SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1763

INTERAKTIVNI POSTUPCI NAD OKTALNOM STRUKTUROM GRAFIČKIH PODATAKA

Davor Čaktaš

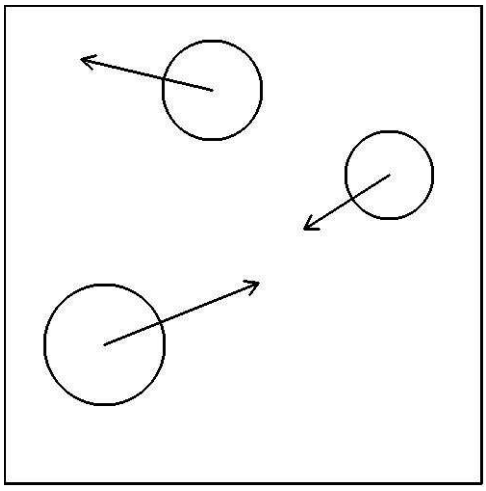
Zagreb, listopad 2008.

Sadržaj

1. Uvod

Kod računanja sudara je posebno važno imati dobre algoritme i strukture podataka koji će proces računanja skratiti što je više moguće, a da se pri tome ne izgubi na preciznosti rezultata. Ne želite čekati satima da dobijete rješenje koje možda neće biti zadovoljavajuće. Također se u većini slučajeva očekuje da proces teče glatko i u stvarnom vremenu. U slučaju da gradite strukturu podataka koja bi vam trebala pomoći u ubrzanju procesa ne možete uvijek proces izgradnje posve prepustiti računalu. U nekim slučajevima morate imati mogućnost pregleda i popravke stvorene strukture. Najlakši način za to je grafički ju prikazati i mijenjati.

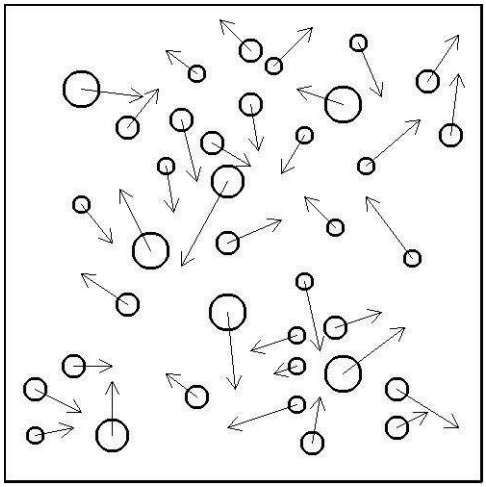
1.1. Jednostavan primjer



Slika 1.1. Jednostavan model sudara pravilnih tijela

Na slici 1.1 je prikazan jednostavan zatvoreni modeltijela koja se sudaraju. Tri tijela kružnog oblika koja se pravocrtno kreću u dvodimenzionom prostoru. Imaju određenu brzinu i smjer. Pri sudaru s drugim tijelom ili zidom slijedi odbijanje prema jednostavnom zakonu odbijanja. Ljudski mozak može na veoma jednostavan način pratiti i predviđati putanje zadanih tijela. Upotreba računala gotovo da nije ni potrebna osim u slučaju preciznih rezultata. Za današnja

računala ovakvi jednostavni modeli ne predstavljaju nikakav problem. Sami algoritmi za simulaciju ovog modela su izrazito jednostavni.



Slika 1.2. Složeniji model sudara pravilnih tijela

Još jedan zatvoreni model prikazan je na slici 1.2. Ista situacija kao u prethodnom primjeru no ovaj put s više tijela. Za ljudski mozak već skoro nemoguće praćenje i predviđanje putanja tijela nakon sudara. Ovaj put je upotreba računala neizbježna. Računalo problem rješava na isti način kao i prošli. Situacija se nije promijenila, algoritam je isti. Unesen je samo veći broj varijabli i slijedi veći broj računanja. No, ovo je samo jednostavan primjer.

1.2. Stvaran problem

U današnjoj industriji je izuzetno bitno krenuti od simulacije. Na taj se način mogu izbjeći mnoge pogreške koje bi uzrokovale velike novčane gubitke. Simulacija otkriva pogreške i mane modela, model se ispravlja ili, u najgorem slučaju, radi ispočetka te se ponovno simulira.

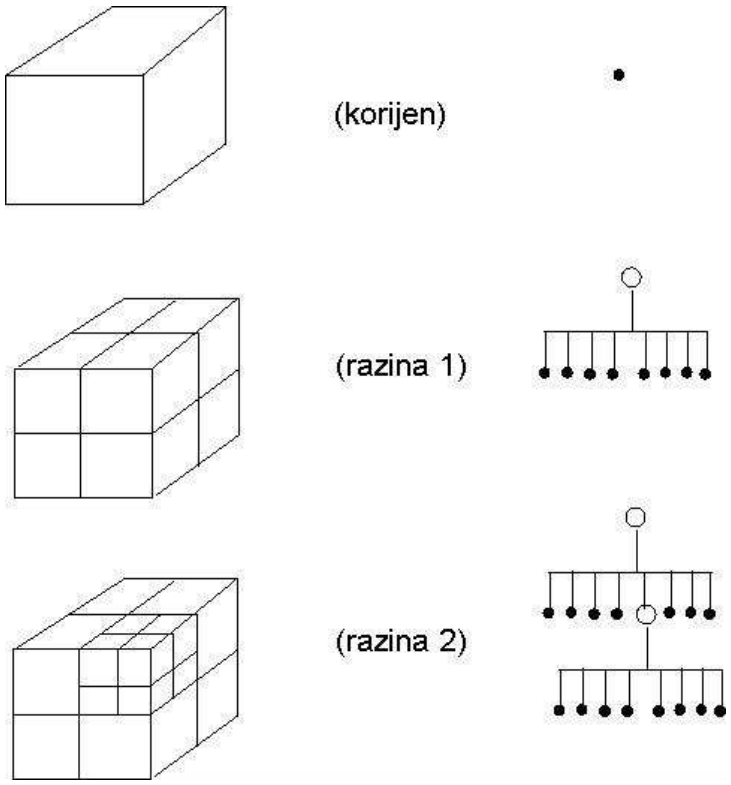
Kao konkretan primjer možemo uzeti automobilsku industriju i projektiranje motora. Motor je izrazito složena komponenta bilo kojeg vozila. Sastoji se od mnoštva pokretnih i nepokretnih dijelova. Kroz njega prolaze razne tekućine, od

goriva do rashladnih tekućina. Svaka komponenta se može promatrati i simulirati kao zaseban zatvoreni sustav ili kao dio cjeline.

Da bismo bilo koju komponentu mogli simulirati moramo prvo napraviti računalnu aproksimaciju te komponente. Iako dosta ovisi o prirodi problema za kruta tijela će to najčešće biti neka mreža dok će za tekućine biti skup čestica. Da bi simulacija bila što vjerodostojnija, a samim tim i korisna, potreban je velik broj elemenata. To za sobom povlači i velik broj izračuna koje računalo mora obaviti u toku simulacije. Da bi nam simulacija bila korisna simulacija se mora završiti u nekom prihvatljivom konačnom vremenu. S nekim jednostavnim algoritmom koji bi svaki element usporedio sa svim ostalim elementima zbog, naprimjer, utvrđivanja kolizije to nije moguće. Javlja se potreba za boljim i bržim algoritmima te podatkovnim strukturama kao potpori tim algoritmima.

2. Oktalno stablo

2.1. Osnova



Slika 2.1. Oktalno stablo podijeljeno do druge razine

Oktalno stablo (engl. octree) je, kako i samo ime nalaže, stablasta struktura podataka. Svaki čvor se širi na osam elemenata koji mogu biti listovi ili daljnji čvorovi. Fizikalna aproksimacija oktalnog stabla je kocka koja obuhvaća cijeli prostor (zatvoreni sustav). Ta kocka predstavlja korijen stabla. Sljedeću razinu dobivamo podjelom kocke na osam jednakih dijelova kako je prikazano na slici. Svaka od novonastalih kocaka je korijen sljedeće razine ukoliko nastavimo s podjelom. U suprotnom predstavlja list trenutne razine.

Osnovna svrha oktalnog stabla je podjela prostora. Prostor se usitnjava podjelom stabla po razinama. Svaka razina označava veću podjelu i manju jedinicu prostora. Samim tim usitnjavanjem se smanjuje broj elemenata po jedinici prostora. Svaka kocka, bilo čvor ili list stabla, grupira susjedne elemente na tom

dijelu prostora. Samim tim se smanjuje broj računanja međudjelovanja između pojedinih elemenata. Kao primjer toga se može uzeti kolizija.

2.2. Primjena kod kolizije

Osnovni i najjednostavniji algoritam računanja kolizije je uspoređivanje svakog elementa sa svakim. Kod malog broja elemenata algoritam daje dobre rezultate no kako složenost ovog algoritma raste eksponencijalno možemo zaključiti da kod praktičnih problema gdje broj elemenata iznosi nekoliko milijuna jedinki algoritam više nije ekonomičan.

Oktalna struktura nam ovdje pomaže. Kako smo već rekli osnovna zadaća oktalne strukture je podjela prostora na manje cjeline i grupiranje susjednih elemenata. Nemamo više potrebu za uspoređivanje svakog elementa sa svakim jer već na početku znamo da određeni elementi nisu dovoljno blizu jedan drugome da bi njihovo međudjelovanje bilo moguće.

