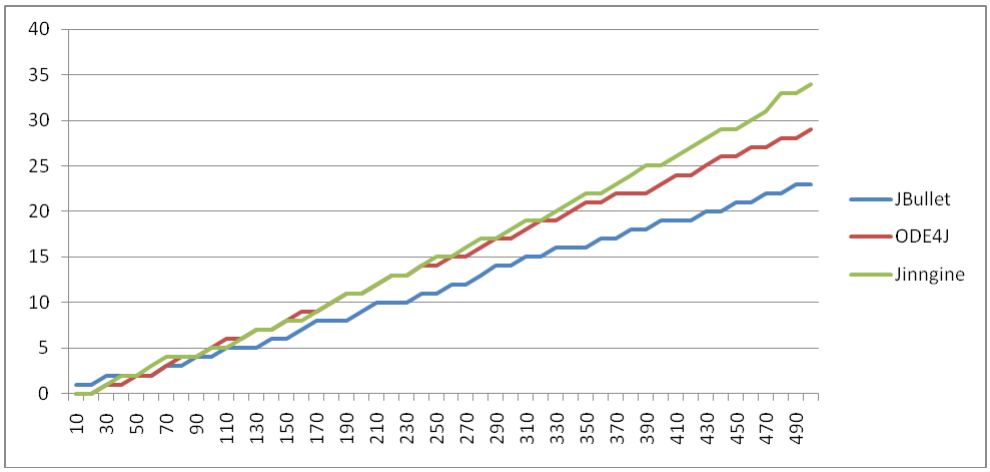
Ko spremenljivka numObjects pade na vrednost 0, oziroma je v okolju, kjer teče naš fizikalni pogon že 500 objektov, lahko test ustavimo. To naredimo tako, da postavimo spremenljivko tipa boolean *isRunning* na vrednost false in preprečimo programu vstop v novo iteracijo.

5.2.3 Težave pri implementaciji

Velikih težav pri implementaciji tega testa nisem imel, saj je v testu potrebno le vsakih N obhodov zanke dodati nekaj novih objektov. Vseeno pa naj omenim, da se lahko test izvaja zelo dolgo časa, saj je veliko odvisno tudi od karakteristik računalnika, na katerem test poganjamo.

5.2.4 Rezultati testa

Spodnji graf nam na enostaven način prikazuje zbrane rezultate zgoraj opisanega testa. Na y osi imamo naveden povprečen čas v milisekundah, obhoda zanke pri poganjanju testa v odvisnosti od količine fizičnih objektov v prostoru.



Slika 5.1: Prikaz časa, potrebnega za obhod zanke v ms (y os) pri določenem številu objektov (x os).

Kot lahko opazimo, se je pri tem testu najbolje odrezal pogon JBullet. Leta je v zadnjem merjenju, ko je bilo v prostoru že 500 objektov, potreboval v povprečju najmanj časa za obhod zanke. Na začetku testa lahko sicer opazimo nekoliko višje vrednosti v primeru tega pogona. Ti začetni časi so najverjetneje posledica inicializacije celotnega programa, saj je bil JBullet pogon prvi v vrsti pri izvdbi testa. Najslabše se je v tem testu odrezal pogon Jinngine, ki je potreboval skoraj 30 milisekund na obhod zanke v zadnji iteraciji programa oziroma testa.

5.3 Test odbojnosti objektov glede na vrednost spremenljivke vračanja

5.3.1 Ideja testa

Ideja tega testa je primerjati, kako dobro se fizikalni pogoni odrežejo pri računanju nekakšnega približka prožnostnega koeficienta. Za potrebe testa, bomo na začetku kreirali prostor s statičnim objektom tal, nad katerimi

bomo kreirali objekte različnih oblik na enaki višini. Nato bomo opazovali odbijanje objektov od tal glede na vrednosti spremenljivke vračanja.

5.3.2 Implementacija testa

Kot je omenjeno v poglavju o ideji testa, bomo na začetku testa kreirali več objektov oblik kocke, krogle in valja na enaki višini. Med ostalimi nastavitvami ob inicializaciji objektov ali pogona pri tem testu ne smemo pozabiti nastaviti vrednosti spremenljivke vračanja, ki jo bomo nato ob vsaki novi inicializaciji določenega pogona manjšali, dokler ne pridemo do vrednosti 0. Spremenljivko vračanja v vsakem pogonu nastavimo, kot kaže spodnja tabela.

`JBullet`	$body.setRestitution(basics.BaseWindow.RESTITUTION);$
`ODE4J`	$contact.surface.bounce = basics.BaseWindow.RESTITUTION;$
`Jinngine`	$Geom.setRestitution(basics.BaseWindow.RESTITUTION);$

Tabela 5.2: Prikaz nastavljanja spremenljivke vračanja za vsak pogon posebej.

Ko spustimo objekte proti tlom, bomo nato vsakih 100 obhodov zanke preverili njihov položaj glede na koordinato Y. Te podatke bomo nato shranili v tabelo in jih na koncu v pravilnem formatu zapisali v datoteko s končnico cvs.

Zgoraj opisani postopek bomo nato ponovili in vrednost spremenljivke vračanja zmanjšali za 0,5. Ko bo vrednost spremenljivke, ki je na začetku nastavljena na 1, dobila vrednost 0, bomo spremenljivko ponovno postavili na 1 in ponovili postopek za vse fizikalne pogone.

5.3.3 Težave pri implementaciji

Prav tako kot pri prejšnjem testu, tudi tu nisem imel večjih težav z implementacijo. Poleg tega je spremenljivka vračanja ena redkih nastavitev, ki

v vseh treh pogonih deluje na enakem intervalu. To je interval $[0, 1]$. Če spremenljivka dosega večjo vrednost od omenjenega intervala, pride do tega, da se objekti od tal odbijajo vedno višje, kar pa ni pogosto v realnem svetu.

