SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1280

PROGRESIVNE MREŽE

Tin Englman

Zagreb, lipanj 2010.

Zahvaljujem Andrei Nikolić na pomoći stilskoj i pravopisnoj obradi, kolegi Ivanu Kravarščanu na pomoći oko spremnika, Danijelu Vickoviću za izradu modela koji je korišten u pokusima, te mentorici prof. dr. sc. Željki Mihajlović na pomoći i strpljenju.

Sadržaj

1. Uvod

Kako bi se zadovoljila sve veća i veća očekivanja u računalnoj grafici, potrebni su sve detaljniji modeli. Prilikom uobičajenog postupka modeliranja, detaljni modeli rade se uz pomoć mnogih operacija, kao što su izvlačenje (eng. *extrude*), konstruktivna geometrija čvrstih tijela ili proizvoljne deformacije. Kako bi se ti modeli mogli prikazati, prvo se moraju pretvoriti u prihvatljiv oblik – mrežu trokuta. Detaljne mreže poligona također se mogu dobiti skeniranjem stvarnih fizičkih objekata. No, bez obzira na koji način se rade modeli, rezultirajuća mreža poligona problematična je i vremenski zahtjevna za spremanje i iscrtavanje, te uzrokuje mnoge probleme. Cilj ovog rada je nizom pokusa isprobati razne varijante algoritma dinamičke promjene razine detalja ostvarene progresivnim mrežama kako bi se ustanovila koje su u praksi primjenjive i za koju situaciju.

1.1. Pojednostavljenje mreže poligona

Izmodelirane ili skeniranjem dobivene mreže poligona rijetko su optimizirane za učinkovito iscrtavanj i vrlo često se mogu zamijeniti sa gotovo identičnom aproksimacijom s puno manje poligona. Do danas su se razvila mnoga rješenja kao samostojeći programi, ali i kao *plug-inovi* za aplikacije za modeliranje (popularniji programi već imaju u sebi ugrađenu nekakvu podršku za optimiziranje mreže poligona).

1.2. Razina detalja (LOD)

Kako bi još više ubrzali iscrtavanje, sve više i više se koristi pristup izrade više verzija modela s različitom razinom detalja. Detaljan model se koristi kada je objekt blizu kameri (gledatelju, tj. korisniku), dok se za udaljene poglede koriste grublje aproksimacije. Gotovo trenutne zamjene između modela različitih razina detalja korisniku se mogu činiti neprirodne (zbog nagle promjene objekta kojeg gleda). Zbog toga je dobra praksa da se

prilikom izrade više modela na različitim razinama detalja više važnosti pridoda glatkim vizualnim prijelazima.

1.3. Progresivni prijenos

Slanju modela kroz mrežu može se pristupiti na dva načina. Prvi je da se pošalje cijeli model, tako da primatelj može vidjeti što je dobio tek kada primi kompletni model. Drugi način je sukcesivno slanje različitih LOD modela. Na taj način primatelj bi dobivao sve bolju i bolju aproksimaciju modela kako prima podatke, no prijenos bi duže trajao.

1.4. Kompresija mreže poligona

Problem spremanja modela može se sagledati na dva načina. Jedan je da se primijeni pojednostavljenje mreže poligona čime smanjujemo broj poligona i time dobivamo manju datoteku te naposlijetku manje mjesta zauzimamo na tvrdom disku (ili nekom drugom mediju). Drugi način je komprimiranje mreže poligona, odnosno minimiziranje internog zapisa pomoću kojeg definiramo model. Format zapisa može biti ASCII ili binarni te se manipuliranjem tih zapisa može postići veliko ubrzanje sustava ili se može smanjiti veličina datoteke u kojoj je spremljen model.

1.5. Selektivno poboljšanje

Svaka mreža poligona prikazana kao LOD aproksimacija prikazuje cijeli model na jednolikoj razini detalja. Ponekad je poželjno prilagoditi razinu detalja ovisno o regijama modela. Proučimo ovu teoriju na primjeru. Uzmimo da, primjerice, korisnik iz zraka gleda krajolik. Krajolik prikazan pomoću mreže poligona potrebno je detaljno prikazati samo na dijelovima koje korisnik vidi iz blizine i koji su mu u vidnom polju. Na ostale dijelove krajolika može se primijeniti grablja aproksimacija.

Koristeći progresivne mreže, proizvoljna mreža poligona $\hat{M}$ može se pohraniti kao mnogo jednostavnija mreža M^0. Uz pojednostavljenu mrežu sprema se i niz od n zapisa pomoću kojih se inkrementalno može iz pojednostavljene mreže M^0 ponovno dobiti originalni oblik - $\hat{M}$. Svaki od tih zapisa sadrži podatak o *razdvajanju točke* (*eng. vertex*

split). Razdvajanje točke je jednostavna transformacija kojom se postojećoj mreži dodaje jedna točka (*eng. vertex)*

Dakle, mreža poligona pomoću progresivnih mreža (eng. *PM*) definira kontinuirani niz mreža *Mo*, *M1*, *M2*,...,*Mn* čija preciznost, a samim time i razina detalja, kontinuirano raste, te se kao krajnji rezultat može dobiti mreža bilo koje razine detalja. Ukratko, progresivne mreže pružaju učinkovit i kontinuirani prikaz modela bez gubitaka.

2. Mreže poligona u računalnoj grafici

Modeli, tj. objekti, u računalnoj su grafici prikazani pomoću mreže poligona, točnije mreže trokuta (*eng. triangle mesh*). Geometrijski gledano, mreža trokuta sastoji se od lineranih trokutastih poligona koji su međusobno spojeni svojim stranicama. Oblik, tj. geometrija mreže, može se označiti parom (*K,V*), pri čemu *K* označava udaljenost točaka, bridova i ploha (poligona), a *V={v1, v2, ... ,vn}* je skup točaka koji definiraju oblik mreže u **R**3 prostoru.

Uz samu definiciju mreže, često se modelu pridružuju i parametri prikaza površine modela. Ti parametri mogu se podijeliti u dvije grupe: *diskretne* i *skalarne* parametre.

Diskretne parametre uobičajeno povezujemo s poligonima mreže. Npr., često korišteni diskretni parametar je identifikator materijala (eng. *material Identifier*) koji se koristi pri sjenčanju prilikom iscrtavanja mreže.

Skalarni parametri su, za razliku od diskretnih, povezani sa samom mrežom, te u njih spadaju: difuzno osvjetljenje (r,g,b), normale (n_x, n_y, n_z) i koordinate teksture (u, v). Općenito, ovi parametri određuju lokalna obilježja sjenčanja poligona i obično su definirani za točke, odnosno vrhove mreže. Međutim, kako bi prikazali razliku u polju skalara, kao i zbog mogućnosti da susjedni poligoni imaju različito definirano sjenčanje, bolje je skalarne parametre povezati s kutevima mreže umjesto s točkama. Kut definiramo kao par *(točka, poligon)* (*eng. vertex, face; v,f*). Dakle, skalarni parametri u kutu (v, f) definiraju sjenčanje poligona *f* u točci *v*. U jednoj točci, recimo, može biti definirano više normala (ovisno o broju poligona) kako bi se postigla nagla promjena boje (sjenčanja).

