UNIVERZITET U BEOGRADU

MATEMATIČKI FAKULTET

Atila Čeki

USAVRŠAVANjE MODELA TESNO DVOJNIH SISTEMA SAGLASNO SA REZULTATIMA POSMATRANjA VISOKE PRECIZNOSTI

doktorska disertacija

Beograd, 2013

UNIVERSITY OF BELGRADE

FACULTY OF MATHEMATICS

Atila Čeki

IMPROVEMENTS TO THE MODEL OF CLOSE BINARIES IN ACCORDANCE WITH RESULTS FROM HIGH PRECISION OBSERVATIONS

Doctoral Dissertation

Belgrade, 2013

Mentor: prof. dr. Olga Atanacković
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije: prof. dr. Olga Atanacković
Univerzitet u Beogradu, Matematički fakultet

dr. Gojko Đurašević
Astronomska opservatorija, Beograd

dr. Ištvan Vince
Astronomska opservatorija, Beograd

Datum odbrane:

...to boldly go where no one has gone before.

Ova doktorska disertacija je nastala pod rukovodstvom Gojka Đuraševića, Ištvana Vincea i Olge Atanacković. Ovim putem im se zahvaljujem na saradnji, nesebičnoj pomoći i korisnim savetima. Posebno moram istaći ulogu Olivere Latković, bez čijeg doprinosa ova disertacija ne bi bila moguća. Hvala na svemu, ljubavi. Zahvalnost dugujem i Tatjani Jakšić na korisnim sugestijama koje su značajno poboljšale kvalitet ovog teksta, kao i na dugogodišnjem prijateljstvu. Na kraju bih se zahvalio mojim roditeljima, Ištvanu i Juditi Čeki, na bezrezervnoj ljubavi i podršci tokom celog mog školovanja.

Rad na disertaciji je finansiralo Ministarstvo prosvete, nauke i tehnološkog razvoja Republike Srbije kroz projekat „Fizika zvezda“ (br. 176004).

USAVRŠAVANjE MODELA TESNO DVOJNIH SISTEMA SAGLASNO SA REZULTATIMA POSMATRANjA VISOKE PRECIZNOSTI

U disertaciji je predstavljen program za modeliranje dvojnih sistema, *Infinity*, koji kombinuje tradicionalne principe modeliranja (Roche-ova geometrija, von Zeipel-ov zakon raspodele temperature, itd), sa novim metodama za geometrijsku reprezentaciju sistema.

Proučavanje tesno dvojnih zvezda ima dugu tradiciju u Srbiji, pre svega zahvaljujući radu dr Gojka Đuraševića. Program predstavljen u ovoj disertaciji je baziran na njegovom programu za modeliranje aktivnih tesno dvojnih sistema sa akrecionim diskom (Đurašević, 1991) i donosi nekoliko značajnih poboljšanja, koja se mogu svrstati u dve grupe:

U prvoj grupi su poboljšanja koja se odnose na generalizaciju modela. Pre svega, novi program je moguće primeniti i na sisteme sa ekscentričnim orbitama. Moguće je računanje i rešavanje inverznog problema za krive radijalnih brzina. Razvijen je nov algoritam za određivanje vidljivih delova komponenata tokom pomračenja koji omogućava preciznije modeliranje krivih sjaja adaptivnom podelom u regionu pomračenja. Takođe omogućava generalizaciju geometrijskog modela akrecionog diska, koji može biti koničnog ili toroidalnog oblika.

Druga grupa unapređenja se odnosi na modeliranje fizičkih procesa na komponentama. Najznačajnija promena se odnosi na računanje izlaznog fluksa zračenja. Naime, u Đuraševićevom programu se pretpostavlja da elementarne ćelije zrače kao apsolutno crno telo, dok se u ovom programu svakoj ćeliji pripisuje jednodimenzionalni Kurucz-ov model zvezdane atmosfere koji je izveden pod pretpostavkom lokalne termodinamičke ravnoteže. Poboljšan je i tretman uticaja fotometrijskih filtera na izračunatu krivu sjaja. Takođe je proširena lista podržanih filtera.

Dvojni sistem se opisuje Roche-ovim modelom. On pretpostavlja da su kretanje i oblik komponenata određeni superpozicijom gravitacione i centrifugalne sile, dok se sve ostale interakcije zanemaruju. Zvezde se tretiraju kao materijalne tačke (tzv. aproksimacija centralne kondenzacije), pa se njihova unutrašnja struktura ne uzima u

obzir. Ukoliko sistem sadrži akrecioni disk, njegova masa se zanemaruje, tj. pretpostavlja se da disk ne utiče na dinamiku sistema i oblik komponenata.

Za predstavljanje površine komponenata se koristi poligona mreža. Zvezde se aproksimiraju geodezijskom, a akrecioni disk osno-simetričnom rotacionom mrežom. U sklopu disertacije je predložen nov algoritam za rešavanje problema detekcije vidljivosti. On se zasniva na inverznom slikarskom algoritmu iz polja računarske grafike. Njegova najbitnija karakteristika je da ne zahteva druge informacije o geometriji dvojnog sistema osim poligonih mreža koje je opisuju. Ovo otvara mogućnost jednostavnog proširenja modela na egzotičnije konfiguracije kao što su hijerarhiski sistemi, sistemi sa ekscentričnim ili cirkumbinarnim akrecionim diskom, planetarni sistemi itd.

Pretpostavlja se da su unutar jedne elementarne ćelije sve relevantne veličine – temperatura, lokalno gravitaciono ubrzanje, izlazni fluks i radijalna brzina – konstante. Njeno zračenje je određeno gravitacionim potamnjenjem, potamnjenjem ka rubu i efektom refleksije. Gravitaciono potamnjenje se opisuje von Zeipel-ovim zakonom, dok se za potamnjenje ka rubu može koristiti jedna od pet aproksimacija (linearna, logaritamska, korena, kvadratna i nelinearna četvrtog reda).

Za modeliranje efekta refleksije, kao i za raspodelu temperature duž akrecionog diska se koristi pristup iz Đuraševićevog programa (Đurašević, 1991). Na lokalni intenzitet zračenja mogu uticati i aktivni regioni koji se opisuju svetlim ili tamnim pegama. Fluks izlaznog zračenja se dobija kao suma intenziteta po vidljivim ćelijama u datoj fazi. Kriva radijalnih brzina se računa kao superpozicija tri vrste kretanja – sopstvenog kretanja centra mase sistema, kretanja centra zvezde oko zajedničkog centra mase i kretanja elementarnih ćelija na površini zvezde oko centra zvezde.

Izračunate krive sjaja i radijalnih brzina se mogu direktno porediti sa posmatranjima. Za rešavanje obrnutog zadatka, tj. za procenu optimalnih parametara sistema pri kojima sintetička kriva sjaja najbolje fituje posmatranja, se koristi Nelder-Mead simpleks algoritam za minimizaciju sume kvadrata odstupanja između posmatranih i izračunatih vrednosti. Provera ispravnosti programa se vrši na nekoliko sistema poređenjem sa objavljenim rešenjima dobijenim Đuraševićevim programom.

Ključne reči: eklipsno dvojne zvezde – akrecioni diskovi – metode: numeričke – tehnike: fotometrijske – tehnike: radijalne brzine – zvezde: pojedinačne: EG Cep - zvezde: pojedinačne: AU Mon

Naučna oblast: Astronomija

Uža naučna oblast: Astrofizika

UDK broj: 524.387:[52-13/-17](043.3)

IMPROVEMENTS TO THE MODEL OF CLOSE BINARIES IN ACCORDANCE WITH RESULTS FROM HIGH PRECISION OBSERVATIONS

The dissertation presents a program for modeling binary systems, *Infinity*, which combines the traditional principles of modeling (Roche geometry, von Zeipel's law of temperature distribution and so forth) with a novel geometrical representation of the system.

The study of binary stars has a long tradition in Serbia, mostly due to the research done by dr Gojko Đurašević. The program presented in the dissertation is based on his program for modeling of active close binary systems with an accretion disk (Đurašević, 1991) and brings several significant improvements that can be classified in two categories:

The first group of improvements has to do with generalizing the model. First and foremost, the new program is applicable to systems with eccentric orbits, and is capable of computing and solving the inverse problem for radial velocities. It also features a new algorithm for determining which elements of the stellar surface are visible during the eclipse. The new algorithm allows for computing the light curves with greater precision using a method of 'adaptive subdivision' in the region of eclipse. The geometrical aspects of modeling the accretion disk are also improved and now include the possibility to create both conical and toroidal disks.

The second group of improvements is related to the modeling of physical processes on the binary system components. The most significant change is in the way the emergent flux is calculated. Namely, the Đurašević program assumes that the elements of stellar surface produce blackbody radiation, while the new program assigns a one-dimensional Kurucz stellar atmosphere model calculated under the LTE assumption. Improvements are also made in the treatment of the influence of photometric filters on the calculated light curve, and the list of supported filters is extended.

A binary system is described by the Roche model, which assumes that the motion and shape of the components are determined by the superposition of gravitational and centrifugal forces, while all the other interactions are neglected. The stars are treated as point masses (under the assumption of total central condensation), so their internal

structure is not considered. If the system contains an accretion disk, it is assumed that its mass can be neglected, and that its presence doesn't influence the dynamics of the system and the shape of the stars.

The surface of the components is represented with a polygonal mesh. The stars are approximated with a geodesic, and the accretion disk with an axially-symmetric mesh. The new algorithm for solving the problem of visibility detection, proposed in the dissertation, is based on the so called inverse painter's algorithm that originates from the field of computer graphics. The most important property of this algorithm is that, other than the polygonal meshes representing the components, it doesn't require any other information about the geometry of the system. This makes the modeling program easily extendable to more exotic geometric configurations such as systems with eccentric or circumbinary disks, planetary and multiple systems, and so forth.

It is assumed that within a single element of stellar surface all the physical quantities – local effective temperature, gravitational acceleration, emergent flux and radial velocity – are constant. Its radiation is affected by gravity darkening, limb darkening and the reflection effect. Gravity darkening is described using von Zeipel's law, and the limb darkening can be calculated using one of the five available approximations (linear, logarithmic, square-root, quadratic and nonlinear fourth-order approximation). The reflection effect, as well as the temperature distribution along the accretion disk, are calculated using the approach from the Đurašević program (Đurašević, 1991). The local amount of radiation can additionally be affected by active regions that are represented with dark or bright circular spots.

The emergent flux is computed as the sum of fluxes from all the surface elements visible in the given moment, and the radial velocity is calculated from the superposition from three kinds of motion: the proper motion of the system center of mass, the motion of the center of the star around the center of mass due to orbital revolution, and the motion of elementary surface around the center of the star due to rotation.

Synthetic light and radial velocity curves obtained in this way can be directly compared to observations. The new program solves the inverse problem – estimating the optimal model parameters to fit the given observations – using the Nelder-Mead simplex

algorithm to minimize the sum of squares of deviations between the observed and calculated light and radial velocity curves. The validity of the program and the proposed improvements is done by analyzing several binary systems and comparing the results with published solutions obtained with the Đurašević program.

Key words: binaries: eclipsing - accretion disks - methods: numerical - techniques: photometric - techniques: radial velocities - stars: individual: EG Cep - stars: individual: AU Mon

Scientific field: Astronomy

Scientific subfield: Astrophysics

UDK number: 524.387:[52-13/-17](043.3)

Sadržaj

1. Uvod

Istraživanja pokazuju da se većina zvezda sa glavnog niza nalazi u sistemima sa dve ili više komponenata (Abt, 1983). Dvojne zvezde su od posebnog značaja za astrofiziku, jer predstavljaju najvažniji izvor informacija o fundamentalnim fizičkim karakteristikama zvezda. Parametri kao sto su masa, radijus, luminoznost itd. se mogu odrediti sa visokom tačnošću. U nekim slučajevima je greška ispod 1%, kao što je na primer slučaj sa određivanjem mase sistema CM Dra (Metcalfe, et al., 1996).

Najkompletnije informacije se dobijaju iz eklipsnih dvojnih sistema kombinovanjem fotometrijskih i spektroskopskih posmatranja. Dramatični napredak posmatračkih tehnika u poslednjoj deceniji je doveo do toga da su postala dostupna veoma kvalitetna posmatranja – od fotometrijskih podataka sa svemirskih misija CoRoT i Kepler, preko javno dostupnih baza krivih sjaja sa robotizovanih teleskopa za snimanje celog neba kao što su OGLE, SuperWASP ili HAT, pa do merenja radijalnih brzina sa greškom manjom od $3\,m/s$ (Kambe, et al., 2008). Ovaj napredak zahteva i usavršavanje postojećih programa za modeliranje dvojnih sistema, odnosno izgradnju novih, koji bi adekvatno mogli da objasne fine detalje u posmatranjima. U disertaciji je izložen program koji može da odgovori na ove zahteve.

Računanje sintetičke krive sjaja i radijalnih brzina se grubo može podeliti u tri faze – uspostavljanje geometrijske reprezentacije sistema, detekciju vidljivosti i računanje izlaznog fluksa zračenja i radijalnih brzina. Jedan od osnovnih ciljeva prilikom pravljenja novog programa je bio striktno razdvajanje ove tri faze, odnosno podelu računa na tri nezavisne celine. Glavna prednost ovakvog pristupa je velika fleksibilnost modela, jer proširivanje jednog modula ne zahteva menjanje ostala dva. To znači da će buduće proširivanje modela biti znatno olakšano – na primer, dodavanje neradijalnih pulsacija na površinu komponenata sistema (Latković, 2013, u pripremi) ne zahteva promene u modulu za detekciju vidljivosti, itd.

Geometrijska reprezentacija obuhvata diskretizaciju površine zvezda i određivanje njihovog položaja na orbiti u datom vremenskom trenutku. U opštem slučaju su ova dva zadatka povezana, jer oblik zvezda opisanih Roche potencijalom na eliptičnoj orbiti

zavisi od njihovog položaja. Neke od strategija za razbijanje površine zvezde na elementarne ćelije su prikazane u poglavlju 2 na primerima pojedinačnih modela koji su od značaja za ovu disertaciju (odeljci 2.2-2.6). Metoda razvijena za potrebe ove disertacije, adaptivna geometrijska mreža, je opisana u poglavlju 3.

Pod detekcijom vidljivosti se podrazumeva algoritam koji za dati vremenski trenutak određuje koje su elementarne ćelije na površini zvezde vidljive iz pozicije posmatrača i samim tim doprinose detektovanom fluksu. Na žalost, većina autora potcenjuje značaj ove faze i ne posvećuje joj dovoljno pažnje u opisima svojih modela ili je u potpunosti preskače. Njen značaj leži u tome što od odabranog algoritma zavise granice primenljivosti modela, odnosno spektar konfiguracija koje se mogu njime opisati. Na primer, ako je algoritam izveden pod pretpostavkom da je projekcija zvezde na ravan nebeske sfere konveksan poligon, to automatski isključuje mogućnost modeliranja akrecionih diskova ili neradijalnih pulsacija na komponentama, itd. U poglavlju 4 se predlaže inovativni pristup ovom problemu koji je dovoljno fleksibilan da može da opiše i sisteme sa pulsirajućim komponentama, hijerarhiske i planetrane sisteme, sisteme sa ekscentričnim ili cirkumbinarnim akrecionim diskom itd.

U poglavlju 5 je izložen postupak sinteze krive sjaja i radijalnih brzina. Postupak računanja izlaznog fluksa u velikoj meri prati Đuraševićev program. Upravo monografija Đuraševića (1991), zajedno sa knjigom Kallrath-a i Milone-a (2009), predstavlja osnovnu literaturu za izradu ove disertacije. Najkrupnija izmena se odnosi na računanje izlaznog zračenja elementarnih ćelija. U Đuraševićevom programu se pretpostavlja da elementarne ćelije zrače kao apsolutno crno telo, dok se u *Infinity*-ju svakoj ćeliji pripisuje jednodimenzionalni Kurucz-ov model atmosfere koji odgovara lokalnim vrednostima temperature i gravitacionog ubrzanja. Druga novina je sinteza krive radijalnih brzina. U *Infinity*-ju je moguće simultano fitovanje krive sjaja i krive radijalnih brzina, odnosno dobijanje kompletnog spektrofotometrijskog rešenja. Takođe je značajno proširena lista podržanih fotometrijskih filtera (sa osam na trinaest).

U 6. glavi se opisuje rešavanje inverznog problema i pokazuje se ispravnost modela na nekoliko sistema koji su analizirani tokom izrade ove disertacije u okviru projekta „Fizika zvezda“ Ministarstva prosvete, nauke i tehnološkog razvoja Republike Srbije.

Objavljeni rezultati, dobijeni Đuraševićevim programom, se porede sa rezultatima iz *Infinity*-ja.

U 7. poglavlju se daje pregled najznačajnih rezultata proisteklih iz disertacije i diskutuje se o mogućim pravcima za buduće istraživanje. U poslednjoj glavi je prikazan listing programa za modeliranje dvojnih sistema.

2. Istorija modeliranja dvojnih sistema

U ovom poglavlju je dat kratak osvrt na istoriju modeliranja dvojnih sistema. Kako je nemoguće ovde pomenuti sve modele i ideje koje su oblikovale polje, izabrano je nekoliko najznačajnijih. Naglasak je stavljen na metode za diskretizaciju površine komponenata i detekciju vidljivosti da bi se prikazala evolucija ideja koje su dovele do algoritma prikazanog u ovoj disertaciji.

2.1. Od otkrića dvojnih zvezda do prvog modela

Krajem XVII veka su bile poznate mnoge vizuelno dvojne zvezde. Smatralo se da su neke od njih i fizički povezane, tj. da čine jedan zvezdani sistem, ali nije bilo čvrstih dokaza za tu tvrdnju. Herschel je bio prvi koji je pokazao da se prividno kretanje nekih zvezda na nebu može objasniti orbitalnim kretanjem oko zajedničkog centra mase u skladu sa Newton-ovim zakonom gravitacije. Poznato je njegovo određivanje orbitalnog perioda sistema koji sadrži Castor (α Gem). Njegov rezultat - 324 godine i 2 meseca (Herschel, 1803) je manji od moderne vrednosti (467 godina), ali je svakako impresivan, posebno s obzirom na dužinu perioda i instrumente kojima je raspolagao. Ovo otkriće je podstaklo formiranje mnogih posmatračkih programa, kao i razvoj metoda za određivanje geometrijskih karakteristika orbite (metode Savary-ja, Thiele-a, Innes-a, ...).

Paralelno sa ovim su se pojavile i ideje da je uzrok promene sjaja kod nekih tipova promenljivih zvezda međusobno pomračivanje komponenata dvojnog sistema. Najpoznatiji primer je Goodricke-ova tvrdnja da je periodična promena sjaja Algol-a (β Per) posledica postojanja velikog objekta koji se kreće oko njega i periodično ga pomračuje (Goodricke, 1783). No, bilo je potrebno više od jednog veka da se nedvosmisleno pokaže da je zaista u pitanju dvojni sistem. Vogel je iz spektroskopskih posmatranja utvrdio da se Algol udaljava od posmatrača pre primarnog minimuma, dok se posle njega približava posmatraču (Vogel, 1890), što je u skladu sa objašnjenjem da se Algol zajedno sa nevidljivim pratiocem kreće oko zajedničkog centra mase. Ovo je ujedno bilo i otkriće novog metoda za utvrđivanje dvojnosti sistema u slučajevima gde to nije bilo moguće učiniti vizuelno. Ova klasa dvojnih zvezda je dobila ime spektroskopski dvojne.

Poslednji deo slagalice neophodan za formulisanje prvog modela eklipsnih dvojnih sistema je bio početak upotrebe fotografije u astronomiji. Do 1900. godine je većina dvojnih zvezda otkrivena vizuelnim posmatranjima. Nova tehnika je donela i prve projekte koji su imali za cilj masovnu pretragu neba za promenljivim zvezdama. To je značajno povećalo količinu i kvalitet dostupnog posmatračkog materijala i omogućilo stvaranje prvog modela dvojnih sistema koji je formulisao Russell 1912 godine.

2.2. Russel-Merrill model

Russell-Merrill-ov model je dugo bio standard u modeliranju eklipsno dvojnih sistema. Čak bi se moglo reći da je do sedamdesetih godina prošlog veka većina rezultata u oblasti dobijena upravo pomoću ovog modela. Njegova upotreba se naglo smanjuje sa pojavom novih modela zasnovanih na Roche geometriji koji mnogo bolje opisuju fiziku dvojnih sistema. Ovu revoluciju je omogućio početak upotrebe računara za rešavanje astrofizičkih problema. Neki od tih modela će biti opisani u narednim odeljcima.

Osnove modela su izložene u Russell-ovim člancima (1912a) i (1912b). Model polazi od pretpostavke da su zvezde sfernog oblika (kasnija proširenja koristie troosni elipsoid) i da se kreću po kružnim putanjama oko zajedničkog centra mase. Zračenje komponenata se opisuje Planck-ovim zakonom. U obzir su uzeti gravitaciono potamnjenje, linearno potamnjenje ka rubovima i jako jednostavan tretman efekta refleksije.

Iako je bilo poznato da u tesno dvojnim sistemima oblik komponenata odstupa od sfernog, to odstupanje nije bilo moguće modelirati analitičkim metodama koje su se tada koristile. Da bi se prevazišao taj problem, Russell je razvio postupak za rektifikaciju krive sjaja. Tim postupkom se uklanjaju efekti koji potiču od blizine komponenata, odnosno posmatranja se svode na oblik na koji se može primeniti Russell-Merrill model. Rektifikacija počinje Fourier-ovom analizom dela posmatrane krive sjaja van pomračenja. Posmatranja se aproksimiraju sledećim konačnim redom:

Jednačina 2.1

$$l(\theta) = \sum_{m=0}^{n} A_m \cos m\theta + B_m \sin m\theta,$$

gde je θ geometrijska faza definisana u odeljku 3.3, a l izmereni fluks zračenja. Koeficijenti A_1 i A_2 opisuju efekat refleksije, dok koeficijent B_1 opisuje razliku u visini maksimuma krive sjaja pre i posle sekundarnog minimuma (O'Connel-ov efekat). Obično su se pri analizi uzimali u obzir samo članovi do $n = 2$. Ovde će biti dat primer za korekciju deformacije oblika usled blizine druge zvezde (elipsoidna varijacija).

Elipsoidna varijacija dovodi do promene fluksa na sledeći način

Jednačina 2.2

$$l_{comp} = (1 - \frac{1}{2} Nz \cos^2 \theta)(l_{max} - l_{1max} f_1 - l_{2max} f_2).$$

Indeksi 1 i 2 se odnose na primarnu i sekundarnu komponentu. Veličine f_1 i f_2 opisuju smanjenje fluksa usled pomračenja u datoj fazi – tzv. izgubljeno svetlo. z se definiše kao

Jednačina 2.3

$$z = \eta^2 \sin^2 i,$$

gde je η mera odstupanja troosnog elipsoida koji opisuje zvezdu od sfere. Veličina N kombinuje uticaj gravitacionog potamnjenja i potamnjenja ka rubu i definiše se na sledeći način

Jednačina 2.4

$$N = \frac{(15 + x)\left(1 - \frac{K - 2}{4K + 2} y\right)}{15 - 5x} \approx \frac{(15 + x)(1 + y)}{15 - 5x}.$$

K je veličina koja zavisi od stepena centralne kondenzacije zvezde i uzima vrednosti iz intervala 0.0018-0.018. i je inklinacija, x koeficijent potamnjenja ka rubu, a

Jednačina 2.5

$$y = \frac{c_2}{4\lambda T} \frac{e^{\frac{c_2}{\lambda T}}}{e^{\frac{c_2}{\lambda T}} - 1}, \quad c_2 = \frac{hc}{k}.$$

Ako je posmatrani fluks dat sa $l = A_0 + A_2 \cos 2\theta$, u maksimumu krive sjaja će biti $l_{max} = A_0 - A_2$ i $Nz = -4 \frac{A_2}{A_0 - A_2}$. Rektifikovana kriva sjaja se dobija kao

Jednačina 2.6

$$l_{rect} = \frac{(A_0 - A_2) l_{comp}}{A_0 + A_2 \cos 2\theta} = l_{max}(1 - L_1 f_1 - L_2 f_2),$$

gde su $L_1 = \frac{l_{1max}}{l_{max}}$ i $L_2 = \frac{l_{2max}}{l_{max}}$.

Geometrijska faza se popravlja na sledeći način:

Jednačina 2.7

$$\sin^2 \Theta = \frac{\sin^2 \theta}{1 - z \cos^2 \theta}.$$

Na sličan način se koriguje kriva sjaja i za ostale efekte. Ovako pripremljena posmatranja se onda analiziraju da bi se odredili parametri sistema. Karakteristika ovog modela je da se posmatranja nisu direktno poredila sa sintetičkom krivom sjaja, već su se iz njih izračunale određene pomoćne veličine koje su bile pogodne za predstavljanje tablicama ili nomogramima.

Russell je definisao veličinu $\alpha = \alpha(\delta, k, x_1, x_2)$ kao odnos izgubljenog svetla u datoj fazi i izgubljenog svetla pri drugom ili trećem kontaktu pomračenja. Ona zavisi od rastojanja projekcija centara komponenata na ravan nebeske sfere δ (o ovom koordinatnom sistemu će biti više reči u odeljku 3.3.1), odnosa radijusa komponenata $k = \frac{r_2}{r_1}$ i koeficijenata potamnjenja ka rubu x_1 i x_2. Veza između ovih veličina je prikazana na slici 2.1.

Ako imamo totalno pomračenje, α će biti jednako

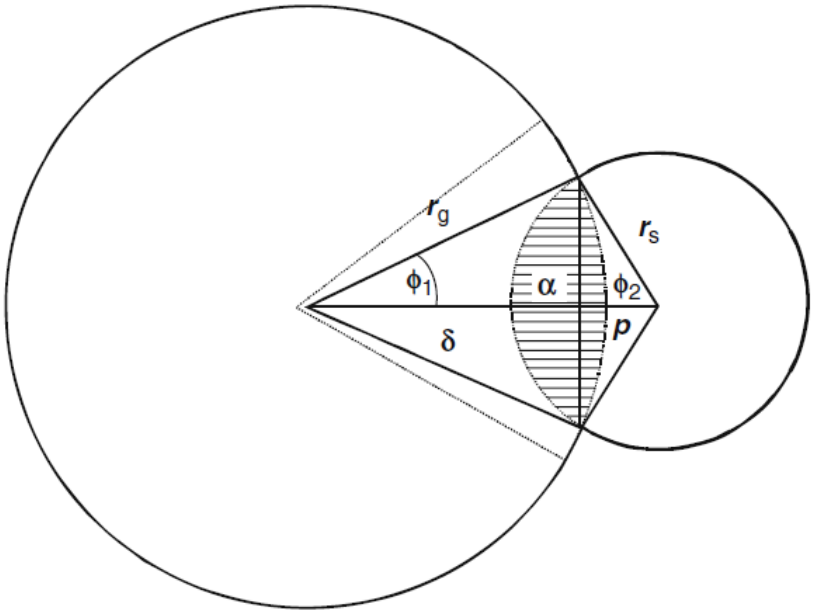
Jednačina 2.8

$$\alpha = \frac{1 - l}{1 - l_{min}},$$

gde je l_{min} posmatrani minimalni intenzitet. Ova jednačina nam daje vezu između α i geometrijske faze. Kao što je označeno na slici 2.1, α će biti proporcionalno pomračenom delu zvezde. Može se pokazati da je:

Jednačina 2.9

$$\alpha = f\left(\frac{\delta}{r_1}, \frac{r_2}{r_1}\right) = f\left(\frac{\delta}{r_1}, k\right).$$



Slika 2.1 Veza između separacije δ, odnosa radijusa komponenata k i α. α je proporcionalan šrafiranoj površini pomračenog dela zvezde. Preuzeto iz Kallrath & Milone (2009).

Za svaku zadatu vrednost k se ova jednačina može invertovati, odnosno napisati u obliku

Jednačina 2.10

$$\frac{\delta}{r_1} = \phi(k, \alpha).$$

Ako uzmemo da je rastojanje između centara komponenata na kružnoj orbiti jedinično, rastojanje njihovih projekcija je za datu geometrijsku fazu θ

Jednačina 2.11

$$\delta^2 = \sin^2\theta + \cos^2 i \cos^2\theta = \cos^2 i + \sin^2 i \sin^2\theta,$$

pa je

Jednačina 2.12

$$\cos^2 i + \sin^2 i \sin^2\theta = r_1^2 \phi^2(k, \alpha).$$

Ako odaberemo tri karakteristične faze, θ, θ_1 i θ_2, odnosno njima odgovarajuće vrednosti α, α_1 i α_2, možemo formirati sledeći količnik:

Jednačina 2.13

$$\frac{\sin^2\theta - \sin^2\theta_1}{\sin^2\theta_1 - \sin^2\theta_2} = \frac{\phi^2(k,\alpha) - \phi^2(k,\alpha_1)}{\phi^2(k,\alpha_1) - \phi^2(k,\alpha_2)} = \psi(k,\alpha,\alpha_1,\alpha_2).$$

Russell preporučuje da se vrednosti α_1 i α_2 fiksiraju na, redom, 0.6 i 0.9, tako da se uz smene $A = \sin^2\theta_1$ i $B = \sin^2\theta_1 - \sin^2\theta_2$ prethodna jednačina može pisati u obliku

Jednačina 2.14

$$\psi(k,\alpha) = \frac{\sin^2\theta - A}{B}.$$

Za sve razumne vrednosti argumenata ove funkcije se može formirati tablica i iz nje se interpolacijom može dobiti k za izabranu fazu θ. Obično se odnos radijusa komponenata određuje na osnovu nekoliko tačaka i za konačnu vrednost uzima srednja vrednost.

Neka je θ' faza kada počinje pomračenje (prvi kontakt). Separacija je u tom slučaju jednaka $\delta' = r_1 + r_2$ i $\alpha' = 0$. Isto tako, trenutku početka totalnog pomračenja (drugi kontakt) odgovaraju separacija $\delta'' = r_1 - r_2$ i $\alpha'' = 0$. Iz jednačina 2.12 i 2.14 imamo

Jednačina 2.15

$$\sin^2\theta' = A + B\psi(k,0) = \frac{r_1^2}{\sin^2 i}(1+k)^2 - \cot^2 i\,,$$

$$\sin^2\theta'' = A + B\,\psi(k,0) = \frac{r_1^2}{\sin^2 i}(1-k)^2 - \cot^2 i.$$

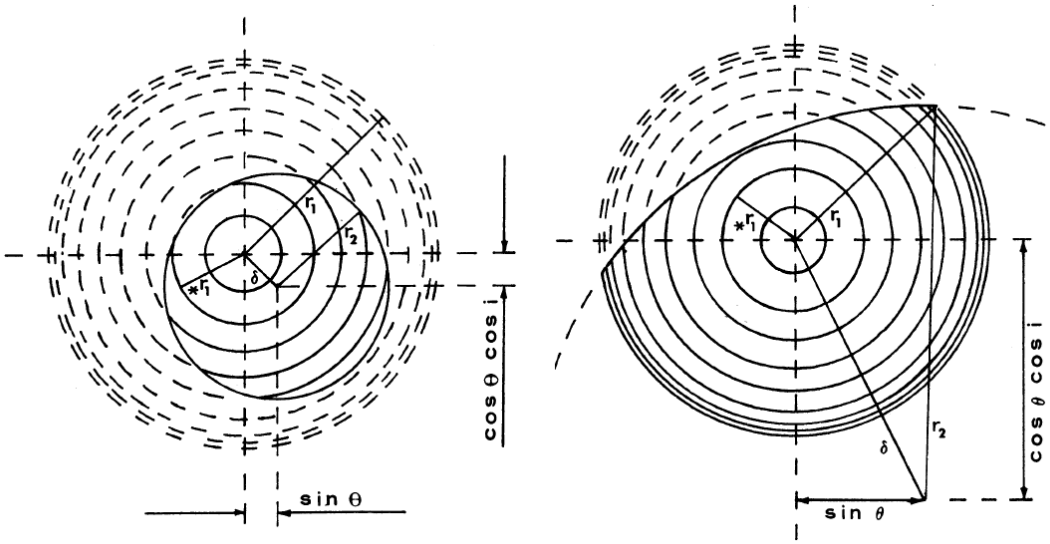
Iz ove dve jednačine se mogu odrediti i r_1 i i. Time smo odredili sve geometrijske parametre dvojnog sistema.

2.3. Eclipsing Binary Orbit Program (EBOP)

EBOP predstavlja implementaciju modela Nelson-a i Davis-a (1972) koju je napisao Etzel (1981). Ovaj model pripada grupi tzv. geometrijskih modela, slično Russell-Merrill-ovom modelu, kod kojih se efekti pomračenja mogu odrediti analitičkim putem. Odlikuje ga izuzetna brzina izvršavanja i zato se i dan danas koristi za analizu velikog broja krivih sjaja (Devor (2005) i Tamuz et al. (2006)) ili proučavanje tranzita egzoplaneta (Southworth (2008)).

Osnovne pretpostavke ovog modela su veoma slične Russell-Merrill modelu. Zvezde se modeliraju sferoidima (dvo-osnim elipsoidima) sa linearnim zakonom potamnjenja ka rubu. Efekti blizine druge komponente se zanemaruju, pa je ovaj model primenljiv samo na dobro razdvojene sisteme. Mogu se modelirati i kružne i eliptične orbite. Za razliku od Russell-Merrill modela, *EBOP* proizvodi krive sjaja koje se mogu direktno porediti sa posmatranjima.

Smanjenje fluksa tokom pomračenja se računa na sledeći način. Pokrivena zvezda se podeli na koncentrične prstenove radijusa $r \sin\theta$ i širine $\Delta\gamma = r\cos\theta d\theta$, gde je θ ugaono rastojanje od centra diska zvezde (slika 2.2). Ukupni fluks se dobija integracijom po celoj površini diska zvezde. Za svaki od ovih prstenova je lako izračunati koji deo je pokrivem drugom zvezdom. Formule za presek prstena i kruga su jednostavne za implemetaciju u vidu računarskog programa i odlikuju se izuzetno brzim izvršavanjem. Tačnost modela zavisi od izabrane širine $\Delta\gamma$. Na primer, $\Delta\gamma = 5°$ daje relativnu grešku od 10^{-4}, što je dovoljno za većinu primena.



Slika 2.2 Podela pomračene zvezde na koncentrične prstenove. Preuzeto iz Nelson & Davis (1972).

Parametri modela za sferne zvezde na kružnoj orbiti su:

- Relativni površinski sjaj centra diska sekundarne komponente J_2 (u odnosu na primarnu komponentu, tj. $J_1 = 1$)
- Relativni radijus primarne komponente r_1 (u odnosu na veliku poluosu orbite)

- Odnos radijusa sekundarne i primarne komponente $k = \frac{r_2}{r_1}$
- Inklinacija i
- Koeficijenti potamnjenja ka rubovima x_1 i x_2
- Treće svetlo $L_3 = 1 - L_1 - L_2$
- Širina prstenova $\Delta\gamma$

Indeksi 1 i 2 se odnose na primarnu i sekundarnu komponentu. Ako se pretpostavi linearni zakon potamnjenja ka rubu, nenormirana luminoznost komponenata će biti

Jednačina 2.16

$$l_2 = \pi J_2 r_2^2 \left(1 - \frac{x_2}{3}\right),$$
$$l_1 = \pi J_1 r_1^2 \left(1 - \frac{x_1}{3}\right),$$
$$\frac{l_2}{l_1} = k^2 J_2 \frac{1 - \frac{x_2}{3}}{1 - \frac{x_1}{3}}.$$

Odnos luminoznosti zavisi samo od odnosa radijusa, odnosa površinskih sjajeva centra diska i koeficijenata za potamnjenje ka rubu. Normiranje se vrši na ukupnu luminoznost sistema. Ako nema trećeg svetla, onda će biti

Jednačina 2.17

$$L_1 = \frac{l_1}{l_1 + l_2}, \;\; L_2 = \frac{l_2}{l_1 + l_2}.$$

Iako model ne uključuje efekte koji potiču od prisustva druge komponente, efekat refleksije je dovoljno značajan da se vrši popravka luminoznosti kojom se obračunava ovaj efekat. Ako se pretpostavi da je sekundarna komponenta tačkasti izvor zračenja, njen doprinos luminoznosti primarne komponente je

Jednačina 2.18

$$L_1' = L_1 + \Delta L_1, \;\; \Delta L_1 = S_1 f(\phi), \;\; S_1 = 0.4 A_1 L_2 r_1^2,$$

gde je A_1 albedo primarne komponente, a

Jednačina 2.19

$$f(\phi) = 0.2 + 0.4\cos\phi + 0.2\cos^2\phi\,, \;\; cos\;\phi = \sin i \cos\theta$$

tzv. bolometrijski fazni zakon. θ je geometrijska faza. Slično se dobija popravka za doprinos primarne komponente luminoznosti sekundarne komponente.

2.4. Wilson-Devinney model

Wilson-Devinney (WD) je jedan od prvih fizičkih modela zasnovanih na Roche geometriji i u velikoj meri je oblikovao moderni pristup modeliranju eklipsnih dvojnih sistema. Program ima dugu istoriju – prva verzija je objavljena početkom sedamdesetih godina prošlog veka (Wilson & Devinney, 1971) i do danas se aktivno razvija. Poslednja verzija donosi podršku za dodatne fotometrijske filtere, novi algoritam za modeliranje zvezdanih pega i računanje izlaznog fluksa u apsolutnim fizičkim jedinicama (Wilson & Van Hamme, 2013). Osim zvanične verzije, u upotrebi su u mnoge modifikovane verzije, kao što je, na primer, verzija koja uključuje akrecioni disk oko primarne komponente (Zola, 1991) ili verzija koja uključuje efekte pritiska zračenja kod toplih zvezda (Drechsel, et al., 1995).

Diskretizacija površine zvezde se vrši u sfernom koordinatnom sistemu tako da površina svih elementarnih ćelija bude ista. Ako je N zadati broj prstenova po latitudi, koordinate centara elementarnih ćelija će biti

Jednačina 2.20

$$\theta_i = \frac{\pi}{2}\frac{i-0.5}{N}\Delta\theta,\ \ \Delta\theta = \frac{\pi}{2N},\ \ i = 1, \dots, N,$$
$$\varphi_i = \pi\frac{k-0.5}{M(\theta_i)}\Delta\varphi,\ \ \Delta\varphi = \frac{\pi}{M(\theta_i)},\ \ k = 1, \dots, M(\theta_i),$$

gde je

Jednačina 2.21

$$M(\theta_i) = 1 + 1.3N \sin\left(\frac{\pi}{2}\ \frac{i-0.5}{N}\right).$$

Da bi se odredilo koje se elementarne ćelije vide u datoj fazi, program prvo odredi položaj horizonta bliže zvezde. Ako se posmatra kosinus ugla između pravca vizure i normale na elementarnu ćeliju, očigledno je da fluksu doprinose samo one ćelije kod kojih je $\cos\gamma = \overrightarrow{r_{obs}}\ \vec{n} < 0$ (u WD modelu, pozitivni smer normale je ka unutrašnjosti zvezde, dok vektor $\overrightarrow{r_{obs}}$ gleda ka posmatraču). Tačke kod kojih dolazi do promene

znaka ovog kosinusa se nalaze na horizontu i kroz njih se provlači analitička funkcija koja je predstavljena Fourier-ovim redom. Koeficijenti ovog reda se određuju metodom najmanjih kvadrata i onda se za svaku elementarnu ćeliju na daljoj komponenti proveri da li je unutar krive koja opisuje horizont.

Doprinos fluksu svake ćelije površine $d\sigma$ se onda računa kao

Jednačina 2.22

$$dl_j(r_s, \cos\gamma, g_l, T_l, \lambda) = G_j D_j R_j I_j \cos\gamma \, d\sigma,$$

gde su G_j, D_j i R_j faktori koji obračunavaju, redom, efekte gravitacionog potamnjenja, potamnjenja ka rubu i refleksije, a I_j je centralni specifični intenzitet koji se računa za lokalne vrednosti temperature T_l i gravitacionog ubrzanja g_l. Ukupni fluks se dobija kao zbir po svim vidljivim ćelijama.

Program podržava nekoliko modova rada koji su optimizovani za konkretne konfiguracije dvojnih sistema:

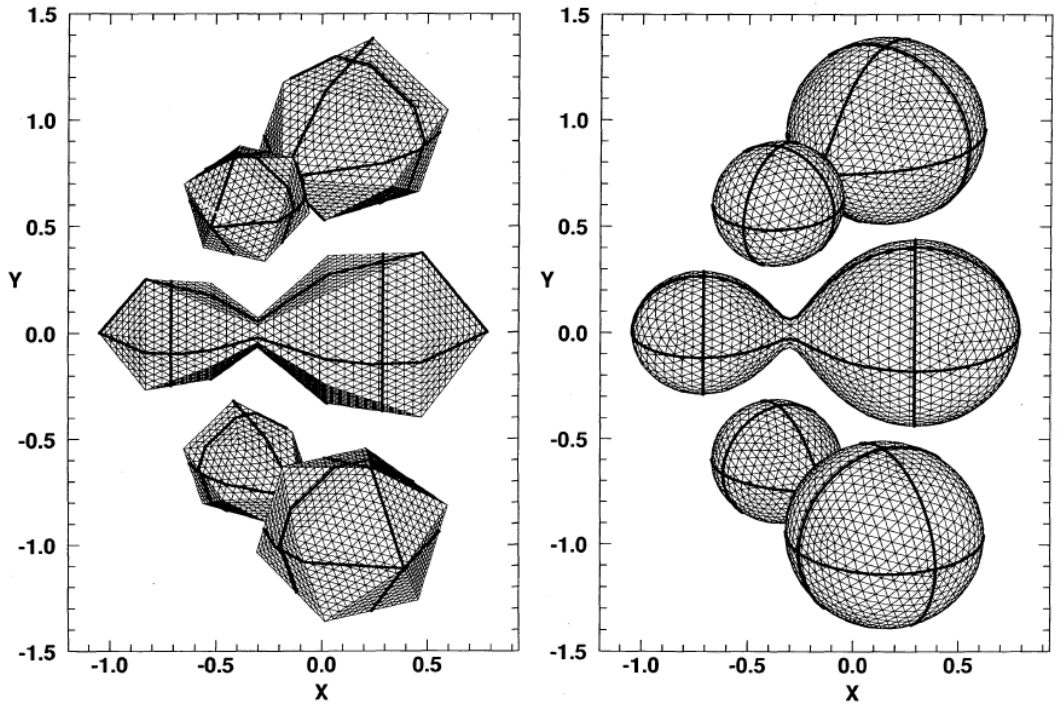
- Mod -1 služi za modeliranje dvojnih sistema koji zrače u X domenu kod kojih je moguće izmeriti dužinu pomračenja kompaktnog objekta.
- Mod 0 je osnovni mod, gde se svi parametri tretiraju kao slobodni parametri.
- Mod 1 je za modeliranje sistema u overkontatku, kao što su W UMa sistemi.
- Mod 2 služi za modeliranje dobro odvojenih sistema.
- Mod 3 liči na mod 1 (sistemi u overkontaktu), osim što ne pretpostavlja da su komponente u termalnom kontaktu.
- Modovi 4 i 5 pokrivaju slučaj poluodvojenih sistema (sistemi kod kojih jedna komponenta ispunjava svoj kritični Roche oval, a druga, ne). Ako primarna komponenta ispunjava svoj oval, onda se koristi mod 4, u suprotnom mod 5.
- Mod 6 služi za modeliranje sistema kod kojih obe komponente tačno ispunjavaju svoj kritični Roche oval, a barem jedna od njih rotira asinhrono (double contact binaries).

2.5. GDDSYN

GDDSYN je evolucija programa GENSYN koji su razvili Mochnacki i Doughty (1972). GENSYN je zasnovan na Roche geometriji i osim krivih sjaja, računa i profile spektralnih linija. Uglavnom je korišćen za modeliranje W UMa sistema i neka geometrijska rešenja su prilagođena modeliranju sistema ovog tipa, kao što je korišćenje Roche potencijala u cilindričnom koordinatnom sistemu prilikom računanja oblika zvezde (u većini modela se Roche potencijal izražava u sfernom koordinatnom sistemu). Program odlikuje velika brzina izvršavanja i numerička stabilnost.

GDDSYN je doneo dodatna poboljšanja, pre svega na polju geometrije. Detaljan opis modela se može naći u članku Hendry-a i Mochnacki-ja (1992). Značaj ovog modela u kontekstu ove disertacije je što je to prvi objavljeni model koji koristi geodezijsku mrežu, kao i prvi model koji koristi trougaone elementarne ćelije (slika 2.3). O prednostima ovog načina predstavljanja geometrije dvojnog sistema će biti reči u sekciji 3.1.1. Interesantan je i metod za detekciju vidljivosti koji se koristi u ovom programu. Posle određivanja oblika i položaja komponenata u datoj fazi, geodezijska mreža se projektuje na ravan nebeske sfere. Zvezda koja pomračuje, odnosno njen disk, se aproksimiraju konveksnim poligonom. Da bi se utvrdilo koji se delovi pomračene komponente vide, dovoljno je onda proći kroz listu njenih trouglova i proveriti da li se oni seku sa poligonom određenim u prvom koraku. Ukoliko presek postoji (a trougao nije unutar poligona), moguće je tačno odrediti kolika je površina preklapanja, odnosno odrediti parcijalnu vidljivost svakog trougla.

Ova informacija se koristi kao težina koja se kasnije pripisuje trouglu pri računanju izlaznog fluksa. To znači da se i sa malim brojem elementarnih ćelija može postići velika numerička tačnost. Autori tvrde da je njihov program u proseku oko sedam puta brži od Wilson-Devinney programa, kao i da je tačniji za ekvivalentni broj elementarnih ćelija. Program modelira i pege na obe komponente i koristi princip maksimalne entropije da bi odredio optimalni raspored i veličinu pega.



Slika 2.3 Geodezijska mreža iz programa GDDSYN. Preuzeto iz Hendry & Mochnacki (1992).

2.6. Đuraševićev model

Đuraševićev model se već decenijama uspešno primenjuje u analizi aktivnih tesno dvojnih sistema. Glavna karakteristika modela je mogućnost modeliranja akrecionih diskova. Opis prve verzije modela je dat u radu Đuraševića (1991), dok se bitnija proširenja mogu naći u radovima Đurašević et al. (1998) i Đurašević et al. (2008).

Geometrija komponenata sistema se opisuje Roche-ovim ekvipotancijalnim površima, a veličina se zadaje tzv. filling factor-om, koji se definiše kao odnos polarnog radijusa zvezde r_p i polarnog radijusa kritičnog sinhronog Roche ovala r_c.

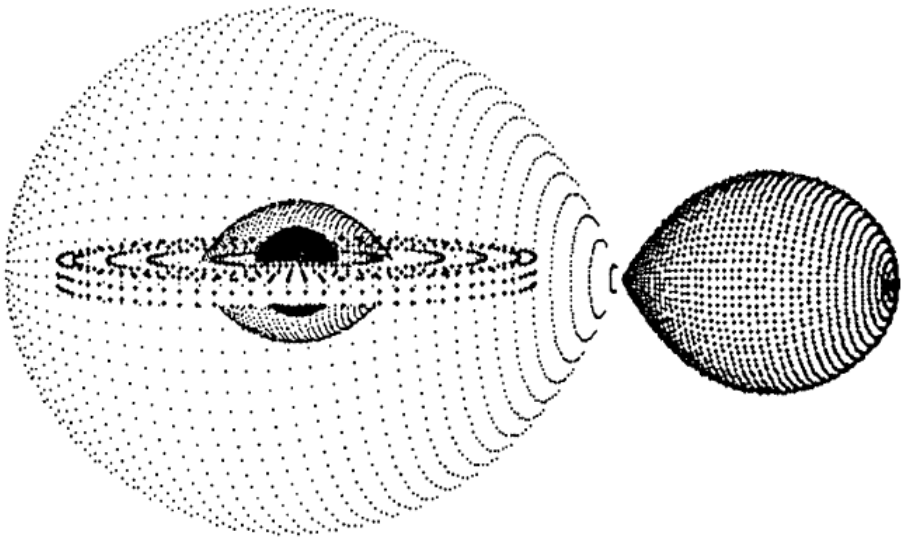
Jednačina 2.23

$$F_i = \frac{r_p}{r_c}.$$

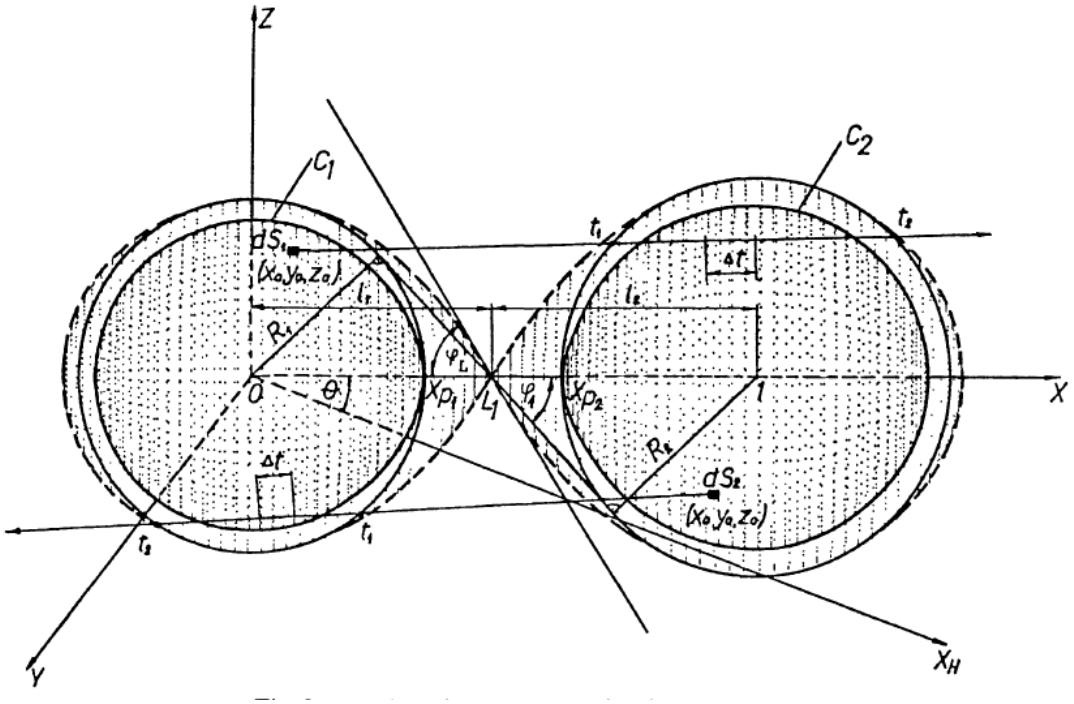
Diskretizacija površine zvezde se vrši pomoću uniformne sferne mreže (ravnomerna raspodela tačaka duž sfernih koordinata θ i φ). Ona je prikazana na slici 2.4. Detekcija vidljivosti se zasniva na efikasnom *ray tracing* algoritmu (slika 2.5). U svakoj fazi se odredi radijus-vektor koji opisije poziciju posmatrača u odnosu na nepokretni

koordinatni sistem vezan za centar primarne komponente (x-osa leži na pravoj koja prolazi kroz centar sekundarne komponente, y-osa je u orbitalnoj ravni, a z-osa je normalna na orbitalnu ravan). Kroz svaku tačku na pomračenoj komponenti se provuče prava paralelna vektoru posmatrača i proverava se da li ona seče Roche površ koja opisuje površinu druge komponente. Đurašević je pokazao da se ova provera može dramatično ubrzati ako se umesto traženja tačke preseka sa Roche površi duž prave sa zadatim korakom, prvo ispituje presek sa sferom opisanom oko komponente koja pomračuje. Iz oblika potencijala je očigledno da će minimalna opisana sfera imati radijus koji je jednak prednjem radijusu zvezde (na x-osi, ka Lagrange-ovoj L1 tački). Ukoliko se ova sfera i prava seku, onda postoji mogućnost da je data elementarna ćelija pomračena. Ako su P i Q tačke preseka, u monografiji Đurašević (1991) se pokazuje da je dovoljno proveriti vrednost potencijala u tri ekvidistantne tačke između P i $\frac{(P+Q)}{2}$. Rezultat je izuzetno brz algoritam koji se odlikuje velikom numeričkom tačnošću. Prisustvo akrecionog diska malo komplikuje izraze za proveru vidljivosti elementarne ćelije (usled ekraniranja primarne komponente diskom), ali se ne menja suština algoritma.

