SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1739

STRUKTURE PODATAKA ZA DETEKCIJU SUDARA

Robert Kurtanjek

Zagreb, travanj 2008.

1. Sadržaj

2. Uvod

Detekcija sudara je vrlo važna u simulaciji dinamičkih scena. Problem izračuna detekcije sudara između *n* pokretnih objekata je problem složenosti $O(n^2)$, tj. točnije, uz n objekata, potrebno je evaluirati $\frac{n \bullet (n-1)}{2}$ potencijalnih kolizijskih parova.

Klasičan pristup za ubrzanje tih kalkulacija je korištenje algoritma od dva koraka. U prvom koraku, poznatom kao *široka faza*, aproksimativni i brzi test pronalazi potencijalne kolizijske parove koji se u drugom koraku, poznatom kao *uska faza*, točno provjeravaju za koliziju.

Kako široka faza algoritma mora biti vrlo brza, testovi se obično rade između jednostavnijih oblika – omeđujućih volumena, kao što su općeniti kvadri (*eng. OBB - Oriented Bounding Box*), kvadri paralelni sa osima (*eng. AABB - Axis-Aligned Bounding Box*) ili sfere.

Dodatna metoda koja se koristi kod ubrzanja široke faze detekcije sudara je korištenje prostornih struktura podataka, čime se smanjuje broj potencijalnih kolizijskih parova koje moramo provjeriti. Za takve strukture je poželjno da imaju mogućnost prilagodbe različitoj gustoći rasporeda objekata po sceni.

Ovaj rad opisuje nekoliko struktura podataka i algoritama koji se koriste u širokoj fazi detekcije sudara za njeno ubrzanje. Algoritmi su implementirani u 3D prostoru pri čemu se testovi kolizije vrše između kugli, koje predstavljaju omeđujuće volumene složenijih objekata koji se mogu nalaziti u njima.

Implementiran je algoritam klasične detekcije sudara (složenosti $O(n^2)$), zatim prostorna struktura *BSP stablo* [2] (*eng. Binary space partitioning*), prostorna struktura *Samo-podesivo BSP stablo* [1] (*eng. Self-adjusting BSP*) koja se prilagođava različitoj gustoći rasporeda objekata na sceni te algoritam *Brojenja parova i pojednostavljivanja* [3] (*eng. Sweep and prune*). Performanse svih tih metoda su testirane kod međusobne kolizije od 50 objekata do 5.000 pokretnih objekata (tj. omeđujućih volumena – kugli) na sceni i to pri različitim rasporedima objekata. Također je ispitan utjecaj različitog broja poligona na sceni te sjenčanja na brzinu iscrtavanja te zauzeće resursa računala.

3. Opis algoritama i struktura podataka

3.1 Klasična metoda

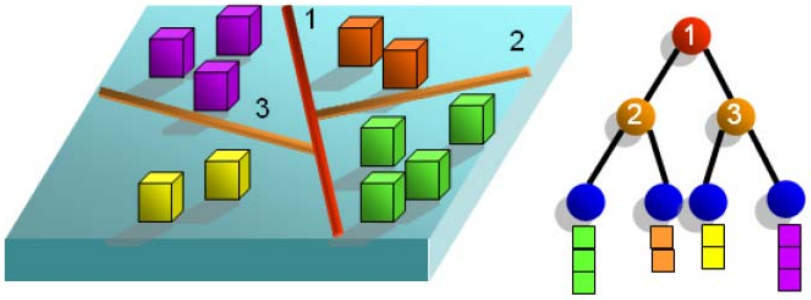
Kod klasične metode detekcije sudara, potrebno je provjeriti za svaki od n objekata u sceni da li je u koliziji sa ostalih n-1 objekata. Dakle, ukupno je za koliziju potrebno provjeriti: $(n-1)+(n-2)+...+2+1=\frac{n\bullet(n-1)}{2}$ parova. Čak i kad je test kolizije trivijalan, kao što je slučaj kod kugli, to postaje problem pri velikom broju objekata. Npr. Kod 10.000 pokretnih objekata na sceni, potrebno je za koliziju ispitati gotovo 50 milijuna parova, i tako u svakom koraku simulacije. Očito je potrebna neka metoda koja bi smanjila broj parova objekata koje ispitujemo za koliziju.

3.2 BSP stablo

BSP stablo je standardno binarno stablo koje se koristi kod pretrage i sortiranja objekata u n-dimenzionalnom prostoru. Stablo kao cjelina predstavlja cijeli prostor, a svaki čvor predstavlja konveksni dio tog prostora. Svaki čvor (osim krajnjih čvorova – listova) ima „*hiper-ravninu*“ koja dijeli prostor koji predstavlja na dvije polovine. Svaki čvor također sadrži i reference na ta dva nova čvora. BSP stablo se najčešće koristi u 2D ili 3D prostoru. U tim slučajevima hiper-ravnine su zapravo pravci i ravnine.

Dakle, BSP stablo je prostorna struktura podataka kojom se postiže rekurzivna, hijerarhijska podjela n-dimenzionalnog prostora u konveksne segmente.

BSP stablo se u širokoj fazi detekcije sudara koristi u svrhu smanjenja potencijalnih kolizijskih parova koje treba evaluirati. Koliko to smanjenje iznosi, ovisi o dubini samog stabla i uspješnosti odabira što boljih ravnina podjele. Poželjno je da ravnina podjele dijeli prostor na pola, tj. da je sa obje njene strane jednak broj objekata. Također je poželjno da je redundancija pri tome što manja, tj. da je što manji broj objekata koji se nalaze sa obje strane ravnine podjele (zbog toga što ih ravnina siječe).



Slika 3.1 Podjela prostora BSP stablom i samo stablo (objekti su u listovima stabla).

Na slici 3.1 je pomoću BSP stabla dubine 2 broj potencijalnih kolizijskih parova smanjen sa $\frac{11 \bullet (11-1)}{2} = 55$ na $\frac{3 \bullet (3-1)}{2} + \frac{2 \bullet (2-1)}{2} + \frac{2 \bullet (2-1)}{2} + \frac{4 \bullet (4-1)}{2} = 11$ (≈5 puta manje).
Kod većeg broja objekata, dobitak je još veći. Tako, npr. dok kod 10.000 objekata klasičnom metodom imamo oko 5 milijuna parova, uz idealnu podjelu objekata unutar BSP stabla dubine 9, broj parova je: $2^9 \bullet \frac{20 \bullet (20-1)}{2} = 97.280$ (≈50 puta manje).

Dubina BSP stabla se postavlja na temelju broja objekata u sceni, tako da je kod idealne raspodjele objekata u stablu u svakom listu najviše 30 objekata, a kod manjeg broja objekata radi se i više podjela kako bi se moglo vidjeti kako radi balansiranje stabla kod samo-podesivog BSP stabla. Kada bi se broj objekata u sceni mijenjao tijekom izvođenja simulacije, mogla bi se dodati dinamička promjena dubine stabla. Na slici 3.2 su zadane dubine stabla u ovisnosti o broju objekata na sceni.

Broj objekata na sceni	Dubina BSP stabla
Broj objekata < 10	1
10 ≤ Broj objekata < 50	2
50 ≤ Broj objekata < 250	3
250 ≤ Broj objekata < 500	4
500 ≤ Broj objekata < 1000	5
1000 ≤ Broj objekata < 2000	6
2000 ≤ Broj objekata < 4000	7
4000 ≤ Broj objekata < 8000	8
Broj objekata ≥ 8000	9

Slika 3.2 Ovisnost dubine BSP stabla o broju objekata na sceni.

Glavni nedostatak BSP stabla u primjeni kod široke faze detekcije sudara u dinamičnim scenama je taj što se raspored objekata u sceni tijekom vremena mijenja, te početno BSP stablo, koje je statična struktura, ne daje jednaki faktor ubrzanja kao što je davalo na početku. Zbog toga su potrebna proširenja BSP stabla, koja ga prilagođavaju promjenjivom rasporedu objekata na sceni.

3.3 Samo-podesivo BSP stablo

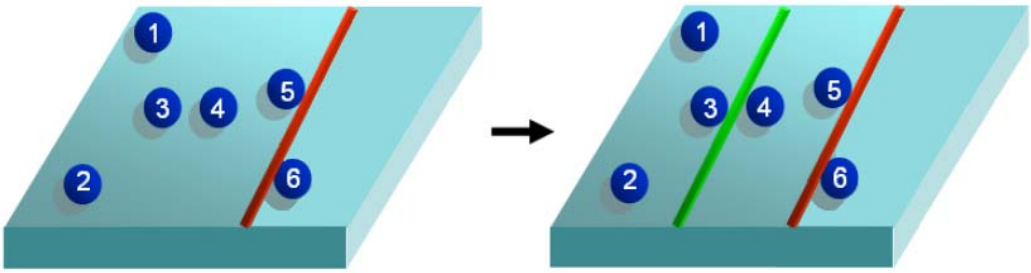
Sve što je rečeno za BSP stablo u prošlom dijelu vrijedi i za *samo-podesivo BSP stablo*. Jedini dodatak je taj što se nad BSP stablom periodički, nakon vremena T, izvodi funkcija podešavanja BSP stabla. Ona prilagođava oblik stabla novom rasporedu objekata u sceni, kako bi se dobio što bolji raspored objekata po listovima stabla.

Potrebno je vršiti balans između želje za odabirom što boljih ravnina podjele i cijene funkcije kojom se takve ravnine biraju, zbog toga što će se ta funkcija vrlo često pozivati tijekom izvođenja simulacije. Zbog toga, i dodatnih

pojednostavljenja koja takva odluka donosi, moguće ravnine podjele sam limitirao na *x-y*, *y-z* te *x-z* ravnine. Korijen stabla počinje sa *y-z* ravninom podjele (ravnina crvene boje kod prikaza stabla), nakon čega slijedi podjela pomoću *x-z* ravnine (zelena boja) te zatim *x-y* ravnine (plava boja). Dobiveni konveksni segmenti prostora se zatim dijele ponovo pomoću *y-z* ravnine, itd. Stablo kod kojeg su ravnine podjele limitirane na taj način se zove kd-stablo (eng. *kd-tree, k-dimensional-tree*) i ono je specijalni slučaj BSP stabla, no u nastavku ću koristiti isključivo naziv BSP stablo.

Nakon što je odabrana vrsta ravnine podjele, samo treba odabrati njen položaj. Kod statičnog BSP stabla, koje se kreira na početku i ne podešava tijekom izvođenja, kao položaj ravnine se odabire onaj koji područje dijeli po pola, bez obzira na trenutni raspored elemenata, i ta podjela se ne mijenja.

Kod *samo-podesivog BSP stabla* svakih T = 0,2s se vrši podešavanje stabla. Ono se sastoji od balansiranja čvorova koji su nebalansirani, tj. odabira novih ravnina podjele za takve čvorove. Čvor se smatra nebalansiranim ako je omjer objekata sa jedne i druge strane njegove ravnine podjele manji od 0,5 (ili recipročno,veći od 2,0).



Slika 3.3 Lijevo – nebalansirani čvor; Desno – nova os balansira čvor.

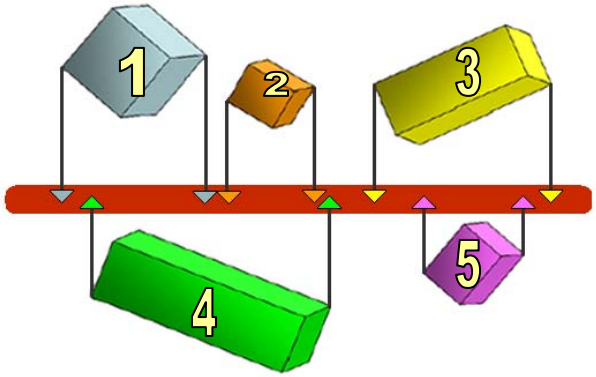
Kod podešavanja stabla, u sve čvorove stabla koji nisu listovi se spremaju objekti, i kada se usporedbom broja objekata utvrdi da neki čvor nije balansiran, za njega i njegovu djecu se odabiru nove osi podjele. Na slici 3.3 se može vidjeti kako radi funkcija balansiranja čvora u 2D. Kako navedeni čvor ima samo 2 lista, balansiranje je završeno.

Kod odabira nove ravnine podjele u čvoru, sortiramo listu objekata koja se nalazi u tom čvoru (i to po x, y ili z osi, ovisno o tome da li je trenutna ravnina podjele u *y-z*, *x-z* ili *x-y* ravnini) i nakon toga se kao položaj ravnine odabire koordinata medijana liste objekata.

Ako čvor nad kojim smo izvršili odabir nove ravnine ima djecu, za svu njegovu djecu se nova ravnina podjele odabire kao sredina konveksnog segmenta prostora koji odgovara pojedinom djetetu. Na taj način se zbog jednostavnog odabira ravnine podjele (nije potrebno sortirati objekte) znatno smanjuje vrijeme koje se troši na balansiranje stabla, no zadržava se prilagodljivost promjenjivom rasporedu objekata na sceni. Ako odabir nove (srednje) ravnine za dijete balansiranog čvora nije balansiralo i dijete, nakon vremena T će se izvršiti preciznije balansiranje (pomoću medijana liste).

3.4 Algoritam brojenja parova i pojednostavljivanja

Algoritam brojenja parova i pojednostavljivanja se koristi u širokoj fazi detekcije sudara, kao i BSP stablo, u svrhu smanjenja potencijalnih kolizijskih parova koje treba evaluirati. Algoritam se sastoji od dvije faze. Prva je prikazana na slici 3.4.



Slika 3.4 Prva faza algoritma brojenja parova i pojednostavljivanja.

Prva faza (faza brojenja parova) počinje projekcijom minimuma i maksimuma omeđujućih volumena (tj. ploha u slučaju 2D) na sve osi (X, Y, Z u slučaju 3D ili samo X, Y u slučaju 2D). Kako se u ovom radu kao omeđujući volumeni koriste kugle, koje sve imaju isti radijus, dovoljno je projicirati na svaku os samo središte kružnice. Zatim se po svim osima projekcije središta sortiraju, te se linearnim algoritmom broje parovi objekata koji se preklapaju. Tako bi za scenu navedenu na prošloj stranici, te zadanu os dobili sljedeće parove objekata: (1,4), (2,4), (3,5). Postupak je potrebno ponoviti i za sve ostale osi. Dobiveni parovi pojedine osi se spremaju u zasebnu listu parova koju ima svaka os.

U drugoj fazi algoritma (fazi pojednostavljivanja) se vrši pojednostavljivanje, tj. dodatna eliminacija parova. Za to se koristi kvadratna matrica dimenzije jednake broju objekata. Svi elementi te matrice se na početku postave na nulu, a zatim, kad se po pojedinoj osi detektiraju parovi, u matrici se inkrementira element čiji redak i stupac je jednak indeksima dvaju objekata koji se po toj osi preklapaju. Pri tome se treba pobrinuti da je indeks prvog elementa manji od indeksa drugog – tako se zapravo koristi samo gornja trokutasta matrica. Za scenu navedenu na slici 3.4, te zadanu os dobili bi sljedeću matricu:

0	0	0	1	0
0	0	0	1	0
0	0	0	0	1
0	0	0	0	0
0	0	0	0	0

Slika 3.5 Matrica kolizijskih parova

Postupak treba napraviti za sve osi. Stvarna provjera kolizije se vrši nakon toga, i to za sve elemente matrice u kojima piše da postoji poklapanje po svim osima (2 za 2D, tj. 3 za 3D). Pri tome se ne čita cijela matrica (za 10.000 objekata to bi bilo 100 milijuna elemenata matrice), pa ni gornja trokutasta (≈50 milijuna elemenata) već samo parovi one osi koja ima najmanji broj preklapanja (elemenata u listi parova te osi).