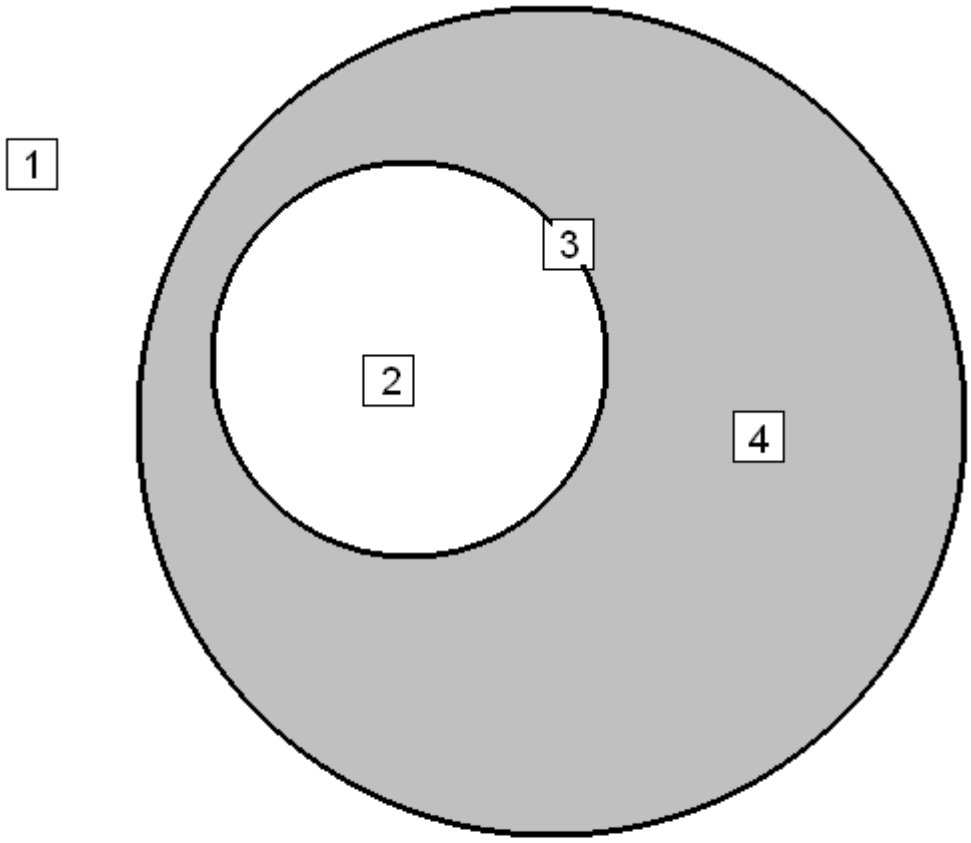
U nekom idealnom slučaju podjele stablaste strukture dobili bismo potpuno izgrađeno stablo. To znači da bi cijeli prostor bio jednoliko usitnjen. U stvarnim situacijama nam to nije potrebno. Ne možemo očekivati da će elementi biti jednoliko raspršeni po prostoru. Na nekim mjestima će ih biti više dok ih na nekim neće biti uopće. Upravo zbog toga neke dijelove prostora, odnosno neke grane stabla ne trebamo dalje dijeliti dok nam je kod drugih podjela još uvijek potrebna. Ako imamo velik dio praznog prostora ili mali broj elemenata na nekom većem prostoru dovoljna nam je i jedna velika kocka koja će sve to obuhvatiti. Suprotno od toga ako je velik dio elemenata grupiran na malom prostoru taj će prostor zahtijevati veću odnosno sitniju podjelu.

2.3. Dodatni način raspodjele oktalnog stabla

Nakon što je podjela obavljena jedini elementi stabla koji nas konkretno zanimaju su listovi stabla. Čvorovi stabla nam služe samo pri pretrazi pojedinih elemenata i pri uvidu u odnose između kocaka. U pojedinim slučajevima moramo

preciznije definirati podjelu prostora. Već smo rekli da jednaka razina podjele na cijelom prostoru nije dobra. U nekim slučajevima nam ni potpuno nepravilna podjela ne pomaže u dobivanju željenih rezultata. Ono što nam je potrebno je do određene razine polupravilna raspodjela prostora, tj. dodajemo određene uvjete pri čemu specificiramo da, npr, razlika razina raspodjele između dvije susjedne kocke ne smije biti veća od jedan.

Ukoliko dijelimo prostor s nekom mrežom u sebi može se javiti potreba klasifikacije listova stabla. Kako smo rekli listovi stabla unutar sebe mogu, ali i ne moraju sadržavati elemente (u ovom slučaju dijelove mreže). S listovima koji sadrže nešto unutar sebe je sve jasno, ali ne i s onima koji obuhvaćaju prazan prostor. Njih možemo klasificirati u tri dodatne skupine: vanjski, unutarnji i okruženi. Vanjski je onaj koji se nalazi izvan mreže, unutarnji onaj koji se nalazi unutar mreže, a okruženi je onaj koji se nalazi izvan mreže, ali je njom okružen što je prikazano na slici 2.2.



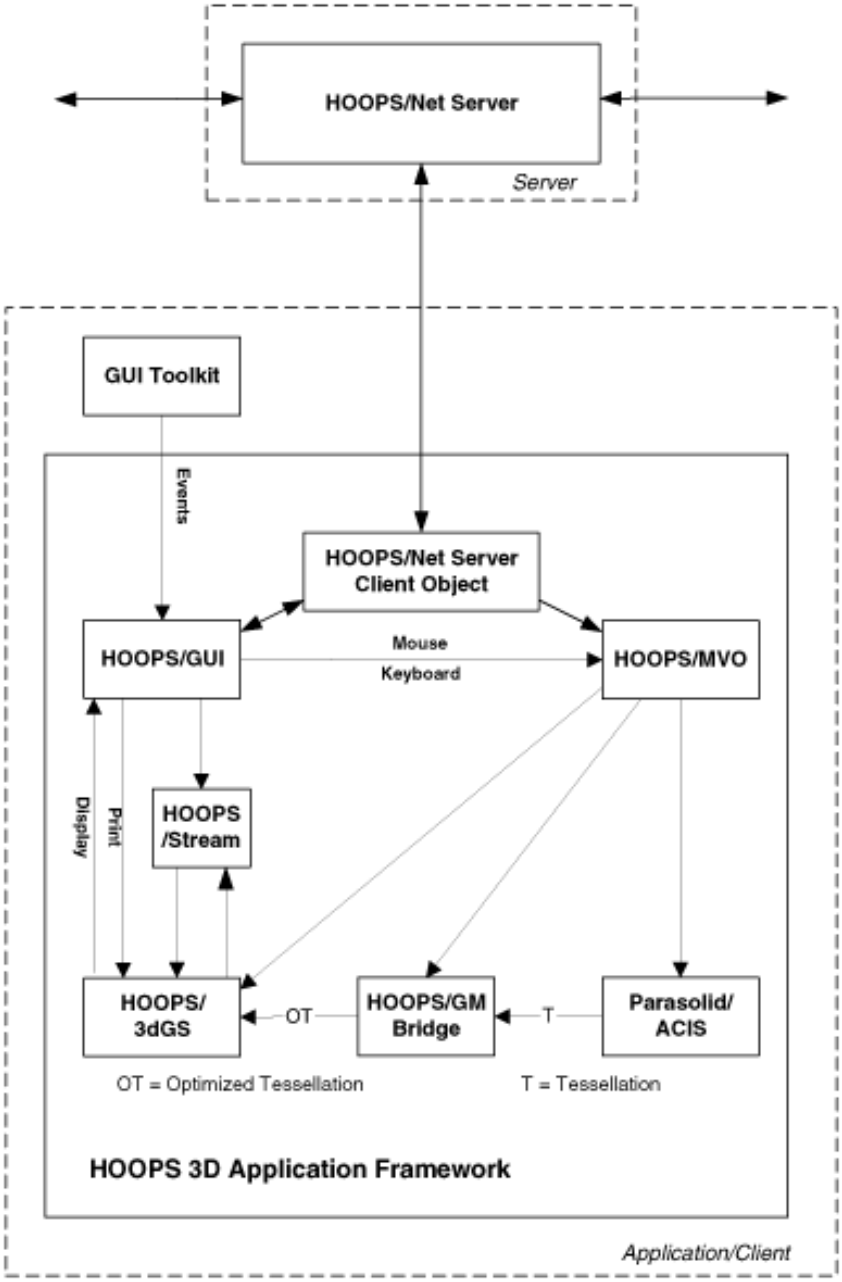
Slika 2.2 Klasifikacija listova oktalnog stabla prema smještaju u prostoru u odnosu na mrežu

3. Grafički paket HOOPS/3dAF

HOOPS 3D Programski Okvir (engl. HOOPS 3D Application Framework) se sastoji od integriranog niza komponenti za ostvarenje brzog razvoja dizajna, vizualizacije i inženjerskih programa visoke učinkovitosti. HOOPS/3dAF održava neovisnost komponenti o platformi pružajući međuplatformska rješenja na operativnim sustavima Windows, Unix, Linux i Mac OS X.