Raspodela temperature na površini komponenata je određena gravitacionim potamnjenjem opisanim von Zeipel-ovim zakonom, efektom refleksije i eventualnim prisustvom zvezdanih pega. Podržana su četiri različita zakona za raspodelu temperature duž radijusa diska, kao i modeliranje aktivnih regiona na rubu diska. Izlazni fluks se računa pod pretpostavkom da elementarne ćelije zrače kao apsolutno crno telo, a za korekciju usled potamnjenja ka rubovima se koriste Claret-ove tablice (Claret & Bloemen, 2011).



Slika 2.4 Geometrija sistema. Preuzeto iz Đurašević (1992b).

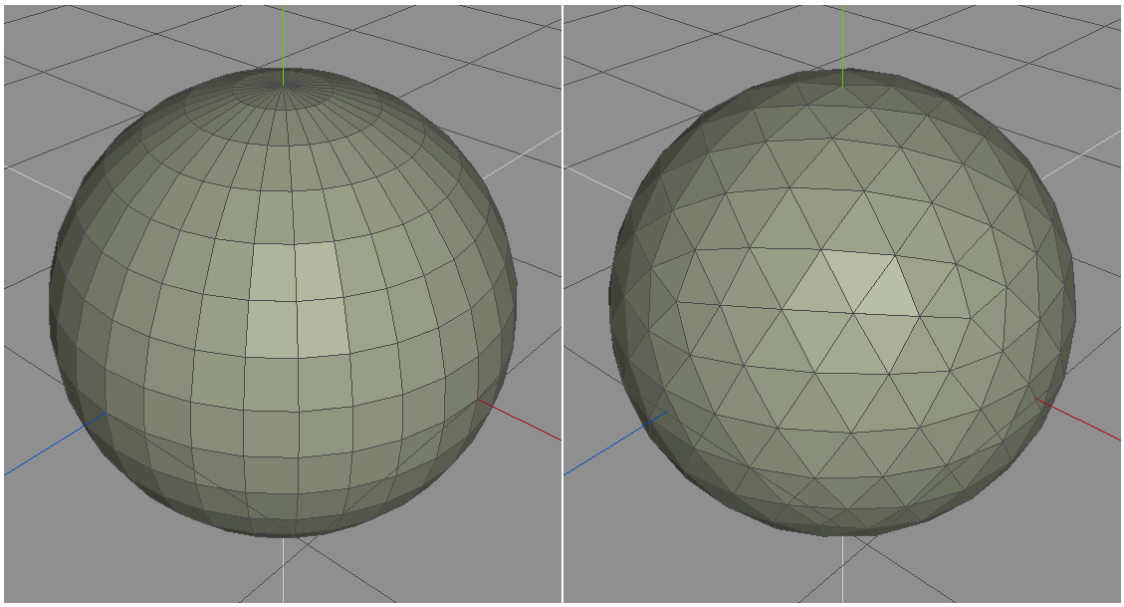


Slika 2.5 Ispitivanje vidljivosti elementarne ćelije. Preuzeto iz Đurašević (1992a).

3. Geometrijska reprezentacija sistema

Komponente dvojnog sistema se u *Infinity*-ju predstavljaju poligonom mrežom. Poligona mreža je skup povezanih poligona u trodimenzionom prostoru koji opisuje neku površ. Iako nije neophodno da svi poligoni budu iste vrste, prilikom implementacije sam se odlučio za mrežu koja sadrži isključivo trouglove. Osim što pojednostavljuje strukture podataka koje reprezentuju mrežu u programu, ovaj izbor znatno olakšava implementaciju algoritma za detekciju vidljivosti, o čemu će detaljnije biti reči u poglavlju 4.

U *Infinity*-ju je moguće modelirati dva tipa objekata – zvezde i akrecione diskove. Različita priroda ovih objekata zahteva i različit pristup u njihovom geometrijskom predstavljanju. Za zvezde se koristi geodezijska mreža, dok se za akrecioni disk koristi osno-simetrična rotaciona mreža (*lathe mesh*). Dobijene poligone mreže se onda razmeštaju na položaje koji odgovaraju njihovim pozicijama na orbiti, koja može biti kružna ili eliptična. Mreže se zatim prosleđuju modulu za detekciju vidljivosti.



Slika 3.1 Sferna (levo) i geodezijska mreža (desno).

3.1. Zvezde

3.1.1. Geodezijska mreža

Geodezijska mreža ima nekoliko prednosti u odnosu na sferne mreže, koje se koriste u velikoj većini modela. Na slici 3.1 s leve strane je prikazana uniformna sferna mreža (elementarne površine se dobijaju tako što se tačke ravnomerno rasporede duž sfernih koordinata θ i φ), a s desne, geodezijska. Ona se sastoji samo od trouglova, što olakšava detekciju vidljivosti. Elementarne ćelije su približno iste veličine (Hendry & Mochnacki, 1992) i nema izražene ose simetrije, čime se izbegavaju problemi sa numeričkom tačnošću koji se javljaju na polovima uniformne sferne mreže. Ova osobina geodezijskih mreža isto tako pogoduje modeliranju neradijalnih pulsacija (Latković, 2013, u pripremi).

Pri konstrukciji geodezijske mreže se polazi od bilo kog poliedra. Svaki trougao početnog poliedra se podeli na zadati broj manjih trouglova i onda se novodobijena temena projektuju na površ koja opisuje oblik zvezde. Najravnomernija podela se dobija ako se za polaznu mrežu uzme ikosaedar, mada su i ostali pravilni poliedri podržani (tetraedar, heksaedar, oktaedar i dodekaedar). Osnovni podaci (broj temena, ivica, stranica i trouglova) za pravilne poliedre su dati u tabeli 1 (Eberly, 2008).

Tabela 1 Osnovni podaci za pravilne poliedre

	Tetraedar	Heksaedar	Oktaedar	Dodekaedar	Ikosaedar
Temena	4	8	6	20	12
Ivica	6	12	12	30	30
Stranica	4	6	8	12	20
Trouglova	4	12	8	36	20

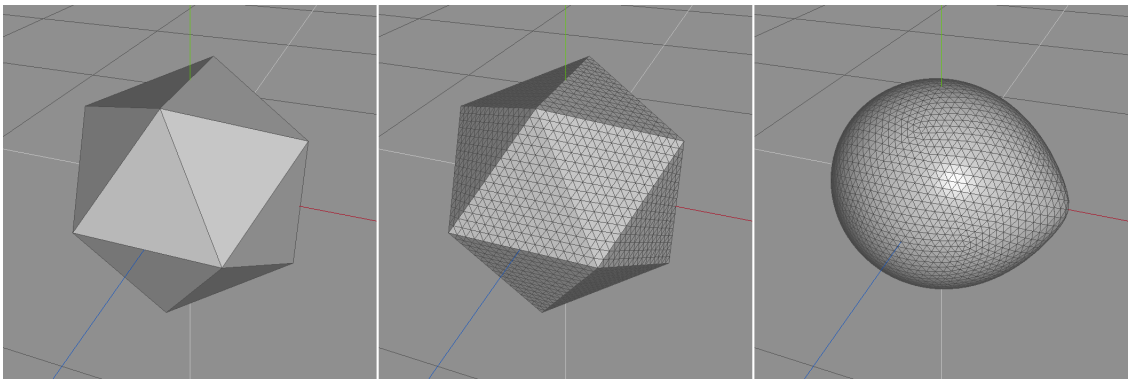
3.1.2. Rekurzivna i frekventna podela trouglova

Algoritam za podelu trouglova (*subdivision*) uzima kao ulaz poligonu mrežu i stepen podele N (celobrojni parametar koji određuje rezultujući broj trouglova). U *Infinity*-ju su implementirana dva algoritma za podelu – rekurzivna i frekventna podela.

Rekurzivna podela je prikazana na slici 3.3. U prvoj iteraciji delimo trougao ABC tako što za svaku ivicu nađemo središnju tačku. Ako je teme A zadato koordinatama (X_0, Y_0, Z_0), teme B sa (X_1, Y_1, Z_1) onda će koordinate središnje tačke P na ivici AB biti

Jednačina 3.1

$$\left(\frac{X_0 + X_1}{2}, \frac{Y_0 + Y_1}{2}, \frac{Z_0 + Z_1}{2}\right).$$



Slika 3.2 Konstrukcija geodezijske mreže. Polazi se od ikosaedra (levo) i frekventnom podelom se svaka stranica izdeli na sitnije trouglove (sredina). Svako teme se projektuje na površ kojom se opisuje površina zvezde. Ovde je prikazana Roche površ (desno).

Isto tako odredimo i koordinate tačaka R i Q na ivicama BC i AC. Trougao ABC zamenimo trouglovima APR, BQP, CRQ i PQR. U sledećoj iteraciji na isti način podelimo svaki od ova četiri trougla. Opisani postupak se ponavlja N puta. Ako je F broj stranica početnog poliedra, posle N podela rezultujući broj trouglova T je dat izrazom

Jednačina 3.2

$$T = F * 4^N.$$

Za ikosaedar je $F = 20$ (videti tabelu 1).

Ekvivalentan izraz za broj trouglova kod frekventne podele je

Jednačina 3.3

$$T = F * N^2.$$

Svaka ivica početnog trougla ABC se podeli na N jednakih segmenata. Koordinate novodobijenih tačaka se najlakše mogu izraziti ako se uvede baricentrični koordinatni sistem. Ako je teme A zadato koordinatama (X_0, Y_0, Z_0), teme B sa (X_1, Y_1, Z_1), onda će koordinata bilo koje tačke na pravoj određenoj tim tačkama biti

Jednačina 3.4

$$P = (1 - t) * A + t * B.$$

t je baricentrična koordinata tačke P. Tački A odgovara koordinata $t = 0$, a tački B koordinata $t = 1$. Koordinate novih tačaka na stranici AB su onda

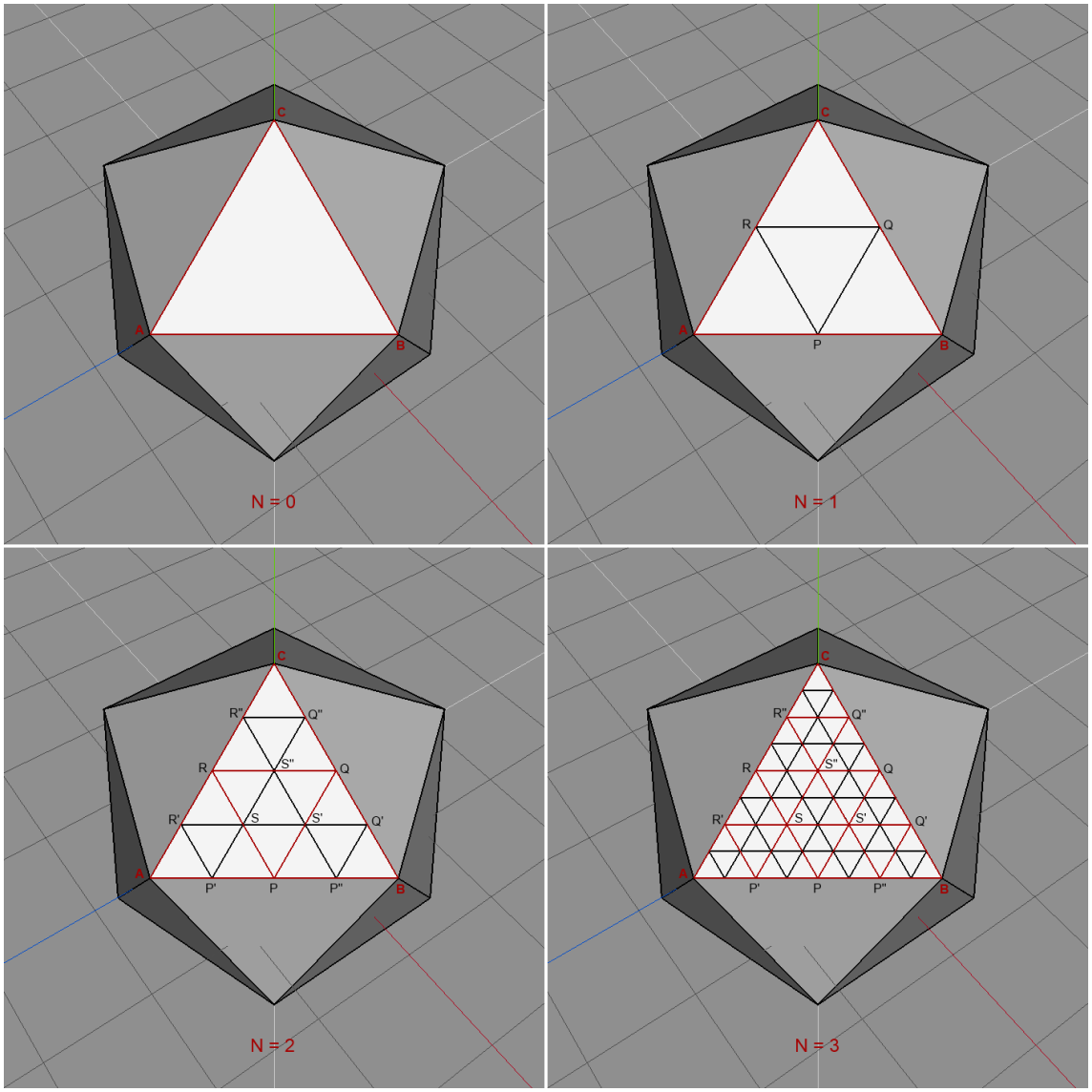
Jednačina 3.5

$$P_j = (1 - t_j) * A + t_j * B,$$

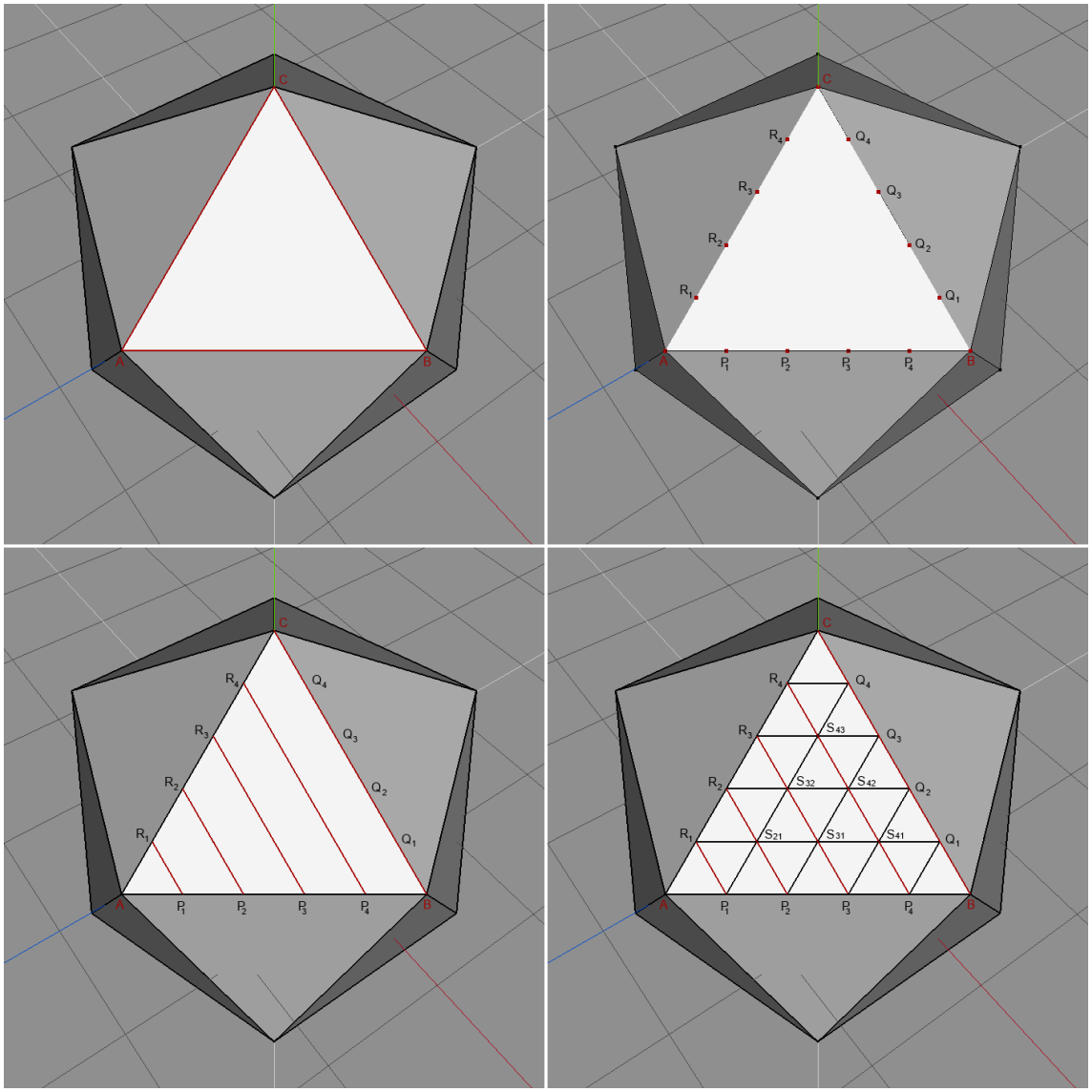
gde je $t_j = \frac{j}{N}$, $j = 1, \dots, N - 1$. Na isti način odredimo i koordinate tačaka R_j na stanici AC. U sledećem koraku konstruišemo prave $P_j R_j$. Svaku od ovih pravih podelimo na j jednakih segmenata umetanjem $j - 1$ tačke. Neka je S_{jk} k-ta tačka na pravoj $P_j R_j$ ($k = 1, \dots, j - 1$). Od tako dobijenih temena formiramo trouglove AP_1R_1, $P_1P_2S_{21}$, $P_1S_{21}R_1$, $R_2R_1S_{21}$,... (videti sliku 3.4).

Za osnovnu konstrukciju geodezijske mreže sam odabrao frekventnu podelu, dok za adaptivnu podelu (odeljak 4.2) koristim rekurzivnu podelu.

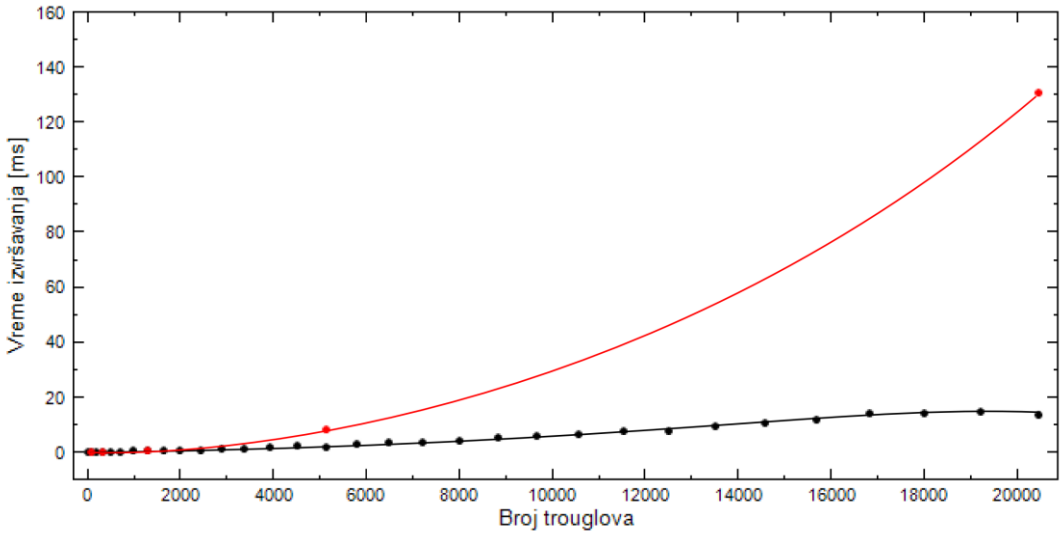
Frekventna podela je nešto komplikovanija za implementaciju, ali je mnogo brža za velike vrednosti N i daje mnogo finiju kontrolu nad rezultujućim brojem trouglova. Sa druge strane, rekurzivna podela je pogodnija kada je potrebno podeliti samo neke trouglove, a ne celu mrežu. Poređenje brzine ove dve metode je prikazano na slici 3.5. Za male vrednosti N nema velike razlike između dva algoritma. Za potrebe adaptivne podele se retko uzima $N > 3$, tako da je izbor rekurzivne podele opravdan.



Slika 3.3 Rekurzivna podela za nivo podele N=3.



Slika 3.4 Frekventna podela za N=5.



Slika 3.5 Poređenje vremena izvršavanja dva algoritma za podelu trouglova. Crvene tačke odgovaraju rekurzivnoj podeli, a crne frekventnoj. Razlika dramatično raste sa povećanjem broja trouglova.

3.1.3. Sferne zvezde

Sledeći korak je projektovanje temena mreže na površ koja opisuje površinu zvezde. Ona se opisuje potencijalom. U *Infinity*-ju su implementirani sferni i Roche potencijal. Iako Roche potencijal mnogo tačnije opisuje oblik komponenata, postoje slučajevi u kojima je odstupanje zvezde od sfernog oblika zanemarljivo, pa je moguće dobiti izvesno ubrzanje izvršavanja programa korišćenjem jednostavnijeg potencijala. Primeri su modeliranje tranzita egzoplaneta ili dobro razdvojenih sistema.

Neka je r_1 radijus zvezde u jedinicima velike poluose orbite. Za svako teme izračunamo sferne koordinate θ i φ i postavimo treću koordinatu na $r = r_1$. Ako je koordinata temena zadata sa (x_0, y_0, z_0) onda će nove koordinate (x, y, z) tačke na sferi radijusa r_1 biti:

Jednačina 3.6

$$\theta = arccos\left(\frac{z_0}{\sqrt{{x_0}^2 + {y_0}^2 + {z_0}^2}}\right),$$

$$\varphi = arctan\left(\frac{y_0}{x_0}\right).$$

Jednačina 3.7

$$x = r_1 \sin\theta \cos\varphi,$$

$$y = r_1 \sin\theta \sin\varphi,$$

$$z = r_1 \cos\theta.$$

3.1.4. Roche model

U opštem slučaju površinu zvezde opisujemo Roche-ovim modelom. Teorija zvezdane strukture pokazuje da zvezde imaju visok stepen centralne kondenzacije, te da se mogu aproksimirati materijalnim tačkama, što znatno pojednostavljuje izraze za gravitacioni potencijal. Smatra se da je period neradijalnih oscilacija zanemarljiv u odnosu na orbitalni period, tako da oblik zvezde zavisi samo od trenutne jačine polja i da je vremenska skala ovih oscilacija reda veličine hidrostatičke skale vremena. Pretpostavlja se da površinama konstantnog potencijala odgovaraju površine konstantne gustine. Zanemaruju se efekti diferencijalne rotacije, odnosno smatra se da zvezde rotiraju kao kruto telo (Kallrath & Milone, 2009).

Definišemo pokretni desni koordinatni sistem $\mathcal{C}_1$ sa koordinatnim početkom u centru primarne komponente tako da x-osa uvek prolazi kroz centar sekundarne komponente. y-osa je u orbitalnoj ravni u smeru revolucije sistema. Prvo razmatramo slučaj kružne orbite (Đurašević, 1991).

Na tačku (x, y, z) koja se nalazi na površini zvezde deluje superpozicija gravitacione i centrifugalne sile čiji je potencijal dat izrazom:

Jednačina 3.8

$$\psi = \gamma \frac{m_1}{r_1} + \gamma \frac{m_2}{r_2} + \frac{1}{2}\omega^2(x^2 + y^2) - \frac{m_2}{m_1 + m_2}\omega_k^2 x,$$

gde je $r_1 = \sqrt{x^2 + y^2 + z^2}$ i $r_2 = \sqrt{(D - x)^2 + y^2 + z^2}$. Kvadrat Keplerove ugaone brzine revolucije sistema dat je izrazom:

Jednačina 3.9

$$\omega_k^2 = \gamma \frac{m_1 + m_2}{D^3}.$$

Ovde su m_1 i m_2 mase komponenata, D rastojanje između njihovih centara, a γ gravitaciona konstanta. Ugaona brzina zvezde se može izraziti preko Keplerove ugaone brzine revolucije sistema kao

Jednačina 3.10

$$\omega = f_1 \omega_k,$$

gde je f_1 parametar asinhronosti. Iz praktičnih razloga se obično uzima da je međusobno rastojanje između komponenata jedinično i uvode se sledeće smene

Jednačina 3.11

$$q = \frac{m_2}{m_1},$$

Jednačina 3.12

$$C_1 = \frac{\psi D}{\gamma m_1},$$

Jednačina 3.13

$$C_1 = \frac{1}{r_1} + q\left(\frac{1}{r_2} - x\right) + \frac{q+1}{2}(x^2 + y^2)f_1^2.$$

Potencijal C_1 je bezdimenziona veličina koja se naziva modifikovanim Kopalovim potencijalom (Kopal, 1959). Za zadati odnos masa q i parametar asinhronosti f_1 on jednoznačno određuje površinu zvezde.

Kao što je već napomenuto u odeljku 3.1.3, svako teme geodezijske mreže projektujemo na površ koja opisuje površinu zvezde tako što je „transliramo“ duž radijalnog pravca. Zato je zgodno potencijal izraziti u sfernom koordinatnom sistemu zadatom jednačinama 3.7, uz sledeće smene:

Jednačina 3.14

$$\begin{aligned} x &= r_1 \sin\theta \cos\varphi = r_1\lambda, \\ y &= r_1 \sin\theta \sin\varphi = r_1\mu, \\ z &= r_1 \cos\theta = r_1\nu. \end{aligned}$$

Jednačina 3.15

$$C_1 = \frac{1}{r_1} + q\left(\frac{1}{\sqrt{1 + r^2 - 2r_1\lambda}} - r_1\lambda\right) + \frac{q+1}{2}r_1^2(1 - \nu^2)f_1^2.$$

U slučaju ekscentrične orbite, rastojanje između komponenata zavisi od faze, što znači da sila koja deluje na probno telo nije konzervativna, odnosno da se za nju ne može definisati skalarni potencijal. Wilson (1979) je pokazao da je ipak moguće koristiti efektivni potencijal ukoliko je vreme za koje se uspostavlja ravnotežno stanje mnogo kraće od orbitalnog perioda. Efektivni potencijal je dat sledećim izrazom:

Jednačina 3.16

$$C_1 = \frac{1}{r_1} + q\left(\frac{1}{\sqrt{d^2 + r_1^2 - 2r_1 d\lambda}} - \frac{r_1\lambda}{d^2}\right) + \frac{q+1}{2} r_1^2 (1-\nu^2) f_1^2,$$

gde je d rastojanje između komponenata u jedinicama velike poluose orbite. Ono se dobija iz rešenja Keplerove jednačine (detaljnije u odeljku 3.3.3). Pošto vrednost potencijala zavisi od faze, oblik i veličina komponenata takođe zavise od faze. U prvoj aproksimaciji se može očekivati da će zapremina komponenata ostati približno konstantna (Wilson, 1979), pa se iz uslova $V = const$ određuje vrednost potencijala u svakoj fazi. Za referentu vrednost se, po definiciji, uzima zapremina zvezde u periastronu V_p.

Neka je r_{p1} polarni radijus zvezde. Na polu će biti $\theta = 0°$ i $\varphi = 0°$, pa je potencijal dat izrazom:

Jednačina 3.17

$$C_1 = \frac{1}{r_{p1}} + q\frac{1}{\sqrt{d^2 + r_{p1}^2}}.$$

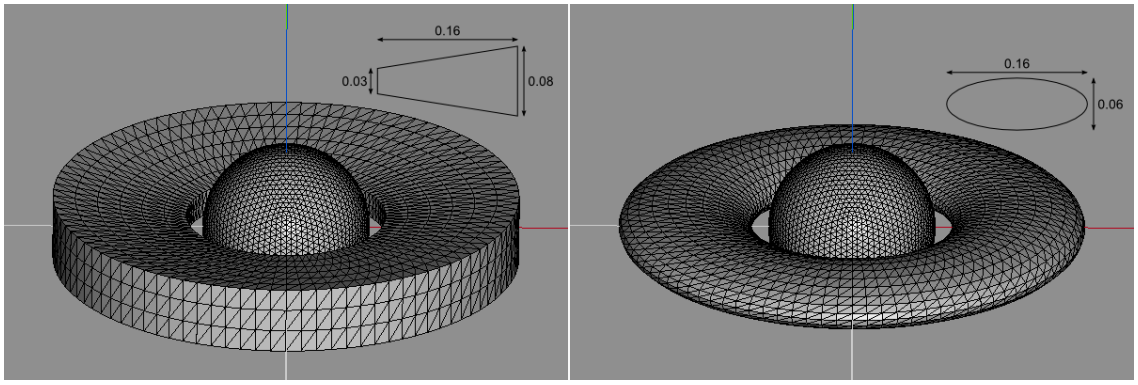
Radijus vektor svakog temena poligone mreže u sfernom koordinatnom sistemu se određuje iz jednačine ekvipotencijala $C = C_1 = const$ Brentovim algoritmom za traženje nule funkcije (Press, et al., 2007), pri čemu je C_1 definisano relacijom 3.16.

Za konstrukciju mreže koja odgovara sekundarnoj komponenti ponavljamo opisani postupak, uz zamenu mesta komponenata. Definišemo na isti način koordinatni sistem $\mathcal{C}_2$ koji je vezan za centar sekundarne komponente. U izrazu za potencijal figurišu odnos masa $q' = {}^1/_q = {}^{m_1}/_{m_2}$ i parametar asinhronosti sekundarne komponente f_2.

3.2. Akrecioni disk

Geometrija akrecionog diska se aproksimira osno-simetričnom površi koja se nalazi u orbitalnoj ravni. Mreža koja odgovara toj površi se dobija rotacijom poprečnog preseka diska oko ose koja je normalna na orbitalnu ravan. Zadaje se nizom tačaka koji opisuju presek diska i ravni normalne na orbitalnu ravan. Ukoliko je taj presek trapez onda dobijamo konusni disk, a ako je u pitanju elipsa onda dobijamo toroidalni disk (slika 3.6). Jednostavnosti radi, pretpostavlja se da se disk uvek nalazi oko primarne

komponente, što znači da se koordinate temena izražavaju u koordinatnom sistemu $\mathcal{C}_1$ vezanom za primarnu komponentu.



Slika 3.6 Konusni i toroidalni disk.

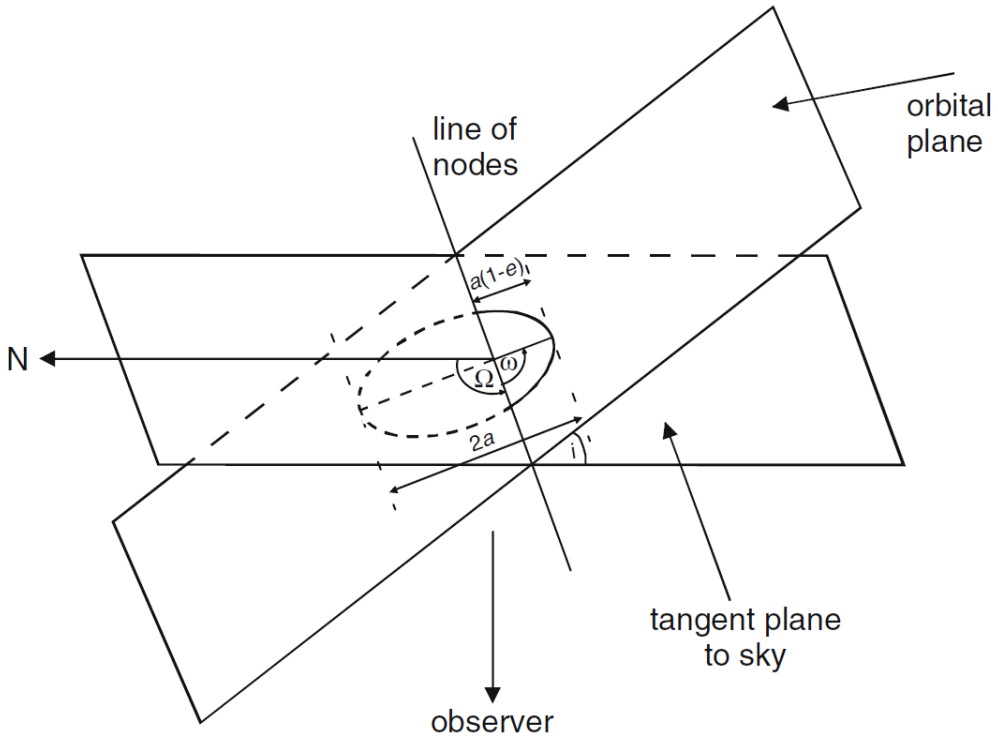
3.3. Orbita

Kada smo odredili oblik komponenata, ostaje još samo da se odredi njihov položaj za svaku pojedinačnu fazu Φ. Faza Φ se naziva još i fotometrijskom fazom i uzima vrednosti iz intervala $[0, 1]$. Po definiciji fazi $\Phi = 0$ odgovara trenutak kada sekundarna komponenta pomračuje primarnu. Zbog pretpostavke da zvezde imaju visoki stepen centralne kondenzacije, problem njihovog kretanja oko zajedničkog centra mase se svodi na klasičan problem dva tela. To kretanje se opisuje u odnosu na centar primarne komponente, tj. opisuje se kretanje sekundarne komponente oko primarne po relativnoj orbiti sa velikom poluosom $a = a_1 + a_2$, periodom $P = P_1 = P_2$ i ekscentričnošću $e = e_1 = e_2$. Orijentacija ove orbite u prostoru se zadaje sa tri ugla (slika 3.7):

- Inklinacija i opisuje nagib orbitalne ravni u odnosu na ravan koja je tangencijalna na nebesku sferu.
- Argument periastrona ω je ugao u orbitalnoj ravni koji zaklapaju pravci ka periastronu i ka uzlaznom čvoru orbite.
- Longituda uzlaznog čvora Ω je ugao u ravni tangencijalnoj na nebesku sferu od referentnog pravca do uzlaznog čvora putanje. Kako referentni pravac možemo proizvoljno izabrati , obično se uzima da je $\Omega = 0$.

Pozicija zvezde na orbiti u fazi Φ se opisuje geometrijskom fazom θ i trenutnom separacijom d. Geometrijska faza je uglovna mera koja odgovara fotometrijskoj fazi Φ i uzima vrednosti između $[0, 2\pi]$. Za kružne orbite je veza između ove dve veličine

jednostavna ($\theta = 2\pi\Phi$), dok je za eliptične orbite izvedena u odeljku 3.3.3. Ova dva slučaja, kružne i eliptične orbite, se razmatraju posebno zbog razlika u praktičnoj implementaciji i efekata koje imaju na krivu sjaja.



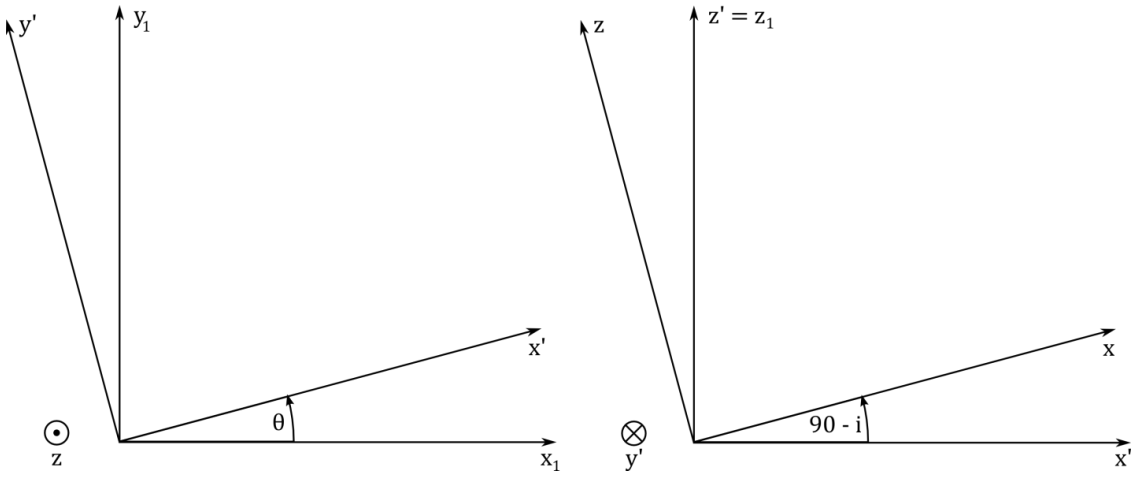
Slika 3.7 Orbitalni elementi. Preuzeto iz Kallrath & Milone (2009).

3.3.1. Koordinatni sistem

Položaj komponenata u svakoj fazi se izražava u koordinatnom sistemu $\mathcal{P}$ vezanom za nebesku sferu. U pitanju je desni koordinatni sistem sa koordinatnim početkom u centru primarne komponente. x-osa je usmerena ka posmatraču, dok je yOz ravan tangencijalna na nebesku sferu. U odeljku 2.1 su definisani lokalni koordinatni sistemi $\mathcal{C}_1$ i $\mathcal{C}_2$ u kojima se zadaju položaji elementarnih ćelija u odnosu na centar svake komponente. Veza između koordinatnih sistema $\mathcal{C}_1$ i $\mathcal{P}$ je prikazana na slici 3.8. Oni se poklapaju u fazi $\Phi = 0$ ukoliko je inklinacija orbite $i = 90°$. Ova transformacija se dobija tako što se sistem $\mathcal{C}_1$ rotira oko z-ose za geometrijsku fazu θ, a onda se rotira za ugao $90° - i$ oko novonastale ose y', gde je i inklinacija orbite. Da bi se sistem $\mathcal{C}_2$ u kome se izražavaju koordinate elementarnih ćelija sekundarne komponente preveo u $\mathcal{C}_1$, potrebno je rotirati $\mathcal{C}_2$ oko z-ose za π i onda ga translirati za rastojanje između komponenata. Jednostavnosti radi, pretpostavljamo da se akrecioni disk (ukoliko postoji

u sistemu) uvek nalazi oko primarne komponente, tj. da se njegov lokalni koordinatni sistem poklapa sa $\mathcal{C}_1$.

Ove transformacije se najprirodnije izražavaju preko transformacionih matrica. U opštem slucaju je to 4x4 matrica (jednačina 3.18). Transformaciona matrica za translaciju duž x-ose za t_{14}, duž y-ose za t_{24} i duž z-ose za t_{34} je data jednačinom 3.19, dok su matrice za rotaciju date jednačinama 3.20 - 3.22 (Weisstein, 2013).



Slika 3.8 Transformacija iz lokalnog koordinatnog sistema $\mathcal{C}_1$ u sistem $\mathcal{P}$.

Jednačina 3.18

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Jednačina 3.19

$$T(t_{14}, t_{24}, t_{34}) = \begin{bmatrix} 1 & 0 & 0 & t_{14} \\ 0 & 1 & 0 & t_{24} \\ 0 & 0 & 1 & t_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Jednačina 3.20

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Jednačina 3.21

$$R_y(\varphi) = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\varphi & 0 & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

Jednačina 3.22

$$R_z(\varphi) = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 & 0 \\ -\sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Ako je (x_1, y_1, z_1) koordinata temena u sistemu $\mathcal{C}_1$, onda će koordinate te tačku u sistemu $\mathcal{P}$ biti

Jednačina 3.23

$$(\mathcal{C}_1 \to \mathcal{P}) \quad \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} R_z(\theta) R_{y'}\left(\frac{\pi}{2} - i\right).$$

Isto tako, ako je (x_2, y_2, z_2) koordinata temena u sistemu $\mathcal{C}_2$, onda će koordinate te tačku u sistemu $\mathcal{P}$ biti:

$$(\mathcal{C}_2 \to \mathcal{P}) \quad \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} R_z(\pi) T(d, 0, 0) R_z(\theta) R_{y'}\left(\frac{\pi}{2} - i\right).$$

Ostaje još samo da se odredi veza između geometrijske i fotometrijske faze. Posebno razmatramo slučajeve kružne i eliptične orbite.

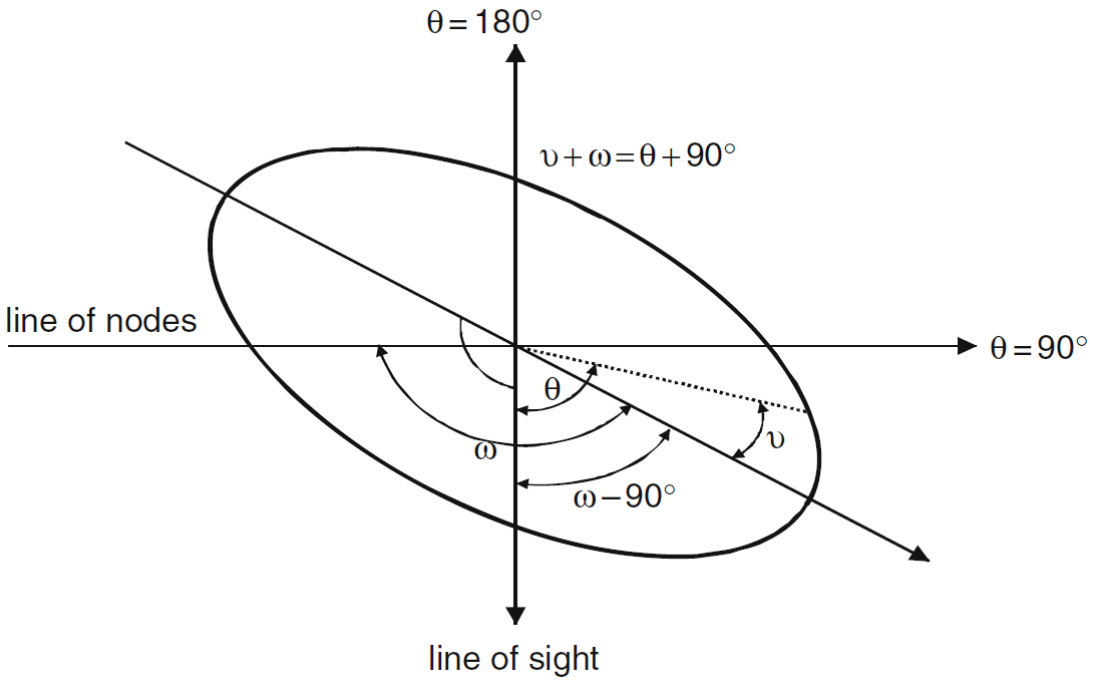
3.3.2. Kružna orbita

Usled dejstva plimskih sila u dvojnom sistemu tokom vremena dolazi do sinhronizacije i cirkularizacije orbite (Hut, 1981). Ovo posebno važi za tesno dvojne sisteme sa periodom manjim od 5 dana, za koje su Lucy i Sweeney (1971) našli da velika većina ima kružne orbite. Veza između položaja zvezde i fotometrijske faze Φ je

Jednačina 3.24

$$\theta = 2\pi\Phi.$$

Separacija ne zavisi od faze i uvek je $D = 1$. Minimumi krive sjaja padaju na $\Phi = 0$ (primarni) i $\Phi = 0.5$ (sekundarni).



Slika 3.9 Veza između geometrijske faze i orbitalnih elemenata. Preuzeto iz Kallrath & Milone (2009).

3.3.3. Eliptična orbita

Veza između geometrijske i fotometrijske faze je nešto komplikovanija za eliptične orbite. Da bi se ona izvela potrebno je uvesti nekoliko veličina (slika 3.10):

- Prava anomalija υ je ugao između pravaca ka periastronu i ka poziciji sekundarne komponente sa temenom u žiži elipse. Meri se u pravcu suprotnom od kretanja kazaljke na satu.
- Ekscentrična anomalija E je ugao između periastrona i projekcije položaja sekundarne komponente na pomoćni krug sa temenom u centru elipse.
- Srednja anomalija M je ugao koji radijus-vektor prebriše od trenutka prolaska kroz periastron τ do datog trenutka t krećući se srednjom ugaonom brzinom $\frac{2\pi}{P}$.

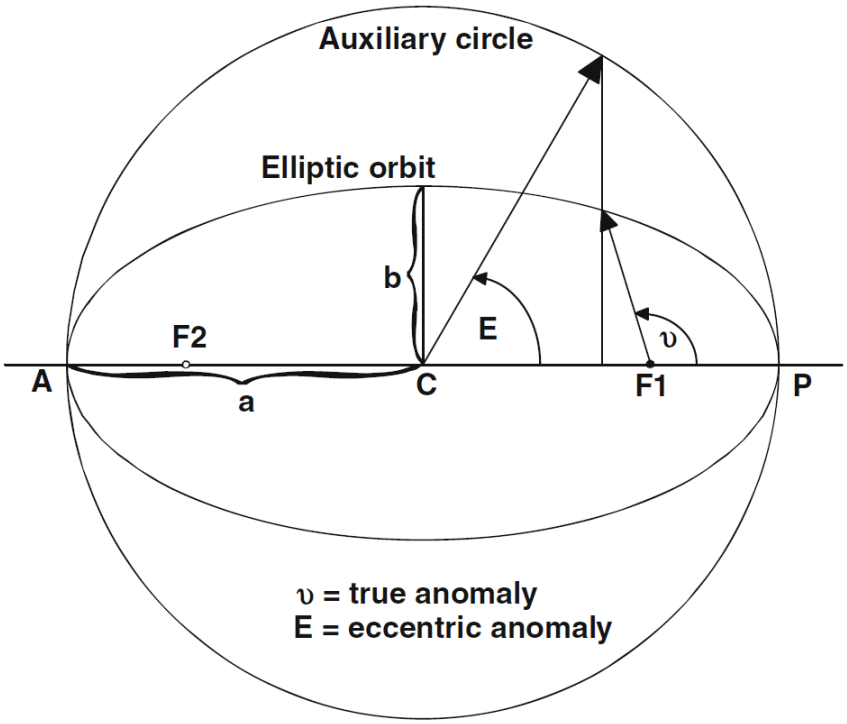
Jednačina 3.25

$$M = 2\pi \frac{t - \tau}{P}.$$

Sa slike 3.9 se vidi da je

Jednačina 3.26

$$\theta = \upsilon + \omega - \frac{\pi}{2}.$$



Slika 3.10 Veza između prave i ekscentrične anomalije. Preuzeto iz Kallrath & Milone (2009).

Prava anomalija je povezana sa ekscentričnom anomalijom sledećim izrazom:

Jednačina 3.27

$$\tan\frac{\upsilon}{2} = \sqrt{\frac{1+e}{1-e}}\tan\frac{E}{2}$$

Ekscentrična anomalija je sa druge strane povezana sa srednjom anomalijom Keplerovom jednačinom:

Jednačina 3.28

$$E - e\sin E = M.$$

Ova jednačina nema analitičko rešenje i rešava se iterativnim postupkom. Iz definicije srednje anomalije (jednačina 3.25) se vidi da je referentna tačka trenutak prolaska kroz periastron. Sa druge strane, fazu merimo od primarnog minimuma, što daje vezu:

Jednačina 3.29

$$M = 2\pi\Phi + M_0.$$

Sa slike 3.9 se vidi da je prava anomalija koja odgovara primarnom minimumu $v_0 = 90 - \omega$, pa se M_0 lako računa iz jednačina 3.27 i 3.28.

Kada smo odredili geometrijsku fazu θ, koordinate centra sekundarne komponente će biti:

Jednačina 3.30

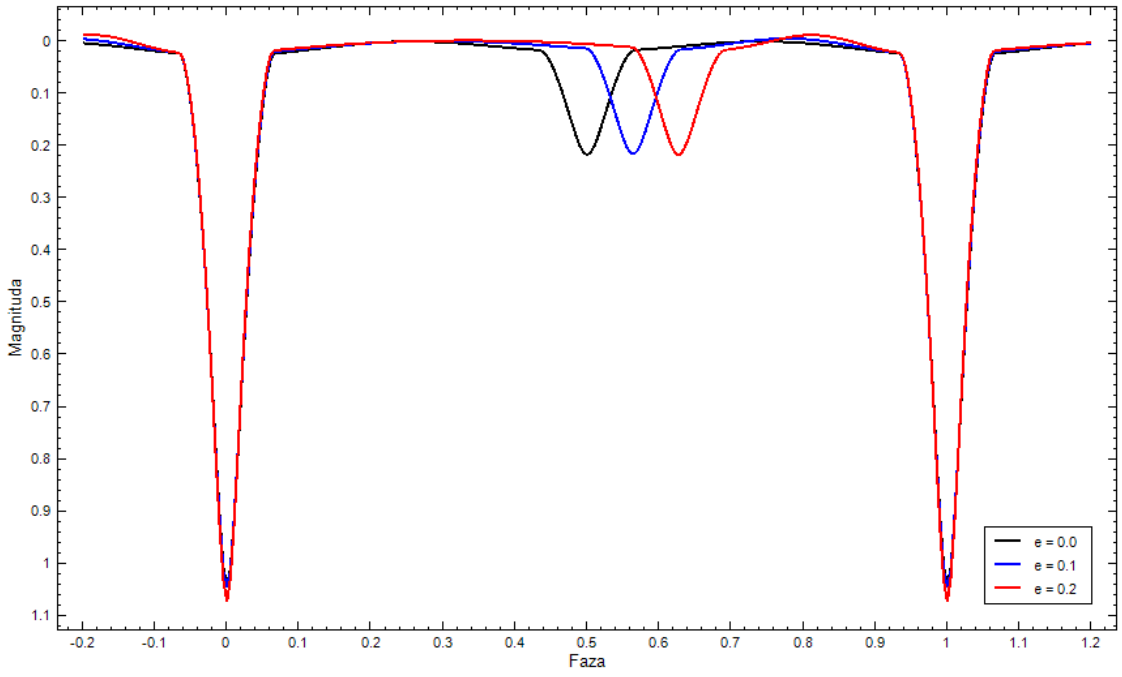
$$x = D\cos\theta\,,$$
$$y = D\sin\theta\,,$$
$$z = 0,$$

gde je separacija data sa:

Jednačina 3.31

$$D = 1 - e\cos E.$$

Uticaj ekscentričnosti na izgled krive sjaja se može videti na slici 3.11.