Mrežu možemo prikazati kao kao četvorku *M=(K, V, D, S)* gdje pomoću *V* definiramo oblik, $V=\{v_1, v_2, \ldots, v_n\}$. Dakle, *V* je skup svih točaka i njihovih pozicija pomoću kojih definiramo objekt. *K* označava udaljenost između pojedinih točaka, bridova i poligona, dok parametri *D* i *S* označavaju diskretne, odnosno skalarne parametre. Skup *D* je skup diskretnih parametara d_f koji se upotrebljavaju nad poligonima $f=\{j, k, l\} \in K$, dok je *S* skup skalarnih atributa $s_{(v,f)}$ koje koristimo nad kutevima *(v, f)*.

Parametri *D* i *S* pridonose diskontinuitetima u smislu vizualnog prikaza mreže. Npr. brid $\{v_j, v_k\}$ je oštar ako vrijedi:

- Da je brid rubni
- Da njihovi susjedni bridovi f_l i f_r imaju različite diskretne parameter (npr. $d_{fl} \neq d_{fr}$)
- Da njihovi susjedni vrhovi imaju različite skalarne parameter (npr. $s_{(vj,fl)} \neq s_{(vj,fr)}$)

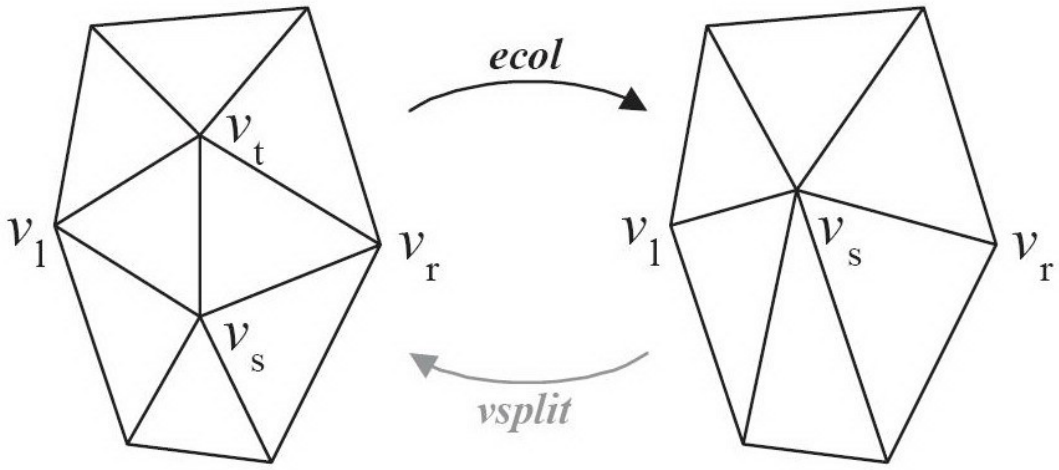
Skup oštrih bridova definira skup diskontinuitetnih krivulja.

3. Progresivne mreže

U ovom poglavlju biti će opisano nekoliko optimizacijaskih metoda pomoću kojih se može napraviti aproksimacija originalne mreže sa mnogo jednostavnijom mrežom.

3.1. Općenito o progresivnim mrežama

Postoje algoritmi koji koriste tri moguće transformacije za optimizaciju mreža: kolaps bridova (*eng. edge collapse*), razdvajanje bridova (*eng. edge split*), te zamjena bridova(*eng. edge swap*). No, korištenjem samo jedne od navedenih transformacija također se može postići učinkovita optimizacija, npr. korištenjem *kolapsa bridova*. Ovom metodom spajamo dva vrha jednog brida u jedan vrh kao što se može vidjeti na slici 1.

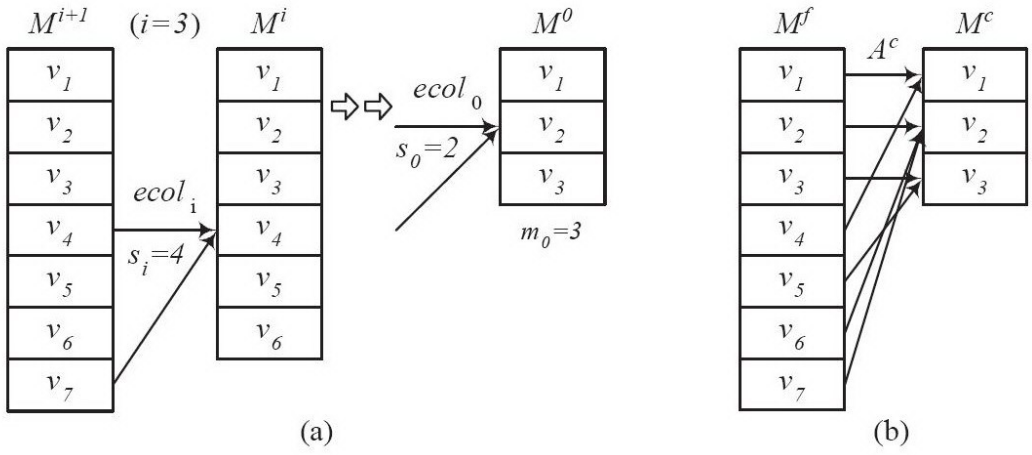


Slika 1. Transformacija *edge collaps (ecol)*

Dakle, transformacija ecol($\{v_s, v_t\}$) spaja dva susjedna vrha v_s i v_t u jedan vrh – v_s. Nakon transformacije vrh vt i poligoni $\{v_s, v_t, v_l\}$ i {vs, vt, vr} nestaju. Dakle, sukcesivnim korištenjem *ecol* transformacije možemo originalnu mrežu $\hat{M} = M^n$ pretvoriti u jednostavniju – M^0:

$$\left(\hat{M} = M^n\right) - \xrightarrow{ecol_{n-1}} \ldots - \xrightarrow{ecol_1} M^1 - \xrightarrow{ecol_0} M^0$$

Redosljed vrhova nad kojima se vrši *ecol* transformacija mora biti pomno odabran, jer upravo tako određujemo kvalitetu aproksimativne mreže M^i, $i<n$. Neka je m_0 broj točaka u mreži M^0, te označimo vrhove mreže M^i sa $V^i = \{v_1, ..., v_{m0+i}\}$. Uz tako definirane parametre, na slici 2 je prikazana operacija $ecol_i$ između dva brida $\{v_{si}, v_{m0+i+1}\}$. S obzirom da vrhovi mogu imati različite pozicija u različitim mrežama, vrh v_j u mreži M^i se označava v_j^i.



