UNIVERZA V LJUBLJANI
FAKULTETA ZA RACUNALNIŠTVO IN INFORMATIKO

UNIVERZITETNI ŠTUDIJ
DIPLOMSKO DELO

ALGORITMI ZA ODKRIVANJE TRKOV

ALEŠ DANEU

prof. dr. SAŠA DIVJAK

LJUBLJANA 2002

Povzetek

Diplomsko delo obravnava tematiko odkrivanja trkov med 3D objekti. Delo je organizirano na štiri poglavja. Preko uvoda se hitro pomaknemo na poglavje s krajšim opisom obstojecih algoritmov in se nato preko predstavitve našega algoritma pomaknemo še do prikaza prakticne uporabe nekaterih tehnik opisanih v tem delu.

V uvodu se nahaja predstavitev problema odkrivanja trkov in z njim povezanih problemov. Opisane so osnovne zahteve, ki jih mora izpolnjevati algoritem za odkrivanje trkov ter vzroki za uporabo takih algoritmov.

V drugem poglavju se nahaja kratek opis obstojecih algoritmov za odkrivanje trkov. Opisanih je sedem razlicnih algoritmov, to so hierarhicna drevesa krogel, SOLID, RAPID, QuickCD, SWIFT++, I-COLLIDE in Q-COLLIDE.

V tretjem poglavju je opisan koncni izdelek, naša knjižnica za odkrivanje trkov, poimenovana CD. Opisana je uporabljena podatkovna struktura ter delovanje algoritma (gradnja drevesa, preiskovanje drevesa, testi trikotnik-trikotnik). Za zakljucek poglavja so opisane še zmogljivosti CD.

V cetrtem poglavju se dotaknemo še algoritmov za ugotavljanje potrebne množice testov odkrivanja trkov. Na primeru preproste igrice je prikazana uporaba knjižnice za odkrivanje trkov, algoritma za ugotavljanje potrebnih testov ter odziv objektov na trke.

Za tiste, ki želijo uporabljati knjižnico CD, je zanimiv še dodatek, kjer je opisan vmesnik knjižnice (API).

Kljucne besede: *odkrivanje trkov, hierarhicna drevesa, mejni volumni*

Zahvala

Najprej bi se zahvalil mentorju, prof. dr. Saša Divjaku, za pripombe in napotke pri pisanju diplomskega dela. Zahvalil bi se tudi James Thomasu Klosowskemu (State University of New York at Stony Brook) , ki mi je dovolil vpogled v kodo knjižnice QuickCD. Koncno se zahvaljujem še vsem, ki so preverili delovanje demo programa, ter vsem, ki so mi v casu pisanja diplomskega dela kakorkoli pomagali.

Kazalo

1. UVOD 6

2. PREGLED OBSTOJECIH ALGORITMOV 8

2.1. HIERARHICNA DREVESA KROGEL 8
2.1.1. Gradnja drevesne hierarhije 8
2.1.2. Odkrivanje trkov 9
2.2. SOLID 9
2.2.1. Gradnja drevesne hierarhije 9
2.2.2. Odkrivanje trkov 10
2.2.3. AABB drevesa in deformacije objektov 11
2.3. RAPID 11
2.3.1. Gradnja drevesne hierarhije 11
2.3.2. Odkrivanje trkov 12
2.4. QUICKCD 13
2.4.1. Gradnja drevesne hierarhije 13
2.4.2. Odkrivanje trkov 14
2.4.3. Izboljšave algoritma 16
2.5. SWIFT++ 17
2.5.1. Gradnja drevesne hierarhije 18
2.5.2. Izvajanje poizvedovanj 20
2.6. I-COLLIDE 21
2.6.1. Odkrivanje trkov 22
2.7. Q-COLLIDE 22
2.7.1. Delovanje algoritma 23

3. CD – KNJIŽNICA ZA ODKRIVANJE TRKOV 24

3.1. IZHODIŠCA 24
3.2. PODATKOVNA STRUKTURA 24
3.3. GRADNJA HIERARHICNEGA DREVESA 27
3.3.1. Dodajanje trikotnikov v koren drevesa 27
3.3.2. Gradnja drevesne hierarhije 30
3.4. ODKRIVANJE TRKOV 32
3.4.1. Preiskovanje drevesa 32
3.4.2. Testi trikotnik-trikotnik 34
3.5. REZULTATI 36
3.5.1. Meritve 37
3.5.2. Zakljucki 38

4. UPORABA ALGORITMOV ZA ODKRIVANJE TRKOV 39

4.1. UGOTAVLJANJE POTREBNIH TESTOV 39
4.1.1. Sortiranje osi 39
4.1.2. Mreža 39
4.1.3. Primerjanje intervalov na X, Y in Z osi 40
4.1.4. Upoštevanje clovekove percepcije 41
4.2. ODZIV OBJEKTA NA TRK 42
4.3. PRIMER IGRE 42
4.3.1. Podatkovna struktura 43

4.3.2. Ugotavljanje potrebnih testov 43
4.3.3. Odziv na trke.. 45

LITERATURA 46

DODATEK: CD API 47

1. Uvod

Problemi odkrivanja trkov in njihovih razlicic so izrednega pomena na podrocjih, kot so racunalniška grafika, simulacija fizicnih sistemov, navidezna resnicnost, robotika, molekularno modeliranje, modeliranje predmetov in še na mnogih drugih podrocjih. Odkrivanje trkov je potrebno zaradi dejstva, da si dva objekta ne moreta deliti istega prostora.

Odkrivanje trkov locimo na dva tipa: *pravo odkrivanje trkov* ter *odkrivanje in porocanje o trku.* V drugem primeru imamo nato obicajno še problem *odzivanja na trke*, ki se ga bomo dotaknili v poglavju 4. Vsi algoritmi, ki jih bomo opisali v poglavjih 2 in 3 so tipa odkrivanje in porocanje o trku. Odkrivanje trkov nato delimo še na *staticno odkrivanje trkov, psevdo dinamicno odkrivanje trkov in dinamicno odkrivanje trkov*. Pri staticnem odkrivanju trkov preverjamo dva objekta v doloceni poziciji in orientaciji. Psevdo dinamicno odkrivanje trkov je podobno staticnemu, le da preverjanje poteka na množici diskretnih parov pozicija/orientacija. Dinamicno preverjanje trkov pomeni, da primerjamo volumen, izrezan med premikanjem objektov.

Algoritem za odkrivanje trkov mora biti hiter, natancen in robusten. Hitrost je pomembna zaradi dejstva, da nekateri sistemi zahtevajo tudi do 1000 testov na sekundo. Tak primer je simulacija sestavljanja, zamenjave ali dostopa do dolocenega dela v prostoru, ki je poln raznih ovir. Natancnost je potrebna za pravilni odziv objektov na trk. Problem, ki zahteva natancnost, je nacrtovanje gibanja robota v prostoru, kjer želimo doseci, da se robotova roka izogne raznim oviram. Robustnost je potrebna zaradi napak na modelih objektov. Želimo si, da algoritem deluje pravilno tudi v takih primerih.

Objekti so sestavljeni iz nekaj tisoc trikotnikov. Ce bi testirali vse trikotnike dveh objektov (vzemimo, da je prvi sestavljen iz 100.000 trikotnikov, drugi pa iz 20.000 trikotnikov), bi to pomenilo, da bi morali narediti 2.000.000.000 testov. V [1] avtor navaja, da bi to pri njihovi implementaciji testa trikotnik-trikotnik, trajalo skoraj dve uri, kar je dalec od zahtev realno casovnih sistemov. V praksi so se uveljavili algoritmi, ki temeljijo na uporabi neke oblike mejnih volumnov v kombinaciji z hierarhicnim drevesom. Obstajajo tudi algoritmi, ki temeljijo na dekompoziciji objekta na manjše konveksne kose ter

grupiranju teh v drevo. Drevo je tako sestavljeno iz konveksnih kosov, pri cemer je koren drevesa osnovni objekt. Ta algoritem je kombinacija drevesne hierarhije in hitrih algoritmov za odkrivanje trkov med dvema konveksnima objektoma. Vec o tem v poglavju 2.

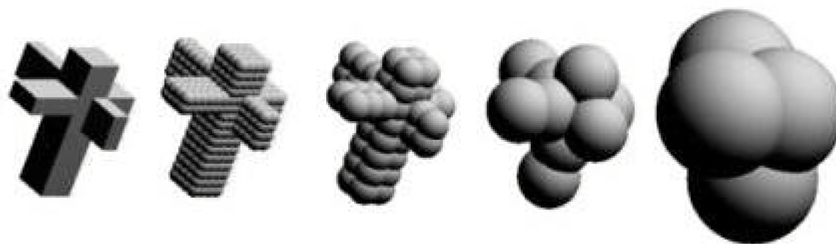
Namen diplomskega dela je narediti knjižnico za odkrivanje trkov primerno za uporabo v igrah, vendar uporaba na drugih podrocjih ni izkljucena. Opisu algoritma in njegove uporabe sta namenjeni poglavji 3 in 4.

2. Pregled obstojecih algoritmov

Algoritmov za odkrivanje trkov je veliko. Locimo jih v dve skupini. Tiste, ki so namenjeni za odkrivanje trkov samo med konveksnimi objekti in tiste, ki niso omejeni na konveksne objekte. Do razlik prihaja tudi pri tipu uporabljenega mejnega volumna.V tem poglavju je opisanih nekaj algoritmov iz obeh omenjenih skupin.

2.1. Hierarhicna drevesa krogel

Krogle se v racunalniški grafiki pogosto uporabljajo kot približek objekta. Eden izmed razlogov za to je zelo preprost test prekrivanja, drugi še pomembnejši razlog pa je, da so neodvisne od rotacije. Zaradi te lastnosti je vse kar je potrebno narediti med primerjanjem dveh vozlišc v drevesu, translacija in rotacija ene tocke - centra. O tem tipu hierarhicnih dreves in delovanju algoritma je nekoliko vec napisanega v [4].



Slika 1 : Primer drevesne hierarhije krogel (povzeto po [4]).

2.1.1. Gradnja drevesne hierarhije

Uporabljeno je drevo v katerem ima lahko vsako vozlišce najvec osem sinov. Drevo se zgradi z rekurzivnim deljenjem. Pri ugotavljanju velikosti krogle, ki omejuje objekt, se najprej izracuna najmanjša kocka, ki omejuje objekt. Ta kocka je nato razdeljena na osem enako velikih kock. Ce taka kocka vsebuje kakšen del objekta, je vkljucena v drevo. Rekurzivno deljenje se nadaljuje dokler vozlišce ne vsebuje le enega trikotnika. Iz tako dobljenih kock se izracunajo krogle. V zadnjem koraku se odstrani še vse krogle, ki so z vseh strani zakrite z drugimi kroglami – ne vsebujejo trikotnikov z zunanje strani objekta.

2.1.2. Odkrivanje trkov

Algoritem zacne s korenoma dreves. Ce se krogli ne prekrivata potem ni trka. V nasprotnem primeru se koren prvega drevesa primerja s sinovi korena drugega drevesa. Po istem postopku se drevo preiskuje v notranjih vozlišcih. Ce po takem preiskovanju dreves pridemo do listov dreves se naredi test trikotnik-trikotnik.

2.2. SOLID

SOLID je eden preprostejših predstavnikov algoritmov, ki uporabljajo hierarhicna drevesa neke oblike mejnih volumnov. V tem primeru je mejni volumen kvader, katerega stranice so poravnane z X, Y in Z osjo objektovega koordinatnega sistema (AABB ali *axis aligned bounding boxes*). Prednost tega algoritma je hitrost izgradnje drevesa, slabost pa sama hitrost odkrivanja trkov. V [2] je navedeno, da je tak algoritem primeren za deformabilne objekte zaradi hitrosti, s katero je drevo možno popraviti, da predstavlja novo obliko.

2.2.1. Gradnja drevesne hierarhije

AABB drevo je binarno. Vozlišce drevesa je predstavljeno s 6 skalarnimi vrednostmi (tri za pozicijo in tri za dolžino stranic), kar je manj v primerjavi z OBB in k-dop mejnimi volumni. Drevo je zgrajeno od zgoraj navzdol z rekurzivnim deljenjem. Na vsakem koraku deljenja se izracuna najmanjši AABB dane množice trikotnikov, nato se množica razdeli na dva dela, glede na dobro izbrano delilno ravnino. Proces deljenja se konca, ko ima vsaka podmnožica le po en element. Za delilno ravnino je izbrana ravnina, ki je ortogonalna na os, kjer je dani AABB najdaljši. Ko je delilna ravnina izbrana, je potrebno izbrati tocko, kjer delilna ravnina seka izbrano os. Trikotniki so potem razdeljeni v dve novi podmnožici glede na stran ravnine, na kateri se nahajajo. Za izbiro tocke, kjer delilna ravnina seka izbrano os je vec možnosti:

(a) Delitev množice trikotnikov na dve (skoraj) enako veliki množici, kar ima za posledico optimalno uravnovešeno drevo.