Slika 3.11 Uticaj ekscentričnosti na izgled krive sjaja.

4. Detekcija vidljivosti

Sledeći korak u sintezi krive sjaja je modeliranje pomračenja, odnosno određivanje mape vidljivosti u datoj fazi. Mapa vidljivosti χ_i je karakteristična funkcija koja se pripisuje svakoj elementarnoj ćeliji i definiše se na sledeći način (Kallrath & Milone, 2009):

Jednačina 4.1

$$\chi_i = \begin{cases} 1, & \text{ako je ćelija } i \text{ vidljiva} \\ 0, & \text{ako ćelija } i \text{ nije vidljiva} \end{cases}.$$

Ukoliko je posmatrana elementarna ćelija vidljiva (tj. nije zaklonjena drugom komponentom ili akrecionim diskom), onda njeno zračenje stiže do posmatrača i doprinosi ukupnom fluksu sistema u toj fazi.

Algoritam za određivanje mape vidljivosti je jedan od značajnijih aspekata modela dvojnog sistema, jer pretpostavke pod kojima je on izveden u velikoj meri određuju koji se efekti i konfiguracije mogu modelirati. Poželjno je da algoritam pretpostavlja što manje kada je u pitanju geometrijska reprezentacija sistema i da bude što jednostavniji za implementaciju, a da sa druge strane bude dovoljno efikasan da omogućava računanje u razumnom vremenskom roku. Primer za ograničenja usled neispunjenosti prvog uslova je algoritam iz modela GDDSYN (Hendry & Mochnacki, 1992) koji pretpostavlja da su projekcije komponenata na ravan tangencijalnu na nebesku sferu konveksni poligoni, što, recimo, onemogućava modeliranje akrecionih diskova kod kojih ovaj uslov nije ispunjen. Primer za drugi uslov je model dr Đuraševica (1991), kod kojeg bi matematička formulacija uslova za pomračenje kod komponenata na kojima je površina perturbovana usled neradijalnih pulsacija bila izuzetno komplikovana. U ovoj tezi se predlaže novi algoritam koji zadovoljava oba ova uslova.

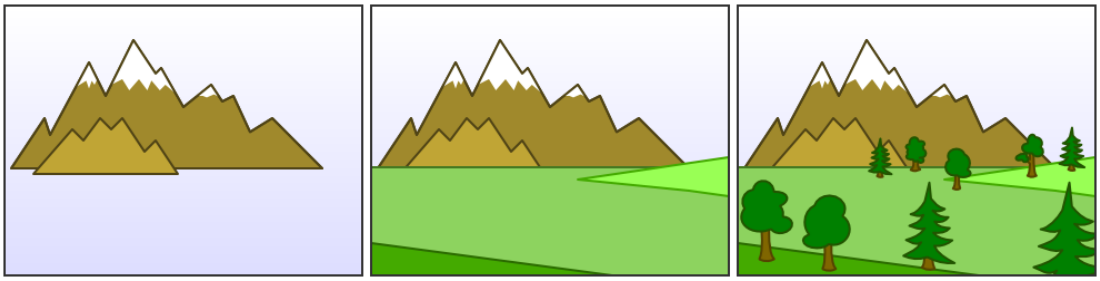
4.1. Inverzni slikarski algoritam

4.1.1. Opšti pregled

U osnovi algoritma je varijanta tzv. slikarskog algoritma iz polja računarske grafike (Foley, et al., 1995). Ime potiče od činjenice da se pri slikanju obično prvo crtaju objekti najudaljeniji od posmatrača, pa onda sve bliži i bliži (slika 4.1). Kako se dodaju novi

objekti na platno, oni prekrivaju objekte koji se nalaze iza njih. Ovo jednostavno zapažanje se može iskoristiti za određivanje koje se elementarne ćelije vide u datoj fazi.

Formiramo dva niza – jedan koji sadrži sve elementarne ćelije (od svih komponenti), sortirane po udaljenosti od posmatrača (najdalji objekat je prvi u nizu) i jedan pomoćni niz koji čuva spisak vidljivih ćelija. Drugi niz je na početku algoritma prazan. Uzmemo prvi član iz niza sa elementarnim ćelijama (najudaljeniji od posmatrača) i dodamo ga u pomoćni niz. Za drugi član niza proverimo da li pokriva prvi. Ako da, onda uklonimo prvi član iz pomoćnog niza i dodamo drugi. Ako ne, onda samo dodamo drugi član. Za svaku sledeću elementarnu ćeliju iz prvog niza proveravamo da li pokriva bilo koju ćeliju iz pomoćnog niza i iz njega uklonimo sve pokrivene ćelije. Na kraju ovog procesa pomoćni niz sadrži sve vidljive ćelije, tj. predstavlja mapu vidljivosti.



Slika 4.1 Slikarski algoritam. Prvo se crtaju najudaljeniji objekti (planina), pa onda bliži (dolina) i na kraju objekti najbliži posmatraču (drveće).

Algoritam u ovom obliku je izuzetno neefikasan jer uključuje veliki broj operacija uklanjanja ćelija iz pomoćnog niza. Mnogo bolji pristup je da se ćelije sortiraju tako da je ćelija najbliža posmatraču prva u nizu, tj. obrnutim redom u odnosu na originalni slikarski algoritam. Otuda i naziv ove varijante – inverzni slikarski algoritam. Za ćeliju koja je najbliža posmatraču je garantovano da će biti vidljiva i nju automatski stavljamo u pomoćni niz. Za svaku sledeću ćeliju se proverava da je preklopljena nekom od ćelija iz pomoćnog niza. Ako nije, onda je ta ćelija vidljiva i dodaje se u pomoćni niz. U suprotnom se preskače i prelazi na sledeći element niza. Na kraju dobijamo isti rezultat kao i u originalnom algoritmu, mapu vidljivosti. Prednost ovog pristupa je to što se elementarne ćelije samo dodaju na kraj pomoćnog niza, tj. nema uklanjanja i pomeranja elemenata što dovodi do značajnog ubrzanja u odnosu na originalni algoritam.

Glavna prednost inverznog slikarskog algoritma je to što radi direktno na elementarnim ćelijama i samim tim ne zavisi od oblika potencijala kojim se opisuje površina

komponenata. Ovo otvara mogućnost jednostavnog proširenja na egzotičnije slučajeve kao što su hijerarhiski sistemi, sistemi sa ekscentričnim ili cirkumbinarnim akrecionim diskom, planetarni sistemi itd. Jedan uspešan primer je i proširenje modela na sisteme koji sadrže komponente sa neradijalnim pulsacijama (Latković, 2013, u pripremi). Na slici 4.2 se mogu videti primeri za neke interesantnije slučajeve.

4.1.2. Detaljan prikaz

U prethodnom odeljku je prikazan algoritam u opštim crtama. Ovde će biti dat detaljniji prikaz.

Ulazni podatak za algoritam je niz poligonih mreža kojima se predstavljaju komponente sistema. Koordinate temena poligonih mreža su date u koordinatnom sistemu $\mathcal{P}$ vezanom za nebesku sferu. Definicija tog sistema, kao i transformacione jednačine su date u odeljku 3.3.1.

Prvi korak je uklanjanje elementarnih ćelija koje su okrenute nauprot posmatraču. Ako je pravac ka posmatraču određen vektorom $\vec{p}(1,0,0)$, sa slike 4.3 vidimo da je elementarna ćelija vidljva samo ako je ispunjen uslov:

Jednačina 4.2

$$\cos\gamma = \vec{p}\cdot\vec{n} > 0,$$

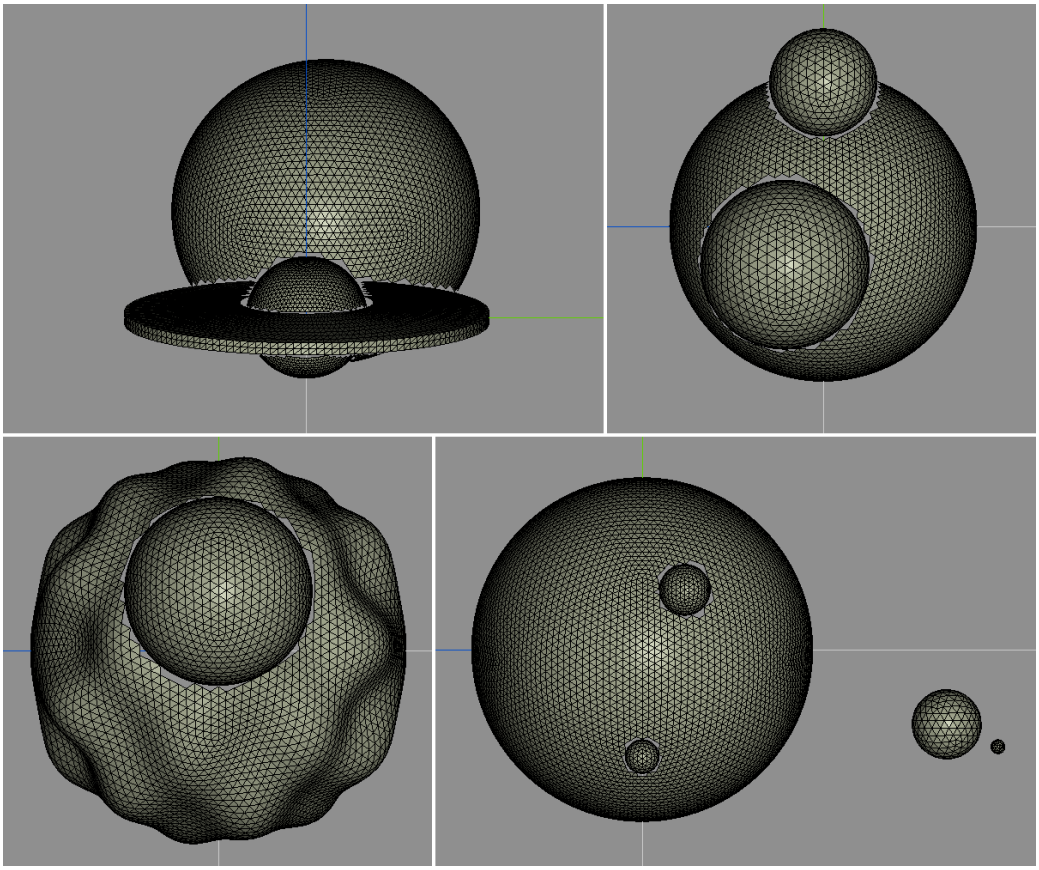
gde je $\vec{n}$ vektor normale na elementarnu ćeliju. Ovo je potreban, ali ne i dovoljan uslov.

Drugi korak je projektovanje mreže na ravan nebeske sfere. U pitanju je ortogonalna projekcija duž x-ose, tako da se koordinate tačke u ravni nebeske sfere dobijaju jednostavnim zanemarivanjem x-komponente temena poligone mreže:

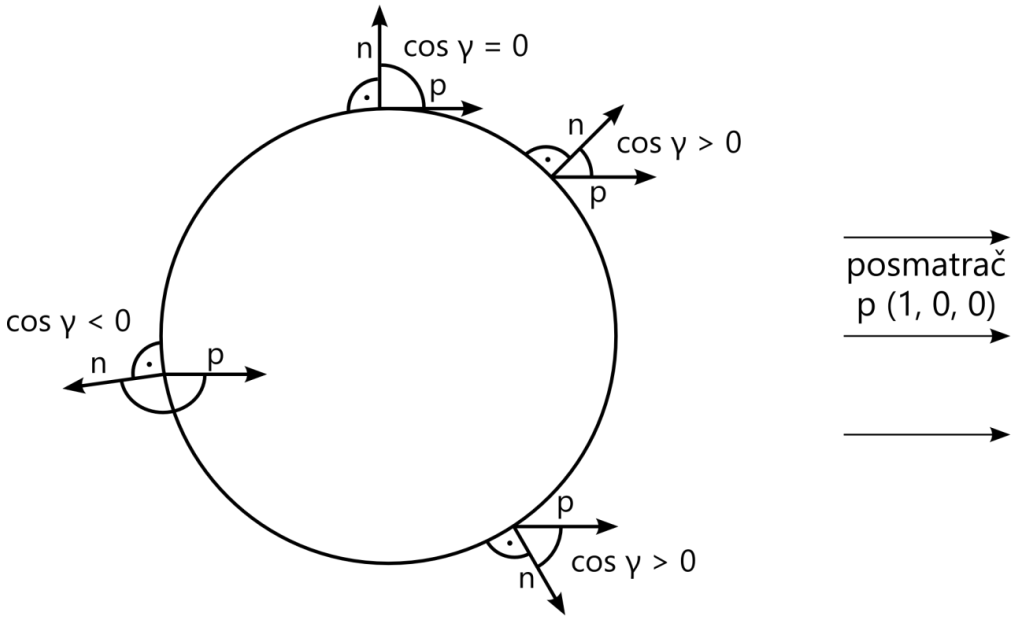
Jednačina 4.3

$$\begin{pmatrix} x \\ z \\ y \end{pmatrix} \xrightarrow{\mathcal{P}\to\mathcal{P}'} \begin{pmatrix} y \\ z \end{pmatrix}.$$

U ovom koraku je potrebno sačuvati udaljenost temena mreže od posmatrača. U suštini, posmatrač je smešten na x-osi u beskonačnosti u odnosu na sistem, ali je za potrebe sortiranja elementarnih ćelija dovoljno pretpostaviti da je posmatrač na nekoj konačnoj, proizvoljnoj udaljenosti koja je dovoljno daleko od sistema, recimo na koordinatama $(5,0,0)$.

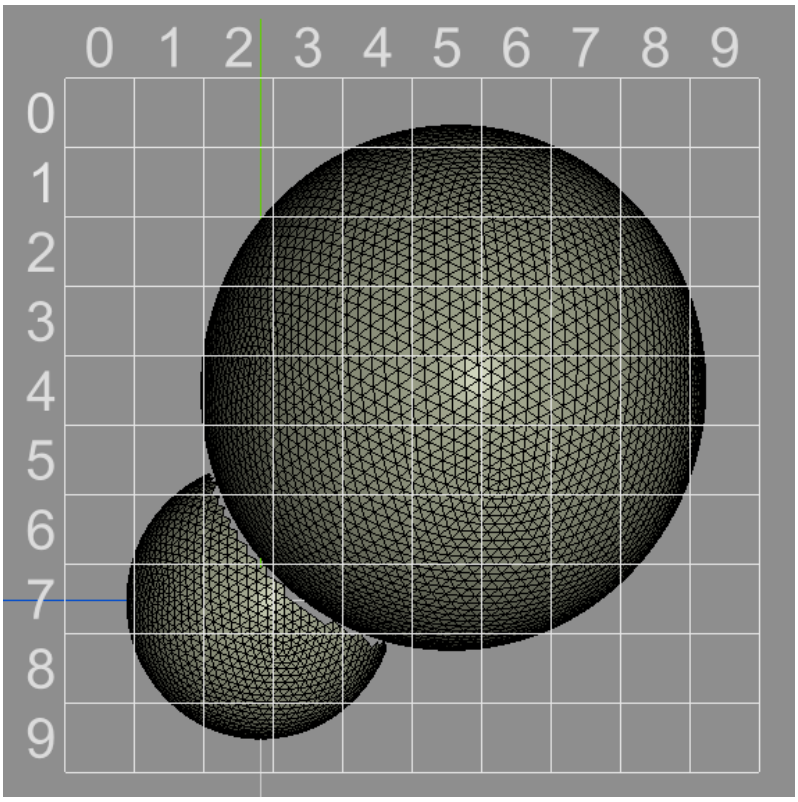


Slika 4.2 Demonstracija algoritma za detekciju vidljivosti. Gore levo: tesno dvojni sistem sa akrecionim diskom oko primarne komponente. Gore desno: trojni sistem. Dole levo: sistem sa pulsirajućom komponentom; amplituda pulsacije je preuveličana da bi samopomračenje delova površine primara bilo jasnije. Dole desno: planetni sistem; planete se modeliraju sferama.

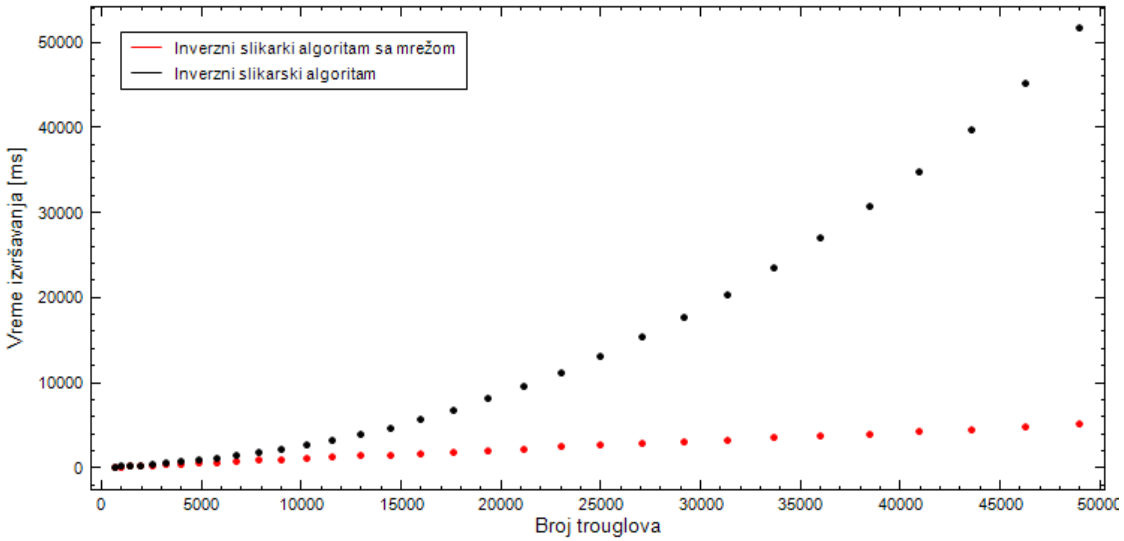


Slika 4.3 Uklanjanje elementarnih ćelija koje su orijentisane od posmatrača.

Sledeći korak je formiranje niza koji sadrži sve elementarne ćelije u sistemu i njegovo sortiranje po udaljenosti od posmatrača, od najbliže do najdalje ćelije. Prva ćelija u nizu je uvek vidljiva i nju odmah dodajemo u pomoćni niz u kojem se čuvaju pomoćne ćelije. Sa slike 4.4 se odmah vidi jedno moguće poboljšanje algoritma. Ako preko projekcije sistema preklopimo tabelu dimenzija $n \times m$ i ako utvrdimo da se elementarna ćelija projektuje u recimo ćeliju tabele (i, j) onda je dovoljno proveriti da li je preklopljena vidljivim elementarnim ćelijama samo iz okolnih ćelija tabele (za $i \pm 1$ i $j \pm 1$, ukupno devet ćelija). Ovo značajno smanjuje broj neophodnih provera i u velikoj meri ubrzava algoritam. Ako veličinu ćelije tabele odaberemo tako da prosečno sadrži do desetak elemenata, onda će vreme izvršavanja algoritma praktično linearno rasti sa ukupnim brojem elementarnih ćelija u sistemu. Poređenja radi, vreme izvršavanja izvornog inverznog slikarskog algoritma opisanog u prethodnom odeljku je proporcionalno kvadratu ukupnog broja elementarnih ćelija (slika 4.5).



Slika 4.4 Inverzni slikarski algoritam sa mrežom.



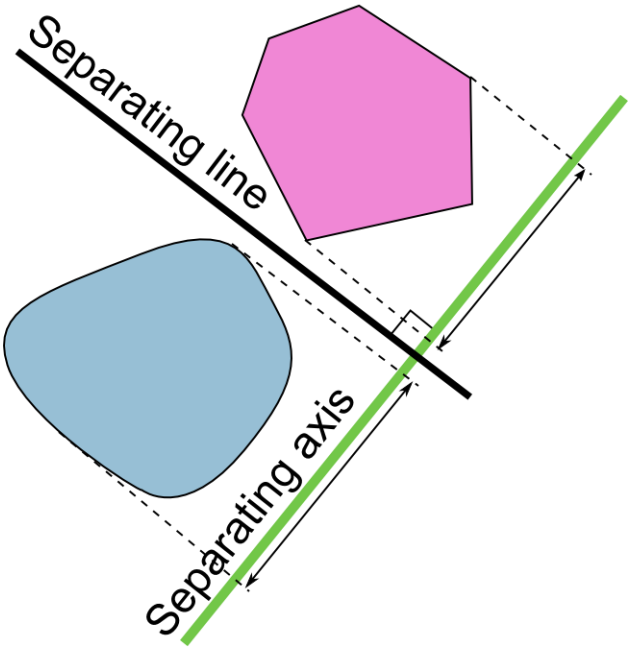
Slika 4.5 Poređenje vremena izvršavanja inverznog slikarskog algoritma i modifikovanog algoritma opisanog u ovom odeljku. Kod originalnog algoritma ona zavisi od kvadrata broja elementarnih ćelija, dok kod modifikovanog algoritma linearno raste sa brojem trouglova.

Iako se u opštem slučaju poligona mreža može sastojati od poligona sa proizvoljnim brojem stranica, ovde zahtevamo da ona bude formirana isključivo od trouglova. Prvo, za trouglove postoje jako efikasni algoritmi za utvrđivanje da li se međusobno preklapaju. Drugo, za njih je garantovano da su koplanarni i da će njihova projekcija na ravan nebeske sfere uvek biti trougao. Sa druge strane se ne gubi na opštosti, jer se svaki poligon može „razbiti“ na niz trouglova. Za određivanje da li se dva trougla preklapaju se koristi implementacija teoreme razdvajajuće prave (*separating axis theorem*) opisana u članku Cozic-a (2006). Ova teorema tvrdi da se dva konveksna poligona u ravni ne seku ako i samo ako postoji prava na kojoj se projekcije ova dva poligona ne preklapaju (slika 4.6).

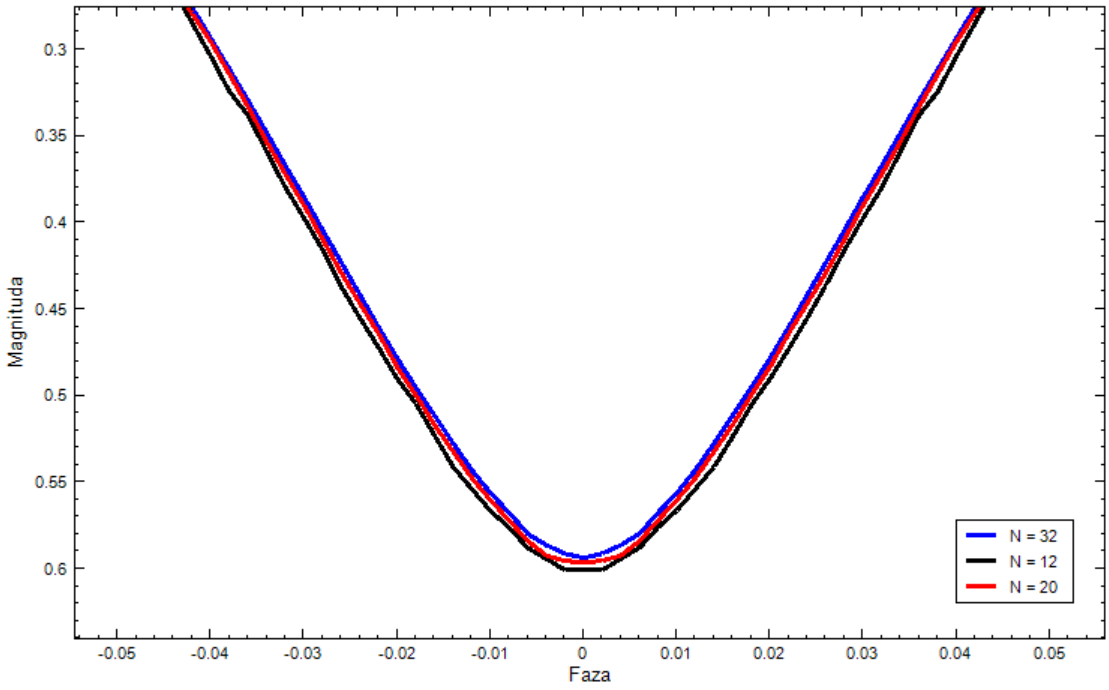
4.2. Adaptivna geodezijska mreža

Iz prikaza algoritma se vidi da će vidljiva površina komponenata biti uvek potcenjena, jer se smatra da je elementarna ćelija pokrivena ako je bilo koji njen deo prekriven ćelijama koje su bliže posmatraču. To znači da se njen doprinos ukupnom fluksu zračenja ne uzima u obzir iako se može desiti da je samo delimično pomračena. Taj efekat je tim izraženiji što je početna podela grublja. Na slici 4.7 je prikazana dubina primarnog minimuma za različit broj elementarnih ćelija. Sa povećanjem broja elementarnih ćelija se povećava tačnost izračunate krive sjaja, ali se sa druge strane

značajno produžava vreme izvršavanja (slika 4.5). Da bi se rešio ovaj problem, odnosno da bi se postigao kompromis između tačnosti i brzine izvršavanja, razvijena je adaptivna geodezijska mreža.



Slika 4.6 Teorema razdvajajuće prave.



Slika 4.7 Dubina primarnog minimuma u zavisnosti od odabranog nivoa podele N.

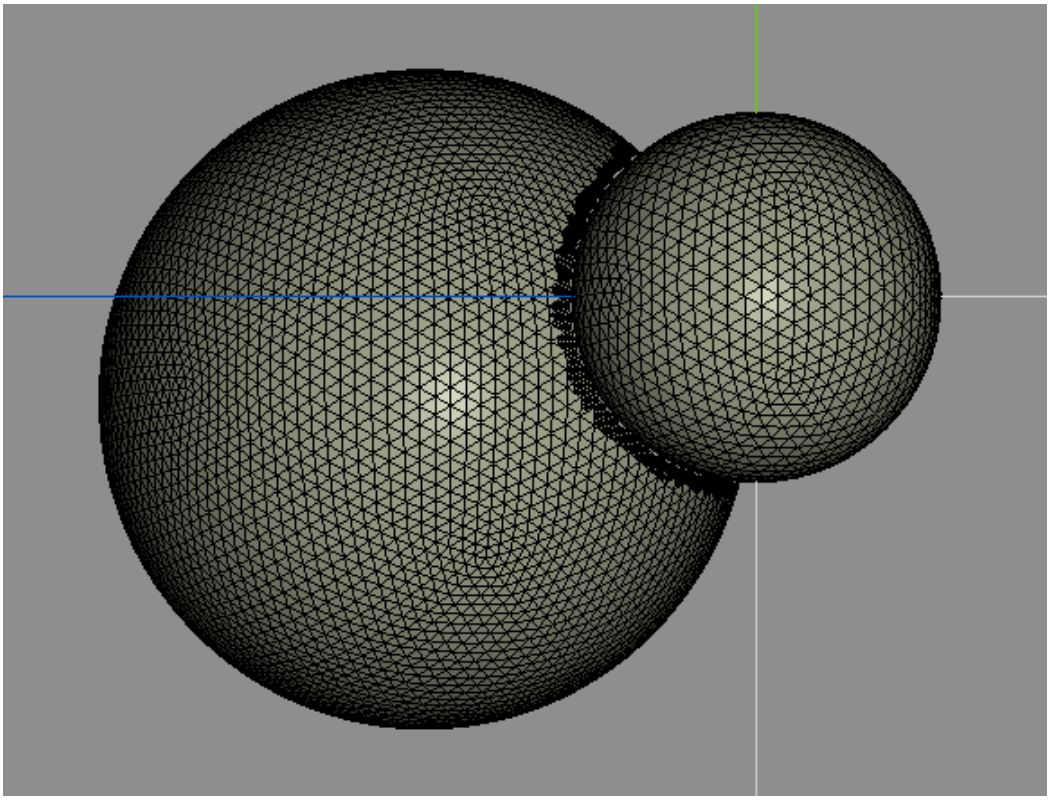
Osnovna ideja metode je da se u svakoj fazi poveća broj elementarnih ćelija na zvezdi samo u regionima na koje pada senka druge komponente. Ovaj korak se izvodi tokom faze detekcije vidljivosti, na samom početku, pre projektovanja poligone mreže. Pošto želimo samo grubo da odredimo oblast u kojoj može doći do pomračenja, a ne tačan oblik projekcije senke komponente koja je bliže posmatraču (koji se nalazi na x-osi, u plus beskonačnosti), senku možemo aproksimirati cilindrom čija je osa paralelna x-osi i prolazi kroz centar zvezde koja pomračuje. Ako oblik zvezde opisujemo Roche potencijalom (odeljak 3.1.4), maksimalni radijus r_{max} će biti onaj koji se nalazi na x-osi, ka Lagrange-ovoj L1 tački, dok će minimalni r_{min} biti polarni radijus. Da bi formirali uslov za određivanje da li postoji mogućnost da elementarna ćelija bude pomračena koristimo zapažanje da će radijus-vektor svake tačke na rubu zvezde uvek biti između r_{min} i r_{max}. To znači da ako se elementarna ćelija zvezde koja je pomračena nalazi van cilindra poluprečnika r_{max} biti uvek vidljiva, dok će ćelija koja se nalazi unutar cilindra radijusa r_{min} uvek biti pomračena. U suprotnom, ako je ćelija između ova dva cilindra, onda postoji mogućnost za pomračenje. Datu ćeliju razbijamo na sitnije delove koristeći rekurzivnu podelu (odeljak 3.1.2) i time povećavamo tačnost određivanja krive sjaja (slika 4.8).

Ovaj uslov u koordinatnom sistemu vezanom za nebesku sferu ima izuzetno jednostavan oblik. Ako su (x_c, y_c, z_c) koordinate centra zvezde koja pomračuje, r_{min} i r_{max} minimalni i maksimalni radijus zvezde koja pomračuje i (x, y, z) koordinate centroida elementarne ćelije na zvezdi koja je pomračena, onda je

Jednačina 4.4

$$r_{min} < \sqrt{(y - y_c)^2 + (z - z_c)^2} < r_{max}.$$

Dubina rekurzije N_A pri podeli elementarne ćelije je parametar modela.



Slika 4.8 Adaptivna mreža za adaptivni nivo podele $N_A = 2$.

5. Sinteza krive sjaja i radijalnih brzina

Poslednji korak u rešavanju direktnog problema je računanje izlaznog fluksa i radijalnih brzina. Izlazni fluks se dobija sabiranjem pojedinačnih doprinosa vidljivih elementarnih ćelija. Pri njegovom računu se uzimaju u obzir sledeći efekti:

- Gravitaciono potamnjenje
- Efekat refleksije
- Aktivni regioni (pege)
- Potamnjenje ka rubu

Tretman ovih efekata uglavnom prati Đuraševićdv program (Đurašević, 1991) i detaljnije je opisan u narednim odeljcima. Računanje krive radijalnih brzina prati postupak iz članka Wilson & Sofia (1976).

5.1. Gravitaciono potamnjenje

Pretpostavljamo da je zvezda u stanju hidrostatičke i radijativne ravnoteže. Ukoliko njen oblik odstupa od sfernog usled rotacije i/ili plimskih interakcija u dvojnom sistemu, doći će do varijacije u površinskom sjaju i gravitacionom ubrzanju od pola ka ekvatoru (Tassoul, 2000). Ova varijacija se naziva gravitacionim potamnjenjem. Von Zeipel (1924) je pokazao da je izlazni fluks $\vec{F}$ proporcionalan lokalnoj vrednosti gravitacionog ubrzanja $\vec{g}$:

Jednačina 5.1

$$\vec{F} = \frac{4ac}{3\kappa\rho} T^3 \frac{dT}{d\psi} \vec{g} = -k(\psi)\vec{g}.$$

Vektor gravitacionog ubrzanja je gradijent potencijala ψ, pa se vrednost njegovog intenziteta za svaku elementarnu ćeliju može preko jednačine 3.12 izraziti kao:

Jednačina 5.2

$$g_i = \gamma \frac{m_1}{D^2} |\nabla C|,$$

gde je C modifikovani Kopalov potencijal. U opštem slučaju će to biti izraz za potencijal u sistemu sa ekscentričnom orbitom i asinhronom rotacijom komponente

(jednačina 3.16). Izraz za intenzitet gradijenta potencijala u pravouglom koordinatnom sistemu je:

Jednačina 5.3

$$|\nabla C| = \sqrt{C_x^2 + C_y^2 + C_z^2},$$

gde su

Jednačina 5.4

$$C_x = \frac{\partial C}{\partial x} = -\frac{x}{\sqrt{(x^2+y^2+z^2)^3}} + q\left[\frac{(D-x)}{\sqrt{((D-x)^2+y^2+z^2)^3}} - \frac{1}{D^2}\right] + (q+1)f^2 x,$$

$$C_y = \frac{\partial C}{\partial y} = -\frac{y}{\sqrt{(x^2+y^2+z^2)^3}} - q\frac{y}{\sqrt{((D-x)^2+y^2+z^2)^3}} + (q+1)f^2 y,$$

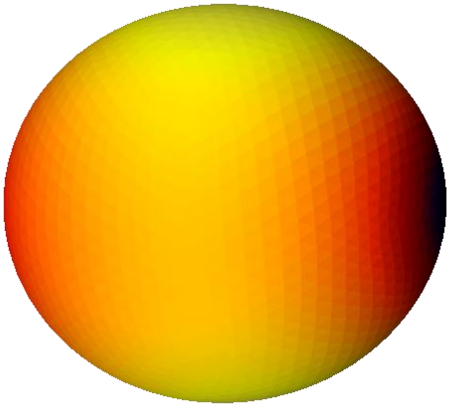
$$C_z = \frac{\partial C}{\partial z} = -\frac{z}{\sqrt{(x^2+y^2+z^2)^3}} - q\frac{z}{\sqrt{((D-x)^2+y^2+z^2)^3}}.$$

Iz Stefan–Boltzmann-ovog zakona $F \sim T^4$ sledi da će i temperatura proizvoljne tačke na površini zvezde biti proporcionalna vrednosti gravitacionog ubrzanja. Ukoliko za referentne vrednosti uzmemo efektivnu temperaturu i efektivno gravitaciono ubrzanje, onda je izraz za temperaturu i-te elementarne ćelije:

Jednačina 5.5

$$T_i = T_{eff}\left(\frac{g_i}{g_{eff}}\right)^{\beta},$$

gde je g_i gravitaciono ubrzanje i-te elementarne ćelije. Vrednost eksponenta β je jednaka $\beta = 0.25$ za radijativne atmosfere (iz von Zeipel-ovog zakona - jednačina 5.1). Lucy je izračunao da je za zvezde u konvektivnoj ravnoteži vrednost ovog eksponenta približno jednaka $\beta = 0.08$ (Lucy, 1967). Obe vrednosti su potvrđene u kasnijim posmatračkim studijama (videti na primer rad Rafert & Twigg (1980)).



Slika 5.1 Gravitaciono potamnjenje. Efektivna temperatura zvezde je $T_{eff} = 6000\ K$. Žutom bojom je označena maksimalna temperatura ($T_{eff} = 6046K$), a crnom minimalna ($T_{eff} = 5909\ K$).

Efektivno gravitaciono ubrzanje je dato izrazom (Đurašević, 1991):

Jednačina 5.6

$$g_{eff} = \frac{\sum_{i=1}^{N} g_i S_i}{\sum_{i=1}^{N} S_i},$$

gde je S_i površina i-te elementarne ćelije. Efektivna temperatura je parametar modela i zadaje se kao ulazna veličina.

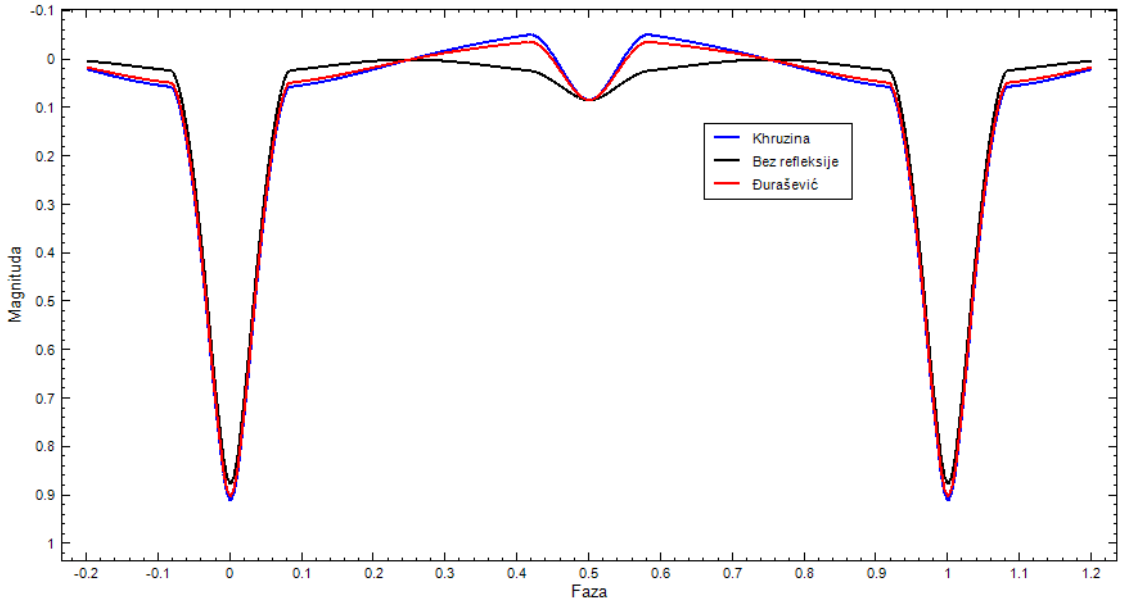
Varijacija temperature od pola ka ekvatoru je prikazana na slici 5.1.

5.2. Efekat refleksije

U dvojnim sistemima dolazi do međusobnog zagrevanja komponenata. Zračenje sa druge komponente dovodi do preraspodele temperature na strani zvezde koja je okrenuta ka njoj, i obrnuto. Ovo je posebno izraženo kod tesno dvojnih sistema kod kojih postoje značajne razlike u efektivnim temperaturama. Primer uticaja na krivu sjaja se može videti na slici 5.2.

Efekat refleksije je izuzetno komplikovan. Tačan tretman ovog efekta bi zahtevao rešavanje 3D jednačine prenosa zračenja uz složenu geometriju modela. Iako postoje značajni napori u tom pravcu, iz praktičnih razloga se pribegava raznim pojednostavljenjima (da bi vreme potrebno za računanje modela bilo u razumnim

granicama). Ovde se prati model efekta refleksije koji je predložio Đurašević (Đurašević, 1991). Geometrija modela je prikazana na slici 5.3.



Slika 5.2 Uticaj refleksije na krivu sjaja. Efektivna temperatura primarne komponente je 31000K, a sekundarne 10000K.

Sa slike 5.3 se vidi da će do elementarne ćelije stizati zračenje sa druge komponente ako je ispunjen uslov:

Jednačina 5.7

$$\cos\alpha > 0,$$

gde je α ugao između normale na ćeliju i pravca ka centru druge komponente. Tada sa te ćelije vidimo drugu komponentu pod prostornim uglom:

Jednačina 5.8

$$\Omega = 2\pi \int_0^{\theta} \sin\theta'\, d\theta' = 2\pi(1-\cos\theta);\ \sin\theta = \frac{r_2}{\rho_*}.$$

To znači da do uočene elementarne ćelije stiže sledeći deo ukupnog zračenja druge komponente:

Jednačina 5.9

$$W = \frac{\Omega}{4\pi}.$$

Ukupna gustina zračenja elementarne ćelije je zbir „sopstvene“ gustine zračenja i gustine zračenja koja stiže sa druge komponente umanjena za W i iznosi:

Jednačina 5.10

$$\rho_{zr} = \rho_1 + \rho_2 A_1 \cos\alpha\, W.$$

Faktor A_1 se naziva bolometrijskim albedom. To je globalni parametar koji opisuje makroskopske osobine zvezdane atmosfere i predstavlja deo zračenja koje je izračeno od ukupnog pristiglog fluksa sa druge komponente. Za zvezde sa atmosferom u radijativnoj ravnoteži vrednost albeda je $A = 1$, dok će za zvezde sa atmosferom u konvektivnoj ravnoteži on biti nešto manji ($0 \leq A \leq 1$). Obično se uzima da je $A = 0.5$ sledeći preporuku koju je dao Rucinski (1969). Ova vrednost je izvedena iz razmatranja sistema Algolovog tipa sa izraženim efektom refleksije kod kojih je sekundarna komponenta u konvektivnoj ravnoteži.

Iz Stefan–Boltzmann-ovog zakona sledi da se jednačina 5.10 za i-tu elementarnu ćeliju može pisati u sledećem obliku:

Jednačina 5.11

$$T_i^* = T_i \sqrt[4]{1 + A_1 \cos\alpha \left(1 - \sqrt{1 - \left(\frac{r_2}{\rho_*}\right)^2}\right)\left(\frac{T_2}{T_1}\right)^4},$$

gde je T_i temperatura koju bi ćelija imala da nema efekta refleksije. Ova korekcija se primenjuje samo ako je ispunjen uslov 5.7.

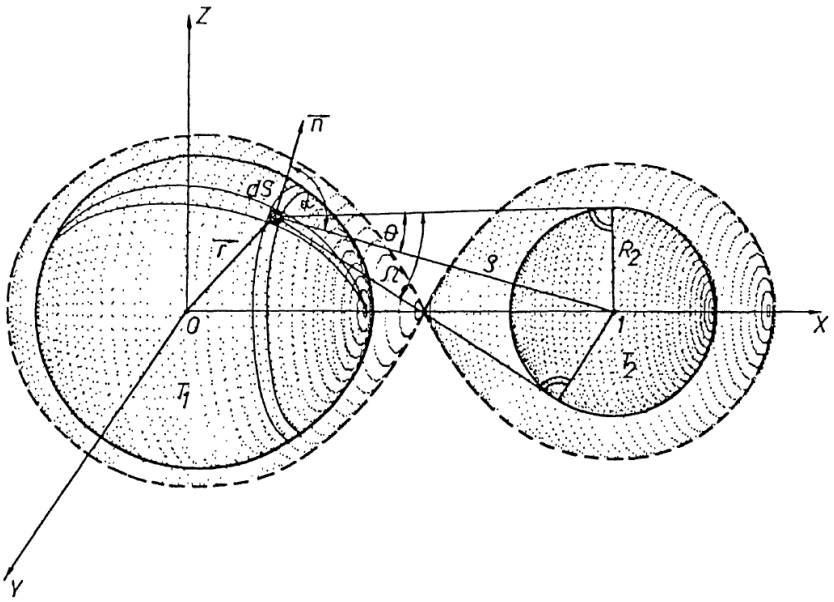
U poslednjim verzijama Đuraševićevog programa se koristi i model refleksije koji je predložila Khruzina (1985). On se bazira na pretpostavci da je druga komponenta tačkasti izvor zračenja. Izraz za korekciju temperature u ovom slučaju glasi

Jednačina 5.12

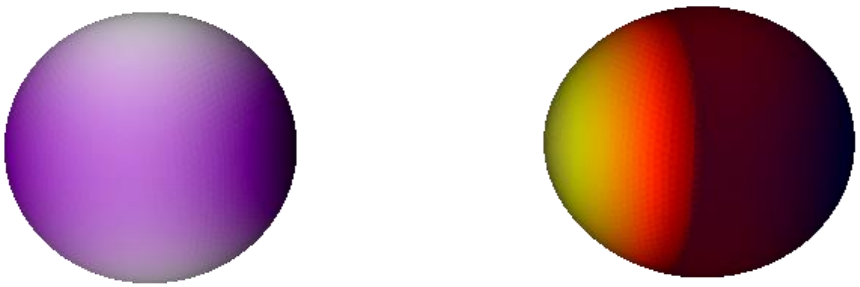
$$T_i^* = \sqrt[4]{T_i^4 + \frac{A_1 \cos\alpha}{4\pi\rho_*^2\sigma} L_2},$$

gde je L_2 bolometrijska luminoznost druge komponente, a σ Stefan-Boltzmann-ova konstanta. Oba ova modela su implementirana u kodu.

Uticaj efekta refleksije na raspodelu temperature zvezde je prikazan na slici 5.4.



Slika 5.3 Model efekta refleksije. Preuzeto iz Đurašević (1992a).



Slika 5.4 Efekat refleksije.

5.3. Zvezdane pege

Zvezdane pege su fotosferski regioni sa većom ili manjom temperaturom u odnosu na okolnu materiju. One se na Suncu proučavaju već jako dugo. U poslednjih nekoliko decenija su razvijene posmatračke tehnike koje su omogućile posmatranje pega i na drugim zvezdama (Sasselov, 1998). One mogu nastati bilo zbog prisustva magnetnog polja i diferencijalne rotacije (kao kod Sunca) ili zbog prisustva nekih hemijskih nehomogenosti na površini zvezde. U nekim slučajevima mogu pokriti i do 40% površine zvezde (Đurašević, 1991).

Prisustvo pega na površini zvezde dovodi do asimetričnih krivih sjaja. Njihovim uključivanjem u model se može objasniti O'Connel-ov efekat (Yamasaki, 1982).

O’Connel–ovim efektom se naziva pojava da maksimumi sjaja pre i posle sekundarnog pomračenja nisu iste visine (slika 5.6).

Geometrija pege je predstavljenja na slici 5.5. U *Infinity*-ju se pretpostavlja da su pege kružnog oblika i da su određene sa četiri parametra – longitudom centra pege φ_p, latitudom centra pege θ_p, uglovnim radijusom R_p i temperaturnim kontrastom pege K_p. Koordinate centra pege se mere u lokalnom koordinatnom sistemu C_1, odnosno C_2 vezanom za centar komponente (odeljak 3.1.4). Temperaturni kontrast pege se definiše kao odnos temperature unutar pege T_p i temperature T_l koju bi imala elementarna ćelija da nema pege:

Jednačina 5.13

$$K_p = \frac{T_p}{T_l}.$$

Ukoliko je $K_p < 1$, onda je pega tamna. U suprotnom, pega je svetla. Primer uticaja tamne pege na izgled krive sjaja je prikazan na slici 5.6, dok je na slici 5.7 data raspodela temperature na površini zvezde usled njenog prisustva.

Da bi se utvrdilo koje elementarne ćelije pripadaju pegi treba konstruisati veliki krug na površini zvezde koji prolazi kroz centar elementarne ćelije i centar pege. Luk l koji spaja ova dva centra predstavlja najkraće rastojanje između ove dve tačke. Iz sferne trigonometrije dobijamo sledeći izraz:

Jednačina 5.14

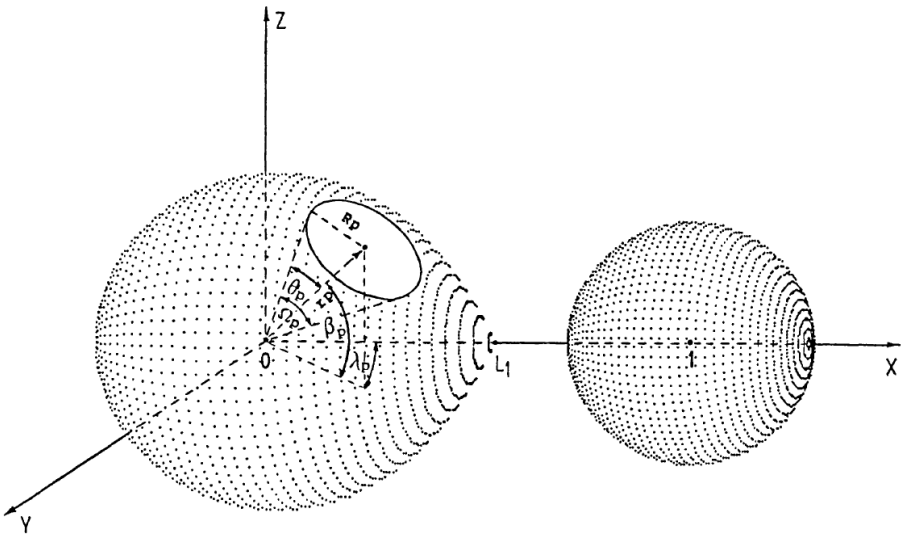
$$\cos l = \cos\theta_p \cos\theta + \sin\theta_p \sin\theta \cos(\varphi - \varphi_p).$$

Ćelija pripada pegi ako je ispunjen uslov $l < R_p$, odnosno:

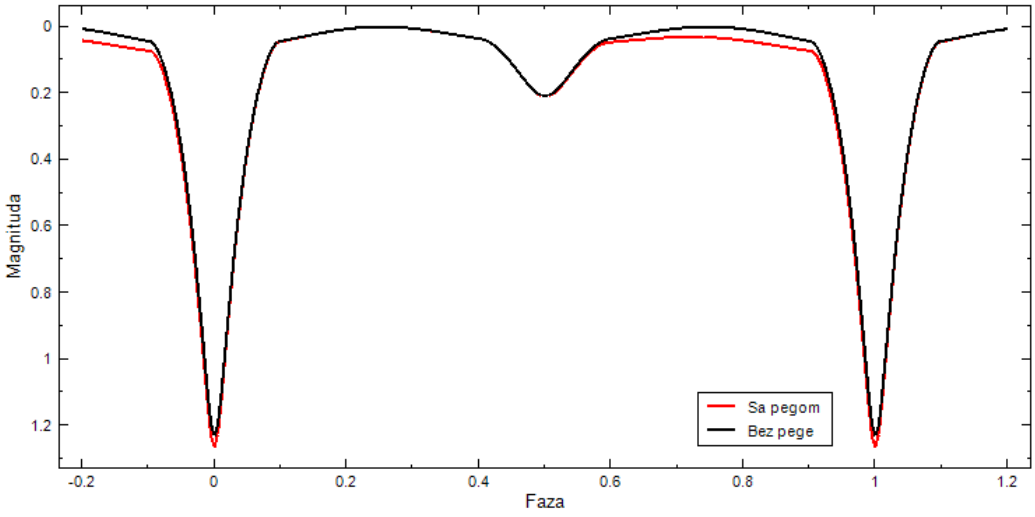
Jednačina 5.15

$$\cos R_p < \cos l,$$

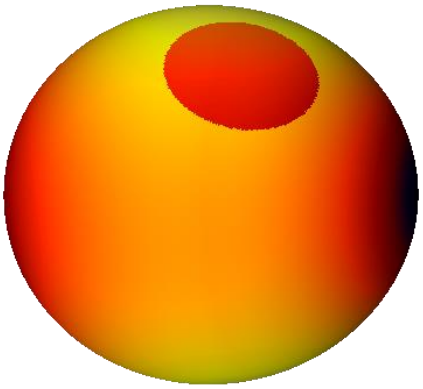
i u tom slučaju vršimo korekciju temperature ćelije uz pomoć jednačine 5.13. U programu ne postoji ograničenje za broj pega na komponentama.