Slika 2. a) Primjer operacije *ecol* b) Promjene vrhova pri operaciji *ecol*

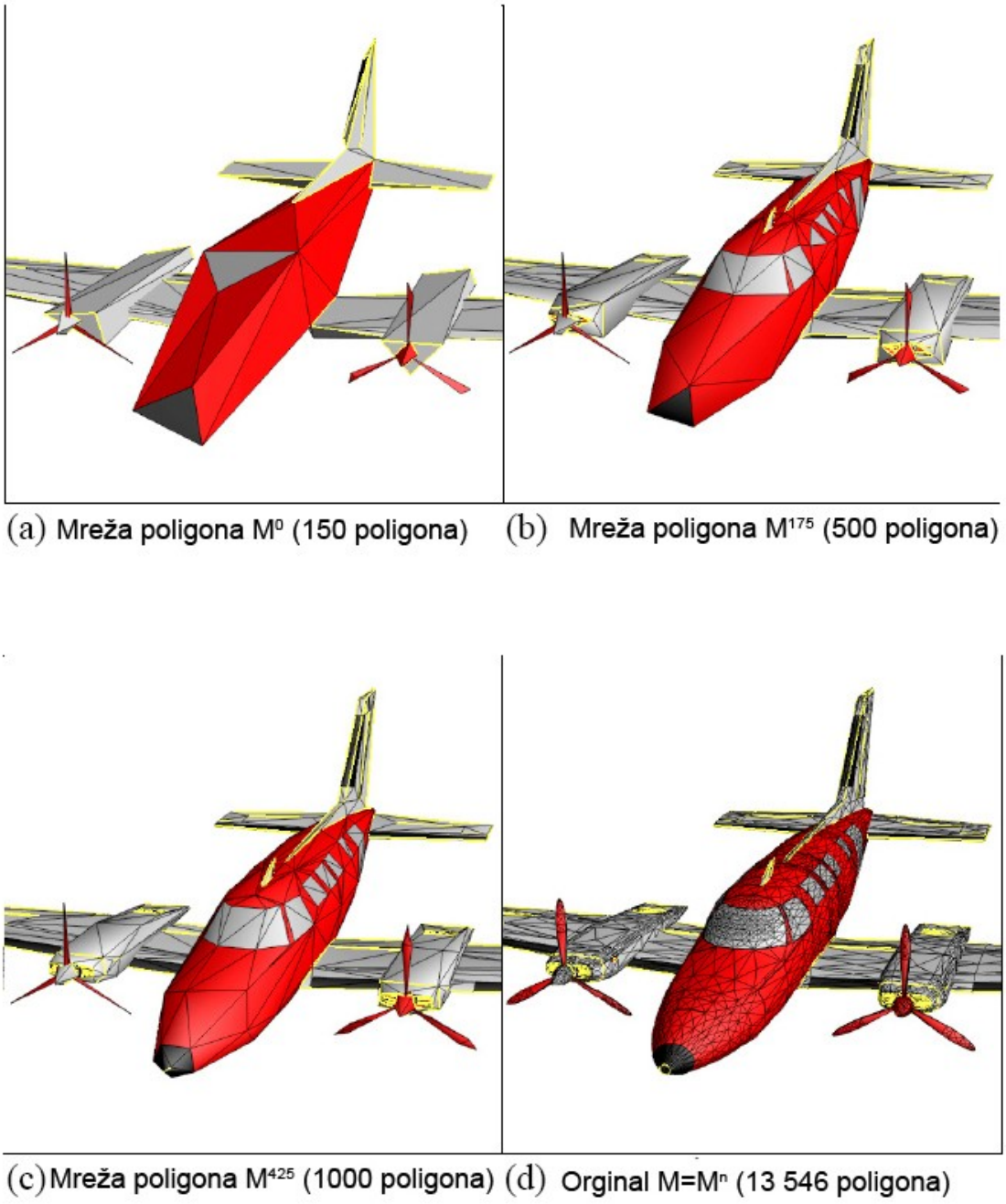
Bitna karakteristika operacije *kolapsa bridova* je da je ta operacija potpuno reverzibilna. Na slici (slika 1.) se može vidjeti inverzna transformacija, razdvajanje vrhova (eng. *vertex split*) – *vsplit*. Operacija *vsplit*(*s, l, r, t A*) dodaje novi vrh v_t blizu vrha v_s, te dva nova poligona: (v_s, v_t, v_l) i (v_t, v_s, v_r). Ako je brid $\{v_s, v_t\}$ granični brid, tada se stavlja $v_r = 0$, te se u mrežu dodaje samo jedan poligon. Transformacija također ponovno računa parametre u okruženju transformacije. Podatci parametara, označeni sa *A*, uključuju pozicije vrhova v_s i v_t (na koje se utječe transformacijom), diskretne parametre $d_{\{vs, vt, vl\}}$ i $d_{\{vt, vs, vr\}}$ novih poligona, te skalarne parametre bridova na koje se utjecalo ($s_{(v1, \{vs, vt, vl\})}$ i $s_{(vr, \{vt, vs, vr\})}$).

Pošto su operacije kolapsa bridova reverzibilne, mreža poligona $\hat{M}$ se može dobiti iz jednostavne mreže M^0 zajedno sa nizom od n *vsplit* operacija:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} \left(M^n = \hat{M}\right)$$

pri čemu je svaki zapis oblika $vsplit(s_i, l_i, r_i, A_i)$. Progresivna mreža mreže poligona *M* se zapisuje kao (M^0, $\{vsplit_0, ..., vsplit_{n-1}\}$). Na slici (3) prikazan je primjer progresivne mreže

u kojoj je originalna mreža poligona $\hat{M}$ (sa 13 546 pligona) pojednostavljena na grubu aproksimaciju (sa 150 poligona) koristeći 6698 *ecol* transformacija.



Slika 3. Prikaz progresivne mreže

4. Implementacija

4.1. Programska potpora

Progresivne mreže implementirane su u okviru grafičkog pogona napisanog u programskom jeziku c++ koristeći DirectX za pristup grafičkom sklopovlju. Pogon je podijeljen na pet osnovnih komponenti: dohvaćanje vremena, obrada kamera, korisnički unos, obrada fizike te iscrtavanje grafike. Srž pogona čini objekt tipa GameManager koji preko referenci pokreće sve ostale dijelove pogona. Glavni program inicijalizira sve komponente te GameManageru proslijedi reference nakon čega u beskonačnoj petlji poziva metodu GameManager::doFrame(). Metoda doFrame() koristi sve komponente kako bi u konačnici iscrtala jednu sliku na ekran. Ovakva arhitektura pruža maksimalnu fleksibilnost jer komponente ne ovise eksplicitno jedna o drugoj.

Objekt tipa GameManager, osim što pokreće sve ostale komponente, zadužen je za kreaciju objekata. Svi objekti u programu izvedeni su iz apstraktne klase Object. Klasa Object sadrži osnovne podatke o nekom objektu u virtualnom svijetu, kao što su pozicija i kut po svim osima. Također sadrži pokazivač na sučelje GraphicObject koje implementiraju klase XFile, ProgressiveMesh i PureProgressiveMesh. Objekti koji se koriste u svim simulacijama su tipa PhysicalObject. PhysicalObject proširuje razred Object dodatnim svojstvima kao što su brzina, sila na objekt ili masa. GameManager može kreirati objekte na dva načina ovisno o potrebama simulacije; prvi način je učitavanje objekata iz datoteke koristeći metodu GameManager::loadObjectsFromFile(string filename) na način da se u datoteci prvo zada tip objekta i potom redom svi parametri koji taj objekt opisuju. Ovaj načine je nezgodan za simulacije koje zahtijevaju veliki broj objekata, stoga je napravljen generator slučajnih objekata kojeg ostvaruje metoda GameManager::createTestScenatio(int numObj, Test type, string graphicFile). Ona generira numObj objekata slučajno razmještenih u prostoru. Type je enum koji može biti nonProgressive, progressive i pureProgressive i označava na koji će se način učitati model iz datoteke graphicFile. U oba slučaja model se pri učitavanju pohranjuje u

GraphicContainer. GraphicContainer je klasa izvedena iz predloška Container. Interno sadrži mapu pokazivača na GraphicObject organiziranu po stringovima koji predstavljaju datoteku iz koje je objekt učitan. U slučaju da GraphicContainer primi objekt koji je već ranije učitan, ne pohranjuje ga ponovo već vraća pokazivač na ranije učitani objekt. Ovim mehanizmom ostvareno je da objekti koji su predstavljeni istim modelom taj model samo "posuđuju" prilikom crtanja, te time ne stvaraju nepotrebne duplikate.