Prije sljedećeg koraka simulacije potrebno je sve elemente matrice koji su različiti od nula postaviti na nulu – to se također radi pomoću liste parova pojedine osi.

Očit nedostatak ovog algoritma je veliko zauzeće memorije pri velikom broju objekata, zbog kvadratne matrice čija je dimenzija jednaka broju objekata. Npr. uz korištenje podatkovnog tipa *char* za pojedine elemente matrice, kod 10.000 objekata je potrebno alocirati 100 milijuna bajtova memorije (≈100MB memorije).

4. Opis implementacije

Za izradu programskog dijela rada korišteno je programsko okruženje Microsoft Visual Studio 2005, te kombinacija programskih jezika C i C++ uz dodatak OpenGL [6] [7] standardnih biblioteka kako bi se omogućio prikaz korištenjem tog grafičkog standarda. Dodatne biblioteke za izradu sučelja nisu korištene, već je korišten samo Win32 API [5].

Sva alokacija memorije se izvodi tijekom generiranja objekata, a sve varijable se prenose preko referenci a ne vrijednosti. Time je postignuto dodatno ubrzanje u izvođenju programa.

Kao spremnička klasa raznih lista objekata koje se koriste u algoritmima koristi se klasa *vector* iz standardne biblioteke predložaka (*eng. STL, Standard Template Library*). Omogućuje lako dodavanje objekata (metoda *push_back*), brisanje sadržaja (metoda *clear*), pristup elementima pomoću operatora „[]“, te što je najvažnije, sortiranje elemenata pomoću *STL* funkcije *sort*.

4.1 Klasična metoda

Kod klasične metode ne koriste se posebni algoritmi, već se jednostavno u petlji provjeravaju svi mogući parovi objekata na sceni za koliziju. Sami objekti se nalaze u polju objekata koje je na početku cijelo alocirano i napunjeno sa generiranim objektima. Tako se postupak detekcije sudara može prikazati sljedećim kodom:

```
// provjera svih parova objekata
for(i = 0; i < brojObjekata - 1; i++){
      for(j = i+1; j < brojObjekata; j++){

            // provjera udaljenosti središta kugli
            dx = obj[j].pos.x - obj[i].pos.x;
            dy = obj[j].pos.y - obj[i].pos.y;
            dz = obj[j].pos.z - obj[i].pos.z;
            distance = sqrt(dx*dx + dy*dy + dz*dz);

            // ako ima kolizije
            if(distance <= 2*objekt::r){
                  // proračun novih brzina
                  ElasticniSudar(obj[i].pos,obj[j].pos,
                        obj[i].v, obj[j].v);
            }
      }
}
```

Reakcija na sudar je implementirana kao savršeno elastičan sudar između dvije kugle koje su u kontaktu. Ako bi se željelo provjeriti kako bi na brzinu

iscrtavanja utjecao složeniji proračun reakcije na sudar, u dijelu nakon proračuna novi brzina bi se moglo staviti proizvoljno kašnjenje.

4.2 BSP stablo

BSP stablo je implementirano pomoću sljedeće podatkovne strukture:

```
class BSPtree{
public:
	BSPtree *lijevo_dolje, *desno_gore;
	bool list;
	char vrstaRavnine;
	float ravnina, xmin, xmax, ymin, ymax, zmin, zmax;
	int trenutnaDubina;
	std::vector<objekt> L;
}
```

Članovi `BSPtree` klase su:

`BSPtree *lijevo_dolje, BSPtree *desno_gore`
: Pokazivači na dva djeteta trenutnog čvora. Ako je trenutni čvor list, oni iznose `NULL`.

`bool list`
: Označava da li je čvor list ili ne.

`char vrstaRavnine`
: Znakovna varijabla koja označava vrstu ravnine podjele trenutnog čvora ('x' za *y-z* ravninu, 'y' za *x-z* ravninu, 'z' za *x-y* ravninu).

`float ravnina`
: Položaj ravnine (npr. ako je ravnina podjele *x-y* ravnina, tada ova varijabla označava položaj te ravnine po *z* osi).

`float xmin, xmax, ymin, ymax, zmin, zmax`
: Konveksni dio prostora koji pripada trenutnom čvoru.

`int trenutnaDubina`
: Trenutna dubina čvora u BSP stablu (korijenski čvor je na dubini 0).

`std::vector<objekt> L;`
: Lista objekata trenutnog čvora (implementirana pomoću *STL* klase *vector*).

Sljedeće funkcije su također dio `BSPtree` klase:

```
BSPtree();  // konstruktor
void GenerirajStablo(objekt *Lobj);
void KreirajCvor(char vrstaRavnine, int dub);

void OsvjeziPolozajObjekata(objekt *Lobj);
void DodajObjektUList(objekt &obj);
void IsprazniListuObjekata();
```

```
void OdaberiSrednjuOs(float &xmin, float &xmax, float &ymin,
      float &ymax, float &zmin, float &zmax);

void ObrisiCvor(int razina);
void FreeBSPtree();
```

Slijedi kratak opis funkcija `BSPtree` klase:

`GenerirajStablo(objekt *Lobj);`

Na temelju ulaznog parametra (liste objekata) generira BSP stablo. Prvo na temelju broja objekata u listi objekata postavlja dubinu BSP stabla (prema slici 3.2). Zatim postavlja ostale parametre korijenskog čvora, te alocira lijevo i desno dijete korijenskog čvora. Nakon toga se nad lijevim i desnim djetetom pozivaju funkcije `KreirajCvor` i `OdaberiSrednjuOs`, koje kreiraju sve čvorove do (i uključujući) listova, te za svaki čvor koji nije list postavljaju ravninu podjele.

`KreirajCvor(char vrstaRavnine, int dub);`

Kreira čvor i postavlja sve njegove parametre (osim položaja ravnine podjele). Alocira čvorove djecu (ako postoje) i nad njima poziva istu funkciju.

`OsvjeziPolozajObjekata(objekt *Lobj);`

Prvo funkcijom `IsprazniListuObjekata` briše objekte iz listova BSP stabla te ponovo puni listove BSP stabla sa objektima nakon što se je osvježen njihov položaj (svaki objekt se nakon Δt pomaknuo za $\Delta\mathbf{v}$).

`DodajObjektUList(objekt &obj);`

Dodaje objekt (ulazni parametar) u list BSP stabla. Ako je funkcija pozvana nad čvorom koji nije list, na temelju trenutne ravnine podjele i položaja objekta će se odlučiti da li objekt treba poslati lijevom ili desnom djetetu.

`IsprazniListuObjekata();`

Prazni liste objekata svih čvorova u stablu.

`OdaberiSrednjuOs(float &xmin, float &xmax, float &ymin, float &ymax, float &zmin, float &zmax);`

Na temelju ulaznih parametara određuje položaj ravnine podjele. Npr. ako je `vrstaRavnine = 'x'` (tj. *y-z* ravnina), tada se položaj ravnine određuje kao `(xmin + xmax)/2.0f`.

`ObrisiCvor(int razina);`

Briše sve čvorove koji su na razini zadanoj sa ulaznim podatkom.

`FreeBSPtree();`

Oslobađa memoriju zauzetu pri kreiranju stabla.

Nakon što se funkcijama `GenerirajStablo`, `KreirajCvor` i `OdaberiSrednjuOs` kreira BSP stablo, te u njegove listove postave objekti funkcijom `DodajObjektUList`, stablo je spremno za korištenje. U svakom koraku simulacije se na temelju položaja i brzine gibanja svakog objekta osvježava njihov položaj u BSP stablu. Nakon toga slijedi određivanje

detekcije sudara i novih brzina objekata. To sa radi na isti način kao i kod klasične metode, samo nad manjim skupovima (listama objekata), zbog toga što su međusobno bliski objekti grupirani u listovima. Na taj način se znatno smanjuje broj parova objekata koje moramo provjeriti za koliziju.

4.3 Samo-podesivo BSP stablo

Sve što je rečeno kod opisa implementacije BSP stabla vrijedi i za samo-podesivo BSP stablo. Kako bi se omogućilo podešavanje stabla, dodatno su klasi `BSPtree` dodane funkcije `PodesiStablo`, `BalansirajStablo`, `DodajObjekt` i `OdaberiNajboljuOs`. Slijedi kratak opis navedenih funkcija:

`PodesiStablo(objekt *Lobj);`
: Po stablu raspoređuje objekte na temelju njihovog trenutnog položaja, nakon čega se poziva funkcija `BalansirajStablo`.

`BalansirajStablo();`
: Funkcija se poziva nad čvorom stabla, i broji koliko objekata pripada lijevom i desnom djetetu čvora. Ako je omjer objekata lijevog i desnog čvora manji od 0,5 (ili recipročno,veći od 2,0) čvor nad kojim je funkcija pozvana nije balansiran i za njega se poziva funkcija `OdaberiNajboljuOs` koja odabire bolju ravninu podjele od trenutne. Ako je čvor nad kojim je pozvana funkcija balansiranja balansiran, tada se ne odabire druga ravnina podjele trenutnog čvora, već se nad njegovom djecom poziva funkcija balansiranja (`BalansirajStablo`).

`DodajObjekt(objekt &obj);`
: Dodaje objekt (ulazni parametar) u čvor BSP stabla nad kojim je funkcija pozvana. Ako je funkcija pozvana nad čvorom koji nije list, na temelju trenutne ravnine podjele i položaja objekta će se odlučiti da li objekt treba dodatno poslati lijevom ili desnom djetetu.

`OdaberiNajboljuOs(float &xmin, float &xmax, float &ymin, float &ymax, float &zmin, float &zmax);`
: Na temelju ulaznih parametara funkcije i položaja objekata u stablu određuje novi položaj ravnine podjele za čvor nad kojim je pozvana. Položaj ravnina podjele se određuje na način opisan u poglavlju 3.3. U nastavku je prikazan način određivanja ravnine podjele ako se ona nalazi u y-z ravnini.

```
// x-os
if(this->vrstaRavnine == 'x'){
      sort(this->L.begin(), this->L.end(), compX);

      // ako nema objekata u listi nova najbolja ravnina je srednja
         ravnina
      if(this->L.size() == 0){
            this->ravnina = (this->xmin + this->xmax) / 2.0f;
```

```
	}

	// inače, nova najbolja ravnina je medijan
	else{
		indeksNajboljeOsi = (int)this->L.size() / 2;
		// za neparan broj objekata
		if(this->L.size() % 2) this->ravnina =
			this->L[indeksNajboljeOsi].pos.x;
		// za paran broj objekata
		else this->ravnina = (this->L[indeksNajboljeOsi].pos.x +
			this->L[indeksNajboljeOsi-1].pos.x) / 2.0f;
	}

	// odabir novih osi za djecu
	if(this->lijevo_dolje->list == false)
		this->lijevo_dolje->OdaberiSrednjuOs(xmin,
			this->ravnina, ymin, ymax, zmin, zmax);

	if(this->desno_gore->list == false)
		this->desno_gore->OdaberiSrednjuOs(this->ravnina,
			xmax, ymin, ymax, zmin, zmax);
}
```

Iz navedenog dijela koda se može vidjeti način na koji se odabire novi položaj ravnine podjele. Novi položaj nije nužno najbolji koji se mogao odabrati, no najčešće je dovoljno dobar, i što je također važno, može se izračunati relativno brzo.

U navedenoj funkciji najviše vremena se troši na sortiranje objekata, tj. na sljedeću liniju koda:

```
	sort(this->L.begin(), this->L.end(), compX);
```

Sortiranje objekata je implementirano pomoću funkcije `sort` iz standardne biblioteke predložaka (*eng. STL, Standard Template Library*). Sortira se lista objekata `L`, koja je implementirana pomoću *STL* klase *vector*. Sortiranje se izvodi u ovom slučaju po *x* osi, a usporedba 2 objekta se vrši pomoću funkcije `compX`:

```
	bool compX(objekt &prvi, objekt &drugi){
		return prvi.pos.x < drugi.pos.x;
	}
```

Kako bi se smanjilo vrijeme koje se koristi na podešavanje stabla, odabir položaja novih ravnina podjele za djecu čvora nad kojim je pozvana funkcija `OdaberiNajboljuOs` se računa funkcijom `OdaberiSrednjuOs` koja se izvršava mnogo brže zbog toga što nema potrebe za sortiranjem položaja objekata.

4.4 Algoritam brojenja parova i pojednostavljivanja

Algoritam brojenja parova i pojednostavljivanja je implementiran pomoću sljedećih podatkovnih struktura:

```
class PKP{                      // PKP = Potencijalni Kolizijski Par
      int prvi, drugi;
}
```

```
class SnP{                      // SnP = Sweep and Prune
      char *matricaPKP;

      std::vector<objekt> Lx;
      std::vector<objekt> Ly;
      std::vector<objekt> Lz;

      std::vector<PKP> LxPKP;
      std::vector<PKP> LyPKP;
      std::vector<PKP> LzPKP;
}
```

Pomoćna klasa `PKP` definira potencijalni kolizijski par, tj. sadrži dva člana koji označavaju indekse objekata koji su možda u koliziji (projekcije im se preklapaju po bar jednoj osi).

Klasa `SnP` (od *eng. Sweep and prune*) je glavna klasa koja sadrži članove i metode potrebne za implementaciju algoritma brojenja parova i pojednostavljivanja.

Članovi `SnP` klase su:

`char *matricaPKP;`
: Kvadratna matrica dimenzije jednake broju objekata. Koristi se za detekciju sudara između parova objekata. Vrijednosti svih njenih elemenata se postavljaju na 0, a kad se detektira preklapanje po osi između i-tog i j-tog elementa, inkrementira se vrijednost elementa matrice koji se nalazi u i-tom retku te j-tom stupcu.

`std::vector<objekt> Lx, Ly, Lz;`
: Liste svih objekata u sceni, pri čemu su objekti sortirani na temelju trenutnog položaja i to po x (`Lx`), y (`Ly`) i z osi (`Lz`).

`std::vector<PKP> LxPKP, LyPKP, LzPKP;`
: Liste potencijalnih kolizijskih parova. Određuju se na temelju preklapanja parova objekata po x (`LxPKP`), y (`LyPKP`) ili z osi (`LzPKP`).

Sljedeće funkcije su također dio `SnP` klase:

```
void AlokacijaMatrice();
void Inicijalizacija(objekt *Lobj);
void Sortiranje(objekt *Lobj);
void OdrediParove(objekt *Lobj);
void PripremiZaKoliziju(objekt *Lobj);
void OslobodiMatricu(){ free(matricaPKP); }
void DetekcijaSudaraSnPJednogPara(PKP &a, objekt *Lobj);
```

Slijedi kratak opis navedenih funkcija:

`AlokacijaMatrice();`

Alocira kvadratnu matricu `matricaPKP`.

`Inicijalizacija(objekt *Lobj);`

Iz liste objekata `Lobj` puni liste `Lx`, `Ly` i `Lz` te ih sortira prema x (`Lx`), y (`Ly`) i z osi (`Lz`)

`Sortiranje(objekt *Lobj);`

Osvježava liste `Lx`, `Ly` i `Lz` sa novim položajima objekata te ih zatim sortira. Kako se liste `Lx`, `Ly` i `Lz` ne prazne pa ponovo pune sa svim objektima, već se samo osvježavaju položaji objekata koji se već nalaze sortirani u tim listama, postiže se određeno ubrzanje, jer se sortiraju liste koje su već većinom sortirane.