Sastoji se od sljedećih komponenti:

- HOOPS/3dGS – HOOPS 3D Grafički Sustav
- HOOPS/Stream – HOOPS alat za podatkovne nizove
- HOOPS/MVO – knjižnica klasa modela, pogleda i operatora
- HOOPS/GUI – moduli za grafičko korisničko sučelje
- HOOPS/GM – poveznice za geometrijsko modeliranje
- HOOPS/Net – alat za klijent/server programe
- HOOPS/3dAF Referentni Programi – međuplatformski programi temeljeni na geometrijskoj i ne-geometrijskoj jezgri



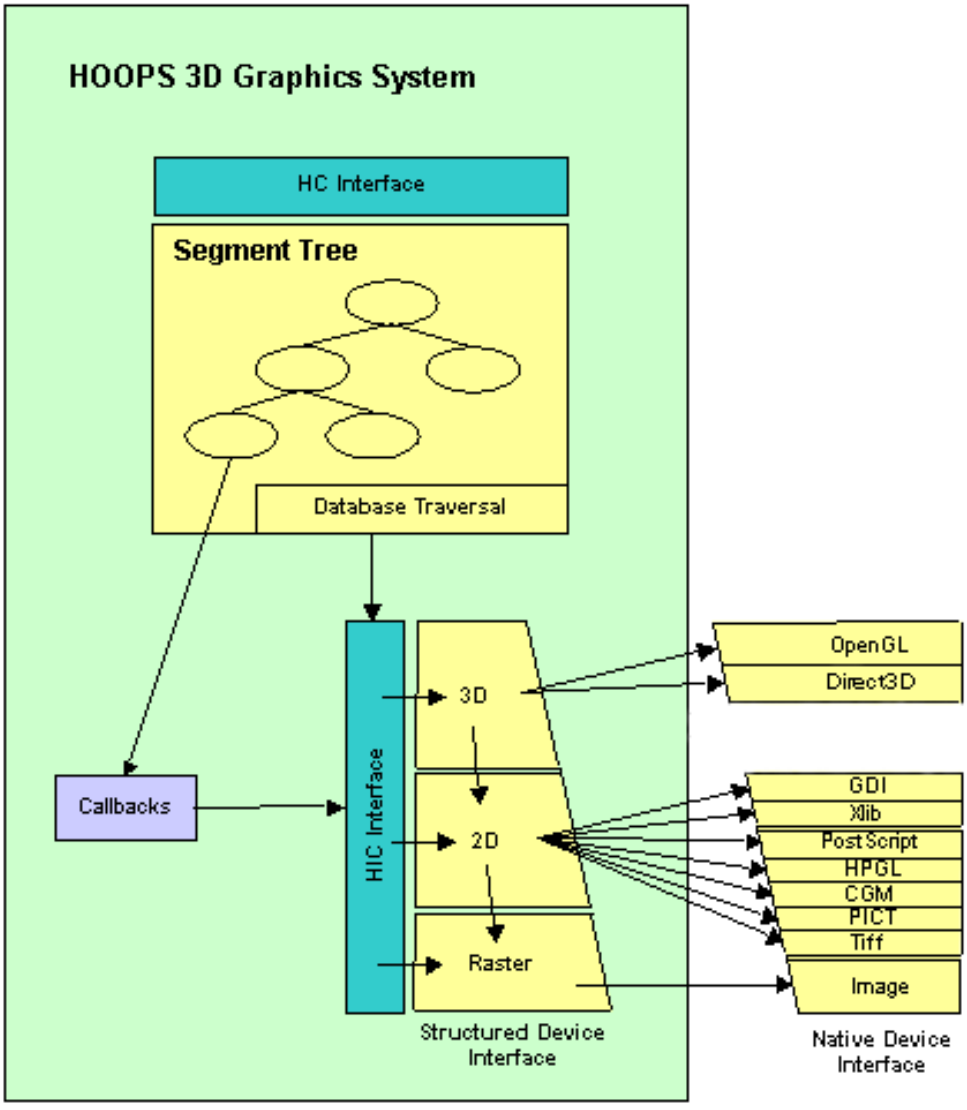
Slika 3.1. Ilustracija međusobne povezanosti HOOPS 3D komponenti

3.1. HOOPS/3dGS

Hoops/3dGS je komercijalni grafički sustav koji nam nudi algoritme i strukture podataka potrebne za unošenje 2D i 3D, vektorske i rasterske grafike u interaktivne inženjerske aplikacije. Sastoji se od knjižnice podrutina koje nude stvaranje, manipulaciju, ostvarivanje prikaza (engl. rendering) i još mnoge druge

mogućnosti za grafičke informacije aplikacija, a spaja se sa aplikacijskim objektnim kodom. Hoops nije aplikacija već prije alat koji nudi široki pojas 2D i 3D interaktivnih grafičkih mogućnosti.

HOOPS/3dGS grafički sustav (engl. HOOPS/3dGS Graphic System) se sastoji od dvije glavne podkomponente: podatkovnu bazu grafičkih objekata imena HOOPS/3dGS stablo segmenata (engl. HOOPS/3dGS Segment Tree) i protočnu strukturu ostvarivanja prikaza imena HOOPS/3dGS sučelje strukturiranih uređaja (engl. HOOPS/3dGS Structured Device Interface).



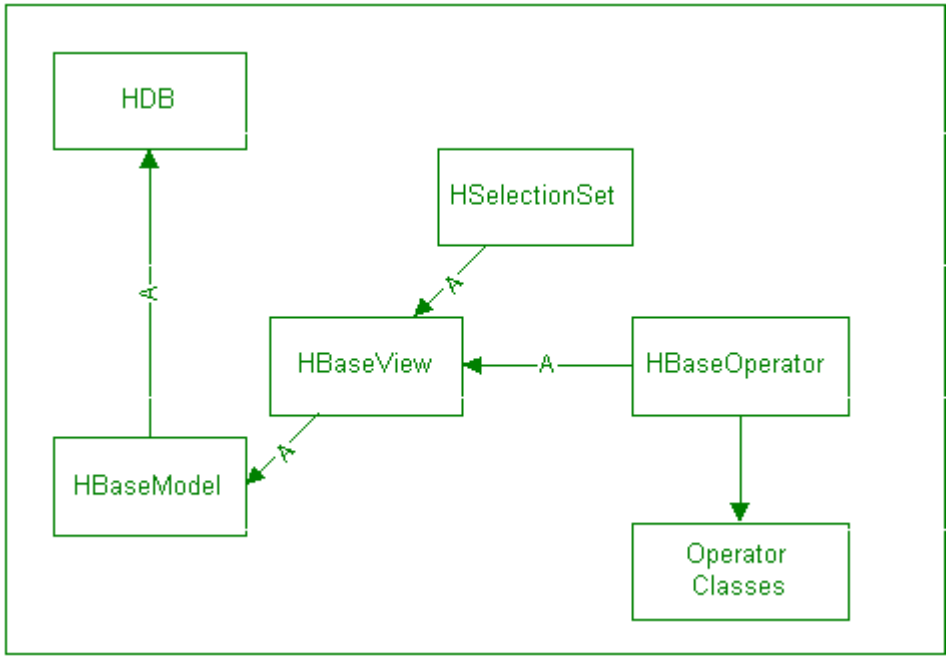
Slika 3.2. Ilustracija HOOPS/3dGS grafičkog sustava.

Hoops/3dGS knjižnica je implementirana direktno u aplikacijski proces izgradnje (engl. build process). Spojena je s aplikacijskim objektnim kodom tako da stvori izvršnu sliku.

3.2. HOOPS/MVO

Hoops/MVO je skup C++ objekata neovisnih o platforni za implementaciju uobičajenih funkcionalnosti pronađenih u CAD/CAM/CAE aplikacijama poput stvaranja, prikaza i manipulacije modelom. Klase su implementirane upotrebom Hoops3D API grafičkog sustava.

HOOPS/MVO objekti mogu biti direktno ugrađeni u program ili korisnički definiran objekt ili mogu biti temelj za izgradnju korisnički definiranih objekata. HOOPS/MVO klase nude specifičnu implementaciju paradigme modela, pogleda i operatora što se može direktno implementirati u program i proširiti.



Slika 3.3 Odnos raznih HOOPS/MVO klasa

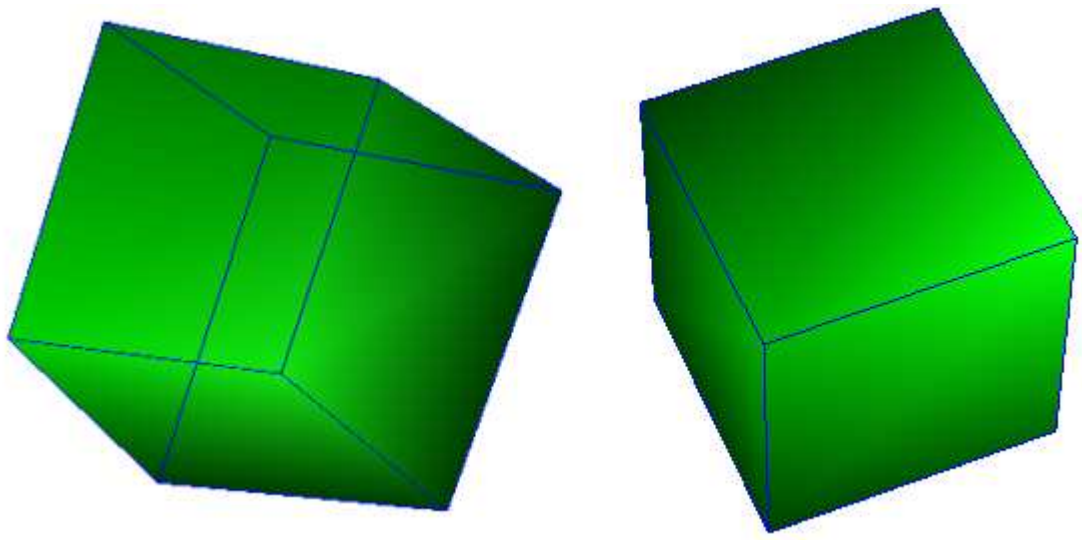
4. Prikaz oktalne strukture pomoću grafičkog paketa HOOPS 3D