(b) Delitev na dve podmnožici tako, da je vecja najvec dvakrat vecja od manjše in da je prekrivanje projekcij novih AABB-jev na najdaljšo os minimizirano.

(c) Enostavna delitev po polovici najdaljše osi.

Po navedbah v [2] daje zadnja možnost (enostavna delitev po polovici najdaljše osi) najboljše rezultate. V primeru, ko se zgodi, da vsi trikotniki spadajo na eno stran delilne ravnine, je uporabljeno deljenje množice na dve množici (skoraj) enake velikosti.

V [2] avtor navaja da je gradnja AABB drevesa približno trikrat hitrejša od gradnje OBB drevesa.

2.2.2. Odkrivanje trkov

Odkrivanje trkov poteka rekurzivno s testiranjem parov vozlišc. Za vsak obiskan par vozlišc je preverjeno, ali se AABB-ja prekrivata. Naprej se raziskuje le vozlišca katerih AABB-ji se prekrivajo. Ko sta obe vozlišci lista pripadajocih dreves, se preveri, ali se trikotnika sekata. V primeru, da je eno vozlišce list, drugo pa notranje vozlišce, se list primerja s sinovi notranjega vozlišca. V primeru, da sta obe vozlišci notranji, se primerja vozlišce z manjšim volumnom s sinovi vozlišca z vecjim volumnom. Ta zadnja hevristika ima za posledico najvecje zmanjšanje skupnega volumna v sledecih AABB primerjanjih, torej najmanjšo verjetnost, da se sledeci AABB-ji prekrivajo.

Ker so lokalni koordinatni sistemi para modelov, ki jih preverjamo, lahko poljubno orientirani in postavljeni, potrebujemo test prekrivanja za relativno orientirane in postavljene AABB-je. Ta test se imenuje *separating axis test* ali krajše SAT in je opisan v [3]. Pri tem testu išcemo os na kateri se projekciji AABB-jev na os ne prekrivajo. Obstoj take osi ima za posledico, da se AABB-ja ne prekrivata. Ce take osi ne najdemo, potem se prekrivata.

V splošnem zahteva AABB algoritem vec testov prekrivanja mejnih volumnov kot OBB algoritem, vendar to slabost nekoliko nadoknadi z manj racunskimi operacijami pri takem preverjanju.

2.2.3. AABB drevesa in deformacije objektov

AABB drevesa so zaradi hitrosti pri prilagajanju drevesa primerni za uporabo v primeru, ko imamo opravka z deformabilnimi objekti. Potem, ko se objekt deformira, je hitreje prilagoditi AABB-je v drevesu, kot pa na novo graditi drevo. Prilagajanje drevesa se zacne v listih, torej se najprej izracuna AABB za lista, nato pa se iz AABB-jev sinov ponovno izracuna AABB starša, in tako naprej proti vrhu drevesa. Prilagajanje drevesa pa ima tudi slabosti. Zaradi sprememb v relativni poziciji trikotnikov objekta po deformaciji imajo lahko AABB-ji v obnovljenem drevesu vecjo stopnjo prekrivanja kot v znova zgrajenem drevesu. Vendar pa to pride bolj do izraza pri radikalnih deformacijah, kot so veliki prevoji, velike povecave dolocenih delov ter ekstremni primeri sekanja samega sebe. Pri drugih, bolj »normalnih« oblikah deformacije, ni vecjega zmanjšanja hitrosti algoritma.

2.3. RAPID

RAPID algoritem uporablja hierarhicno drevo, ki je sestavljeno iz kvadrov, ki so orientirani tako, da se kar najbolje prilegajo modelu (OBB ali *oriented bounding boxes*). Kljub temu, da so kvadri orientirani je test prekrivanja dveh kvadrov z uporabo SAT koncan v, kot je navedeno v [3], povprecno 100 operacijah, kar je približno enkrat hitreje od prejšnjih algoritmov, ki so uporabljali OBB-je. OBB drevesa se zelo izkažejo v veliko primerih, kjer sta si modela zelo blizu (*close proximity*). V primerjavi z drevesi, ki za mejni volumen uporabljajo AABB ali krogle, je OBB asimptoticno veliko hitrejši.

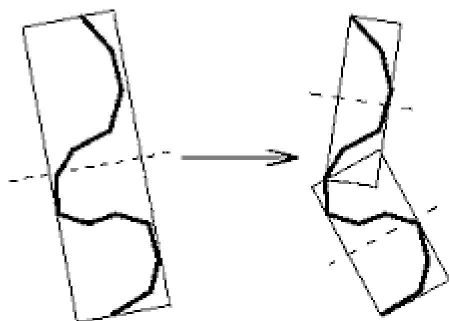
2.3.1. Gradnja drevesne hierarhije

Gradnja OBB drevesa poteka podobno kot gradnja drevesa v SOLID algoritmu s pristopom od zgoraj navzdol. Na vsakem koraku je izracunan kvader v neki orientaciji, ki se najbolje prilega množici trikotnikov. Algoritem za izracun se opira na uporabo srednje vrednosti in kovariancne matrike C, ki jih izracuna iz koordinat tock. Lastni vektorji simetricne matrike (C je simetricna matrika), so

medsebojno ortogonalni. Po normalizaciji so uporabljeni kot bazni vektorji za izracun velikosti OBB-ja.

Slabost takega pristopa je, da tocke v notranjosti modela, ki ne bi smele imeti vpliva na izbiro orientacije OBB, vplivajo na lastne vektorje. Primer takega ucinka je majhna ploskev v notranjosti modela z zelo gosto množico tock, ki povzroci, da se OBB poravna s to ploskvijo. Algoritem je možno izboljšati z uporabo konveksne lupine (konveksna lupina je najmanjša konveksna množica, ki vsebuje vse tocke - algoritmi za izracun konveksne lupine so opisani v [8]), vendar opisana slabost ostaja. Rešitev tega problema je cimbolj gosto vzorcenje konveksne lupine, in izracun srednje vrednosti in kovariancne matrike C iz vzorcenih tock. Vec o tem je napisano v [3].

Deljenje vozlišca poteka podobno kot pri SOLID algoritmu. Uporablja se delilna ravnina, ki je ortogonalna na os, kjer je OBB najdaljši. Razlika v primerjavi s SOLID algoritmom je le v tem, da se tukaj uporabljajo osi s katerimi je poravnan OBB. Tocka ob izbrani osi, ki se uporablja za mesto deljenja je srednja vrednost vseh tock. Ce delitev na najdaljši osi ni možna se izbere drugo najdaljšo os, ce pa tudi to ni možno, se uporabi najkrajšo os.

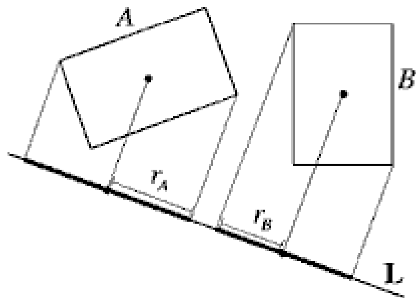


Slika 2 : Deljenje vecjega OBB-ja na dva manjša (povzeto po [3]).

2.3.2. Odkrivanje trkov

Ob danih dveh OBB drevesih algoritem za odkrivanje trkov porabi vecino casa za preverjanje, ali se dva OBB-ja prekrivata. Pri testu prekrivanja dveh OBB-jev se uporablja projiciranje OBB-jev na 15 osi (tri za prvi kvader, tri za drugi kvader in devet za vse možne kombinacije robov obeh kvadrov). Na os se projicira center OBB-ja. Nato se izracuna radij intervala za vsak OBB. Ce je razdalja med projiciranima centroma OBB-jev vecja od vsote obeh radijev, potem

se projekciji OBB-jev na to os ne prekrivajo in nadaljnje preverjanje ni potrebno – OBB-ja se ne prekrivata. OBB-ja se prekrivata v primeru, ko se njune projekcije na vseh 15 oseh prekrivajo. Ko sta obe vozlišci lista OBB drevesa se izvede test trikotnik-trikotnik.



Slika 3 : Projekcija OBB-jev na os L. Iz projekcij je razvidno, da se na osi L OBB-ja ne prekrivata. Povzeto po [3].

2.4. QuickCD

Quick CD je algoritem, ki za mejni volumen uporablja konveksne politope, pri katerih so normale stranic dolocene z majhno množico vnaprej dolocenih *k* orientacij. V [1] avtor take politope imenuje *discrete orientation polytopes* ali krajše k-dop. V knjižnici so implementirani algoritmi za 6-dop (to je v bistvu AABB), 14-dop, 18-dop in 26-dop. Ta algoritem je zasnovan s predpostavko, da imamo en staticen objekt (okolje), ki ga primerjamo z nekim premikajocim objektom. Vec o nekaterih detajlih tega algoritma je napisano v poglavju 3, saj je bil izbran za osnovo našega algoritma za odkrivanje trkov.

2.4.1. Gradnja drevesne hierarhije

Drevo je, podobno kot pri vecini algoritmov, binarno in se gradi s pristopom od zgoraj navzdol. Za mejni volumen se uporablja enega izmed zgoraj naštetih k-dopov. V [1] je navedeno, da najboljše rezultate dajejo 18-dop mejni volumni. Deljenje volumnov na manjše se vedno vrši na X, Y ali Z osi, kljub temu da imamo pri 14-dop, 18-dop in 26-dopih na voljo vec osi. Za izbiro osi so na voljo naslednji kriteriji:

(a) Minimalna vsota volumnov (izbere os, pri kateri je vsota volumnov dobljenih sinov najmanjša).

(b) Minmax (izbere tako os, da je volumen vecjega od sinov cim manjši).
(c) Splatter metoda (izracuna se varianca projekcij centroidov na vse tri osi; izbere se os, ki ima najvecjo varianco).
(d) Najdaljša os (izbere os na kateri je k-dop najdaljši).

V algoritmu je uporabljena splatter metoda, ki sicer daje nekoliko slabše rezultate pri odkrivanju trkov v primerjavi z minimalno vsoto volumnov, vendar pa je precej hitrejša pri gradnji drevesa. Naslednji potreben podatek pri gradnji drevesa je tocka deljenja. Pri nacrtovanju algoritma so se avtorji odlocali med srednjo vrednostjo (*mean*) in mediano (*median*). V [1] je avtor s poskusi pokazal, da z uporabo srednje vrednosti vedno dobimo drevo z manjšim skupnim volumnom. V nekaterih primerih je razlika tudi 27%. Prednost mediane je bolj uravnoteženo drevo, vendar je tudi tu avtor pokazal da se višina drevesa z uporabo srednje vrednosti ne poveca bistveno. Drevo se preneha graditi, ko je v listih samo en trikotnik (za staticni objekt) oziroma pri neki mejni vrednosti števila trikotnikov ? za premikajoci objekt. Izbira vrednosti ? je odvisna od velikosti objekta. Za objekte z manj kot 1.000 trikotniki se uporablja vrednost 20, ne glede na vrednost *k*. Za vecje objekte pa se uporabljajo mejne vrednosti 40 (za 6-dop in 14-dop), 60 za 18-dop in 100 za 26-dop.

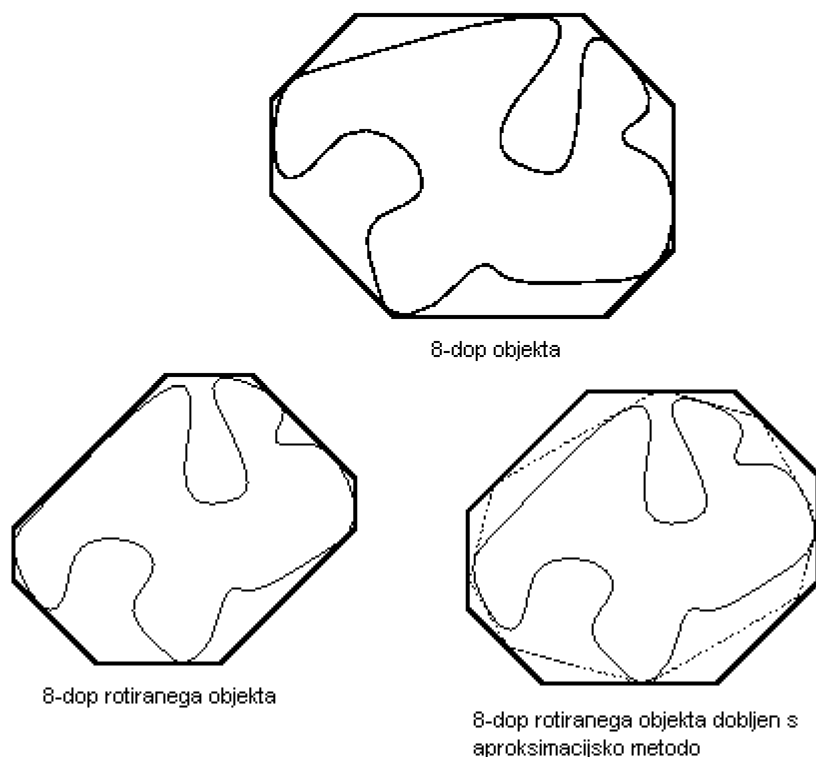
2.4.2. Odkrivanje trkov

Pozicija in rotacija premikajocega objekta, ki ga primerjamo s staticnim okoljem, se s casom spreminja. Zato je potrebno pred preverjanjem, ali se dva k-dopa prekrivata, ponovno izracunati k-dop v trenutnem vozlišcu drevesa premikajocega objekta. Ponovna izgradnja drevesa ne pride v poštev, ker je to prevec potratna casovna operacija. Ravno tako ne pride v poštev racunanje novega k-dopa iz trikotnikov v trenutnem vozlišcu (ponovno zaradi casa in ker bi morali v vozlišcu shraniti še vse tocke, ki jih vsebuje to vozlišce). Zato sta na voljo dve možnosti:

(a) *Hill climbing* algoritem, ki shrani konveksno lupino objekta in jo uporabi za lokalno optimizacijo (na vsaki izmed *k/2* smeri preveri, ali je prejšnja maksimalna (minimalna) tocka še vedno maksimalna (minimalna)). Tako dobimo nov najmanjši k-dop.