`OdrediParove(objekt *Lobj);`

Određuje parove objekata koji su potencijalno u koliziji, tj. parove objekata čije projekcije se preklapaju po bar jednoj osi. Za provjeru preklapanja po x osi koristi se sljedeći kod:

```
// od središta početnog objekta provjeri da li postoji objekt koji
je desno od njega sa središtem udaljenim najviše za najviše 2 * r
for(i=0; i<brojObjekata-1; i++){
      for(j=i+1; j<brojObjekata; j++){
            if(Lx[j].pos.x - Lx[i].pos.x <= 2*objekt::r){
                  // ako takav objekt postoji, označi par u matrici.
                        Pri tome indeks prvog objekta mora biti
                        manji od indeksa drugog
                  if(Lx[i].indeks < Lx[j].indeks){
            LxPKP.push_back(PKP(Lx[i].indeks, Lx[j].indeks));
            matricaPKP[Lx[i].indeks*objekt::n + Lx[j].indeks] += 1;
                  }
                  else{
            LxPKP.push_back(PKP(Lx[j].indeks, Lx[i].indeks));
            matricaPKP[Lx[j].indeks*objekt::n + Lx[i].indeks] += 1;
                  }
            }
            // ako ne postoji, prijeđi na sljedeći početni objekt
            else break;
      }
}
```

`PripremiZaKoliziju();`

Provjerava koja lista potencijalnih kolizijskih parova (`LxPKP`, `LyPKP` ili `LzPKP`) ima najmanje članova. Za sve parove objekata te liste poziva funkciju `DetekcijaSudaraSnPJednogPara`.

`OslobodiMatricu();`

Oslobađa memoriju alociranu za kvadratnu matricu `matricaPKP` u funkciji `AlokacijaMatrice`.

`DetekcijaSudaraSnPJednogPara(PKP &a, objekt *Lobj);`

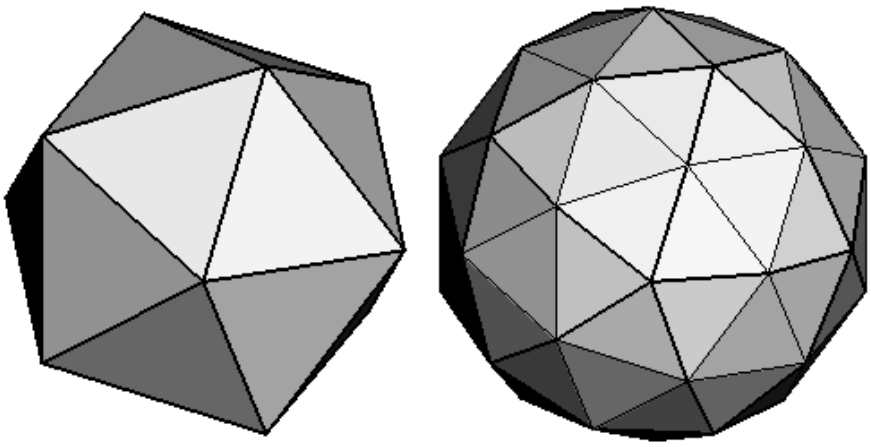
Određuje da li su objekti označeni indeksima `a.prvi` i `a.drugi` stvarno u koliziji.

Korištenje algoritma brojenja parova i pojednostavljivanja pomoću navedenih klasa je vrlo jednostavno. Prvo se svi objekti spreme u listu `Lobj` te se alocira matrica pozivom funkcije `AlokacijaMatrice`. Zatim se pozivom funkcije `Inicijalizacija` napune i sortiraju liste objekata `Lx`, `Ly` i `Lz`.

Nakon toga se jednostavno u svakom koraku simulacije nad objektom klase `SnP` pozivaju redom funkcije `Sortiranje`, `OdrediParove` i `PripremiZaKoliziju`. Zatim se mogu osvježiti položaji objekata njihovim novim brzinama.

4.5 Iscrtavanje objekata

Ako je za prikaz odabrana aproksimacija kugle sa 20 trokuta, tada je kugla aproksimirana ikosaedrom (*eng. icosahedron*) – geometrijskim tijelom koje je definirano sa 12 vrhova, 30 bridova i 20 trokuta. Prema uputama iz [7]
je dodatno implementirana podjela svakog trokuta ikosaedra na 4 dodatna, te je tako moguće kuglu aproksimirati i sa 80 trokuta. Rezultati aproksimacija se mogu vidjeti na slici 4.1.



Slika 4.1 Aproksimacija kugle ikosaedrom te dodatna podjela za detaljniji prikaz.

5. Simulacija reakcije na sudar

Kao reakcija na sudar, implementiran je savršeno elastični sudar [4] između kuglica. Kod savršeno elastičnog sudara vrijede dva zakona:

1. Zakon očuvanja količine gibanja
   „Količina gibanja izoliranog sustava je konstantna, odnosno, ukupna promjena količine gibanja u vremenu unutar izoliranog sustava jednaka je nuli.“

2. Zakon očuvanja energije
   „Energija zatvorenog sustava ne može nestati niti iz ničega nastatati, energija može samo prelaziti iz jednog oblika u drugi, i ona je konstantna.“

Ako promatramo izolirani sustav od dvije kuglice koje se sudaraju, te sa indeksom *i* označimo početnu (inicijalnu) brzinu gibanja kuglica, a sa indeksom *f* završnu (finalnu) brzinu gibanja kuglica, tada na temelju opisanih zakona možemo dobiti sljedeće izraze:

$$m_1 v_{1i} + m_2 v_{2i} = m_1 v_{1f} + m_2 v_{2f} \qquad (1)$$

$$\frac{m_1 v_{1i}^2}{2} + \frac{m_2 v_{2i}^2}{2} = \frac{m_1 v_{1f}^2}{2} + \frac{m_2 v_{2f}^2}{2} \qquad (2)$$

Iz izraza (1), (2) lako se dobivaju brzine kuglica nakon sudara:

$$v_{1f} = (\frac{m_1 - m_2}{m_1 + m_2}) v_{1i} + (\frac{2m_2}{m_1 + m_2}) v_{2i} \qquad (3)$$

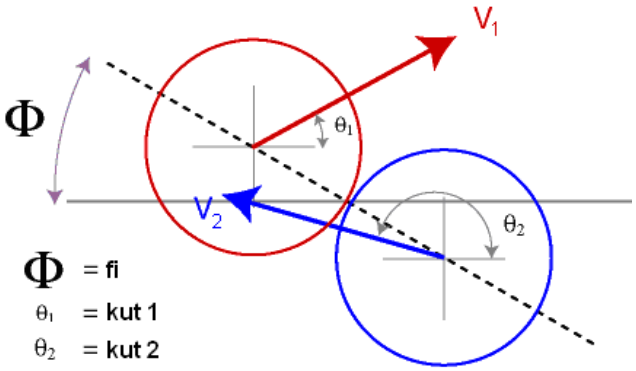
$$v_{2f} = (\frac{2m_1}{m_1 + m_2}) v_{1i} + (\frac{m_2 - m_1}{m_1 + m_2}) v_{2i} \qquad (4)$$

Može se primijetiti da se kod kugli jednake mase izrazi (3) i (4) pojednostavljuju u izraze (5) i (6), tj. kugle samo izmijene brzine:

$$v_{1f} = v_{2i} \qquad (5)$$

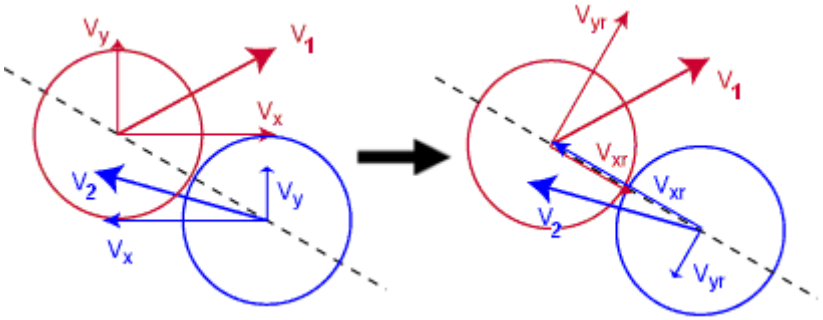
$$v_{2f} = v_{1i} \qquad (6)$$

Nakon što su određeni iznosi konačnih brzina, potrebno je još odrediti pojedine komponente brzina. U tu svrhu je u slučaju 2D kolizije potrebno odrediti kutove Φ, θ_1 i θ_2 koji se mogu vidjeti na slici 5.1.



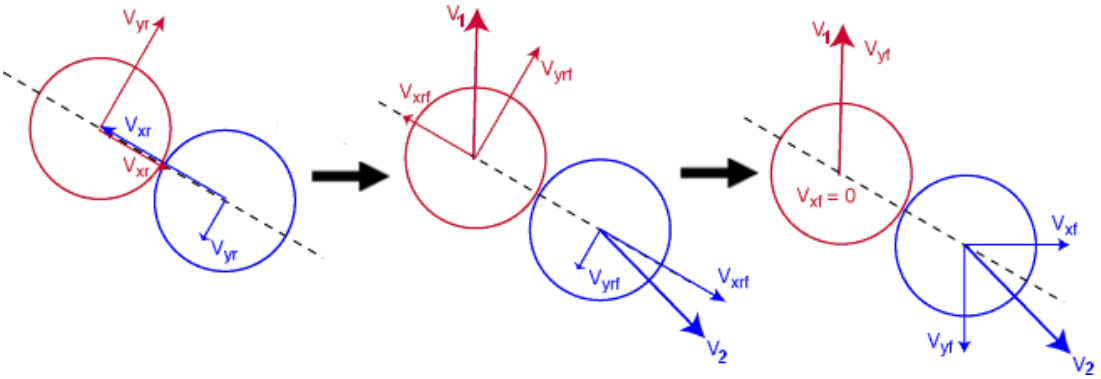
Slika 5.1 Kutovi kod sudara.

Zatim se, pomoću kutova izračunatih u prošlom koraku, učini promjena iz referentnog x-y koordinatnog sustava u koordinatni sustav gdje x osi leži na pravcu koji spaja središta dvije kugle, a y os je okomita na taj pravac.



Slika 5.2 Promjena koordinatnog sustava.

Kod sudara, komponenta brzine okomita na spojnicu središta kugli (v_{yr}) ostaje nepromijenjena, a komponenta okomita na nju (v_{xr}) se mijenja na temelju izraza (3) i (4) (ili (5) i (6) ako imamo kugle jednake mase). Time je 2D problem pojednostavljen na 1D problem.

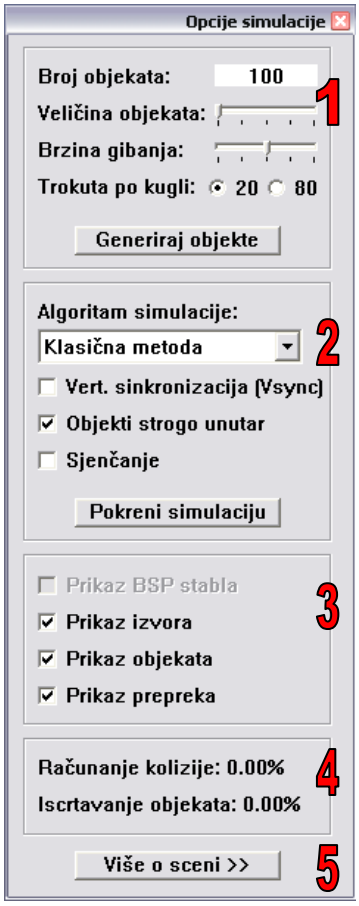


Slika 5.3 Brzine prije sudara, nakon sudara i povratak u x-y koordinatni sustav.

Jedino što je na kraju još potrebno napraviti je vratiti se natrag u x-y koordinatni sustav.
3D kolizija kuglica se temelji na istom principu. Samo je potrebno izračunati dodatne kutove, a postupak je teže jasno ilustrirati crtežom.

6. Opis sučelja programa

Simulacija se kontrolira grafičkim sučeljem prikazanim na slici 6.1.



1. Prva grupa opcija omogućuje promjenu broja objekata koji će biti generirani na sceni, promjenu njihove veličine, brzine gibanja te broja trokuta pomoću kojih su prikazani. Sve opisane opcije se počinju primjenjivati nakon klika na gumb „Generiraj objekte“.

2. Druga grupa opcija omogućuje odabir algoritma kojim će se vršiti detekcija sudara, pokretanje i zaustavljanje simulacije, te uključivanje i isključivanje sjenčanja i vertikalne sinkronizacije iscrtavanja. Opcija „Objekti strogo unutar“ označava da će se tijekom simulacije svi objekti nalaziti strogo unutar prvog izvora iz liste izvora.

3. Treća grupa opcija omogućuje odabir onoga što će se iscrtati u prozoru za iscrtavanje simulacije. Ako je kao algoritam iscrtavanja odabrano BSP stablo ili samo-podesivo BSP stablo, moguće je odabrati i prikaz BSP stabla.

Slika 6.1 Grafičko sučelje programa.

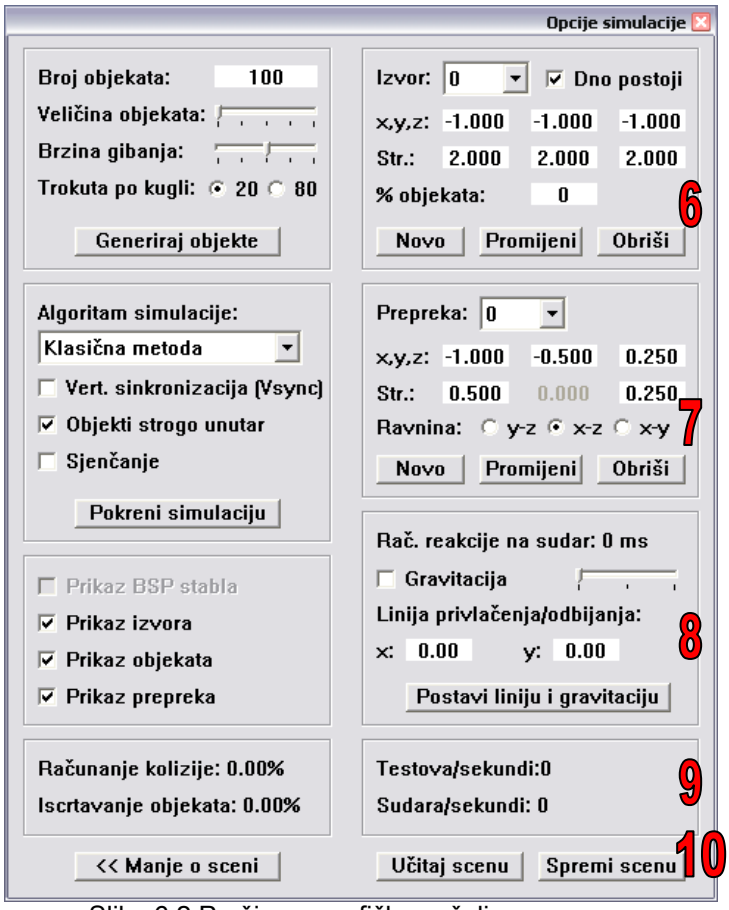
4. Četvrta grupa prikazuje koliki postotak vremena program troši na računanje kolizije, a koliki na iscrtavanje samih objekata.

5. Klikom na gumb „Više o sceni“ dobiva se prošireno sučelje programa i više detalja o samoj sceni koja je iscrtana u prozoru za iscrtavanje. Prošireno sučelje programa se može vidjeti na slici 6.2.