4.1. Prikaz jedne kocke

Uobičajen način crtanja kocke ili bilo kojeg drugog elementa poput kugle, valjka, piramide i dr. jeste crtanje mreže trokuta koja će to tijelo obuhvatiti i definirati. Pošto je HOOPS 3D alat više razine od OpenGl-a i DirectX-a osim ovakvog načina crtanja postoji i napredniji način, tj. postoje već ugrađene funkcije za crtanje jednostavnijih oblika. To nije samo jednostavna mreža koju program stvori umjesto nas već posve novi objekt sa svojim svojstvima i mogućnostima. Nešto poput objekata u objektno orjentiranim jezicima.

```
key_box = HUtility::InsertBox( max, min );
```

Za stvaranje objekta kocke koristimo ovu naredbu. Njena namjena je stvaranje kvadra pomoću dvije suprotne prostorne koordinate. Laički rečeno, pomoću gornjeg lijevog stražnjeg i donjeg desnog prednjeg vrha kvadra. Posebno treba paziti na redoslijed zadavanja vrhova jer oni određuju orijentaciju normala na svakoj od šest strana kvadra, tj. određuju orijentaciju samog kvadra.

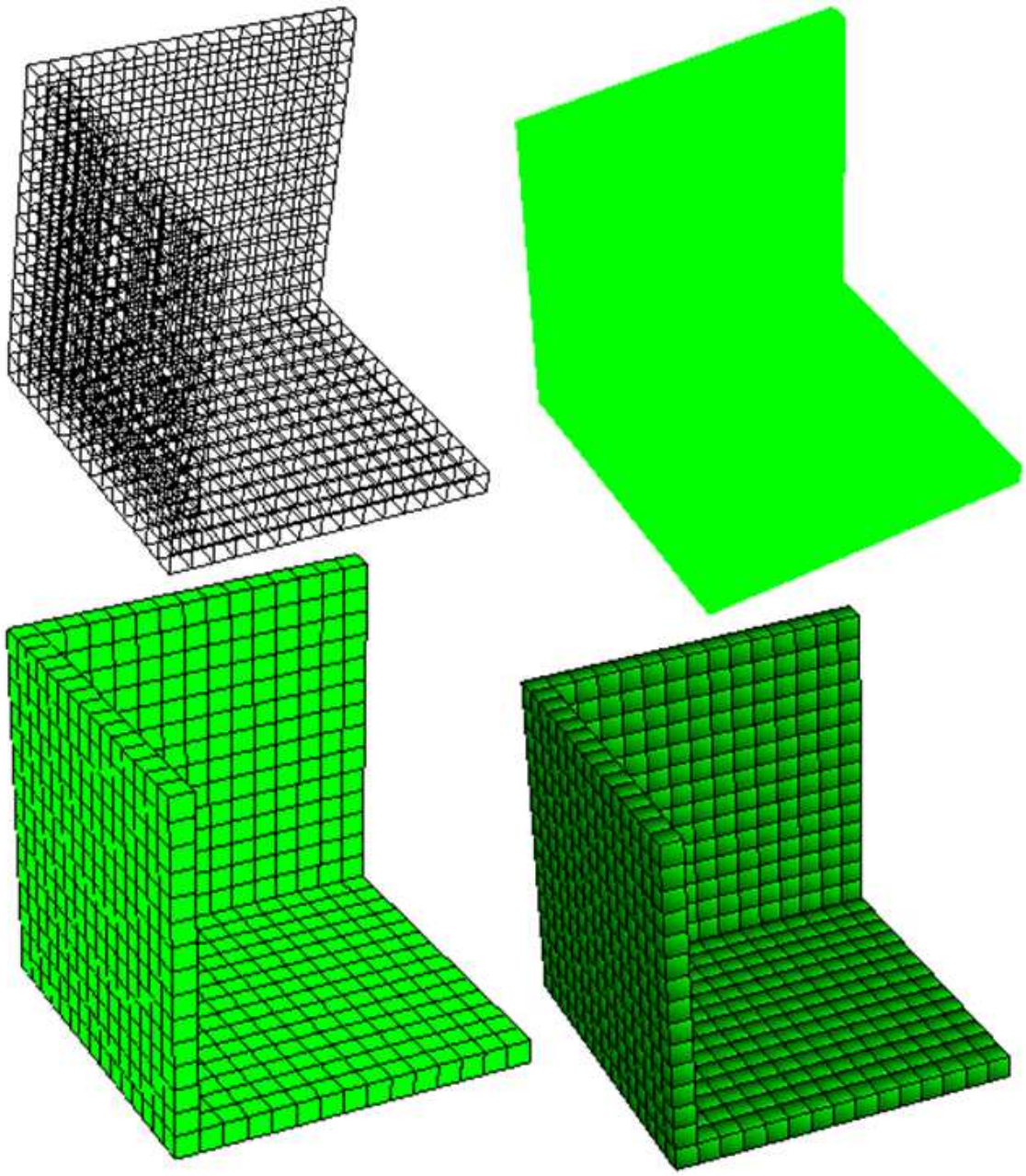


Slika 4.1 Iscrtane kocke s normalama stranica orijentiranim prema unutra i van

Ovako stvorene kocke nisu samo mreža trokuta spojenih na način da predstavljaju kocku. Čitava mreža s osvjetljenjem, rubovima, prikazom, vlastitim lokalnim koordinatnim sustavom i dr. čini jedan objekt, odnosno jednu geometriju. Ova geometrija se dalje nalazi unutar nekog segmenta koji predstavlja skup geometrija i drugih segmenata koji su na neki način međusobno povezani. Čitav prikaz u samom HOOPS 3D alatu je jedan segment. Na ovaj način HOOPS 3D raščlanjuje sve elemente prikaza. Može se reći da je čitav prikaz stablasta struktura segmenata i geometrija.

4.2. Manipulacija svojstvima objekata

Sve se opcije, poput boje, osvjetljenja, prikaza pojedinih elemenata i dr, mogu mijenjati na razini segmenta ili geometrije. Ako postavimo neke opcije na vrhovnom segmentu svi podsegmenti i njihove geometrije će naslijediti te opcije. No, ako promijenimo neku opciju na nižoj razini, bilo na nižem segmentu ili geometriji, ta opcija će prevladati nad opcijom zadanoj na višoj razini, odnosno u roditeljskom segmentu. Problemi nastaju kad ponovno pokušamo promijeniti neku od opcija u roditeljskom segmentu koju smo već mijenjali na segmentu ili geometriji niže razine. Promjena na segmentu više razine neće nadjačati promjenu na nižoj razini. Da bismo to postigli moramo onemogućiti promjenu na nižoj razini.



Slika 4.2 Neki od mogućih načina prikaza iste strukture elemenata

Na slici su prikazani neki rezultati manipilacije samo tri svojstva, prikaza ruba (engl. edge), prikaza poligona (engl. face) i korištenja osvjetljenja. Korišteni kod za ovu manipulaciju je sljedeći:

```
HC_Set_Visibility( "edges=on, faces=off, lights=off " );
HC_Set_Visibility( "edges=off, faces=on, lights=off" );
HC_Set_Visibility( "edges = on, faces = on, lights = off" );
HC_Set_Visibility( "edges = on, faces = on, lights = on" );
```

Primijećeno je da ovakva manipulacija svojstava ima velikog utjecaja na brzinu iscrtavanja. Jednostavna promjena iscrtavanja rubova objekta može imati velikog utjecaja na brzinu iscrtavanja ovisno o broju objekata koje se iscrtava.

4.3. Selekcija pojedinih kocaka oktalnog stabla

Selekcija prikazanih objekata unutar HOOPS 3D prikaza nije tako složena kako bi se očekivalo. Dok bismo kod OpenGl-a sami morali računati selekciju kod HOOPS 3D alata je ona gotovo automatska. Sve što je potrebno je izabrati željenu selekciju i unijeti potrebne koordinate. Koordinate koje se unose su iz lokalnog koordinatnog sustava pogleda. HOOPS 3D će napraviti potrebnu pretvorbu iz lokalnog u globalni koordinatni sustav.

Osnovna naredba za selekciju je sljedeća:

```
int pom = HC_Compute_Selection( "?picture", 0, "^", event.GetMouseWindowPos().x,
event.GetMouseWindowPos().y );
```

Ova naredba služi samo za pokretanje procesa selekcije. Predajemo joj koordinate na kojima želimo izvršiti selekciju i daljnji posao prepuštamo njoj. Nakon što se ona izvrši pokrećemo proces dohvata rezultata. Ova naredba nam ne vraća rezultat selekcije. Naredba vraća samo oznaku da li je selekcija bila uspješna, odnosno da li je nešto selektirano ili ne.