(b) Aproksimacijska metoda poskuša najti približek k-dopa za orientiran ter transformiran originalni k-dop. Nov k-dop je izracunan iz tock trenutnega, vendar pa tak k-dop ni nujno najmanjši k-dop za trikotnike v vozlišcu.

V algoritmu je uporabljena aproksimacijska metoda.

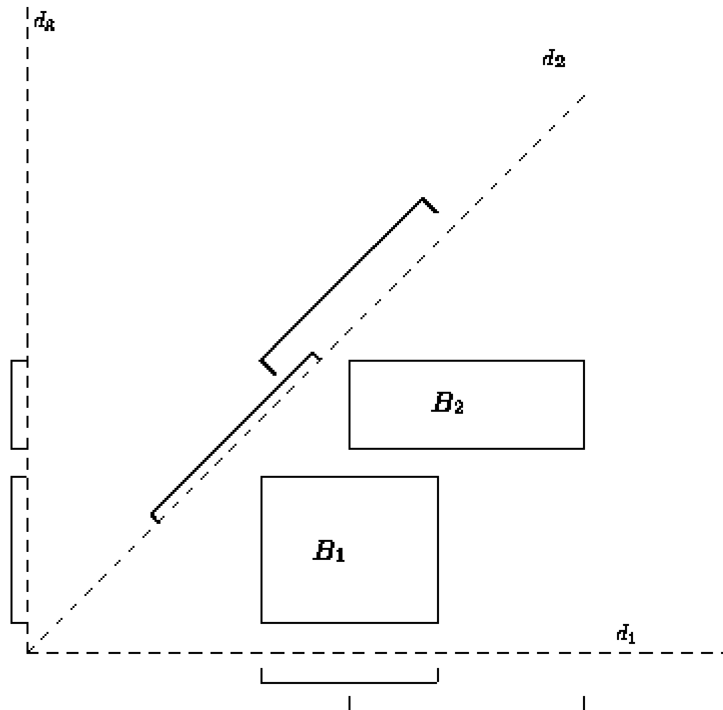


Slika 4 : Primerjava k-dopov rotiranega objekta dobljenih s *hill climbing* metodo (levo spodaj) in aproksimacijsko metodo (desno spodaj). Povzeto po [1].

Drevo je preiskano na podoben nacin kot pri algoritmih SOLID in RAPID. Ce sta obe trenutni vozlišci lista se naredijo testi trikotnik-trikotnik. Ce je eno izmed vozlišc notranje, drugo pa list, se notranje vozlišce razvija naprej (v primeru, da se k-dopa prekrivata). Ce pa imamo opravka z dvemi notranjimi vozlišci, ki se prekrivata, se naprej razvija vozlišce ki pripada staticnemu objektu (okolju).

Pri preverjanju prekrivanja dveh k-dopov se uporablja projiciranje k-dopa na vsako izmed *k/2* osi in preverjanje, ali se intervala prekrivata. Ce najdemo os, na kateri se intervala ne prekrivata, potem se k-dopa ne prekrivata. Ta test je nekoliko konservativen, saj se lahko zgodi, da se k-dopa ne prekrivata kljub temu, da test javi da se (za popolni test bi morali uporabiti zaprtje (*closure*) množice *k* vektorjev, vec o tem piše v [1]). Pomemben je tudi vrstni red preverjanja osi, ki ga

uporabljamo pri testu, saj je verjetno, da se bosta k-dopa, ki se prekrivata na neki osi, prekrivala tudi na osi, ki je precej blizu tej. Zato so osi razporejene tako, da se smeri osi v zaporednih testih cimbolj razlikujejo z upanjem, da bo algoritem prej našel os, kjer se k-dopa ne prekrivata.



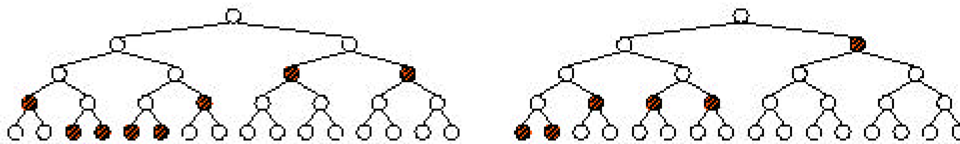
Slika 5 : Primer razvršcanja osi - intervale na d3 preverjamo pred d2 (ker je os d3 bolj oddaljena od d1, kot os d2). Povzeto po [1].

Za teste trikotnik-trikotnik QuickCD uporablja knjižnico ERIT, ki je opisana v [9].

2.4.3. Izboljšave algoritma

Pomembna lastnost algoritma je uporaba ospredja (*front tracking*), ki pa se v nekaterih primerih ne obnese najbolje (glej [5]). Pri primerjanju premikajocega objekta s staticnim okoljem moramo vedno znova preiskovati drevo okolja, pri cemer pogosto ugotovimo, da se neko vozlišce, ki se že prej ni sekalo s premikajocim objektom, ponovno ne seka. Z izkorišcanjem predpostavke da se premikajoci objekt med dvema preverjanjema ne premakne veliko (casovna koherenca), si algoritem zapomni, kje v drevesu okolja se nahaja koren premikajocega objekta. Tako se shranjujejo vozlišca pri katerih je algoritem pri preiskovanju drevesa ugotovil, da se ne prekrivajo s korenom premikajocega

objekta ter listi drevesa, ki se prekrivajo s korenom premikajocega objekta. Naslednje preiskovanje drevesa lahko zacnemo v teh vozlišcih. Ospredje je potrebno osveževati v skladu s premikanjem premikajocega objekta. Ce se neko vozlišce okolja, ki se prej ni prekrivalo, sedaj prekriva spustimo ospredje (*drop the front*). Z rekurzivnim ponavljanjem tega postopka za sinove tega vozlišca se pomikamo proti listom drevesa oziroma novemu vozlišcu, pri katerem je ugotovljeno, da se ne prekriva s korenom premikajocega objekta. Poleg tega je potrebno skrbeti tudi za dvigovanje ospredja (*raising the front*). To se naredi le za en korak navzgor v primeru, ko se ugotovi, da se vozlišce v okolju, ki se je prej prekrivalo, sedaj ne prekriva vec in je njegov naslednik v ospredju (gledano od leve proti desni) njegov brat, ki se tudi ne prekriva s premikajocim objektom.



Slika 6 : Primer osveževanja ospredja. Premikajoci objekt se je pomaknil proti levemu delu drevesne hierarhije staticnega objekta. Povzeto po [1].

Prednost uporabe ospredja je v zmanjšanju potrebnih testov med k-dopi. Avtor v [1] navaja povprecno 15% izboljšanje casa za odkrivanje trka pri uporabi ospredja.

2.5. SWIFT++

SWIFT++ je algoritem, ki zna poleg odkrivanja trkov reševati še druga poizvedovanja povezana z odnosom med dvema objektoma v prostoru. Poizvedovanja, ki jih ta algoritem zna izvesti so:

(a) Odkrivanje trkov

(b) Preverjanje tolerance (Ob dani toleranci algoritem preveri ali je razdalja med objektoma znotraj tolerance ali ne. Ta tip poizvedovanja je v bistvu posplošen problem odkrivanja trkov.).

(c) Racunanje tocne najmanjše razdalje (Ob podani toleranci algoritem izracuna najmanjšo razdaljo, ce je le ta manjša od tolerance.).

(d) Racunanje približne najmanjše razdalje (Podana je absolutna napaka, relativna napaka in toleranca, v primeru, da je razdalja manjša od

tolerance, vrne algoritem razdaljo, ki je znotraj katerekoli izmed obeh podanih mej za napako.).

(e) Ugotavljanje dotikov (S pomocjo podane tolerance se izracuna množica parov delov objektov, tako da vsak par predstavlja lokalni minimum v funkciji razdalje med dvema objektoma ter, da je lokalni minimum manjši od tolerance. Iz teh parov se potem lahko izracuna razdalje, najbližje tocke in normale dotika, kar nam da preprost opis podrocja dotika, ki je uporaben pri odzivanju na trk.).

Poleg tega, da nam ta algoritem zna odgovoriti na najvec vprašanj, je po navedbah v [5] tudi najhitrejši v skupini algoritmov, ki znajo odkrivati trke med nekonveksnimi objekti.

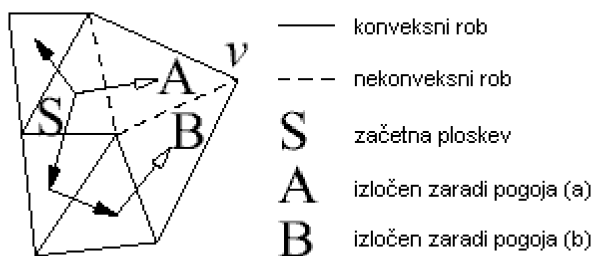
2.5.1. Gradnja drevesne hierarhije

Prvi korak pri gradnji drevesa, ki predstavlja objekt, je dekompozicija objekta na konveksne kose. To so konveksne lupine izracunane iz konveksnih površin. Algoritem med sestavljanjem površin raziskuje relacije med trikotniki. Uporablja se raziskovanje grafa in inkrementalni algoritem za izracun konveksne lupine. Za dobro dekompozicijo velja tista katere množica konveksnih kosov porabi malo racunalniškega spomina. Graf modela so preprosto njegovi robovi in tocke. Temu dualen graf je graf, kjer so vloge robov in tock obrnjene. Algoritem raziskuje dualni graf. Ko se dodaja trikotnike v trenutno konveksno površino c, se racuna tudi konveksna lupina C. Po koncu iskanja je c množica trikotnikov iz ploskve S, ki so bile dodane med iskanjem, C pa njihova konveksna lupina. Ko iskanje pride do trikotnika f, imamo tocko v, ki leži nasproti roba e na trikotniku. Dodajanje v v konveksno lupino C, je enakovredna dodajanju f v c. Da je v dodana v C, mora izpolnjevati naslednje zahteve:

(a) e ni konveksni rob (Konveksni rob je rob med trikotnikoma, katerih normali kažeta v nasprotni smeri).

(b) Noben trikotnik iz S, ki so v c ni viden iz v.

(c) Dodajanje v v C ne povzroci da bi se C' sekal s katerimkoli delom S, ki ni v c.



Slika 7 : Primer gradnje konveksnega kosa. Trikotnik A ni dodan, ker je rob, ki si ga deli z enim izmed sosednjih trikotnikov na ploskvi S, nekonveksen. Trikotnik B je izlocen zaradi pogoja (b) (nek drugi trikotnik na ploskvi je viden s trikotnika B). Povzeto po [5].

Naslednji korak je gradnja drevesa. Tudi tu se uporablja rekurzivno deljenje. V tem primeru so primitivi konveksni kosi izracunani v prejšnjem koraku. Tocka, ki predstavlja konveksni kos, je masna tocka konveksnega kosa. Za delilno os se uporablja os, na kateri so konveksni kosi najbolj razpršeni. Masne tocke konveksnih kosov so projicirane na delilno os, nato pa se množico konveksnih kosov razdeli na dva dela, kot delilna tocka pa se lahko uporabi:

(a) Mediana (*median*).

(b) Povprecje prve in zadnje vrednosti.

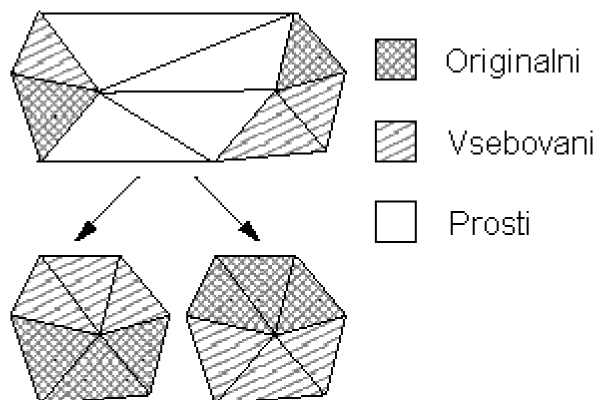
(c) Povprecje vrednosti vseh tock (*mean*).