Slika 5.5 Geometrijski model pege. Preuzeto iz Đurašević (1992a).



Slika 5.6 Uticaj tamne pege na krivu sjaja. Pega se nalazi na primarnoj komponenti. Njeni parametri su $\theta_p = 45°$, $\varphi_p = 70°$, $R_p = 29.8°$ i $K_p = 0.95$.



Slika 5.7 Zvezdana pega. Efektivna temperatura zvezde je $T_{eff} = 6000\ K$. Temperaturni kontrast pege je $K_p = 0.95$.

5.4. Akrecioni disk

Pod određenim uslovima se u tesno dvojnom sistemu može formirati akrecioni disk. Ako jedna komponenta u toku evolucije ispuni svoj kritični Roche oval, postaje nestabilna i počinje da gubi masu kroz Lagrange-ovu tačku L1. Formira se gasna struja koja po spiralnoj putanji pada na površinu druge komponente. Ukoliko je intenzitet gasne struje mali, materija će direktno padati na površinu druge komponente. U suprotnom će se formirati akrecioni disk (Lubow & Shu, 1975). Kretanje materije u akrecionom disku je blisko kružnom keplerovskom kretanju (ugaona brzina opada od centra ka rubu diska). Usled unutrašnjeg trenja, kinetička energija gasa se transformiše u toplotnu i materija po spiralnoj putanji pada na zvezdu. To znači da je potreban konstantni priliv materije sa komponente koja je ispunila Roche oval da bi se obezbedila stabilnost diska tokom dužeg vremenskog perioda.

Jednostavnosti radi pretpostavljamo da se akrecioni disk uvek nalazi oko primarne komponente i da sekundarna komponenta ispunjava svoj Roche oval. Geometriju akrecionog diska opisujemo poprečnim profilom kao što je objašnjeno u odeljku 3.2. Ovakva reprezentacija je opravdana poslednjim rezultatima iz hidrodinamičkog transfera mase u tesno dvojnim sistemima – videti Bisikalo et al. (2000) ili Harmanec et al. (2002). Promenu temperature od centra ka rubu diska opisujemo jednom od četiri raspodele iz Đuraševićevog modela izložene u Đurašević et al. (2008). One su date sledećim izrazima:

Jednačina 5.16

$$T(\rho) = T_{out}\left(\frac{R_{out}}{\rho}\right)^{a_T}\left(1-\sqrt{\frac{R_{in}}{\rho}}\right)^{\frac{1}{4}}\left(1-\sqrt{\frac{R_{in}}{R_{out}}}\right)^{-\frac{1}{4}}, \qquad (1)$$

$$T(\rho) = T_{out}\left(\frac{R_{out}}{\rho}\right)^{a_T}, \qquad (2)$$

$$T(\rho) = T_{out} + (T_{in} - T_{out})\left(1-\frac{\rho - R_{in}}{R_{out} - R_{in}}\right)^{a_T}, \qquad (3)$$

$$T(\rho) = T_{out} + (T_{in} - T_{out})\left(1-\left(\frac{\rho - R_{in}}{R_{out} - R_{in}}\right)^{a_T}\right). \qquad (4)$$

$\rho = \sqrt{x^2 + y^2}$ je radijalna udaljenost elementarne ćelije od centra diska, R_{in} je unutrašnji radijus, a R_{out} spoljašnji. T_{in} je temperatura unutrašnje ivice diska, T_{out} spoljašnje, dok je a_T temperaturni eksponent.

Aktivni regioni na disku se modeliraju pegama. One su smeštene na rubu diska i prostiru se duž cele njegove debljine (slika 5.8). Na primer, na mestu gde gasna struja pada na akrecioni disk dolazi do nagomilavanja i zagrevanja materije, što dovodi do narušavanja osno-simetrične raspodele zračenja. Veličina i temperatura ove svetle pege zavise od početnih parametara gasne struje, kao i od parametara sistema. Dodatne pege se mogu upotrebiti za simuliranje drugih vrsta nehomogenosti u disku, kao što je postojanje spiralnih grana (Heemskerk, 1994), regiona na disku koji odstupaju od osne simetrije i sličnih pojava.

Pega se karakteriše sledećim parametrima - longitudom centra pege φ_a, uglovnim radijusom R_a i temperaturnim kontrastom pege K_a.

Da bi se odredilo da li elementarna ćelija pripada pegi potrebno je izračunati longitudu φ_i centroida elementarne ćelije na rubu diska. Ukoliko je ispunjen uslov

Jednačina 5.17

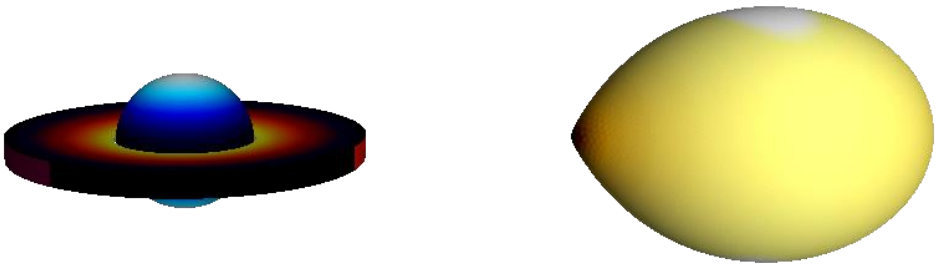
$$\varphi_a - \frac{R_a}{2} < \varphi_i < \varphi_a + \frac{R_a}{2}$$

korigujemo njenu temperaturu za

Jednačina 5.18

$$T_a = K_a T_i,$$

gde je T_i temperatura koju bi imala ćelija da nema pege.



Slika 5.8 Akrecioni disk sa pegama na rubu.

5.5. Računanje izlaznog fluksa

Kao što smo videli u prethodnim odeljcima, kod komponenata dvojnog sistema postoji raspodela temperature i gravitacionog ubrzanja na površini zvezde. To znači da je nemoguće precizno opisati zračenje komponenata koje stiže do posmatrača jednim modelom zvezdane atmosfere. Umesto toga se svakoj ćeliji dodeljuje model atmosfere koji odgovara lokalnim vrednostima temperature i gravitacionog ubrzanja. Za svaku ćeliju se izračuna specifični intenzitet izlaznog zračenja $I(\cos\gamma, T, g, \lambda)$, gde je γ ugao između pravca ka posmatraču i normale na elementarnu ćeliju. Ukupni monohromatski izlazni fluks sa komponente k je onda po definiciji:

Jednačina 5.19

$$F_k = \int I_\lambda(\cos\gamma, T, g)\cos\gamma \, dS,$$

gde se integracija vrši po vidljivim elementarnim ćelijama. Kada integral zamenimo konačnom sumom ovaj izraz postaje:

Jednačina 5.20

$$F_k = \sum_i \chi_i I_{\lambda i}(\cos\gamma, T, g)\cos\gamma \, S_i = \sum_i \chi_i I_{\lambda i}(\cos\gamma, T, g)\, S_i',$$

gde je χ_i mapa vidljivosti (poglavlje 4), a S_i' površina projekcije elementarne ćelije na ravan nebeske sfere. Ovde se sumiranje vrši po svim elementarnim ćelijama.

U Đuraševićevom programu se pretpostavlja da elementarne ćelije zrače kao crno telo. Njihovo zračenje je opisano Planck-ovim zakonom:

Jednačina 5.21

$$B_\lambda(T) = \frac{2hc}{\lambda^5}\frac{1}{e^{\frac{hc}{\lambda kT}} - 1}.$$

Za talasnu dužinu λ u gornjem izrazu se uzima efektivna talasna dužina korišćenog fotometrijskog filtera. To znači da se fotometrijski filter aproksimira delta funkcijom koja propušta zračenje samo na toj talasnoj dužini. Ovo je dosta gruba pretpostavka i u sledećem odeljku će biti predstavljen model koji bolje opisuje uticaj filtera na izračunatu krivu sjaja.

Fotometrijski filteri se opisuju krivom propusne moći. Ona nam govori koliki procenat upadnog zračenja prolazi kroz filter na datoj talasnoj dužini. Krive za standardne filtere se mogu naći u Moro & Munari (2000). Mnogo realniji pristup sintezi krive sjaja je da se umesto monohromatskog specifičnog intenziteta koristi ukupni intenzitet koji prolazi kroz filter, odnosno:

Jednačina 5.22

$$I(\cos\gamma, T, g) = \int I_\lambda(\cos\gamma, T, g)\, P(\lambda) d\lambda,$$

gde je $P(\lambda)$ kriva propusne moći filtera.

Proces sinteze spektara je jako spor i zato je poželjno specifične intenzitete unapred izračunati za svaki podržani filter (spisak podržanih filtera se nalazi u tabeli 2). U *Infinity*-ju je implementiran postupak opisan u Van Hamme & Wilson (2003). Ovde je ukratko prikazan metod dobijanja tablica sa centralnim specifičnim intenzitetom (za $\cos\gamma = 1$) za proizvoljan filter, dok se detaljniji opis može naći u Cséki & Prša (2010).

Tabela 2 Spisak podržanih fotometrijskih filtera

Set	Filter
CoRoT	CoRoT
Cousins	R
	I
Johnson	U
	B
	V
	J

Set	Filter
	K
Kepler	Kepler
Strömgren	U
	V
	B
	y

Svakoj elementarnoj ćeliji se pripisuje jednodimenzioni model zvezdane atmosfere izveden pod pretpostavkom lokalne termodinamičke ravnoteže koji odgovara lokalnim vrednostima temperature i efektivne gravitacije. Koristio sam mrežu modela koju su izračunali Castelli & Kurucz (2004) i koja pokriva opseg efektivnih temperatura od 3500K do 50000K, logaritma gravitacionog ubrzanja ($\log g$) od 0.0 do 5.0 i metaličnosti od -2.5 do +1.0. Za svaki model iz mreže sam uz pomoć programa SPECTRUM (Gray & Corbally, 1994) izračunao sintetičke spektre u opsegu od 3000A do 12000A, pomnožio sa krivom propusne moći filtera i dobijenu raspodelu integralio po celom opsegu talasnih dužina (jednačina 5.22). Za svaki par metaličnost - $\log g$ se nacrta

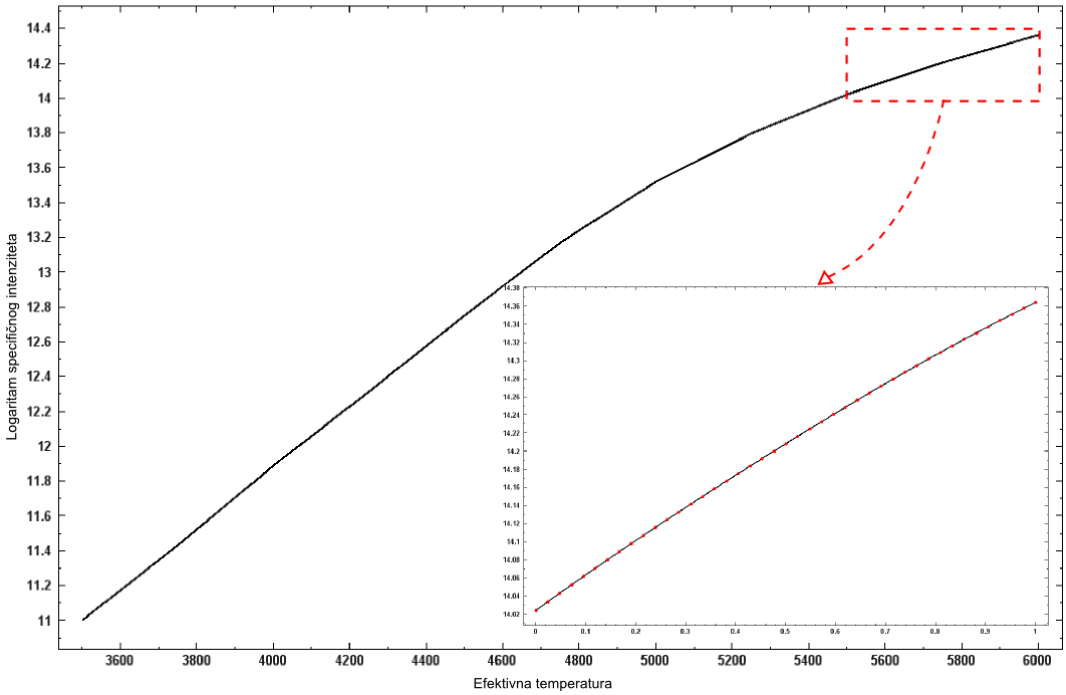
zavisnost centralnog intenziteta od temperature modela i tako dobijeni grafik se podeli na četiri temperaturska intervala. Svaki od intervala se fituje sledećim polinomom:

Jednačina 5.23

$$\sum_{i=0}^{9} a_i P_i\left(\frac{T - T_{low}}{T_{high} - T_{low}}\right),$$

gde je P_i Legendre-ov polinom i-tog stepena, T_{low} i T_{high} donja i gornja granica intervala i a_i koeficijenti polinoma određeni fitovanjem. Optimalne granice intervala su određene Monte Karlo metodom. Na slici 5.9 je prikazan primer fita za $\log g = 0.5$ i $\left[\frac{M}{H}\right] = -2.5$. Ovako izračunate koeficijente možemo sačuvati u datoteku.

Prednost ovakvog pristupa je to što se složeni problem računanja specifičnog intenziteta svodi na jednostavno učitavanje koeficijenata iz tabele. Takođe, interpolacija se vrši samo po $\log g$ tako da je ova procedura izuzetno efikasna.



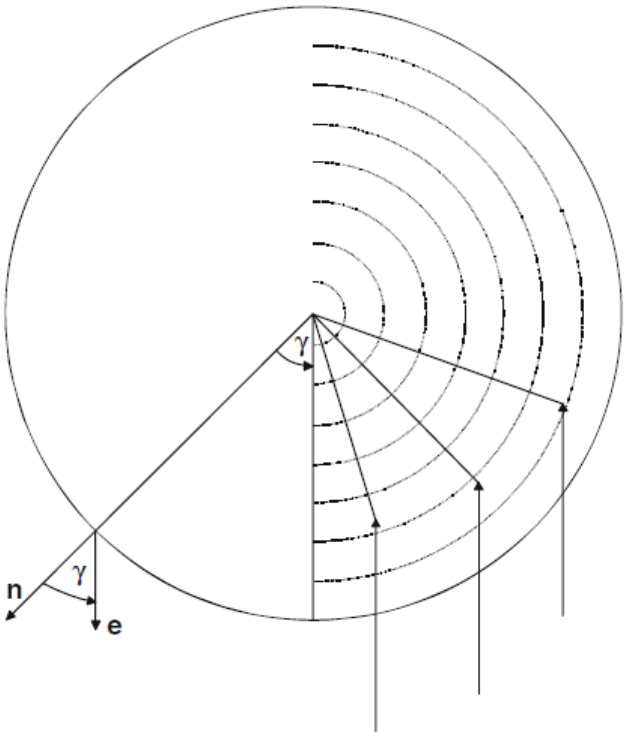
Slika 5.9 Zavisnost specifičnog intenziteta od efektivne temperature za $\log g = 0.5$ i $\left[\frac{M}{H}\right] = -2.5$ u Johnson V filteru. Zumirani deo prikazuje fit Legendre-ovim polinomima.

5.5.1. Potamnjenje ka rubu

Zavisnost specifičnog intenziteta od položaja na disku zvezde se naziva potamnjenje ka rubu. Ono je dobro proučeno kod Sunca i direktno se posmatra kod nekih drugih zvezda (npr. Sasselov (1998)). U pitanju je geometrijski efekat prikazan na slici 5.10. Iz centra diska zvezde vidimo zračenje iz dubljih slojeva nego na rubu, odnosno iz slojeva sa višom temperaturom, pa se primećuje opadanje intenziteta zračenja od centra ka rubu. U ovom modelu dvojnog sistema, potamnjenje ka rubu se opisuje semi-empirijskom jednačinom 5 iz rada Claret & Bloemen (2011):

Jednačina 5.24

$$D(\cos\gamma) = \frac{I(\cos\gamma)}{I(1)} = 1 - \sum_{k=1}^{1} a_k (1 - (\cos\gamma)^{\frac{k}{2}}).$$



Slika 5.10 Potamnjenje ka rubu. Izračeni specifični intenzitet zavisi od ugla između pravca ka posmatraču i normale na elementarnu površ. Preuzeto iz Kallrath & Milone (2009).

Rezultati ovog rada (Claret & Bloemen, 2011), koji daje tablice sa izračunatim koeficijentima a_k za sve podržane filtere (vidi tabelu 2), pokazuju da ova aproksimacija najbolje fituje teorijske vrednosti za specifične intenzitete na celom opsegu podržanih

temperatura (od 3000K do 50000K) i gravitacionog ubrzanja (od $\log g = 0.0$ do $\log g = 5.0$). Po potrebi je moguće koristiti i ostale zakone potamnjenja ka rubovima - linearni, kvadratni, koreni i logaritamski (jednačine 1-4 iz Claret & Bloemen (2011)).

Koeficijente a_k za svaku elementarnu ćeliju dobijamo bilinearnom interpolacijom po T_l i $\log g_l$ iz Claret-ovih tablica i onda iz jednačine 5.23 određujemo popravku usled potamnjenja ka rubu. U prethodnom odeljku je opisano kako se računa centralni specifični intenzitet. Konačno, iz jednačine 5.20 dobijamo izraz za fluks koji sa komponente k posmatramo kroz dati fotometrijski filter

Jednačina 5.25

$$F_k = \sum_i \chi_i I_{0i}(T, g) D_i(\cos\gamma)\ S_i'.$$

Ukupni fluks izračen u datoj fazi će biti:

Jednačina 5.26

$$F = F_{prim} + F_{sec} + F_{disc}.$$

Ovaj fluks normiramo na maksimalnu vrednost fluksa u fazi 0.25.

5.6. Računanje radijalnih brzina

Komponente dvojnog sistema se kreću oko zajedničkog centra mase. Dok se jedna zvezda približava posmatraču, druga se udaljava i obrnuto, što dovodi do pomeranja spektralnih linija usled Doppler-ovog efekta. Šematski prikaz ove pojave je dat na slici 5.11.

Izmerena radijalna brzina je rezultat superpozicije nekoliko kretanja:

- sopstveno kretanje centra mase sistema
- kretanje centra zvezde oko centra mase sistema
- kretanje elementarnih ćelija na površini zvezde oko centra zvezde

Brzina kretanja centra mase je parametar modela i opisuje se konstantnom brzinom γ.

Brzina centra zvezde u odnosu na centar mase je:

Jednačina 5.27

$$\overrightarrow{v_c} = \overrightarrow{r_c} \times \overrightarrow{\omega_k},$$

gde je $\overrightarrow{r_c}$ radijus-vektor centra komponente (odeljak 2.3). Ugaona brzina revolucije sistema je:

Jednačina 5.28

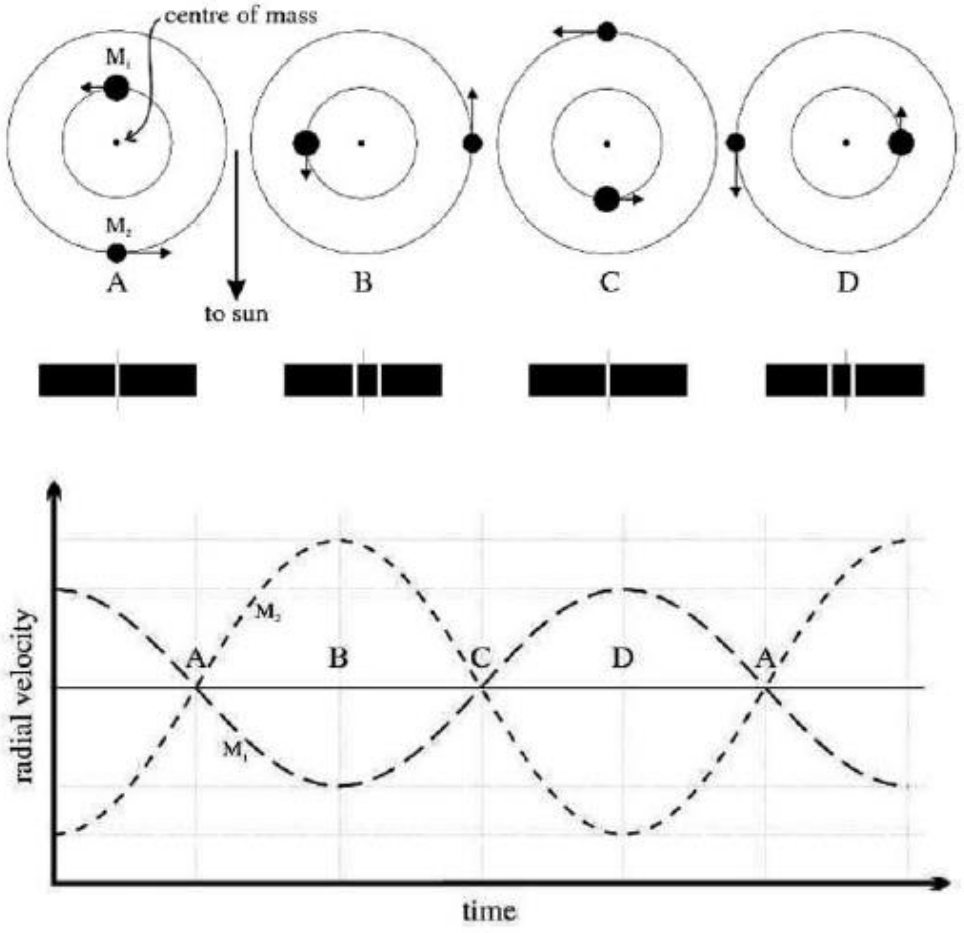
$$\overrightarrow{\omega_k} = \frac{2\pi}{86400\ P}\overrightarrow{e_z},$$

gde je P period u danima.

Elementarna ćelija se kreće u odnosu na centar zvezde ugaonom brzinom $\omega = f\omega_k$ (jednačina 3.10), pa je njena brzina:

Jednačina 5.29

$$\overrightarrow{v_i} = \overrightarrow{r_i} \times f\overrightarrow{\omega_k}.$$



Slika 5.11 Kriva radijalnih brzina. Preuzeto iz Kallrath & Milone (2009).

Ukupni doprinos brzine elementarnih ćelija ukupnoj radijalnoj brzini se dobija usrednjavanjem po celoj vidljivoj površini u zadatoj fazi:

Jednačina 5.30

$$\overrightarrow{v_*} = \frac{\sum_i \overrightarrow{v_i} S_i}{\sum_i S_i}.$$

Ovaj doprinos je najveći kada je komponenta delimično pomračena. Van pomračenja se pojedinačni doprinosi poništavaju, dok je to poništavanje tokom pomračenja samo delimično. Ovaj efekat se naziva Rossiter-ovim efektom (Rossiter, 1924).

Konačno za radijalnu brzinu dobijamo:

Jednačina 5.31

$$r_v = (\overrightarrow{v_c} + \overrightarrow{v_*})\overrightarrow{r_o} + \gamma,$$

gde je $\overrightarrow{r_o} = (1, 0, 0)$ jedinični vektor koji je paralelan sa pravcem ka posmatraču.

Na slici 5.12 se može videti primer izračunate krive radijalnih brzina za sistem V455 Cyg. Parametri sistema su preuzeti iz rada Đurašević et al. (2012) u čijoj izradi sam učestvovao.

5.7. Parametri modela

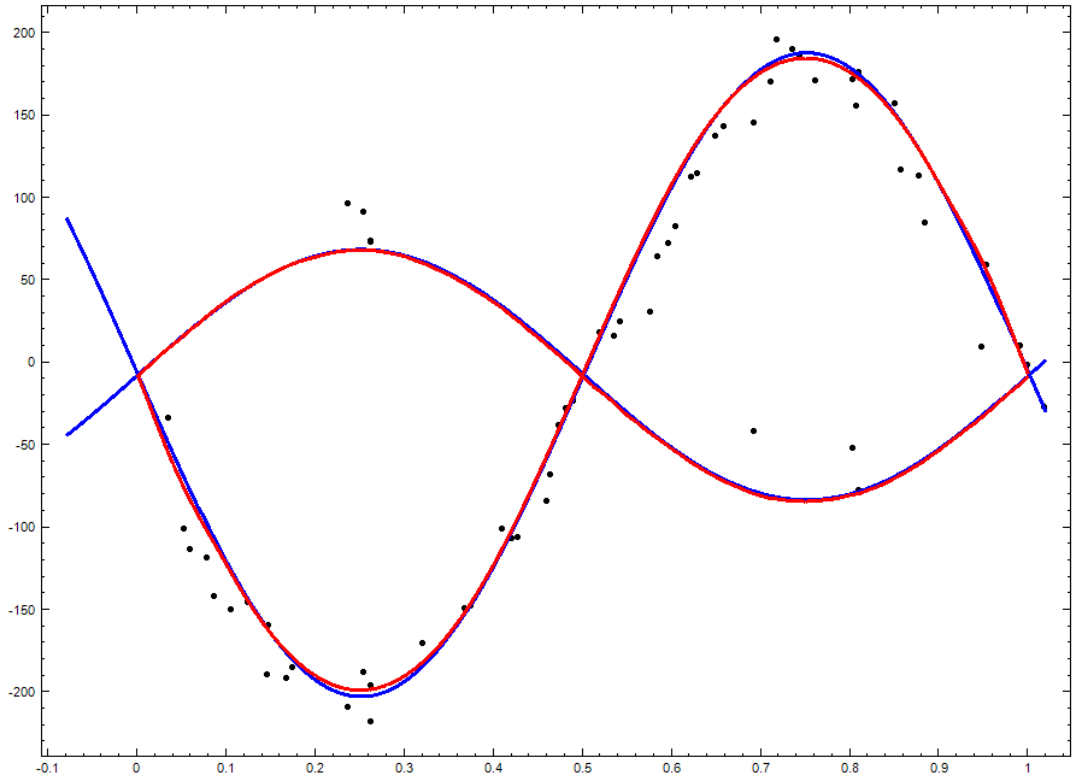
Kao rezultat pokretanja programa se dobijaju sintetičke krive sjaja za svaki fotometrijski filter i krive radijalnih brzina za primarnu i sekundarnu komponentu. Oblik ovih krivih zavisi od sledećih parametara:

1. Parametri dvojnog sistema.
   - $q = \frac{m_2}{m_1}$ - odnos masa komponenata
   - P – period sistema u danima
   - a – velika poluosa sistema u radijusima Sunca
   - i – inklinacija orbite
   - e – ekscentričnost orbite
   - ω – argument periastrona
   - γ – brzina centra mase
   - $l_3/(l_1 + l_2 + l_3)$ – doprinos trećeg svetla u datom fotometrijskom filteru

2. Parametri kojim se opisuju zvezde. Indeksi 1 i 2 odgovaraju primarnoj i sekundarnoj komponenti.
   - r_{p1}, r_{p2} – polarni radijus komponente
   - T_1, T_2 – efektivna temperatura
   - f_1, f_2 – parametar asinhronosti
   - β_1, β_2 – eksponent gravitacionog potamnjenja
   - A_1, A_2 – albedo
   - $[M/H]_1, [M/H]_2$ – metaličnost
3. Parametri kojima se opisuje pega na zvezdi. Nema ograničenja za broj pega koji se može modelirati.
   - φ_p - longituda centra pege
   - θ_p - latituda centra pege
   - R_p - uglovni radijus
   - K_p temperaturni kontrast pege
4. Parametri koničnog diska.
   - r_{in} – unutrašnji radijus
   - r_{out} – spoljašnji radijus
   - d_{in} – debljina unutrašnjeg prstena
   - d_{out} – debljina spoljašnjeg prstena
   - T_{in} – temperatura unutrašnjeg prstena
   - T_{out} – temperatura spoljašnjeg prstena
   - a_T - temperaturni eksponent
5. Parametri toroidalnog diska.
   - r – udaljenost centra elipse koja opisuje poprečni presek od centra primarne komponente
   - a – velika poluosa elipse koja opisuje poprečni presek
   - b – mala poluosa elipse koja opisuje poprečni presek
   - T_{in} – temperatura unutrašnjeg prstena
   - T_{out} – temperatura spoljašnjeg prstena
   - a_T - temperaturni eksponent

6. Parametri koji opisuju aktivne regione na disku. Kao i kod zvezdanih pega, nema ograničenja na njihov broj.
    - φ_a - longituda centra pege
    - R_a - uglovni radijus
    - K_a temperaturni kontrast pege

Kao što se vidi, model zavisi od jako velikog broja parametara. Prilikom određivanja optimalne krive sjaja koja odgovara posmatranjima poželjno je fiksirati što veći broj parametara iz nezavisnih merenja ili na osnovu teorijskih vrednosti. Ostali parametri se određuju rešavanjem inverznog problema.



Slika 5.12 Krive radijalnih brzina za sistem V455 Cyg. Crnim kružićima su označena posmatranja. Plavom linijom je nacrtano rešenje dobijeno pomoću Wilson-Devinney modela objavljeno u Đurašević et al. (2012), a crvenom krive radijalnih brzina dobijene pomoću programa *Infinity*.

6. Rešavanje inverznog problema

U prethodnim glavama je opisano rešavanje direktnog problema – računanje sintetičke krive sjaja i radijalnih brzina za zadate parametre modela. Da bi se model uspešno primenio na posmatranja, potrebno je rešiti obrnuti zadatak – odrediti set optimalnih parametara koji pokazuju najbolje slaganje sa posmatranjima. U *Infinity*-ju se u tu svrhu koristi Nelder-Mead simpleks algoritam. Za procenu kvaliteta slaganja modela i posmatranja se koristi suma kvadrata razlike između posmatranih i sintetičkih krivih, odnosno traži se takav set parametara modela koji će dati minimalnu vrednost ove sume:

Jednačina 6.1

$$\sum_i (O_i - C_i)^2.$$

Posmatranja obično sadrže veliki broj merenja. U većini slučajeva se broj tačaka meri hiljadama. Sa druge strane, sintetičke krive su glatke funkcije i stoga je opravdano da se one izračunaju na manjem skupu tačaka i da se nedostajuće vrednosti dobiju interpolacijom. U većini slučajeva je dovoljno između 100 i 200 ekvidistantnih tačaka da bi se u potpunosti opisala posmatranja. Još jedna prednost ovog pristupa je da ako imamo multikolor fotometriju nije potrebno računati mapu vidljivosti za svaki filter ponaosob – samo izlazni fluks. Ovo dovodi do značajnog ubrzanja računanja sintetičke krive sjaja. U *Infinity*-ju se koristi Neville-ov algoritam za interpolaciju (Press, et al., 2007).

6.1. Nelder-Mead simplex

Nelder-Mead simpleks (Nelder & Mead, 1965) je algoritam za nelinearnu optimizaciju koji je našao široku primenu u astrofizici. Njegova glavna prednost je to što nije neophodno računati izvode funkcije koja se minimizuje. Opširan opis metode se može naći u radu Takahama & Sakai (2005), dok je ovde dat kratak pregled.

Ako funkcija f koju minimizujemo zavisi od N parametara, simpleks je politop dimenzije $N+1$ u prostoru parametara. Za $N=2$ geometrijska reprezentacija simpleksa je trougao, za $N=3$ tetraedar, itd. Svako teme simpleksa x_k predstavlja

jedno rešenje direktnog problema. U svakoj iteraciji se odredi novo teme na osnovu vrednosti ostalih temena i njime se zameni najgore rešenje iz prethodne iteracije. Ovaj postupak se ponavlja sve dok se ne postigne željena konvergencija.

Neka su x_l, x_h i x_s, redom, najbolje, najgore i drugo najgore rešenje. Teme x_h će biti zamenjeno na kraju iteracije. Da bi se odredilo novo teme, definiše se centroid svih temena osim x_h kao:

Jednačina 6.2

$$x_0 = \frac{1}{N}\sum_{k \neq h} x_k.$$

Nad ove četiri tačke se definišu sledeće operacije:

- *Refleksija* generiše novo teme x_r kao refleksiju temena x_h oko x_0:

Jednačina 6.3

$$x_r = (1 + a)x_0 - ax_h \quad (a > 0).$$

- *Kontrakcija* generiše novo teme x_c između temena x_h i x_0:

Jednačina 6.4

$$x_c = bx_h + (1 - b)x_0 \quad (0 < b < 1).$$

- *Ekspanzija* generiše novo teme x_e na pravcu od x_0 do x_r:

Jednačina 6.5

$$x_e = cx_r + (1 - c)x_0 \quad (c > 1).$$

- *Redukcija* zamenjuje sva temena simpleksa x_k sa:

Jednačina 6.6

$$x_k = \frac{1}{2}(x_k + x_l).$$

Algoritamska šema se onda može predstaviti na sledeći način:

Korak 0 – Generiše se početni simpleks.

Korak 1 – Odrede se tačke x_l, x_h, x_s i x_0.

Korak 2 – Izvrši se refleksija i izračuna se x_r. Ako je x_r bolje od x_l ($f(x_r) < f(x_l)$), ide se na korak 3, u suprotnom na korak 4.

Korak 3 – Izvrši se ekspanzija i izračuna se x_e. Ako je x_e bolje od x_r, x_h se zamenjuje sa x_e. U suprotnom, x_h se zamenjuje sa x_r. Povratak na korak 1.

Korak 4 – Ako je x_r bolje ili jednako od x_s, x_h se zamenjuje sa x_r. i vraća se na korak 1. U suprotnom se ide na korak 5.

Korak 5 –Ako je x_r bolje od x_h, x_h se zamenjuje sa x_r.

Korak 6 – Izvrši se kontrakcija i izračuna se x_c. Ako je x_c bolje od x_h, x_h se zamenjuje sa x_c. U suprotnom se izvrši redukcija za sva temena osim x_l. Povratak na korak 1.

Kriterijum za konvergenciju je da standardna devijacija δ vrednosti funkcije f za sva temena postane dovoljno mala:

Jednačina 6.7

$$\delta = \sqrt{\frac{1}{N+1}\sum_k \left(f(x_k) - \bar{f}\right)^2},$$

$$\bar{f} = \frac{1}{1+N}\sum_k f(x_k).$$

6.2. Primena na posmatranja

6.2.1. EG Cep

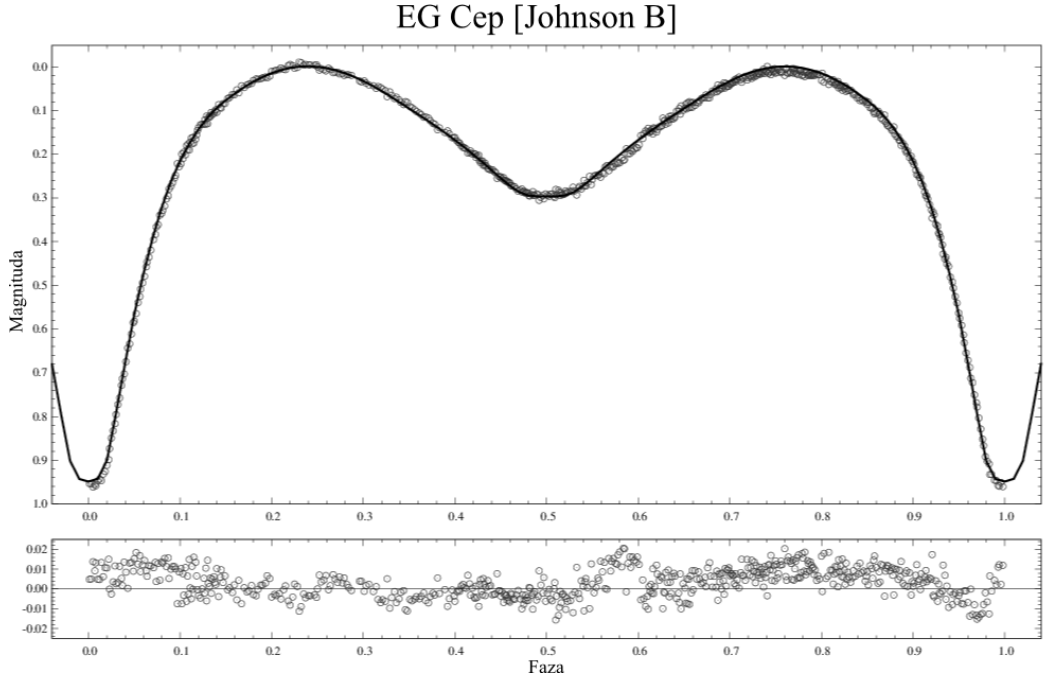
EG Cep (HD 194089) je eklipsna dvojna zvezda koju je otkrio Strohmeier (1958). U pitanju je aktivni tesno dvojni sistem koji je od otkrića predmet brojnih studija. Poslednja studija u nizu je sprovedena na Astronomskoj opservatoriji i objavljena je u Đurašević et al. (2013). Istorija istraživanja ovog interesantnog sistema je prikazana u tom radu. Zaključak rada je da je u pitanju poluodvojeni sistem u kome sekundarna komponenta ispunjava svoj Roche oval. Prisustvo svetle pege na primarnoj komponenti na mestu na kome se može očekivati pad gasne struje sa sekundara ukazuje da ovaj sistem trenutno prolazi kroz fazu razmene materije između komponenata.

Fotometrijska posmatranja u B , V i R_C filterima su dobijena sa opservatorije Univerziteta u Ankari uz pomoć Schmidt-Cassegrain teleskopa prečnika 40cm sa Apogee ALTA U47 CCD kamerom, dok su spektroskopski elementi preuzeti iz rada Rucinski et al. (2008). Autori su koristili Đuraševićev program za interpretaciju krivih

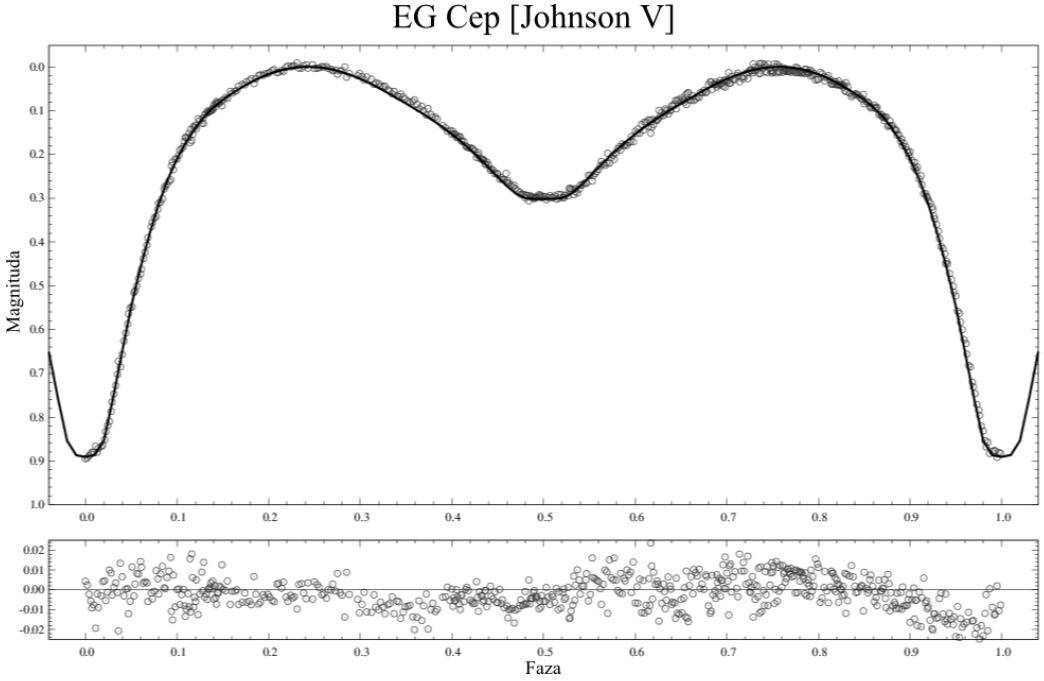
sjaja. U cilju testiranja modela predloženog u ovoj disertaciji ponovio sam analizu fotometrijskih posmatranja.

Posmatranja za sva tri filtera su simultano fitovana. Postupak analize i određivanja grešaka prati rad Đurašević et al. (2013). Fiksirani su sledeći parametri: odnos masa, temperatura primarne komponente određena spektroskopski, radijus sekundarne komponente, albeda i parametri asinhronosti za obe komponente, dok su ostali parametri tretirani kao slobodni. Greške prikazane u tabeli su formalne i predstavljaju donju granicu greške parametara. Dobijene su kao maksimalno odstupanje vrednosti parametara prilikom variranja odnosa mase između minimalne i maksimalne vrednosti ($q = 0.459$ i $q = 0.469$).

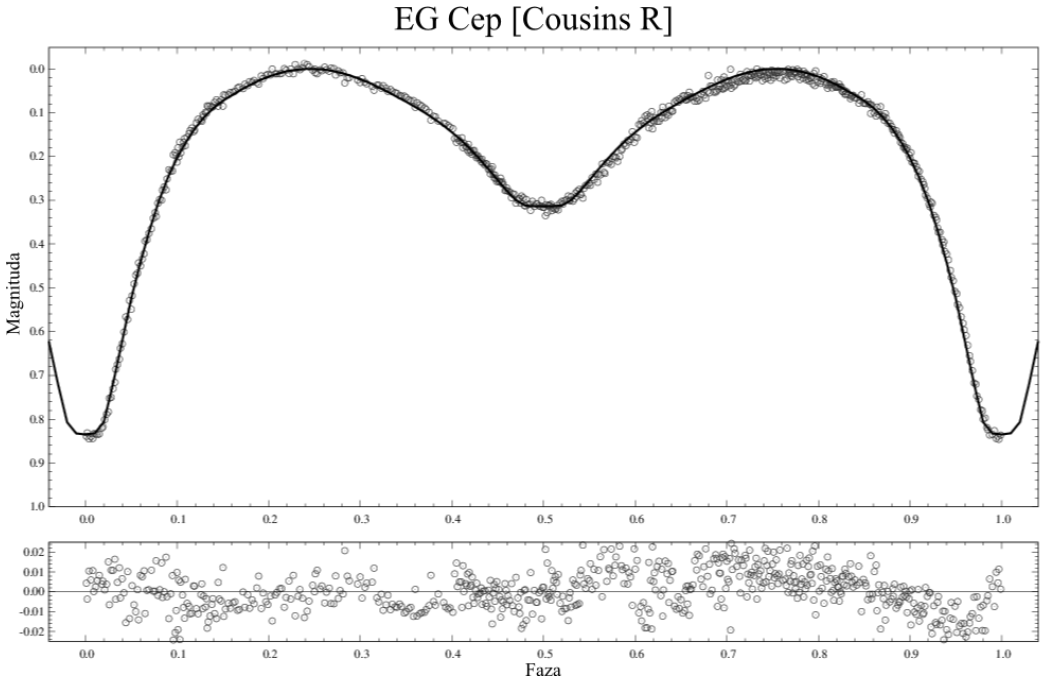
Pregled rezultata i poređenje sa Đuraševićevim programom je dat u tabeli 3. Dobijeni parametri sistema su u dobroj saglasnosti sa rezultatima iz Đurašević et al. (2013) i nalaze se unutar granica greške. Krive sjaja i reziduali za pojedinačne filtere su prikazani na slikama 6.1 - 6.3.



Slika 6.1 Johnson B filter



Slika 6.2 Johnson V filter



Slika 6.3 Johnson-Cousins R filter

Tabela 3 Parametri sistema EG Cep

	Infinity		Đurašević et al. (2013)	
Parametri dvojnog sistema				
q	0.464 ± 0.005		0.464 ± 0.005	
$P\ [d]$	0.5446226		0.5446226	
$a_{orb}\ [R_\odot]$	3.76 ± 0.02		3.76 ± 0.02	
$i\ [°]$	87.3 ± 0.4		87.5 ± 0.2	
$l_3/(l_1+l_2+l_3)\ [B]$	0.001 ± 0.005		0.000 ± 0.005	
$l_3/(l_1+l_2+l_3)\ [V]$	0.007 ± 0.005		0.007 ± 0.005	
$l_3/(l_1+l_2+l_3)\ [R_C]$	0.012 ± 0.005		0.013 ± 0.005	
Zvezdani parametri	Primar	Sekundar	Primar	Sekundar
$r_p\ [a_{orb}]$	0.416 ± 0.003	0.293	0.41 ± 0.02	0.29 ± 0.02
$T_{eff}\ [K]$	7850	5400 ± 100	7850	5360 ± 20
f	1.0	1.0	1.0	1.0
β	0.25	0.342 ± 0.02	0.25 ± 0.01	0.34 ± 0.02
A	1.0	0.5	1.0	0.5
Parametri pege				
$\varphi_p\ [°]$	359.8 ± 0.6		0.0 ± 0.5	
$\theta_p\ [°]$	80 ± 5		81 ± 2	
$R_p\ [°]$	31.9 ± 0.8		29.8 ± 0.5	
K_p	1.04 ± 0.01		1.04 ± 0.02	
Apsolutni parametri				
$m\ [m_\odot]$	1.64 ± 0.05	0.76 ± 0.09	1.64 ± 0.02	0.77 ± 0.02
$R\ [R_\odot]$	1.67 ± 0.01	1.17 ± 0.01	1.66 ± 0.02	1.18 ± 0.02
$\log_{10} g$	4.17 ± 0.01	4.15 ± 0.01	4.22 ± 0.02	4.18 ± 0.02
M_{bol}	2.37 ± 0.01	4.69 ± 0.03	2.36 ± 0.02	4.75 ± 0.03

6.2.2. AU Mon

AU Mon (HD 50846) je eklipsna, spektroskopski dvojna zvezda. Primarna komponenta je Be zvezda, dok sekundarna pripada spektralnom tipu G. Ovaj sistem pripada klasi novootkrivenih dvojnih zvezda sa duplim periodom (Mennickent, et al., 2003), što znači da je pored orbitalnog perioda od oko 11 dana posmatrana i dodatna promenljivost sjaja sa periodom od oko 417 dana i amplitudom 0.25 mag.

Ovaj zanimljiv sistem smo proučavali na Astronomskoj opservatoriji početkom 2010. godine. Naši rezultati su kasnije te godine objavljeni u Đurašević et al. (2010). Analiza je bila zasnovana na javno dostupnim fotometrijskim posmatranjima visoke preciznosti sa satelita CoRoT i spektroskopskim elementima objavljenim u radu Desmet et al. (2010). Naš zaključak je bio da posmatranja najbolje opisuje model u kome se oko

primarne komponente nalazi optički debeo akrecioni disk. Ovaj rezultat je kasnije potvrđen u radu Atwood-Stone et al. (2012) korišćenjem nezavisne metode. Akrecioni disk u našem modelu je konkavnog oblika (proširuje se od centra ka rubu). Dugoperiodične promene se tumače promenama u strukturi akrecionog diska, koja varira sa oscilacijama u brzini transfera mase sa sekundarne komponente. Ovo objašnjenje je u skladu sa prihvaćenim objašnjenjem za druge zvezde sa duplim periodom.

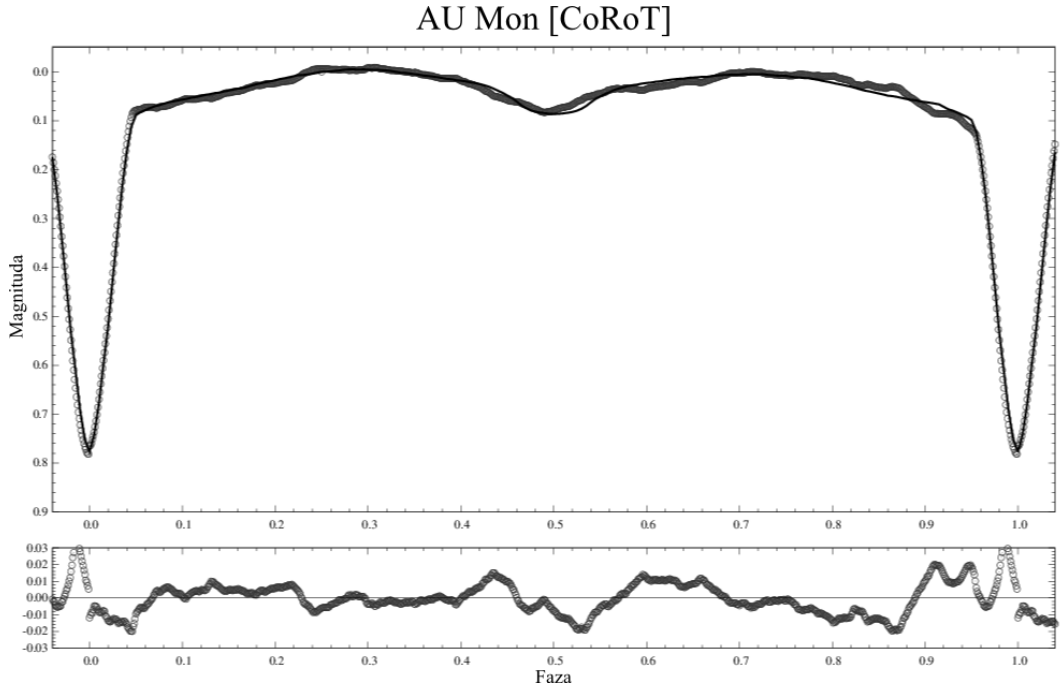
Posmatranja se sastoje od pet uzastopnih orbitalnih ciklusa. Usled malih varijacija u izgledu krive sjaja svaki od njih sam nezavisno analizirao prateći postupak iz rada Đurašević et al. (2010). Fiksirani su sledeći parametri: odnos masa, temperatura sekundarne komponente dobijena iz spektroskopije, temperatura primarne komponente na osnovu preliminarne analize, parametri asinhronosti, kao i albeda i eksponenti gravitacionog potamnjenja, za koje su uzete teorijske vrednosti. Raspodela temperature diska se opisuje Zolinom rapodelom (jednačina 5.16.3), a potamnjenje ka rubu Claret-ovom nelinearnom aproksimacijom četvrtog reda (jednačina 5.24).

U tabeli 4 je prikazano rešenje dobijeno usrednjavanjem vrednosti parametara za svih pet ciklusa. Greške u tabeli predstavljaju maksimalna odstupanja vrednosti pojedinih parametara od srednje vrednosti.