Proces dohvata rezultata selekcije je sljedeći:

```
if( pom > 0 )
{
    HC_KEY* key = new HC_KEY[ pom ];
    int* d1 = new int[ pom ];
    int* d2 = new int[ pom ];
    int* d3 = new int[ pom ];
    HC_Show_Selection_Element( key, d1, d2, d3 );
}
```

Nakon provjere da li je uopće došlo do selekcije možemo tražiti konkretnu selekciju. Rezultat selekcije dobivamo pomoću skupa naredbi „Show_Selection()“. Ove naredbe nam služe za povrat informacija o izvedenim selekcijama.

Naravno, osim selekcije jednog elementa možemo vršiti i selekcije većeg broja elemenata. Navedena naredba nam neće selektirati samo jedan element već će nakon pronalaska jedne selekcije nastaviti pretragu za drugim elementima koji se nalaze na istim koordinatama. Jedina razlika ove selekcije i selekcije jednog elementa je u dohvatu rezultata:

```
if( pom > 0 )
{
    do
    {
            HC_KEY* key = new HC_KEY[ pom ];
            int* d1 = new int[ pom ];
            int* d2 = new int[ pom ];
            int* d3 = new int[ pom ];
            HC_Show_Selection_Element( key, d1, d2, d3 );
    } while( HC_Find_Related_Selection() );
}
```

Jedina razlika je u petlji koja se vrti dok god u memoriji ima selektiranih elemenata. Rutina „Show_Selection()“ vraća informaciju o najboljem selektiranom elementu. Pod najboljim elementom se smatra onaj najbliži pokazivaču prilikom selekcije (koordinatama selekcije). Ako se ispod pokazivača nalazi više od jednog elementa onda je to onaj najbliži kameri (očištu). Rutina „Find_Related_Selection()“ traži sljedeću najbolju selekciju pri svakom pozivu i odbacuje trenutnu najbolju. Kad više nema selektiranih elemenata rutina vraća laž (engl. false).

Možemo zaključiti da ovaj algoritam nije optimalan za situacije kada želimo selektirati samo jedan element, a ispod pokazivača se nalaze milijuni drugih elemenata. Ovaj oblik selekcije bi bespotrebno zauzimao previše resursa i bespotrebno bi usporavao čitav proces. Da se takve stvari ne bi dešavale uvedena

je opcija ograničenja dubinske pretrage selekcije. Opcija se postavlja pomoću sljedeće naredbe:

```
void Set_Heuristics (const char *list)
```

„Heuristika“ u HOOPS-u je pretpostavka o prikazu koju programer predaje sustavu. Razlog postojanja ovih pretpostavki je smanjenje vremena potrebnog za osvježenje prikaza, smanjenje vremena potrebnog za spremanje podataka u bazu podataka ili veličina baze podataka potrebna da se svi ti podaci spreme. Generalno, da biste dobili što efikasniji rad programa trebate sustavu predati sve optimizacijske pretpostavke koje znate. Lista heuristika se sustavu predaje kao znakovni niz pri čemu su različite specifikacije odvojene zarezom.

Što se tiče specifikacija za selekcije, zanimljiva nam je samo jedna:

```
HC_Set_Heuristics( "related selection limit = broj_ograničenja" );
```

Ova specifikacija nam govori koliko daleko da idemo sa potragom za elementima koje možemo selektirati. Ako ništa ne odaberemo imat ćemo inicijalnu specifikaciju koja kaže da pretraga traje dok osim prvog elementa ne nađemo još dodatnih pet osim ako dođemo do kraja pretraživanja. Vidimo da specifikaciju ne možemo postaviti na nula elemenata jer bi takva selekcija bila beskorisna. Možemo naznačiti samo koliko daleko da se ide nakon prvog pronađenog elementa. Ukoliko želimo selektirati samo jedan element umjesto da stavljamo da je broj_ograničenja jednak nuli možemo napisati i sljedeću naredbu:

```
HC_Set_Heuristics( "no related selection limit" );
```

Kod naredbe „Set_Heuristics()“ postoji jedna značajna razlika od ostalih naredbi. Razlika je u mjestu pisanja ove naredbe. Dok se ostale naredbe pišu unutar segmenta modela „Set_Heuristics()“ se piše unutar segmenta pogleda. Ako se ipak napiše unutar segmenta modela među ostalim naredbama neće imati nikakvog utjecaja na rad samog programa. Dodatna primjedba na rad ove naredbe je ta da sustav ne mora prihvatiti specifikacije zadane putem ove naredbe.

Osim točkaste selekcije gdje selektiramo jedan element koji se nalazi direktno ispod pokazivača ili cijeli niz elemenata koji se sijeku sa pravcem koji prolazi kroz vrh pokazivača i okomit je na ekran imamo i druge vrste selekcija:

```
int Compute_Selection_By_Key (  const char *  action, const char *  start_seg, HC_KEY  key,
const float *  matrix )
```

Ova selekcija se koristi za računanje kolizije zbog svoje specifičnosti selektiranja. Naredbi se predaje volumen definiran svojom ljuskom na osnovu kojeg se dohvaćaju informacije svih objekata koji diraju zadani volumen.

```
int Compute_Selection_By_Shell (const char *action, const char *start_seg, const int pcount,
const HC_POINT *points, const int flist_length, const int *face_list)
```

Slično naredbi Compute_Selection_By_Key s tim da ne zahtijeva već postojeću ljusku.

```
int Compute_Selection_By_Ray (const char *action, const char *start_seg, const HC_POINT
*start_point, const HC_POINT *direction)
```

Zraka je definirana sa svojom početnom točkom i smjerom. Kao selekcija se uzimaju svi objekti „probodeni“ tom zrakom.

```
int Compute_Selection_By_Area (  const char *  display, const char *  start_seg, const char *
action, double  left, double  right, double  bottom, double  top )
```

Selekcija nad područjem definiranim pravokutnikom. Parametri „left“, „right“, „bottom“ i „top“ označavaju koordinate koje čine taj pravokutnik. Koordinata (left, top) čini gornji lijevi kut dok koordinata (right, bottom) čini donji desni kut pravokutnika. Pretpostavka je da su stranice pravokutnika paralelne sa stranicama zaslona.

```
int Compute_Selection_By_Polygon (  const char *  display, const char *  start_seg, const char
*  action, int  pcount, const HC_POINT *  points )
```

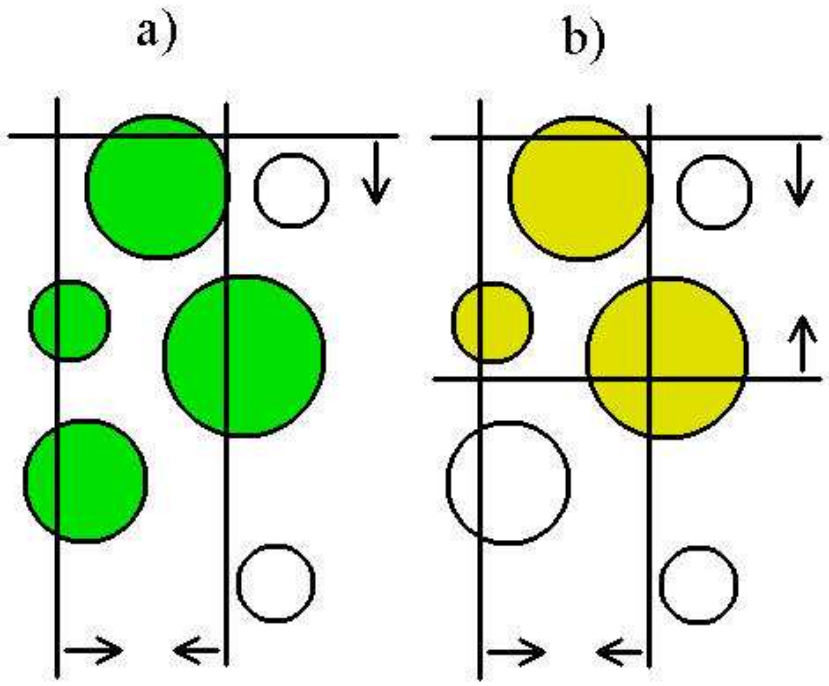
Slično kao kod prošle selekcije samo što područje selekcije nije pravokutnik već poligon.

int Compute_Selection_By_Polylin (const char * *display*, const char * *start_seg*, const char * *action*, int *pcount*, const HC_POINT * *points*)

Istovjetno sa prošlom selekcijom.

int Compute_Selection_By_Volume (const char * *display*, const char * *start_seg*, const char * *action*, double *left*, double *right*, double *bottom*, double *top*, double *hither*, double *yon*)

Slično Compute_Selection_By_Area s tim da u ovom slučaju selekcija nije određena plohom već volumenom. Dok smo kod selekcije s pravokutnikom selektirali sve što se nalazilo unutar pravokutnika neovisno o dubini selekcije ovdje smo i dubinski ograničeni.



Slika 4.3 Usporedba selekcija Compute_Selection_By_Area (primjer a) i Compute_Selection_By_Volume (primjer b)

5. Strukture podataka za ostvarivanje veze između prikazane i stvarne strukture oktalnoga stabla

Kako smo već rekli svaku kocku smo crtali pomoću naredbe:

```
key_box = HUtility::InsertBox( max, min );
```

Možemo vidjeti da je na lijevoj strani naredbe varijabla *key_box.* Ova varijabla je tipa HC_KEY što predstavlja identifikacijski broj svih objekata u HOOPS 3D grafičkom alatu. Svaki objekt, geometrija ili segment ima svoj identifikacijski broj koji je u stvarnosti broj tipa long. Ovaj broj je potreban za upravljanje objektima HOOPS 3D alatom. Preko njega možemo HOOPS-u zadavati naredbe za pojedine objekte, možemo im mijenjati svojstva, manipulirati njima i dr.