Med gradnjo drevesa se racunajo tudi novi trikotniki. Trikotnike v drevesu lahko razvrstimo v tri kategorije:

(a) Originalni (objektovi trikotniki).

(b) Vsebovani (trikotniki, ki smo jih dobili z izracunom konveksne lupine; to so trikotniki, ki so znotraj originalnega objekta, ce je le ta zaprt)

(c) Prosti (trikotniki dodani med gradnjo drevesne hierarhije; prosti so v smislu, da niso vsebovani v nobenem konveksnem kosu).



Slika 8 : Primer konveksnega kosa v drevesni hierarhiji in tipi trikotnikov, ki sestavljajo tak kos. Povzeto po [5].

2.5.2. Izvajanje poizvedovanj

Jedro algoritma je procedura, ki racuna razdaljo med parom konveksnih kosov. Za to se uporablja iterativen algoritem za minimizacijo, ki je osnovan na lokalni optimizaciji. Algoritem se sprehaja po površini obeh konveksnih objektov, da bi našel par njunih najbližjih trikotnikov. Za usmerjanje sprehajanja po površini se uporabljajo Voronoi obmocja. Voronoi obmocje za nek trikotnik je doloceno s tockami, ki so bližje temu trikotniku kot kateremukoli drugemu trikotniku. O Voronoi obmocjih je nekoliko vec napisanega v [11].

Pri raziskovanju hierarhicnega drevesa se najprej izracuna par najbližjih trikotnikov. Ce oba trikotnika nista tipa prost, se rekurzija prekine in lahko se izracuna odgovor. V nasprotnem primeru se drevo preiskuje naprej.

Raziskovanje drevesa se pri posameznih poizvedovanjih razlikuje. Pri preverjanju tolerance se preiskovanje drevesa lahko prekine, ce je razdalja med trikotnikoma, ki jih je vrnilo sprehajanje po površini, vecja od tolerance, ker vemo, da nobeno od poddreves ne more biti bližje. Rezultat je *false*. V primeru, ko pa je razdalja manjša od tolerance in sta oba trikotnika ali originalna ali vsebovana, potem lahko vrnemo *true*. V nasprotnem primeru se drevo preiskuje naprej.

Pri racunanju razdalje med objektoma, se vsakemu rekurzivnemu klicu poda tudi najmanjšo do sedaj najdeno razdaljo. V primeru, ko je razdalja med trenutno primerjanim parom vecja od trenutne minimalne razdalje, ali pa, ko poddrevo ne more vplivati na razdaljo, ni potrebe, da bi to vejo drevesa

raziskovali naprej. Osnovni pogoj rekurzije je dosežen, ko sta oba najbližja trikotnika drugacnega tipa kot prost. Ce se trikotnika sekata se vrne negativno razdaljo. Razdalja od tu naprej ne more biti še bolj zmanjšana, ker se objekta sekata. V nasprotnem primeru se vrne razdaljo med trikotnikoma, ce je ta manjša od razdalje, ki je bila podana v rekurzivnem klicu. Ce je eden izmed trikotnikov v paru prost, potem se nadaljuje rekurzivno raziskovanje drevesa.

Ugotavljanje dotikov je podobno racunanju razdalje med objektoma. V primeru, ko sta oba trikotnika drugega tipa kot prost in, ce se trikotnika sekata, je iskanje prekinjeno in poizvedovanje vrne odgovor, da je prišlo do trka. V nasprotnem primeru se najbližji par trikotnikov doda v seznam parov. Ce je eden izmed trikotnikov prost se nadaljuje rekurzija.

Algoritem za uporablja tri razlicne pristope za pohitritev opisanih poizvedovanj. To so:

(a) Pomnenje konveksnih kosov (Shranjevanje para konveksnih kosov, ki je odgovoril na prejšnje poizvedovanje z upanjem, da se objekt v vmesnem casu ni veliko premaknil in bo ta par spet odgovoril na poizvedovanje.).

(b) Usmerjeno iskanje (Prednost pri preiskovanju imajo pari poddreves, katerih koreni so najmanj oddaljeni.).

(c) Posplošeno ospredje (*front tracking*) (S to metodo se želimo izogniti vsakokratnemu ponovnemu preiskovanju celotnega drevesa, s tem da si zapomnimo dolocena vozlišca v drevesu, ki jih nato med poizvedovanjem prilagajamo. Princip delovanja ospredja je isti kot pri QuickCD).

Natancnejši opis algoritma in rezultatov se nahaja v [5].

2.6. I-COLLIDE

Ta algoritem je skupaj s Q-COLLIDE, ki je opisan v naslednjem podpoglavju predstavnik algoritmov, ki so v osnovi namenjeni odkrivanju trkov med konveksnimi objekti. I-COLLIDE so avtorji z uporabo hierarhicnih dreves priredili tudi za odkrivanje trkov med nekonveksnimi objekti.

2.6.1. Odkrivanje trkov

Algoritem je osnovan na Lin-Canny algoritmu, ki si shranjuje najbližje tocke med parom konveksnih objektov. Metoda izkorišca casovno koherenco – najbližji trikotniki se redko spreminjajo, ce se objekta premikata po majhnih korakih. Algoritem zacne z dvema kandidatoma za najbližja trikotnika in preveri, ce najbližji tocki ležita na teh dveh trikotnikih. Ker sta objekta konveksna je to lokalno iskanje. Ce tocki ne ležita na teh dveh trikotnikih, se izbere sosednji trikotnik od enega ali obeh trenutnih trikotnikov in poskusi znova. Naslednji trikotnik se izbere s pomocjo Voronoi obmocij.

Odkrivanje penetracije enega objekta v drugega uporablja nekatere lastnosti Voronoi obmocij za konveksne objekte. Notranjost objekta je razdeljena na psevdo-Voronoi obmocja, ki jih dobimo tako, da vsako tocko na objektu povežemo s centrom objekta. Objekta sta trcila, ce katerikoli izmed najbližjih trikotnikov, ki jih dobimo po koncu sprehajanja, spada v psevdo-Voronoi obmocje. Natancnejši opis tega postopka se nahaja v [6].

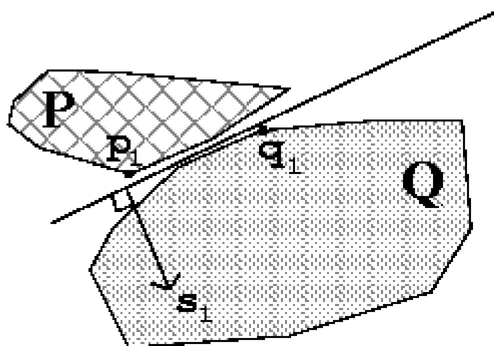
Prireditev algoritma za nekonveksne objekte je narejena z uporabo hierarhicnih dreves. Listi takega drevesa so konveksni objekti. Drevo je zgrajeno od spodaj navzgor z združevanjem konveksnih objektov in racunanjem konveksne lupine ali kakšne druge vrste mejnega volumna. Drevo se raziskuje rekurzivno. Ce trka ni algoritem vrne par najbližjih lastnosti mejnih volumnov. I-COLLIDE zna tudi ugotavljati potrebne teste med objekti v okolju, za to uporablja metodo opisano v poglavju 4.1.3 (primerjanje intervalov na X, Y in Z osi).

2.7. Q-COLLIDE

Q-COLLIDE je izboljšava algoritma I-COLLIDE. Glavna izboljšava je v tem da algoritem ne racuna para najbližjih trikotnikov. Namesto tega se uporablja algoritem iskanja delilnega vektorja (*separating vector*). Kot navaja avtor v [7] je to velika izboljšava v smislu hitrosti, preprostosti implementacije ter porabe spomina. Ker je algoritem podoben algoritmu I-COLLIDE ga ne bom podrobneje opisoval. Natancnejši opis algoritma lahko bralec dobi v [7].

2.7.1. Delovanje algoritma

Osnovna ideja algoritma je uporaba delilne ravnine med objektoma za preverjanje trkov med objektoma. Ta ravnina je shranjena in v naslednjem koraku algoritem najprej preveri ali objekta ležita na nasprotnih straneh ravnine, ce to ne velja algoritem poskuša najti novo delilno ravnino, kar naredi v koncnem številu korakov. Ce delilna ravnina ne obstaja bo algoritem porocal o trku, informacija iz prejšnjih pozicij objekta pa je uporabljena za izracun najbližjih tock.



Slika 9 : Delilna ravnina med dvema konveksnima objektoma (povzeto po [7]).

3. CD – knjižnica za odkrivanje trkov

V tem poglavju je opisana knjižnica za odkrivanje trkov (CD). Ciljno podrocje uporabe so predvsem igrice, zato je implementacija prilagojena temu. Knjižnica se seveda lahko uporablja tudi za drugacne namene.

3.1. Izhodišca

Za osnovo algoritma je bil vzet algoritem QuickCD, ki temelji na uporabi k-dop mejnih volumnov. Ker so avtorji tega algoritma pokazali (glej [1]), da se najbolje obnese 18-dop mejni volumen je ta uporabljen v CD. Tudi nekaj drugih odlocitev, narejenih med razvojem algoritma je osnovanih na rezultatih iz [1]. Bistvene razlike med CD in QuickCD algoritmom sta dve:

(a) Algoritem QuickCD je zasnovan s predpostavko, da je eden od objektov staticen (okolje). CD pa zna odkrivati trke tudi med dvema premikajocima objektoma.

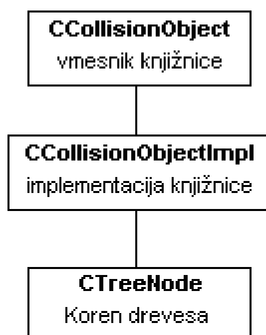
(b) QuickCD ima v listih drevesa doloceno število trikotnikov. Maksimalno število trikotnikov v listih je pri QuickCD za staticen objekt enako ena, za premikajoci objekt pa je doloceno z mejno vrednostjo. CD ima v listih drevesa konveksne ploskve, ki jih dobimo z dekompozicijo osnovnega objekta (tudi tu imamo mejno vrednost za maksimalno število trikotnikov na taki ploskvi).

Konveksnost ploskev CD izkorišca na zadnji stopnji preverjanja trkov – testi trikotnik-trikotnik. Zaradi konveksnosti ploskev lahko izvajamo lokalno optimizacijo in se tako hitreje približujemo paru trikotnikov, ki se sekata. To pomeni, da ni potrebno narediti testov med vsemi trikotniki v listih, ki jih primerjamo. Vec o tem v podpoglavju 3.4.

3.2. Podatkovna struktura

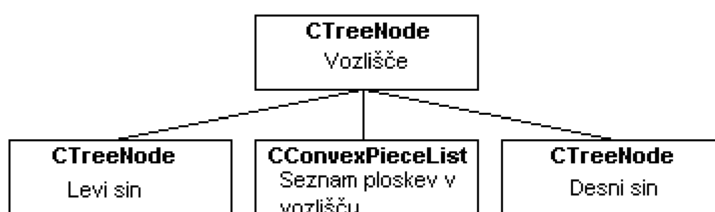
Razred, ki predstavlja objekt in je viden uporabniku knjižnice je *CCollisionObject*. Njegova funkcionalnost je implementirana v razredu *CCollisionObjectImpl*. Ta razred vsebuje podatke o drevesu in poziciji objekta ter

metode za gradnjo drevesa, preverjanje trkov, nastavljanje pozicije in branje rezultatov testa. Vec o teh metodah in njihovi uporabi je napisanega v dodatku.



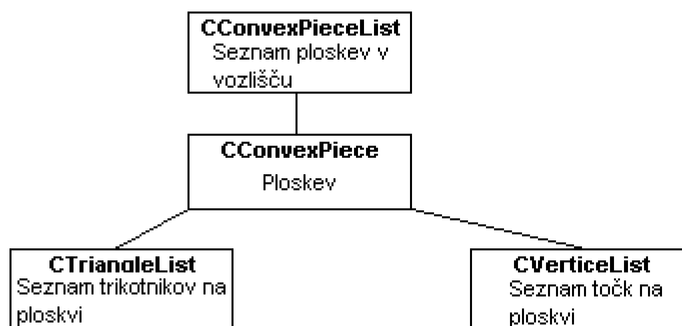
Slika 10 : CcollisionObject predstavlja funkcionalnost knjižnice, ki je vidna uporabniku. CcollisionObjectImpl je implementacija celotne funkcionalnosti knjižnice, podatki potrebni za delovanje so shranjeni v razredu CtreeNode.

Razred *CTreeNode* predstavlja vozlišce v drevesu. Drevo je binarno. Vozlišce ima podatke o mejah 18-dopa, svojih sinovih, seznam konveksnih ploskev, ki jih vsebuje, ter njihovo število. Operacije nad tem razredom so dodajanje trikotnikov, dodajanje konveksnih ploskev, izracun podatkov o mejnem volumnu ter deljenje vozlišca.