Kljub vsemu pa pri tem testu pridemo do težav. Le-te se pojavijo, ko želimo interpretirati rezultate testa, saj zaradi različnih hitrosti padanja, ki so posledica različnih učinkov vrednosti gravitacijske sile na objekte različnih pogonov, iz zgornjih podatkov ne moremo izdelati grafov, ki bi vsebovali podatke različnih pogonov. Vseeno bomo poizkušali interpretirati podatke v taki obliki, kot smo jih zajeli.

5.3.4 Rezultati testa

Rezultati testa so v tem primeru grafi pozicij posameznih objektov glede na navpično os Y v odvisnosti od časa. Pri pregledu spodnjih grafov lahko opazimo, da imajo skoraj vsi pogoni določene pomanjklivosti pri računanju odboja glede na vrednost spremenljivke vračanja.

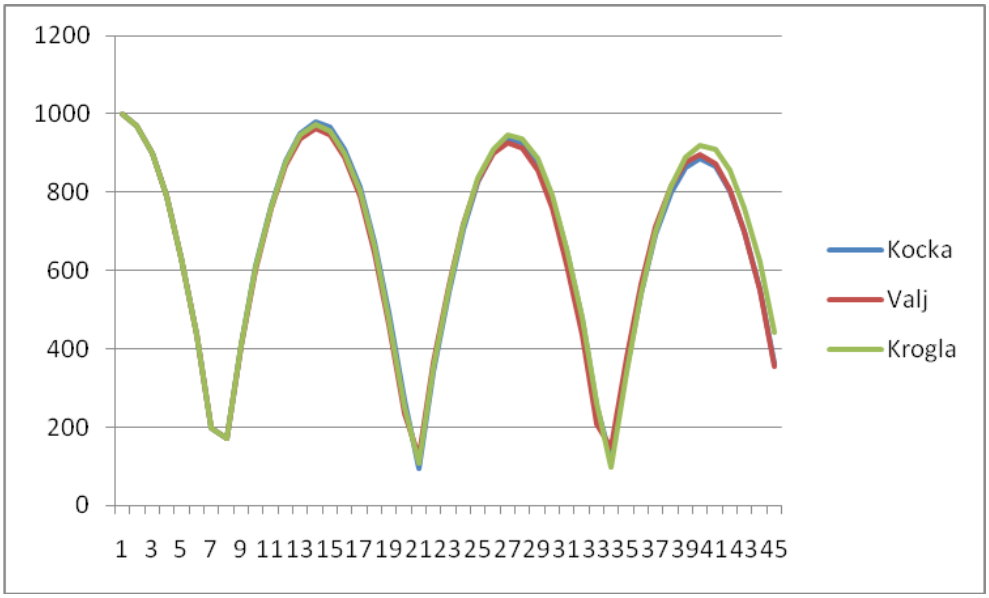
Najbolj izmed vseh zagotovo izstopa pogon Jinngine. Tu izgleda, kot da pri objektu tipa krogla nismo spreminjali spremenljivke vračanja. Temu ni tako in zato lahko ugotovimo, da gre pri tem pogonu najverjetneje za programsko napako, ki onemogoča nastavljanje spremenljivke objektom tipa krogla ali pa ta del implementacije preprosto še ni dokončan.

Pri pogonu ODE4J lahko pri prvem grafu opazimo nekakšen padec višine ob vsakem odskoku objektov. Glede na to, da spremenljivka vračanja tu dosega vrednost 1, bi se morali objekti ob vsakem odskoku vrniti na izhodiščni položaj.

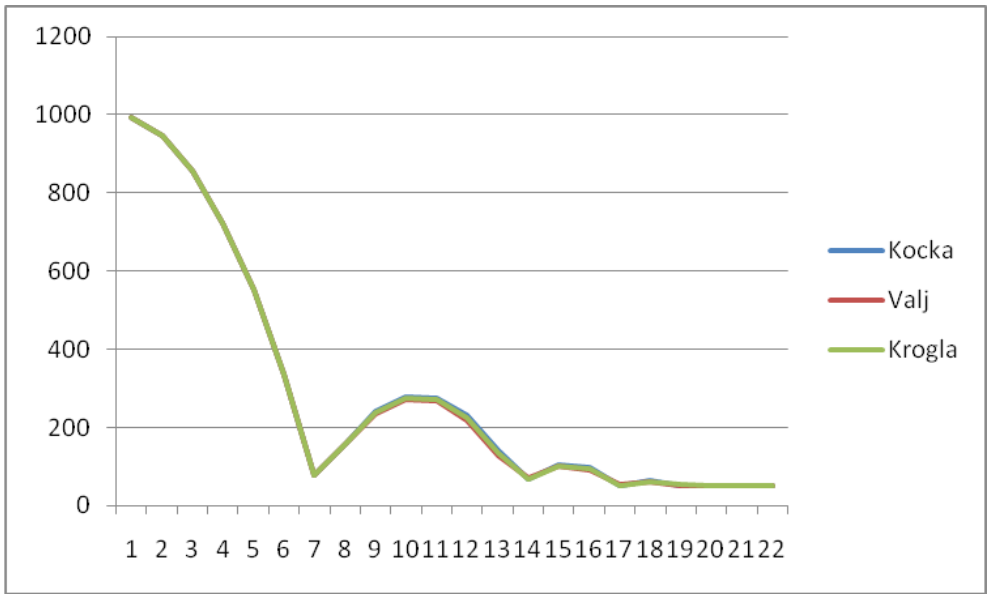
Pri pogonu JBullet pa lahko, nasprotno kot pri ODE4J, v zadnjem odskoku opazimo, da je objekt krogle odskočil višje od izhodišča. Prav tako lahko opazimo, da objekta kocka in valj ne odskakujeta enakomerno. Težava je v tem, da omenjeni telesi ne odskakujeta navpično gor in dol, temveč se premikata tudi levo in desno. Zaradi pomankljivosti dokumentacije JBullet na žalost nisem uspel ugotoviti, ali so ti odskoki v levo in desno stran posledica napake v programu, ali je potrebno le nastaviti še kakšno dodatno

spremenljivko.