Kod većine osnovnih opcija je iz samog imena jasno što rade. Jedino bi opciju *Vertikalna sinkronizacija* (*eng. Vsync*) bilo potrebno detaljnije objasniti.

Uz uključenu vertikalnu sinkronizaciju, vrši se sinkronizacija između zamjene spremnika slike na grafičkoj kartici (*eng. color buffer*) i frekvencije osvježavanja monitora. Tada broj iscrtanih slika na grafičkoj kartici (*eng. FPS – Frames Per Second*) nikad ne prelazi frekvenciju osvježavanja monitora (najčešće vrijednost između 60 i 85Hz).

Kako je broj iscrtanih slika u sekundi glavna mjera performanse pojedinih algoritama, potrebno je (ako je uključena), isključiti vertikalnu sinkronizaciju iscrtavanja. Kada stvarna brzina iscrtavanja padne ispod brzine osvježavanja monitora, nije važno da li je vertikalna sinkronizacija uključena ili ne.



Slika 6.2 Prošireno grafičko sučelje programa.

6. Šesta grupa opcija omogućuje kreiranje jednog ili više izvora objekata. Izvor objekata na sceni je kvadar unutar kojeg će se nakon klika na gumb „Generiraj objekte“ generirati objekti. Položaj jednog vrha izvora se definira upisivanjem vrijednosti „x,y,z:“ (u ovom primjeru to je -1.000, -1.000, -1.000) a ostalih 7 vrhova je definirano dodavanjem iznosa stranice pojedinoj koordinati (u ovom primjeru imamo kocku čije sve stranice iznose 2.000).
   „% objekata:“ označava koliki postotak od ukupnog broja objekata će se nalaziti u trenutnom izvoru. Opcijom „Dno postoji“ se definira da li trenutan izvor (kvadar) ima dno.
   Promjene izvora se spremaju klikom na gumb „Promijeni“, a gumbi „Novo“ i „Obriši“ dodaju novi izvor u listu ili brišu postojeći.

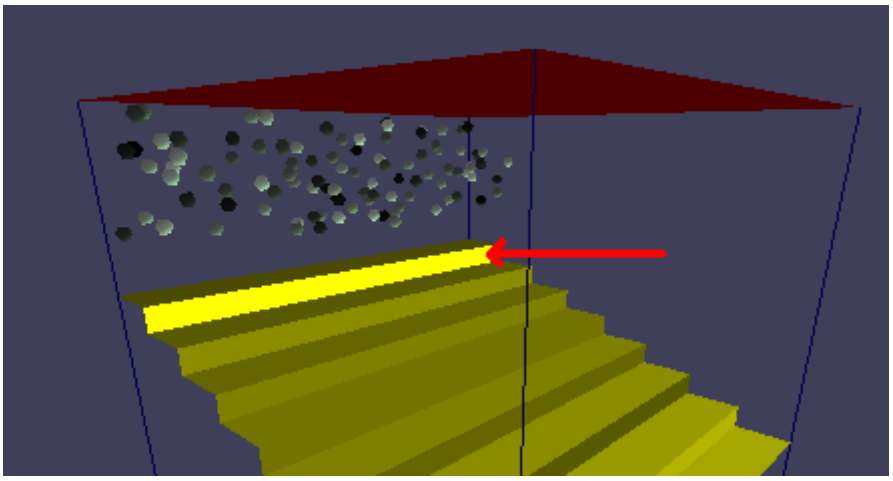
7. Sedma grupa opcija omogućuje kreiranje jedne ili više prepreka. Prepreke na sceni su pravokutnici od kojih se objekti odbijaju. Oni se nalaze u „x-y“, „y-z“ ili „x-z“ ravnini, i definirani su svojim položajem te

duljinom stranica. Trenutno odabrana prepreka u listi je iscrtana svjetlijom bojom u prozoru za iscrtavanje (slika 6.3)

8. Osma grupa opcija omogućuje uključivanje i isključivanje gravitacije u sceni. Ako je uključena, može se postaviti na slabu, srednju i jaku. Linija privlačenja/odbijanja je pravac okomit na x-y ravninu, definiran sa svojim položajem u x-y ravnini. Kada je fokus na prozoru za iscrtavanje, pritisak na tipku „p“ na tipkovnici uzrokuje privlačenje svih objekata prema toj liniji. Tipka „o“ uzrokuje odbijanje od linije.
   Kada je fokus na prozoru za upravljanje simulacijom, pritisak na tipke „-“ i „+“ smanjuje i povećava vrijeme u koracima od po 1ms (od najmanje 0 do najviše 100ms) koje se dodatno troši pri računanju reakcije na sudar. Tako se može provjeriti utjecaj algoritama kod računanja složenijih reakcija na sudar.

9. U devetoj grupi se može pročitati koliko se parova objekata testira za koliziju po sekundi te koliko se stvarno dogodi kolizija po sekundi.

10. Gumbi „Učitaj scenu“ i „Spremi scenu“ učitavaju scenu definiranu u datoteci ili spremaju trenutno kreiranu scenu u datoteku.

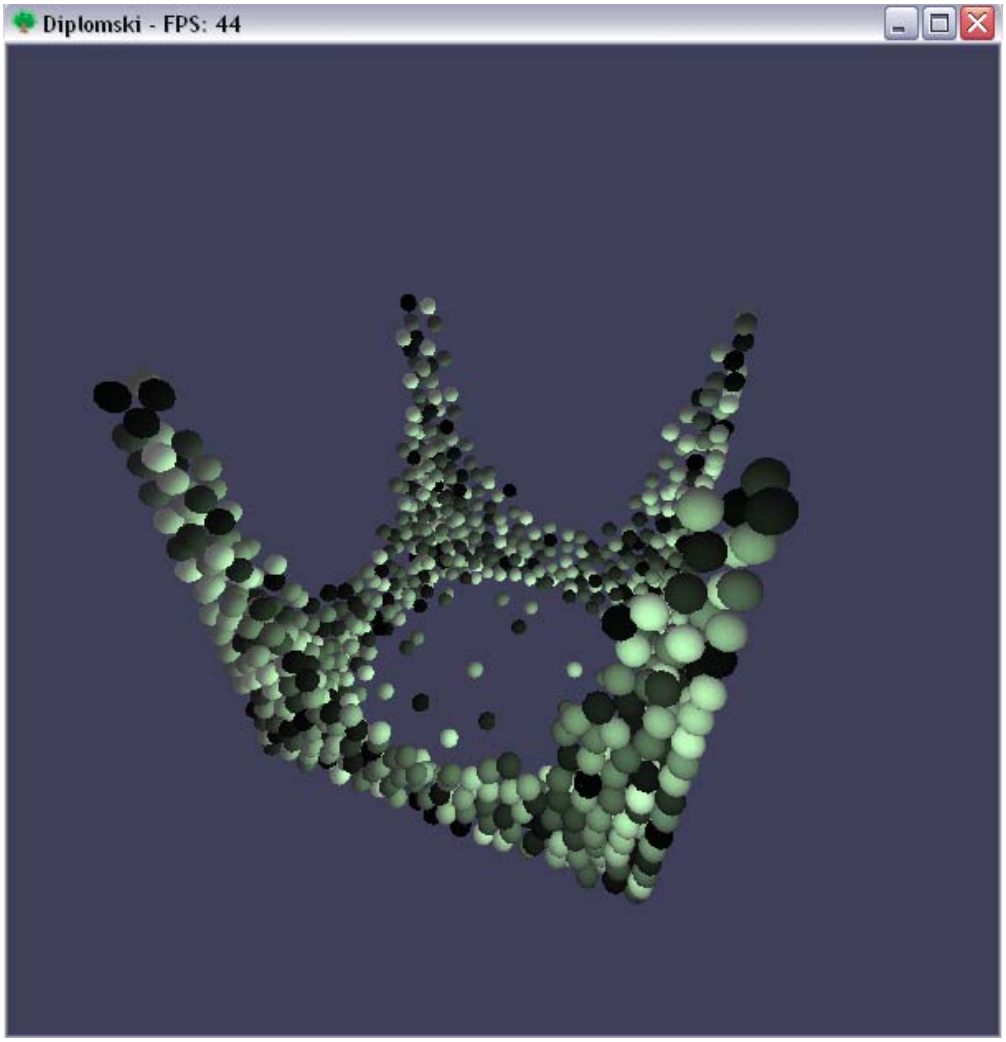


Slika 6.3 Odabrana prepreka je označena svjetlijom bojom.

Na slici 6.4 se može vidjeti prozor u kojem se iscrtava simulacija. Na slici je prikazano 1000 kugli uz uključeno sjenčanje, svaka kugla je prikazana sa 80 trokuta, a da bi se dobio zadani raspored, uključena je gravitacija te postavljena linija odbijanja u sredinu kocke (izvora) u kojoj se nalaze kugle.

Na naslovnoj traci prozora za iscrtavanje simulacije se u svakom trenutku može pročitati trenutna brzina iscrtavanja, tj. broj slika po sekundi (*eng. Frames Per Second – FPS*).

Lijevim klikom miša unutar prozora (uz zadržavanje gumba) te povlačenjem miša se postiže rotacija kamere oko točke 0,0,0.



Slika 6.4 Prozor u kojem se iscrtava simulacija.

Ostala interakcija korisnika s prozorom za iscrtavanje uključuje pokretanje i zaustavljanje simulacije pritiskom tipke „s“ na tipkovnici, te privlačenje i odbijanje objekata u odnosu na postavljenu liniju pritiskom na tipke „p“ i „o“.

7. Format ulazne datoteke

Osim što se scena može definirati interaktivno pomoću programa pa zatim spremiti u datoteku, također je scenu moguće definirati bilo kojim tekstualnim editorom direktno u datoteci, a zatim učitati u program. Redoslijed definicije elemenata scene u datoteci nije važan, jedino će prvi izvor definiran u datoteci biti onaj na kojeg će se odnositi opcija „Objekti strogo unutar“ iz druge grupe opcija.

Kod definiranja ulazne datoteke vrijede sljedeća pravila:

- ako redak počinje sa '#' to je komentar
- ako redak počinje sa 'i' to je izvor (kvadar unutar kojeg se generiraju objekti)
- ako redak počinje sa 'p' to je prepreka
- ako redak počinje sa 's' nakon čega slijedi 0/1, objekti nisu/jesu strogo unutar prvog izvora

Sintaksa za definiranje izvora i prepreke:

- izvor : `"i x_koord y_koord z_koord stranica_x stranica_y stranica_z postotak_ukupnog_broja_objekata ima_dno(t/f)"`
- prepreka : `"p x_koord y_koord z_koord stranica_x stranica_y stranica_z ravnina('x' za y-z ravninu/'y' za x-z/'z' za x-y)"`

U izvor koji je posljednji definiran u datoteci će se spremiti svi objekti koji nisu spremljeni u ostalim izvorima, tj. za posljednji izvor se ne čita podatak o postotku objekata već je on postavljen na: „100 – suma postotaka svih ostalih izvora“. Na taj način je osigurano da će svi objekti biti generirani i prikazani na sceni.

Za najjednostavniju scenu je dovoljno definirati samo jedan izvor unutar kojeg će se generirati objekti. Npr. sljedećim retkom se definira scena koja se sastoji smo od jednog izvora – kocke unutar koje se generiraju svi objekti:

```
i     -1.0  -1.0  -1.0  2.0   2.0   2.0   100   t
```

Vrhovi kocke su u koordinatama: (-1,-1,-1), (-1,1,-1), (-1,-1,1), (-1,1,1), (1,-1,-1), (1,1,-1), (1,-1,1), (1,1,1)

Kocka ima dno, te sadrži 100% objekata (kako je navedena kocka jedini izvor, umjesto 100 je mogao pisati bilo koji broj, no bio bi postavljen 100, da se osigura da svi objekti na sceni budu generirani i iscrtani).

8. Rezultati simulacija

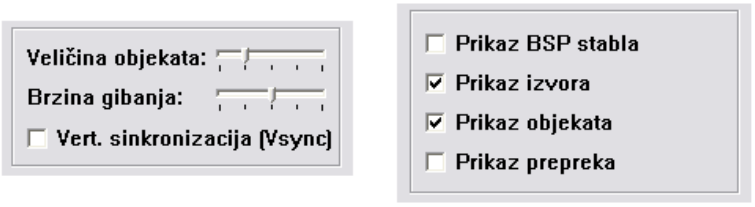
Tijekom simulacija je ispitano kako različiti broj objekata te različiti rasporedi objekata na sceni utječu na brzinu iscrtavanja kod primjene opisanih algoritama i prostornih struktura podataka. Također je ispitan utjecaj različitog broja trokuta na sceni te sjenčanja na brzinu iscrtavanja te zauzeće resursa računala.

Dodatna ispitivanja su pokazala da brzina gibanja objekata ne utječe na brzinu iscrtavanja, dok povećanje veličine objekata utječe negativno na sve metode – uzrokuje povećanje broja kolizija, kod BSP stabala dodatno i povećanje redundancije, a kod algoritma brojenja parova i pojednostavljivanja uzrokuje veći broj preklapanja po osima.

Sva ispitivanja su provedena na sljedećem računalu:

```
Intel® Core(TM) 2 Duo E6550 2x2.33 GHz
ATI Radeon HD3870, 512MB GDDR3
2GB DDR2 RAM, 800 MHz
```

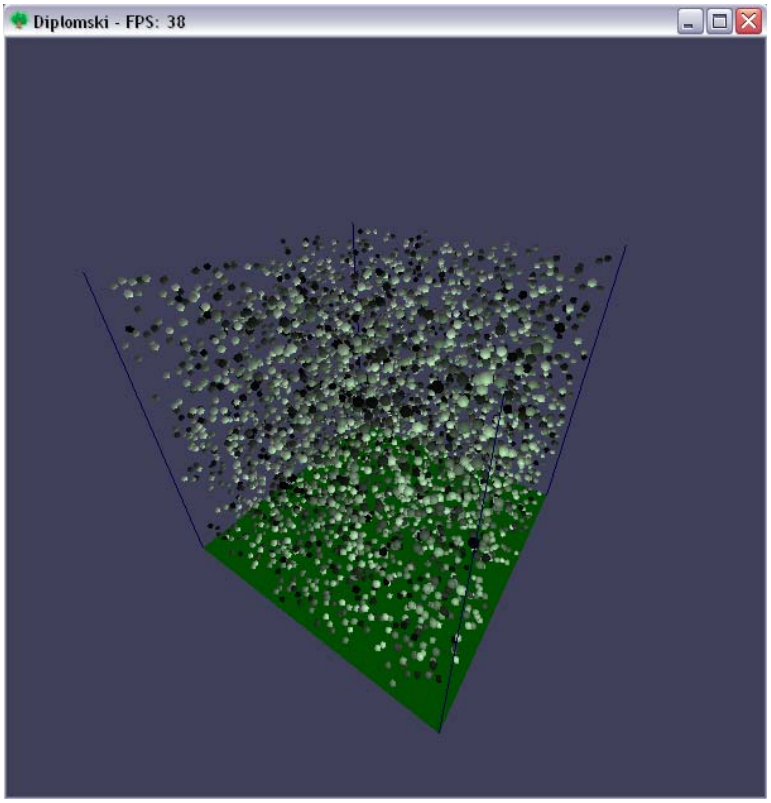
Tijekom svih testova brzina gibanja objekata, njihova veličina i opcije prikaza su bile postavljene kao na slici 8.1 (ako nije drugačije navedeno).