Protok vremena simulira klasa Timer. Timer mjeri proteklo vrijeme između dva poziva i vraća ga GameManageru, koji taj podatak proslijedi ostalim komponentama koje ovise o protoku vremena, kao što je PhysicsEngine. Timer se poziva na početku svake iteracije doFrame() funkcije.

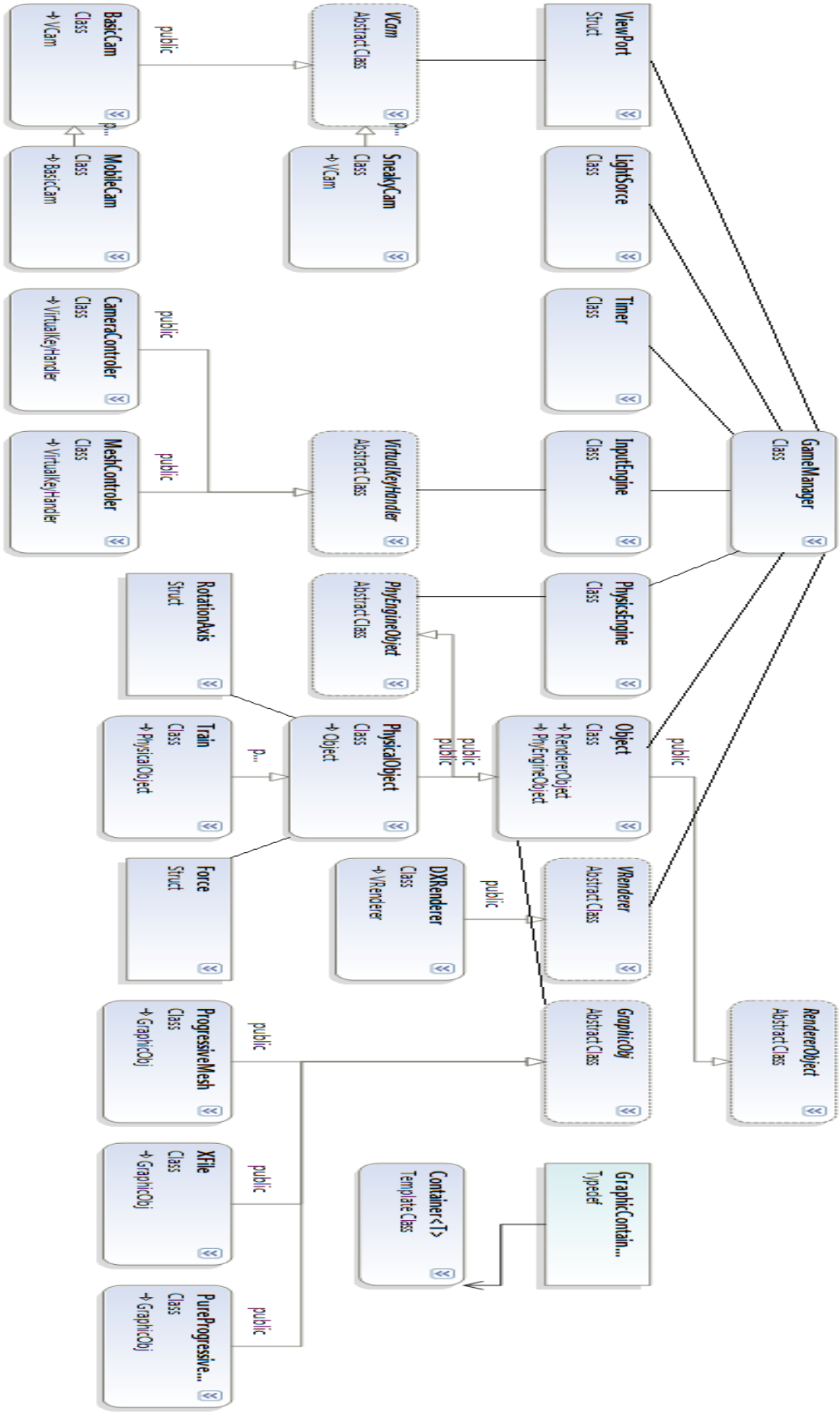
Obrada kamere se vrši preko apstraktne klase VirtualCamera. VirtualCamera sadrži trodimenzionalni vektor očišta, gledišta te vektor koji označava gdje je kameri "gore". Također sadrži metodu VirtualCamera::handleCam(double dt) koja vrši specifično ponašanje kamera koje su izvedene iz VirtualCamera. VirtualCamera implementiraju BasicCamera i MobileCamera. BasicCamera je najjednostavniji oblik kamere, ima fiksno očište i gledište i predstavlja kameru koja konstantno prikazuje neki prostor. MobileCamera nasljeđuje BasicCamera te joj dodaje funkcionalnost prečenja objekta. Za potrebu nekih simulacija kameri BasicCamera je privremeno dodana mogućnost gibanja po pravcu.

Korisnički unos ostvaren je klasom InputEngine koja enkapsulira funkcionalnost DirectInputa. InputEngine sadrži podatkovne strukture nužne za rad DirectInputa te polje pokazivača na sučelje KeyHandler. InputEngine dohvaća stanje tipkovnice koje pohranjuje u strukturu, te ga preko sučelja KeyHandler proslijedi objektima zaduženih za obradu unosa. U trenutnoj verziji programa dva su objekta za obradu unosa: jedan tipa CameraControler te jedan tipa MeshControler. CameraControler je, kao što mu ime govori, zadužen za pomicanje kamere, konkretno 'w' i 's' pomiću kameru po X osi, 'a' i 'd' po Y, dok 'q' i 'w' kameru pomiću po Z osi. MeshControler je zadužen za podešavanje prikaza progresivne mreže: 'y' smanjuje broj poligona dok ga 'x' povećava, 'u' prikazuje poligon u obliku mreže a 'i' ga vraća u prvotno stanje.

Obradu fizike ostvaruje objekt tipa PhysicsEngine. PhysicEngine objektima pristupa preko sučelja PhyEngineObject. PhyEngineObject daje PhysicsEngineu pristup svim atributima vezanim za fiziku, kao što su sila, brzina, položaj, dok zabranjuje pristup

atributima koji nisu u domeni odgovornosti fizike, npr. pokazivaču na grafiku. PhysicsEngine poziva apstraktnu metodu PhyEngineObject::move(double dt) koju implementiraju konkretni objekti; u slučaju ove verzije to je samo PhysicalObject koji u toj metodi ostvaruje translaciju objekta na temelju sila koje djeluju na objekt, te eksplicitno zadanih akceleracija i brzina.