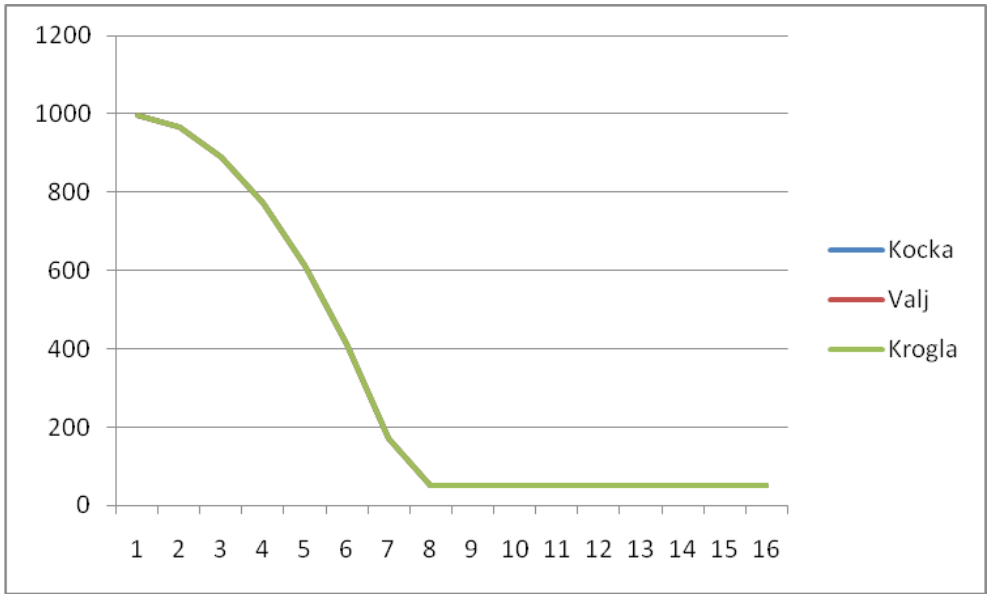
Kljub napaki pri prvem grafu, bi lahko rekli, da se je pogon ODE4J pri tem testu najbolje izkazal in najbolj natančno simuliral odboje od tal. Preden pa poizkušamo delati določene zaključke, se moramo spomniti tudi na poglavje o težavah pri implementaciji. Bralca vabim, da si poglavje še enkrat prebere in premisli o vrednosti ugotovitev pri tem testu.



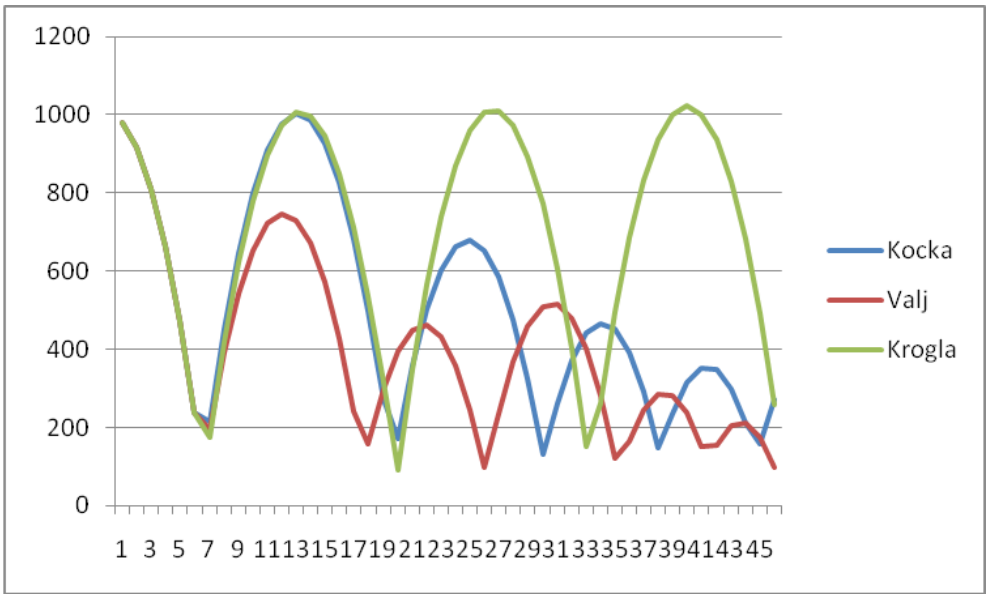
Slika 5.2: Prikaz odboja objektov pogona ODE4J, pri čemer je vrednost spremenljivke vračanja enaka 1.