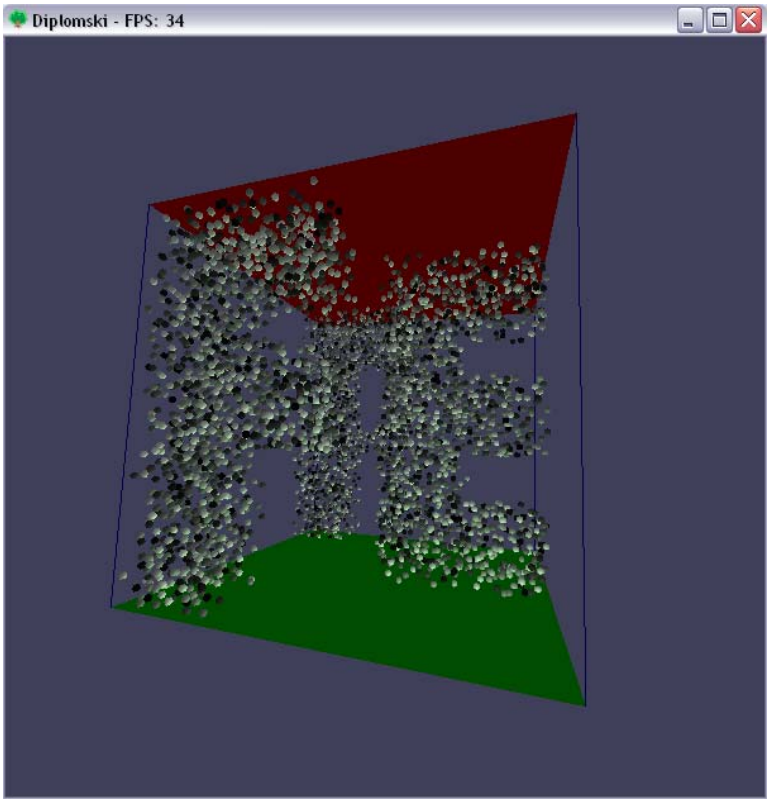
Slika 8.1 Postavke simulacije.

Korištene su dvije vrste rasporeda objekata na sceni. Kod slučajnog rasporeda (slika 8.2) kao izvor objekata se koristi kocka duljine stranice 2.0 unutar koje se generiraju objekti. Za objekte unutar kocke ne postoje druga ograničenja gibanja.

Kod drugog, „FER“ rasporeda objekta (slika 8.3), objekti su raspoređeni tako da nakon nekog vremena oblikuju slova „F“ „E“ i „R“. To je postignuto pomoću četiri izvora objekata te nekoliko prepreka. Izvor koji sve omeđuje je ponovo kocka duljine stranice 2.0. Ostala 3 izvora čine dijelove slova „F“ „E“ i „R“. Njihovim brisanjem iz liste, objekti mogu oblikovati slova „F“ „E“ i „R“.

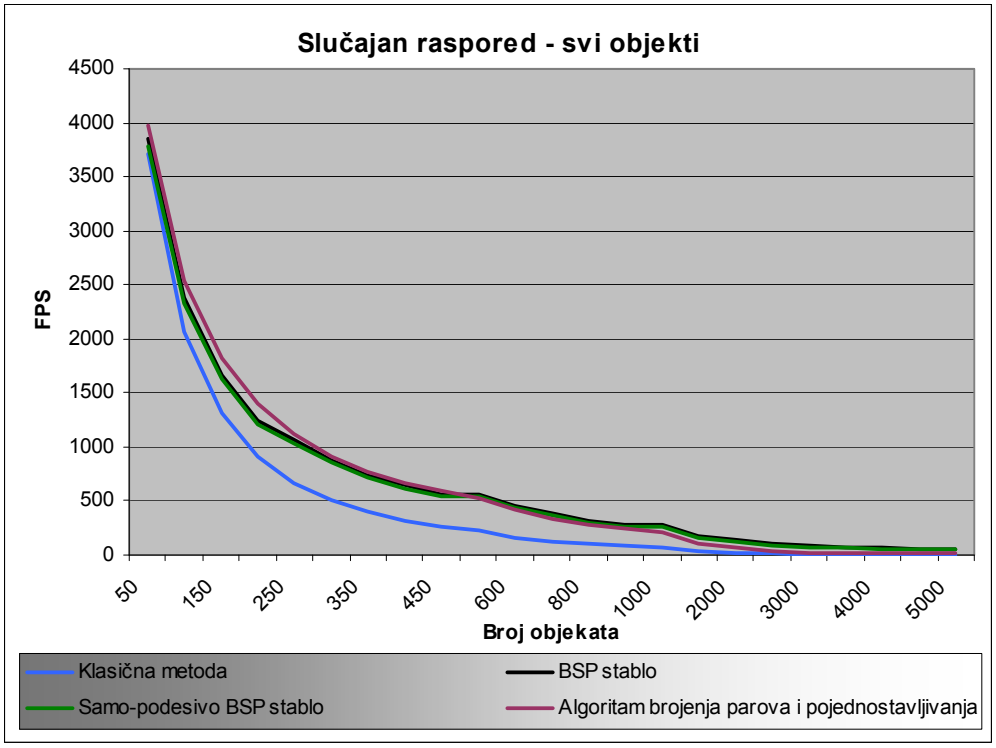


Slika 8.2 Slučajan raspored objekata.

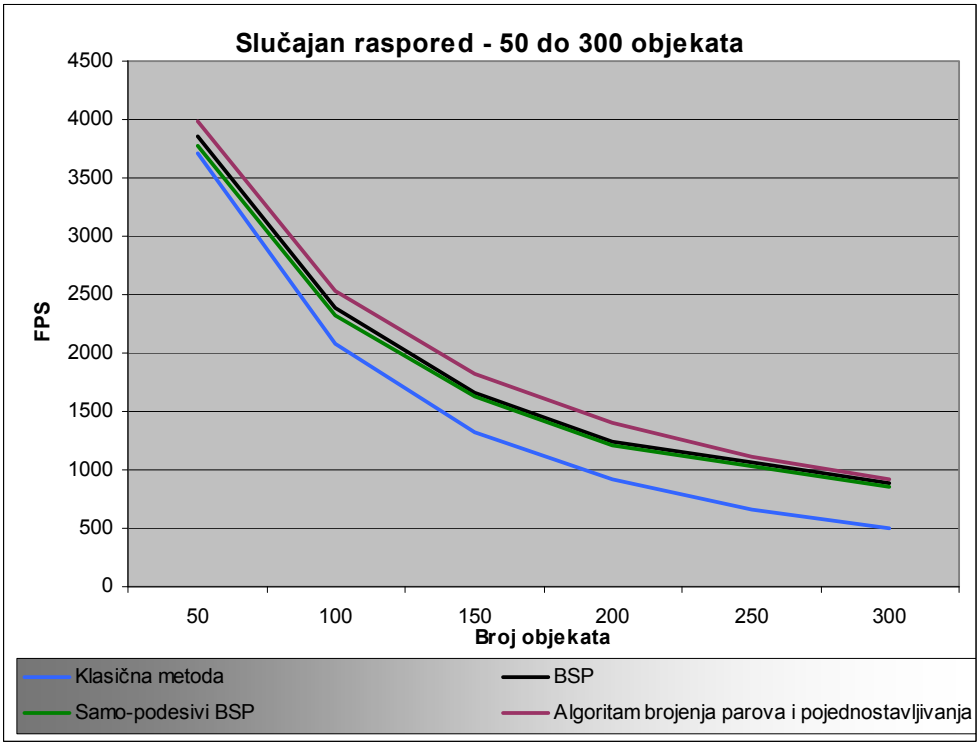


Slika 8.3 „FER“ raspored objekata.

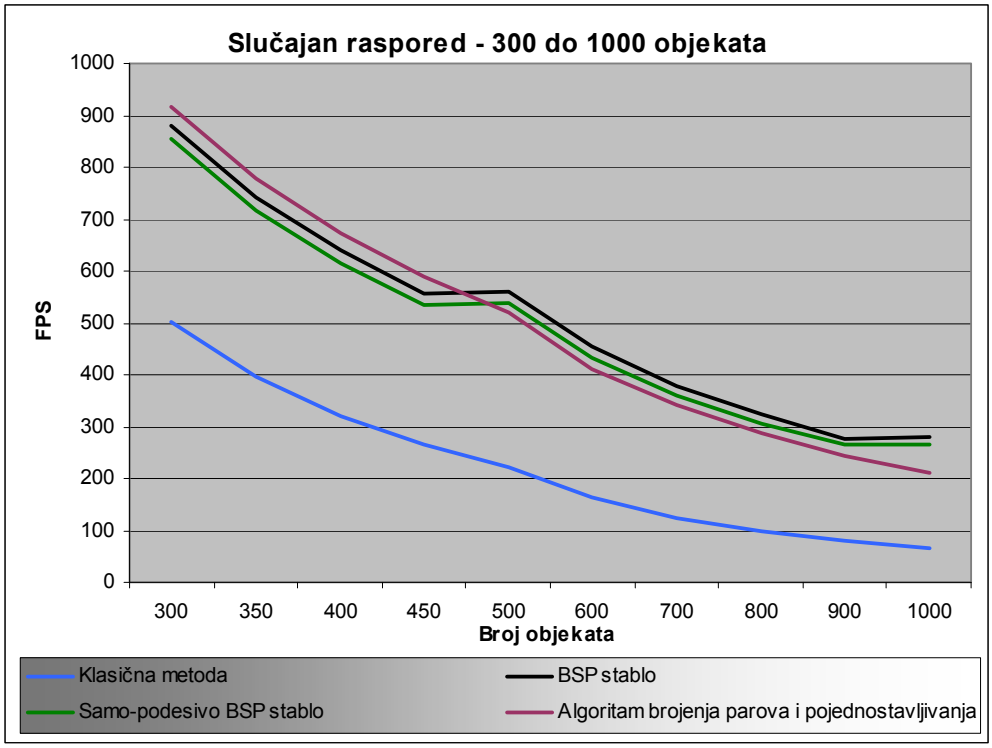
8.1 Slučajan raspored objekata uz njihovo iscrtavanje



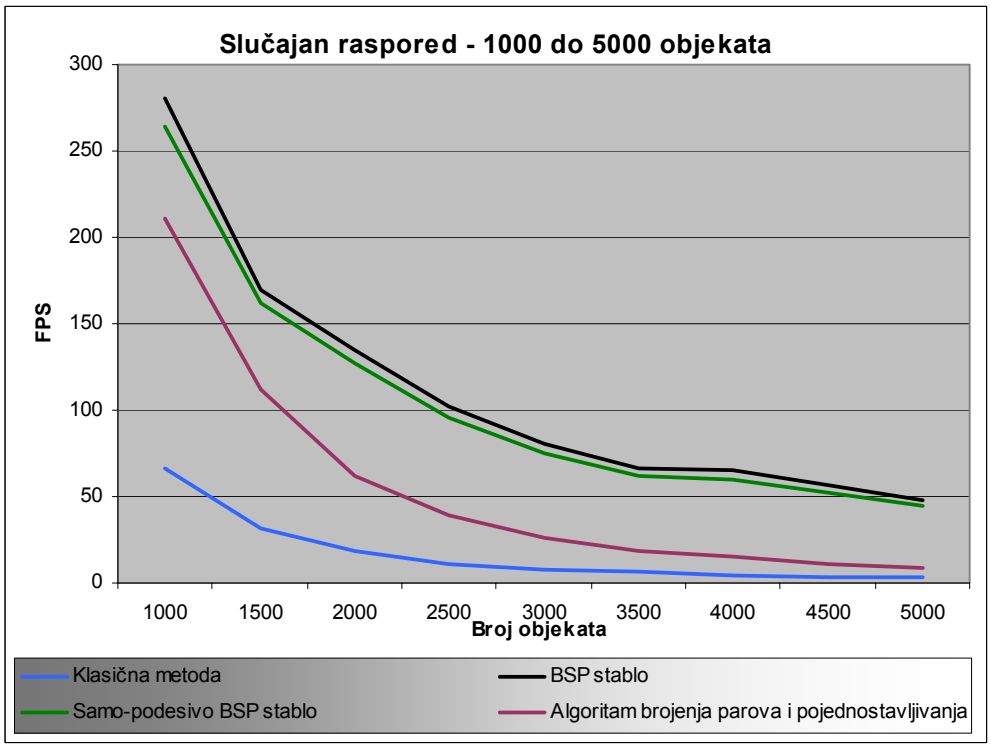
Slika 8.4 Graf zavisnosti brzine iscrtavanja i broja objekata (50 do 5000) kod slučajnog rasporeda objekata uz njihovo iscrtavanje.



Slika 8.5 Graf zavisnosti brzine iscrtavanja i broja objekata (50 do 300) kod slučajnog rasporeda objekata uz njihovo iscrtavanje.



Slika 8.6 Graf zavisnosti brzine iscrtavanja i broja objekata (300 do 1000) kod slučajnog rasporeda objekata uz njihovo iscrtavanje.



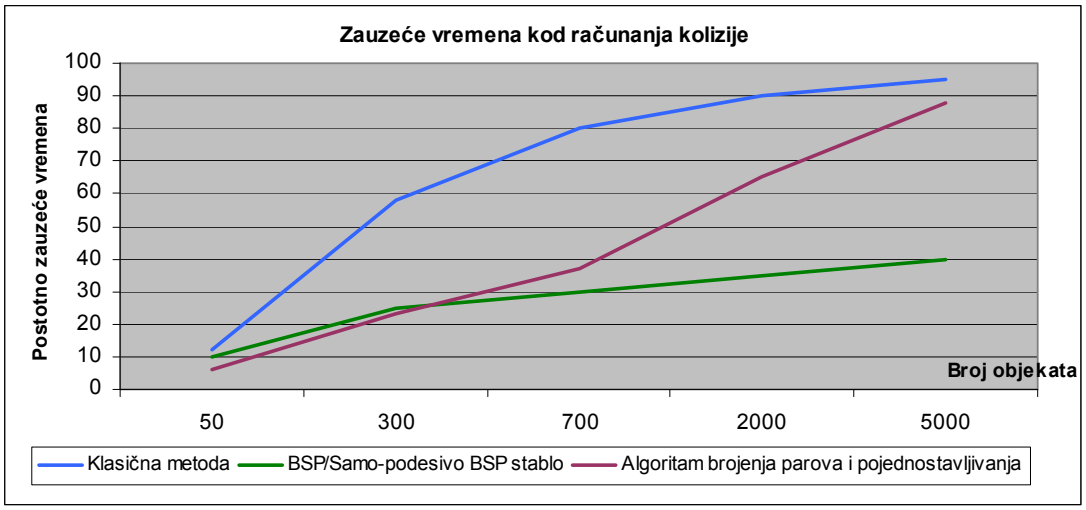
Slika 8.7 Graf zavisnosti brzine iscrtavanja i broja objekata (1000 do 5000) kod slučajnog rasporeda objekata uz njihovo iscrtavanje.

Iz slika 8.4 – 8.7 se može vidjeti da klasična metoda detekcije sudara pokazuje konstantno najlošije performanse, tj. najmanji broj iscrtanih slika u sekundi. Brzina iscrtavanja pada sa povećanjem broja objekata i već kod 2000 objekata pada ispod 20 slika u sekundi, što bi se moglo smatrati minimumom za interaktivni prikaz simulacije.

Kod slučajnog rasporeda objekata BSP stablo i samo-podesivo BSP stablo pokazuju vrlo slične performanse. Zbog slučajnog, prilično jednolikog rasporeda objekata, prilagodljivost samo-podesivog BSP stabla ne dolazi do izražaja i to stablo pokazuje (zanemarivo) lošije performanse od standardnog BSP stabla, zbog čestog evaluiranja balansiranosti čvorova.