Sama oktalna struktura ima svoje identifikacijske brojeve. Svrha toga je da se različite kocke mogu raspoznavati jedna od druge i da se lakše nađe potrebna kocka. Svaki list kao i svaki čvor, koji je jednom prilikom procesa rafiniranja i sam bio list, imaju svoj jedinstveni identifikacijski broj.

Prilikom prikaza listova strukture svaki prikazani list (kocka) je dobio svoj HC_KEY. Postavlja se pitanje kako sada prilikom manipuliranja prikazom dobiti vezu između prikazanih kocki i onih smještenih u oktalnoj strukturi pomoću kojih je prikaz i napravljen. Obje strukture, i prikazana i stvarna, imaju svoje identifikacijske brojeve, ali to nisu isti brojevi. HOOPS ima mogućnost unošenja korisničkih atributa unutar samog modela. Na taj se način identifikacijski broj elementa strukture može spremiti direktno sa prikazanim objektom. Međutim u drugim alatima takvo što nije moguće te se javlja potreba za dodatnom strukturom podataka koja će služiti samo za ostvarivanje veze između stvarne strukture i njenog prikaza.

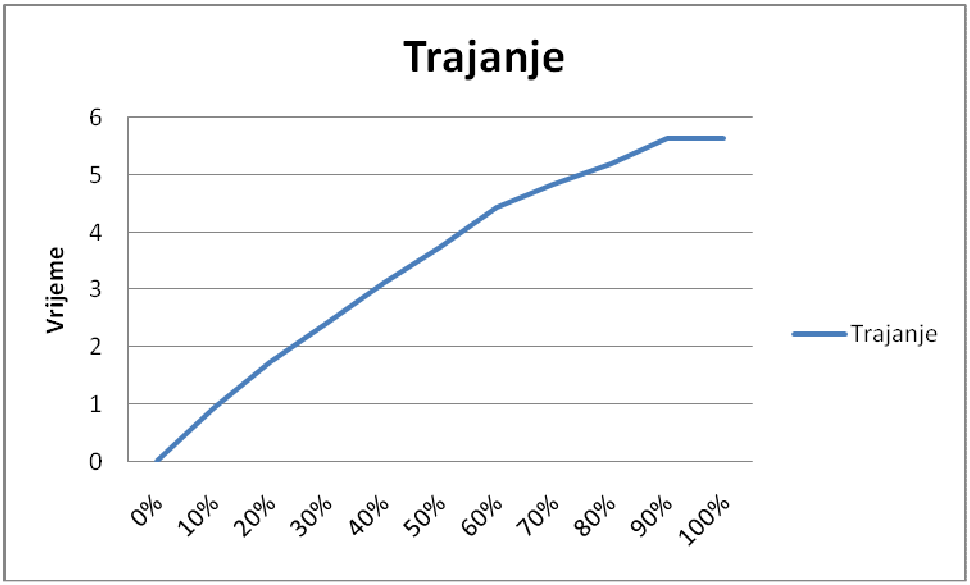
5.1. Polje

Najjednostavniji način ostvarivanja veze između stvarne strukture i njenog prikaza je polje. Stvara se dvostruko polje elemenata gdje se u prvom redu upisuju

identifikacijski brojevi elemenata prikaza, a u drugom identifikacijski brojevi elemenata strukture. Nije potrebno sortiranje elemenata jer će pretraga ići od početka prema kraju, element po element. Elementi ne moraju imati neprekinut niz identifikacijskih brojeva što onemogućuje direktan skok na zadani element. Postupak pretrage je moguće ubrzati određenim algoritmima, ali zbog prirode primjene nije ekonomično jer će se struktura nakon svake upotrebe uništavati i ponovo graditi.

Svaki put kad napravimo selekciju i nadogradimo stablo, tj. proširimo određenu granu za jednu razinu, unutar polja se gubi jedan element jer je taj list postao čvor i dobivamo osam novih elemenata. Dakle, moramo polje proširiti za sedam novih mjesta, obrisati selektirani element te ga zamijeniti novim i dodati još sedam ostalih novih listova.

Proširivanje polja znači njegovo uništenje i izgradnju novog, većeg, polja na drugoj memorijskoj lokaciji te kopiranje elemenata iz starog polja u novo. Stvaranje cijele jedne nove strukture zbog samo jednog elementa je izrazito neekonomično.



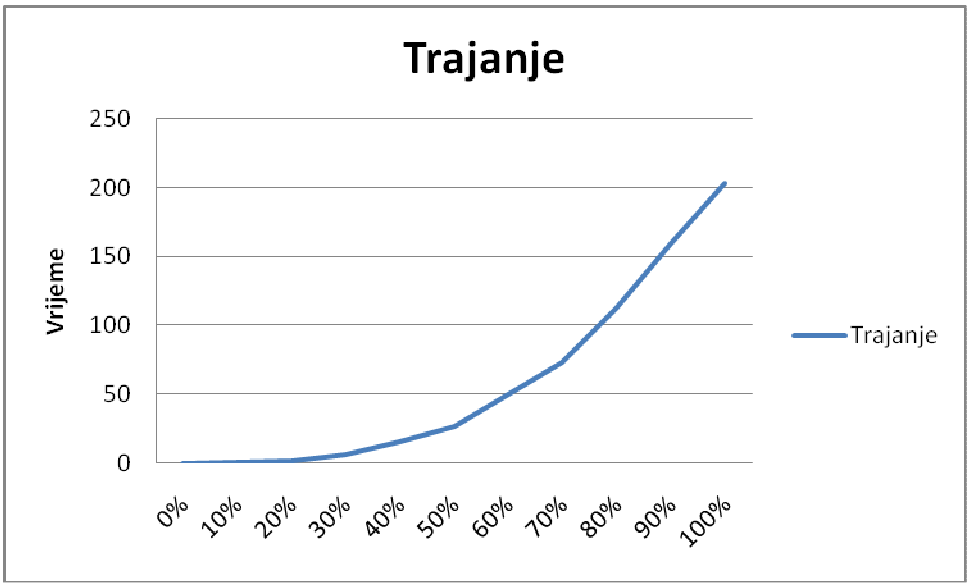
Slika 5.1 Graf ovisnosti broja mijenjanih elemenata o vremenu unutar polja

Na grafu se može vidjeti odnos broja elemenata koje je potrebno zamijeniti novim i trajanja postupka stvaranja nove strukture. Možemo primijetiti da se ekonomičnost postupka mijenja u ovisnosti o broju elemenata. Kako taj broj raste

tako imamo sve više elemenata koje moramo usporediti prilikom kopiranja iz starog polja u novo.

5.2. Lista

Nešto složeniji način ostvarivanja veze je lista. Stvaraju se povezani objekti koji unutar sebe čuvaju par identifikacijskih brojeva elemenata prikaza i strukture te pokazivač na sljedeći objekt. Prilikom stvaranja liste, elementi se redom ubacuju u listu. Svaki novi element se stavlja na kraj liste. Prilikom brisanja već postojećeg elementa pretraga ide od početka liste prema kraju, kao i kod polja no nije potrebno stvarati posve novu strukturu. Traženi element se izbacuje iz strukture i stvara se veza između dva okolna elementa.

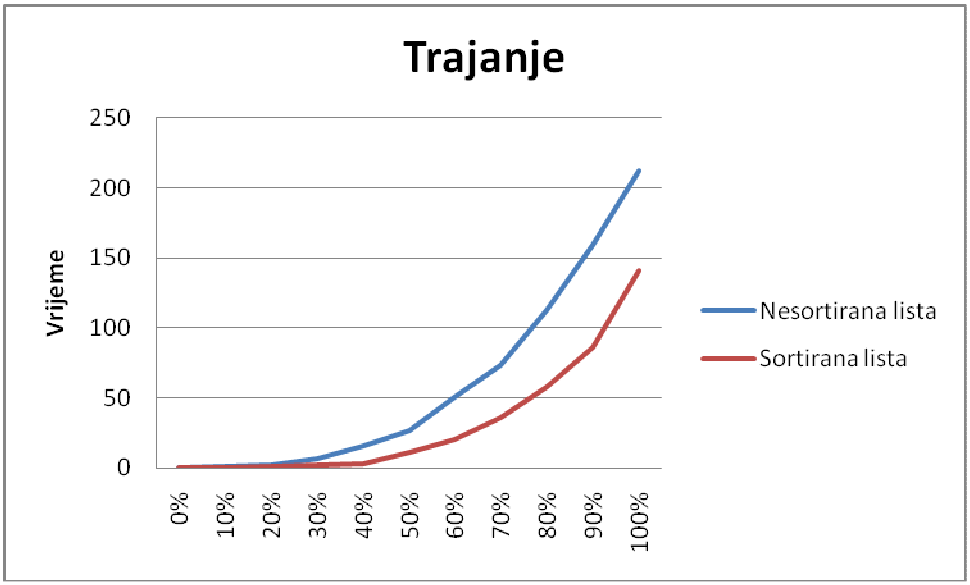


Slika 5.2 Graf ovisnosti broja mijenjanih elemenata o vremenu unutar liste

Prema grafu vidimo da trajanje brisanja starih i unošenja novih elemenata eksponencijalno raste sa brojem spomenutih elemenata. Razlog toga je što pretraga za svakim elementom kojeg želimo obrisati kreće od početka liste kao i za unošenje svakog pojedinog elementa. Kod polja smo nove elemente unosili na kraj, ali nismo morali proći kroz čitavo polje nego smo direktno adresirali željene adrese. Kod liste to nije moguće.