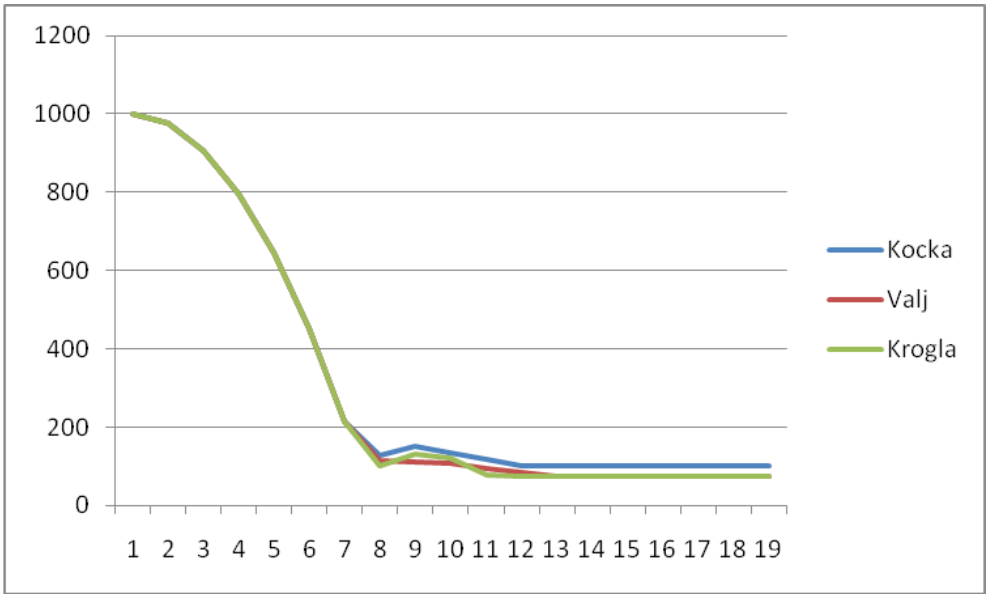
Slika 5.3: Prikaz odboja objektov pogona ODE4J, pri čemer je vrednost spremenljivke vračanja enaka 0,5.



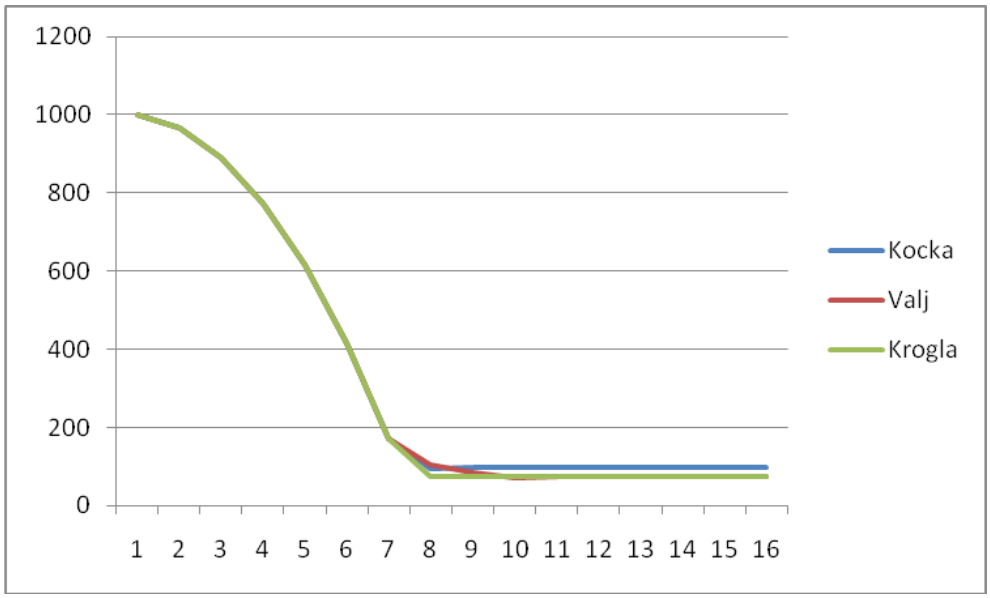
Slika 5.4: Prikaz odboja objektov pogona ODE4J, pri čemer je vrednost spremenljivke vračanja enaka 0.



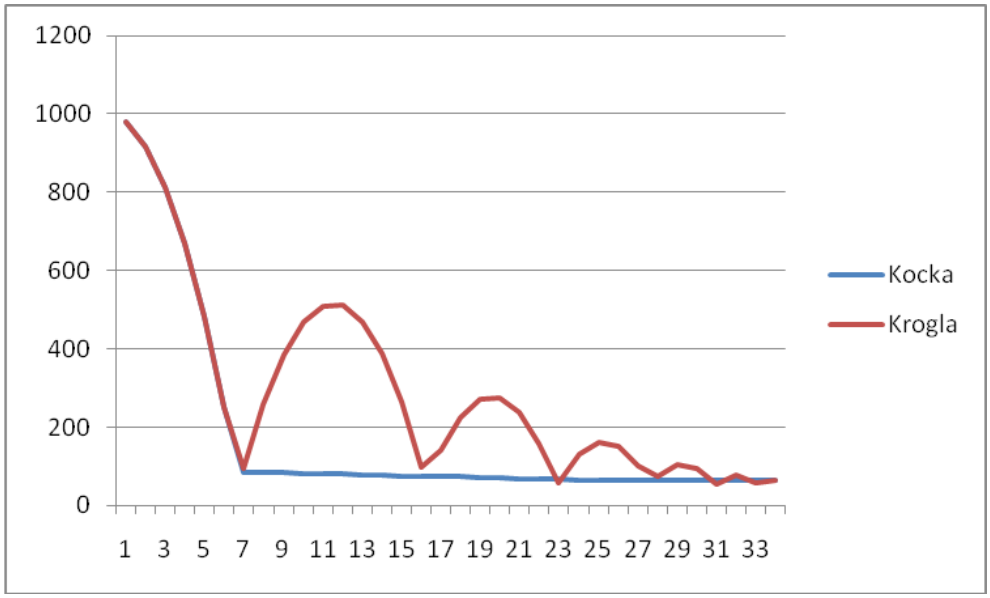
Slika 5.5: Prikaz odboja objektov pogona JBullet, pri čemer je vrednost spremenljivke vračanja enaka 1.