Algoritam brojenja parova i pojednostavljivanja do 500 objekata pokazuje najbolje performanse, jer zbog male gustoće objekata na sceni te malo preklapanja projekcija objekata po osima, nije potrebno raditi puno stvarnih testova kolizije. Pri većem broju objekata raste broj preklapanja i broj testova kolizije te se performanse te metode pri većem broju objekata približavaju klasičnoj metodi. No čak i kod 5000 objekata, brzina iscrtavanja je 3 puta veća od klasične metode (iznosi 9 *FPS*), a pada ispod 20 slika u sekundi tek kod 3500 objekata.

Povećanje brzine iscrtavanja kod samo-podesivog i standardnog BSP stabla vidljivo kod 500, 1000 i 4000 objekata (kao ravni ili blago rastući segment) je posljedica povećanja dubine stabla kod tog broja objekata. Povećanje dubine stabla se kod tih vrijednosti radi, kako bi broj objekata u svakom listu bio manji od 30 (kod idealnog rasporeda objekata na sceni).



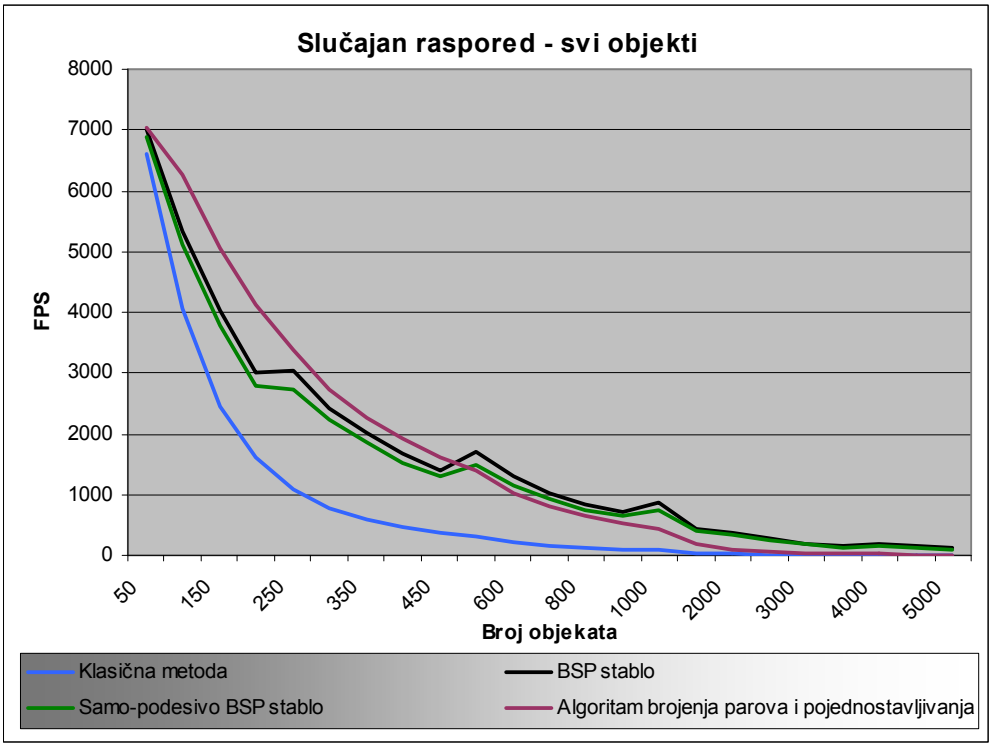
Slika 8.8 Graf zauzeća vremena tijekom računanja kolizije
u ovisnosti o broju objekata za razne metode.

Kod ovog testa treba znati da se osim računanja same kolizije, također vrši i iscrtavanje objekata na sceni te se i za to troše resursi računala. Kako je svaka kugla iscrtana sa 20 trokuta, kod 50 kugli iscrtava se 1000 trokuta u svakom koraku simulacije, a kod 5000 kugli, čak 100.000 trokuta. Postotak

vremena koji se kod različitih metoda troši na samo računanje kolizije se može vidjeti na slici 8.8. Ostatak vremena se troši na iscrtavanje objekata (većina) te na iscrtavanje ostalih elemenata scene te poziv funkcija.

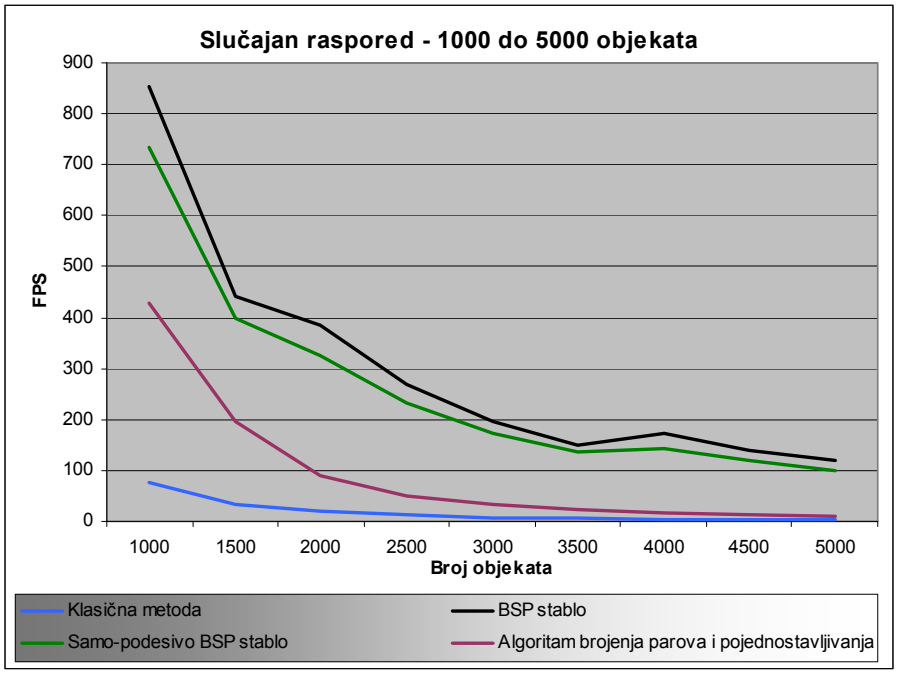
Sa slike 8.8 se može vidjeti da se, očekivano, najviše vremena za računanje kolizije troši kod klasične metode, što rezultira najsporijom brzinom iscrtavanja. Vrijeme potrošeno na samo računanje kolizije kod ostalih metoda je manje pa je stoga teže vršiti usporedbe samih algoritama. Netko možda želi koristiti opisane algoritme i strukture podataka kod ubrzanja detekcije sudara i ne zanima ga iscrtavanje objekata, no želio bi vidjeti kakvi se faktori ubrzanja dobivaju korištenjem navedenih algoritama. Stoga je u poglavlju 8.2 ponovljen test, no ovaj put bez iscrtavanja objekata.

8.2 Slučajan raspored objekata bez njihova iscrtavanja



Slika 8.9 Graf zavisnosti brzine iscrtavanja i broja objekata (50 do 5000) kod slučajnog rasporeda objekata bez njihova iscrtavanja.

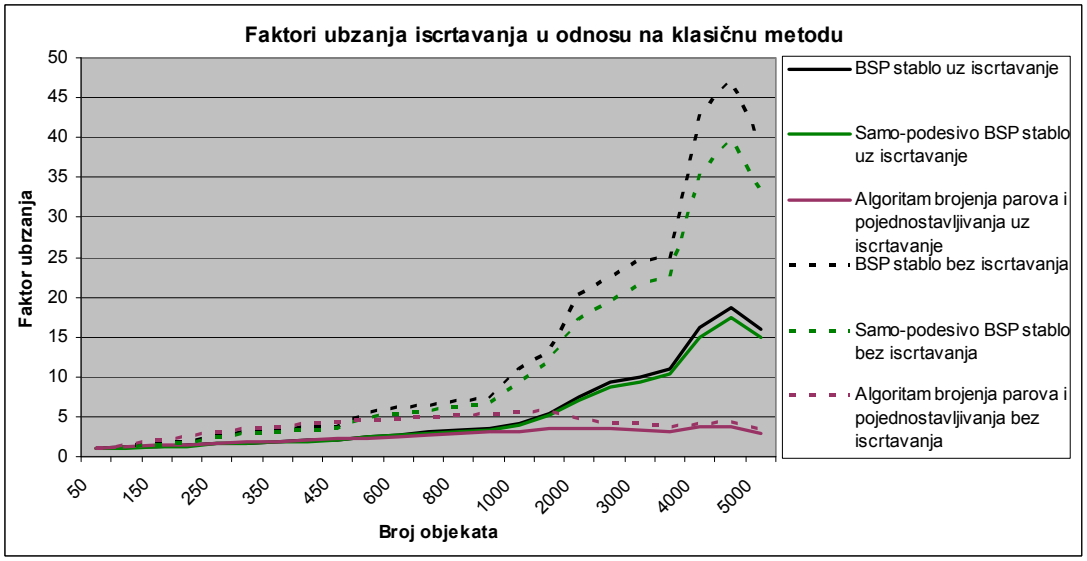
Na slici 8.9 se bolje mogu vidjeti ubrzanja iscrtavanja do kojih dolazi kod 250, 500 te 1000 objekata – umjesto ravnih segmenata sada postoje šiljci, zbog toga što ne postoji povećanje broja objekata (koji se moraju iscrtati) koje bi kompenziralo ubrzanje koje se dobiva zbog povećanja dubine stabla.



Slika 8.10 Graf zavisnosti brzine iscrtavanja i broja objekata (1000 do 5000) kod slučajnog rasporeda objekata bez njihova iscrtavanja.

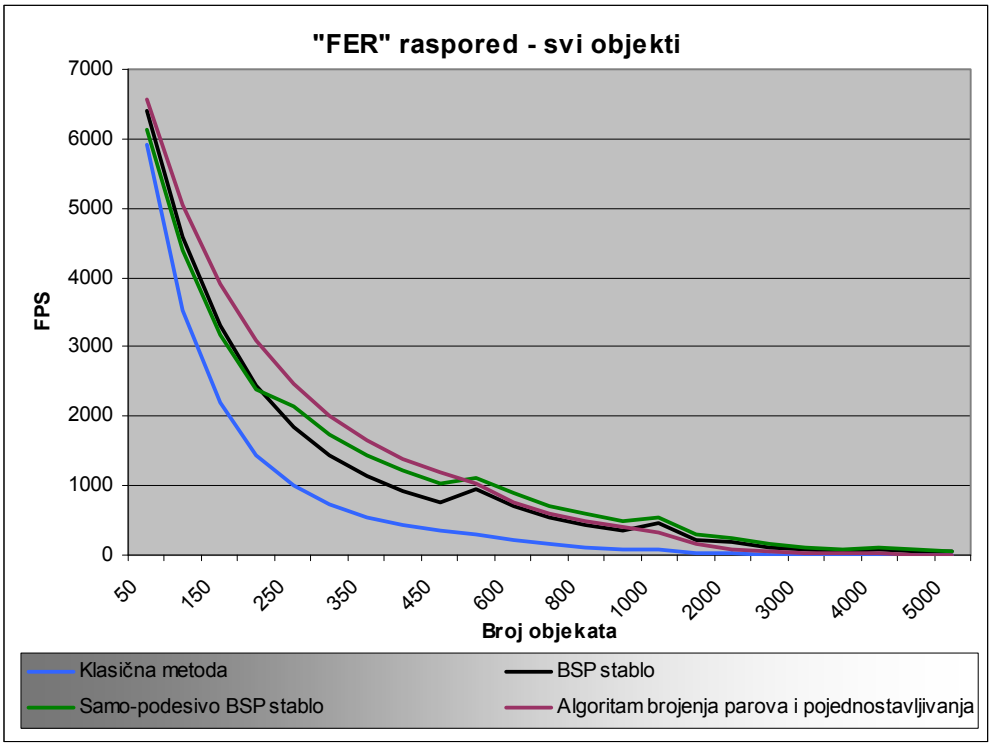
Na slici 8.10 se može vidjeti da i kod 4000 objekata sada postoji šiljak, te se i kod 2000 objekata također može vidjeti manje ubrzanje.

Bez iscrtavanja objekata i trošenja više vremena na računanje kolizije, više do izražaja dolazi ubrzanje koje daju pojedini algoritmi i strukture podataka. Na slici 8.11 se mogu vidjeti dobiveni faktori ubrzanja za sve metode i usporediti dobiveni faktori uz iscrtavanje objekata i bez njihova iscrtavanja. Može se vidjeti da su faktori ubrzanja bez iscrtavanja objekata veći.

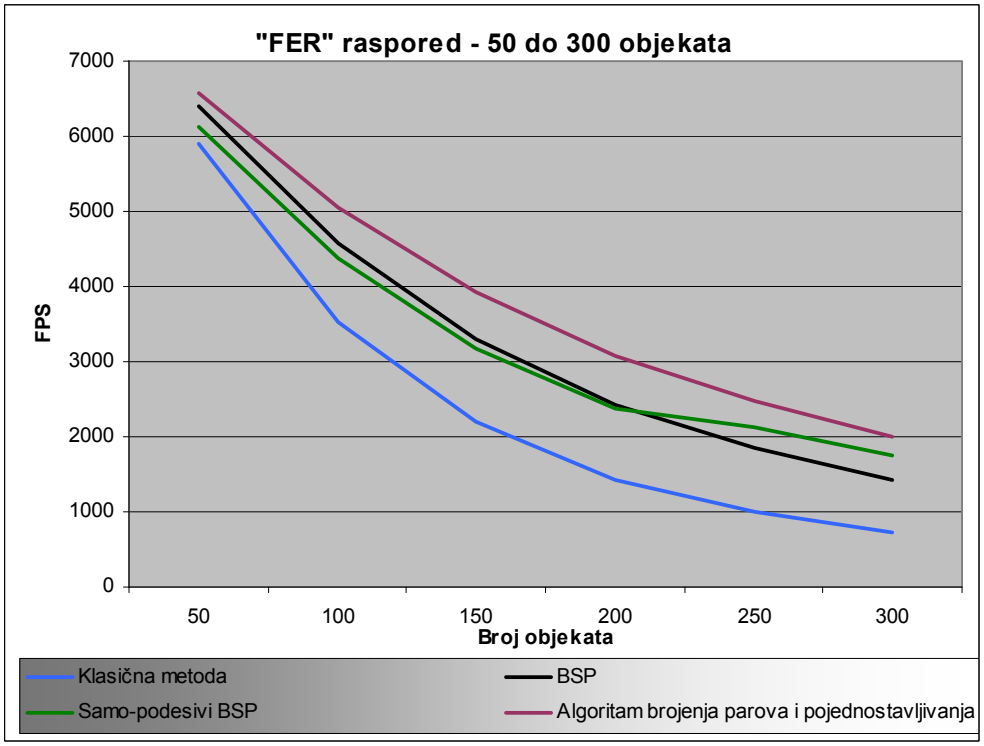


Slika 8.11 Faktori ubrzanja iscrtavanja dobiveni korištenjem raznih metoda uz i bez iscrtavanja objekata.