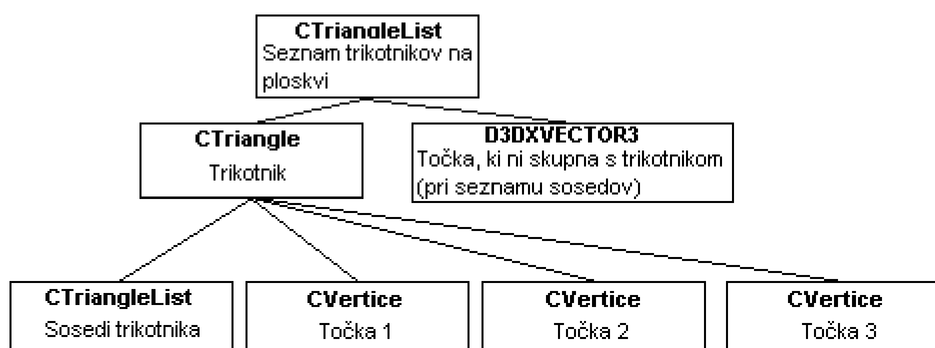
Slika 11 : Struktura drevesnega vozlišca. Vozlišce ima levega in desnega sina ter seznam ploskev v vozlišcu.

CConvexPieceList predstavlja seznam konveksnih ploskev v vozlišcu. Konveksno ploskev predstavlja razred *CConvexPiece*. Nad konveksno ploskvijo lahko delamo naslednje operacije: dodajanje trikotnikov na ploskev (dodajanje uspe, ce trikotnik spada na ploskev) in izracun centra konveksne ploskve. Podatki, ki predstavljajo konveksno ploskev so: center konveksne ploskve (uporabljeno je kar povprecje vseh tock na ploskvi), meje 18-dopa za to ploskev, seznam tock na ploskvi (*CVerticeList*) ter seznam trikotnikov na ploskvi (*CTriangleList*). V seznam konveksnih ploskev lahko dodajamo nove trikotnike (vnese se v eno od obstojecih ploskev, oz. na novo ploskev, ce ne spada na nobeno od obstojecih) in vanj dodajamo nove ploskve.



Slika 12 : Struktura seznama konveksnih ploskev. Ploskev vodi podatke o trikotnikih na ploskvi ter tockah, ki jo sestavljajo.

Seznam trikotnikov predstavlja razred *CTriangleList*. Omogoca operacije dodajanja novih trikotnikov v seznam, kopiranja trikotnikov iz drugih seznamov ter brisanja prvega elementa v seznamu. Element seznama hrani podatke o trikotniku (razred *CTriangle*). Ce seznam predstavlja seznam sosedov nekega trikotnika je tu še podatek o tocki, ki ni skupna z trikotnikom kateremu pripada seznam sosedov. Razred *CTriangle* vsebuje podatke o tockah trikotnika (kazalci na tocke, ki so shranjene v seznamu tock (*CVerticeList*) za pripadajoco konveksno ploskev), seznam sosedov (*CTriangleList*) ter nekaj zacasnih vrednosti, ki se uporabljajo med odkrivanjem trkov.



Slika 13 : Struktura seznama trikotnikov. S trikotnikom so povezani podatki o njegovih sosedih ter ogliščih. Ce seznam predstavlja seznam sosedov nekega trikotnika ima vsak element seznama še podatke o oglišcu, ki ni skupno s trikotnikom kateremu seznam sosedov pripada.

Seznam tock je predstavljen z razredoma *CVerticeList* in *CVertice*. Možno je vstavljanje novih tock v seznam, koordinate tocke so edini podatek, ki je shranjen za posamezno tocko.

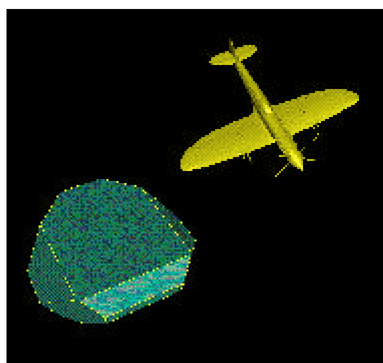
Slika 14 : Struktura seznama tock.

3.3. Gradnja hierarhicnega drevesa

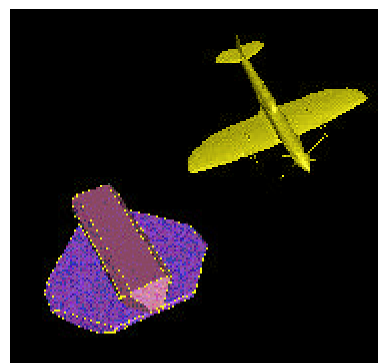
Gradnja hierarhicnega drevesa je sestavljena iz dveh korakov. Prvi je dodajanje trikotnikov v koren drevesa. Drugi je gradnja drevesne hierarhije (deljenje vozlišc).

1	2	3	4	5	6	7	8	9
(1,0,0)	(0,1,0)	(0,0,1)	(1,1,0)	(1,0,1)	(0,1,1)	(1,-1,0)	(1,0,-1)	(0,1,-1)

Tabela 1 : Devet smeri (v formatu (X,Y,Z)), v katerih se racunajo meje 18-dopa.



Slika 15 : Letalo Spitfire in pripadajoci 18-dop na prvem nivoju drevesne hierarhije (koren). Povzeto po [1].



Slika 16 : Drugi nivo drevesne hierarhije za letalo Spitfire. Povzeto po [1].

3.3.1. Dodajanje trikotnikov v koren drevesa

Za vsak trikotnik, ki ga dodajamo v drevo, je potrebno najti ustrezno konveksno ploskev kateri pripada. Ce le te ne najdemo je tak trikotnik del nove konveksne ploskve. Med dodajanjem trikotnikov se racunajo tudi nekateri podatki, ki jih rabimo kasneje v drugem koraku gradnje drevesa in med samim postopkom odkrivanja trkov. Taki podatki so seznam sosednih trikotnikov dodanega trikotnika, nove meje konveksne ploskve (po tem, ko je novi trikotnik dodan) in novi center konveksne ploskve. Poleg tega je potrebno v seznam tock

vnašati morebitne nove tocke. Trikotniki se v drevo dodajajo po naslednjem postopku:

```
procedure AddTriangle(v1,v2,v3); //v1,v2,v3 are vertices
begin
  //search for piece
  Piece = PieceList.First;
  PieceFound = false;
  while not (Piece = NULL) do
  begin
    //is threshold for number of triangles already reached?
    if not (Piece.TriangleNum > MaxTriangleNum) then
    begin
      //compare triangle to other triangles
      Triangle = Piece.TriangleList.First;
      TriangleAdded = true;
      while not (Triangle = NULL) do
      begin
        //check convexity conditions
        if not FormsConvexPiece(Triangle,v1,v2,v3) then
        begin
          TriangleAdded = false;
          break;
        end else
          //store triangle to new triangle's neighbours list
          StoreToTriangleList(Neighbours,Triangle);
        Triangle = Triangle.Next;
      end;
      //if all convexitiy checks were successfull do the insert
      if TriangleAdded then
      begin
        //set data for new triangle
        newTriangle = Piece.TriangleList.Insert(v1,v2,v3,
                                                Neighbours);
        //update existing triangles neighbours lists
        Neighbour = Neighbours.First;
        while not (Neighbour = NULL) do
        begin
          //determine which vertice of new triangle is different
          distinctVertice = CheckVertices(Neighbour,v1,v2,v3);
          //insert triangle into neighbours list
          Neighbour.NeighboursList.Insert(newTriangle,
                                          distinctVertice);
        end;
        //update piece data (k-dop limits, center)
        Piece.UpdateInternalData();
        //stop piece list search
        PieceFound = true;
        break;
      end;
    end;
    Piece = Piece.Next;
  end;

  //if corresponding piece was not found create new piece and
  //calculate its internal data
  if not PieceFound then
  begin
    newPiece = PieceList.CreateNewPiece(v1,v2,v3);
```

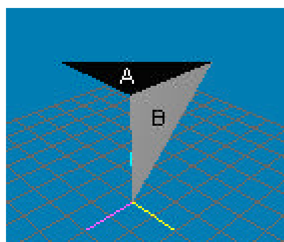
```
    newPiece.UpdateInternalData();
    inc(PieceCount);
  end;
end;
```

Algoritem gre v tem koraku skozi seznam konveksnih ploskev in preverja ali je možno trikotnik dodati v ploskev na kateri se trenutno nahaja. To preverjanje je potrebno narediti za vse trikotnike na ploskvi. Preverjanje samo je sestavljeno iz treh pogojev:

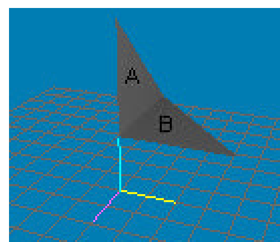
(a) Normali obeh trikotnikov morata biti usmerjeni na isto stran ploskve.

(b) Površina drugega trikotnika (stran v katero kaže normala) ne sme biti vidna iz prvega trikotnika in obratno.

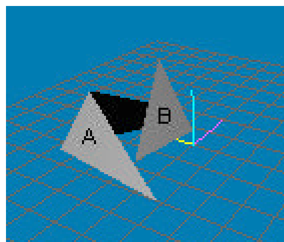
(c) Ce trikotnika nimata skupnih tock, potem trikotnik, ki ga dodajamo, ne sme sekati ravnine trikotnika s katerim ga trenutno primerjamo.



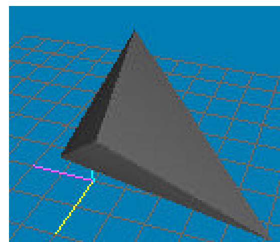
Slika 17 : Primer neizpolnjevanja prvega pogoja - trikotnik A ima normalo usmerjeno na drugo stran ploskve kot trikotnik B.



Slika 18 : Primer neizpolnjevanja drugega pogoja – trikotnik B je viden s trikotnika A.



Slika 19 : Primer neizpolnjevanja tretjega pogoja - ce dodamo trikotnik A, bo ta sekal ravnino trikotnika B.



Slika 20 : Primer konveksne ploskve sestavljene iz treh trikotnikov.

V primeru da pri nekem trikotniku eden izmed pogojev ni izpolnjen se iskanje na trenutni konveksni ploskvi lahko prekine. Ce je trikotnik uspešno dodan na neko ploskev je potrebno ponovno izracunati meje mejnega volumna in center ploskve. Ce trikotnik ni uspelo dodati na nobeno izmed obstojecih konveksnih ploskev se naredi nova konveksna ploskev, v katero se doda novi trikotnik.

3.3.2. Gradnja drevesne hierarhije

Ko so dodani vsi trikotniki, se zacne drugi korak gradnje drevesa. To je gradnja drevesne hierarhije, le ta poteka rekurzivno. Rekurzija se konca ko je v vsakem vozlišcu le še ena konveksna ploskev. Postopek delitve je takšen:

```
procedure DivideTree(node);
begin
  //divide only if there are more than one piece in this node
  if (node.PieceCount > 1) then
  begin
    //compute mean and variance of all piece centers in node
    //(computed separately for X,Y and Z axis)
    ComputeVariance(node,mean_x,mean_y,mean_z,var_x,var_y,var_z);
    //create left and right son
    Left = new TreeNode;
    Right = new TreeNode;
    //choose direction to split (splatter method)
    sort(var_x,var_y,var_z,outArray);
    if (var_x = outArray[0]) then //var_x is largest
    begin
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.x < mean_x) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.x > mean_x) then begin
          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.x on mean_x
          //put first, third, fifth etc in left node
          //others in right node
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end else if (var_y = outArray[0]) then var_y is largest
    begin
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.y < mean_y) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.y > mean_y) then begin

          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.y on mean_y
          //put first, third, fifth etc in left node
          //others in right node
```

```
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end else begin //var_z is largest
      piece = node.PieceList.First;
      while not (piece = NULL) do
      begin
        if (Piece.Center.z < mean_z) then begin
          Left.InsertPiece(Piece);
          Left.ProcessNode();
        end else if (Piece.Center.z > mean_z) then begin
          Right.InsertPiece(Piece);
          Right.ProcessNode();
        end else begin //pieces with Center.z on mean_z
          //put first, third, fifth etc in left node
          //others in right node
          if evenPos then begin
            Left.InsertPiece(Piece);
            Left.ProcessNode();
          end else begin
            Right.InsertPiece(Piece);
            Right.ProcessNode();
          end;
        end;
      end;
    end;
  end;
end;
```

Pravkar opisani algoritem razdeli eno vozlišce na dve (levi in desni sin). Sama rekurzija se izvaja na enem nivoju višje, in se konca ko vozlišc ni vec možno deliti (vozlišce vsebuje eno samo konveksno ploskev). Za delilno os se podobno kot pri QuickCD uporablja eno izmed osi X, Y in Z. Izbere se tisto os na kateri je varianca centrov konveksnih ploskev najvecja, v [1] je ta metoda poimenovana splatter. Pri splatter metodi projiciramo centre konveksnih ploskev na osi X, Y in Z ter racunamo varianco tock na projekcijah. Izberemo tisto os, pri kateri je varianca najvecja. Tocka deljenja je srednja vrednost projekcij tock na izbrani osi. V primeru, ko je projekcija centra neke konveksne ploskve na izbrano os enaka srednji vrednosti, dodamo vsako drugo tako ploskev v desnega sina, ostale pa v levega sina.