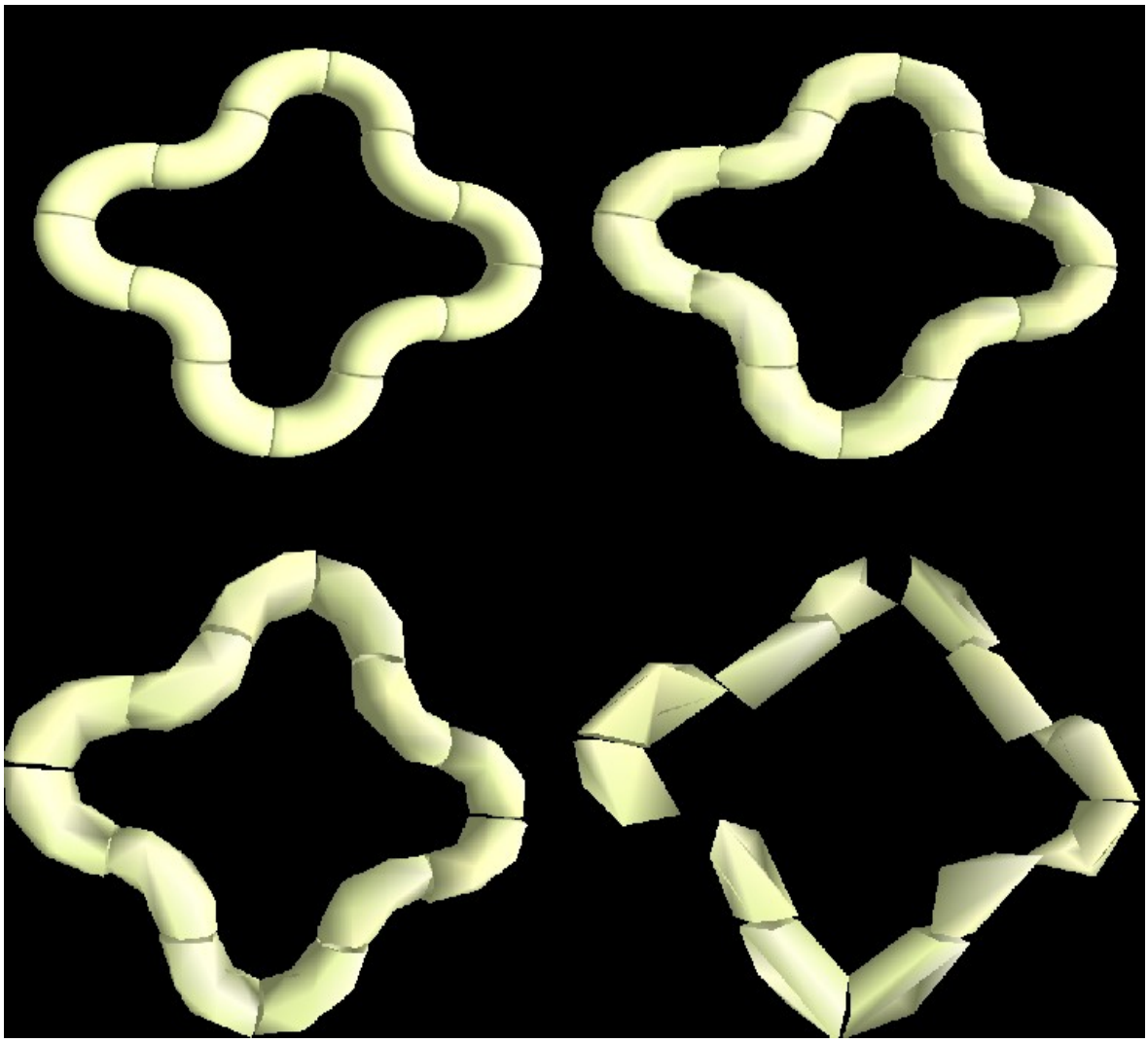
Iscrtavanje objekata se vrši preko sučelja VirtualRenderer. VirtualRenderer implementira DXRenderer. Kao što InputEngine enkapsulira DirectInput, tako i DXRenderer skriva detalje rada Direct3D-a od višeg konteksta. DXRenderer prima objekt tipa ViewPort koji sadrži strukturu tip D3DVIEWPORT te pokazivač na kameru. Na temelju D3DVIEWPORT strukture DXRenderer zaključuje na koji dio ekrana treba iscrtavati. Kamera se koristi pri izračunavanju projekcijske transformacije i pri transformaciji pogleda. DXRenderer objektima pristupa preko sučelja RenderObject koji, slično kao kod PhyEngineObject-a, dozvoljava pristup samo djelovima koji se odnose na iscrtavanje. Na temelju položaja i orijentacije objekta izračunaju se matrice translacije i rotacije te se dohvaća pokazivač na model preko sučelja GraphicObject i iscrtava model. Sučelje služi kako bi se transparentno koristili različiti tipovi prikaza modela, čime je omogućeno brzo i jednostavno testiranje različitih metoda prikaza.



Slika 4. Klasni diagram pogona

4.2. Testovi

Svi testovi vršeni su u istom scenariju tako da jedini parametar koji utječe na brzinu izvođenja je algoritam iscrtavanja koji se koristi. Test se sastoji od kamere koja se 20 sekundi kreće u prostoru te prikazuje 60 identičnih objekata. Svaki objekt ima oko 3000 poligona, stoga se u jednom trenutku obrađuje oko 180 000 poligona. Efikasnost grafičkog prikaza općenito se mjeri brojem crteža u sekundi(skračeno FPS, eng. *frames per second*) pa se to mjerilo koristi i ovdje. Krajnji cilj je pronaći algoritam koji će postići što veći FPS uz prihvatljive gubitke kvalitete i povečanja memoriske zahtjevnosti. Mjerenje FPS-a vršeno je besplatnom verziom aplikacije FRAPS. Svaki test se ponovi tri puta.



Slika 5. Pokusni model na raznim razinama detalja

4.2.1. Bez optimizacije

Ovaj test je nužan kao referentna točka za uvid u poboljšanje koje će ostvariti ostali algoritmi. Ovaj algoritam jednostavno učita modele iz datoteke te ih, bez optimizacije, iscrta. Implementacija se nalazi u klasi XFile. Izvršna datoteka testa nalazi se u mapi "IzvrsneDatoteke\BezMreze".

Rezultati:

Algoritam je u prosjeku iscrtao **311** crteža pri prosječnoj brzini od **15,6** FPS-a pri čemu FPS nikad nije pao ispod 15.

4.2.2. Osnovni algoritam prograsivnih mreža.

Klasa PureProgressiveMesh implementira ideju dinamičke promjene detaljnosti modela u ovisnosti o udaljenosti očišta od središta modela. Algoritam prvo učita model iz datoteke te iz njega napravi osnovnu mrežu. Osnovna mreža se zatim propušta kroz funkciju D3DXCleanMesh koja mrežu priprema za pojednostavljivanje. Očišćenu mrežu zatim obrađuje funkcija D3DXWeldVertices koja uklanja vrlo slične vrhove. Sličnost vrhova se određuje temeljem epsilon vrijednosti koja je u ovom slučaju 10^{-6}. Progresivna mreža se generira funkcijom D3DXGeneratePMesh tako da joj se proslijedi prethodno obrađena mreža. Progresivna mreža se pohranjuje u strukturu ID3DXPMesh, nakon čega je spremna za korištenje. Iscrtavanje objekta vrši se metodom PureProgressiveMesh::draw koja od DXRenderera prima pokazivač na kameru i poziciju objekta na koji se model odnosi. Na temelju udaljenosti kamere i pozicije objekta izračuna se broj vrhova koji model treba imati, te se mreža postavi na tu vrijednost funkcijom ID3DXPMesh::SetNumVertices. Broj vrhova računa se po formuli: brojVrhova = ((maxV – minV)*(k/(k+d)) + minV, gdje su "minV" i "maxV" minimalni i maksimalni broj vrhova, "d" udaljenost kamere do središta objekta, a "k" konstanta. Što je "k" veći to će brzina izvođenja biti manja, a detaljnost modela veća. U testnim primjerima "k" je postavljen na 500. Iz formule se vidi da kada d teži u beskonačno, broj vrhova teži u minV, a kad d teži u

nulu, broj vrhova teži u maxV, čime se osigurava da će broj vrhova uvijek biti između minV i maxV. Nakon što se postavi broj vrhova, model se iscrtava na isti način kao i neoptimizirani objekt. Očekuje se porast performansi u odnosu na neoptimizirani model zato što će se ukupni broj poligona u sceni značajno smanjiti. Izvršna datoteka se nalazi u mapi "IzvrsneDatoteke\OsnovniModel".