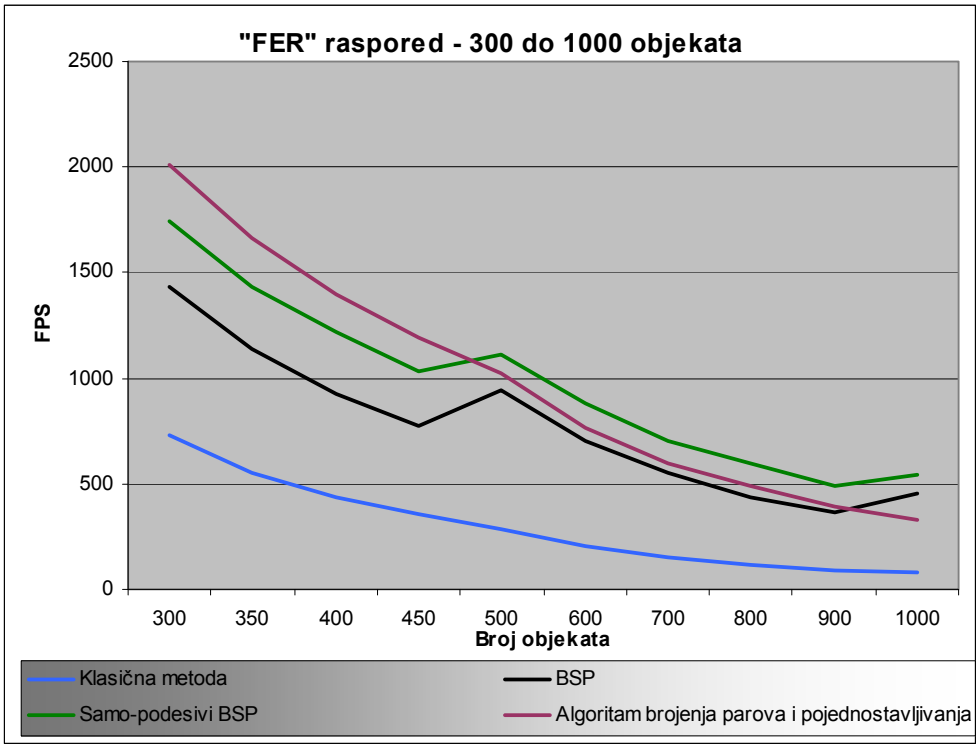
8.3 „FER“ raspored objekata bez njihova iscrtavanja



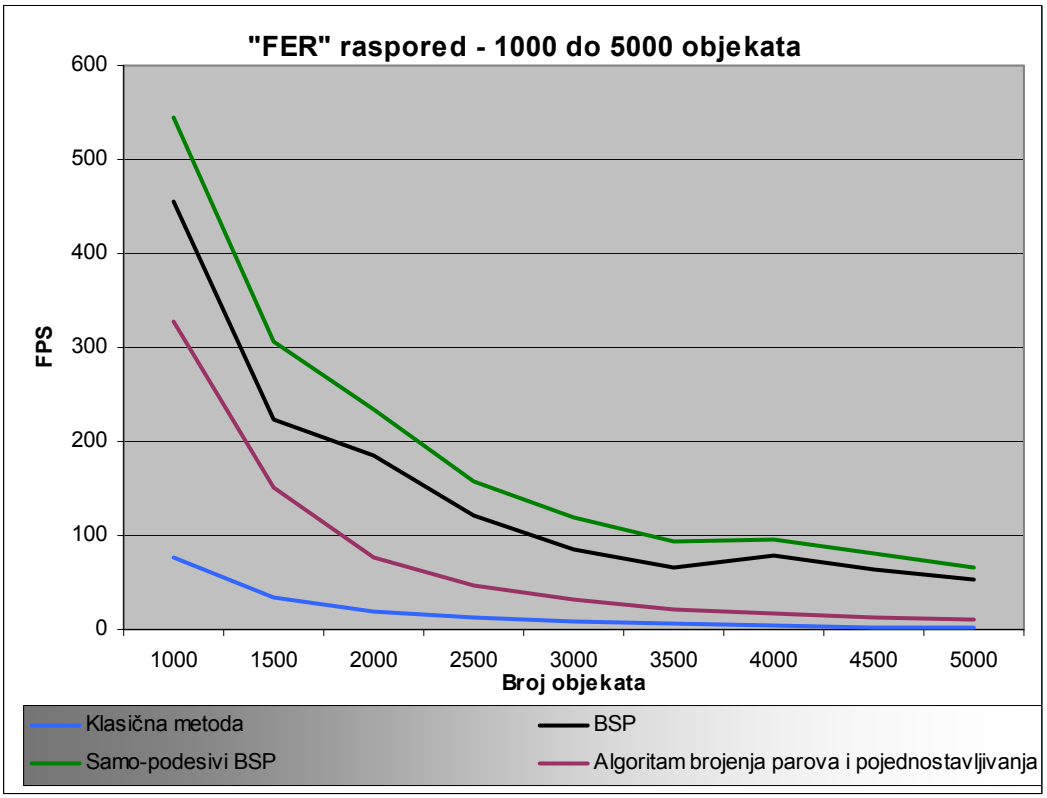
Slika 8.12 Graf zavisnosti brzine iscrtavanja i broja objekata (50 do 5000) kod „FER“ rasporeda objekata bez njihova iscrtavanja.



Slika 8.13 Graf zavisnosti brzine iscrtavanja i broja objekata (50 do 300) kod „FER“ rasporeda objekata bez njihova iscrtavanja.



Slika 8.14 Graf zavisnosti brzine iscrtavanja i broja objekata (300 do 1000) kod „FER“ rasporeda objekata bez njihova iscrtavanja.



Slika 8.15 Graf zavisnosti brzine iscrtavanja i broja objekata (1000 do 5000) kod „FER“ rasporeda objekata bez njihova iscrtavanja.

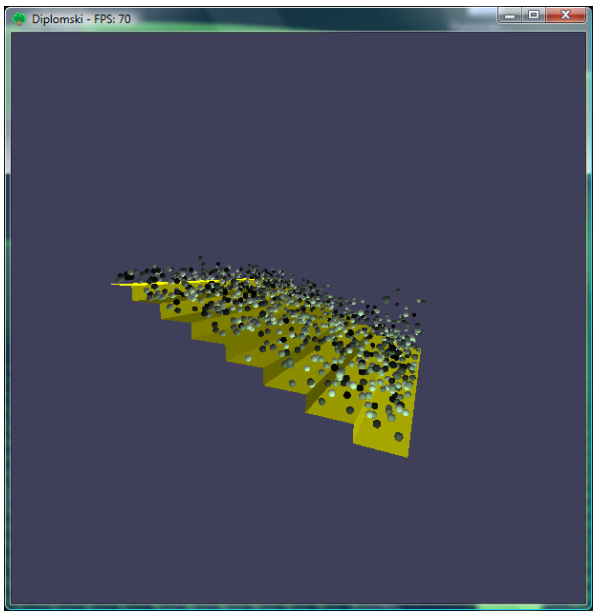
Iz slika 8.12 – 8.15 se može vidjeti da klasična metoda detekcije sudara ponovno daje najlošije rezultate.

Sa slike 8.13 se može vidjeti da kod „FER“ rasporeda objekata samo-podesivo BSP stablo pokazuje bolje performanse od standardnog BSP stabla i to od povećanja dubine stabla kod 250 objekata, pa sve do kraja simulacije. To je posljedica prilagodljivosti samo-podesivog BSP stabla različitom rasporedu objekata na sceni. Standardno BSP stablo ne mijenja svoj početni oblik pri novom rasporedu objekata, no samo-podesivo BSP stablo mijenja svoj oblik i pokušava svakim listom obuhvatiti jednak broj objekata. Zbog toga je negativan utjecaj neuniformnog rasporeda objekata na brzinu iscrtavanja mnogo veći kod standardnog BSP stablo, no što je kod samo-podesivog BSP stablo. Do 250 objekata dubina stabla nije dovoljna da samo-podesivo BSP stablo znatno bolje podijeli objekte unutar scene od standardnog BSP stabla.

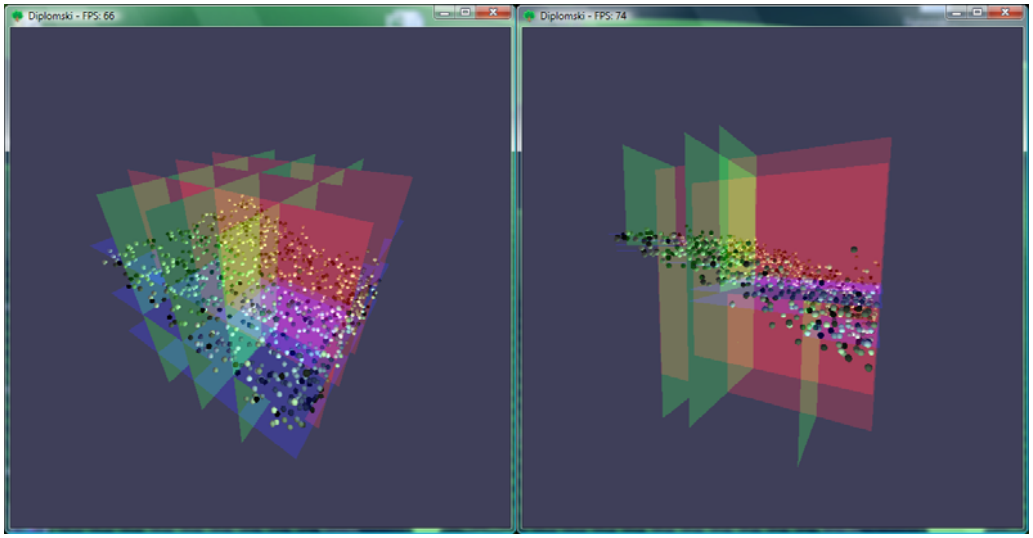
Algoritam brojenja parova i pojednostavljivanja pri „FER“ rasporedu objekata ponovo pri manjem broju objekata (do 500) pokazuje najbolje performanse zbog male gustoće objekata na sceni. Kod 1000 objekata na sceni performanse algoritma padaju ispod performansi standardnog BSP stabla (slika 8.14), a do kraja simulacije se približavaju klasičnoj metodi (slika 8.15).

U poglavlju 8.4 se može vidjeti nekoliko scena te uz njih i prikaz standardnog te samo-podesivog BSP stabla. Za prikaz ravnina podjele koristi se crvena (*y-z* ravnina podjele), zelena (*x-z* ravnina) i plava boja (*x-y* ravnina) uz korištenje prozirnosti (*eng. Alpha blending*).

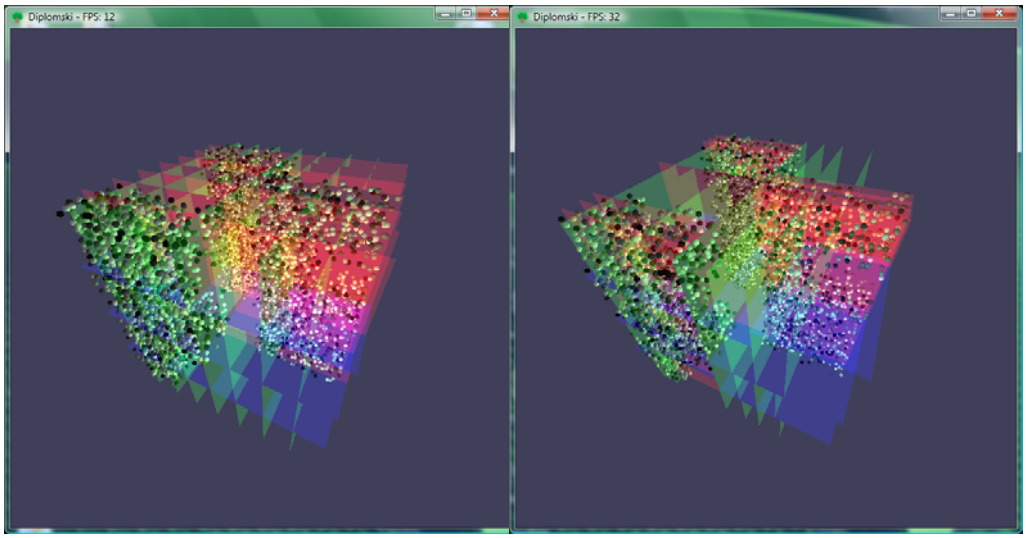
8.4 Primjeri izgleda BSP stabala



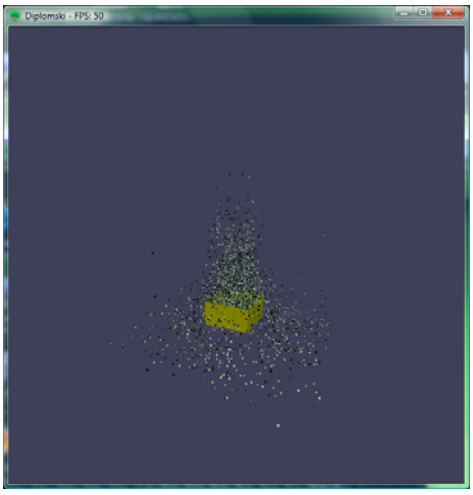
Slika 8.16 Scena „stepenice“.



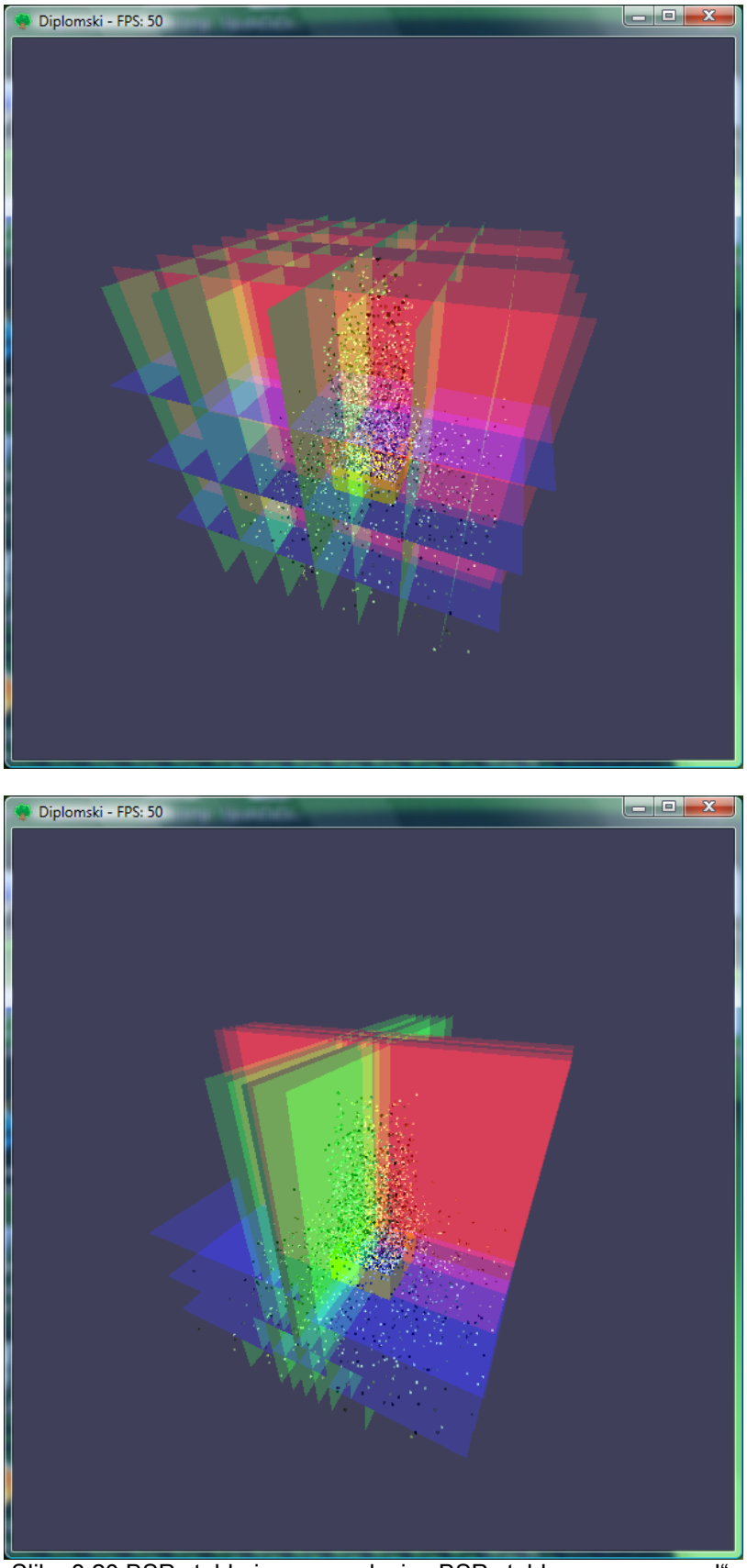
Slika 8.17 BSP stablo i samo-podesivo BSP stablo za scenu „stepenice“.