Nekoliko natancneje je potrebno opisati še proceduro *ProcessNode*, ki se klice v zgoraj opisanem algoritmu. Ko dodamo neko ploskev v vozlišce, se s tem spremenijo meje mejnega volumna za to vozlišce. Tu izkorišcamo že izracunano

informacijo o mejah konveksne ploskve, ki je bila izracunana v predhodnem koraku izgradnje drevesa (dodajanje trikotnikov). Za izracun novih mej nam tako ni potrebno preverjati vseh trikotnikov v vozlišcu, ampak je dovolj le združitev starih mej z mejami nove konveksne ploskve v tem vozlišcu. To pa je preprosta operacija primerjanja mejnih vrednosti in uporabe manjše oziroma vecje vrednosti.

3.4. Odkrivanje trkov

Algoritem odkrivanja trkov deluje v dveh korakih. Najprej preiskuje hierarhicno drevo, ce pa v obeh primerjanih drevesih pridemo do listov, se naredijo še testi trikotnik-trikotnik.

3.4.1. Preiskovanje drevesa

Preiskovanje drevesa poteka rekurzivno, algoritem omogoca tudi prekinjanje preiskovanja v primeru, ko je prekoracen dovoljen cas za izvedbo testa (ce je dovoljeni cas podan). Pri preiskovanju drevesa se uporablja naslednja hevristika za izbiro vozlišca, ki se bo razvijalo naprej:

(a) Primerjaj trikotnike v vozlišcu, ce sta obe vozlišci lista drevesa.

(b) Ce je eno vozlišce list, drugo pa notranje vozlišce, se razvija notranje vozlišce, pri tem je najprej obiskan manjši sin.

(c) Ce imamo opravka z dvemi notranjimi vozlišci se vedno najprej razvije vecje vozlišce.

Psevdo koda za preiskovanje drevesa je takšna:

```
function TraverseTree(node1,node2) : boolean;
begin
  //check if allowable time elapsed
  if Timeouted then Exit;
  //transform limits of object two
  TransformLimits(transformMatrix,node2.Limits,Limits1);
  //check if intervals overlap
  if not IntervalsOverlap(Limits1,node1.Limits) then
    return false;
  else begin

    if Leaf(node1) and Leaf(node2) then
      CheckTriangles(node1,node2);
    else if Node(node1) and Leaf(node2) then
    begin
      c1 = TraverseTree(SmallerSon(node1),node2);
```

```
      if not StopCondition then
      begin
        c2 = TraverseTree(LargerSon(node1),node2);
        collision = c1 or c2;
      end;
    end else if Leaf(node1) and Node(node2) then
    begin
      c1 = TraverseTree(node1,SmallerSon(node2));
      if not StopCondition then
      begin
        c2 = TraverseTree(node1,LargerSon(node2));
        collision = c1 or c2;
      end;
    end else begin //both nodes are internal
      if Volume(node1) > Volume(node2) then begin
        Larger = node1; Smaller = node2;
      end else begin
        Larger = node2; Smaller = node1;
      end;
      c1 = TraverseTree(Larger.Left,Smaller);
      if not StopCondition then
      begin
        c2 = TraverseTree(Larger.Right,Smaller);
        collision = c1 or c2;
      end;
    end;
    result = collision;
  end;
end;
```

Algoritem najprej preveri, ali je se je dovoljeni cas iztekel. V naslednjem koraku se s z uporabo transformacijske matrike transformirajo meje 18-dopa drugega vozlišca. Uporabljena transformacijska matrika nam 18-dop drugega vozlišca pripelje v isti relativni položaj glede na 18-dop prvega vozlišca, kot bi ga dobili, ce bi transformirali položaja obeh 18-dopov z njima pripadajocima transformacijskima matrikama. Matrika je izracunana že vnaprej in jo dobimo z množenjem transformacijske matrike drugega vozlišca z inverzno transformacijsko matriko prvega vozlišca. Razlog za uporabo take matrike je pohitritev algoritma (manj racunskih operacij). Transformirane meje se nato primerjajo z mejami prvega vozlišca. Ce se 18-dopa na eni izmed 9-ih osi ne prekrivata, se nadaljnje preiskovanje tega dela drevesa prekine. V nasprotnem primeru se z uporabo že zgoraj omenjene hevristike drevesi preverjata naprej, vse dokler ne pridemo do zakljucka, ali sta objekta trcila ali ne. *StopCondition*, ki se uporablja v zgornjem algoritmu, preveri, ali je bilo definirano maksimalno število tock trka, ki jih CD lahko vraca, že izracunanih. Algoritem ima namrec doloceno maksimalno število tock, ki jih vraca v primeru trka. Trenutna verzija knjižnice

ima to vrednost nastavljeno na 2, kar je dovolj natancno za uporabo v preprostih igricah.

3.4.2. Testi trikotnik-trikotnik

List drevesa vsebuje eno konveksno ploskev, ki je lahko sestavljena iz vec trikotnikov. Z upoštevanjem dejstva, da je ploskev konveksna, se lahko izognemo preverjanju vseh trikotnikov enega lista s tistimi iz drugega lista. Iz zacetnega trikotnika se preko njegovih sosedov pomikamo vse bližje ravnini na kateri leži trikotnik iz drugega lista. Ce pri iskanju ugotovimo, da se je predznak razdalje od ravnine spremenil, naredimo test sekanja trikotnikov. Algoritem je takšen:

```
function CheckTriangles(node1,node2) : boolean;
begin
  collision = false;
  //select starting triangle for outer loop
  outertriangle = node2.PieceList.Piece.TriangleList.First;
  while not (outerTriangle = NULL) do
  begin
    //transform vertices according to transform matrix
    TransformVertices(outerTriangle,transV1_o,transV2_o,transV3_o,
                      transformMatrix);
    //compute plane on which triangle resides (normal is plane
    //normal, d is fourth parameter from plane equation)
    ComputePlane(transV1_o,transV2_o,transV3_o,normal,d);
    //choose starting triangle for inner loop
    innerTriangle = node1.PieceList.Piece.TriangleList.First;
    //put triangle into test queue
    testQueue.CopyTriangleToList(innerTriangle);
    //compute starting minimal distance from plane on which
    //outerTriangle resides, indicate if innerTriangle
    //intersects that plane
    mindist = MinimumDistance(normal,d,
                              testQueue.Triangle.V1,
                              testQueue.Triangle.V2,
                              testQueue.Triangle.V3,
                              testQueue.Triangle.IntersectsPlane);
    while not (testQueue.Triangle = NULL) do
    begin
      if testQueue.Triangle.IntersectsPlane then
      begin
        //check if triangles truly intersect, coplanar indicates
        //that both triangles share all three points which define
        //them, P1 and P2 are intersection points
        if TrianglesIntersect(transV1_o,transV2_o,transV3_o,
                              testQueue.Triangle.V1,
                              testQueue.Triangle.V2,
                              testQueue.Triangle.V3,
                              P1,P2,coplanar) then
        begin
          if coplanar then //any two points of triangle will do
          begin
            point[PointCount] = transV1_o;
```

```
            inc(PointCount);
            point[PointCount] = transV2_o;
            inc(PointCount);
          end else begin
            point[PointCount] = P1;
            inc(PointCount);
            point[PointCount] = P2;
            inc(PointCount);
          end;
          //indicate that collision occured and check if we need
          //to stop detection
          collision = true;
          if StopCondition then
            break;
        end;

        //current triangle was checked, we need to check it's
        //neighbours
        tempTriangle = testQueue.Triangle.Neighbours.First;
        //initialize starting values for neighbours
        Initialize(mindist,testQueue.Triangle.MinDist,
                   IntPlane,testQueue.Triangle.IntersectsPlane);
        //now we can remove current triangle from test queue
        testQueue.RemoveFirstFromList;
        while not (tempTriangle = NULL) do
        begin
          if NotAlreadyChecked(tempTriangle) then begin
            if (tempTriangle.DistinctVertice <> NULL) then
            begin
              temp = PointDistance(normal,d,
                                   tempTriangle.DistinctVertice);
              tempTriangle.IntersectsPlane =
                                    SignChanged(temp,mindist);
            end else begin
              temp = mindist;
              tempTriangle.IntersectsPlane = IntPlane;
            end;
            //check if we are any closer to plane
            if Closer(temp,mindist) then
            begin
              tempTriangle.MinDist = temp;
              testQueue.CopyTriangleToList(tempTriangle);
            end else begin
              tempTriangle.MinDist = mindist;
              if tempTriangle.IntersectsPlane then
                testQueue.CopyTriangleToList(tempTriangle);
            end;
            MarkTriangleAsChecked(tempTriangle);
          end;
          tempTriangle = tempTriangle.Next;
        end;
      end;
    end;
    //check stop condition
    if StopCondition then
      break;
    outerTriangle = outerTriangle.Next;
  end;
  //return result
  return collision;
end;
```

V zunanji zanki algoritma so trikotniki iz drugega vozlišca. Tocke trikotnikov iz tega vozlišca je potrebno transformirati tako, da so v enakem relativnem položaju glede na tocke prvega vozlišca, kot bi bile, ce bi transformirali tocke obeh vozlišc. Pri tem se uporablja transformacijska matrika, omenjena že pri preiskovanju drevesa. V naslednjem koraku se izracuna ravnina na kateri leži trenutni trikotnik iz zunanje zanke. Notranja zanka za vsak trikotnik, ki ga je potrebno testirati, izracuna njegovo razdaljo od ravnine trikotnika v zunanji zanki. V primeru, ko trikotnik iz notranje zanke seka ravnino se naredi natancnejši test. Ce je potrebno nadaljnje preverjanje (*StopCondition* ni izpolnjen), zacne algoritem pregledovati sosede trenutnega trikotnika v notranji zanki. V vrsto za preverjanje so dodani tisti trikotniki, pri katerih je najmanjša razdalja od ravnine manjša ali enaka trenutni razdalji ali pa ravnino sekajo. Na ta nacin se vse bolj približujemo ravnini. V primeru da najdemo soseda, ki seka ravnino, ga oznacimo. Taki trikotniki se vedno dodajo na vrh vrste. Ko bo ta trikotnik prišel na vrsto bo narejen natancnejši test. Ko se vrsta za preverjanje izprazni se notranja zanka zakljuci in v zunanji zanki se pomaknemo na naslednji trikotnik. Celoten postopek se prekine v primeru, ko je izpolnjen *StopCondition* oziroma, ko so bili vsi trikotniki preiskani.

Oris algoritma za preverjanje sekanja trikotnikov sledi v naslednjih stavkih. Algoritem najprej preveri, ali so tocke enega trikotnika na razlicnih straneh ravnine (kar naredimo že v zgoraj opisanem algoritmu), na kateri leži drugi trikotnik. Ce je to res nadaljujemo in izracunamo isto še v obratni smeri (ali drugi trikotnik seka ravnino prvega trikotnika). Ce je tudi to res izracunamo še presecišca trikotnika z ravnino za oba primera, kar nam da dve daljici. Nato je potrebno narediti primerjavo daljic, in vrniti rezultat (ali se trikotnika sekata in tocki v katerih se sekata). Vec o algoritmih za preverjanje sekanja trikotnikov in drugih geometrijskih objektov si lahko preberete v [9]. V CD je bil uporabljen test objavljen na [12], ki je bil rahlo modificiran v smislu odprave racunanja že izracunanih informacij.

3.5. Rezultati

CD sem testiral s simulacijo gibanja enega avtomobila skozi drugega ter s primerjanjem dveh krogel, pri cemer je bila ena za malenkost manjša od druge in postavljena znotraj vecje krogle. Pri tem sem meril povprecne case, ki jih je

algoritem dosegal pri trkih, ter case, ki jih je dosegal v položajih kjer ni bilo trka. Case sem meril za deset razlicic algoritma – dovoljeno število trikotnikov na konveksni ploskvi se je gibalo od ena do deset. Vsi izmerjeni casi so bili dobljeni na racunalniku z naslednjo konfiguracijo:

? Celeron 433, 128Mb RAM

? Inštalirani Windows98, DirectX 8.0 SDK (Retail verzije knjižnic)

3.5.1. Meritve

V naslednji tabeli so podani casi izmerjeni pri gibanju avtomobila skozi drugi avtomobil. Gibajoci avtomobil je bil sestavljen iz 905 trikotnikov, mirujoci pa iz 878 trikotnikov. Avtomobil se je najprej približeval drugemu avtomobilu, ko pa sta bila oba v isti tocki, se je gibajoci avtomobil zavrtel za 360? in se nato oddaljeval od mirujocega avtomobila do koncne tocke.

Število dovoljenih trikotnikov	Cas za odkritje trka [ms]	Cas za ugotovitev da ni trka [ms]	Povprecni cas [ms]
1	0.626	0.033	0.429
2	0.475	0.034	0.320
3	0.553	0.030	0.349
4	0.459	0.032	0.296
5	0.513	0.031	0.333
6	0.422	0.033	0.274
7	0.460	0.037	0.296
8	0.459	0.035	0.297
9	0.467	0.032	0.309
10	0.469	0.033	0.313

Tabela 2 : Casi prvega testa (dva avtomobila).