Rezultujući parametri sistema se dobro slažu sa rešenjem dobijenim pomoću Đuraševićevog programa. Akrecioni disk je neznatno veći, kao i temperatura na rubu diska ($\Delta T \approx 200K$). Sa druge strane, temperaturni kontrasti pega na rubu diska su nešto niži. Kriva sjaja i fit za prvi orbitalni ciklus je prikazana na slici 6.4.

Tabela 4 Parametri sistema AU Mon

	Infinity		Đurašević et al. (2010)	
Parametri dvojnog sistema				
q	0.17		0.17	
$P\ [d]$	11.1130374		11.1130374	
$a_{orb}\ [R_\odot]$	42.11		42.11	
$i\ [°]$	80.0 ± 0.5		80.1 ± 0.6	
Zvezdani parametri	Primar	Sekundar	Primar	Sekundar
$r_p\ [a_{orb}]$	0.1166(2)	0.2221	0.1184(3)	0.2221
$T_{eff}\ [K]$	15890	5750	15890	5750
f	5.2	1.0	5.2	1.0
β	0.25	0.08	0.25	0.08
A	1.0	0.5	1.0	0.5
Parametri diska				
$r_{in}\ [a_{orb}]$	0.1166(2)		0.1184(3)	
$r_{out}\ [a_{orb}]$	0.310(1)		0.3016(2)	
$d_{in}\ [a_{orb}]$	0.01		0.01	
$d_{out}\ [a_{orb}]$	0.0366(8)		0.040(5)	
T_{in}	15890		15890	
T_{out}	5400 ± 100		5190 ± 70	
a_T	6.5		6.5	
Parametri pega na disku				
$\varphi_{p1}\ [°]$	315 ± 10		331 ± 6	
$R_{p1}\ [°]$	18.3 ± 0.6		19.7 ± 0.4	
K_{1p}	1.56 ± 0.09		1.7 ± 0.1	
$\varphi_{p2}\ [°]$	191 ± 19		171 ± 13	
$R_{p2}\ [°]$	40 ± 5		33 ± 12	
K_{2p}	1.5 ± 0.3		1.4 ± 0.2	
$\varphi_{p3}\ [°]$	69 ± 23		53 ± 14	
$R_{p3}\ [°]$	49 ± 3		49 ± 6	
K_{3p}	1.18 ± 0.2		1.3 ± 0.1	
Apsolutni parametri				
$m_1\ [m_\odot]$	6.9 ± 0.2		7.0 ± 0.3	
$m_2\ [m_\odot]$	1.2 ± 0.3		1.2 ± 0.2	
$R_1\ [R_\odot]$	4.9 ± 0.5		5.1 ± 0.5	
$R_2\ [R_\odot]$	10.1 ± 0.5		10.1 ± 0.5	
$\log_{10} g_1$	3.87 ± 0.03		3.87 ± 0.05	
$\log_{10} g_2$	2.47 ± 0.03		2.50 ± 0.02	
M_{bol1}	-3.1 ± 0.5		-3.1 ± 0.4	
M_{bol2}	-0.23 ± 0.07		-0.21 ± 0.09	
$R_d\ [R_\odot]$	13.1 ± 0.04		12.7 ± 0.06	
$d_{in}\ [R_\odot]$	0.42 ± 0.01		0.42 ± 0.01	
$d_{out}\ [R_\odot]$	1.54 ± 0.07		1.66 ± 0.05	



Slika 6.4 Prvi orbitalni ciklus iz CoRoT posmatranja

7. Zaključak

U okviru ove disertacije je napisan novi program za modeliranje dvojnih sistema, *Infinity*. Njime je moguće uspešno opisati širok spektar konfiguracija dvojnih sistema – od dobro razdvojenih do tesno dvojnih sistema na eliptičnim ili kružnim orbitama, sistema sa akrecionim diskom i/ili aktivnim regionima na komponentama. Prilikom računanja izlaznog fluksa svakoj elementarnoj ćeliji se prisuje jednodimenzioni Kurucz-ov model zvezdane atmosfere za lokalne vrednosti temperature i gravitacionog ubrzanja. Uzima se u obzir i uticaj krive propusne moći fotometrijskih filtera na izračunatu krivu sjaja. Uz nju se računa i kriva radijalnih brzina, što omogućava dobijanje kompletnog spektrofotometrijskog rešenja primenom Nelder-Mead simpleks algoritma za optimizaciju.

Glavni doprinos disertacije je novi pristup geometrijskoj reprezentaciji dvojnih sistema. Ključna su dva originalna algoritma – adaptivna geodezijska mreža za opis površine zvezde u dvojnom sistemu i inverzni slikarski algoritam za detekciju vidljvosti.

U opštem slučaju, komponente sistema se aproksimiraju poligonom mrežom - skupom povezanih trouglova u trodimenzionom prostoru kojim se opisuje neka površ. Iako se geodezijska mreža koristi i u drugim programima (npr. Hendry & Mochnacki (1992)), ovde je prvi put implementirana adaptivna podela. Da bi se povećala tačnost izračunate krive sjaja, u svakoj fazi se odredi region na koji pada senka druge komponente i u njemu se dodatnim korakom podele povećava broj elementarnih ćelija. Ovo dovodi do mnogo preciznijeg modeliranja pomračenja u dvojnom sistemu.

Korišćenje poligonih mreža je omogućilo i uopštenje geometrijskog modela akrecionog diska. Upotreba rotacione osno-simetrične površi (*lathe mesh*) dovodi do jednostavnog opisa diskova različitih oblika. U *Infinity*-ju su trenutno implementirani konusni i toroidalni disk.

Predloženi algoritam za određivanje vidljivih ćelija u datoj fazi (detekcija vidljivosti) se zasniva na inverznom slikarskom algoritmu iz polja računarske grafike. Njegova najbitnija karakteristika je da ne zahteva nikakve informacije o geometriji dvojnog sistema osim samih poligonih mreža koje je opisuju. To znači da se sve komponente dvojnog sistema (zvezde i disk) mogu tretirati na isti način, što dovodi do velike

fleksibilnosti modela i otvara mogućnost za jednostavno proširenje na egzotičnije konfiguracije kao što su hijerarhiski sistemi, sistemi sa ekscentričnim ili cirkumbinarnim akrecionim diskom, planetarni sistemi itd.

Upravo to je glavni pravac razvoja modela u budućnosti – proširivanje fizičkog opisa dvojnog sistema tako da obuhvati i ove slučajeve. U planu je i dodavanje sinteze spektara dvojnog sistema da bi se proučavao uticaj pomračenja na promenu profila spektralnih linija sa fazom, kao i implementacija alternativnih algoritama za optimizaciju kao što su Levenberg-Marquardt algoritam ili Powell-ov metod. Ovo je bitno sa stanovišta problema pronalaženja globalnog minimuma u parametarskom prostoru, odnosno „pravog“ rešenja, koji još uvek predstavlja otvoreno pitanje ne samo u oblasti proučavanja dvojnih zvezda, već i u polju matematičkog modeliranja uopšte. U idealnom slučaju bi pretraga parametarskog prostora sa dovoljno malim korakom po svakom parametru dovela do pronalaženja globalnog minimuma. U praksi je to nemoguće zbog velikog broja parametara kojim se opisuje dvojni sistem i ogromnog računarskog vremena koje bi takva pretraga zahtevala. Zato su i razvijeni različiti algoritmi za optimizaciju, svaki sa svojim prednostima i manama.

Jedan od nedostataka primenjenog algoritma u *Infinity*-ju (Nelder-Mead simplex) je to što on ne daje procenu grešaka dobijenih parametara. Alternativno, potrebno je razviti neku proceduru za pouzdanu procenu ovih grešaka. Jedna mogućnost je da se napravi pretraga parametarskog prostora koja je ograničena samo na granice pouzdanosti (min-max) fiksiranih parametara za koje znamo greške.

Na kraju bih se osvrnuo na još jednu dimenziju modeliranja dvojnih sistema o kojoj nije bilo reči do sada. Posmatračke tehnike napreduju vrtoglavom brzinom – bez obzira da li je reč o fotometriji ili spektroskopiji. Količina i kvalitet dostupnih posmatranja su rasli eksponencijalno u poslednjoj deceniji i nema naznaka da će se ovaj trend promeniti u bliskoj budućnosti. Ova eksplozija podataka predstavlja veliki izazov za teoriju modeliranja dvojnih sistema. Programi masovne pretrage neba (SuperWASP, HAT, OGLE, MACHO…) proizvode krive sjaja za milione promenljivih zvezda. Da bi se ovi podaci u potpunosti iskoristili potrebno je razviti pouzdane automatske procedure za njihovu klasifikaciju i rešavanje. Postoje značajni napori u tom pravcu koji uključuju

neuronske mreže, genetske algoritme i slične pristupe. Prvi rezultati su ohrabrujući, ali smo još daleko od opšte prihvaćenog i, na kraju krajeva, pouzdanog pristupa.

Sa druge strane, svemirski teleskopi daju neprekidna posmatranja sa tačnošću koja ide ispod jedne milimagnitude. Kvalitet ovih merenja nam omogućava da direktno posmatramo komplikovane fizičke procese kao što su neradijalne pulsacije, Doppler beaming itd. Da bi se ove krive sjaja uspešno interpretirale potrebno je proširiti fizički model tako da obuhvata i ove efekte. Oba ova aspekta dovode do toga da moderni modeli dvojnih sistema postaju sve kompleksniji i kompleksniji programi. Na primer, na sledećih 130 strana je dat listing dela *Infinity*-ja koji služi za rešavanje direktnog zadatka. Zbog obima nisu uključeni grafički interfejs, deo za vizuelizaciju, kao ni pomoćni programi. Ovo predstavlja ozbiljan izazov sa stanovišta programiranja. Modularnost, lakoća proširivanja i čitljivost postaju najvažniji aspekti strukture programa. Kallrath i Milone ističu upravo ove kvalitete u poslednjoj glavi svoje knjige (Kallrath & Milone, 2009) koja se bavi strukturom idealnog programa za modeliranje dvojnih sistema koji bi bio u stanju da odgovori na izazove u narednoj deceniji. Nadam se da je *Infinity* korak ka tom cilju.

8. Infinity - program za modeliranje dvojnih sistema

8.1. Pomoćni objekti

Calc.h

```
#pragma once

#include <cmath>
#include <limits>
#include <exception>

namespace Infinity {

inline int sign(double x)
{ return (x > 0) - (x < 0); }

//Switches the sign of the first argument to match the
//sign of the second argument.
inline double switchSign(double a, double b)
{return b >= 0 ? (a >= 0 ? a : -a) : (a >= 0 ? -a : a);}

inline bool isOdd(int value)
{ return ((value & 1) == 1); }

inline bool isEqual(double x, double y,
    double epsilon = 16 * std::numeric_limits<double>::epsilon())
{ return abs(x - y) < epsilon; }

inline bool isEqual(float x, float y,
    float epsilon = 16 * std::numeric_limits<float>::epsilon())
{ return abs(x - y) < epsilon; }

inline bool isNotEqual(double x, double y,
    double epsilon = 16 * std::numeric_limits<double>::epsilon())
{ return abs(x - y) >= epsilon; }

inline bool isNotEqual(float x, float y,
    float epsilon = 16 * std::numeric_limits<float>::epsilon())
{ return abs(x - y) >= epsilon; }

inline bool isEven(int value)
{ return ((value & 1) == 0); }

inline double rad2deg(double value)
{ return value * 180.0 / M_PI; }

inline float rad2deg(float value)
{ return static_cast<float>(value * 180.0 / PI_F); }

inline double deg2rad(double value)
{ return value * M_PI / 180.0; }

inline float deg2rad(float value)
```

```
49  { return static_cast<float>(value * PI_F / 180.0); }
50
51  inline double pow2(double x)
52  { return x * x; }
53
54  inline float pow2(float x)
55  { return x * x; }
56
57  inline int pow2(int x)
58  { return x * x; }
59
60  inline double pow3(double x)
61  { return x * x * x; }
62
63  inline float pow3(float x)
64  { return x * x * x; }
65
66  inline int pow3(int x)
67  { return x * x * x; }
68
69  inline double pow4(double x)
70  { return x * x * x * x; }
71
72  inline float pow4(float x)
73  { return x * x * x * x; }
74
75  inline int pow4(int x)
76  { return x * x * x * x; }
77
78  inline double pow5(double x)
79  { return x * x * x * x * x; }
80
81  inline float pow5(float x)
82  { return x * x * x * x * x; }
83
84  inline int pow5(int x)
85  { return x * x * x * x * x; }
86
87  inline double powN(double x, int n)
88  {
89      if (n == 0 || x == 1.0)
90          return 1.0;
91
92      if (x == -1.0)
93          return isOdd(n) ? -1.0 : 1.0;
94
95      double result = 1.0;
96
97      int absN = n < 0 ? -n : n;
98      for (int i = 1; i <= absN; i++)
99          result *= x;
100
101     if (n < 0)
102         return 1 / result;
103     else
104         return result;
105 }
```

```
106
107 inline double root2(double x)
108 { return sqrt(x); }
109
110 inline double root3(double x)
111 { return pow(x, 1/3); }
112
113 inline double root4(double x)
114 { return sqrt(sqrt(x)); }
115
116 inline double rootN(double x, double N)
117 { return pow(x, 1/N); }
118
119 // Calculates the factorial (n!).
120 inline unsigned long long int factorial(unsigned int n)
121 {
122     unsigned long long int result = 1;
123
124     for (unsigned int i = 1; i <= n; i++)
125         result *= i;
126
127     return result;
128 }
129
130 //Calculates the double factorial (n!!).
131 inline unsigned long long int doubleFactorial(int n)
132 {
133     unsigned long long int result = 1;
134
135     for (int i = n; i > 1; i -= 2)
136         result *= i;
137
138     return result;
139 }
140
141 //Calculates the binomial coefficient "n over k", where
142 // n and k are integers.
143 inline unsigned long long int binomial(unsigned int n, unsigned int k)
144 {
145     if (n <= 20)
146         return factorial(n) / (factorial(k) * factorial(n - k));
147     else
148         throw std::exception("Result too big for long long int.");
149 }
150
151 //Calculates the rising factorial of a double number x,
152 //x^(n) = x(x+1)(x+2)..(x+(n-1)).
153 inline double risingFactorial(double x, unsigned int n)
154 {
155     if (n == 0)
156         return 1;
157     else
158     {
159         double result = x;
160
161         for (unsigned int i = 1; i < n; i++)
162             result *= (x + i);
```

```
163
164             return result;
165         }
166     }
167
168     //Calculates the falling factorial of a double number x,
169     //(x)_n = x(x-1)(x-2)..(x-(n-1)).
170     inline double fallingFactorial(double x, unsigned int n)
171     {
172         if (n == 0)
173             return 1;
174         else
175         {
176             double result = x;
177
178             for (unsigned int i = 1; i < n; i++)
179                 result *= (x - i);
180
181             return result;
182         }
183     }
184
185     //Calculates the binomial coefficient "x over k",
186     //where x is a double and k is an integer.
187     inline double binomial(double x, unsigned int k)
188     { return fallingFactorial(x, k) / factorial(k); }
```

BrentRootFinder.h

```
1   #pragma once
2
3   #include "RootFinder.h"
4   #include "Calc.h"
5
6   #include <cmath>
7   #include <limits>
8
9   namespace Infinity {
10  class BrentRootFinder : public RootFinder
11  {
12  public:
13      BrentRootFinder(double convergenceTolerance, unsigned short
14          maxEvaluations);
15      virtual ~BrentRootFinder();
16
17      double find(Function& function, double lowerBound,
18          double upperBound, double constant = 0.0);
19      double find(Function& function, Function& derivative,
20          double lowerBound, double upperBound, double constant = 0.0);
21  }; }
```

BrentRootFinder.cpp

```
1   #include "stdafx.h"
2   #include "BrentRootFinder.h"
3
```

```
4   using namespace std;
5   namespace Infinity {
6   BrentRootFinder::BrentRootFinder(double convergenceTolerance,
7       unsigned short maxEvaluations) :
8   RootFinder(convergenceTolerance, maxEvaluations) { }
9
10  BrentRootFinder::~BrentRootFinder() { }
11
12  double BrentRootFinder::find(Function& function, double
13      lowerBound, double upperBound, double constant)
14  {
15      //Implementation from Numerical Recipies, 3rd Edition
16      //(page 454)
17      if (lowerBound >= upperBound)
18          throw std::exception("Bad interval.");
19
20      double low = lowerBound;
21      double high = upperBound;
22      double current = upperBound;
23
24      double lowValue = function(low) - constant;
25      double highValue = function(high) - constant;
26      double currentValue = highValue;
27
28      double accuracy, delta, previousCorrection, correction;
29
30      if (isEqual(lowValue, 0.0))
31          return low;
32
33      if (isEqual(highValue, 0.0))
34          return high;
35
36      if (sign(lowValue) == sign(highValue))
37          throw std::exception("Functions is not bracketed properly.");
38
39      for (unsigned short eval = 2; eval < maxEval; eval++)
40      {
41          if (sign(currentValue) == sign(highValue))
42          {
43              high=low; //Rename a, b, c and
44                        //adjust bounding interval d.
45              highValue = lowValue;
46              correction = current - low;
47              previousCorrection = correction;
48          }
49          if (abs(highValue) < abs(currentValue))
50          {
51              low = current;
52              current = high;
53              high = low;
54              lowValue = currentValue;
55              currentValue = highValue;
56              highValue = lowValue;
57          }
58
59          accuracy = 2.0 * numeric_limits<double>::epsilon() * abs(current) +
60  0.5 * tolerance;
```

```
61          delta = 0.5 * (high - current);
62
63          if (abs(delta) <= accuracy || isEqual(currentValue, 0.0))
64              return current;
65
66          double p, q, r, s; //Local variables used
67                             //for interpolation.
68
69          //Is interpolation necessary?
70          if (abs(previousCorrection) >= accuracy
71              && abs(lowValue) > abs(currentValue))
72          {
73              //Attempt inverse quadratic interpolation.
74              s = currentValue / lowValue;
75              if (isEqual(low, high))
76              {
77                  p = 2.0 * delta * s;
78                  q = 1.0 - s;
79              }
80              else
81              {
82                  q = lowValue / highValue;
83                  r = currentValue / highValue;
84                  p = s * (2.0 * delta * q * (q - r)
85                      - (current - low) * (r - 1.0));
86                  q = (q - 1.0) * (r - 1.0) * (s - 1.0);
87              }
88
89              if (p > 0.0)
90                  q = -q;
91              else
92                  p = abs(p);
93
94              double min1 = 3.0 * delta * q - abs(accuracy * q);
95              double min2 = abs(previousCorrection * q);
96
97              if (2.0 * p < min(min1, min2))
98              {
99                  //Accept interpolation.
100                 previousCorrection = correction;
101                 correction = p / q;
102             }
103             else
104             {
105                 //Interpolation failed, use bisection.
106                 correction = delta;
107                 previousCorrection = correction;
108             }
109         }
110         else
111         {    //Bounds decreasing too slowly, use bisection.
112             correction = delta;
113             previousCorrection = correction;
114         }
115
116     low = current; //Move last best guess to a.
117     lowValue = currentValue;
```

```
118
119         if (abs(correction) > accuracy) //Evaluate new trial root.
120             current += correction;
121         else
122             current += switchSign(accuracy, delta);
123
124             currentValue = function(current) - constant;
125         }
126
127         throw std::exception("Number of maximum evaluations reached.");
128     }
129
130     double BrentRootFinder::find(Function& function, Function& derivative,
131         double lowerBound, double upperBound, double constant)
132     { return find(function, lowerBound, upperBound, constant); } }
```

FalsePositionRootFinder.h

```
 1   #pragma once
 2
 3   #include "RootFinder.h"
 4   #include "Calc.h"
 5
 6   #include <cmath>
 7   #include <utility>
 8
 9   namespace Infinity {
10   class FalsePositionRootFinder : public RootFinder
11   {
12   public:
13       FalsePositionRootFinder(double convergenceTolerance,
14           unsigned short maxEvaluations);
15       virtual ~FalsePositionRootFinder();
16
17       double find(Function& function, double lowerBound,
18           double upperBound, double constant = 0.0);
19       double find(Function& function, Function& derivative, double lowerBound,
20   double upperBound, double constant = 0.0);
21   }; }
```

FalsePositionRootFinder.cpp

```
 1   #include "stdafx.h"
 2   #include "FalsePositionRootFinder.h"
 3
 4   using namespace std;
 5
 6   namespace Infinity {
 7   FalsePositionRootFinder::FalsePositionRootFinder(double convergenceTolerance,
 8       unsigned short maxEvaluations)
 9       : RootFinder(convergenceTolerance, maxEvaluations)
10   { }
11
12   FalsePositionRootFinder::~FalsePositionRootFinder()
13   { }
14
```

```
15  double FalsePositionRootFinder::find(Function& function, double lowerBound,
16  double upperBound, double constant)
17  {
18      //Implementation from Numerical Recipies, 3rd Edition
19      //(page 449)
20      if (lowerBound >= upperBound)
21          throw std::exception("Bad interval.");
22
23      double low, high, mid;
24
25      double lowValue = function(lowerBound) - constant;
26      double highValue = function(upperBound) - constant;
27      double midValue;
28
29      if (isEqual(lowValue, 0.0))
30          return lowerBound;
31
32      if (isEqual(highValue, 0.0))
33          return upperBound;
34
35      if (sign(lowValue) == sign(highValue))
36          throw std::exception("Functions is not bracketed properly.");
37
38      if (lowValue < 0.0)
39      {
40          low = lowerBound;
41          high = upperBound;
42      }
43      else
44      {
45          low = upperBound;
46          high = lowerBound;
47          swap(lowValue, highValue);
48      }
49
50      double correction;
51
52      for (unsigned short eval = 2; eval < maxEval; eval++)
53      {
54          mid = low + (high - low) * lowValue / (lowValue - highValue);
55          midValue = function(mid) - constant;
56
57          //adjust the interval
58          if (midValue < 0.0)
59          {
60              correction = low - mid;
61              low = mid;
62              lowValue = midValue;
63          }
64          else
65          {
66              correction = high - mid;
67              high = mid;
68              highValue = midValue;
69          }
70
71          if (isEqual(midValue, 0.0) || abs(correction) < tolerance)
```

```
72                return mid;
73            }
74
75            throw std::exception("Number of maximum evaluations reached.");
76        }
77
78        double FalsePositionRootFinder::find(Function& function, Function&
79        derivative,
80            double lowerBound, double upperBound, double constant)
81        { return find(function, lowerBound, upperBound, constant); } }
```

Color.h

```
1   #pragma once
2
3   namespace Infinity {
4   class Color
5   {
6   public:
7       Color(int r, int g, int b);
8       Color(const Color &original);
9       Color(void);
10      ~Color(void);
11
12      int r, g, b;
13
14      static Color Interpolate(const Color& first, const Color& second, double
15  amount);
16
17      Color& operator=(const Color &original);
18  private:
19      static int linearIntepolation(int c0, int c1, double t);
20  };
21
22  inline int Color::linearIntepolation(int c0, int c1, double t)
23  { return static_cast<int>((1-t) * c0 + t * c1); }
24
25  inline Color Color::Interpolate(const Color& first, const Color& second,
26      double amount)
27  {
28      Color result;
29      result.r = linearIntepolation(first.r, second.r, amount);
30      result.g = linearIntepolation(first.g, second.g, amount);
31      result.b = linearIntepolation(first.b, second.b, amount);
32
33      return result;
34  } }
```

Color.cpp

```
1   #include "stdafx.h"
2   #include "Color.h"
3
4   namespace Infinity {
5   Color::Color(int r, int g, int b)
6       : r(r), g(g), b(b)     { }
```

```
7
8   Color::Color(const Color &original)
9   {
10      r = original.r;
11      g = original.g;
12      b = original.b;
13  }
14
15  Color::Color(void)
16      : r(0), g(0), b(0)
17  { }
18
19  Color::~Color(void) { }
20
21  Color& Color::operator=(const Color &original)
22  {
23      r = original.r;
24      g = original.g;
25      b = original.b;
26
27      return *this;
28  } }
```

IniReader.h

```
1   #pragma once
2
3   #include "Utilities.h"
4
5   #include <string>
6   #include <vector>
7   #include <clocale>
8   #include <tchar.h>
9
10  namespace Infinity {
11  class IniReader
12  {
13  public:
14      IniReader(void);
15      ~IniReader(void);
16
17      static double GetDoubleValue(LPCTSTR section, LPCTSTR key, LPCTSTR file);
18      static std::vector<double> GetDoubleVector(LPCTSTR section,
19          LPCTSTR key, LPCTSTR file);
20
21      static int GetIntValue(LPCTSTR section, LPCTSTR key, LPCTSTR file);
22      static std::vector<int> GetIntVector(LPCTSTR section,
23          LPCTSTR key, LPCTSTR file);
24
25      static std::wstring GetStringValue(LPCTSTR section,
26          LPCTSTR key, LPCTSTR file);
27      static std::vector<std::wstring> GetStringVector(LPCTSTR section,
28          LPCTSTR key, LPCTSTR file);
29  }; }
```

IniReader.cpp

```
1   #include "stdafx.h"
2   #include "IniReader.h"
3
4   using namespace std;
5
6   namespace Infinity {
7   IniReader::IniReader(void)
8   { }
9
10  IniReader::~IniReader(void)
11  { }
12
13  double IniReader::GetDoubleValue(LPCTSTR section, LPCTSTR key, LPCTSTR file)
14  {
15      TCHAR value[256];
16      _locale_t locale = _create_locale(LC_NUMERIC, "english");
17
18      DWORD charCount = GetPrivateProfileString(section, key, L"0.0",
19          value, 256, file);
20      return _tstof_l(value,locale);
21  }
22
23  vector<double> IniReader::GetDoubleVector(LPCTSTR section, LPCTSTR key,
24      LPCTSTR file)
25  {
26      vector<double> result;
27      TCHAR value[256];
28      _locale_t locale = _create_locale(LC_NUMERIC, "english");
29
30      DWORD charCount = GetPrivateProfileString(section, key, L"0.0",
31          value, 256, file);
32      vector<wstring> tokens = split(value, L" \t");
33      for(auto it = tokens.begin(); it != tokens.end(); it++)
34          result.push_back( _tstof_l(it->c_str(), locale));
35
36      return result;
37  }
38
39  int IniReader::GetIntValue(LPCTSTR section, LPCTSTR key, LPCTSTR file)
40  {
41      _locale_t locale = _create_locale(LC_NUMERIC, "english");
42      TCHAR value[256];
43      DWORD charCount = GetPrivateProfileString(section, key, L"0",
44          value, 256, file);
45      return _tstoi_l(value,locale);
46  }
47
48  vector<int> IniReader::GetIntVector(LPCTSTR section, LPCTSTR key, LPCTSTR
49  file)
50  {
51      vector<int> result;
52      TCHAR value[256];
```

```
53      _locale_t locale = _create_locale(LC_NUMERIC, "english");
54
55      DWORD charCount = GetPrivateProfileString(section, key, L"0.0",
56          value, 256, file);
57      vector<wstring> tokens = split(value, L" \t");
58      for(auto it = tokens.begin(); it != tokens.end(); it++)
59          result.push_back( _tstoi_l(it->c_str(), locale));
60
61      return result;
62  }
63
64  wstring IniReader::GetStringValue(LPCTSTR section, LPCTSTR key, LPCTSTR file)
65  {
66      TCHAR value[256];
67      DWORD charCount = GetPrivateProfileString(section, key, L"0.0",
68          value, 256, file);
69      return value;
70  }
71
72  vector<wstring> IniReader::GetStringVector(LPCTSTR section, LPCTSTR key,
73      LPCTSTR file)
74  {
75      TCHAR value[256];
76      _locale_t locale = _create_locale(LC_NUMERIC, "english");
77
78      DWORD charCount = GetPrivateProfileString(section, key, L"0.0",
79          value, 256, file);
80      return split(value, L" \t");
81  } }
```

Palette.h

```
1   #pragma once
2
3   #include <string>
4   #include <vector>
5   #include <fstream>
6   #include "Color.h"
7   #include "Calc.h"
8
9   namespace Infinity {
10  class Palette
11  {
12  public:
13      Palette(std::wstring fileName);
14      Palette(const Palette &source);
15      Palette& operator= (const Palette &source);
16      ~Palette(void);
17
18      Color getColor(double value, double minValue, double maxValue) const;
19
20  private:
21      std::vector<Color> palette;
22  }; }
```

Palette.cpp

```
1   #include "stdafx.h"
2   #include "Palette.h"
3
4   using namespace std;
5
6   namespace Infinity {
7   Palette::Palette(wstring fileName)
8   {
9       ifstream paletteFile(L"palettes/" + fileName);
10      int r, g, b;
11
12      while(!paletteFile.eof())
13      {
14          paletteFile >> r;
15          paletteFile >> g;
16          paletteFile >> b;
17
18          palette.push_back(Color(r, g, b));
19      }
20
21      paletteFile.close();
22  }
23
24  Palette::Palette(const Palette& source)
25  {
26      palette = source.palette;
27  }
28
29  Palette& Palette::operator= (const Palette& source)
30  {
31      palette = source.palette;
32
33      return *this;
34  }
35
36  Palette::~Palette(void)
37  { }
38
39  Color Palette::getColor(double value, double minValue, double maxValue) const
40  {
41      double absAmount = (value - minValue) / (maxValue - minValue);
42
43      size_t index = static_cast<size_t>(floor(absAmount
44          * (palette.size() - 1)));
45      double amount = absAmount * (palette.size() - 1) - index;
46
47
48      Color color;
49      if (isEqual(value, maxValue, 0.001))
50          color = palette[palette.size()-1];
51      else
52          color = Color::Interpolate(palette[index], palette[index+1], amount);
53
54      return color; } }
```

RootFinder.h

```
#pragma once

#include "UnivariateFunction.h"
#include <exception>

namespace Infinity {
class RootFinder
{
public:
    RootFinder(double convergenceTolerance, unsigned short maxEvaluations);
    virtual ~RootFinder();

    virtual double find(Function& function, double lowerBound,
        double upperBound, double constant = 0.0) = 0;
    virtual double find(Function& function, Function& derivative,
        double lowerBound, double upperBound, double constant = 0.0) = 0;

protected:
    double tolerance;
    unsigned short maxEval;
}; }
```

RootFinder.cpp

```
#include "stdafx.h"
#include "RootFinder.h"

namespace Infinity {
RootFinder::RootFinder(double convergenceTolerance,
    unsigned short maxEvaluations)
    : tolerance(convergenceTolerance), maxEval(maxEvaluations)
{ }

RootFinder::~RootFinder(void)
{ } }
```

stdafx.h

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, // but are
changed infrequently

#pragma once

#include "targetver.h"

#define WIN32_LEAN_AND_MEAN
#define _USE_MATH_DEFINES
#define PI_F    3.141592653589793f
#define PI_2_F    1.5707963267948966f
#define PI_4_F    0.7853981633974483f
#define TWO_PI_F    6.28318530717959f
```

```
#define NO_REFLECTION 0
#define No_REFLECTION_STR L"No”
#define DJURASEVIC_REFLECTION 1
#define DJURASEVIC_REFLECTION_STR L"Djurasevic"
#define KHRUZINA_REFLECTION 2
#define KHRUZINA_REFLECTION_STR L"Khruzina"
#define NO_OUTPUT 0
#define NO_OUTPUT_STR L"No"
#define OUTPUT_ONCE 1
#define OUTPUT_ONCE_STR L"Once"
#define OUTPUT_ON_EACH_PHASE 2
#define OUTPUT_ON_EACH_PHASE_STR L"OnEachPhase"
#define MESH_3D 0
#define MESH_3D_STR L"ThreeD"
#define MESH_2D 1
#define MESH_2D_STR L"TwoD"
#define MESH_HIDDEN 2
#define MESH_HIDDEN_STR L"Hidden"
#define DISC_SHAPE_NONE 0
#define DISC_SHAPE_NONE_STR L"None"
#define DISC_SHAPE_CONICAL 1
#define DISC_SHAPE_CONICAL_STR L"Conical"
#define DISC_SHAPE_TOROIDAL 2
#define DISC_SHAPE_TOROIDAL_STR L"Toroidal"
#define DISC_TEMP_DIST_ZOLA 0
#define DISC_TEMP_DIST_ZOLA_STR L"Zola"
#define DISC_TEMP_DIST_ALPHA 1
#define DISC_TEMP_DIST_ALPHA_STR L"Alpha"
#define DISC_TEMP_DIST_DJURASEVIC 2
#define DISC_TEMP_DIST_DJURASEVIC_STR L"Djurasevic"
#define GRID_VISIBILITY 0
#define GRID_VISIBILITY_STR L"Grid"
#define CONVEX_HULL_VISIBILITY 1
#define CONVEX_HULL_VISIBILITY_STR L"ConvexHull"
#define TRIANGLE_FULL 0
#define TRIANGLE_FULL_STR L"Full"
#define TRIANGLE_POINT 1
#define TRIANGLE_POINT_STR L"Point"
#define XUNITS_PHASE 0
#define XUNITS_PHASE_STR L"Phase"
#define XUNITS_TIME 1
#define XUNITS_TIME_STR L"Time"
#define YUNITS_FLUX 0
#define YUNITS_FLUX_STR L"Flux"
#define YUNITS_MAGNITUDE 1
#define YUNITS_MAGNITUDE_STR L"Magnitude"
#define YUNITS_ABSOLUTE 0
#define YUNITS_ABSOLUTE_STR L"Absolute"
#define YUNITS_NORMALIZED 1
#define YUNITS_NORMALIZED_STR L"Normalized"
#define PLANCK_FLUX 0
#define PLANCK_FLUX_STR L"Planck"
#define KURUTZ_FLUX 1
#define KURUTZ_FLUX_STR L"Kurutz"

// TODO: reference additional headers your program requires here
#include <Windows.h>
```

UnivariateFunction.h

```
#pragma once

namespace Infinity {
template <class TOut, class TArg>
class UnivariateFunction
{
public:
    //Evaluate
    virtual TOut operator() (TArg x) const = 0;

protected:
    UnivariateFunction() { }

    //Constructors/destructors
    ~UnivariateFunction()
    { }
};

typedef UnivariateFunction<double, double> Function; }
```

Utilities.h

```
#pragma once

#include <vector>

namespace Infinity {
std::vector<std::wstring> split(wchar_t* string, wchar_t* separators);
std::vector<std::string> split(char* string, char* separators); }
```

Utilities.cpp

```
#include "stdafx.h"
#include "Utilities.h"

namespace Infinity {
std::vector<std::wstring> split(wchar_t* string, wchar_t* separators)
{
    std::vector<std::wstring> result;
    wchar_t* token;
    wchar_t* next_token = NULL;

    token = wcstok_s(string, separators, &next_token);

    while(token != NULL)
    {
        result.push_back(token);
        token = wcstok_s(NULL, separators, &next_token);
    }

    return result;
}
```

```
21
22  std::vector<std::string> split(char* string, char* separators)
23  {
24      std::vector<std::string> result;
25      char* token;
26      char* next_token = NULL;
27
28      token = strtok_s(string, separators, &next_token);
29
30      while(token != NULL)
31      {
32          result.push_back(token);
33          token = strtok_s(NULL, separators, &next_token);
34      }
35
36      return result;
37  } }
```

8.2. Geometrija

Circle.h

```
 1  #pragma once
 2
 3  #include "SimpleMath.h"
 4  #include "Calc.h"
 5
 6  namespace Infinity {
 7  class Circle
 8  {
 9  public:
10      //Constructors/destructors
11      Circle(float x, float y, float radius);
12      Circle(DirectX::SimpleMath::Vector2 center, float radius);
13      Circle(const Circle& c);
14      Circle();
15      ~Circle();
16
17      //Methods
18      const DirectX::SimpleMath::Vector2* getCenter() const;
19      float getRadius() const;
20
21      static bool Intersects(const Circle* first, const Circle* second);
22      static Circle GetBoundingCircle(DirectX::SimpleMath::Vector2& pointA,
23          DirectX::SimpleMath::Vector2& pointB, DirectX::SimpleMath::Vector2&
24  pointC);
25
26      //Operators
27      Circle& operator=(const Circle& right);
28      friend bool operator==(const Circle& left, const Circle& right);
29      friend bool operator!=(const Circle& left, const Circle& right);
30
31  private:
32      DirectX::SimpleMath::Vector2 center;
33      float radius;
```

```
};

inline const DirectX::SimpleMath::Vector2* Circle::getCenter() const
{ return &center; }

inline float Circle::getRadius() const
{ return radius; } }
```

Circle.cpp

```
#include "stdafx.h"
#include "Circle.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {

Circle::Circle(float x, float y, float radius) : center(x, y), radius(radius)
{ }

Circle::Circle(Vector2 center, float radius) : center(center), radius(radius)
{ }

Circle::Circle(const Circle& c) : center(c.center), radius(c.radius)
{ }

Circle::Circle() : center(0, 0), radius(1)
{ }

Circle::~Circle()
{ }

//Assignment
Circle& Circle::operator=(const Circle& right)
{
    center = right.center;
    radius = right.radius;

    return *this;
}

//Comparison
bool operator==(const Circle& left, const Circle& right)
{
    return (left.center == right.center) && isEqual(left.radius,
right.radius);
}

bool operator!=(const Circle& left, const Circle& right)
{
    return !(left == right);
}

//Methods
```

```
Circle Circle::GetBoundingCircle(Vector2& pointA, Vector2& pointB, Vector2&
pointC)
{
    float radius;
    Vector2 center;

    //represents sufficiently small number to be considered zero
    float epsilon = numeric_limits<float>::epsilon() * 16;

        //Center and radius of the bounding circle taken from
        //http://realtimecollisiondetection.net/blog/?p=20

    Vector2 AB = pointB - pointA;
    Vector2 AC = pointC - pointA;


    float dotABAB = AB.Dot(AB);
    float dotABAC = AB.Dot(AC);
    float dotACAC = AC.Dot(AC);
    float d = 2.0f * (dotABAB * dotACAC - dotABAC * dotABAC);

    Vector2* referencePoint = &pointA;

    if (abs(d) <= epsilon)
    {
       // a, b, and c lie on a line. Circle center is center
       //of AABB of the points, and radius is distance from
       //circle center to AABB corner

    //AABB bbox = ComputeAABB(a, b, c);
    //circle.c = 0.5f * (bbox.min + bbox.max);
    //referencePt = bbox.min;

    float AB = Vector2::DistanceSquared(pointA, pointB);
    float AC = Vector2::DistanceSquared(pointA, pointC);
    float BC = Vector2::DistanceSquared(pointB, pointC);

    if ((AB > AC) && (AB > BC)) //then AB is the longest side.
    {
        radius = AB / 2.0f;
        center = 0.5 * (pointA + pointB);
    }
    else //either AC or BC will be the longest side
        if (AC > BC) //then AC is the loongest side
        {
            radius = AC / 2.0f;
            center = 0.5 * (pointA + pointC);
        }
        else //BC is the longest side
        {
            radius = BC / 2.0f;
            center = 0.5 * (pointB + pointC);
        }
    }
    else
    {
    float s = (dotABAB * dotACAC - dotACAC * dotABAC) / d;
```

```
103             float t = (dotACAC * dotABAB - dotABAB * dotABAC) / d;
104           // s controls height over AC, t over AB, (1-s-t) over BC
105             if (s <= 0)
106             center = 0.5 * (pointA + pointC);
107         else
108             if (t <= 0)
109                 center = 0.5 * (pointA + pointB);
110                     else
111                 if (s + t >= 1)
112                             {
113                     center = 0.5 * (pointB + pointC);
114                     referencePoint = &pointB;
115                             }
116                             else
117                     center = pointA + s * AB + t * AC;
118         }
119
120         radius = sqrt(pow2(center.x - referencePoint->x)
121             + pow2(center.y - referencePoint->y));
122
123         return Circle(center, radius);
124     }
125
126     bool Circle::Intersects(const Circle* first, const Circle* second)
127     {
128         float distanceSquared = pow2(first->getCenter()->x - second->getCenter()-
129     >x) + pow2(first->getCenter()->y - second->getCenter()->y);
130
131         float radiiSquared = pow2(first->getRadius() + second->getRadius());
132
133         if (distanceSquared > radiiSquared)
134             return false;
135         else
136             return true;
137     } }
```

CrossSection.h

```
1   #pragma once
2
3   #include "SimpleMath.h"
4
5   namespace Infinity {
6   class CrossSection
7   {
8   public:
9       CrossSection(size_t pointCount);
10      CrossSection(const CrossSection& source);
11      ~CrossSection();
12
13      CrossSection& operator=(const CrossSection& right);
14      DirectX::SimpleMath::Vector2& operator[](size_t index );
15
16      size_t getCount() const;
17
18      static CrossSection CreateConical(float innerRadius, float outerRadius,
```

```
        float innerThickness, float outerThickness, size_t radialSegments,
        size_t innerEdgeSegments, size_t outerEdgeSegments);
    static CrossSection CreateToroidal(float radius, float semiMajorAxis,
        float semiMinorAxis, size_t segments);

private:
    size_t count;
    DirectX::SimpleMath::Vector2* points;
};

inline size_t CrossSection::getCount() const
{
    return count;
}

inline DirectX::SimpleMath::Vector2& CrossSection::operator[](size_t index )
{
    return points[index];
} }
```

CrossSection.cpp

```
#include "stdafx.h"
#include "CrossSection.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {
CrossSection::CrossSection(size_t pointCount) : count(pointCount)
{
    points = new Vector2[pointCount];
}

CrossSection::CrossSection(const CrossSection &source)
{
    count = source.count;
    points = new Vector2[count];

    for (size_t i = 0; i < count; i++)
    {
        points[i].x = source.points[i].x;
        points[i].y = source.points[i].y;
    }
}

CrossSection::~CrossSection()
{
    delete[] points;
}

CrossSection& CrossSection::operator=(const CrossSection& source)
{
    delete[] points;

    count = source.count;
```

```
35      points = new Vector2[count];
36
37      for (size_t i = 0; i < count; i++)
38      {
39          points[i].x = source.points[i].x;
40          points[i].y = source.points[i].y;
41      }
42
43      return *this;
44  }
45
46  CrossSection CrossSection::CreateConical(float innerRadius, float
47  outerRadius,
48      float innerThickness, float outerThickness, size_t radialSegments,
49      size_t innerEdgeSegments, size_t outerEdgeSegments)
50  {
51      CrossSection result(2 * radialSegments + innerEdgeSegments +
52  outerEdgeSegments);
53
54      //upper edge;
55      size_t index = 0;
56      for (size_t i = 0; i < radialSegments; i++)
57      {
58          float s = static_cast<float>(i) / radialSegments;
59          result[index].x = (1 - s) * innerRadius + s * outerRadius;
60          result[index].y = 0.5f * ((1 - s) * innerThickness + s *
61  outerThickness);
62          index++;
63      }
64
65      //outer rim
66      for (size_t i = 0; i < outerEdgeSegments; i++)
67      {
68          float s = static_cast<float>(i) / outerEdgeSegments;
69          result[index].x = outerRadius;
70          result[index].y = 0.5f * (1 - 2 * s) * outerThickness;
71          index++;
72      }
73
74      //lower edge;
75      for (size_t i = 0; i < radialSegments; i++)
76      {
77          float s = static_cast<float>(i) / radialSegments;
78          result[index].x = (1 - s) * outerRadius + s * innerRadius;
79          result[index].y = 0.5f * ((s - 1) * outerThickness - s *
80  innerThickness);
81          index++;
82      }
83
84      //inner rim
85      for (size_t i = 0; i < innerEdgeSegments; i++)
86      {
87          float s = static_cast<float>(i) / innerEdgeSegments;
88          result[index].x = innerRadius;
89          result[index].y =  0.5f * (2 * s - 1) * innerThickness;
90          index++;
91      }
```

```
92      return result;
93  }
94
95  CrossSection CrossSection::CreateToroidal(float radius, float semiMajorAxis,
96      float semiMinorAxis, size_t segments)
97  {
98      CrossSection result(segments);
99
100     const float phi = TWO_PI_F  / segments;
101
102     for (size_t i = 0; i < segments; i++)
103     {
104         result[i].x = radius + semiMajorAxis * cos((segments - i) * phi);
105         result[i].y = semiMinorAxis * sin((segments - i) * phi);
106     }
107     return result;
108 }
```

Mesh3.h

```
1   #pragma once
2
3   #include "SimpleMath.h"
4
5   #include <vector>
6   #include <array>
7   #include <algorithm>
8   #include <fstream>
9
10  #include "Surface.h"
11  #include "Calc.h"
12  #include "CrossSection.h"
13  #include "Circle.h"
14
15  namespace Infinity {
16  class Mesh3
17  {
18  public:
19      //Constrors, destructors and assigment operator
20      Mesh3(size_t vertexCount, size_t triangleCount);
21      Mesh3(const Mesh3& source);
22
23      ~Mesh3(void);
24
25      Mesh3& operator=(const Mesh3& right);
26
27      //Subdivision
28      enum SubdivisionType { FrequencySubdivision, DepthSubdivision };
29      void subdivide(size_t level, SubdivisionType type =
30  FrequencySubdivision);
31
32      //Inflate & init. These go hand in hand...
33      void pushTo(Surface* surface);
34      void init(size_t lcCount = 1);
35
36      //Mesh export
```

```
37      void writeObj(std::wstring fileName, bool onlyVisibleFaces = false);
38
39      //Transform
40      void transform(DirectX::SimpleMath::Matrix &matrix);
41
42      //Properties
43      size_t getVertexCount() const;
44      size_t getTriangleCount() const;
45      size_t getIndexCount() const;
46
47      DirectX::SimpleMath::Vector3* getPoint(size_t index);
48      DirectX::SimpleMath::Vector3* getVertex(size_t index);
49      size_t getIndex(size_t index);
50
51      DirectX::SimpleMath::Vector3* getCentroid(size_t index);
52      DirectX::SimpleMath::Vector3* getNormal(size_t index);
53
54      double getArea(size_t index) const;
55      double getMeshArea() const;
56      double getProjectedArea(size_t index) const;
57      void setProjectedArea(size_t index, double value);
58      double getVolume() const;
59
60      float getInnerRadius() const;
61      float getOuterRadius() const;
62
63      double getVisibility(size_t index) const;
64      void setVisibility(size_t index, double value);
65
66      DirectX::SimpleMath::Vector3* getCenter();
67
68      double getGravityAcceleration(size_t index) const;
69      void setGravityAcceleration(size_t index, double value);
70      double getTemperature(size_t index) const;
71      void setTemperature(size_t index, double value);
72      DirectX::SimpleMath::Vector3* getVelocity(size_t index);
73      void setVelocity(size_t index, DirectX::SimpleMath::Vector3 value);
74
75      double getFlux(size_t passbandID, size_t index) const;
76      void setFlux(size_t passbandID, size_t index, double value);
77
78      //Static methods
79      static Mesh3 CreateTetrahedron();
80      static Mesh3 CreateHexahedron();
81      static Mesh3 CreateOctahedron();
82      static Mesh3 CreateDodecahedron();
83      static Mesh3 CreateIcosahedron();
84      static Mesh3 CreateLathe(CrossSection& crossSection, size_t
85  segmentCount);
86
87      void adaptive(size_t level, DirectX::SimpleMath::Vector3* center,
88          double innerRadius, double outerRadius);
89  private:
90      std::vector<DirectX::SimpleMath::Vector3> vertices;
91      std::vector<size_t> indices;
92
93      float outerRadius, innerRadius;
```

```
94        DirectX::SimpleMath::Vector3 center;
95
96        void subdivideByDepth(size_t level);
97        void subdivideByFrequency(size_t level);
98
99
100       size_t findIndex(const DirectX::SimpleMath::Vector3& point, size_t lower,
101           size_t upper);
102
103       std::vector<DirectX::SimpleMath::Vector3> normals;
104       std::vector<DirectX::SimpleMath::Vector3> centroids;
105
106       //Physical properties
107       std::vector<double> area;
108
109       double totalArea;
110       double totalVolume;
111
112       std::vector<double> temperature;
113       std::vector<double> gravity;
114       std::vector<double> flux;
115       std::vector<DirectX::SimpleMath::Vector3> velocity;
116
117       //2D properties
118       std::vector<double> visibility;
119       std::vector<double> projectedArea;
120
121       size_t lcCount;
122   };
123
124   //Properties
125   inline size_t Mesh3::getVertexCount() const { return vertices.size(); }
126   inline size_t Mesh3::getTriangleCount() const { return indices.size() / 3; };
127   inline size_t Mesh3::getIndexCount() const { return indices.size(); };
128   inline DirectX::SimpleMath::Vector3* Mesh3::getCentroid(size_t index)
129   { return &centroids[index]; };
130   inline DirectX::SimpleMath::Vector3* Mesh3::getNormal(size_t index)
131   { return &normals[index]; };
132   inline DirectX::SimpleMath::Vector3* Mesh3::getCenter() { return &center; };
133
134   inline double Mesh3::getGravityAcceleration(size_t index) const
135   { return gravity[index]; }
136   inline void Mesh3::setGravityAcceleration(size_t index, double value)
137   { gravity[index] = value; };
138   inline double Mesh3::getTemperature(size_t index) const
139   { return temperature[index]; };
140   inline void Mesh3::setTemperature(size_t index, double value)
141   { temperature[index] = value; };
142   inline DirectX::SimpleMath::Vector3* Mesh3::getVelocity(size_t index)
143   { return &velocity[index]; };
144   inline void Mesh3::setVelocity(size_t index, DirectX::SimpleMath::Vector3
145   value)
146   { velocity[index] = value; };
147
148   inline double Mesh3::getFlux(size_t passbandID, size_t index) const
149   { size_t triangleCount = indices.size() / 3;
150   return flux[passbandID * triangleCount + index]; }
```

```
inline void Mesh3::setFlux(size_t passbandID, size_t index, double value)
{ size_t triangleCount = indices.size() / 3;
flux[passbandID * triangleCount + index] = value; }

inline double Mesh3::getArea(size_t index) const { return area[index]; }
inline double Mesh3::getMeshArea() const { return totalArea; }
inline double Mesh3::getProjectedArea(size_t index) const
{ return projectedArea[index]; }
inline void Mesh3::setProjectedArea(size_t index, double value)
{ projectedArea[index] = value; }
inline double Mesh3::getVolume() const { return totalVolume; }

inline float Mesh3::getInnerRadius() const { return innerRadius; }
inline float Mesh3::getOuterRadius() const { return outerRadius; }
inline DirectX::SimpleMath::Vector3* Mesh3::getPoint(size_t index)
{ return &vertices[indices[index]]; }
inline DirectX::SimpleMath::Vector3* Mesh3::getVertex(size_t index)
{ return &vertices[index]; }
inline size_t Mesh3::getIndex(size_t index) { return indices[index]; }

inline double Mesh3::getVisibility(size_t index) const
{ return visibility[index]; }
inline void Mesh3::setVisibility(size_t index, double value)
{ visibility[index] = value; }

inline void Mesh3::transform(DirectX::SimpleMath::Matrix& matrix)
{
    center = DirectX::SimpleMath::Vector3::Transform(center, matrix);

    for (auto it = vertices.begin(); it !=  vertices.end(); ++it)
        *it = DirectX::SimpleMath::Vector3::Transform(*it, matrix);

    for (auto it = centroids.begin(); it !=  centroids.end(); ++it)
        *it = DirectX::SimpleMath::Vector3::Transform(*it, matrix);

    for (auto it = normals.begin(); it !=  normals.end(); ++it)
        *it = DirectX::SimpleMath::Vector3::TransformNormal(*it, matrix);

    for (auto it = velocity.begin(); it !=  velocity.end(); ++it)
        *it = DirectX::SimpleMath::Vector3::TransformNormal(*it, matrix);
} }
```

Mesh3.cpp