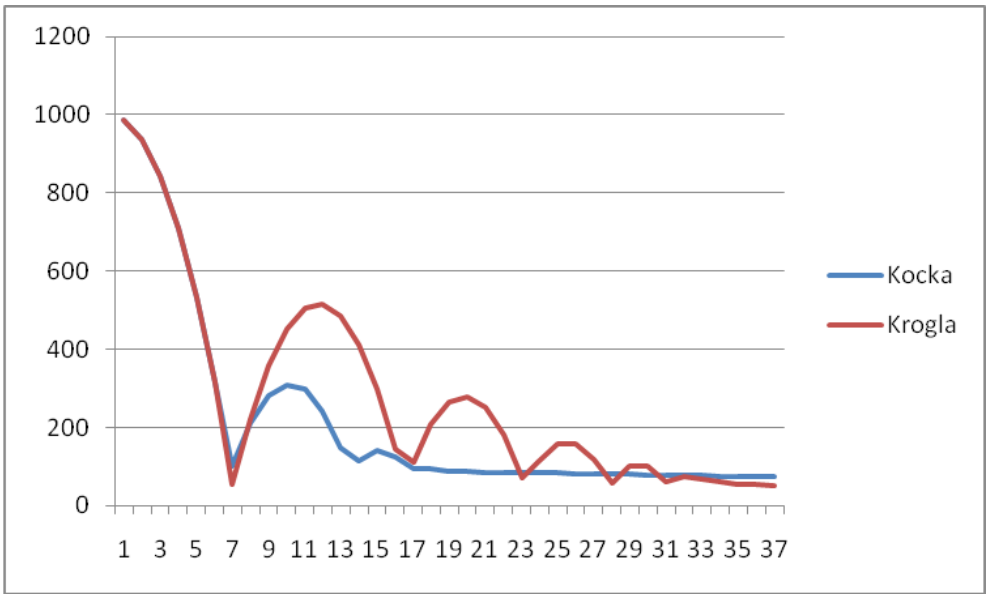
Slika 5.6: Prikaz odboja objektov pogona JBullet, pri čemer je vrednost spremenljivke vračanja enaka 0,5.



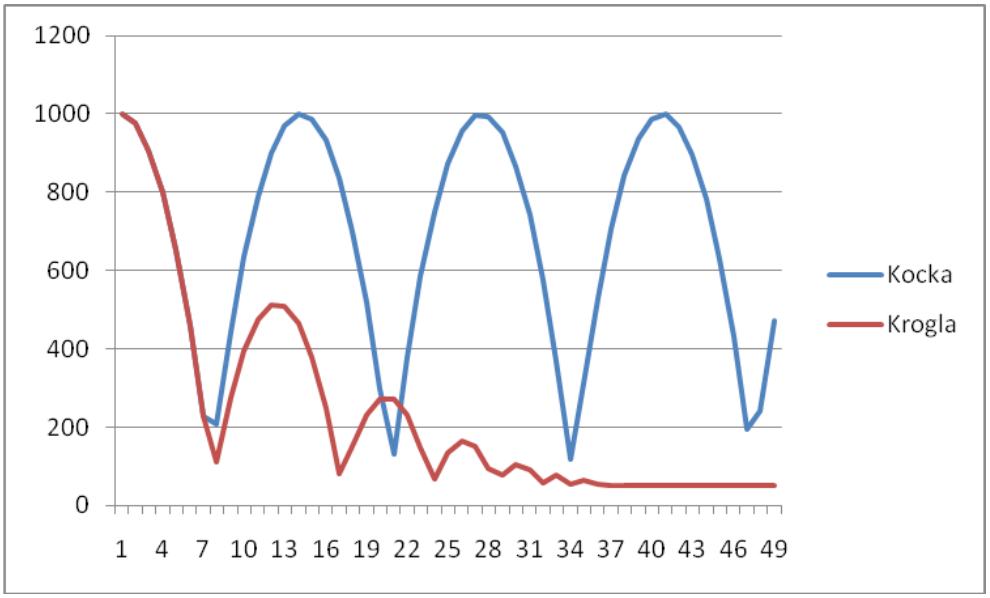
Slika 5.7: Prikaz odboja objektov pogona JBullet, pri čemer je vrednost spremenljivke vračanja enaka 0.



Slika 5.8: Prikaz odboja objektov pogona Jinngine, pri čemer je vrednost spremenljivke vračanja enaka 0.



Slika 5.9: Prikaz odboja objektov pogona Jinngine, pri čemer je vrednost spremenljivke vračanja enaka 0,5.



Slika 5.10: Prikaz odboja objektov pogona Jinngine, pri čemer je vrednost spremenljivke vračanja enaka 1.

5.4 Test hitrosti, potrebne za prehod dinamičnega objekta skozi statičnega

5.4.1 Ideja testa

Pri tem testu si bomo pogledali, pri kateri hitrosti bodo določeni dinamični objekti določenih fizikalnih pogonov prebili steno statičnega objekta. Debelina le tega mora biti v vseh testih enaka in sicer 25 enot. Prav tako bomo poskrbeli, da bodo dinamični objekti enakih geometrij enakih dimenzij.