Iz tabele je razvidno, da se algoritem bolje obnaša, ce imamo v listu drevesa vec trikotnikov, kar pomeni, da je uporaba konveksnih ploskev, kot primitivov v listih drevesa, koristna. Cas za ugotovitev, da do trka ni prišlo je pri vseh razlicicah algoritma enak, saj število trikotnikov v listih drevesa na ta cas nima vpliva. Najboljše rezultate algoritem dosega pri uporabi ploskev z najvec 6 trikotniki. V splošnem dajejo boljše rezultate ploskve z sodim maksimalnim številom trikotnikov. Rezultati testov so odvisni tudi od strukture hierarhicnega drevesa. Torej so rezultati odvisni od oblike objekta in števila trikotnikov, ki ga sestavljajo. To pa pomeni, da bi se morda ob uporabi drugacnih modelov v testu za boljšo izkazala kakšna druga velikost konveksne ploskve.

Drugi test, ki sem ga uporabil za oceno obnašanja algoritma v najslabšem primeru, je test z dvema kroglama. Obe krogli sta sestavljeni iz 1984 trikotnikov, ena od njih je rahlo manjša in postavljena znotraj vecje krogle. Zaradi tega, ker so krogli tako blizu druga drugi, mora algoritem preiskati velik del drevesa preden lahko zakljuci da trka ni. Casi iz tega testa so podani v spodnji tabeli.

Število trikotnikov	1	2	3	4	5	6	7	8	9	10
Cas [ms]	2.614	1.819	1.767	1.266	2.419	1.537	3.030	1.792	3.240	2.590

Tabela 3 : Casi drugega testa (manjša krogla znotraj vecje krogle).

Tudi tu dajejo razlicice s sodim najvecjim številom trikotnikov (z izjemo ploskev najvecje velikosti 10) boljše rezultate. Od ploskev z lihim najvecjim številom trikotnikov dajejo dobre rezultate le ploskve najvecje velikosti 3. Torej je tudi v tem primeru boljša uporaba sodega najvecjega števila trikotnikov na ploskvi.

3.5.2. Zakljucki

V CD se uporabljajo ploskve z najvec šestimi trikotniki na ploskvi. Pri drugem testu se sicer bolje obnesejo ploskve najvecje velikosti štiri, vendar je ta testna situacija skrajni primer, zato je izbira bolj temeljila na rezultatih prvega testa. Naslednja vecja izboljšava algoritma bi bila uporaba casovne koherence. S tem bi algoritem porabil manj casa za preiskovanje drevesa v zaporednih testih dveh objektov. Trenutno algoritem v primeru trka porabi 60-80% vsega casa za preiskovanje drevesa. Izboljšave so možne tudi pri preiskovanju ploskev, saj se trenutna verzija približuje paru najbližjih (sekajocih) trikotnikov le z ene strani (ene ploskve). S približevanjem z obeh ploskev bi se cas za odkritje trka zmanjšal. Zadnje podrocje izboljšav pa je seveda optimizacija kode.

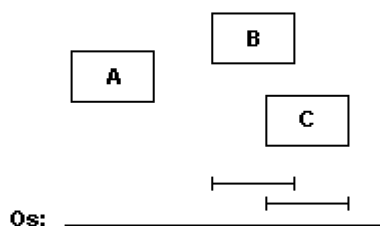
4. Uporaba algoritmov za odkrivanje trkov

4.1. Ugotavljanje potrebnih testov

Ko imamo na voljo algoritem, ki zna odkrivati trke med dvema objektoma moramo poskrbeti tudi za ugotavljanje potrebnih testov med objekti v okolju. Objektov je lahko zelo veliko in bilo bi casovno potratno, ce bi vsakic delali test, ceprav je ocitno da se objekta ne sekata. Prva pohitritev, ki nam pride na misel je da testov med dvema staticnima objektoma ni potrebno izvajati. Torej imamo najvec $(N(N\text{-}1)(N\text{-}3)...(3)(1)) + NM$ potrebnih testov, pri cemer je N število premikajocih in M število staticnih objektov. V nadaljevanju bodo opisani nekateri pristopi pri ugotavljanju potrebnih testov.

4.1.1. Sortiranje osi

Pri temu tipu ugotavljanja potrebnih testov vzamemo seznam vseh objektov v okolju in jih projiciramo na neko izbrano os. Pri tem dobimo intervale, ki pripadajo mejnim volumnom objektov. Ce se dva tako dobljena intervala na izbrani osi prekrivata potem je potrebno narediti natancnejši test med pripadajocima objektoma. Algoritem se zelo dobro obnaša pri velikih okoljih in okoljih, kjer so objekti dalec narazen. Ce pa je okolje majhno, se hitrost zelo hitro zmanjša na podobno raven, kot ce bi racunali teste med vsemi objekti na sceni.



Slika 21 : Sortiranje osi, potreben je test med objektoma B in C.

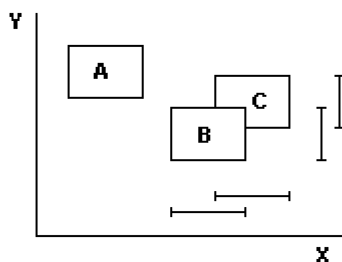
4.1.2. Mreža

Ta metoda uporablja vec informacije o poziciji objektov v okolju. Lahko uporabljamo 2D ali 3D mreže. Velikost celic v mreži mora biti takšna, da je vecja

od najvecjega objekta. To zagotovi, da so objekti, ki lahko trcijo, najvec eno celico narazen. Pozicija objektov v mreži se ugotavlja glede na center mejnega volumna (AABB ali krogla), ki ga obdaja, kar ima za posledico to, da v sosednjo celico sega najvec polovica objekta. Iz tega sledi, da se objekti, ki se nahajajo dve celici narazen nikoli ne dotikajo. Pohitritev algoritma lahko dosežemo z uporabo casovne koherence. Ker lahko pricakujemo, da bo objekt redko menjaval celico v kateri se nahaja, lahko delamo popravke v seznamu potrebnih testov samo, ko je to potrebno - eden izmed objektov preide v drugo celico. V nadaljevanju sledi nekaj znacilnosti algoritma za 2D mrežo. Izracun potrebnih testov za 2D mrežo poteka po naslednjih dveh pravilih:

(a) Ce je v celici vec kot en objekt, naredimo obicajen test na vseh teh objektih.
(b) Ce celica ni prazna, in je neka celica desno od nje ali pod njo tudi neprazna je potrebno narediti test med objekti, ki so v teh dveh celicah.

Algoritem zacne v zgornjem levem kotu in preverja samo celice desno in pod trenutno celico. Tako se pomika do konca proti desnemu spodnjemu kotu. S tem se izogne dvakratnemu preverjanju celic. Algoritem za 3D mrežo je podoben, s tem da je potrebno upoštevati še tretjo dimenzijo in s tem posledicne spremembe pravil in nacina preiskovanja mreže.



Slika 22 : Primer 2D mreže, potreben je test med B in C.

4.1.3. Primerjanje intervalov na X, Y in Z osi

Ta algoritem vzdržuje seznam aktivnih parov objektov (objekti, za katere je potrebno opraviti test) ter urejen seznam intervalov za X, Y in Z os. Za mejni volumen se uporablja AABB. Vsak par AABB-jev ima tri zastavice. Te povedo, ali se prekrivata na X, Y oziroma Z osi. Ce so nastavljene vse tri zastavice, se

AABB-ja prekrivata. Glede na vrednosti zastavic imamo eno izmed naslednjih treh situacij:

(a) Vse tri dimenzije AABB-jev se prekrivajo, pripadajoc par objektov se doda v seznam aktivnih parov objektov.
(b) AABB-ja sta se prej prekrivala, sedaj pa se ne vec. Pripadajoci par objektov se odstrani iz seznama aktivnih objektov.
(c) par AABB-jev se tudi sedaj ne prekriva.

Ko je seznam aktivnih objektov dokoncan, se izvedejo vsi testi, ki so shranjeni v njem. Ta algoritem je uporabljen v I-COLLIDE in je natancneje opisan v [6].

4.1.4. Upoštevanje clovekove percepcije

Ta metoda se opira na dokaj neraziskano podrocje clovekove percepcije okolja. Vecja prioriteta je dana objektom, ki jih prej opazimo (so vecji, bolj blizu). Lahko se upošteva še druge dejavnike. Algoritem uporablja preprost model clovekove percepcije trkov, o katerem vec piše v [4]. Shranjujeta se dva seznama: seznam aktivnih trkov (objekti, katerih mejni volumni se prekrivajo in potrebujejo test) ter seznam resnicnih trkov (objekti, ki so dejansko trcili). Seznam aktivnih trkov se polni z objekti katerih mejni volumni se prekrivajo. Ko je test koncan se pripadajoce objekte odstrani iz seznama, pri cemer damo objekte ki so dejansko trcili v seznam resnicnih trkov. Za razvršcanje testov so možni razlicni pristopi, dva izmed njih sta:

(a) Percepcijsko urejeno sekvencno razvršcanje (*perceptualy sorted sequential scheduling*) - vsakemu testu se priredi prioriteto izracunano glede na model clovekove percepcije trkov. Testi na koncu vrste se izvedejo le ce je na voljo dovolj casa.
(b) Razvršcanje s prioritetnimi cakalnimi vrstami (*priority queue scheduling*) - vec vrst znotraj katerih se izvaja round robin algoritem. Model clovekove percepcije je uporabljen za odlocitev v katero vrsto test spada.

Drugi seznam, ki ga vodi algoritem se uporablja pri racunanju odziva objektov na trk. Ta metoda je opisana v [4].

4.2. Odziv objekta na trk

Obicajno želimo imeti odziv objekta na trk. Pri tem je realnost odziva odvisna od rezultata, ki ga je vrnil algoritem, ter natancnosti fizikalnega modela v aplikaciji. Preprost fizikalni model vkljucuje vsaj podatke o masi ter hitrosti objekta, lahko tudi o pospešku. Podatki o hitrosti in pospešku so podani kot vektor, ki nam pove hitrost oz. pospešek objekta v treh smereh (X, Y in Z). Iz rezultata algoritma za odkrivanje trkov izracunamo smeri v katerih se bosta objekta odbila (preprost pristop za izracun smeri je razlika med geometrijskim centrom objekta in tocko trka), nato pa z upoštevanjem gibalne kolicine izracunamo še hitrosti objektov po trcenju. Boljši fizikalni modeli upoštevajo še vsaj naslednje sile:

(a) sila težnosti
(b) upor zraka
(c) sila lepljenja
(d) sila trenja

Vec o fizikalnih modelih za preproste igre je napisanega v [10].

4.3. Primer Igre

Za prikaz delovanja našega algoritma za odkrivanje trkov (CD) sem naredil preprosto igrico – zaletavanje avtomobilckov. Igralec se mora v 30 sekundah cimveckrat zaleteti. V igrici so uporabljeni nekateri pristopi opisani v tem poglavju.

Slika 23 : Preprosta igra - zaletavanje avtomobilckov.

4.3.1. Podatkovna struktura

Deli igre, ki nas zanimajo so v razredih *CCollidableMesh* in *CGrid*. *CCollidableMesh* vsebuje vse potrebne podatke o modelu na sceni. To so podatki o teži, hitrosti in podatkovne strukture potrebne za odkrivanje trkov. Preverjamo lahko ali je model trcil in se odzivamo na trk (nastavljanje smeri gibanja, hitrosti). Odzivom na trk je posveceno podpoglavje 4.3.3. *CGrid* je razred, ki skrbi za ugotavljanje potrebnih testov, implementira preprosto 2D mrežo, vec o delovanju mreže v podpoglavju 4.3.2.

4.3.2. Ugotavljanje potrebnih testov

Igra uporablja 2D mrežo velikosti 3x3. Vsakic, ko se eden izmed avtomobilckov premakne (naš ali racunalnikov) se izracuna njegova pozicija v mreži, algoritem je v osnovi tak:

```
procedure SaveGridPosition(object);
begin
  for each_cell_in_grid do
  begin
    if IsInCell(object) then
      StorePosition(Object);
  end;
end;
```

Pri tem se preiskovanje zacne v zgornjem levem kotu in se konca takoj ko najdemo celico v kateri se avto nahaja.