```
#include "stdafx.h"
#include "Mesh3.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {
Mesh3::Mesh3(size_t vertexCount, size_t triangleCount) :
    outerRadius(1.0f),
    innerRadius(1.0f),
    center(0.0f)
{
```

```
13      vertices.reserve(vertexCount);
14      indices.reserve(3 * triangleCount);
15  }
16
17  Mesh3::Mesh3(const Mesh3& source)
18  {
19      vertices = source.vertices;
20      indices = source.indices;
21      normals = source.normals;
22      centroids = source.centroids;
23
24      area = source.area;
25      projectedArea = source.projectedArea;
26
27      totalArea = source.totalArea;
28      totalVolume = source.totalVolume;
29
30      temperature = source.temperature;
31      gravity = source.gravity;
32      flux = source.flux;
33      velocity = source.velocity;
34
35      visibility = source.visibility;
36
37      center = source.center;
38      innerRadius = source.innerRadius;
39      outerRadius = source.outerRadius;
40
41      lcCount = source.lcCount;
42  }
43
44  Mesh3::~Mesh3(void) { }
45
46  Mesh3& Mesh3::operator=(const Mesh3& source)
47  {
48      vertices = source.vertices;
49      indices = source.indices;
50      normals = source.normals;
51      centroids = source.centroids;
52
53      area = source.area;
54      projectedArea = source.projectedArea;
55
56      totalArea = source.totalArea;
57      totalVolume = source.totalVolume;
58
59      temperature = source.temperature;
60      gravity = source.gravity;
61      flux = source.flux;
62      velocity = source.velocity;
63
64      visibility = source.visibility;
65
66      center = source.center;
67      innerRadius = source.innerRadius;
68      outerRadius = source.outerRadius;
69
```

```
70      lcCount = source.lcCount;
71
72      return *this;
73  }
74
75  //Static methods
76  //Data taken from: David Eberly - Platonic Solids (2008)
77  Mesh3 Mesh3::CreateTetrahedron()
78  {
79      Mesh3 tetra(4, 4);
80
81      tetra.vertices.push_back(
82          Vector3(0.0f,  0.0f,  1.0f));
83      tetra.vertices.push_back(
84          Vector3(2.0f * sqrt(2.0f) / 3.0f, 0.0f, -1.0f / 3.0f));
85      tetra.vertices.push_back(
86          Vector3(-sqrt(2.0f) / 3.0f,   sqrt(6.0f) / 3.0f, -1.0f / 3.0f));
87      tetra.vertices.push_back(
88          Vector3(-sqrt(2.0f) / 3.0f,  -sqrt(6.0f) / 3.0f, -1.0f / 3.0f));
89
90      tetra.indices.push_back(0); tetra.indices.push_back(1);
91      tetra.indices.push_back(2); //Triangle 1
92      tetra.indices.push_back(0); tetra.indices.push_back(2);
93      tetra.indices.push_back(3); //Triangle 2
94      tetra.indices.push_back(0); tetra.indices.push_back(3);
95      tetra.indices.push_back(1); //Traingle 3
96      tetra.indices.push_back(1); tetra.indices.push_back(3);
97      tetra.indices.push_back(2); //Triangle 4
98
99      tetra.outerRadius = 1.0f;
100     tetra.innerRadius = 1.0f / 3.0f;
101
102     return tetra;
103 }
104
105 //Data taken from: David Eberly - Platonic Solids (2008)
106 Mesh3 Mesh3::CreateHexahedron()
107 {
108     Mesh3 cube(8, 12);
109
110     float a = 1.0f / sqrt(3.0f);
111
112     cube.vertices.push_back(Vector3(-a, -a, -a));
113     cube.vertices.push_back(Vector3( a, -a, -a));
114     cube.vertices.push_back(Vector3( a,  a, -a));
115     cube.vertices.push_back(Vector3(-a,  a, -a));
116     cube.vertices.push_back(Vector3(-a, -a,  a));
117     cube.vertices.push_back(Vector3( a, -a,  a));
118     cube.vertices.push_back(Vector3( a,  a,  a));
119     cube.vertices.push_back(Vector3(-a,  a,  a));
120
121     cube.indices.push_back(0); cube.indices.push_back(3);
122     cube.indices.push_back(2); //Triangle 1
123     cube.indices.push_back(0); cube.indices.push_back(2);
124     cube.indices.push_back(1); //Triangle 2
125     cube.indices.push_back(0); cube.indices.push_back(1);
126     cube.indices.push_back(5); //Traingle 3
```

```
127         cube.indices.push_back(0); cube.indices.push_back(5);
128         cube.indices.push_back(4); //Triangle 4
129         cube.indices.push_back(0); cube.indices.push_back(4);
130         cube.indices.push_back(7); //Triangle 5
131         cube.indices.push_back(0); cube.indices.push_back(7);
132         cube.indices.push_back(3); //Triangle 6
133         cube.indices.push_back(6); cube.indices.push_back(5);
134         cube.indices.push_back(1); //Traingle 7
135         cube.indices.push_back(6); cube.indices.push_back(1);
136         cube.indices.push_back(2); //Triangle 8
137         cube.indices.push_back(6); cube.indices.push_back(2);
138         cube.indices.push_back(3); //Triangle 9
139         cube.indices.push_back(6); cube.indices.push_back(3);
140         cube.indices.push_back(7); //Triangle 10
141         cube.indices.push_back(6); cube.indices.push_back(7);
142         cube.indices.push_back(4); //Traingle 11
143         cube.indices.push_back(6); cube.indices.push_back(4);
144         cube.indices.push_back(5); //Triangle 12
145 
146         cube.outerRadius = 1.0f;
147         cube.innerRadius = 1.0f / sqrt(3.0f);
148 
149         return cube;
150     }
151 
152     //Data taken from: David Eberly - Platonic Solids (2008)
153     Mesh3 Mesh3::CreateOctahedron()
154     {
155         Mesh3 octa(6, 8);
156 
157         octa.vertices.push_back(Vector3( 1.0f,  0.0f,  0.0f));
158         octa.vertices.push_back(Vector3(-1.0f,  0.0f,  0.0f));
159         octa.vertices.push_back(Vector3( 0.0f,  1.0f,  0.0f));
160         octa.vertices.push_back(Vector3( 0.0f, -1.0f,  0.0f));
161         octa.vertices.push_back(Vector3( 0.0f,  0.0f,  1.0f));
162         octa.vertices.push_back(Vector3( 0.0f,  0.0f, -1.0f));
163 
164         octa.indices.push_back(4); octa.indices.push_back(0);
165         octa.indices.push_back(2); //Triangle 1
166         octa.indices.push_back(4); octa.indices.push_back(2);
167         octa.indices.push_back(1); //Triangle 2
168         octa.indices.push_back(4); octa.indices.push_back(1);
169         octa.indices.push_back(3); //Traingle 3
170         octa.indices.push_back(4); octa.indices.push_back(3);
171         octa.indices.push_back(0); //Triangle 4
172         octa.indices.push_back(5); octa.indices.push_back(2);
173         octa.indices.push_back(0); //Triangle 5
174         octa.indices.push_back(5); octa.indices.push_back(1);
175         octa.indices.push_back(2); //Triangle 6
176         octa.indices.push_back(5); octa.indices.push_back(3);
177         octa.indices.push_back(1); //Traingle 7
178         octa.indices.push_back(5); octa.indices.push_back(0);
179         octa.indices.push_back(3); //Triangle 8
180 
181         octa.outerRadius = 1.0f;
182         octa.innerRadius = 0.577350269189626f;
183 
```

```
    return octa;
}

//Data taken from: David Eberly - Platonic Solids (2008)
Mesh3 Mesh3::CreateDodecahedron()
{
    Mesh3 deca(20, 36);

    float a = 1.0f / sqrt(3.0f);
    float b = sqrt((3.0f - sqrt(5.0f)) / 6.0f);
    float c = sqrt((3.0f + sqrt(5.0f)) / 6.0f);

    deca.vertices.push_back(Vector3( a,  a,  a));
    deca.vertices.push_back(Vector3( a,  a, -a));
    deca.vertices.push_back(Vector3( a, -a,  a));
    deca.vertices.push_back(Vector3( a, -a, -a));
    deca.vertices.push_back(Vector3(-a,  a,  a));
    deca.vertices.push_back(Vector3(-a,  a, -a));
    deca.vertices.push_back(Vector3(-a, -a,  a));
    deca.vertices.push_back(Vector3(-a, -a, -a));
    deca.vertices.push_back(Vector3( b,  c, 0.0f));
    deca.vertices.push_back(Vector3(-b,  c, 0.0f));
    deca.vertices.push_back(Vector3( b, -c, 0.0f));
    deca.vertices.push_back(Vector3(-b, -c, 0.0f));
    deca.vertices.push_back(Vector3( c, 0.0f,  b));
    deca.vertices.push_back(Vector3( c, 0.0f, -b));
    deca.vertices.push_back(Vector3(-c, 0.0f,  b));
    deca.vertices.push_back(Vector3(-c, 0.0f, -b));
    deca.vertices.push_back(Vector3(0.0f,  b,  c));
    deca.vertices.push_back(Vector3(0.0f, -b,  c));
    deca.vertices.push_back(Vector3(0.0f,  b, -c));
    deca.vertices.push_back(Vector3(0.0f, -b, -c));

    deca.indices.push_back(0); deca.indices.push_back(8);
    deca.indices.push_back(9);   //Triangle 1
    deca.indices.push_back(0); deca.indices.push_back(9);
    deca.indices.push_back(4);   //Triangle 2
    deca.indices.push_back(0); deca.indices.push_back(4);
    deca.indices.push_back(16);  //Traingle 3
    deca.indices.push_back(0); deca.indices.push_back(12);
    deca.indices.push_back(13); //Triangle 4
    deca.indices.push_back(0); deca.indices.push_back(13);
    deca.indices.push_back(1);  //Triangle 5
    deca.indices.push_back(0); deca.indices.push_back(1);
    deca.indices.push_back(8);   //Triangle 6
    deca.indices.push_back(0); deca.indices.push_back(16);
    deca.indices.push_back(17); //Traingle 7
    deca.indices.push_back(0); deca.indices.push_back(17);
    deca.indices.push_back(2); //Triangle 8
    deca.indices.push_back(0); deca.indices.push_back(2);
    deca.indices.push_back(12); //Triangle 9
    deca.indices.push_back(8); deca.indices.push_back(1);
    deca.indices.push_back(18); //Triangle 10
    deca.indices.push_back(8); deca.indices.push_back(18);
    deca.indices.push_back(5); //Traingle 11
    deca.indices.push_back(8); deca.indices.push_back(5);
    deca.indices.push_back(9); //Triangle 12
```

```
241         deca.indices.push_back(12); deca.indices.push_back(2);
242         deca.indices.push_back(10); //Triangle 13
243         deca.indices.push_back(12); deca.indices.push_back(10);
244         deca.indices.push_back(3); //Triangle 14
245         deca.indices.push_back(12); deca.indices.push_back(3);
246         deca.indices.push_back(13); //Traingle 15
247         deca.indices.push_back(16); deca.indices.push_back(4);
248         deca.indices.push_back(14); //Triangle 16
249         deca.indices.push_back(16); deca.indices.push_back(14);
250         deca.indices.push_back(6); //Triangle 17
251         deca.indices.push_back(16); deca.indices.push_back(6);
252         deca.indices.push_back(17); //Triangle 18
253         deca.indices.push_back(9); deca.indices.push_back(5);
254         deca.indices.push_back(15); //Traingle 19
255         deca.indices.push_back(9); deca.indices.push_back(15);
256         deca.indices.push_back(14); //Triangle 20
257         deca.indices.push_back(9); deca.indices.push_back(14);
258         deca.indices.push_back(4); //Triangle 21
259         deca.indices.push_back(6); deca.indices.push_back(11);
260         deca.indices.push_back(10); //Triangle 22
261         deca.indices.push_back(6); deca.indices.push_back(10);
262         deca.indices.push_back(2); //Traingle 23
263         deca.indices.push_back(6); deca.indices.push_back(2);
264         deca.indices.push_back(17); //Triangle 24
265         deca.indices.push_back(3); deca.indices.push_back(19);
266         deca.indices.push_back(18); //Triangle 25
267         deca.indices.push_back(3); deca.indices.push_back(18);
268         deca.indices.push_back(1); //Triangle 26
269         deca.indices.push_back(3); deca.indices.push_back(1);
270         deca.indices.push_back(13); //Traingle 27
271         deca.indices.push_back(7); deca.indices.push_back(15);
272         deca.indices.push_back(5); //Triangle 28
273         deca.indices.push_back(7); deca.indices.push_back(5);
274         deca.indices.push_back(18); //Triangle 29
275         deca.indices.push_back(7); deca.indices.push_back(18);
276         deca.indices.push_back(19); //Triangle 30
277         deca.indices.push_back(7); deca.indices.push_back(11);
278         deca.indices.push_back(6); //Traingle 31
279         deca.indices.push_back(7); deca.indices.push_back(6);
280         deca.indices.push_back(14); //Triangle 32
281         deca.indices.push_back(7); deca.indices.push_back(14);
282         deca.indices.push_back(15); //Triangle 33
283         deca.indices.push_back(7); deca.indices.push_back(19);
284         deca.indices.push_back(3); //Triangle 34
285         deca.indices.push_back(7); deca.indices.push_back(3);
286         deca.indices.push_back(10); //Traingle 35
287         deca.indices.push_back(7); deca.indices.push_back(10);
288         deca.indices.push_back(11); //Triangle 36
289
290         deca.outerRadius = 1.0f;
291         deca.innerRadius = 0.794654472291766f;
292
293         return deca;
294     }
295
296     //Data taken from: David Eberly - Platonic Solids (2008)
297     Mesh3 Mesh3::CreateIcosahedron()
```

```
298   {
299       Mesh3 ico(12, 20);
300
301       float t = (1 + sqrt(5.0f)) * 0.5f;
302       float s = sqrt(1 + t * t);
303
304       ico.vertices.push_back(Vector3( t/s,  1.0f/s, 0.0f));
305       ico.vertices.push_back(Vector3(-t/s,  1.0f/s, 0.0f));
306       ico.vertices.push_back(Vector3( t/s, -1.0f/s, 0.0f));
307       ico.vertices.push_back(Vector3(-t/s, -1.0f/s, 0.0f));
308       ico.vertices.push_back(Vector3( 1.0f/s, 0.0f,  t/s));
309       ico.vertices.push_back(Vector3( 1.0f/s, 0.0f, -t/s));
310       ico.vertices.push_back(Vector3(-1.0f/s, 0.0f,  t/s));
311       ico.vertices.push_back(Vector3(-1.0f/s, 0.0f, -t/s));
312       ico.vertices.push_back(Vector3(0.0f,  t/s,  1.0f/s));
313       ico.vertices.push_back(Vector3(0.0f, -t/s,  1.0f/s));
314       ico.vertices.push_back(Vector3(0.0f,  t/s, -1.0f/s));
315       ico.vertices.push_back(Vector3(0.0f, -t/s, -1.0f/s));
316
317       ico.indices.push_back(0); ico.indices.push_back(8);
318       ico.indices.push_back(4);   //Triangle 1
319       ico.indices.push_back(0); ico.indices.push_back(5);
320       ico.indices.push_back(10);   //Triangle 2
321       ico.indices.push_back(2); ico.indices.push_back(4);
322       ico.indices.push_back(9);  //Traingle 3
323       ico.indices.push_back(2); ico.indices.push_back(11);
324       ico.indices.push_back(5); //Triangle 4
325       ico.indices.push_back(1); ico.indices.push_back(6);
326       ico.indices.push_back(8);  //Triangle 5
327       ico.indices.push_back(1); ico.indices.push_back(10);
328       ico.indices.push_back(7);   //Triangle 6
329       ico.indices.push_back(3); ico.indices.push_back(9);
330       ico.indices.push_back(6); //Traingle 7
331       ico.indices.push_back(3); ico.indices.push_back(7);
332       ico.indices.push_back(11); //Triangle 8
333       ico.indices.push_back(0); ico.indices.push_back(10);
334       ico.indices.push_back(8); //Triangle 9
335       ico.indices.push_back(1); ico.indices.push_back(8);
336       ico.indices.push_back(10); //Triangle 10
337       ico.indices.push_back(2); ico.indices.push_back(9);
338       ico.indices.push_back(11); //Traingle 11
339       ico.indices.push_back(3); ico.indices.push_back(11);
340       ico.indices.push_back(9); //Triangle 12
341       ico.indices.push_back(4); ico.indices.push_back(2);
342       ico.indices.push_back(0); //Triangle 13
343       ico.indices.push_back(5); ico.indices.push_back(0);
344       ico.indices.push_back(2); //Triangle 14
345       ico.indices.push_back(6); ico.indices.push_back(1);
346       ico.indices.push_back(3); //Traingle 15
347       ico.indices.push_back(7); ico.indices.push_back(3);
348       ico.indices.push_back(1); //Triangle 16
349       ico.indices.push_back(8); ico.indices.push_back(6);
350       ico.indices.push_back(4); //Triangle 17
351       ico.indices.push_back(9); ico.indices.push_back(4);
352       ico.indices.push_back(6); //Triangle 18
353       ico.indices.push_back(10); ico.indices.push_back(5);
354       ico.indices.push_back(7); //Traingle 19
```

```
355         ico.indices.push_back(11); ico.indices.push_back(7);
356         ico.indices.push_back(5); //Triangle 20
357 
358         ico.outerRadius = 1.0f;
359         ico.innerRadius = 0.794654472291766f;
360 
361         return ico;
362     }
363 
364     Mesh3 Mesh3::CreateLathe(CrossSection& crossSection, size_t segmentCount)
365     {
366         size_t crossSectionCount = crossSection.getCount();
367 
368         Mesh3 mesh(segmentCount * crossSectionCount,
369             2 * segmentCount * crossSectionCount);
370 
371         vector<Vector3> generator(crossSectionCount);
372 
373         for (size_t i = 0; i < crossSectionCount; i++)
374         {
375             generator[i].x = crossSection[i].x;
376             generator[i].y = 0;
377             generator[i].z = crossSection[i].y;
378         }
379 
380         float phi = TWO_PI_F / segmentCount;
381 
382         for (size_t i = 0; i < segmentCount; i++)
383         {
384             Matrix matrix = Matrix::CreateRotationZ(phi * i);
385             for (size_t j = 0; j < crossSectionCount; j++)
386             {
387                 Vector3 point = Vector3::Transform(generator[j], matrix);
388                 mesh.vertices.push_back(point);
389             }
390         }
391 
392         for (size_t i = 0; i < segmentCount - 1; i++)
393         {
394             for (size_t j = 1; j < crossSectionCount; j++)
395             {
396                 //Triangle 1
397                 mesh.indices.push_back(crossSectionCount * i + j);
398                 mesh.indices.push_back(crossSectionCount * (i + 1) + j - 1);
399                 mesh.indices.push_back(crossSectionCount * i + j - 1);
400 
401                 //Triangle 2
402                 mesh.indices.push_back(crossSectionCount * (i + 1) + j - 1);
403                 mesh.indices.push_back(crossSectionCount * i + j);
404                 mesh.indices.push_back(crossSectionCount * (i + 1) + j);
405             }
406 
407             //Triangle 1
408             mesh.indices.push_back(crossSectionCount * i);
409             mesh.indices.push_back(crossSectionCount * (i + 2) - 1);
410             mesh.indices.push_back(crossSectionCount * (i + 1) - 1);
411 
```

```
412             //Triangle 2
413             mesh.indices.push_back(crossSectionCount * (i + 2) - 1);
414             mesh.indices.push_back(crossSectionCount * i);
415             mesh.indices.push_back(crossSectionCount * (i + 1));
416         }
417
418         for (size_t j = 1; j < crossSectionCount; j++)
419         {
420             //Triangle 1
421             mesh.indices.push_back(crossSectionCount * (segmentCount - 1) + j);
422             mesh.indices.push_back(j - 1);
423             mesh.indices.push_back(crossSectionCount * (segmentCount - 1) + j -
424     1);
425
426             //Triangle 2
427             mesh.indices.push_back(j - 1);
428             mesh.indices.push_back(crossSectionCount * (segmentCount - 1) + j);
429             mesh.indices.push_back(j);
430         }
431
432         //Triangle 1
433         mesh.indices.push_back(crossSectionCount * (segmentCount - 1));
434         mesh.indices.push_back(crossSectionCount - 1);
435         mesh.indices.push_back(crossSectionCount * segmentCount - 1);
436
437         //Triangle 2
438         mesh.indices.push_back(crossSectionCount - 1);
439         mesh.indices.push_back(crossSectionCount * (segmentCount - 1));
440         mesh.indices.push_back(0);
441
442         return mesh;
443     }
444
445     void Mesh3::writeObj(std::wstring fileName, bool onlyVisibleFaces)
446     {
447         ofstream file;
448         file.open(fileName);
449
450         size_t vertexCount = vertices.size();
451         for (size_t i = 0; i < vertexCount; i++)
452             file << "v " << vertices[i].x << " " << vertices[i].y
453             << " " << vertices[i].z << endl;
454
455         size_t triangleCount = indices.size() / 3;
456         for (size_t i = 0; i < triangleCount; i++)
457         {
458             if (onlyVisibleFaces && (visibility[i] == 0))
459                 continue;
460
461             file << "f " << indices[3*i]   + 1 << "//" << indices[3*i]   + 1
462                     << " "  << indices[3*i+1] + 1 << "//" << indices[3*i+1] + 1
463                     << " "  << indices[3*i+2] + 1 << "//"
464                     << indices[3*i+2] + 1 << endl;
465         }
466         file.close();
467     }
468
```

```
469  size_t Mesh3::findIndex(const Vector3& point, size_t lower, size_t upper)
470  {
471      for (size_t i = lower; i < upper; i++)
472          if (vertices[i] == point)
473              return i;
474
475      return upper;
476  }
477
478  void Mesh3::subdivide(size_t level, SubdivisionType type)
479  {
480      switch (type)
481      {
482          case FrequencySubdivision:
483              subdivideByFrequency(level);
484              break;
485          case DepthSubdivision:
486              subdivideByDepth(level);
487              break;
488      }
489  }
490
491  void Mesh3::subdivideByFrequency(size_t level)
492  {
493      //Total number of vertices and triangles:
494      // V = (generator.VertexCount - 2) * subdivisionLevel^2 + 2;
495      // T = generator.TriangleCount * subdivisionLevel^2;
496      size_t vertexCount = (vertices.size() - 2) * level * level + 2;
497      size_t indicesCount = indices.size() * level * level;
498
499      vector<size_t> oldIndices = indices;
500      indices.clear();
501
502      vertices.reserve(vertexCount);
503      indices.reserve(indicesCount);
504
505      Vector3* prev;
506      Vector3* curr;
507      size_t* prevIndices;
508      size_t* currIndices;
509
510      for (auto it = oldIndices.begin(); it != oldIndices.end(); it += 3)
511      {
512          Vector3 A = vertices[*it];
513          Vector3 B = vertices[*(it+1)];
514          Vector3 C = vertices[*(it+2)];
515
516          //We are starting from one of the vertices.
517          prev = new Vector3[1];
518          prev[0] = A;
519          prevIndices = new size_t[1];
520          prevIndices[0] = *it;
521
522          //For each subdivision level we slice the triangle with the
523          //line parallel to the side BC.
524          for (unsigned short j = 1; j < level; j++)
525          {
```

```
526             curr = new Vector3[j+1];
527             currIndices = new size_t[j+1];
528
529             float t = static_cast<float>(j) / static_cast<float>(level);
530             curr[0] = (1.0f - t) * A + t * B;
531
532
533             size_t index;
534             //Find if we already had the first vertex
535             index = findIndex(curr[0], 0, vertices.size());
536             if (index == vertices.size())
537                 vertices.push_back(curr[0]);
538             currIndices[0] = index;
539
540             //we should do the same for the last vertex;
541             curr[j] = (1.0f - t) * A + t * C;
542
543             index = findIndex(curr[j], 0, vertices.size());
544             if (index == vertices.size())
545                 vertices.push_back(curr[j]);
546             currIndices[j] = index;
547
548             for(int k = 1; k < j; k++)
549             {
550                 float s = static_cast<float>(k) / static_cast<float>(j);
551                 curr[k] = (1.0f - s) * curr[0] + s * curr[j];
552
553                 //we know that these vertices are unique, so we skip the
554                 //FindIndex call.
555                 currIndices[k] = vertices.size();
556                 vertices.push_back(curr[k]);
557             }
558
559             //All that is left is to create triangles..
560             //in the first pass we create triangles that are facing to the
561             //previus array.
562             for(int k = 0; k < j; k++)
563             {
564                 indices.push_back(currIndices[k]);
565                 indices.push_back(currIndices[k+1]);
566                 indices.push_back(prevIndices[k]);
567             }
568
569             //and now all those who face the current array.
570             for(int k = 0; k < j - 1; k++)
571             {
572                 indices.push_back(prevIndices[k]);
573                 indices.push_back(currIndices[k+1]);
574                 indices.push_back(prevIndices[k+1]);
575             }
576
577             //Clean up;
578             delete[] prev;
579             delete[] prevIndices;
580
581             prev = curr;
582             prevIndices = currIndices;
```

```
583             }
584 
585             //and now, we should subdivide the BC side.
586             curr = new Vector3[level+1];
587             currIndices = new size_t[level+1];
588 
589             curr[0] = B;
590             currIndices[0] = *(it+1);
591 
592             //we should do the same for the last vertex;
593             curr[level] = C;
594             currIndices[level] = *(it+2);
595 
596             for(unsigned short k = 1; k < level; k++)
597             {
598                 float s = static_cast<float>(k) / static_cast<float>(level);
599 
600                 curr[k] = (1.0f - s) * curr[0] + s * curr[level];
601 
602                 size_t index = findIndex(curr[k], 0, vertices.size());
603                 if (index == vertices.size())
604                     vertices.push_back(curr[k]);
605                 currIndices[k] = index;
606             }
607 
608             //All that is left is to create triangles..
609             //in the first pass we create triangles that are facing to the
610             //previus array.
611             for(unsigned short k = 0; k < level; k++)
612             {
613                 indices.push_back(currIndices[k]);
614                 indices.push_back(currIndices[k+1]);
615                 indices.push_back(prevIndices[k]);
616             }
617             //and now all those who face the current array.
618             for(size_t k = 0; k < level - 1; k++)
619             {
620                 indices.push_back(prevIndices[k]);
621                 indices.push_back(currIndices[k+1]);
622                 indices.push_back(prevIndices[k+1]);
623             }
624             //Clean up;
625             delete[] prev;
626             delete[] prevIndices;
627 
628             delete[] curr;
629             delete[] currIndices;
630         }
631     }
632 
633     void Mesh3::subdivideByDepth(size_t level)
634     {
635         //Total number of vertices and triangles:
636         // V = (generator.VertexCount - 2) * 4^subdivisionLevel + 2;
637         // T = generator.TriangleCount * 4^subdivisionLevel;
638         size_t oldVertexCount = vertices.size();
639         size_t oldTriangleCount = indices.size();
```

```
640
641         size_t vertexCount = vertices.size() - 2;
642         size_t indexCount = indices.size();
643
644         for (unsigned short i = 0; i < level; i++)
645         {
646             vertexCount *= 4;
647             indexCount *= 4;
648         }
649
650         vertexCount += 2;
651
652         vertices.reserve(vertexCount);
653         indices.reserve(indexCount);
654
655         ////For each subdivision level...
656         size_t currentVertex = vertices.size();
657
658         for (unsigned short n = 1; n <= level; n++)
659         {
660             size_t currentVertexCount = vertices.size();
661             size_t currentIndexCount = indices.size();
662
663             //For each triangle at the current subdivision level...
664             for (size_t i = 0; i < currentIndexCount; i+= 3)
665             {
666                 Vector3 A = vertices[indices[i]];
667                 Vector3 B = vertices[indices[i+1]];
668                 Vector3 C = vertices[indices[i+2]];
669
670                 //calculate the midpoints
671                 Vector3 AB = 0.5 * (A + B);
672                 Vector3 AC = 0.5 * (A + C);
673                 Vector3 BC = 0.5 * (B + C);
674
675                 //Find if we already had that vertex
676                 size_t indexAB = findIndex(AB, currentVertexCount,
677     vertices.size());
678                 if (indexAB == vertices.size())
679                     vertices.push_back(AB);
680
681                 size_t indexAC = findIndex(AC, currentVertexCount,
682     vertices.size());
683                 if (indexAC == vertices.size())
684                     vertices.push_back(AC);
685
686                 size_t indexBC = findIndex(BC, currentVertexCount,
687     vertices.size());
688                 if (indexBC == vertices.size())
689                     vertices.push_back(BC);
690
691                 //Update the cuurent triangle...
692                 size_t indexA = indices[i];
693                 size_t indexB = indices[i+1];
694                 size_t indexC = indices[i+2];
695
696                 //Replace the original triangle
```

```
            indices[i] = indexA;
            indices[i+1] = indexAB;
            indices[i+2] = indexAC;

            //Triangle 1
            indices.push_back(indexAB);
            indices.push_back(indexB);
            indices.push_back(indexBC);

            //Triangle 2
            indices.push_back(indexAC);
            indices.push_back(indexBC);
            indices.push_back(indexC);

            //Triangle 3
            indices.push_back(indexAB);
            indices.push_back(indexBC);
            indices.push_back(indexAC);
        }
    }
}

void Mesh3::pushTo(Surface* surface)
{
    outerRadius = surface->getOuterRadius();
    innerRadius = surface->getInnerRadius();

    for (auto it = vertices.begin(); it != vertices.end(); it++)
    {
        float phi = atan2(it->y, it->x);
        phi = (phi >= 0) ? phi : phi + TWO_PI_F;
        float theta = acos(it->z / it->Length());

        float r = surface->getRadius(theta, phi);

        it->x = r * sin(theta) * cos(phi);
        it->y = r * sin(theta) * sin(phi);
        it->z = r * cos(theta);
    }
}

void Mesh3::init(size_t passbandCount)
{
    lcCount = passbandCount;
    size_t triangleCount = indices.size() / 3;

    area.resize(triangleCount);
    projectedArea.resize(triangleCount);

    normals.resize(triangleCount);
    centroids.resize(triangleCount);

    temperature.resize(triangleCount);
    gravity.resize(triangleCount);
    flux.resize(triangleCount * lcCount);
    velocity.resize(triangleCount);

```

```
754         visibility.resize(triangleCount);
755 
756         totalArea = 0.0;
757         totalVolume = 0.0;
758 
759         for (size_t i = 0; i < triangleCount; ++i)
760         {
761             Vector3 sideCA = vertices[indices[3*i+2]] - vertices[indices[3*i]];
762             Vector3 sideBA = vertices[indices[3*i+1]] - vertices[indices[3*i]];
763 
764             normals[i] = sideBA.Cross(sideCA);
765             centroids[i] = (vertices[indices[3*i]] + vertices[indices[3*i+1]]
766             + vertices[indices[3*i+2]]) / 3.0;
767 
768             area[i] = 0.5 * normals[i].Length();
769             totalArea += area[i];
770             totalVolume += vertices[indices[3*i]].Dot(
771                 vertices[indices[3*i+1]].Cross(vertices[indices[3*i+2]])) / 6.0;
772 
773             normals[i].Normalize();
774         }
775     }
776 
777     void Mesh3::adaptive(size_t level, Vector3* c, double r0, double r1)
778     {
779         //For each subdivision level...
780         size_t currentVertex = vertices.size();
781 
782         for (unsigned short n = 1; n <= level; n++)
783         {
784             size_t currentVertexCount = vertices.size();
785             size_t currentIndexCount = indices.size();
786 
787             //For each triangle at the current subdivision level...
788             for (size_t i = 0; i < currentIndexCount; i+= 3)
789             {
790                 size_t triangleIndex = i / 3;
791 
792                 float distance = pow2(centroids[triangleIndex].y - c->y)
793                     + pow2(centroids[triangleIndex].z - c->z);
794 
795                 if (centroids[triangleIndex].x > center.x && distance > pow2(r0)
796                     && distance < pow2(r1))
797                 {
798                     Vector3 A = vertices[indices[i]];
799                     Vector3 B = vertices[indices[i+1]];
800                     Vector3 C = vertices[indices[i+2]];
801 
802                     //calculate the midpoints
803                     Vector3 AB = 0.5 * (A + B);
804                     Vector3 AC = 0.5 * (A + C);
805                     Vector3 BC = 0.5 * (B + C);
806 
807                     //Find if we already had that vertex
808                     size_t indexAB = findIndex(AB, currentVertexCount,
809                         vertices.size());
810                     if (indexAB == vertices.size())
```

```
811                     vertices.push_back(AB);
812
813                 size_t indexAC = findIndex(AC, currentVertexCount,
814                     vertices.size());
815                 if (indexAC == vertices.size())
816                     vertices.push_back(AC);
817
818                 size_t indexBC = findIndex(BC, currentVertexCount,
819                     vertices.size());
820                 if (indexBC == vertices.size())
821                     vertices.push_back(BC);
822
823                 //Update the cuurent triangle...
824                 size_t indexA = indices[i];
825                 size_t indexB = indices[i+1];
826                 size_t indexC = indices[i+2];
827
828                 //Replace the original triangle
829                 indices[i] = indexA;
830                 indices[i+1] = indexAB;
831                 indices[i+2] = indexAC;
832
833                 double temp = temperature[triangleIndex];
834                 double grav = gravity[triangleIndex];
835                 Vector3 vel = velocity[triangleIndex];
836
837                 //Calc new area, centroid and normal
838                 Vector3 sideCA = AC - A;
839                 Vector3 sideBA = AB - A;
840
841                 Vector3 n = sideBA.Cross(sideCA);
842                 double a = 0.5 * n.Length();
843                 n.Normalize();
844
845                 centroids[triangleIndex] = (A + AB + AC) / 3.0;
846                 normals[triangleIndex]= n;
847                 area[triangleIndex] = a;
848
849                 //Triangle 1
850                 indices.push_back(indexAB);
851                 indices.push_back(indexB);
852                 indices.push_back(indexBC);
853
854                 //Calc new area, centroid and normal
855                 sideCA = BC - AB;
856                 sideBA = B - AB;
857
858                 n = sideBA.Cross(sideCA);
859                 a = 0.5 * n.Length();
860                 n.Normalize();
861
862                 centroids.push_back((AB + B + BC) / 3.0);
863                 normals.push_back(n);
864                 area.push_back(a);
865
866                 temperature.push_back(temp);
867                 gravity.push_back(grav);
```

```
                velocity.push_back(vel);

                //Triangle 2
                indices.push_back(indexAC);
                indices.push_back(indexBC);
                indices.push_back(indexC);

                //Calc new area, centroid and normal
                sideCA = C - AC;
                sideBA = BC - AC;

                n = sideBA.Cross(sideCA);
                a = 0.5 * n.Length();
                n.Normalize();

                centroids.push_back((AC + BC + C) / 3.0);
                normals.push_back(n);
                area.push_back(a);

                temperature.push_back(temp);
                gravity.push_back(grav);
                velocity.push_back(vel);

                //Triangle 3
                indices.push_back(indexAB);
                indices.push_back(indexBC);
                indices.push_back(indexAC);

                //Calc new area, centroid and normal
                sideCA = AC - AB;
                sideBA = BC - AB;

                n = sideBA.Cross(sideCA);
                a = 0.5 * n.Length();
                n.Normalize();

                centroids.push_back((AB + BC + AC) / 3.0);
                normals.push_back(n);
                area.push_back(a);

                temperature.push_back(temp);
                gravity.push_back(grav);
                velocity.push_back(vel);
            }
        }
    }

    visibility.resize(indices.size()/3);
    projectedArea.resize(indices.size()/3);
    flux.resize(indices.size()/3 * lcCount);
} }
```

Triangle2.h

```
#pragma once

#include "SimpleMath.h"
#include "Circle.h"
#include "Mesh3.h"
#include <vector>

namespace Infinity {
class Triangle2
{

public:
    Triangle2(Mesh3* parent, size_t index, double distance);
    ~Triangle2(void);

    const DirectX::SimpleMath::Vector2* const getCentroid();
    const Circle* getCircumCircle() const;

    double intersectsWith(const Triangle2* triangle, int method) const;
    double getDistance() const;
    double getArea() const;
    size_t getIndex() const;
    Mesh3* getParent() const;

    bool intersects(Triangle2* second, int method);

private:
    void projectToAxis(DirectX::SimpleMath::Vector2& axis,
        float* min, float* max);
    bool intersectsFull(Triangle2* second);
    bool intersectsPoint(Triangle2* second);

    Mesh3* parent;

    DirectX::SimpleMath::Vector2 centroid;
    Circle circumcircle;

    DirectX::SimpleMath::Vector2 points[3];
    DirectX::SimpleMath::Vector2 edges[3];

    double distance;
    double area;

    size_t index;
};

inline const Circle* Triangle2::getCircumCircle() const
{ return &circumcircle; }
inline double Triangle2::getDistance() const { return distance; }
inline double Triangle2::getArea() const { return area; }
inline size_t Triangle2::getIndex() const { return index; }
inline const DirectX::SimpleMath::Vector2* const Triangle2::getCentroid()
{ return &centroid; }
inline Mesh3* Triangle2::getParent() const { return parent; } }
```

Triangle2.cpp

```
#include "stdafx.h"
#include "Triangle2.h"

using namespace DirectX::SimpleMath;

namespace Infinity {
Triangle2::Triangle2(Mesh3* parent, size_t index, double distance)
    : parent(parent), index(index), distance(distance)
{
    points[0] = Vector2(parent->getPoint(3*index)->y,
        parent->getPoint(3*index)->z);
    points[1] = Vector2(parent->getPoint(3*index+1)->y,
        parent->getPoint(3*index+1)->z);
    points[2] = Vector2(parent->getPoint(3*index+2)->y,
        parent->getPoint(3*index+2)->z);

    edges[0] = points[1] - points[0];
    edges[1] = points[2] - points[1];
    edges[2] = points[0] - points[2];

    area = 0.5 * abs(edges[0].x * edges[2].y - edges[2].x * edges[0].y);
    circumcircle = Circle::GetBoundingCircle(points[0], points[1],
points[2]);
    centroid = (points[0] + points[1] + points[2]) / 3.0;
}

Triangle2::~Triangle2(void)
{ }

bool Triangle2::intersects(Triangle2* second, int method)
{
    //if two triangles are part of the same mesh, and share at least one
    //index, they will not intersect.
    if ((parent == second->parent) &&
        ((parent->getIndex(3*index  )
        == second->parent->getIndex(3*second->index))
        || (parent->getIndex(3*index  )
        == second->parent->getIndex(3*second->index+1))
        || (parent->getIndex(3*index)
        == second->parent->getIndex(3*second->index+2)) ||
        ( parent->getIndex(3*index+1)
        == second->parent->getIndex(3*second->index))
        || (parent->getIndex(3*index+1)
        == second->parent->getIndex(3*second->index+1))
        || (parent->getIndex(3*index+1)
        == second->parent->getIndex(3*second->index+2)) ||
        ( parent->getIndex(3*index+2)
        == second->parent->getIndex(3*second->index))
        || (parent->getIndex(3*index+2)
        == second->parent->getIndex(3*second->index+1))
        || (parent->getIndex(3*index+2)
        == second->parent->getIndex(3*second->index+2))))
            return false;
```

```
54
55      //First we will check the bounding circles
56      if (!Circle::Intersects(&circumcircle, &second->circumcircle))
57          return false;
58
59      if (method == TRIANGLE_FULL)
60          return intersectsFull(second);
61      else
62          return intersectsPoint(second);
63  }
64
65  bool Triangle2::intersectsFull(Triangle2* second)
66  {
67      //And then use the separating axis theorem to see if there is an
68  intersection
69      //http://www.codeproject.com/KB/GDI-plus/PolygonCollision.aspx
70
71      // Loop through all the edges of both polygons
72      int totalEdgeCount = 6;
73      Vector2 edge, axis;
74
75      // Loop through all the edges of both polygons
76      for (int i = 0; i < totalEdgeCount; i++)
77      {
78          edge = (i < 3) ? edges[i] : second->edges[i - 3];
79
80          // Find the axis perpendicular to the current edge
81          axis.x = -edge.y;
82          axis.y =  edge.x;
83          axis.Normalize();
84
85          // Find the projection of the polygon on the current axis
86          float minFirst = 0; float minSecond = 0;
87          float maxFirst = 0; float maxSecond = 0;
88          projectToAxis(axis, &minFirst, &maxFirst);
89          second->projectToAxis(axis, &minSecond, &maxSecond);
90
91          // Check if the polygon projections are currently intersecting
92          float t0 = maxSecond - minFirst;
93          float t1 = maxFirst - minSecond;
94          if (abs(t0 - t1) > t0 + t1 - 16 *
95  std::numeric_limits<float>::epsilon())
96              return false;
97      }
98
99      return true;
100 }
101
102 //Algorithm taken from http://www.blackpawn.com/texts/pointinpoly/
103 //See also Real-Time Collision Detection, page 204 and
104 //http://stackoverflow.com/questions/2049582/how-to-determine-a-point-in-a-
105 triangle
106 bool Triangle2::intersectsPoint(Triangle2* second)
107 {
108     Vector2 v0 = second->points[2] - second->points[0];
109     Vector2 v1 = second->points[1] - second->points[0];
110     Vector2 v2 = centroid - second->points[0];
```

```
111
112         float dot00 = v0.Dot(v0);
113         float dot01 = v0.Dot(v1);
114         float dot02 = v0.Dot(v2);
115         float dot11 = v1.Dot(v1);
116         float dot12 = v1.Dot(v2);
117
118         float invDenom = 1 / (dot00 * dot11 - dot01 * dot01);
119         float u = (dot11 * dot02 - dot01 * dot12) * invDenom;
120         float v = (dot00 * dot12 - dot01 * dot02) * invDenom;
121
122         // Check if point is in triangle
123         return (u >= 0) && (v >= 0) && (u + v < 1);
124     }
125
126     void Triangle2::projectToAxis(Vector2 &axis, float* minimum, float* maximum)
127     {
128         float projectionA = axis.Dot(points[0]);
129         float projectionB = axis.Dot(points[1]);
130         float projectionC = axis.Dot(points[2]);
131
132         *maximum = max(max(projectionA, projectionB), projectionC);
133         *minimum = min(min(projectionA, projectionB), projectionC);
134     } }
```

8.3. Račun fluksa

ATMRow.h

```
 1   #pragma once
 2
 3   #include <vector>
 4   #include "LegendrePolynomial.h"
 5   #include "Calc.h"
 6
 7   namespace Infinity {
 8   class ATMRow
 9   {
10   public:
11       ATMRow(double low, double high, double p0, double p1, double p2,
12           double p3, double p4, double p5, double p6, double p7,
13           double p8, double p9);
14       ATMRow(const ATMRow& source);
15       ~ATMRow(void);
16
17       double getLow();
18       double getHigh();
19       double getIntensity(double temp);
20
21       ATMRow& operator=(const ATMRow& source);
22   private:
23       double low, high;
24       std::vector<double> coefs;
25
26       LegendrePolynomial p0, p1, p2, p3, p4, p5, p6, p7, p8, p9;
```

```
};

inline double ATMRow::getLow() { return low; }
inline double ATMRow::getHigh() { return high; }

inline double ATMRow::getIntensity(double temperature)
{
    double normTemp = (temperature - low) / (high - low);

    double result = coefs[0] * p0(normTemp) + coefs[1] * p1(normTemp)
        + coefs[2] * p2(normTemp) + coefs[3] * p3(normTemp)
        + coefs[4] * p4(normTemp) + coefs[5] * p5(normTemp)
        + coefs[6] * p6(normTemp) + coefs[7] * p7(normTemp)
        + coefs[8] * p8(normTemp) + coefs[9] * p9(normTemp);

    return pow(10.0, result);
} }
```

ATMRow.cpp

```
#include "stdafx.h"
#include "ATMRow.h"

namespace Infinity {
ATMRow::ATMRow(double low, double high, double c0, double c1,
    double c2, double c3, double c4, double c5, double c6,
    double c7, double c8, double c9) :
    low(low),
    high(high),
    coefs(10),
    p0(0), p1(1), p2(2), p3(3), p4(4), p5(5), p6(6), p7(7), p8(8), p9(9)
{
    coefs[0] = c0;
    coefs[1] = c1;
    coefs[2] = c2;
    coefs[3] = c3;
    coefs[4] = c4;
    coefs[5] = c5;
    coefs[6] = c6;
    coefs[7] = c7;
    coefs[8] = c8;
    coefs[9] = c9;
}

ATMRow::ATMRow(const ATMRow& source) :
    low(source.low),
    high(source.high),
    coefs(source.coefs),
    p0(0), p1(1), p2(2), p3(3), p4(4), p5(5), p6(6), p7(7), p8(8), p9(9)
{ }

ATMRow& ATMRow::operator=(const ATMRow& source)
{
    low = source.low;
    high = source.high;
    coefs = source.coefs;
```

```
37
38      p0 = LegendrePolynomial(0);
39      p1 = LegendrePolynomial(1);
40      p2 = LegendrePolynomial(2);
41      p3 = LegendrePolynomial(3);
42      p4 = LegendrePolynomial(4);
43      p5 = LegendrePolynomial(5);
44      p6 = LegendrePolynomial(6);
45      p7 = LegendrePolynomial(7);
46      p8 = LegendrePolynomial(8);
47      p9 = LegendrePolynomial(9);
48
49      return *this;
50  }
51
52  ATMRow::~ATMRow(void)
53  { } }
```

ATMTable.h

```
1   #pragma once

2   #include "ATMRow.h"
3   #include <vector>
4   #include <string>
5   #include <fstream>
6   #include <sstream>
7
8   namespace Infinity {
9   class ATMTable
10  {
11  public:
12      ATMTable(double metalicity, std::wstring passband);
13      ATMTable(const ATMTable& source);
14      ATMTable& operator=(const ATMTable& source);
15
16      ~ATMTable(void);
17
18      double getIntensity(double temperature, double logG);
19  private:
20      std::vector<ATMRow> rows;
21
22      std::wstring getMetalicityString(double metalicity);
23      int getIndex(double logG);
24      double getLogG(int index);
25  };
26
27  inline double ATMTable::getIntensity(double temperature, double logG)
28  {
29      int index = getIndex(logG);
30      double intensity0, intensity1;
31      double amount = (logG - index / 8.0) / 0.5; //Empirical formula :)
32
33      //determine where is the temperature...
34      if (temperature < rows[index].getHigh())
35          intensity0 = rows[index].getIntensity(temperature);
```

```
    if (temperature >= rows[index+1].getLow()
        && temperature < rows[index+1].getHigh())
        intensity0 = rows[index+1].getIntensity(temperature);
    if (temperature >= rows[index+2].getLow()
        && temperature < rows[index+2].getHigh())
        intensity0 = rows[index+2].getIntensity(temperature);
    if (temperature >= rows[index+3].getLow())
        intensity0 = rows[index+3].getIntensity(temperature);

    if (temperature < rows[index+4].getHigh())
        intensity1 = rows[index+4].getIntensity(temperature);
    if (temperature >= rows[index+5].getLow()
        && temperature < rows[index+5].getHigh())
        intensity1 = rows[index+5].getIntensity(temperature);
    if (temperature >= rows[index+6].getLow()
        && temperature < rows[index+6].getHigh())
        intensity1 = rows[index+6].getIntensity(temperature);
    if (temperature >= rows[index+7].getLow())
        intensity1 = rows[index+7].getIntensity(temperature);

    return (1-amount) * intensity0 + amount * intensity1;
} }
```

ATMTable.cpp

```
#include "stdafx.h"
#include "ATMTable.h"

using namespace std;

namespace Infinity {

ATMTable::ATMTable(double metalicity, std::wstring passband)
{
    wstringstream ssPrim;
    ssPrim << L"atmcof/" << getMetalicityString(metalicity)
        << L"." << passband;

    ifstream file(ssPrim.str());

    double low, high;
    double c0, c1, c2, c3, c4, c5, c6, c7, c8, c9;
    while(!file.eof())
    {
        file >> low;
        file >> high;
        file >> c0;
        file >> c1;
        file >> c2;
        file >> c3;
        file >> c4;
        file >> c5;
        file >> c6;
        file >> c7;
        file >> c8;
        file >> c9;
```

```
32
33          rows.push_back(ATMRow(low, high, c0, c1, c2, c3, c4,
34              c5, c6, c7, c8, c9));
35      }
36      file.close();
37  }
38
39
40  ATMTable::ATMTable(const ATMTable& source) :
41      rows(source.rows)
42  { }
43
44  ATMTable& ATMTable::operator=(const ATMTable& source)
45  {
46      rows = source.rows;
47
48      return *this;
49  }
50
51  ATMTable::~ATMTable(void)
52  { }
53
54  std::wstring ATMTable::getMetalicityString(double metalicity)
55  {
56      //Possible values
57      // -5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5,
58      // -0.3, -0.2, -0.1, 0.0, 0.1, 0.2, 0.3, 0.5, 1.0
59      //Values are rounded to the nearest supported value.
60
61      if (metalicity < -4.75)
62          return L"M50";
63      if (metalicity >= -4.75 && metalicity < -4.25)
64          return L"M45";
65      if (metalicity >= -4.25 && metalicity < -3.75)
66          return L"M40";
67      if (metalicity >= -3.75 && metalicity < -3.25)
68          return L"M35";
69      if (metalicity >= -3.25 && metalicity < -2.75)
70          return L"M30";
71      if (metalicity >= -2.75 && metalicity < -2.25)
72          return L"M25";
73      if (metalicity >= -2.25 && metalicity < -1.75)
74          return L"M20";
75      if (metalicity >= -1.75 && metalicity < -1.25)
76          return L"M15";
77      if (metalicity >= -1.25 && metalicity < -0.75)
78          return L"M10";
79      if (metalicity >= -0.75 && metalicity < -0.4)
80          return L"M05";
81      if (metalicity >= -0.4 && metalicity < -0.25)
82          return L"M03";
83      if (metalicity >= -0.25 && metalicity < -0.15)
84          return L"M02";
85      if (metalicity >= -0.15 && metalicity < -0.05)
86          return L"M01";
87      if (metalicity >= -0.05 && metalicity < 0.05)
88          return L"P00";
```