5.3. Sortirana lista

Kod sortirane liste imamo sličan slučaj kao kod nesortirane. Prilikom unošenja elemenata u listu moramo krenuti od početka liste i kretati se prema njenom repu, no u ovom slučaju ne moramo nužno ići do kraja liste. Nove elemente ne stavljamo na kraj liste kao kod nesortirane već ih stavljamo ispred prvog elementa veće vrijednosti. Na taj način štedimo nešto vremena prilikom unošenja elemenata. Kod brisanja elemenata nema razlike. Još uvijek moramo pretraživati listu. Činjenica da je lista sortirana nam ne pomaže mnogo jer nam elementi koje tražimo neće dolaziti redom. Može se dogoditi da proces brisanja u nekim situacijama teče brže kod nesortirane liste.

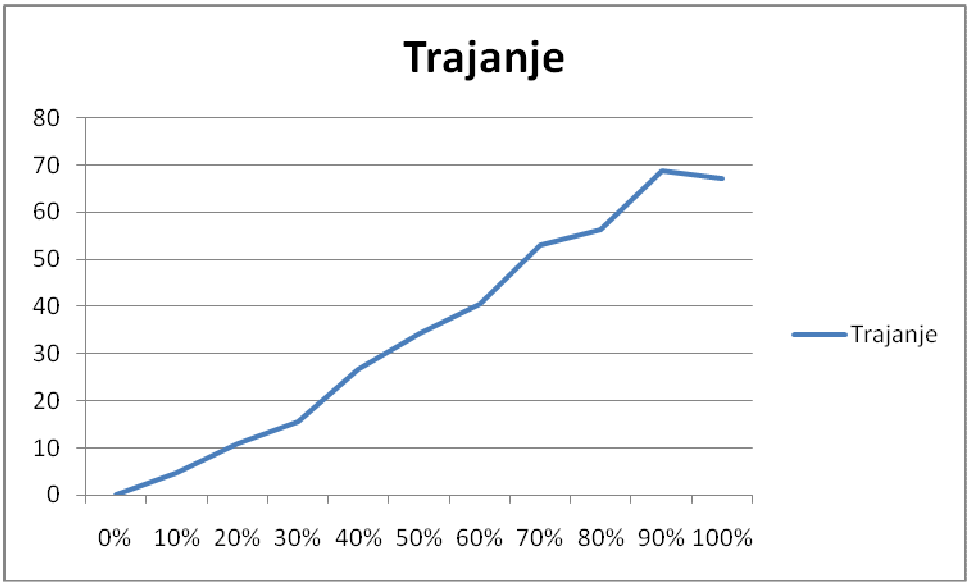


Slika 5.3 Graf usporedbe sortirane i nesortirane liste. Ovisnost vremena o broju elemenata u listama

Po grafu vidimo da je trajanje kod sortirane liste isto kao i kod nesortirane. Vrijeme i ovdje raste eksponencijalno sa porastom broja elemenata. Jedina razlika je da su rezultati kod sortirane liste nešto bolji od onih od nesortirane.

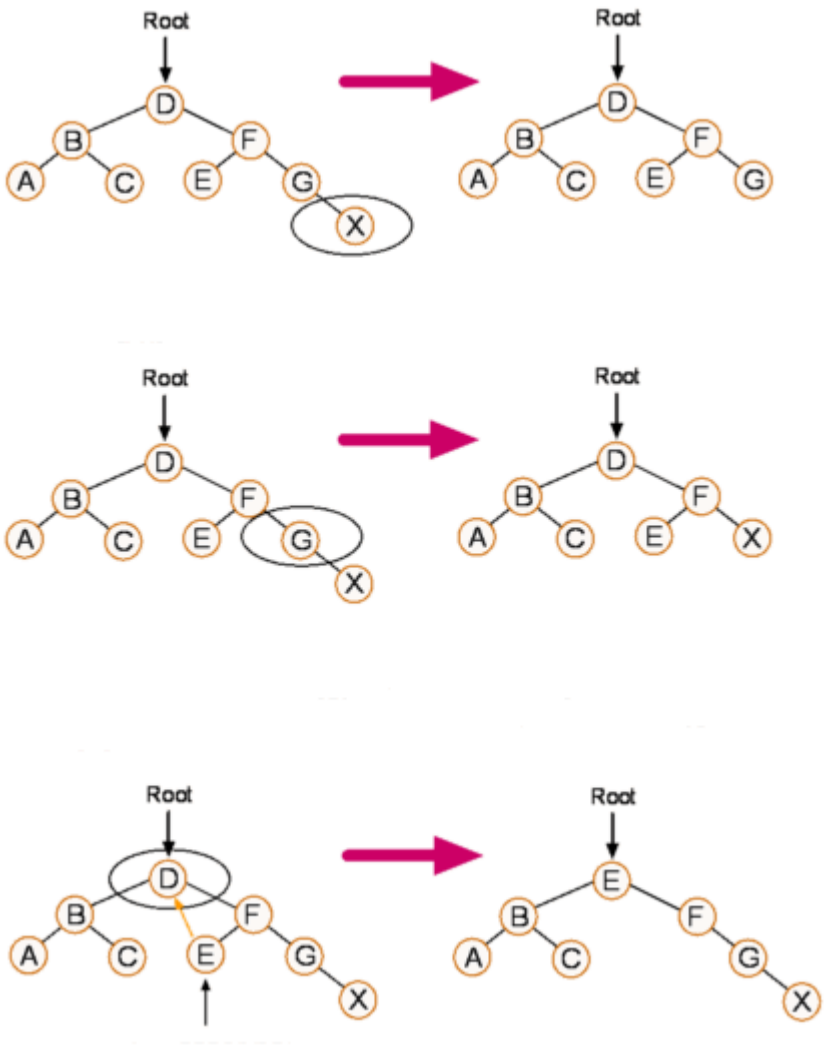
5.4. Binarno stablo

Nešto složeniji, ali ujedno i bolji način spremanja podataka jest unutar binarnog stabla. Elementi se prilikom unosa automatski sortiraju po veličini. Osnovno pravilo je da manji elementi od trenutnog idu u „lijevu“ granu dok veći ili jednaki elementi idu u „desnu“ granu. Najbolja varijanta popune stabla jest potpuna popuna stabla gdje se dobiva ista dubina stabla na svim granama. Najlošija je kad za unos dobijemo već sortirane elemente jer u tom slučaju konačno stablo izgleda poput liste. Dubina stabla je u tom slučaju jednaka broju elemenata u stablu.



Slika 5.4 Graf ovisnosti broja mijenjanih elemenata o vremenu u binarnom stablu

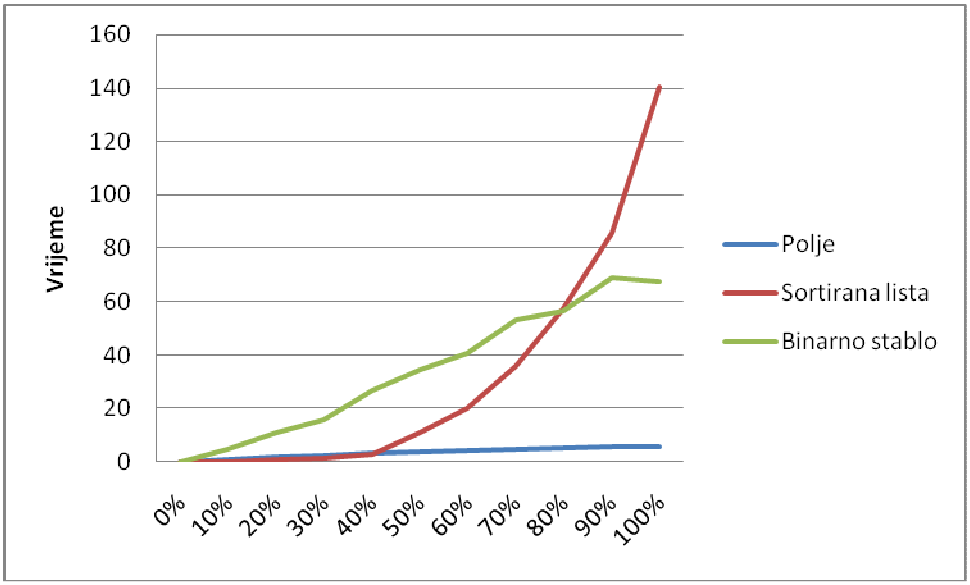
Unos i pretraživanje elemenata su optimalni koliko je i stablo optimalno razgranato, tj. koliko blizu potpunom stablu je korišteno stablo. Najlošiji slučaj je jednak dubini stabla. Pri brisanju elemenata iz stabla se koriste određeni algoritmi za ubrzanje rada. Najjednostavniji modeli brisanja su kad se brišu listovi ili čvorovi sa samo jednim djetetom. U slučaju brisanja čvora sa oba djeteta koristi se poseban postupak preraspodjele elemenata stabla:



Slika 5.5 Primjer različitih slučajeva brisanja elemenata iz binarnog stabla

Kako vidimo na slici brisanje lista je najjednostavniji slučaj. Obriše se objekt koji predstavlja list, a pokazivač u čvoru roditelju se postavlja na „null“ vrijednost. Slučaj brisanja čvora s jednim djetetom je nešto složeniji i svodi se na to da se nakon brisanja objekta pokazivač roditelja premjesti na dijete obrisanog čvora. Treći slučaj je dosta složeniji. Za početak moramo pronaći „čvor nasljednika“. On je zadnji lijevi čvor u desnoj grani.