Preden zacnemo izvajati potrebne teste med avtomobili, se za avto, ki ga želimo testirati, ugotovi kateri testi so potrebni. Psevdo koda tega algoritma:

```
procedure DetermineTests(object);
begin
  cells = DetermineCellsToCheck(object);
  for cells do
  begin
    if NeighbourFound then begin
      StoreInObjectList(Neighbour);
      inc(numberOfTestNeeded);
    end;
  end;
end;
```

Nato se izracuna cas ki je dovoljen za izvedbo posameznega testa. Le ta se izracuna na podlagi seznama potrebnih preverjanj in maksimalnega možnega števila še potrebnih testov. Dovoljeni skupni cas za vse teste je *TOTALTIME*, in je v demo igrici enak 30 milisekund. Ker imamo 4 avtomobile, torej najvec 6 razlicnih trkov, to pomeni v prvem koraku (primerjanje našega avtomobila z racunalnikovimi) *TOTALTIME/(3+numberOfTestNeeded)*. Dovoljen cas je torej najvec *TOTALTIME/3* (vendar v tem primeru ni potrebno narediti nobenega testa) in najmanj *TOTALTIME/6*. V drugem koraku (primerjanje enega racunalnikovega avtomobila s preostalimi racunalnikovimi avtomobili) se dovoljeni cas izracuna s *TOTALTIME/(4–numberOfTestNotDoneInFirstStep+numberOfTestNeeded)*. Tu je na voljo najvec *TOTALTIME* casa (primer, ko naš avtomobil ni imel nobenega testa in tudi za izbrani racunalnikov ni potrebnih testov) in najmanj *TOTALTIME/6*. V zadnjem koraku je dovoljeni cas izracunan s naslednjo formulo: *TOTALTIME/(6–numberOfTestNotDoneBefore)*. Tukaj imamo od *TOTALTIME/6* pa do *TOTALTIME* casa. V vsakem koraku se takoj za izracunom casa izvedejo še testi. Test je prekinjen, ce je prekoracen dovoljeni cas za izvedbo testa. V tem primeru se predpostavi, da trk obstaja, tocka trka pa je ena izmed tock na mejnem volumnu.

Namen opisane uporabe dovoljenega casa, je zgolj prikaz možnega nacina uporabe, saj je CD dovolj hiter da na poizvedovanja odgovori pred iztekom dovoljenega casa. Igra trkov s prizorišcem ne preverja, ker je le to okroglo, tako

da je ta del rešen s tem da se avto ne more pomakniti zunaj kroga z nekim radijem (radijem arene na prizorišcu). O odzivanju na trke pa v naslednjem podpoglavju.

4.3.3. Odziv na trke

Odziv na trke je implementiran v *CCollidableMesh*. Upošteva se gibalna kolicina avtomobilov. Psevdo koda tega dela igrice:

```
...
point = ReadPointOfCollision

dir1 = Object1.Position - point;
Normalize(dir1);

dir2 = Object2.Position - point;
Normalize(dir2);

momentum1 = Length(Object1.Speed) * Object1.mass;

momentum2 = Length(Object2.Speed) * Object2.mass;

SetSpeed(Object1,(momentum2/Object1.mass)*dir1);
SetSpeed(Object2,(momentum1/Object2.mass)*dir2);
...
```

Najprej se prebere tocka trka, ki je potrebna da izracunamo smer odboja za oba avtomobila. Nato se izracuna trenutna gibalna kolicina avtomobilov in na osnovi tega se izracuna nova hitrost avtomobila v X, Y in Z smeri.

Literatura

[1] Efficient collision detection for interactive 3D graphics and virtual enviroments (James Thomas Klosowski; State University of New York at Stony Brook; 1998)

[2] Efficient collision detection of complex deformable models using AABB trees (Gino Van Den Bergen; Eindhoven University of Technology; 1998)

[3] OBBTree: A hierarchical structure for rapid interference detection (S. Gottschalk, M. C. Lin, D. Manocha; University of North Carolina; 1996)

[4] Real-time collision detection and response using sphere-trees (O'Sullivan, C. Dingliana; J. Image Synthesis group, Trinity College Dublin)

[5] Accurate and fast proximity queries between polyhedra using convex surface decomposition (Stephen A. Ehmann, Ming C. Lin; University of North Carolina; 2001)

[6] I-COLLIDE: An interactive and exact collision detection system for large scale enviroments (Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, Madhav K. Ponamgi; University of North Carolina)

[7] An efficient collision detection algorithm for polytopes in virtual enviroments (Kelvin Chung Tat Leung; University of Hong Kong; 1996)

[8] The QuickHull algorithm for convex hulls (C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa; 1996)

[9] ERIT – A collection of efficient and reliable intersection tests (Martin Held; Institut fur Computerwissenschaften Universitat Salzburg)

[10] Beginning DIRECT3D game programming (Wolfgang F. Engel, Amir Geva; Prima Tech's game development series; 2001)

[11] Collision detection : Algorithms and applications (Ming C. Lin, Dinesh Manocha, Jon Cohen, Stefan Gottschalk; Army Research Office, University of North Carolina)

[12] http://www.acm.org/jgt/papers/Moller97/tritri.html (Tomas Moller, 1997)

Dodatek: CD API

Za uporabo knjižnice v programu je potrebno vkljuciti datoteko *CollisionObject.h*. Knjižnica ima preprost vmesnik. Glavni razred v knjižnici je *CCollisionObject*, ki vsebuje naslednje metode:

```
void addTriangle(D3DXVECTOR3 v1,D3DXVECTOR3 v2,
                 D3DXVECTOR3 v3)
void processTriangles()
void SetTransformMatrix(D3DXMATRIX* m)
bool CheckCollision(CCollisionObject* other,int timeout=0)
void GetCollisionPoint(D3DXVECTOR3 *point[MaxPointNum],
                       int* pointsDetected)
```

Opis metod sledi v nadaljevanju. Drugi razred ki je viden uporabniku je *CInconsistency*, njegov namen pa je porocanje napak. Tak primer je, ko uporabnik pozabi na klic metode *processTriangles*, nato pa klice metodo *CheckCollision*. Zgodila se bo izjema *CInconsistency* in uporabnik jo lahko ulovi ter reši problem. Primer uporabe v kodi:

```
...
try {
  if (obj1->CheckCollision(obj2)) {
    D3DXVECTOR3 *p[MaxPointNum], temp;
    int num;

    cout << "collision!\n";
    obj1->GetCollisionPoint(p,&num);
    cout << "Number of detected points: " << num << "\n";
    for (int i=0; i<num; i++) {
      temp = *p[i];
      cout << "P" << i <<": x: " << temp.x << " y: " <<
      temp.y << " z: " << temp.z << "\n";
      cout.flush();
      delete *p[i];
    }
  } else cout << "no collision!\n";
}
catch (CInconsistency) {
  cout << "Data error!\n";
}
...
```

Vmesnik vsebuje še eno funkcijo to je:

```
CCollisionObject* newCollisionObject()
```

Ta funkcija nam vrne instanco objekta *CCollisionObject*. To je prvi klic v programu, ki ga moramo narediti ce želimo uporabljati metode v *CCollisionObject*. Primer klica:

```
...
CCollisionObject* obj = newCollisionObject();
...
```

V nadaljevanju sledi opis metod v razredu *CCollisionObject*.

void addTriangle(D3DXVECTOR3 v1, D3DXVECTOR3 v2,
D3DXVECTOR3 v3)

Opis parametrov:

D3DXVECTOR3 v1,
D3DXVECTOR3 v2,
D3DXVECTOR3 v3 : tocke trikotnika, pomemben je vrstni red tock, ker algoritem potrebuje informacijo o normali ploskve na kateri se nahaja trikotnik.

Opis delovanja:

Metodo se uporablja za dodajanje trikotnikov v podatkovno strukturo, ki jo uporablja *CCollisionObject*. Za primer uporabe glej primer pri metodi *processTriangles*.

void processTriangles()

Opis delovanja:

Ta metoda se mora obvezno klicati potem, ko so bili vsi trikotniki s klicem metode *addTriangle* shranjeni v podatkovno strukturo, ki jo uporablja *CCollisionObject*. Primer uporabe:

```
HRESULT CCollidableMesh::Create(LPDIRECT3DDEVICE8 pd3dDevice,
        TCHAR* strFilename,
                                bool createtree)
{
  HRESULT rc=CD3DMesh::Create(pd3dDevice,strFilename);
  //read triangles to tree structure
  if (SUCCEEDED(rc) && createtree) {
    m_ColModel = newCollisionObject();
    LPD3DXMESH clone;
    if (SUCCEEDED(m_pSysMemMesh->CloneMeshFVF
                  (D3DXMESH_SYSTEMMEM,D3DFVF_XYZ,
                   pd3dDevice,&clone))) {
      int vnum=clone->GetNumVertices();
      int fnum=clone->GetNumFaces();
      CUSTOMVERTEX* v;
      if (SUCCEEDED(clone->LockVertexBuffer
                    (D3DLOCK_READONLY,(BYTE**)&v))) {
        WORD* ind;
        if (SUCCEEDED(clone->LockIndexBuffer
                      (D3DLOCK_READONLY,(BYTE**)&ind))) {
          for(int i=0;i<fnum;i++) {
            m_ColModel->addTriangle((float*)&v[ind[i*3+0]],
                                    (float*)&v[ind[i*3+1]],
                                    (float*)&v[ind[i*3+2]]);
          }
          m_ColModel->processTriangles();
          clone->UnlockIndexBuffer();
        }
        clone->UnlockVertexBuffer();
      }
      clone->Release();
    }
  }
  return rc;
}
```

void SetTransformMatrix(D3DXMATRIX* m)

Opis parametrov:

D3DXMATRIX m* : matrika velikosti 4x4, ki vsebuje pozicijo in rotacijo objektovega lokalnega koordinatnega sistema glede na koordinatni sistem okolja.

Opis delovanja:

Vsakic, ko se spremeni pozicija ali orientacija objekta v prostoru je potrebno s klicem te metode nastaviti interno transformacijsko matriko. Rezultati odkrivanja trkov so nezanesljivi, ce tega ne storimo. Primer uporabe:

```
...
D3DXMATRIX tr;

tr._11=1.0f; tr._12=0.0f; tr._13=0.0f; tr._14=0.0f;
tr._21=0.0f; tr._22=1.0f; tr._23=0.0f; tr._14=0.0f;
tr._31=0.0f; tr._32=0.0f; tr._33=1.0f; tr._14=0.0f;
tr._41=-5.0f; tr._42=-5.0f; tr._43=-5.0f; tr._14=1.0f;

obj->SetTransformMatrix(&tr);
...
```

bool CheckCollision(CCollisionObject* other, int timeout = 0)

Opis parametrov:

CCollisionObject other* : kazalec na drugi objekt, katerega želimo preveriti ali je trcil s tem objektom.
int timeout : cas, ki je na voljo za odkritje trka, ce cas ni podan je na algoritmu na voljo toliko casa kot ga potrebuje da natancno doloci rezultat.

Rezultat:

true ce sta objekta trcila, interno se shranijo tocke kjer sta objekta trcila, drugace je rezultat *false*.

Opis delovanja:

To je funkcija, ki preveri ali sta objekta trcila. Pri klicu lahko podamo cas, ki je na voljo da nam vrne rezultat. V primeru ko algoritem v tem casu ni uspel vrniti rezultata se vrne rezultat *true*, kot tocke trka pa se vzamejo dve tocki z mejnega volumna, ki vsebuje trenutno vozlišce. Ce cas ni podan se bo algoritem izvršil do konca in vrnil *true* oziroma *false*, ce objekta nista trcila. Primer uporabe:

```
...
timeout = 10; //10 ms timeout
//call with timeout setting...
bool Colliding1 = m_ColModel->CheckCollision(other->m_ColModel,
                                             timeout);
//...and without it...
bool Colliding2 = m_ColModel->CheckCollision(other->m_ColModel)
...
```

void GetCollisionPoint(D3DXVECTOR3 *point[MaxPointNum], int* pointsDetected)

Opis parametrov:

*D3DXVECTOR3 *point[MaxPointNum]* : polje kazalcev na strukturo D3DXVECTOR3, ki vsebuje tocke trka. *MaxPointNum* je definiran v *CollisionObject.h* in doloca maksimalno število tock, ki jih metoda *CheckCollision* poišce.
int pointsDetected : kazalec na število tock trka, ki jih je algoritem našel.

Opis delovanja:

Ta funkcija se uporablja, ce želimo izvedeti, kje sta objekta trcila. Primer uporabe:

```
...
m_ColModel->GetCollisionPoint(p,&num);

D3DXVECTOR3 dir1=D3DXVECTOR3(m_Transform(3,0),m_Transform(3,1),
                             m_Transform(3,2)) - *p[0];
dir1 /= D3DXVec3Length(&dir1);

float momentum1 = D3DXVec3Length(&m_Speed) * m_Mass;
float momentum2 = D3DXVec3Length(&other->m_Speed) * other->m_Mass;

D3DXVECTOR3 dir2=D3DXVECTOR3(other->m_Transform(3,0),
                             other->m_Transform(3,1),
                             other->m_Transform(3,2)) - *p[0];
dir2 /= D3DXVec3Length(&dir2);

SetSpeed((momentum2/m_Mass)*dir1); //first object
other->SetSpeed((momentum1/other->m_Mass)*dir2); //second object
...
```

Izjava

Izjavljam, da sem diplomsko delo samostojno izdelal pod vodstvom mentorja prof. dr. Saša Divjaka. Izkazano pomoc drugih sodelavcev sem v celoti navedel v zahvali.

Aleš Daneu