```
89      if (metalicity >= 0.05 && metalicity < 0.15)
90          return L"P01";
91      if (metalicity >= 0.15 && metalicity < 0.25)
92          return L"P02";
93      if (metalicity >= 0.25 && metalicity < 0.4)
94          return L"P03";
95      if (metalicity >= 0.4 && metalicity < 0.75)
96          return L"P05";
97
98      //if metalicity > 0.75
99      return L"P10";
100 }
101
102 int ATMTable::getIndex(double logG)
103 {
104     //Possible values:
105     //0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0
106     //Values are rounded to the lower supported value.
107     if (logG < 0.5)
108         return 0;
109     if (logG >= 0.5 && logG < 1.0)
110         return 4;
111     if (logG >= 1.0 && logG < 1.5)
112         return 8;
113     if (logG >= 1.5 && logG < 2.0)
114         return 12;
115     if (logG >= 2.0 && logG < 2.5)
116         return 16;
117     if (logG >= 2.5 && logG < 3.0)
118         return 20;
119     if (logG >= 3.0 && logG < 3.5)
120         return 24;
121     if (logG >= 3.5 && logG < 4.0)
122         return 28;
123     if (logG >= 4.0 && logG < 4.5)
124         return 32;
125     if (logG >= 4.5 && logG < 5.0)
126         return 36;
127
128     //if logG >= 4.25
129     return 40;
130 }
131
132 double ATMTable::getLogG(int index)
133 {
134     //Possible values:
135     //0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0
136     //Values are rounded to the lower supported value.
137     if (index == 0)
138         return 0.0;
139     if (index == 4)
140         return 0.5;
141     if (index == 8)
142         return 1.0;
143     if (index == 12)
144         return 1.5;
145     if (index == 16)
```

```
146         return 2.0;
147     if (index == 20)
148         return 2.5;
149     if (index == 24)
150         return 3.0;
151     if (index == 28)
152         return 3.5;
153     if (index == 32)
154         return 4.0;
155     if (index == 36)
156         return 4.5;
157 
158     return 5.0;
159 }
```

KuruczFlux.h

```
 1 #pragma once
 2 
 3 #include <cmath>
 4 #include <string>
 5 #include <vector>
 6 #include "Calc.h"
 7 #include "Mesh3.h"
 8 #include "Binary.h"
 9 #include "Passband.h"
10 #include "SystemEffect.h"
11 #include "LegendrePolynomial.h"
12 
13 namespace Infinity {
14 class KuruczFlux : public SystemEffect
15 {
16 public:
17     KuruczFlux(Binary* binary, Mesh3* primary, Mesh3* secondary,
18         Mesh3* disc, std::vector<Passband>* passbands);
19     ~KuruczFlux();
20     void run();
21 private:
22         std::vector<Passband>* passbands;
23         LDTable* primaryLD;
24         LDTable* secondaryLD;
25 
26         std::vector<double> coef;
27         std::vector<double> ranges;
28 }; }
```

KuruczFlux.cpp

```
1 #include "stdafx.h"
2 #include "KuruczFlux.h"
3 
4 using namespace std;
5 
6 namespace Infinity {
7 KuruczFlux::KuruczFlux(Binary* binary, Mesh3* primary, Mesh3* secondary,
8     Mesh3* disc, vector<Passband>* passbands)
```

```
 9      : SystemEffect(binary, primary, secondary, disc), passbands(passbands)
10  { }
11
12  KuruczFlux::~KuruczFlux()
13  { }
14
15  void KuruczFlux::run()
16  {
17      size_t passbandCount = passbands->size();
18      size_t primaryTriangleCount = primary->getTriangleCount();
19      size_t secondaryTriangleCount = secondary->getTriangleCount();
20      size_t discTriangleCount = disc->getTriangleCount();
21
22      for (size_t p = 0; p < passbandCount; p++)
23      {
24          LDTable* primaryLD = passbands->at(p).getPrimaryLDTable();
25          LDTable* secondaryLD = passbands->at(p).getSecondaryLDTable();
26
27          ATMTable* primaryATM = passbands->at(p).getPrimaryATMTable();
28          ATMTable* secondaryATM = passbands->at(p).getSecondaryATMTable();
29
30          //Primary
31          for (size_t i = 0; i < primaryTriangleCount; ++i)
32          {
33              if (primary->getVisibility(i) > 0.0)
34              {
35                  //Vector3 unitX(1.0, 0.0, 0.0);
36                  //double cosGamma = unitX.Dot(primary->getNormal(i));
37                  double cosGamma = primary->getNormal(i)->x;
38
39                  if (cosGamma < 0.0)
40                      continue;
41
42                  double temperature = primary->getTemperature(i);
43                  double gravity = primary->getGravityAcceleration(i);
44
45                  double ldCorrection = 1 -
46                      primaryLD->getX1(temperature, log10(gravity))
47                      * (1 - sqrt(cosGamma)) -
48                      primaryLD->getX2(temperature, log10(gravity))
49                      * (1 - cosGamma) -
50                      primaryLD->getX3(temperature, log10(gravity))
51                      * (1 - sqrt(pow3(cosGamma))) -
52                      primaryLD->getX4(temperature, log10(gravity))
53                      * (1 - pow2(cosGamma));
54
55                  if (ldCorrection < 0)
56                          ldCorrection = 0;
57
58                  double flux = primaryATM->getIntensity(temperature,
59                      log10(gravity));
60                  primary->setFlux(p, i, flux * ldCorrection);
61              }
62          }
63
64          //Secondary
65          for (size_t i = 0; i < secondaryTriangleCount; ++i)
```

```
66          {
67              if (secondary->getVisibility(i) > 0.0)
68              {
69                  //Vector3 unitX(1.0, 0.0, 0.0);
70                  //double cosGamma = unitX.Dot(primary->getNormal(i));
71                  double cosGamma = secondary->getNormal(i)->x;
72
73                  if (cosGamma < 0.0)
74                      continue;
75
76                  double temperature = secondary->getTemperature(i);
77                  double gravity = secondary->getGravityAcceleration(i);
78
79                  double ldCorrection = 1 -
80                      secondaryLD->getX1(temperature, log10(gravity))
81                      * (1 - sqrt(cosGamma)) -
82                      secondaryLD->getX2(temperature, log10(gravity))
83                      * (1 - cosGamma) -
84                      secondaryLD->getX3(temperature, log10(gravity))
85                      * (1 - sqrt(pow3(cosGamma))) -
86                      secondaryLD->getX4(temperature, log10(gravity))
87                      * (1 - pow2(cosGamma));
88
89                  if (ldCorrection < 0)
90                          ldCorrection = 0;
91
92                  double flux = secondaryATM->getIntensity(temperature,
93                      log10(gravity));
94                  secondary->setFlux(p, i, flux * ldCorrection);
95              }
96          }
97
98          //Disc
99          for (size_t i = 0; i < discTriangleCount; ++i)
100         {
101             if (disc->getVisibility(i) > 0.0)
102             {
103                 //Vector3 unitX(1.0, 0.0, 0.0);
104                 //double cosGamma = unitX.Dot(primary->getNormal(i));
105                 double cosGamma = disc->getNormal(i)->x;
106
107                 if (cosGamma < 0.0)
108                     continue;
109
110                 double temperature = disc->getTemperature(i);
111                 double gravity = disc->getGravityAcceleration(i);
112
113                 double ldCorrection = 1 -
114                     primaryLD->getX1(temperature, log10(gravity))
115                     * (1 - sqrt(cosGamma)) -
116                     primaryLD->getX2(temperature, log10(gravity))
117                     * (1 - cosGamma) -
118                     primaryLD->getX3(temperature, log10(gravity))
119                     * (1 - sqrt(pow3(cosGamma))) -
120                     primaryLD->getX4(temperature, log10(gravity))
121                     * (1 - pow2(cosGamma));
122
```

```
                if (ldCorrection < 0)
                        ldCorrection = 0;

                double flux = primaryATM->getIntensity(temperature,
                    log10(gravity));
                disc->setFlux(p, i, flux * ldCorrection);
            }
        }
    }
} }
```

LDTable.h

```
#pragma once

#include <string>
#include <fstream>
#include <vector>

namespace Infinity {
class LDTable
{
public:
    LDTable(std::wstring passband, std::wstring atmModel, double metalicity,
        double microturbulence);
    ~LDTable();

    double getX1(double temperature, double logG);
    double getX2(double temperature, double logG);
    double getX3(double temperature, double logG);
    double getX4(double temperature, double logG);
private:
    size_t tempCount, logGCount;

    std::vector<double> temperatures;
    std::vector<double> logGs;

    std::vector<double> x1;
    std::vector<double> x2;
    std::vector<double> x3;
    std::vector<double> x4;

    double interpolate(std::vector<double>& data, double temperature,
        double logG);
};

inline double LDTable::getX1(double temperature, double logG)
{ return interpolate(x1, temperature, logG); }

inline double LDTable::getX2(double temperature, double logG)
{ return interpolate(x2, temperature, logG); }

inline double LDTable::getX3(double temperature, double logG)
{ return interpolate(x3, temperature, logG); }

inline double LDTable::getX4(double temperature, double logG)
```

```
44  { return interpolate(x4, temperature, logG);}
45  }
```

LDTable.cpp

```
 1  #include "stdafx.h"
 2  #include "LDTable.h"
 3
 4  using namespace std;
 5
 6  namespace Infinity {
 7  LDTable::LDTable(std::wstring passband, std::wstring atmModel,
 8                   double metallicity, double microturbulence)
 9  {
10      wstring fileName = L"ldtables/" + atmModel + L"-"
11          + (metallicity < 0 ? L"M" : L"P")
12          + (abs(metallicity) < 1.0 ? L"0" : L"")
13          + std::to_wstring(static_cast<int>(10.0 * metallicity))
14          + L"-" + std::to_wstring(static_cast<int>(microturbulence))
15          + L"." + passband;
16
17      ifstream table(fileName);
18
19      table >> logGCount;
20      table >> tempCount;
21
22      logGs.resize(logGCount);
23      for (unsigned int i = 0; i < logGCount; i++)
24          table >> logGs[i];
25
26      temperatures.resize(tempCount);
27      for (unsigned int i = 0; i < tempCount; i++)
28          table >> temperatures[i];
29
30      // Allocate memory
31      x1.resize(tempCount * logGCount);
32      x2.resize(tempCount * logGCount);
33      x3.resize(tempCount * logGCount);
34      x4.resize(tempCount * logGCount);
35
36      for (unsigned short i = 0; i < tempCount * logGCount; i++)
37      {
38          string sx1, sx2, sx3, sx4;
39
40          table >> sx1;
41          table >> sx2;
42          table >> sx3;
43          table >> sx4;
44
45          if(sx1 == "Infinity" || sx2 == "Infinity" ||
46              sx3 == "Infinity" || sx4 == "Infinity")
47          {
48              x1[i] = numeric_limits<double>::infinity();
49              x2[i] = numeric_limits<double>::infinity();
50              x3[i] = numeric_limits<double>::infinity();
51              x4[i] = numeric_limits<double>::infinity();
```

```
52          }
53          else
54          {
55              x1[i] = std::stof(sx1);
56              x2[i] = std::stof(sx2);
57              x3[i] = std::stof(sx3);
58              x4[i] = std::stof(sx4);
59          }
60      }
61  }
62
63  LDTable::~LDTable()
64  { }
65
66  double LDTable::interpolate(std::vector<double>& data, double temperature,
67              double logG)
68  {
69      size_t tempIndex = tempCount + 1;
70      size_t logGIndex = logGCount + 1;
71
72      for (size_t i = 0;  i < tempCount - 1; i++)
73          if (temperature >= temperatures[i] && temperature < temperatures[i +
74  1])
75          {
76              tempIndex = i;
77              break;
78          }
79
80      for (size_t i = 0;  i < logGCount - 1; i++)
81          if (logG >= logGs[i] && logG < logGs[i + 1])
82          {
83              logGIndex = i;
84              break;
85          }
86
87      if (tempIndex >= tempCount && temperature < temperatures[0])
88          tempIndex = 0;
89
90      if (tempIndex >= tempCount && temperature > temperatures[tempCount-1])
91          tempIndex = tempCount-1;
92
93      if (logGIndex >= logGCount && logG < logGs[0])
94          logGIndex = 0;
95
96      if (logGIndex >= logGCount && logG > logGs[logGCount-1])
97          logGIndex = logGCount-1;
98
99       double     result =
100         data[tempIndex * logGCount + logGIndex] * (temperatures[tempIndex +
101 1]
102             - temperature) * (logGs[logGIndex + 1] - logG)
103             / (temperatures[tempIndex + 1] - temperatures[tempIndex])
104             / (logGs[logGIndex + 1] - logGs[logGIndex]) +
105         data[(tempIndex + 1) * logGCount + logGIndex] * (temperature
106             - temperatures[tempIndex]) * (logGs[logGIndex + 1] - logG)
107             / (temperatures[tempIndex + 1] - temperatures[tempIndex])
108             / (logGs[logGIndex + 1] - logGs[logGIndex]) +
```

```
                data[tempIndex * logGCount + logGIndex + 1] *
(temperatures[tempIndex + 1]
            - temperature) * (logG - logGs[logGIndex])
            / (temperatures[tempIndex + 1] - temperatures[tempIndex])
            / (logGs[logGIndex + 1] - logGs[logGIndex]) +
        data[(tempIndex + 1) * logGCount + logGIndex + 1] * (temperature
            - temperatures[tempIndex]) * (logG - logGs[logGIndex])
            / (temperatures[tempIndex + 1] - temperatures[tempIndex])
            / (logGs[logGIndex + 1] - logGs[logGIndex]);

    return result;
} }
```

Passband.h

```
#pragma once

#include <string>
#include "LDTable.h"
#include "ATMTable.h"
#include "Binary.h"

namespace Infinity {
class Passband
{
public:
    Passband(std::wstring id, double effectiveWavelenght, Binary* binary);
    //Passband(const Passband& source);
    ~Passband(void);

    double getEffectiveWavelenght();
    std::wstring getID();
    LDTable* getPrimaryLDTable();
    LDTable* getSecondaryLDTable();

    ATMTable* getPrimaryATMTable();
    ATMTable* getSecondaryATMTable();

    //Static methods
    static Passband FromID(std::wstring id, Binary* binary);
private:
    double effectiveWavelenght;
    std::wstring id;

    LDTable primaryLD;
    LDTable secondaryLD;

    ATMTable primaryATM;
    ATMTable secondaryATM;
};
```

Passband.cpp

```
#include "stdafx.h"
#include "Passband.h"

namespace Infinity
{
Passband::Passband(std::wstring id, double effectiveWavelenght, Binary*
binary) :
    id(id),
    effectiveWavelenght(effectiveWavelenght),
    primaryATM(binary->getPrimary()->getMetallicity(), id),
    secondaryATM(binary->getSecondary()->getMetallicity(), id),
    primaryLD(id, binary->getPrimary()->getAtmosphereModel(),
    binary->getPrimary()->getMetallicity(),
    binary->getPrimary()->getMicroturbulence()),
    secondaryLD(id, binary->getSecondary()->getAtmosphereModel(),
    binary->getSecondary()->getMetallicity(),
    binary->getSecondary()->getMicroturbulence())
{ }

Passband::~Passband(void)
{ }

LDTable* Passband::getPrimaryLDTable()
{ return &primaryLD; }

LDTable* Passband::getSecondaryLDTable()
{ return &secondaryLD; }

ATMTable* Passband::getPrimaryATMTable()
{ return &primaryATM; }

ATMTable* Passband::getSecondaryATMTable()
{ return &secondaryATM; }

double Passband::getEffectiveWavelenght()
{ return effectiveWavelenght; }

std::wstring Passband::getID()
{ return id; }

Passband Passband::FromID(std::wstring id, Binary* binary)
{
    //Effective WL taken from Phoebe
    if (id == L"Johnson_U")
        return Passband(L"Johnson_U", 3600.0, binary);
    if (id == L"Johnson_B")
        return Passband(L"Johnson_B", 4400.0, binary);
    if (id == L"Johnson_V")
        return Passband(L"Johnson_V", 5500.0, binary);
    if (id == L"Cousins_R")
        return Passband(L"Cousins_R", 7000.0, binary);
    if (id == L"Cousins_I")
        return Passband(L"Cousins_I", 9000.0, binary);
    if (id == L"Stromgren_u")
```

```
55          return Passband(L"Stromgren_u", 3500.0, binary);
56      if (id == L"Stromgren_v")
57          return Passband(L"Stromgren_v", 4110.0, binary);
58      if (id == L"Stromgren_b")
59          return Passband(L"Stromgren_b", 4670.0, binary);
60      if (id == L"Stromgren_y")
61          return Passband(L"Stromgren_y", 5470.0, binary);
62
63      throw std::exception("Passband not supported");
64  } }
```

PlanckFlux.h

```
1   #pragma once
2
3   #include <cmath>
4   #include <string>
5   #include <vector>
6   #include "Calc.h"
7   #include "Mesh3.h"
8   #include "Binary.h"
9   #include "Passband.h"
10  #include "SystemEffect.h"
11
12  namespace Infinity {
13  class PlanckFlux : public SystemEffect
14  {
15  public:
16      PlanckFlux(Binary* binary, Mesh3* primary, Mesh3* secondary, Mesh3* disc,
17          std::vector<Passband>* passbands);
18      ~PlanckFlux();
19      void run();
20  private:
21          std::vector<Passband>* passbands;
22          LDTable* primaryLD;
23          LDTable* secondaryLD;
24  }; }
```

PlanckFlux.cpp

```
1   #include "stdafx.h"
2   #include "PlanckFlux.h"
3
4   using namespace std;
5
6   namespace Infinity {
7   PlanckFlux::PlanckFlux(Binary* binary, Mesh3* primary, Mesh3* secondary,
8       Mesh3* disc, vector<Passband>* passbands)
9       : SystemEffect(binary, primary, secondary, disc), passbands(passbands)
10  { }
11
12  PlanckFlux::~PlanckFlux()
13  { }
14
15  void PlanckFlux::run()
16  {
```

```
17      size_t passbandCount = passbands->size();
18      size_t primaryTriangleCount = primary->getTriangleCount();
19      size_t secondaryTriangleCount = secondary->getTriangleCount();
20      size_t discTriangleCount = disc->getTriangleCount();
21
22      for (size_t p = 0; p < passbandCount; p++)
23      {
24          LDTable* primaryLD = passbands->at(p).getPrimaryLDTable();
25          LDTable* secondaryLD = passbands->at(p).getSecondaryLDTable();
26
27          double effectiveWL = passbands->at(p).getEffectiveWavelenght();
28          double C1 = 3.7418498E35 / pow5(effectiveWL);
29          double C2 = 143883361 / effectiveWL;
30
31          //Primary
32          for (size_t i = 0; i < primaryTriangleCount; ++i)
33          {
34              if (primary->getVisibility(i) > 0.0)
35              {
36                  //Vector3 unitX(1.0, 0.0, 0.0);
37                  //double cosGamma = unitX.Dot(primary->getNormal(i));
38                  double cosGamma = primary->getNormal(i)->x;
39
40                  if (cosGamma < 0.0)
41                      continue;
42
43                  double temperature = primary->getTemperature(i);
44                  double gravity = primary->getGravityAcceleration(i);
45
46                  double ldCorrection = 1 -
47                      primaryLD->getX1(temperature, log10(gravity))
48                      * (1 - sqrt(cosGamma)) -
49                      primaryLD->getX2(temperature, log10(gravity))
50                      * (1 - cosGamma) -
51                      primaryLD->getX3(temperature, log10(gravity))
52                      * (1 - sqrt(pow3(cosGamma))) -
53                      primaryLD->getX4(temperature, log10(gravity))
54                      * (1 - pow2(cosGamma));
55
56                  if (ldCorrection < 0)
57                          ldCorrection = 0;
58
59                  primary->setFlux(p, i, C1 / (exp(C2 / temperature) - 1)
60                      * ldCorrection);
61              }
62          }
63
64          //Secondary
65          for (size_t i = 0; i < secondaryTriangleCount; ++i)
66          {
67              if (secondary->getVisibility(i) > 0.0)
68              {
69                  //Vector3 unitX(1.0, 0.0, 0.0);
70                  //double cosGamma = unitX.Dot(primary->getNormal(i));
71                  double cosGamma = secondary->getNormal(i)->x;
72
73                  if (cosGamma < 0.0)
```

```
                    continue;

                double temperature = secondary->getTemperature(i);
                double gravity = secondary->getGravityAcceleration(i);

                double ldCorrection = 1 -
                    secondaryLD->getX1(temperature, log10(gravity))
                    * (1 - sqrt(cosGamma)) -
                    secondaryLD->getX2(temperature, log10(gravity))
                    * (1 - cosGamma) -
                    secondaryLD->getX3(temperature, log10(gravity))
                    * (1 - sqrt(pow3(cosGamma))) -
                    secondaryLD->getX4(temperature, log10(gravity))
                    * (1 - pow2(cosGamma));

                if (ldCorrection < 0)
                    ldCorrection = 0;

                secondary->setFlux(p, i, C1 / (exp(C2 / temperature)-1)
                    * ldCorrection);
            }
        }

        //Disc
        for (size_t i = 0; i < discTriangleCount; ++i)
        {
            if (disc->getVisibility(i) > 0.0)
            {
                //Vector3 unitX(1.0, 0.0, 0.0);
                //double cosGamma = unitX.Dot(primary->getNormal(i));
                double cosGamma = disc->getNormal(i)->x;

                if (cosGamma < 0.0)
                    continue;

                double temperature = disc->getTemperature(i);
                double gravity = disc->getGravityAcceleration(i);

                double ldCorrection = 1 -
                    primaryLD->getX1(temperature, log10(gravity))
                    * (1 - sqrt(cosGamma)) -
                    primaryLD->getX2(temperature, log10(gravity))
                    * (1 - cosGamma) -
                    primaryLD->getX3(temperature, log10(gravity))
                    * (1 - sqrt(pow3(cosGamma))) -
                    primaryLD->getX4(temperature, log10(gravity))
                    * (1 - pow2(cosGamma));

                if (ldCorrection < 0)
                    ldCorrection = 0;

                disc->setFlux(p, i, C1 / (exp(C2 / temperature) - 1))
                    * ldCorrection);
            }
        }
    }
}
```

8.4. Račun orbite

CircularOrbit.h

```
#pragma once
#include "Orbit.h"

namespace Infinity {
class CircularOrbit :
    public Orbit
{
public:
    CircularOrbit(Binary* binary, std::vector<double>& phases);
    ~CircularOrbit(void);
}; }
```

CircularOrbit.cpp

```
#include "stdafx.h"
#include "CircularOrbit.h"

using namespace DirectX::SimpleMath;

namespace Infinity {

CircularOrbit::CircularOrbit(Binary* binary, std::vector<double>& phases) :
Orbit()
{
    const float pi = 3.14159265359f;
    size_t pointCount = phases.size();

    separations.reserve(pointCount);
    primaryWorld.reserve(pointCount);
    secondaryWorld.reserve(pointCount);
    diskWorld.reserve(pointCount);

    for (size_t i = 0; i < pointCount; ++i)
    {
        separations.push_back(1.0);
        primaryWorld.push_back(
            Matrix::CreateRotationZ(2 * pi * static_cast<float>(phases[i])) *
            Matrix::CreateRotationY(pi / 2.0f
                - static_cast<float>(binary->getInclination())));

        secondaryWorld.push_back(
            Matrix::CreateRotationZ(pi) *
            Matrix::CreateTranslation(1.0f, 0.0f, 0.0f) *
            Matrix::CreateRotationZ(2 * pi * static_cast<float>(phases[i])) *
            Matrix::CreateRotationY(pi / 2.0f
                - static_cast<float>(binary->getInclination())));

        diskWorld.push_back(
            Matrix::CreateRotationZ(2 * pi * static_cast<float>(phases[i])) *
            Matrix::CreateRotationY(pi / 2.0f
```

```
                - static_cast<float>(binary->getInclination())));
    }
}

CircularOrbit::~CircularOrbit(void) { }
}
```

EccentricOrbit.h

```
#pragma once
#include "Orbit.h"

namespace Infinity {
class EccentricOrbit :
    public Orbit
{
public:
    EccentricOrbit(Binary* binary, std::vector<double>& phases);
    ~EccentricOrbit(void);
}; }
```

EccentricOrbit.cpp

```
#include "stdafx.h"
#include "EccentricOrbit.h"

using namespace DirectX::SimpleMath;

namespace Infinity {
EccentricOrbit::EccentricOrbit(Binary* binary, std::vector<double>& phases)
    : Orbit()
{
    const float pi = 3.14159265359f;
    size_t pointCount = phases.size();

    separations.reserve(pointCount);
    primaryWorld.reserve(pointCount);
    secondaryWorld.reserve(pointCount);
    diskWorld.reserve(pointCount);

    double e = binary->getEccentricity();
    double omega = binary->getArgumentOfPeriastron();

    double nuZero = M_PI_2 - omega;
    double EZero = 2 * atan(sqrt((1 - e) / (1 + e)) * tan(nuZero / 2.0));
    double MZero = EZero - e * sin(EZero);

    for (size_t i = 0; i < pointCount; ++i)
    {
        double M = 2 * M_PI * phases[i] + MZero;
        double E = M;
        double corr = 0;

        do
        {
            corr = (M + e* sin(E) - E)/ (1 - e * cos(E)); //From WD.
```

```
34
35              //Halley's method. Described in Kallrath & Milone.
36              // corr = (E - e*sin(E) - M) * (1 - e*cos(E))
37              // / (pow2(1 - e*cos(E)) - 0.5*(E - e*sin(E) - M)*e*sin(E));
38              E+= corr;
39          }
40          while (abs(corr) > 1E-12);
41
42          double anomaly = 2 * atan(sqrt((1 + e) / (1 - e)) * tan(E / 2.0));
43          if (anomaly < 0) anomaly += 2.0 * M_PI;
44
45          double phaseAngle = anomaly + omega - M_PI_2;
46          double separation = 1 - e * cos(E);
47
48          //double x = separation * cos(phaseAngle);
49          //double y = separation * sin(phaseAngle);
50
51          separations.push_back(separation);
52          primaryWorld.push_back(
53              Matrix::CreateRotationZ(static_cast<float>(phaseAngle))
54              * Matrix::CreateRotationY(pi / 2.0f
55              - static_cast<float>(binary->getInclination())));
56
57          secondaryWorld.push_back(
58              Matrix::CreateRotationZ(pi)
59          * Matrix::CreateTranslation(static_cast<float>(separation), 0.0f,
60  0.0f)
61              * Matrix::CreateRotationZ(static_cast<float>(phaseAngle))
62              * Matrix::CreateRotationY(pi / 2.0f
63              - static_cast<float>(binary->getInclination())));
64
65          diskWorld.push_back(
66              Matrix::CreateRotationZ(static_cast<float>(phaseAngle))
67              * Matrix::CreateRotationY(pi / 2.0f
68              - static_cast<float>(binary->getInclination())));
69      }
70  }
71
72  EccentricOrbit::~EccentricOrbit()
73  { } }
```

Orbit.h

```
 1  #pragma once
 2
 3  #include "SimpleMath.h"
 4  #include "Binary.h"
 5
 6  namespace Infinity {
 7  class Orbit
 8  {
 9  public:
10      ~Orbit(void);
11
12      double getSeparation(size_t index) const;
13      DirectX::SimpleMath::Matrix getPrimaryWorldMatrix(size_t index) const;
```

```
14      DirectX::SimpleMath::Matrix getSecondaryWorldMatrix(size_t index) const;
15      DirectX::SimpleMath::Matrix getDiskWorldMatrix(size_t index) const;
16
17  protected:
18      Orbit();
19
20      std::vector<double> separations;
21      std::vector<DirectX::SimpleMath::Matrix> primaryWorld;
22      std::vector<DirectX::SimpleMath::Matrix> secondaryWorld;
23      std::vector<DirectX::SimpleMath::Matrix> diskWorld;
24  };
25
26
27
28  inline double Orbit::getSeparation(size_t index)
29      const { return separations[index]; }
30  inline DirectX::SimpleMath::Matrix Orbit::getPrimaryWorldMatrix(size_t index)
31      const { return primaryWorld[index]; }
32  inline DirectX::SimpleMath::Matrix Orbit::getSecondaryWorldMatrix(size_t
33  index)
34      const { return secondaryWorld[index]; }
35  inline DirectX::SimpleMath::Matrix Orbit::getDiskWorldMatrix(size_t index)
36      const { return diskWorld[index]; } }
```

Orbit.cpp

```
1   #include "stdafx.h"
2   #include "Orbit.h"
3
4   namespace Infinity {
5   Orbit::Orbit()
6   { }
7
8   Orbit::~Orbit(void)
9   { } }
```

8.5. Model zvezde

GravityAcceleration.h

```
1   #pragma once
2
3   #include <cmath>
4   #include "Calc.h"
5   #include "StellarEffect.h"
6   #include "Mesh3.h"
7   #include "Binary.h"
8
9
10  namespace Infinity {
11  class GravityAcceleration : public StellarEffect
12  {
13  public:
14      GravityAcceleration(Binary* binary, Star* star,
15          Mesh3* mesh, double massRatio);
```

```
16      ~GravityAcceleration();
17
18      void run(double time = 0);
19
20  private:
21      double rocheDelX(double x, double y, double z);
22      double rocheDelY(double x, double y, double z);
23      double rocheDelZ(double x, double y, double z);
24
25      double massRatio;
26      double async;
27  };
28
29  inline double GravityAcceleration::rocheDelX(double x, double y, double z)
30  {
31      return - x / ((x*x + y*y + z*z) * sqrt(x*x + y*y + z*z))
32              + massRatio * (1 - x) / (((1 - x)*(1 - x) + y*y + z*z)
33              * sqrt((1 - x)*(1 - x) + y*y + z*z))
34              - massRatio + (1 + massRatio) * x * pow2(async);
35  }
36
37  inline double GravityAcceleration::rocheDelY(double x, double y, double z)
38  {
39      return - y / ((x*x + y*y + z*z) * sqrt(x*x + y*y + z*z))
40              - massRatio * y / (((1 - x)*(1 - x) + y*y + z*z)
41              * sqrt((1 - x)*(1 - x) + y*y + z*z))
42              + (1 + massRatio) * y * pow2(async);
43  }
44
45  inline double GravityAcceleration::rocheDelZ(double x, double y, double z)
46  {
47      return - z / ((x*x + y*y + z*z) * sqrt(x*x + y*y + z*z))
48              - massRatio * z / (((1 - x)*(1 - x) + y*y + z*z)
49              * sqrt((1 - x)*(1 - x) + y*y + z*z));
50  } }
```

GravityAcceleration.cpp

```
1   #include "stdafx.h"
2   #include "GravityAcceleration.h"
3
4   using namespace std;
5   using namespace DirectX::SimpleMath;
6
7   namespace Infinity {
8   GravityAcceleration::GravityAcceleration(Binary* binary,
9       Star* star, Mesh3* mesh, double q)
10      : StellarEffect(binary, star, mesh), massRatio(q)
11  {
12      async = star->getAsynchronicity();
13  }
14
15  GravityAcceleration::~GravityAcceleration(void)
16  { }
17
18  void GravityAcceleration::run(double time)
```

```
{
    double gravityConstant = 6.67428E-8;
    double solarRadius = 6.955E10;

    double effectiveGravity = 0.0;

    //Roche star
    if (star->getShape() == Star::Shape::Roche)
    {
        double scalingFactor = gravityConstant * star->getMass()
            / pow2(binary->getSemimajorAxis() * solarRadius);

        size_t triangleCount = mesh->getTriangleCount();
        for (size_t i = 0; i < triangleCount; ++i)
        {
            double x = mesh->getCentroid(i)->x;
            double y = mesh->getCentroid(i)->y;
            double z = mesh->getCentroid(i)->z;

            double g = scalingFactor * sqrt(pow2(rocheDelX(x, y, z))
                + pow2(rocheDelY(x, y, z)) + pow2(rocheDelZ(x, y, z)));

            mesh->setGravityAcceleration(i, g);
            effectiveGravity += g * mesh->getArea(i);
        }

        effectiveGravity /= mesh->getMeshArea();
    }

    //Spherical star
    if (star->getShape() == Star::Shape::Sphere)
    {
        effectiveGravity = gravityConstant * star->getMass()
            / pow2(star->getPolarRadius() * binary->getSemimajorAxis()
            * solarRadius);

        size_t triangleCount = mesh->getTriangleCount();
        for (size_t i = 0; i < triangleCount; ++i)
            mesh->setGravityAcceleration(i, effectiveGravity);
    }

    star->setEffectiveGravity(effectiveGravity);
} }
```

RocheSurface.h

```
#pragma once

#include "UnivariateFunction.h"
#include "RootFinder.h"
#include "BisectionRootFinder.h"
#include "BrentRootFinder.h"
#include "Surface.h"

namespace Infinity {
class RocheSurface : public Surface
```

```
{
public:
    RocheSurface(double massRatio, double separation, double polarRadius,
        double asynchronicity, Star* star);
    ~RocheSurface(void);

    double calculateRadius(RootFinder* rootFinder, double theta, double phi,
        double from, double to, double potential = 0.0);
    double getCriticalPolarRadius();
    double getSychronousCriticalPolarRadius();
    double getFrontRadius();
    double getCriticalFrontRadius();
    double getSynchronousCriticalFrontRadius();
    double getBackRadius();
    double getSideRadius();
    double getMeanEggletonRadius();
    double getFillingFactor();
    double getPotential();
    double getCriticalPotential();
    double getSynchronousCriticalPotential();

    float getOuterRadius() const;
    float getInnerRadius() const;
    float getRadius(float theta, float phi, double time = 0);

private:
    class RocheFunction : public Function
    {
    public:
        RocheFunction(double q, double d, double f, double t = 0, double p =
0)
            : massRatio(q), separation(d), async(f), theta(t), phi(p)
        { }

        ~RocheFunction() { }

        double theta, phi;

        double operator() (double r) const
        {
            double lambda = sin(theta) * cos(phi);
            return 1 / r + massRatio * (1 / sqrt(pow2(separation)
                + pow2(r) - 2 * r * separation * lambda)
                - r * lambda/pow2(separation)) + (massRatio + 1)
                / 2 * pow2(r * sin(theta) * async);
        }

        double operator() (double r, double t, double p)
        {
            theta = t;
            phi = p;
            return (*this)(r);
        }

    private:
        double massRatio, async, separation;
    };
```

```
68
69      class RocheDerivative : public Function
70      {
71      public:
72          RocheDerivative(double q, double d, double f, double t = 0, double p
73  = 0)
74              : massRatio(q), separation(d), async(f), theta(t), phi(p)
75          { }
76
77          ~RocheDerivative() { }
78
79          double theta, phi;
80
81          double operator() (double r) const
82          {
83              double lambda = sin(theta) * cos(phi);
84              double A = sqrt(pow3(pow2(separation) + pow2(r)
85                  - 2 * r * separation * lambda));
86              return -1.0 / pow2(r) + massRatio * (lambda*separation/A
87                  - lambda / pow2(separation) - r / A) + (massRatio + 1)
88                  * r * pow2(sin(theta) * async);
89          }
90
91          double operator() (double r, double t, double p)
92          {
93              theta = t;
94              phi = p;
95              return (*this)(r);
96          }
97
98      private:
99          double massRatio, async, separation;
100
101     };
102
103     double massRatio, polarRadius, asynchronicity, separation;
104
105     double criticalPolarRadius, synchronousCriticalPolarRadius;
106     double frontRadius, criticalFrontRadius, synchronousCriticalFrontRadius;
107     double backRadius, sideRadius;
108     double meanEggletonRadius;
109     double fillingFactor, potential, criticalPotential,
110         synchronousCriticalPotential;
111      RocheFunction function;
112     RocheDerivative derivative;
113     RootFinder *safeRF, *fastRF;
114 };
115
116 inline double RocheSurface::calculateRadius(RootFinder* rootFinder,
117     double theta, double phi, double from, double to, double potential)
118 {
119     function.theta = theta;
120     function.phi = phi;
121
122     return rootFinder->find(function, from, to, potential);
123 }
124
```

```
125 inline double RocheSurface::getCriticalPolarRadius()
126 { return criticalPolarRadius; }
127 inline double RocheSurface::getSychronousCriticalPolarRadius()
128 { return synchronousCriticalPolarRadius; }
129 inline double RocheSurface::getFrontRadius()
130 { return frontRadius; }
131 inline double RocheSurface::getCriticalFrontRadius()
132 { return criticalFrontRadius; }
133 inline double RocheSurface::getSynchronousCriticalFrontRadius()
134 { return synchronousCriticalFrontRadius; }
135 inline double RocheSurface::getBackRadius()
136 { return backRadius; }
137 inline double RocheSurface::getSideRadius()
138 { return sideRadius; }
139 inline double RocheSurface::getMeanEggletonRadius()
140 { return meanEggletonRadius; }
141 inline double RocheSurface::getFillingFactor()
142 { return fillingFactor; }
143 inline double RocheSurface::getPotential()
144 { return potential; }
145 inline double RocheSurface::getCriticalPotential()
146 { return criticalPotential; }
147 inline double RocheSurface::getSynchronousCriticalPotential()
148 { return synchronousCriticalPotential; }
149 inline float RocheSurface::getOuterRadius() const
150 { return static_cast<float>(frontRadius); }
151 inline float RocheSurface::getInnerRadius() const
152 { return static_cast<float>(polarRadius); }
153
154 inline float RocheSurface::getRadius(float theta, float phi, double time)
155 {
156     double perturbation = 0.0;
157
158     if (star->getPerturbSurface())
159     {
160         size_t modeCount = star->getModeCount();
161         for (size_t i = 0; i < modeCount; ++i)
162             perturbation +=
163             star->getMode(i)->GetDisplacementPerturbationAt(theta, phi,
164 time);
165     }
166
167     return static_cast<float>(calculateRadius(fastRF, theta, phi,
168         0.999 * polarRadius, 1.001 * frontRadius, potential) + perturbation);
169 } }
```

RocheSurface.cpp

```
1  #include "stdafx.h"
2  #include "RocheSurface.h"
3
4  namespace Infinity {
5  RocheSurface::RocheSurface(double massRatio, double separation, double
6  polarRadius, double asynchronicity, Star* star) :
7      Surface(star),
8      massRatio(massRatio),
```

```
 9       separation(separation),
10       polarRadius(polarRadius),
11       asynchronicity(asynchronicity),
12       function(massRatio, separation, asynchronicity),
13       derivative(massRatio, separation, asynchronicity)
14   {
15       double epsilon = 1E-10;
16       unsigned short maxEval = 100;
17
18       safeRF = new BisectionRootFinder(epsilon, maxEval);
19       fastRF = new BrentRootFinder(epsilon, maxEval);
20
21       //Critical potential
22       derivative.theta = M_PI_2;
23       derivative.phi = 0.0;
24       criticalFrontRadius = safeRF->find(derivative, 0.01, 0.99 * separation);
25
26       criticalPotential = function(criticalFrontRadius, M_PI_2, 0.0);
27       criticalPolarRadius = calculateRadius(safeRF, 0.0, 0.0, 0.01, 1.01
28           * criticalFrontRadius, criticalPotential);
29
30       if (asynchronicity == 1.0)
31       {
32           synchronousCriticalFrontRadius = criticalFrontRadius;
33           synchronousCriticalPolarRadius = criticalPolarRadius;
34           synchronousCriticalPotential = criticalPotential;
35       }
36       else
37       {
38           RocheFunction syncFunction(massRatio, separation, 1.0);
39           RocheDerivative syncDerivative(massRatio, separation, 1.0);
40           syncDerivative.theta = M_PI_2;
41           syncDerivative.phi = 0.0;
42
43           synchronousCriticalFrontRadius = safeRF->find(syncDerivative, 0.01,
44               0.99 * separation);
45
46           synchronousCriticalPotential
47               = syncFunction(synchronousCriticalFrontRadius, M_PI_2, 0.0);
48           synchronousCriticalPolarRadius
49               = calculateRadius(safeRF, 0.0, 0.0, 0.01, 1.01
50               * synchronousCriticalFrontRadius, synchronousCriticalPotential);
51       }
52
53       //Filling factor and potential...
54       fillingFactor = polarRadius / criticalPolarRadius;
55       potential = function(polarRadius, 0.0, 0.0);
56
57       if (fillingFactor > 1.0)
58           throw std::exception("Overcontact mode not supported.");
59
60       //...and rest of the radii
61       frontRadius = calculateRadius(safeRF, M_PI_2, 0.0, 0.99
62           * polarRadius, 1.01 * criticalFrontRadius, potential);
63       sideRadius = calculateRadius(safeRF, M_PI_2, M_PI_2, 0.99
64           * polarRadius, 1.01 * frontRadius, potential);
65       backRadius = calculateRadius(safeRF, M_PI_2, M_PI, 0.99
```

```
66          * polarRadius, 1.01 * frontRadius, potential);
67
68      //Mean Eggleton radius
69      meanEggletonRadius = 0.49 * pow(massRatio, 2.0 / 3.0) /
70          (0.6 *  pow(massRatio, 2.0 / 3.0) + log(1 + pow(massRatio, 1.0 /
71  3.0)));
72  }
73
74  RocheSurface::~RocheSurface(void)
75  { } }
```

SphericalSurface.h

```
1   #pragma once
2
3   #include "Surface.h"
4
5   namespace Infinity {
6   class SphericalSurface : public Surface
7   {
8   public:
9       SphericalSurface(Star* star);
10      ~SphericalSurface();
11
12      float getOuterRadius() const;
13      float getInnerRadius() const;
14      float getRadius(float theta, float phi, double time = 0);
15  private:
16      float radius;
17  };
18
19  inline float SphericalSurface::getOuterRadius() const
20  { return radius; }
21
22  inline float SphericalSurface::getInnerRadius() const
23  { return radius; }
24
25  inline float SphericalSurface::getRadius(float theta, float phi, double time)
26  {
27      double perturbation = 0.0;
28
29      if (star->getPerturbSurface())
30      {
31          size_t modeCount = star->getModeCount();
32          for (size_t i = 0; i < modeCount; ++i)
33              perturbation += star->getMode(i)->
34              GetDisplacementPerturbationAt(theta, phi, time);
35      }
36
37      return radius + static_cast<float>(perturbation);
38  } }
```

SphericalSurface.cpp

```
1   #include "stdafx.h"
2   #include "SphericalSurface.h"
```

```
3
4   namespace Infinity {
5   SphericalSurface::SphericalSurface(Star* star) :
6       Surface(star),
7       radius(static_cast<float>(star->getPolarRadius()))
8   { }
9
10  SphericalSurface::~SphericalSurface()
11  { } }
```

Spot.h

```
1   #pragma once
2
3   namespace Infinity {
4   class Spot
5   {
6   public:
7       Spot(double theta, double phi, double size, double temperautreRatio);
8       ~Spot();
9
10      double getTheta();
11      double getPhi();
12      double getSize();
13      double getTemperatureRatio();
14
15  private:
16      double theta, phi, size, temperatureRatio;
17  };
18
19  inline double Spot::getTheta() { return theta; }
20  inline double Spot::getPhi() { return phi; }
21  inline double Spot::getSize() { return size; }
22  inline double Spot::getTemperatureRatio() { return temperatureRatio; } }
```

Spot.cpp

```
1   #include "stdafx.h"
2   #include "Spot.h"
3
4   namespace Infinity {
5   Spot::Spot(double theta, double phi, double size, double temperatureRatio):
6       theta(theta),
7       phi(phi),
8       size(size),
9       temperatureRatio(temperatureRatio)
10  { }
11
12  Spot::~Spot() { } }
```

Spots.h

```
1   #pragma once
2
3   #include "StellarEffect.h"
```

```
#include <cmath>
#include "Calc.h"
#include "Mesh3.h"
#include "Binary.h"
#include "Spot.h"
#include <vector>

namespace Infinity {
class Spots : public StellarEffect
{
public:
    Spots(Binary* binary, Star* star, Mesh3* mesh);
    ~Spots();

    void run(double time = 0);
}; }
```

Spots.cpp

```
#include "stdafx.h"
#include "Spots.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {
Spots::Spots(Binary* binary, Star* star, Mesh3* mesh)
    : StellarEffect(binary, star, mesh)
{ }

Spots::~Spots(void)
{ }

void Spots::run(double time)
{
    size_t spotCount = star->getSpotCount();
    size_t triangleCount =  mesh->getTriangleCount();

    for(size_t j = 0; j < spotCount; j++)
    {
        double spotTheta = star->getSpot(j)->getTheta();
        double spotPhi = star->getSpot(j)->getPhi();
        double spotSize = star->getSpot(j)->getSize();
        double temperatureRatio = star->getSpot(j)->getTemperatureRatio();

        for (size_t i = 0; i < triangleCount; i++)
        {
            double x = mesh->getCentroid(i)->x;
            double y = mesh->getCentroid(i)->y;
            double z = mesh->getCentroid(i)->z;

            double phi = atan2(y, x);
            phi = (phi >= 0) ? phi : phi + 2 * M_PI;
            double theta = acos(z / sqrt(x*x + y*y + z*z));

            double cosL = cos(theta) * cos(spotTheta) + sin(theta)
```

```
                * sin(spotTheta) * cos(phi - spotPhi);

            if (cos(spotSize) < cosL)
                mesh->setTemperature(i, mesh->getTemperature(i)
                * temperatureRatio);
        }
    }
} }
```

Star.h

```
#pragma once

#include "Mode.h"
#include "Spot.h"
#include "Palette.h"
#include <vector>

namespace Infinity {
class Star
{
public:
    enum Shape { Roche, Sphere };

    Star();
    Star& operator= (const Star &source);
    Star(const Star& source);
    ~Star();

    double getAsynchronicity() const;
    double getPolarRadius() const;
    double getTemperature() const;
    double getGravityDarkeningExponent() const;
    double getAlbedo() const;
    size_t getSubdivisionLevel() const;
    std::wstring getAtmosphereModel() const;
    double getMetallicity() const;
    double getMicroturbulence() const;
    Shape getShape() const;
    Palette* getPalette();
    bool getPerturbSurface() const;

    //Calculated stuff
    double getMass() const;
    double getEffectiveGravity() const;

    void setMass(double value);
    void setEffectiveGravity(double value);

    size_t getSpotCount();
    Spot* getSpot(size_t index);
    void addSpot(double theta, double phi, double size, double
temperatureRatio);

    static Star CreateRoche(double asynchronicity, double polarRadius,
        double temperature, double gdExponent, double albedo,
```

```
46          size_t subdivisionLevel,         std::wstring atmosphereModel,
47          double metallicity, double microturbulence, bool perturbSurface,
48          std::wstring palette);
49      static Star CreateSphere(double asynchronicity, double polarRadius,
50          double temperature, double albedo, size_t subdivisionLevel,
51          std::wstring atmosphereModel, double metallicity,
52          double microturbulence, bool perturbSurface, std::wstring palette);
53
54  private:
55      Star(Shape shape, double asynchronicity, double polarRadius,
56          double temperature, double gdExponent, double albedo,
57          size_t subdivisionLevel,
58          std::wstring atmosphereModel, double metallicity, double
59  microturbulence,
60          bool perturbSurface, std::wstring palette);
61
62      std::vector<Mode> modes;
63      std::vector<Spot> spots;
64
65      Shape shape;
66
67      double asynchronicity, polarRadius, temperature, gdExponent,
68          albedo, metallicity, microturbulence;
69      size_t subdivisionLevel;
70      std::wstring atmosphereModel;
71      double mass, effectiveGravity, potential;
72
73      bool massCalculated, effectiveGravityCalculated;
74      bool perturbSurface;
75
76      Palette palette;
77  };
78
79  inline double Star::getAsynchronicity() const { return asynchronicity; }
80  inline double Star::getPolarRadius() const { return polarRadius; }
81  inline double Star::getTemperature() const { return temperature; }
82  inline double Star::getGravityDarkeningExponent() const { return gdExponent;
83  }
84  inline double Star::getAlbedo() const { return albedo; }
85  inline size_t Star::getSubdivisionLevel() const { return subdivisionLevel; }
86  inline std::wstring Star::getAtmosphereModel() const { return
87  atmosphereModel; }
88  inline double Star::getMetallicity() const { return metallicity; }
89  inline double Star::getMicroturbulence() const { return microturbulence; }
90  inline Star::Shape Star::getShape() const { return shape; }
91  inline Palette* Star::getPalette() { return &palette; }
92  inline bool Star::getPerturbSurface() const { return perturbSurface; }
93
94  inline double Star::getMass() const
95  {
96      if(massCalculated) return mass;
97      throw std::exception("Mass not determined yet");
98  }
99
100 inline void Star::setMass(double value)
101 {
102     mass = value;
```

```
103         massCalculated = true;
104     }
105
106     inline double Star::getEffectiveGravity() const
107     {
108         if(effectiveGravityCalculated) return effectiveGravity;
109         throw std::exception("Gravity not determined yet");
110     }
111
112     inline void Star::setEffectiveGravity(double value)
113     {
114         effectiveGravity = value;
115         effectiveGravityCalculated = true;
116     } }
```

Star.cpp

```
 1   #include "stdafx.h"
 2   #include "Star.h"
 3
 4   namespace Infinity
 5   {
 6   Star::Star() :
 7       palette(L"red")
 8   { }
 9
10   Star::Star(Shape shape, double asynchronicity, double polarRadius,
11              double temperature, double gdExponent, double albedo,
12              size_t subdivisionLevel,
13              std::wstring atmosphereModel, double metallicity,
14              double microturbulence, bool perturbSurface,
15              std::wstring paletteFile) :
16       shape(shape),
17       asynchronicity(asynchronicity),
18       polarRadius(polarRadius),
19       temperature(temperature),
20       gdExponent(gdExponent),
21       albedo(albedo),
22       subdivisionLevel(subdivisionLevel),
23       atmosphereModel(atmosphereModel),
24       metallicity(metallicity),
25       microturbulence(microturbulence),
26       perturbSurface(perturbSurface),
27       palette(paletteFile)
28   {
29       massCalculated = false;
30       effectiveGravityCalculated = false;
31   }
32
33   Star::Star(const Star& source) :
34       palette(source.palette)
35   {
36       shape = source.shape;
37       asynchronicity = source.asynchronicity;
38       polarRadius = source.polarRadius;
39       temperature = source.temperature;
```