Rezultati:

Na moje veliko iznenađenje ovaj algoritam je značajno sporiji i nestabilniji od varijante bez optimizacije. Iscrtao je samo **92** slike, uz prosječnu brzinu od **4,6** FPS-a, pri čemu je FPS u nekim trenutcima pao čak na 0. Kad bi se zanemario iznenadni pad FPS koji se pojavljuje naizgled slučajno, FPS bi bio oko **7,** što je i dalje malo. Pretpostavljam da razlog leži u tome što se progresivna mreža ponaša kao dvostruko vezana lista. Da bi došlo od člana n do člana m treba prvo proći sve članove između n i m. U simulaciji su položaji svih objekata generirani slučajno i dijele isti model, tako da kad neki objekt zatraži svoje iscrtavanje, isti mora podesiti detaljnost zajedničkog modela da odgovara vlastitoj udaljenosti od kamere. Posljedica toga je da zajednički model "skače" od vrlo visoke do vrlo niske detaljnosti, što je ekvivalentno slučajnom pristupu dvostruko vezanoj listi.

4.2.3. Najgori slučaj

Da bi se testirala teorija predstavljena u prethodnom odlomku napravljen je scenarij u kojem jedna polovica objekata zahtjeva maksimalnu detaljnost, dok druga zahtijeva minimalnu detaljnost neovisno o udaljenosti kamere. Objekti se iscrtavaju takvim redoslijedom da se naizmjenice iscrtavaju najdetaljniji i najjednostavniji model. U ovom scenariju prosječan broj poligona u sceni je otprilike isti kao u prethodnom scenariju, stoga je svaka razlika u performansama posljedica dinamičke promjene detaljnosti. Ako je prepostavka točna, pad perfomansi bi trebao biti značajan. Izvršna datoteka se nalazi u mapi "IzvrsneDatoteke\NajgoriSlucaj".

Rezultati:

Kao što je očekivano, performanse su u ovom slučaju pale na **2,6** FPS-a, odnosno **51** iscrtanu sliku. Na temelju ovih rezultata nameće se zaključak da optimizacija treba ići u smjeru minimizacije dinamičke promjene detaljnosti modela.

4.2.4. Srotirani objekti

Jednostavan način na koji se može značajno smanjiti potrebu za opsežnim dinamičkim promjenama je postupak kojim se svi objekti sortiraju prema udaljenosti od kamere. Na ovaj način će se minimizirati ranije opisani efekt dvostruko povezane liste. Anologija je da se sada zamišljenoj listi pristupa što je više moguće slijedno umjesto potpuno slučajno. Izvršna datoteka nalazi se u mapi "IzvorneDatoteke\SortiraniObjekti".

Rezultati:

Performanse su očekivano porasle, no značajno manje nego je bilo očekivano. FPS je porastao na **11,3** odnosno broj iscrtanih slika je porastao na **185,6**, a kad bi se zanemarili nagli padi FPS-a opisani ranije, FPS bi porasto na oko **14**. Poražavajuća činjenica je da je ovo i dalje manje od neoptimiziranog modela, što vodi do zaključka da osim efekta dvostruko vezane liste još nešto značajno usporava ovaj algoritam. Pretpostavljam da je glavni problem u tome što se model prvo mora prenijeti s grafičke memorije u radnu, obraditi, te vratiti natrag u grafičku memoriju, da bi se ostvarila promjena broja vrhova modela. To je problem iz dva razloga; prvi je sporost takvog prijenosa, a drugi, mnogo veći problem je to što se narušava koncept komunikacije grafičke kartice s procesorom. U općem slučaju grafička kartica i procesor rade paralelno na način da procesor iz radne memorije prenese sve modele u memoriju grafičke kartice. Od ovog trenutka nadalje grafička kartica od procesora samo prima instrukcije koje primjenjuje na sve modele koje sadrži. Ovakvom arhitekturom minimizira se ovisnost procesora i grafičke kartice čime se omogućuje efikasni paralelni rad. Da bi se ostvario siguran prijenos, grafička kartica mora privremeno prestati s radom kako bi dopustila procesoru prijenos podataka. Ovo je izrazito štetno za preformanse jer je brzina rada grafičke kartice u razmjeru s brzinom prijenosa podataka značajna. Dakle, daljnja optimizacija treba ići u smjeru ograničenja prijenosa podataka između grafičke i radne memorije tijekom izračuna scena.

4.2.5. Diskretne razine detalja(LOD)

Najjednostavniji način da se ograniči prijenos podataka je da se on obavi prije samog izračuna slike. Ovaj algoritam unaprijed podijeli model na razine detaljnosti i pohrani ih u memoriju grafičke kartice. Algoritam to radi na način da, kao i svi algoritmi dosad, od orginalno učitanog modela načini progresivnu mrežu koja se zatim dijeli na diskretne razine; odbace se sve operacije kolapsa i spajanja bridova izvan okvira pojedine

razine. Potom na temelju udaljenosti od kamere algoritam samo bira koji od modela se treba iscrtati. Očiti nedostatak algoritma je velika količina memorije potrebna za pohranu svih razina modela. U testu je broj diskretnih razina postavljen na 20. Izvršna datoteka nalazi se u mapi "IzvorneDatoteke\LOD60".

Rezultati:

Algoritam je postigao impresivnih **59,95** FPS-a i **1199** iscrtanih slika. Korištenje LOD algoritma ovdje je opravdano jer svi objekti u testu koriste isti model, tako da je memorijski "overhead" (predugo izračunavanje algoritma ili pretjerano korištenje memorije) u ovom slučaju minimalan. U općem slučaju, svaki objekt bi mogao imati vlastiti model. Memorijska zahtjevnost bi vrlo brzo postala neprihvatljiva, posebice kod vrlo velikih modela, stoga je nužno napraviti alternativni algoritam. Algoritam je također izgubio svojstvo glatkog prijelaza između razina detaljnosti, no sa dovoljno velikim brojem razina ovo može biti neprimjetno.

4.2.6. Virtualne diskretizirane razine detajla

Kao i prethodni algoritam, virtualni LOD dijeli model na diskretne razine detaljnosti, samo što je podjela sada samo logička. Umjesto da algoritam bira model s brojem vrhvova koji je najbliži izračunatom, on postavlja broj vrhova na neku od diskretnih razina ukoliko već nije dovoljno blizu toj razini. Na ovaj način izbjegnuta je memorijska zahtjevnost prethodnog algoritma, kao i prijenos podataka između grafičke i radne memorije.

Rezultati:

Performanse su očekivano pale u razmjeru s LOD-om na **27,2** FPS-a i **533** iscrtane slike. Razlog je djelomično u tome što je ovaj test najgori slučaj za primjenu progresivnih mreža pa je usporedba s LOD modelom neopravdana. Isto kao i LOD, ovaj algoritam ne posjeduje mogučnost glatkog prijelaza između razina. Algoritam je najprikladniji za iscrtavanje vrlo velikih modela koji posjeduju mali broj objekata.