5.4.2 Implementacija testa

Kot že rečeno, bomo pri tem testu opazovali, kdaj in pri kateri hitrosti bodo naši dinamični objekti prebili statični objekt. Ker ne moremo natančno vedeti, kdaj bodo dinamični objekti dosegli potrebno hitrost, bomo v tem testu statični objekt, ki bo predstavljal tla, dodajali dinamično.

Pri izvajanju tega testa moramo ob vsakem obhodu zanke poklicati metodo *testDrop*(), ob klicu metode za izris objektov pa bomo preverjali tudi njihovo hitrost. Poglejmi si najprej, kaj se dogaja v prvi metodi.

Na začetku spustimo dinamični objekt iz določene višine. To naredimo z naslednjim klicem.

```
1 dropObjOnBox(typeOfEngine.values()[engineNum]+""+typeOfObject.
    values()[objectNum1],
2    15000, x, z, thickness);
```

Pri tem nam prvi argument predstavlja pogon in tip objekta, drugi, tretji in četrti argument pa predstavljajo koordinate dinamičnega objekta. Zadnji argument nam predstavlja debelino statičnega objekta, ki smo ga postavili na vrednost 25 enot. Nato moramo v zanki preverjati, kdaj bo objekt dosegel končno hitrost.

```
1 Objects o1 = objects.get(objects.size()-1);
2 o2 = null;
3 if(o1.getSpeed() >= maxSpeed){
```

```
4   distance = (loopTime1*(maxSpeed*10))+500;
5   if(engine.equals("Bullet")){
6     dynamics.addBox(new Vector3f(x,
7     o1.getPositionY()-distance, z),
8     new Vector3f(2500,thickness,2500), 0);
9   }else if(engine.equals("Jinngine")){
10    jinngineDynamics.addJinngineBox("box", new Vector3(x,
11      o1.getPositionY()-distance, z),
12      new Vector3f(2500,thickness,2500), true);
13  }else{
14    odeDynamics.createBox(new Vector3f(x,
15    o1.getPositionY()-distance, z),
16    new Vector3f(5000, thickness*2, 5000), true);
17  }
18  thrown = true;
19 }
```

Kot lahko vidimo iz zgornjega dela kode, na začetku preverimo, ali je določen objekt dosegel maksimalno dovoljeno hitrost. Če je temu tako, bomo v naslednjem koraku poizkušali predvidevati, na kateri višini se bo znašel objekt v naslednjem obhodu zanke. Ko izračunamo razdaljo od objekta v tem trenutku, do razdalje v naslednjem trenutku, lahko izdelamo statični objekt glede na pogon, ki je trenutno inicializiran.

Ko je dinamični objekt v bližini statičnega, bo program še nekaj časa počakal, nato pa bomo preverili, ali se objekt nahaja pod statičnim ali nad. V prvem primeru je dinamični objekt prebil statičnega, v drugem pa ne. V naslednji iteraciji povišamo maksimalno dovoljeno hitrost in postopek ponovimo.

5.4.3 Težave pri implementaciji

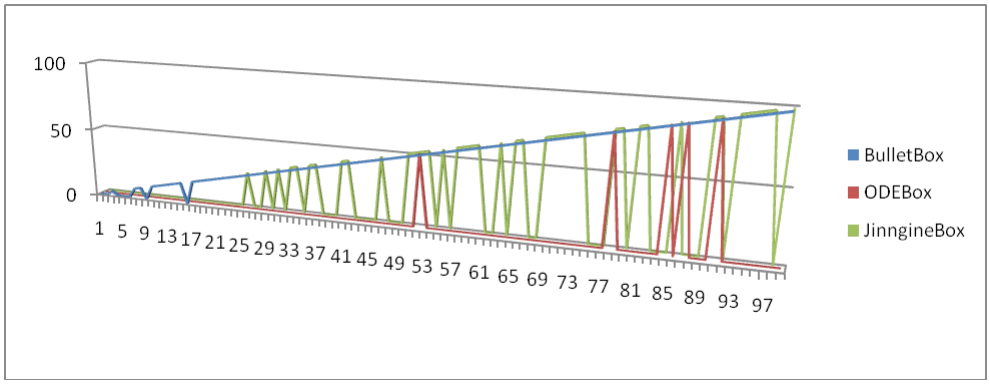
Na začetku pisanja praktičnega dela te diplomske naloge je bila ideja testa ta, da testiram, iz katere višine je potrebno spustiti dinamični objekt, da bo le-ta

prebil statični objekt. Na žalost, kot sem že omenil, je velikost komponente Y, ki predstavlja vektor gravitacijske sile v prostoru, pri vsakem pogonu malo drugačna. Zato sem moral test malce prilagoditi in implementirati merjenje hitrosti objekta.

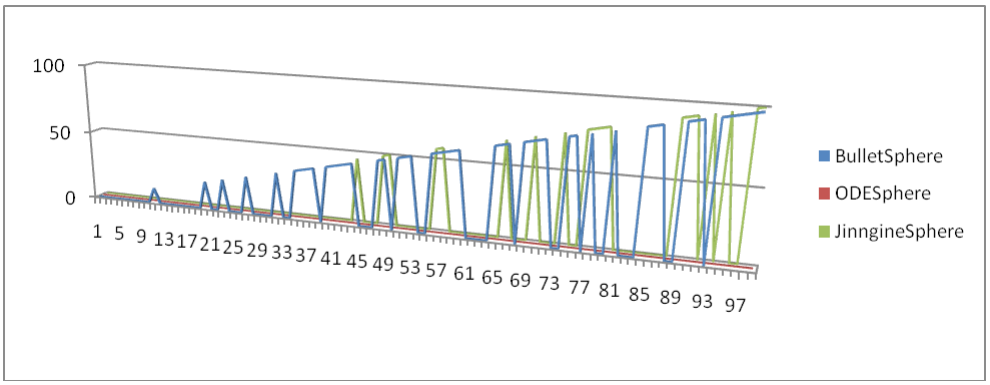
5.4.4 Rezultati testa

Z rezultati zadnjega testa sem imel kar nekaj težav, saj je podatkov zelo veliko in grafi lahko kaj hitro postanejo nekoliko nepregledni. Hitrosti, pri katerih sem meril preboj statičnega objekta, so se gibale na intervalu od 1 do 100 enot na milisekundo.