Slika 8.18 BSP stablo i samo-podesivo BSP stablo za scenu „FER“.



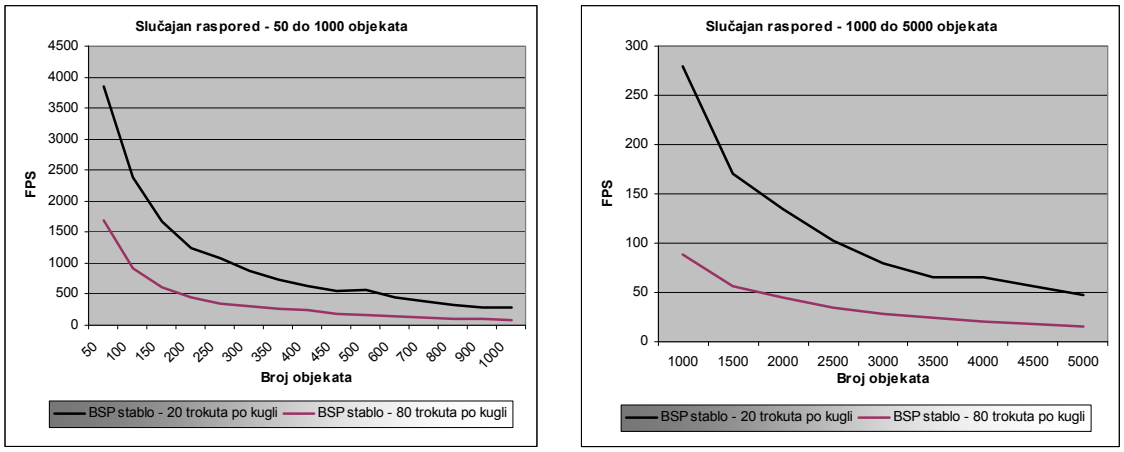
Slika 8.19 Scena „pad“.



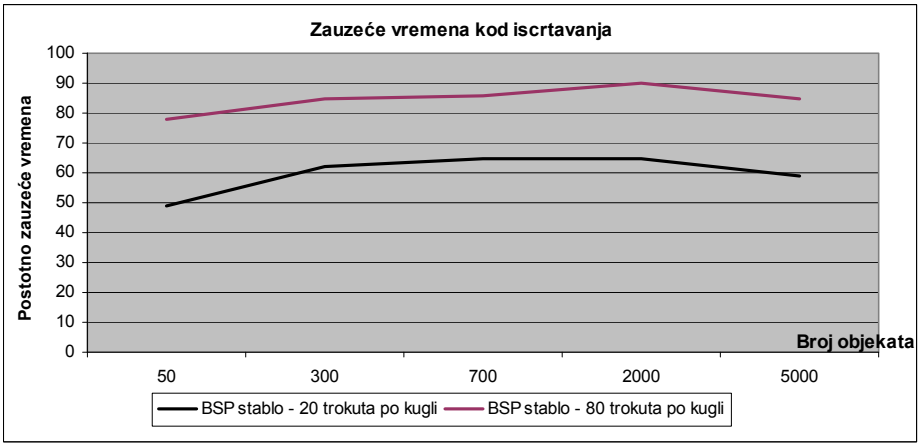
Slika 8.20 BSP stablo i samo-podesivo BSP stablo za scenu „pad“.

8.5 Utjecaj različitog broja trokuta

Kako se svaka kugla može iscrtati pomoću 20 ili pomoću 80 trokuta, ispitano je kako četverostruko povećanje broja trokuta utječe na brzinu iscrtavanja, te vrijeme koje se troši za iscrtavanje. Ispitivanje je provedeno za metodu BSP stablo na sceni sa slučajnim rasporedom objekata.



Slika 8.21 Graf zavisnosti brzine iscrtavanja i broja objekata
kod slučajnog rasporeda objekata te 20 i 80 trokuta po kugli (BSP).

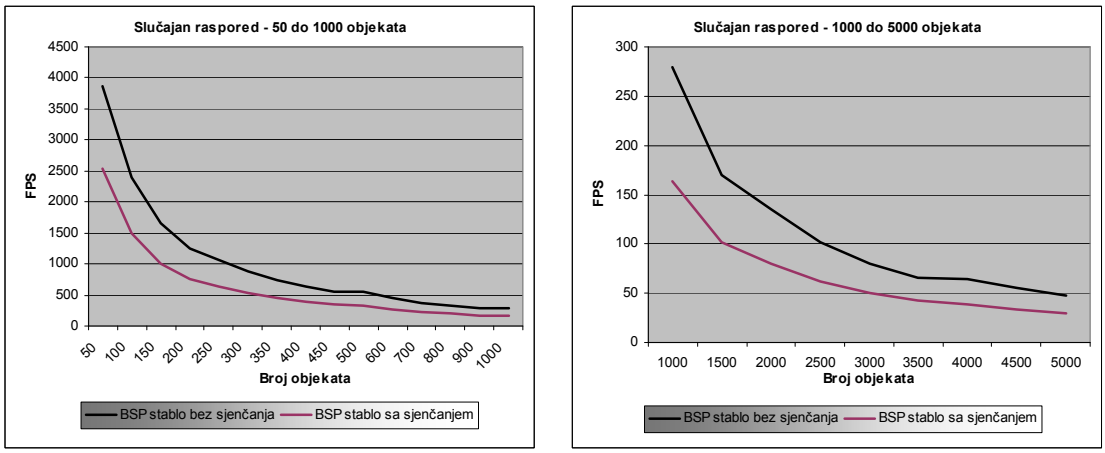


Slika 8.22 Graf zauzeća vremena tijekom iscrtavanja
u ovisnosti o broju objekata za 20 i 80 trokuta po kugli.

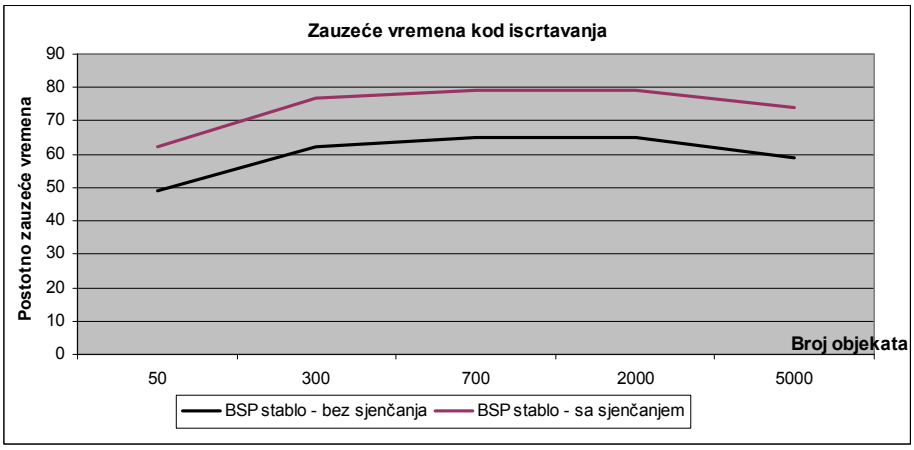
Iz rezultata mjerenja se može vidjeti da povećanje broja trokuta na sceni 4 puta rezultira smanjenjem brzine iscrtavanja od otprilike 66%, dok zauzeće programa sa samim iscrtavanjem se povećava za otprilike 25%.

8.6 Utjecaj sjenčanja

Također je ispitano kako uključivanje sjenčanja svih kugli u sceni utječe na brzinu iscrtavanja, te vrijeme koje se troši tada za iscrtavanje. Ispitivanje je provedeno za metodu BSP stablo na sceni sa slučajnim rasporedom objekata.



Slika 8.23 Graf zavisnosti brzine iscrtavanja i broja objekata
kod slučajnog rasporeda objekata te uključenog i isključenog sjenčanja (BSP).



Slika 8.24 Graf zauzeća vremena tijekom iscrtavanja
u ovisnosti o broju objekata uključeno i isključeno sjenčanje.

Iz rezultata mjerenja se može vidjeti da uključivanje sjenčanja rezultira smanjenjem brzine iscrtavanja od otprilike 40%, dok zauzeće programa sa samim iscrtavanjem se povećava za otprilike 14%.

8.7 Usporedba sa postojećim radovima

Ovaj rad je većinom nastao na temelju rada [1]. U tom radu se opisuje prostorna struktura polu-podesivog BSP stabla (eng. *semi-adjusting BSP-tree*) koja se od samo-podesivog BSP stabla (eng. *self-adjusting BSP-tree*) razlikuje po tome što ravnine podjele računa na složeniji način te one nisu limitirane na *x-y*, *y-z* i *x-z* ravnine. Kako je uz to i vrijeme koje se može potrošiti na podešavanje stabla limitirano, ako postoji potreba za većim restrukturiranjem stabla, odabir dijela novih ravnina podjele se odgađa do sljedećeg podešavanja stabla – zbog tog odgađanja se stablo naziva polu-podesivim (eng. *semi-adjusting*). Osim toga se za podešavanje stabla u tom radu koristi više operatora, tj. ne samo operator balansiranja čvora.

U ovom radu se tijekom svakog podešavanja, ako je potrebno, restrukturira cijelo stablo, tj. ne vrši se odgađanje, no kako bi se smanjilo vrijeme koje se troši na podešavanje stabla, novi položaji ravnina podjele djece rebalansiranog čvora se računaju na jednostavniji način.

Ideja za implementaciju BSP stabla je preuzeta iz [2], dok je osnovna ideja za implementaciju algoritma brojenja parova i pojednostavljivanja preuzeta iz [1] i [3].

Rezultati simulacija se većinom slažu sa rezultatima dobivenim u [1]. U tome radu su performanse polu-podesivog BSP stabla uspoređene sa performansama nekoliko algoritama koji se koriste u širokoj fazi detekcije sudara, uključujući i algoritam brojenja parova i pojednostavljivanja. I tamo je algoritam brojenja parova i pojednostavljivanja kreirao najmanje potencijalnih kolizijskih parova koje detaljnije treba provjeriti za koliziju, no to je činio uz relativno visok trošak. Zbog toga su i tamo performanse tog algoritma kod većeg broja objekata zaostajale za performansama polu-podesivog BSP stabla.

9. Zaključak

Rezultati simulacija su potvrdili većinu pretpostavki. Klasična metoda detekcije sudara se je pokazala vrlom sporom i neprimjenjivom u širokoj fazi detekcije sudara većeg broj objekata. BSP stablo, samo-podesivo BSP stablo te algoritam brojenja parova i pojednostavljivanja su konstantno pokazivali bolje performanse od klasične metode. Kod većeg broja objekata i njihovog slučajnog rasporeda bez iscrtavanja, faktori ubrzanja su iznosili redom čak 47 (BSP stablo), 40 (samo-podesivo BSP stablo) i 3,33 (algoritam brojenja parova i pojednostavljivanja).

Algoritam brojenja parova i pojednostavljivanja je pokazao najbolje performanse (najveće faktore ubrzanja iscrtavanja u odnosu na klasičnu metodu) kod manjeg broja objekata (do 500), te općenito, na scenama gdje gustoća objekata nije velika te postoji malo preklapanja projekcija objekata po osima i nije potrebno raditi puno stvarnih testova kolizije. Pri većem broju objekata raste broj preklapanja i broj testova kolizije pa se performanse te metode pri većem broju objekata približavaju klasičnoj metodi.

BSP stablo te samo-podesivo BSP stablo pokazuju najbolje performanse kod većeg broja objekata (>500). Kod slučajnog rasporeda objekata BSP stablo pokazuje najbolje performanse, jer tijekom cijele simulacije može vršiti dovoljno dobru podjelu prostora i bez samo-podešavanja. Kod detekcije sudara većeg broja objekata gdje se raspored objekata tijekom simulacije znatno razlikuje od početnog, preporučljivo je koristi samo-podesivo BSP stablo jer ono daje najbolje faktore ubrzanja.

10. Popis slika

11. Literatura

[1] Luque R.G., Comba J.L.D., Freitas C.M.D.S.:
Broad-phase collision detection using semi-adjusting BSP-trees. Proceedings of the 2005 symposium on Interactive 3D graphics and games (2005), str. 179-186.
www.inf.ufrgs.br/~comba/papers/papers.html, 17.6.2008

[2] BSP Tree Frequently Asked Questions (FAQ)
ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html, 17.6.2008

[3] Cohen J.D., Lin M.C., Manocha D., Ponamgi M.:
I-COLLIDE: an interactive and exact collision detection system for large-scale environments. Proceedings of the 1995 symposium on Interactive 3D graphics (1995), str. 189-196,.
http://www.cs.unc.edu/~dm/collision.html, 17.6.2008

[4] Pfaff Raman: The Physics of an Elastic Collision
http://director-online.com/buildArticle.php?id=532, 17.6.2008

[5] Petzold C.: Programming Windows, Fifth Edition, Microsoft Press, 1998.

[6] Astle D., Hawkins K.: Beginning OpenGL Game Programming, Course Technology PTR, 2004.

[7] Woo M., Neider J., Davis T., Shreiner D.:
OpenGL® Programming Guide: The Official Guide to Learning OpenGL, Addison-Wesley Professional, 2005.

[8] C/C++ Program Code for 3D Elastic Collision of 2 Balls,
http://www.plasmaphysics.org.uk/programs/coll3d_cpp.htm, 17.6.2008

12. Sažetak / Abstract

U ovom radu su opisane strukture podataka pogodne za izradu i analizu detekcije sudara u širokoj fazi. U okviru programskog dijela rada opisane strukture podataka i algoritmi su implementirani u programskom jeziku C++ pomoću razvojnog okruženja Microsoft Visual Studio 2005™. Kao standard za prikaz korištena je OpenGL grafička biblioteka. Na trodimenzijskim primjerima su ocijenjeni pojedini algoritmi te ocijenjeni slučajevi za koje pojedini algoritam pokazuje najbolje performanse.

Ključne riječi: BSP stablo, samo-podesivo BSP stablo, kd stablo, algoritam brojenja parova i pojednostavljivanja, detekcija sudara, računalna grafika, računalna animacija

This paper describes data structures suitable for implementation and analysis of broad phase collision detection. As part of practical work, the described algorithms and data structures were implemented in C++ programming language using the Microsoft Visual Studio 2005™ programming environment. OpenGL graphics library was used for displaying simulations. Individual algorithms and their performance were compared using various 3D examples.

Key words: BSP tree, self-adjusting BSP tree, kd tree, sweep and prune algorithm, collision detection, computer graphics, computer animation