4.2.7. Progresivne diskretizirane razine detalja

Prethodna dva algoritma zanemarila su vrlo bitnu karakteristiku progresivnih mreža – gotovo neprimjetan prijelaz između razine detalja. Algoritam progresivne diskretizirane

razine detalja radi identično kao i LOD, samo što nakon odabira prikladnog modela istom podesi detaljnost sukladno s točnim iznosom broja vrhova koji bi trebao imati. Na ovaj način se zadržava svojstvo glatkog prijelaza uz cijenu pada performansi.

Rezultati:

Algoritam je ostvario **21,61** FPS i **434** iscrtane slike. Prikladan je samo u slučaju kad je glatkost prijelaza nužna.

5. Moguće nadogradnje i druga riješenja

Problem prosljeđivanja podataka grafičkoj kartici nije izoliran na progresivne mreže. Vrlo malo efekata se može napraviti samo s elementarnim transformacijama, stoga su oko 2001. godine neke grafičke kartice omogućile direktan unos programa u memoriju grafičke kartice. Takvi programi se zovu *shaderi*; u okviru DirectX-a 9.0c postoje *vertex* i *pixel shaderi*. Premda su izrazito korisni za razne efekte, zbog nemogućnosti promjene broja vrhova modela ne mogu se primijeniti na progresivne mreže. Oko 2007. godine u okviru DirectX-a 10 predstavljeni su *geometry shaderi* koji mogu promijeniti broj vrhova modela unutar grafičke memorije kartice bez prenošenja modela u radnu memoriju. Ova mogućnost rješava glavni problem performansi pri upotrebi progresivnih mreža kao dinamičnu alternativu LOD algoritma.

6. Zaključak

Od svih testiranih algoritama samo posljednja tri mogu se iskoristiti u praksi. Algoritam direktnih promjena detaljnosti pokazao se kao najefikasniji od svih testiranih algoritama, uz cijenu potencijalno vidljivih prijelaza i velike memorijske zahtjevnosti. Najbolje je primjenjiv u scenariju gdje puno objekata koristi isti model i gdje taj model nije prekompleksan. Algoritam virtualnih diskretiziranih promjena detaljnosti je najbolje koristiti kad malo objekata dijeli isti model, te kad je model prevelik da bi se od njega konstruiralo klasični LOD. Algoritam progresivne diskretizirane promjene detaljnosti je prkladno koristiti samo u slučaju kada je dobitak glatkog prijelaza između modela opravdan.

7. Literatura

[1] Hugues Hoppe., *Progressive meshes,* Microsoft Research, 1996.

[2] Peter Walsh., I. *Advanced 3D Game Programming with DirectX 9.0*,Wordware Publishing © 2003.

[3] Microsoft DirectX 9.0 SDK, *DirectX documentation for c++,* 2007.

8. Sažetak

Geometrijski modeli s mnogo detaljne geometrije postaju sve češća praksa u računalnoj grafici. Takvi kompleksni modeli (prikazani mrežom trokuta) zadaju probleme prilikom iscrtavanja, prijenosa preko mreže, te prilikom samog spremanja na disk (zbog svoje veličine). Kako bismo riješili ove probleme koristimo se progresivnim mrežama, odnosno metodama za sažeto spremanje mreža trokuta proizvoljnog oblika (modela). Progresivne mreže (eng. *Progressive meshes, PM*) na učinkovit način pojednostavljuju komplicirane modele bez gubitaka, te rješavaju više od jednog problema u računalnoj grafici:

a. Pojednostavljenje mreže poligona

b. Razinu detalja (*eng. Level of Detail, LOD*)

c. Progresivni prijenos

d. Kompresija mreže poligona

e. Selektivno poboljšanje

Fokus ovog rada je izrada aplikacije pomoću koje će se nizom pokusa istražiti potencijalna primjena progresivnig mreža na problem dinamičke promjene razine detalja tijekom izvođenja aplikacije u realnom vremenu.

9. Abstract

Geometrical models with a lot of detailed geometry are becoming more popular in computer graphics. Such complex models (represented by triangle meshes) are problematic during rendering, network transfer and saving to the disk (because of their size). In order to solve these problems we use progressive meshes (PMs), various methods of saving compressed triangle meshes of arbitrary shapes (models). Progressive meshes efficiently simplify complicated models without losses, solving more than one problem in computer graphics:

- simplification of the poligon grid
- level of detail (LOD)
- progressive transfer
- poligon grid compression
- selective improvement

The focus of this paper is the development of an application with which the potential application of progressive meshes to the problem of dynamic detail changes in run-time will be explored via a series of tests.

10. Privitak

10.1. Ispis svih provedenih testova

U mapi "Testovi" na CD nalaze se ispisi FPS-a svake sekunde, trajanje proračuna svake slike, minimalni, maksimalni i prosječni FPS te broj iscrtanih slika za svaki opisani pokus.

10.2. Izvorni kod bitnijih algoritama

10.2.1. Implementicija virtualne diskretizirane razine detalja

```
class PureProgressiveMesh: public GraphicObj {
private:
        LPD3DXPMESH progressiveMesh;
        LPD3DXBUFFER pAdjacencyBuffer;
        D3DMATERIAL9* material;    // define the material object
        LPDIRECT3DTEXTURE9* texture;    // a pointer to a texture
        DWORD numMaterials;
        DWORD cVerticesMin, cVerticesMax;
        DWORD cVerticesPerMesh;
        DWORD numPMeshes;

public:
        int numVertices;
        PureProgressiveMesh(LPCWSTR file, LPDIRECT3DDEVICE9 d3ddev) {
                LPD3DXMESH mesh;
                LPD3DXBUFFER bufShipMaterial;
                D3DXLoadMeshFromX(file,    // load this file
```

```
                                                    D3DXMESH_MANAGED,      // load the mesh into
system memory
                                                    d3ddev,    // the Direct3D Device
                                                    &pAdjacencyBuffer,    // we aren't using adjacency
                                                    &bufShipMaterial,    // put the materials here
                                                    NULL,    // we aren't using effect instances
                                                    &numMaterials,     // the number of materials in this
model
                                                    &mesh);    // put the mesh here

                    // retrieve the pointer to the buffer containing the material information
                    D3DXMATERIAL*       tempMaterials       =       (D3DXMATERIAL*)bufShipMaterial-
>GetBufferPointer();

                    // create a new material buffer and texture for each material in the mesh
                    material = new D3DMATERIAL9[numMaterials];
                    texture = new LPDIRECT3DTEXTURE9[numMaterials];

                    for(DWORD i = 0; i < numMaterials; i++)    // for each material...
                    {
                              material[i] = tempMaterials[i].MatD3D;    // get the material info
                              material[i].Ambient = material[i].Diffuse;

                              USES_CONVERSION;    // allows certain string conversions
                              // if there is a texture to load, load it
                              if(FAILED(D3DXCreateTextureFromFile(d3ddev,

CA2W(tempMaterials[i].pTextureFilename),

          &texture[i])))
                                        texture[i] = NULL;    // if there is no texture, set the texture to NULL
```

```
                }

            // Perform simple cleansing operations on mesh
                LPD3DXMESH pTempMesh;

                D3DXCleanMesh( D3DXCLEAN_SIMPLIFICATION, mesh,
(DWORD*)pAdjacencyBuffer->GetBufferPointer(), &pTempMesh,
                    (DWORD*)pAdjacencyBuffer->GetBufferPointer(), NULL );
                mesh->Release();
                mesh = pTempMesh;

                D3DXWeldVertices( mesh, 0, NULL,
                    (DWORD*)pAdjacencyBuffer->GetBufferPointer(),
                   (DWORD*)pAdjacencyBuffer->GetBufferPointer(), NULL, NULL);
  // Generate progressive meshes
                D3DXGeneratePMesh( mesh, (DWORD*)pAdjacencyBuffer->GetBufferPointer(),
                                    NULL, NULL, 0, D3DXMESHSIMP_VERTEX, &progressiveMesh );

    cVerticesMin = progressiveMesh->GetMinVertices();
          cVerticesMax = progressiveMesh->GetMaxVertices();
        cVerticesPerMesh = ( cVerticesMax - cVerticesMin + 10 ) / 20;

        }
        virtual void draw(LPDIRECT3DDEVICE9 d3ddev,VCam* cam, D3DXVECTOR3 position)
        {
        D3DXVECTOR3 cameraDistance = *(cam->getPosition()) - position;

        numVertices = ((cVerticesMax - cVerticesMin) * (500/
(500+D3DXVec3Length(&cameraDistance)))) + cVerticesMin;

        if(abs((int)(numVertices - progressiveMesh->GetNumVertices()))>cVerticesPerMesh)
```

```
                progressiveMesh->SetNumVertices(numVertices);
        for(DWORD i = 0; i < numMaterials; i++)    // loop through each subset_
                {
                        d3ddev->SetMaterial(&material[i]);    // set the material for the subset
                        if(texture[i] != NULL)    // if the subset has a texture (if texture is not NULL)
                                d3ddev->SetTexture(0, texture[i]);    // ...then set the texture
                        else
                                d3ddev->SetTexture(0, NULL);



                        progressiveMesh->DrawSubset(i);    // draw the subset
                }

        }
};
```