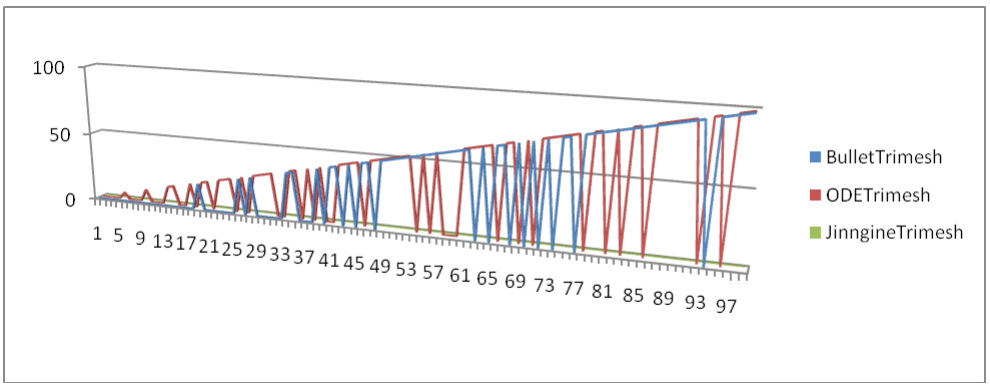
Končni produkt tega testa so štirje grafi. Na vsakem od grafov prikazujemo objekte različnih pogonov in enakih oblik. Preboj statičnega objekta na grafu nakazuje visoko stanje krivulje, ki zasega vrednost, ki je enaka maksimalni hitrosti objekta v tej iteraciji testa.



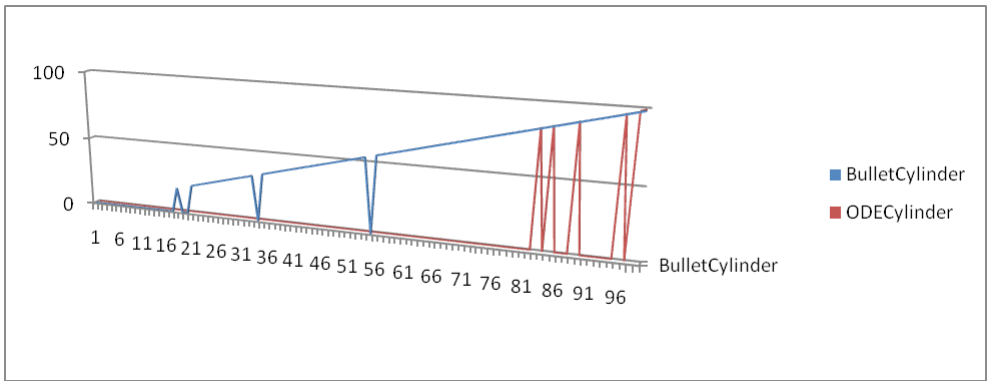
Slika 5.11: Prikaz preboja statičnega objekta z objektom oblike kvadra.



Slika 5.12: Prikaz preboja statičnega objekta z objektom oblike krogle.



Slika 5.13: Prikaz preboja statičnega objekta z objektom, sestavljenim s trikotniki.



Slika 5.14: Prikaz preboja statičnega objekta z objektom oblike cilindra.

Po temeljitem pregledu zgornjih grafov lahko razberemo, da se je v tem testu ponovno najbolje izkazal pogon ODE4J, saj je na krivuljah rdeče barve, ki predstavljajo ta fizikalni pogon, opaziti najmanj visokih stanj. Najslabše se je dotični pogon odrezal v primeru objektov, sestavljenih iz trikotnikov, ki so se po mojih izkušnjah tudi v praksi izkazali za zelo nestabilne in so nemalokrat prebijali stene drugih objektov v prostoru. Najslabše se je pri tem testu izkazal pogon JBullet. Kot lahko razberemo iz modrih krivulj, so objekti tega pogona največkrat prebili stene statičnega objekta.

Poglavje 6

Zaključek

Tema tega diplomskega dela je bilo ugotavljanje zmogljivosti fizikalnih pogonov Jinngine, JBullet in ODE4J. Cilj je bil ugotoviti, kateri od naštetih pogonov se najbolje obnese v določenem primeru uporabe.

Kot sem že omenil v jedru diplomske naloge, se današnji fizikalni pogoni uporabljajo v skoraj vseh računalniških igrah, fizikalnih simulacijah in animiranih filmih. Na eni strani nas pri fizikalnih simulacijah zanima predvsem natančnost pogona, ki ga bomo uporabili, saj so od tega odvisni rezultati naše simulacije. Na drugi strani pa imamo računalniške igre, kjer je bolj od točnosti gibanja objektov pomembno to, kako gladko bo naša igra delovala. Gladkost delovanja pogona je torej odvisna od tega, koliko časa bo pogon porabil za izračun pozicije objektov v naslednjem koraku, kako hiter bo pogon pri ugotavljanju trka med dvema objektoma in koliko računalniških virov bo pogon prepustil tudi drugim procesom, ki delujejo na računalniškem sistemu. V primeru računalniških iger torej ne bomo uporabljali časovno zahtevnih algoritmov, ki bi sicer omogočili natančnost delovanja pogona, a hkrati porabili tudi veliko prostih virov računalniškega sistema in časa.