5.5. Usporedba efikasnosti navedenih struktura podataka

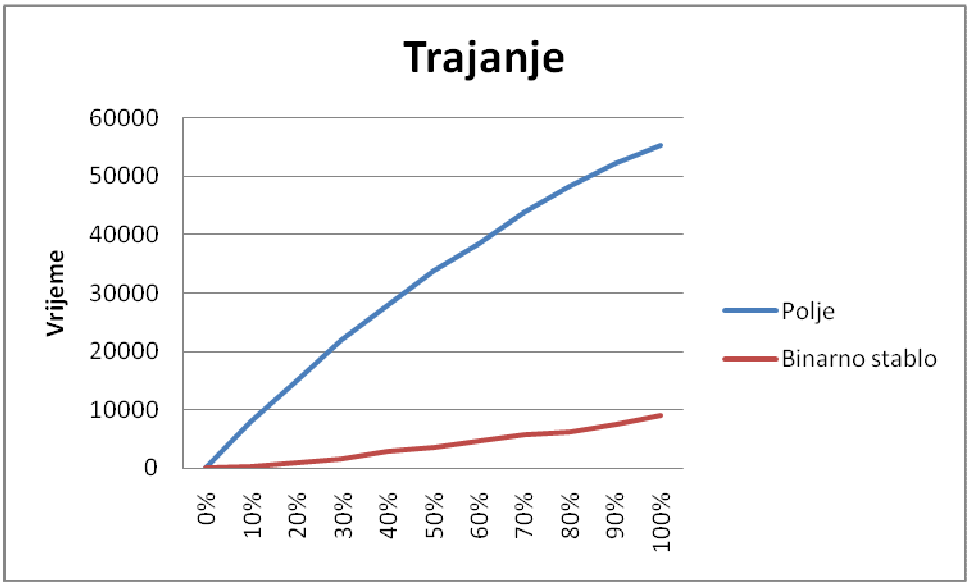


Slika 5.6 Usporedba efikasnosti polja, sortirane liste i binarnog stabla

Test usporedbe je rađen na računalu AMD Sempron(tm) 2800+. Svaki test je kretao od već izgrađene strukture elemenata. Test je ponavljan veći broj puta sa različitim inicijalnim veličinama struktura. Mjerenje pri istoj veličini je također ponavljano oko 1000 puta (ovisno o veličini strukture) radi preciznijih rezultata.

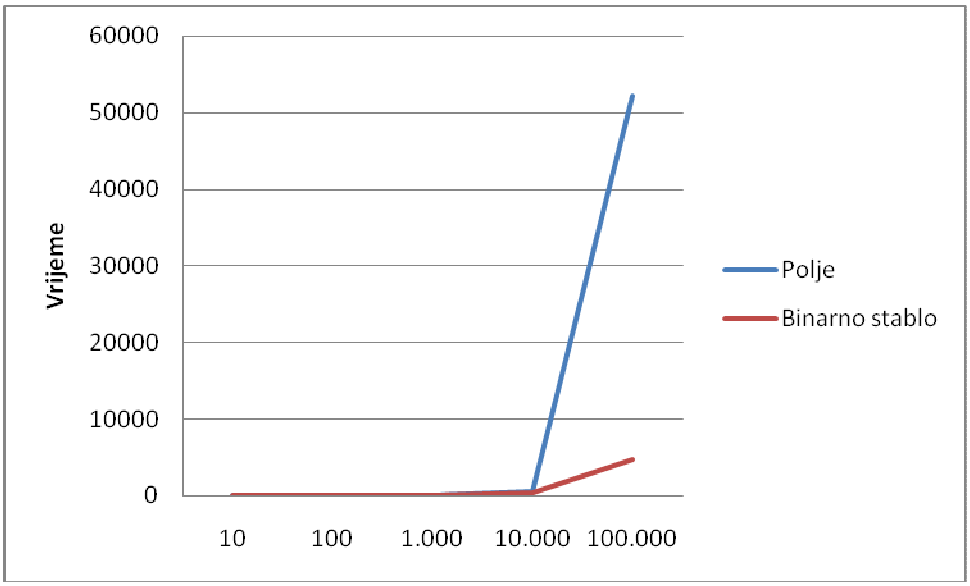
Test se sastojao od toga da se usporede brzine brisanja starih i spremanja novih elemenata u strukture podataka. Test je ponavljan za različit broj podataka koji se mijenjaju u jednom pokušaju. Izmjereno vrijeme služi samo za usporedbu efikasnosti mjerenih struktura.

Pri usporedbi efikasnosti struktura podataka sa 1000 spremljenih elemenata polje je dalo najbolje rezultate (slika 5.6). Sortirana lista je davala malo bolje rezultate pri mijenjanju do 40% elemenata odjednom, ali nakon toga joj efikasnost veoma brzo opada. Binarno stablo je dalo najlošije rezultate iako se može primijetiti da mu efikasnost opada daleko sporije od liste te da daje bolje rezultate pri više od 80% mijenjanih elemenata. Ako uzmemo u obzir da će se rijetko mijenjati više od 20% elemenata odjednom možemo zaključiti da će polje kao najjednostavnija struktura davati posve zadovoljavajuće rezultate.



Slika 5.7 Usporedba polja i binarnog stabla pri većem broju elemenata

Pri veličini strukture od 100.000 elemenata se situacija mijenja. Možemo primijetiti da u ovom slučaju binarno stablo od početka daje daleko bolje rezultate. Polje više nije dominantna struktura.



Slika 5.8 Usporedba efikasnosti polja i binarnog stabla u odnosu na broj elemenata u strukturi

Polje je struktura koju možemo koristiti pri manjem broju elemenata. Tada se njegove mane ne očituju u tolikoj mjeri da bi postale značajne. Nakon određene granice polje postaje preneefikasno za rad. Binarno stablo je dobra i efikasna struktura za upravljanje podacima, brzo pronalaženje potrebnih elemenata i njihovu manipulaciju. Prednosti binarnog stabla se ne očituju pri malom broju elemenata, ali postaju značajne pri većem.

6. Zaključak

Cilj zadatka je bio ostvariti prikaz i mogućnost interakcije sa oktalnim stablom. Oktalno stablo je dobar način rada sa kolizijama, ali nekada inicijalna podjela stabla nije zadovoljavajuća i traži naknadne dorade. Da ne bismo uništavali i gradili cijelo stablo ispočetka potrebni su nam postupci za izoliranje pojedinih dijelova stabla za dodatno grananje.

Prikaz stabla je relativno jednostavno za izvesti no zbog njegove veličine je potrebno provesti dodatne optimizacije da bi konačni produkt bio upotrebljiv na današnjim računalima i u stvarnom vremenu. HOOPS 3D alat nam uveliko pomaže u tome, ali nas i sputava svojim mogućnostima. Potrebno je dosta manipuliranja da se dobije željeni rezultat.

7. Sažetak / Abstract

Interaktivni postupci nad oktalnom strukturom grafičkih podataka

Ukratko su opisane osnove oktalne strukture podataka i grafičkog alata HOOPS 3D. Opisan je prikaz osnovnih elemenata kao i njihovo manipuliranje grafičkim alatom. Za programiranje je korišten jezik C++ pomoću razvojnog okruženja Microsoft Visual Studio 2003 sa korištenjem GTK+ za izradu sučelja. Grafički prikaz kao i njegova manipulacija su ostvareni pomoću grafičkog alata HOOPS 3D. Dodatno su prikazane i ocijenjene osnovne strukture za spremanje i rad podataka.

Ključne riječi: oktalno stablo, HOOPS 3D, prikaz, strukture podataka, računalna grafika, računalna animacija

Interactive methods over the graphic data structure Octree

This paper describes basics of the Octree structure and the HOOPS 3D graphic tool. It describes the viewing of basics objects and their manipulation with the graphic tools. Programming language C++ is used for raw programming using the Microsoft Visual Studio 2003 with the use of GTK+ for the creation of a user interface. Graphical viewing and its manipulation has been accomplished with the HOOPS 3D graphic tool. In addition, basic structures for data use and storage have been compared and evaluated.

Key words: octree, HOOPS 3D, view, data structures, computer graphics, computer animation

Popis slika

8. Literatura

1. Octree – Wikipedia, s Interneta, http://en.wikipedia.org/wiki/Octree, srpanj 2008
2. Constructive Solid Geometry Methods, s Interneta, http://groups.csail.mit.edu/graphics/classes/6.837/F98/talecture/, rujan 2008
3. Kurtanjek, Robert: Strukture podataka za detekciju sudara, s Interneta, http://www.zemris.fer.hr/predmeti/rg/diplomski/08Kurtanjek/index.html, rujan 2008
4. HOOPS Documentation, The HOOPS 3D Product Suite v15.0
5. Ewington, Shean: CodeProject: A Simple Binary Search Tree written in C#, s Interneta, http://www.codeproject.com/KB/recipes/BinarySearchTree.aspx, rujan 2008