10.2.2. Implementacija progresivne diskretizirane razine detalja

```
class ProgressiveMesh: public GraphicObj {
private:
        LPD3DXPMESH* LODPMeshes;
        LPD3DXBUFFER pAdjacencyBuffer;
   D3DMATERIAL9* material;    // define the material object
        LPDIRECT3DTEXTURE9* texture;    // a pointer to a texture
        DWORD numMaterials;
        DWORD numPMeshes;
        DWORD cVerticesPerMesh;
        DWORD cVerticesMin, cVerticesMax;

public:
        int numVertices;
```

```
ProgressiveMesh(LPCWSTR file, LPDIRECT3DDEVICE9 d3ddev) {
        LPD3DXMESH mesh;
        LPD3DXBUFFER bufShipMaterial;
        LPD3DXPMESH progressiveMesh;
        D3DXLoadMeshFromX(file,    // load this file
                          D3DXMESH_MANAGED,      // load the mesh into system memory
                          d3ddev,    // the Direct3D Device
                          &pAdjacencyBuffer,    // we aren't using adjacency
                          &bufShipMaterial,    // put the materials here
                          NULL,    // we aren't using effect instances
                          &numMaterials,     // the number of materials in this model
                          &mesh);    // put the mesh here

        // retrieve the pointer to the buffer containing the material information
        D3DXMATERIAL*      tempMaterials      =      (D3DXMATERIAL*)bufShipMaterial->GetBufferPointer();

        // create a new material buffer and texture for each material in the mesh
        material = new D3DMATERIAL9[numMaterials];
        texture = new LPDIRECT3DTEXTURE9[numMaterials];

        for(DWORD i = 0; i < numMaterials; i++)    // for each material...
        {
                material[i] = tempMaterials[i].MatD3D;    // get the material info
                material[i].Ambient = material[i].Diffuse;

                USES_CONVERSION;    // allows certain string conversions
                // if there is a texture to load, load it
                if(FAILED(D3DXCreateTextureFromFile(d3ddev,
```

```
CA2W(tempMaterials[i].pTextureFilename),

            &texture[i])))
                                    texture[i] = NULL;    // if there is no texture, set the texture to NULL
                    }
                    LPD3DXMESH pTempMesh;

                    D3DXCleanMesh(          D3DXCLEAN_SIMPLIFICATION,          mesh,
(DWORD*)pAdjacencyBuffer->GetBufferPointer(), &pTempMesh,
                        (DWORD*)pAdjacencyBuffer->GetBufferPointer(), NULL );
                    mesh->Release();
                    mesh = pTempMesh;
                    // Weld the mesh using all epsilons of 0.0f.  A small epsilon like 1e-6 works well too
                    D3DXWeldVertices( mesh, 0, NULL,
                        (DWORD*)pAdjacencyBuffer->GetBufferPointer(),
                        (DWORD*)pAdjacencyBuffer->GetBufferPointer(), NULL, NULL);
      // Generate progressive meshes
                    D3DXGeneratePMesh( mesh, (DWORD*)pAdjacencyBuffer->GetBufferPointer(),
                                    NULL, NULL, 0, D3DXMESHSIMP_VERTEX, &progressiveMesh );

                    cVerticesMin = progressiveMesh->GetMinVertices();
                cVerticesMax = progressiveMesh->GetMaxVertices();

                cVerticesPerMesh = ( cVerticesMax - cVerticesMin + 10 ) / 20;

                    numPMeshes = max( 1, (DWORD)ceil( (cVerticesMax - cVerticesMin + 1) /
(float)cVerticesPerMesh ) );
                    LODPMeshes = new LPD3DXPMESH[numPMeshes];
                    ZeroMemory( LODPMeshes, sizeof(LPD3DXPMESH) * numPMeshes );
    // Clone full size pmesh
```

```
                LPD3DXPMESH pMeshFull;
                progressiveMesh->ClonePMeshFVF( D3DXMESH_MANAGED | D3DXMESH_VB_SHARE, progressiveMesh->GetFVF(), d3ddev, &pMeshFull );

  // Clone all the separate pmeshes
                for( UINT iPMesh = 0; iPMesh < numPMeshes; iPMesh++ )
                {
                        progressiveMesh->ClonePMeshFVF( D3DXMESH_MANAGED | D3DXMESH_VB_SHARE, progressiveMesh->GetFVF(), d3ddev, &LODPMeshes[iPMesh] );

    // Trim to appropriate space
                        LODPMeshes[iPMesh]->TrimByVertices( cVerticesMin + cVerticesPerMesh * iPMesh, cVerticesMin + cVerticesPerMesh * (iPMesh+1), NULL, NULL);

                        LODPMeshes[iPMesh]->OptimizeBaseLOD( D3DXMESHOPT_VERTEXCACHE, NULL );
                        LODPMeshes[iPMesh]->SetNumVertices(iPMesh*numPMeshes+30);
                }
}
        virtual void draw(LPDIRECT3DDEVICE9 d3ddev,VCam* cam, D3DXVECTOR3 position)
        {
        D3DXVECTOR3 cameraDistance = *(cam->getPosition()) - position;
        numVertices = ((cVerticesMax - cVerticesMin) * (500/(500+D3DXVec3Length(&cameraDistance)))) + cVerticesMin;

        LPD3DXPMESH progressiveMesh = LODPMeshes[numVertices/cVerticesPerMesh];
        progressiveMesh->SetNumVertices(numVertices);

        for(DWORD i = 0; i < numMaterials; i++)    // loop through each subset
                {
                        d3ddev->SetMaterial(&material[i]);    // set the material for the subset
```

```
                        if(texture[i] != NULL)    // if the subset has a texture (if texture is not NULL)
                                d3ddev->SetTexture(0, texture[i]);    // ...then set the texture
                        else
                                d3ddev->SetTexture(0, NULL);
                        progressiveMesh->DrawSubset(i);    // draw the subset
                }

        }
};
```