V tej diplomski nalogi smo torej tekom raziskovanja poizkusili ugotoviti, keteri izmed zgoraj naštetih fizikalnih pogonov bi se bolje obnesel v določenem okolju.

Dela smo se lotili tako, da smo na začetku zgradili projekt, kjer smo

uporabili knjižnice vseh treh pogonov in prikazali primere inicializacij le-teh, ter dinamičnih in statičnih objektov, ki jih pogoni omogočajo. Eden zelo pomembnih argumentov pri ocenjevanju katerekoli knjižnice ali modula je tudi enostavnost implementacije ali uporabe le-tega. Na tej točki naj torej omenim, da je bila implementacija fizikalnega pogona Jinngine, po mojem mnenju, najlažja in hkrati najbolj intuitivna od vseh treh fizikalnih pogonov. Poleg tega je tu potrebno omeniti tudi to, da sem se implementacije tega pogona lotil najkasneje in je morda tudi to razlog, da se mi je zdela uporaba tega pogona najlažja. Hkrati je pogon Jinngine trenutno nekoliko omejen z naborom funkcionalnosti, ki jih ponuja, saj se že pri številu možnih oblik objektov ne more nikakor kosati z ostalima dvema.

V nadaljnjem delu smo nad vsemi pogoni izvedli nekaj enostavnih testov. Točke posameznih testov lahko vidite v spodnji tabeli. Pogon, ki je pri določenem testu prejel najmanj točk, je zmagovalec.

Ime testa	`JBullet`	`ODE4J`	`Jinngine`
Test prostega pada	1	2	1
Test z velikim številom dinamičnih objektov v prostoru	1	2	3
Test odbojnosti objektov glede na vrednost spremenljivke vračanja	2	1	3
Test hitrosti, potrebne za prehod dinamičnega objekta skozi statičnega	3	1	2
`Skupaj:`	7	6	9

Tabela 6.1: Prikaz doseženih točk pri posameznem testu.

Iz zgornje tabele lahko torej razberemo, da je po prej navedenem kriteriju zmagovalec javanska različica pogona ODE. Kljub temu pa teh testov ne moremo upoštevati tako ozkogledno, saj si testi med seboj niso enakovredni. Poleg tega nam rezultati posameznih testov omogočajo ugotavljanje, kateri od pogonov bi bil bolj primeren za uporabo pri računalniških simulacijah in kateri pri računalniških igrah. Kot lahko vidimo, je pogon JBullet zelo dobro opravil s testom z velikim številom dinamičnih objektov. Slabše pa se je odrezal pri zadnjih dveh testih, kjer je potrebna večja natančnost. Zato bi lahko rekli, da bi bil pogon JBullet bolj primeren za uporabo v računalniških igrah. Na drugi strani imamo pogon ODE4J, ki se je zelo dobro odrezal v zadnjih dveh testih, kjer je potrebna večja natančnost. Zato bi, če bi imeli na voljo le te tri fizikalne pogone, tega uporabili v primeru izdelave fizikalnih računalniških simulacij. Kljub temu bi se komercialni pogoni verjetno veliko bolje odrezali v vseh štirih testih od naših odprtokodnih pogonov.

Ob zaključku je potrebno povedati tudi nekaj besed o dokumentaciji, ki je na voljo pri posameznem pogonu. Najbolje od javanskih različic pogonov je dokumentiran pogon JBullet. Poleg tega je bolj obširna dokumentacija napisana tudi za različici pogonov Bullet in ODE, napisani v jeziku $C++$. Kljub drugemu programskemu jeziku sta ti dve dokumentaciji zelo uporabni, saj so imena metod in spremenljivk v večini primerov zelo podobna. Pri pogonu Jinngine pa dokumentacija praktično ne obstaja. To lahko zelo oteži delo programerju. Na srečo so poleg izvorne kode pogona dodani tudi primeri uporabe, s katerimi si lahko pomagamo pri implementaciji.

Kot smo lahko ugotovili, je tema o fizikalnih pogonih in detekciji trkov zelo zanimiva. Poleg tega ponuja še veliko dodatnega gradiva, kjer se bralec lahko bolje poglobi v določene pristope, ki na žalost, zaradi velikega obsega, v to diplomsko delo niso našli poti. Zato bralcu priporočam, da temo raziskuje še naprej in mu pri tem želim veliko veselja.

Literatura

[1] C. Ericson, "Real-Time Collision Detection", Elsevier inc., Združene države Amerike, 2005.

[2] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, "Computational Geometry - Algoritms and Applications", Springer-Verlag, Heidelberg, Berlin, 2008.

[3] D. H. Eberly, "Game Physics", Elsevier inc., Združene države Amerike, 2010.

[4] LWJGL - Lightweight Java Game Library, 2013. Dostopno na: http://lwjgl.org/

[5] JBullet - Java port of Bullet Physics Library, 2013. Dostopno na: http://jbullet.advel.cz/

[6] Bullet Physics Library - Game Physics Simulation, 2013. Dostopno na: http://bulletphysics.org/

[7] ode4j - A Java 3D Physics Engine and Library, 2013. Dostopno na: http://ode4j.sourceforge.net/

[8] Open Dynamics Engine, 2013. Dostopno na: http://www.ode.org/

[9] Jinngine - Real time lightweight 3d physics engine written in Java, 2013. Dostopno na: https://code.google.com/p/jinngine/