```
40      gdExponent = source.gdExponent;
41      albedo = source.albedo;
42      subdivisionLevel = source.subdivisionLevel;
43      atmosphereModel = source.atmosphereModel;
44      metallicity = source.metallicity;
45      microturbulence = source.microturbulence;
46
47      //Do we really use this?
48      potential = source.potential;
49
50      massCalculated = source.massCalculated;
51      if (massCalculated)
52          mass = source.mass;
53
54      effectiveGravityCalculated = source.effectiveGravityCalculated;
55      if (effectiveGravityCalculated)
56          effectiveGravity = source.effectiveGravity;
57
58      spots = source.spots;
59
60      perturbSurface = source.perturbSurface;
61  }
62
63  Star& Star::operator= (const Star &source)
64  {
65      shape = source.shape;
66      asynchronicity = source.asynchronicity;
67      polarRadius = source.polarRadius;
68      temperature = source.temperature;
69      gdExponent = source.gdExponent;
70      albedo = source.albedo;
71      subdivisionLevel = source.subdivisionLevel;
72      atmosphereModel = source.atmosphereModel;
73      metallicity = source.metallicity;
74      microturbulence = source.microturbulence;
75      palette = source.palette;
76
77      //Do we really use this?
78      potential = source.potential;
79
80      massCalculated = source.massCalculated;
81      if (massCalculated)
82          mass = source.mass;
83
84      effectiveGravityCalculated = source.effectiveGravityCalculated;
85      if (effectiveGravityCalculated)
86          effectiveGravity = source.effectiveGravity;
87
88      spots = source.spots;
89
90      perturbSurface = source.perturbSurface;
91
92      return *this;
93  }
94
95  Star::~Star() { }
96
```

```
Star Star::CreateRoche(double asynchronicity, double polarRadius,
    double temperature, double gdExponent, double albedo,
    size_t subdivisionLevel, std::wstring atmosphereModel, double
metallicity,
    double microturbulence, bool perturbSurface, std::wstring palette)
{
    return Star(Shape::Roche, asynchronicity, polarRadius, temperature,
        gdExponent, albedo, subdivisionLevel, atmosphereModel, metallicity,
        microturbulence, perturbSurface, palette);
}

Star Star::CreateSphere(double asynchronicity, double polarRadius,
    double temperature, double albedo, size_t subdivisionLevel,
    std::wstring atmosphereModel, double metallicity, double microturbulence,
    bool perturbSurface, std::wstring palette)
{
    return Star(Shape::Sphere, asynchronicity, polarRadius, temperature, 0.0,
        albedo, subdivisionLevel, atmosphereModel, metallicity,
        microturbulence, perturbSurface, palette);
}

size_t Star::getSpotCount()
{ return spots.size(); }

Spot* Star::getSpot(size_t index)
{ return &spots[index]; }

void Star::addSpot(double theta, double phi, double size,
    double temperatureRatio)
{ spots.push_back(Spot(theta, phi, size, temperatureRatio)); } }
```

StellarEffect.h

```
#pragma once

#include <cmath>
#include "Calc.h"
#include "Mesh3.h"
#include "Binary.h"

namespace Infinity {
class StellarEffect
{
public:
    StellarEffect(Binary* binary, Star* star, Mesh3* mesh);
    ~StellarEffect(void);

    virtual void run(double time = 0) = 0;
protected:
    Binary* binary;
    Mesh3* mesh;
    Star* star;
}; }
```

StellarEffect.cpp

```
#include "stdafx.h"
#include "StellarEffect.h"

namespace Infinity {
StellarEffect::StellarEffect(Binary* binary, Star* star, Mesh3* mesh)
    : binary(binary), star(star), mesh(mesh)
{ }

StellarEffect::~StellarEffect(void)
{ } }
```

Surface.h

```
#pragma once

#include "Star.h"

namespace Infinity {
class Surface
{
public:
    Surface(Star* star);
    Surface(const Surface& source);

    ~Surface();

    Surface& operator=(const Surface& source);

    virtual float getInnerRadius() const = 0;
    virtual float getOuterRadius() const = 0;
    virtual float getRadius(float theta, float phi, double time = 0) = 0;
protected:
    Star* star;
}; }
```

Surface.cpp

```
#include "stdafx.h"
#include "Surface.h"

namespace Infinity {
Surface::Surface(Star* star) :
    star(star)
{ }

Surface::Surface(const Surface& source) :
    star(source.star)
{ }

Surface::~Surface()
{ }

Surface& Surface::operator=(const Surface& source)
```

```
{
    star = source.star;

    return *this;
} }
```

VonZeipelGravityDarkening.h

```
#pragma once

#include "StellarEffect.h"

namespace Infinity {
class VonZeipelGravityDarkening : public StellarEffect
{
public:
    VonZeipelGravityDarkening(Binary* binary, Star* star, Mesh3* mesh);
    ~VonZeipelGravityDarkening();

    void run(double time = 0);
}; }
```

VonZeipelGravityDarkening.cpp

```
#include "stdafx.h"
#include "VonZeipelGravityDarkening.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {
VonZeipelGravityDarkening::VonZeipelGravityDarkening(Binary* binary,
    Star* star, Mesh3* mesh) : StellarEffect(binary, star, mesh)
{ }

VonZeipelGravityDarkening::~VonZeipelGravityDarkening(void)
{ }

void VonZeipelGravityDarkening::run(double time)
{
    if (star->getShape() == Star::Shape::Roche)
    {
        double temperature = star->getTemperature();
        double effectiveGravity = star->getEffectiveGravity();
        double gdExponent = star->getGravityDarkeningExponent();

        size_t triangleCount =  mesh->getTriangleCount();
        for (size_t i = 0; i < triangleCount; i++)
            mesh->setTemperature(i, temperature
            * pow(mesh->getGravityAcceleration(i) / effectiveGravity,
            gdExponent));
    }
} }
```

Velocity.h

```
#pragma once

#include <cmath>
#include "Calc.h"
#include "StellarEffect.h"
#include "Mesh3.h"
#include "Binary.h"
#include "Orbit.h"

namespace Infinity {
class Velocity : public StellarEffect
{
public:
    Velocity(Binary* binary, Star* star, Mesh3* mesh);
    ~Velocity();

    void run(double time = 0);
}; }
```

Velocity.cpp

```
#include "stdafx.h"
#include "Velocity.h"

using namespace std;
using namespace DirectX::SimpleMath;

namespace Infinity {
Velocity::Velocity(Binary* binary, Star* star, Mesh3* mesh)
    : StellarEffect(binary, star, mesh)
{ }

Velocity::~Velocity(void)
{ }

void Velocity::run(double time)
{
    //Primary
    size_t triangleCount = mesh->getTriangleCount();
    for (size_t i = 0; i < triangleCount; i++)
    {
        //v = omega x r; omega(0, 0, w); r(x, y, z);
        //if we expand the cross product
        //v.x = -y * w;
        //v.y =  x * w;

        float w = static_cast<float>(star->getAsynchronicity()) * TWO_PI_F
            / static_cast<float>(binary->getPeriod() *  86400.0); //in 1/s

        float x = mesh->getCentroid(i)->x * static_cast<float>
            (binary->getSemimajorAxis() * 6.955E5);  //in km;
        float y = mesh->getCentroid(i)->y * static_cast<float>
            (binary->getSemimajorAxis() * 6.955E5);  //in km;

```

```
34          mesh->getVelocity(i)->x = y * w;
35          mesh->getVelocity(i)->y = -x * w;
36      }
37  } }
```

8.6. Model dvojnog sistema

Binary.h

```
 1  #pragma once
 2  #include "Calc.h"
 3  #include "Star.h"
 4  #include "Disc.h"
 5  #include <string>
 6  #include <exception>
 7
 8  namespace Infinity {
 9  class Binary
10  {
11  public:
12      Binary();
13      Binary(double massRatio, double period, double semimajorAxis,
14          double inclination, double eccentricity, double argumentOfPeriastron,
15          double epoch, double systemVelocity);
16      Binary(const Binary& source);
17      ~Binary();
18
19      double getMassRatio();
20      double getPeriod();
21      double getSemimajorAxis();
22      double getInclination();
23      double getEpoch();
24      double getTotalMass();
25      double getEccentricity();
26      double getArgumentOfPeriastron();
27      double getSystemVelocity();
28
29      bool hasDisc();
30
31      Star* getPrimary();
32      Star* getSecondary();
33      Disc* getDisc();
34
35      void createPrimary(Star::Shape shape, double asynchronicity,
36          double polarRadius, double temperature, double gdExponent,
37          double albedo, size_t subdivisionLevel,
38          std::wstring atmosphereModel, double metallicity,
39          double microturbulence, bool perturbSurface, std::wstring palette);
40      void createSecondary(Star::Shape shape, double asynchronicity,
41          double polarRadius, double temperature, double gdExponent,
42          double albedo, size_t subdivisionLevel,
43          std::wstring atmosphereModel, double metallicity,
44          double microturbulence, bool perturbSurface, std::wstring palette);
45      void createDisc(Disc::Shape shape, CrossSection& cs, int segments,
46          double innerRadius, double outerRadius, double innerTemperature,
```

```
47          double outerTemperature, double exponent, int distributionType,
48          std::wstring paletteFile);
49
50  private:
51      double massRatio, period, semimajorAxis, inclination, epoch,
52          eccentricity, argumentOfPeriastron, systemVelocity;
53      double totalMass;
54
55      bool hasDiskFlag;
56
57      Star primary;
58      Star secondary;
59      Disc disc;
60  };
61
62  inline double Binary::getMassRatio() { return massRatio; }
63  inline double Binary::getPeriod() { return period; }
64  inline double Binary::getSemimajorAxis() { return semimajorAxis; }
65  inline double Binary::getInclination() { return inclination; }
66  inline double Binary::getEpoch() { return epoch; }
67  inline double Binary::getTotalMass() { return totalMass; }
68  inline double Binary::getEccentricity() { return eccentricity; }
69  inline double Binary::getArgumentOfPeriastron()
70  { return argumentOfPeriastron; }
71  inline double Binary::getSystemVelocity() { return systemVelocity; }
72  inline bool Binary::hasDisc() {return hasDiskFlag; }
73
74  inline Star* Binary::getPrimary() { return &primary; }
75
76  inline void Binary::createPrimary(Star::Shape shape, double asynchronicity,
77      double polarRadius, double temperature, double gdExponent, double albedo,
78      size_t subdivisionLevel, std::wstring atmosphereModel,
79      double metallicity, double microturbulence, bool perturbSurface,
80      std::wstring palette)
81  {
82      switch (shape)
83      {
84      case Star::Shape::Roche:
85      primary = Star::CreateRoche(asynchronicity, polarRadius, temperature,
86          gdExponent, albedo, subdivisionLevel, atmosphereModel, metallicity,
87          microturbulence, perturbSurface, palette);
88          break;
89      case Star::Shape::Sphere:
90      primary = Star::CreateSphere(asynchronicity, polarRadius, temperature,
91          albedo, subdivisionLevel, atmosphereModel, metallicity,
92          microturbulence, perturbSurface, palette);
93          break;
94      default:
95          break;
96      }
97      primary.setMass(totalMass / (1 + massRatio));
98  }
99
100 inline Star* Binary::getSecondary() { return &secondary; }
101
102 inline void Binary::createSecondary(Star::Shape shape, double asynchronicity,
103     double polarRadius, double temperature, double gdExponent, double albedo,
```

```
104         size_t subdivisionLevel, std::wstring atmosphereModel, double
105     metallicity,
106         double microturbulence, bool perturbSurface, std::wstring palette)
107     {
108         switch (shape)
109         {
110         case Star::Shape::Roche:
111         secondary = Star::CreateRoche(asynchronicity, polarRadius, temperature,
112             gdExponent, albedo, subdivisionLevel, atmosphereModel, metallicity,
113             microturbulence, perturbSurface, palette);
114             break;
115         case Star::Shape::Sphere:
116         secondary = Star::CreateSphere(asynchronicity, polarRadius, temperature,
117             albedo, subdivisionLevel, atmosphereModel, metallicity,
118             microturbulence, perturbSurface, palette);
119             break;
120         default:
121             break;
122         }
123
124         secondary.setMass(totalMass * massRatio / (1 + massRatio));
125     }
126
127     inline Disc* Binary::getDisc() { return &disc; }
128
129     inline void Binary::createDisc(Disc::Shape shape, CrossSection& cs,
130         int segments, double innerRadius, double outerRadius,
131         double innerTemperature, double outerTemperature, double exponent,
132         int distributionType, std::wstring paletteFile)
133     {
134         disc = Disc(shape, cs, segments, innerRadius, outerRadius,
135             innerTemperature, outerTemperature, exponent, distributionType,
136             paletteFile);
137         hasDiskFlag = true;
138     } }
```

Binary.cpp

```
 1   #include "stdafx.h"
 2   #include "Binary.h"
 3
 4   namespace Infinity {
 5   Binary::Binary() { }
 6
 7   Binary::Binary(double massRatio, double period, double semimajorAxis,
 8       double inclination, double eccentricity,
 9       double argumentOfPeriastron, double epoch, double systemVelocity) :
10       massRatio(massRatio),
11       period(period),
12       semimajorAxis(semimajorAxis),
13       inclination(inclination),
14       eccentricity(eccentricity),
15       argumentOfPeriastron(argumentOfPeriastron),
16       epoch(epoch),
17       systemVelocity(systemVelocity),
18       hasDiskFlag(false)
```

```
{
    double solarMass = 1.98892E33;
    double solarRadius = 6.955E10;
    double astronomicalUnit = 14959787070000;
    double sideralYear = 365.256363051;

    totalMass = solarMass
        * pow3(semimajorAxis / (astronomicalUnit / solarRadius))
        / pow2(period / sideralYear);
}

Binary::Binary(const Binary& source) :
    massRatio(source.massRatio),
    period(source.period),
    semimajorAxis(source.semimajorAxis),
    inclination(source.inclination),
    eccentricity(source.eccentricity),
    argumentOfPeriastron(source.argumentOfPeriastron),
    epoch(source.epoch),
    totalMass(source.totalMass),
    primary(source.primary),
    secondary(source.secondary),
    disc(source.disc),
    systemVelocity(source.systemVelocity),
    hasDiskFlag(source.hasDiskFlag)
{ }

Binary::~Binary()
{ } }
```

Model.h

```
#pragma once

#include <string>
#include <sstream>
#include <iostream>
#include <fstream>
#include <exception>
#include <clocale>
#include <Shlwapi.h>
#include <iomanip>

#include <tchar.h>
#include "IniReader.h"
#include "Calc.h"
#include "Binary.h"
#include "Passband.h"
#include "Mesh3.h"
#include "RocheSurface.h"
#include "SphericalSurface.h"
#include "Orbit.h"
#include "CircularOrbit.h"
#include "EccentricOrbit.h"
#include "StellarEffect.h"
#include "SystemEffect.h"
```

```
25  #include "GravityAcceleration.h"
26  #include "VonZeipelGravityDarkening.h"
27  #include "Pulsation.h"
28  #include "Spots.h"
29  #include "DjurasevicReflection.h"
30  #include "KhruzinaReflection.h"
31  #include "NoReflection.h"
32  #include "VisibilityDetection.h"
33  #include "GridVisibility.h"
34  #include "Color.h"
35  #include "PlanckFlux.h"
36  #include "KuruczFlux.h"
37  #include "Palette.h"
38  #include "DiscEffect.h"
39  #include "DiscTemperature.h"
40  #include "DiscGravity.h"
41  #include "DiscSpots.h"
42  #include "Velocity.h"
43  #include "PaintersVisibility.h"
44
45  namespace Infinity {
46  class Model
47  {
48  public:
49      Model(Binary binary, std::vector<double> times,
50          std::vector<Passband> passbands, std::wstring prefix, double
51  phaseShift,
52          double magnitudeShift, int reflectionType, int fluxType, int mesh,
53          int meshType, int detailed, int xUnits, int yUnits, int yUnitsType,
54          int visibilityDetectionFlag, int triangleIntersectionFlag,
55          int adaptiveLevel);
56      Model(const Model& source);
57      ~Model(void);
58
59      Model& operator=(const Model& rhs);
60
61      void run();
62
63      static Model Read(LPCTSTR configFile);
64
65  private:
66      Binary binary;
67
68      std::vector<double> phases;
69
70      std::wstring prefix;
71      std::vector<Passband> passbands;
72
73      double phaseShift, magnitudeShift;
74      int reflectionTypeFlag, fluxTypeFlag;
75      int meshFlag, meshTypeFlag;
76      int detailedFlag;
77      int xUnitsFlag, yUnitsFlag, yUnitsTypeFlag;
78      int adaptiveLevelFlag;
79
80      int visibilityDetectionFlag, triangleIntersectionFlag;
81      void writePly(Mesh3& primary, Mesh3& secondary, Mesh3& disk,
```

```
82          std::wstring prefix);
83
84      Surface* getSurface(Star* star, double massRatio, double separation,
85          Mesh3& mesh);
86      Orbit* getOrbit(bool circular);
87      SystemEffect* getReflection(Mesh3* primary, Mesh3* secondary, Mesh3*
88  disc);
89      SystemEffect* getFlux(Mesh3* primary, Mesh3* secondary, Mesh3* disc);
90
91      void writeLC(std::vector<double>& fluxes);
92      void writeRV(std::vector<double>& rv, std::wstring sufix);
93      void writeOutput(Binary *binary, RocheSurface *primarySurface,
94          RocheSurface *secondarySurface, Mesh3 *primary, Mesh3 *secondary,
95          Mesh3* disc, std::wstring prefix);
96
97      void getMinMax(Mesh3& mesh, double& minTemp, double& maxTemp,
98          double& minLogG, double& maxLogG, double& minFlux, double& maxFlux,
99          double& minRV, double& maxRV);
100 };
101
102 inline Orbit* Model::getOrbit(bool circular)
103 {
104     if (circular)
105         return new CircularOrbit(&binary, phases);
106     else
107         return new EccentricOrbit(&binary, phases);
108 }
109
110 inline SystemEffect* Model::getReflection(Mesh3* primary, Mesh3* secondary,
111     Mesh3* disc)
112 {
113     SystemEffect* result;
114
115     switch (reflectionTypeFlag)
116     {
117     case NO_REFLECTION:
118         result = new NoReflection(&binary, primary, secondary, disc);
119         break;
120     case DJURASEVIC_REFLECTION:
121         result = new DjurasevicReflection(&binary, primary, secondary, disc);
122         break;
123     default:
124     case KHRUZINA_REFLECTION:
125         result = new KhruzinaReflection(&binary, primary, secondary, disc);
126         break;
127     }
128
129     return result;
130 }
131
132 inline SystemEffect* Model::getFlux(Mesh3* primary, Mesh3* secondary,
133     Mesh3* disc)
134 {
135     SystemEffect* result;
136
137     switch (fluxTypeFlag)
138     {
```

```
139         case KURUTZ_FLUX:
140             result = new KuruczFlux(&binary, primary, secondary, disc,
141     &passbands);
142             break;
143         default:
144         case PLANCK_FLUX:
145             result = new PlanckFlux(&binary, primary, secondary, disc,
146     &passbands);
147             break;
148         }
149 
150         return result;
151     } }
```

Model.cpp

```
 1   #include "stdafx.h"
 2   #include "Model.h"
 3   
 4   using namespace std;
 5   using namespace DirectX::SimpleMath;
 6   
 7   namespace Infinity {
 8   Model::Model(Binary binary, vector<double> phases, vector<Passband>
 9   passbands,
10                wstring prefix, double phaseShift, double magnitudeShift,
11                int reflectionType, int fluxType, int mesh, int meshType,
12                int detailed, int xUnits, int yUnits, int yUnitsType,
13                int visibilityDetectionFlag, int triangleIntersectionFlag,
14                int adaptiveLevel) :
15       binary(binary),
16       phases(phases),
17       passbands(passbands),
18       prefix(prefix),
19       phaseShift(phaseShift),
20       magnitudeShift(magnitudeShift),
21       reflectionTypeFlag(reflectionType),
22       fluxTypeFlag(fluxType),
23       meshFlag(mesh),
24       meshTypeFlag(meshType),
25       detailedFlag(detailed),
26       xUnitsFlag(xUnits),
27       yUnitsFlag(yUnits),
28       yUnitsTypeFlag(yUnitsType),
29       visibilityDetectionFlag(visibilityDetectionFlag),
30       triangleIntersectionFlag(triangleIntersectionFlag),
31       adaptiveLevelFlag(adaptiveLevel)
32   { }
33   
34   Model::Model(const Model& source)
35   {
36       binary = source.binary;
37       phases = source.phases;
38       passbands = source.passbands;
39       prefix = source.prefix;
40       phaseShift = source.phaseShift;
```

```
41      magnitudeShift = source.magnitudeShift;
42      reflectionTypeFlag = source.reflectionTypeFlag;
43      fluxTypeFlag = source.fluxTypeFlag;
44      meshFlag = source.meshFlag;
45      meshTypeFlag = source.meshTypeFlag;
46      detailedFlag = source.detailedFlag;
47      xUnitsFlag = source.xUnitsFlag;
48      yUnitsFlag = source.yUnitsFlag;
49      yUnitsTypeFlag = source.yUnitsTypeFlag;
50      visibilityDetectionFlag = source.visibilityDetectionFlag;
51      triangleIntersectionFlag = source.triangleIntersectionFlag;
52      adaptiveLevelFlag = source.adaptiveLevelFlag;
53  }
54
55  Model& Model::operator=(const Model& source)
56  {
57      binary = source.binary;
58      phases = source.phases;
59      passbands = source.passbands;
60      prefix = source.prefix;
61      phaseShift = source.phaseShift;
62      magnitudeShift = source.magnitudeShift;
63      reflectionTypeFlag = source.reflectionTypeFlag;
64      fluxTypeFlag = source.fluxTypeFlag;
65      meshFlag = source.meshFlag;
66      meshTypeFlag = source.meshTypeFlag;
67      detailedFlag = source.detailedFlag;
68      xUnitsFlag = source.xUnitsFlag;
69      yUnitsFlag = source.yUnitsFlag;
70      yUnitsTypeFlag = source.yUnitsTypeFlag;
71      visibilityDetectionFlag = source.visibilityDetectionFlag;
72      triangleIntersectionFlag = source.triangleIntersectionFlag;
73      adaptiveLevelFlag = source.adaptiveLevelFlag;
74
75      return *this;
76  }
77
78  Model::~Model(void)
79  { }
80
81  void Model::run()
82  {
83      //Determine if we can make some optimizations
84      bool circular = isEqual(binary.getEccentricity(), 0.0);
85      bool hasDisc = binary.hasDisc();
86      bool surfacePerturbations = (binary.getPrimary()->getPerturbSurface()
87          || binary.getSecondary()->getPerturbSurface());
88      bool hasPrimaryPulsations = (binary.getPrimary()->getModeCount() > 0);
89      bool hasSecondaryPulsations = (binary.getSecondary()->getModeCount() >
90  0);
91      bool hasPrimarySpots = (binary.getPrimary()->getSpotCount() > 0);
92      bool hasSecondarySpots = (binary.getSecondary()->getSpotCount() > 0);
93      bool hasDiscSpots = (binary.getDisc()->getSpotCount() > 0);
94
95      //Create components
96      Mesh3 initialPrimary = Mesh3::CreateIcosahedron();
97      initialPrimary.subdivide(binary.getPrimary()->getSubdivisionLevel());
```

```
98      Mesh3 initialSecondary = Mesh3::CreateIcosahedron();
99      initialSecondary.subdivide(binary.getSecondary()->getSubdivisionLevel());
100
101     Mesh3 initialDisc(0, 0);
102     if (hasDisc)
103         initialDisc = Mesh3::CreateLathe(*binary.getDisc()-
104 >getCrossSection(),
105         binary.getDisc()->getSegments());
106
107     //Add the reference phase for the output at the end
108     phases.push_back(0.25);
109
110     Orbit* orbit = getOrbit(circular);
111     size_t pointCount = phases.size();
112     size_t lcCount = passbands.size();
113
114     vector<double> fluxes(passbands.size() * pointCount, 0.0);
115     vector<double> primaryRV(pointCount, 0.0);
116     vector<double> secondaryRV(pointCount, 0.0);
117
118     streamsize defaultWidth = cout.width();
119
120     for (unsigned short i = 0; i < pointCount; ++i)
121     {
122         Mesh3 primary = initialPrimary;
123         Mesh3 secondary = initialSecondary;
124         Mesh3 disc = initialDisc;
125
126         double separation = orbit->getSeparation(i);
127
128         Surface* primarySurface = getSurface(binary.getPrimary(),
129             binary.getMassRatio(), separation, primary);
130         Surface* secondarySurface = getSurface(binary.getSecondary(),
131             1.0 / binary.getMassRatio(), separation, secondary);
132         primary.pushTo(primarySurface);
133         secondary.pushTo(secondarySurface);
134
135         primary.init(lcCount);
136         secondary.init(lcCount);
137         disc.init(lcCount);
138
139         //Gravity acceleration
140         StellarEffect* primaryGravityAcceleration =
141             new GravityAcceleration(&binary, binary.getPrimary(), &primary,
142             binary.getMassRatio());
143         StellarEffect* secondaryGravityAcceleration  =
144             new GravityAcceleration(&binary, binary.getSecondary(),
145 &secondary,
146             1.0/ binary.getMassRatio());
147         primaryGravityAcceleration->run();
148         secondaryGravityAcceleration->run();
149
150         StellarEffect* primaryVelocity = new Velocity(&binary,
151             binary.getPrimary(), &primary);
152         StellarEffect* secondaryVelocity = new Velocity(&binary,
153             binary.getSecondary(), &secondary);
154         primaryVelocity->run();
```

```
155          secondaryVelocity->run();
156
157          //Gravity darkening
158          StellarEffect* primaryGravityDarkening =
159              new VonZeipelGravityDarkening(&binary, binary.getPrimary(),
160              &primary);
161          StellarEffect* secondaryGravityDarkening  =
162              new VonZeipelGravityDarkening(&binary, binary.getSecondary(),
163              &secondary);
164          primaryGravityDarkening->run();
165          secondaryGravityDarkening->run();
166
167          //Disc temperature & gravity distribution
168          DiscEffect* discGravity = new DiscGravity(&binary, binary.getDisc(),
169              &disc);
170          discGravity->run();
171
172          DiscEffect* diskTemp = new DiscTemperature(&binary, binary.getDisc(),
173              &disc);
174          diskTemp->run();
175
176          //Spots
177          StellarEffect* primarySpots = new Spots(&binary, binary.getPrimary(),
178              &primary);
179          StellarEffect* secondarySpots = new Spots(&binary,
180  binary.getSecondary(),
181              &secondary);
182
183          if (hasPrimarySpots)
184              primarySpots->run();
185
186          if (hasSecondarySpots)
187              secondarySpots->run();
188
189          DiscEffect* discSpots = new DiscSpots(&binary, binary.getDisc(),
190  &disc);
191          if (hasDisc && hasDiscSpots)
192              discSpots->run();
193
194          //Reflection
195          SystemEffect* reflection = getReflection(&primary, &secondary,
196  &disc);
197          reflection->run();
198
199          //Position the components
200          primary.transform(orbit->getPrimaryWorldMatrix(i));
201          secondary.transform(orbit->getSecondaryWorldMatrix(i));
202          if (hasDisc)
203              disc.transform(orbit->getDiskWorldMatrix(i));
204
205          //Visibility detection
206          VisibilityDetection* visibility = new GridVisibility(&primary,
207              &secondary, &disc, adaptiveLevelFlag,
208              triangleIntersectionFlag, true);
209          visibility->run();
210
211          //Flux
```

```
212          SystemEffect* flux = getFlux(&primary, &secondary, &disc);
213          flux->run();
214
215          //Calculate the light curve
216          double totalArea = 0.0;
217          double primaryArea = 0.0;
218          double secondaryArea = 0.0;
219          double discArea = 0.0;
220
221          size_t primaryCount = primary.getTriangleCount();
222          size_t secondaryCount = secondary.getTriangleCount();
223          size_t discCount = disc.getTriangleCount();
224
225          for (size_t j = 0; j < primaryCount; ++j)
226          {
227              if (primary.getVisibility(j) > 0.0)
228              {
229                  double area = primary.getProjectedArea(j);
230                  double cosGamma = primary.getNormal(j)->x;
231                  primaryArea += area;
232                  for (size_t lc = 0; lc < lcCount; ++lc)
233                      fluxes[i * lcCount + lc] += primary.getFlux(lc, j) *
234  area;
235              }
236          }
237
238          for (size_t j = 0; j < secondaryCount; ++j)
239          {
240              if (secondary.getVisibility(j) > 0.0)
241              {
242                  double area = secondary.getProjectedArea(j);
243                  double cosGamma = secondary.getNormal(j)->x;
244
245                  secondaryArea += area;
246                  for (size_t lc = 0; lc < lcCount; ++lc)
247                      fluxes[i * lcCount + lc] += secondary.getFlux(lc, j) *
248  area;
249              }
250          }
251
252          if (hasDisc)
253          {
254              for (size_t j = 0; j < discCount; ++j)
255              {
256                  if (disc.getVisibility(j) > 0.0)
257                  {
258                      double area = disc.getProjectedArea(j);
259                      double cosGamma = disc.getNormal(j)->x;
260
261                      discArea += area;
262                      for (size_t lc = 0; lc < lcCount; ++lc)
263                          fluxes[i * lcCount + lc] += disc.getFlux(lc, j) *
264  area;
265                  }
266              }
267          }
268          totalArea = primaryArea + secondaryArea + discArea;
```

```
269
270         //And the radial velocity curve;
271         Vector3 omega(0, 0, TWO_PI_F / static_cast<float>
272             (binary.getPeriod() * 86400.0));
273         float q = static_cast<float>(binary.getMassRatio());
274         float absSeparation = static_cast<float>
275             (orbit->getSeparation(i) * binary.getSemimajorAxis() * 6.955E5);
276
277         Vector3 primaryCOM( q/(1+q) * absSeparation, 0, 0);
278         Vector3 secondaryCOM( 1/(1+q) * absSeparation, 0, 0);
279
280         omega = Vector3::TransformNormal(omega,
281             orbit->getPrimaryWorldMatrix(i));
282         primaryCOM = Vector3::TransformNormal(primaryCOM,
283             orbit->getPrimaryWorldMatrix(i));
284         secondaryCOM = Vector3::TransformNormal(secondaryCOM,
285             orbit->getSecondaryWorldMatrix(i));
286
287         Vector3 primaryCOMVelocity = omega.Cross(primaryCOM);
288         Vector3 secondaryCOMVelocity = omega.Cross(secondaryCOM);
289         Vector3 primaryAverageVelocity(0, 0, 0);
290         Vector3 secondaryAverageVelocity(0, 0, 0);
291
292         if (primaryArea > 0)
293             for (size_t j = 0; j < primaryCount; ++j)
294             {
295                 if (primary.getVisibility(j) > 0.0)
296                 {
297                     double area = primary.getProjectedArea(j);
298                     primaryAverageVelocity +=  *primary.getVelocity(j)
299                         * static_cast<float>(area / primaryArea);
300                 }
301             }
302
303         primaryRV[i] = binary.getSystemVelocity() + (primaryCOMVelocity
304             + primaryAverageVelocity).x;
305
306         if (secondaryArea > 0)
307             for (size_t j = 0; j < secondaryCount; ++j)
308             {
309                 if (secondary.getVisibility(j) > 0.0)
310                 {
311                     double area = secondary.getProjectedArea(j);
312                     secondaryAverageVelocity +=  *secondary.getVelocity(j)
313                         * static_cast<float>(area / secondaryArea);
314                 }
315             }
316
317         secondaryRV[i] = binary.getSystemVelocity() + (secondaryCOMVelocity
318             + secondaryAverageVelocity).x;
319
320         // OUTPUTS
321         cout << "\r" << setw(3) << (int)(i * 100.0 / pointCount)
322             << setw(defaultWidth) << "% completed." << flush;
323
324         //Mesh
325         switch (meshFlag)
```

```
326         {
327         case OUTPUT_ONCE:
328             if (i == pointCount - 1)
329                 writePly(primary, secondary, disc, prefix);
330             break;
331         case OUTPUT_ON_EACH_PHASE:
332             {
333                 wstringstream ss;
334                 ss << prefix << L"." << setfill(L'0') << setw(4) << i;
335                 writePly(primary, secondary, disc, ss.str());
336             }
337             break;
338         default:
339         case NO_OUTPUT:
340             break;
341         }
342
343         switch (detailedFlag)
344         {
345         case OUTPUT_ONCE:
346             if (i == (pointCount - 1) && binary.getPrimary()->getShape() ==
347                 Star::Shape::Roche && binary.getSecondary()->getShape() ==
348                 Star::Shape::Roche)
349                 writeOutput(&binary, (RocheSurface*)primarySurface,
350                 (RocheSurface*)secondarySurface, &primary, &secondary,
351                 &disc, prefix);
352             break;
353         case OUTPUT_ON_EACH_PHASE:
354             if  (binary.getPrimary()->getShape() == Star::Shape::Roche
355                 && binary.getSecondary()->getShape() == Star::Shape::Roche)
356             {
357                 wstringstream ss;
358                 ss << prefix << L"." << setfill(L'0') << setw(4)  << i;
359                 writeOutput(&binary, (RocheSurface*)primarySurface,
360                     (RocheSurface*)secondarySurface, &primary, &secondary,
361                     &disc, prefix);
362             }
363             break;
364         default:
365         case NO_OUTPUT:
366             break;
367         }
368
369         //Clean up
370         delete flux;
371         delete visibility;
372         delete reflection;
373         delete diskTemp;
374         delete discGravity;
375         delete primarySpots;
376         delete secondarySpots;
377         delete discSpots;
378         delete primaryPulsation;
379         delete secondaryPulsation;
380         delete primaryGravityDarkening;
381         delete secondaryGravityDarkening;
382         delete primaryVelocity;
```

```
383             delete secondaryVelocity;
384             delete primaryGravityAcceleration;
385             delete secondaryGravityAcceleration;
386             delete primarySurface;
387             delete secondarySurface;
388         }
389
390         cout << "\r" << setw(3) << 100 << setw(defaultWidth)
391             << "% completed.\n" << flush;
392
393         writeLC(fluxes);
394         writeRV(primaryRV, L".primaryRV");
395         writeRV(secondaryRV, L".secondaryRV");
396
397         delete orbit;
398     }
399
400     Model Model::Read(LPCTSTR configFile)
401     {
402         BOOL fileExists = PathFileExists(configFile);
403
404         if (fileExists == FALSE)
405                 throw std::exception("Model file is missing. Unable to
406     continue.");
407
408         TCHAR prefix[MAX_PATH];
409         _tcscpy_s(prefix, configFile);
410         PathRemoveExtension(prefix);
411
412         double massRatio = IniReader::GetDoubleValue(L"Binary",
413             L"MassRatio", configFile);
414         double period = IniReader::GetDoubleValue(L"Binary",
415             L"Period", configFile);
416         double semimajorAxis = IniReader::GetDoubleValue(L"Binary",
417             L"SemimajorAxis", configFile);
418         double inclination = deg2rad(IniReader::GetDoubleValue(L"Binary",
419             L"Inclination", configFile));
420         double eccentricity = IniReader::GetDoubleValue(L"Binary",
421             L"Eccentricity", configFile);
422         double argumentOfPeriastron =
423     deg2rad(IniReader::GetDoubleValue(L"Binary",
424             L"ArgumentOfPeriastron", configFile));
425         double epoch = IniReader::GetDoubleValue(L"Binary", L"Epoch",
426     configFile);
427         double systemVelocity = IniReader::GetDoubleValue(L"Binary",
428             L"SystemVelocity", configFile);
429
430         Binary binary(massRatio, period, semimajorAxis, inclination,
431     eccentricity,
432             argumentOfPeriastron, epoch, systemVelocity);
433
434         wstring shapeString = IniReader::GetStringValue(L"Primary",
435             L"Shape", configFile);
436         double asynchronicity = IniReader::GetDoubleValue(L"Primary",
437             L"Asynchronicity", configFile);
438         double polarRadius = IniReader::GetDoubleValue(L"Primary",
439             L"PolarRadius", configFile);
```

```
440         double temperature = IniReader::GetDoubleValue(L"Primary",
441             L"EffectiveTemperature", configFile);
442         double gdExponent = IniReader::GetDoubleValue(L"Primary",
443             L"GravityDarkeningExponent", configFile);
444         double albedo = IniReader::GetDoubleValue(L"Primary",
445             L"Albedo", configFile);
446         size_t subdivisionLevel = IniReader::GetIntValue(L"Primary",
447             L"SubdivisionLevel", configFile);
448         wstring atmosphereModel = IniReader::GetStringValue(L"Primary",
449             L"AtmosphereModel", configFile).c_str();
450         double metallicity = IniReader::GetDoubleValue(L"Primary",
451             L"Metallicity", configFile);
452         double turbulence = IniReader::GetDoubleValue(L"Primary",
453             L"Microturbulence", configFile);
454         wstring palette = IniReader::GetStringValue(L"Primary",
455             L"Palette", configFile);
456
457         Star::Shape shape;
458
459         if (shapeString == L"Roche")
460             shape = Star::Shape::Roche;
461         if (shapeString == L"Sphere")
462             shape = Star::Shape::Sphere;
463
464         wstring perturbSurfaceStr = IniReader::GetStringValue(L"Primary",
465             L"PerturbSurface", configFile);
466         bool perturbSurface = false;
467         if (perturbSurfaceStr == L"Yes")
468             perturbSurface = true;
469
470         binary.createPrimary(shape, asynchronicity, polarRadius, temperature,
471             gdExponent, albedo, subdivisionLevel, atmosphereModel, metallicity,
472             turbulence, perturbSurface, palette);
473
474         int spotCount;
475         spotCount = IniReader::GetIntValue(L"Primary", L"SpotCount", configFile);
476         for(int i = 0; i < spotCount; i++)
477         {
478             wstring key = L"Spot" + std::to_wstring(i+1);
479             vector<double> spot = IniReader::GetDoubleVector(L"Primary",
480                 key.c_str(), configFile);
481
482             bool activeSpot = spot[4] == 1 ? true : false;
483             if(activeSpot)
484                 binary.getPrimary()->addSpot(deg2rad(spot[0]), deg2rad(spot[1]),
485                 deg2rad(spot[2]), spot[3]);
486         }
487
488         shapeString = IniReader::GetStringValue(L"Secondary",
489             L"Shape", configFile);
490         asynchronicity = IniReader::GetDoubleValue(L"Secondary",
491             L"Asynchronicity", configFile);
492         polarRadius = IniReader::GetDoubleValue(L"Secondary",
493             L"PolarRadius", configFile);
494         temperature = IniReader::GetDoubleValue(L"Secondary",
495             L"EffectiveTemperature", configFile);
496         gdExponent = IniReader::GetDoubleValue(L"Secondary",
```

```
497             L"GravityDarkeningExponent", configFile);
498         albedo = IniReader::GetDoubleValue(L"Secondary",
499             L"Albedo", configFile);
500         subdivisionLevel = IniReader::GetIntValue(L"Secondary",
501             L"SubdivisionLevel", configFile);
502         atmosphereModel = IniReader::GetStringValue(L"Secondary",
503             L"AtmosphereModel", configFile).c_str();
504         metallicity = IniReader::GetDoubleValue(L"Secondary",
505             L"Metallicity", configFile);
506         turbulence = IniReader::GetDoubleValue(L"Secondary",
507             L"Microturbulence", configFile);
508         palette = IniReader::GetStringValue(L"Secondary",
509             L"Palette", configFile);
510 
511         if (shapeString == L"Roche")
512             shape = Star::Shape::Roche;
513         if (shapeString == L"Sphere")
514             shape = Star::Shape::Sphere;
515 
516         perturbSurfaceStr = IniReader::GetStringValue(L"Secondary",
517             L"PerturbSurface", configFile);
518         perturbSurface = false;
519         if (perturbSurfaceStr == L"Yes")
520             perturbSurface = true;
521 
522         binary.createSecondary(shape, asynchronicity, polarRadius, temperature,
523             gdExponent, albedo, subdivisionLevel, atmosphereModel, metallicity,
524             turbulence, perturbSurface, palette);
525 
526         spotCount = IniReader::GetIntValue(L"Secondary", L"SpotCount",
527     configFile);
528         for(int i = 0; i < spotCount; i++)
529         {
530             wstring key = L"Spot" + std::to_wstring(i+1);
531             vector<double> spot = IniReader::GetDoubleVector(L"Secondary",
532                 key.c_str(), configFile);
533 
534             bool activeSpot = spot[4] == 1 ? true : false;
535             if(activeSpot)
536                 binary.getSecondary()->addSpot(deg2rad(spot[0]),
537     deg2rad(spot[1]),
538                 deg2rad(spot[2]), spot[3]);
539         }
540 
541         //And now, the disc
542         shapeString = IniReader::GetStringValue(L"Disc", L"Shape", configFile);
543         if (shapeString == DISC_SHAPE_CONICAL_STR)
544         {
545             double innerRadius = IniReader::GetDoubleValue(L"Disc",
546                 L"InnerRadius", configFile);
547             double outerRadius = IniReader::GetDoubleValue(L"Disc",
548                 L"OuterRadius", configFile);
549             double innerThickness = IniReader::GetDoubleValue(L"Disc",
550                 L"InnerThickness", configFile);
551             double outerThickness = IniReader::GetDoubleValue(L"Disc",
552                 L"OuterThickness", configFile);
553             double innerTemp = IniReader::GetDoubleValue(L"Disc",
```

```
554             L"InnerTemperature", configFile);
555         double outerTemp = IniReader::GetDoubleValue(L"Disc",
556             L"OuterTemperature", configFile);
557         double exponent = IniReader::GetDoubleValue(L"Disc",
558             L"Exponent", configFile);
559         wstring distributionString = IniReader::GetStringValue(L"Disc",
560             L"Distribution", configFile);
561         int distribution = -1;
562         if (distributionString == DISC_TEMP_DIST_ALPHA_STR)
563             distribution = DISC_TEMP_DIST_ALPHA;
564         if (distributionString == DISC_TEMP_DIST_DJURASEVIC_STR)
565             distribution = DISC_TEMP_DIST_DJURASEVIC;
566         if (distributionString == DISC_TEMP_DIST_ZOLA_STR)
567             distribution = DISC_TEMP_DIST_ZOLA;
568
569         int radialSegments = IniReader::GetIntValue(L"Disc",
570             L"RadialSegments", configFile);
571         int polarSegments = IniReader::GetIntValue(L"Disc",
572             L"PolarSegments", configFile);
573         int innerSegments = IniReader::GetIntValue(L"Disc",
574             L"InnerSegments", configFile);
575         int outerSegments = IniReader::GetIntValue(L"Disc",
576             L"OuterSegments", configFile);
577
578         palette = IniReader::GetStringValue(L"Disc", L"Palette", configFile);
579
580         CrossSection cs = CrossSection::CreateConical(
581             static_cast<float>(innerRadius),
582             static_cast<float>(outerRadius),
583             static_cast<float>(innerThickness),
584             static_cast<float>(outerThickness),
585             radialSegments, innerSegments, outerSegments);
586         binary.createDisc(Disc::Shape::Conical,  cs, polarSegments,
587 innerRadius,
588             outerRadius, innerTemp, outerTemp, exponent, distribution,
589 palette);
590     }
591
592     if (shapeString == DISC_SHAPE_TOROIDAL_STR)
593     {
594         double radius = IniReader::GetDoubleValue(L"Disc",
595             L"Radius", configFile);
596         double discSemiMajorAxis = IniReader::GetDoubleValue(L"Disc",
597             L"SemiMajorAxis", configFile);
598         double discSemiMinorAxis = IniReader::GetDoubleValue(L"Disc",
599             L"SemiMinorAxis", configFile);
600         double innerTemp = IniReader::GetDoubleValue(L"Disc",
601             L"InnerTemperature", configFile);
602         double outerTemp = IniReader::GetDoubleValue(L"Disc",
603             L"OuterTemperature", configFile);
604         double exponent = IniReader::GetDoubleValue(L"Disc",
605             L"OuterTemperature", configFile);
606         wstring distributionString = IniReader::GetStringValue(L"Disc",
607             L"Distribution", configFile);
608         int distribution = -1;
609         if (distributionString == DISC_TEMP_DIST_ALPHA_STR)
610             distribution = DISC_TEMP_DIST_ALPHA;
```

```
611             if (distributionString == DISC_TEMP_DIST_DJURASEVIC_STR)
612                 distribution = DISC_TEMP_DIST_DJURASEVIC;
613             if (distributionString == DISC_TEMP_DIST_ZOLA_STR)
614                 distribution = DISC_TEMP_DIST_ZOLA;
615 
616             int radialSegments = IniReader::GetIntValue(L"Disc",
617                 L"RadialSegments", configFile);
618             int polarSegments = IniReader::GetIntValue(L"Disc",
619                 L"PolarSegments", configFile);
620             int innerSegments = IniReader::GetIntValue(L"Disc",
621                 L"InnerSegments", configFile);
622             int outerSegments = IniReader::GetIntValue(L"Disc",
623                 L"OuterSegments", configFile);
624 
625             palette = IniReader::GetStringValue(L"Disc", L"Palette", configFile);
626 
627             CrossSection cs =
628                 CrossSection::CreateToroidal(static_cast<float>(radius),
629                 static_cast<float>(discSemiMajorAxis),
630                 static_cast<float>(discSemiMinorAxis),
631                 radialSegments);
632             binary.createDisc(Disc::Shape::Toroidal, cs, polarSegments,
633                 radius - discSemiMajorAxis, radius + discSemiMajorAxis,
634     innerTemp,
635                 outerTemp, exponent, distribution, palette);
636         }
637 
638         spotCount = IniReader::GetIntValue(L"Disc", L"SpotCount", configFile);
639         for(int i = 0; i < spotCount; i++)
640         {
641             wstring key = L"Spot" + std::to_wstring(i+1);
642             vector<double> spot = IniReader::GetDoubleVector(L"Disc",
643                 key.c_str(), configFile);
644 
645             bool activeSpot = spot[3] == 1 ? true : false;
646             if(activeSpot)
647                 binary.getDisc()->addSpot(deg2rad(spot[0]),
648                 deg2rad(spot[1]), spot[2]);
649         }
650 
651         double phaseShift = IniReader::GetDoubleValue(L"Model",
652             L"PhaseShift", configFile);
653         double magnitudeShift = IniReader::GetDoubleValue(L"Model",
654             L"MagnitudeShift", configFile);
655 
656         vector<wstring> passbandIDs = IniReader::GetStringVector(L"Model",
657             L"Passbands", configFile);
658         vector<Passband> passbands;
659         for (auto it = passbandIDs.begin(); it != passbandIDs.end(); it++)
660             passbands.push_back(Passband::FromID(*it, &binary));
661 
662         //Load phases
663         TCHAR phasesPath[MAX_PATH];
664 
665         _tcscpy_s(phasesPath, configFile);
666         PathRemoveFileSpec(phasesPath);
667         PathAppend(phasesPath, IniReader::GetStringValue(L"Model",
```

```
668         L"Phases", configFile).c_str());
669
670     ifstream phasesFile(phasesPath);
671     vector<double> phases;
672
673     string row;
674     _locale_t locale = _create_locale(LC_NUMERIC, "english");
675
676     while( getline( phasesFile, row ) )
677     {
678         if(row == "" || row[0] == '#')
679             continue;
680
681         vector<string> tokens = split(const_cast<char*> (row.c_str()), "
682 \t");
683         double value =  _atof_l(tokens[0].c_str(), locale);
684         phases.push_back(value);
685     }
686
687     wstring reflectionStr = IniReader::GetStringValue(L"Model",
688         L"Reflection", configFile);
689     int reflectionType;
690     if(reflectionStr == KHRUZINA_REFLECTION_STR)
691         reflectionType = KHRUZINA_REFLECTION;
692     else if(reflectionStr == DJURASEVIC_REFLECTION_STR)
693         reflectionType = DJURASEVIC_REFLECTION;
694     else
695         reflectionType= NO_REFLECTION;
696
697     wstring fluxStr = IniReader::GetStringValue(L"Model", L"Flux",
698 configFile);
699     int fluxType;
700     if(fluxStr == PLANCK_FLUX_STR)
701         fluxType = PLANCK_FLUX;
702     else
703         fluxType= KURUTZ_FLUX;
704
705     wstring meshStr = IniReader::GetStringValue(L"Output", L"Mesh",
706 configFile);
707     int mesh;
708     if(meshStr == OUTPUT_ON_EACH_PHASE_STR)
709         mesh = OUTPUT_ON_EACH_PHASE;
710     else if(meshStr == OUTPUT_ONCE_STR)
711         mesh = OUTPUT_ONCE;
712     else
713         mesh = NO_OUTPUT;
714
715     wstring meshTypeStr = IniReader::GetStringValue(L"Output",
716         L"MeshType", configFile);
717     int meshType;
718     if(meshTypeStr == MESH_3D_STR)
719         meshType = MESH_3D;
720     else if (meshTypeStr == MESH_2D_STR)
721         meshType = MESH_2D;
722     else
723         meshType = MESH_HIDDEN;
724
```

```
725       wstring detailedStr = IniReader::GetStringValue(L"Output",
726           L"Detailed", configFile);
727
728       int detailed;
729       if(detailedStr == OUTPUT_ON_EACH_PHASE_STR)
730           detailed = OUTPUT_ON_EACH_PHASE;
731       else if(detailedStr == OUTPUT_ONCE_STR)
732           detailed = OUTPUT_ONCE;
733       else
734           detailed = NO_OUTPUT;
735
736       wstring xUnitsStr = IniReader::GetStringValue(L"Output",
737           L"XUnits", configFile);
738       int xUnits;
739       if(xUnitsStr == XUNITS_PHASE_STR)
740           xUnits = XUNITS_PHASE;
741       else
742           xUnits = XUNITS_TIME;
743
744       wstring yUnitsStr = IniReader::GetStringValue(L"Output",
745           L"YUnits", configFile);
746       int yUnits;
747       if(yUnitsStr == YUNITS_FLUX_STR)
748           yUnits = YUNITS_FLUX;
749       else
750           yUnits = YUNITS_MAGNITUDE;
751
752       wstring yUnitsTypeStr = IniReader::GetStringValue(L"Output",
753           L"YUnitsType", configFile);
754       int yUnitsType;
755       if(yUnitsTypeStr == YUNITS_ABSOLUTE_STR)
756           yUnitsType = YUNITS_ABSOLUTE;
757       else
758           yUnitsType = YUNITS_NORMALIZED;
759
760       wstring visibilityDetectionStr =
761           IniReader::GetStringValue(L"VisibilityDetection", L"Type",
762   configFile);
763       int visibilityDetection;
764       if (visibilityDetectionStr == GRID_VISIBILITY_STR)
765           visibilityDetection = GRID_VISIBILITY;
766       else visibilityDetection = CONVEX_HULL_VISIBILITY;
767
768       wstring triangleStr =
769           IniReader::GetStringValue(L"VisibilityDetection",
770           L"TriangleIntersections", configFile);
771       int triangle;
772       if (triangleStr == TRIANGLE_FULL_STR)
773           triangle = TRIANGLE_FULL;
774       else triangle = TRIANGLE_POINT;
775
776       int adaptiveLevel = IniReader::GetIntValue(L"VisibilityDetection",
777           L"AdaptiveLevel", configFile);
778
779
780       return Model(binary, phases, passbands, prefix, phaseShift,
781   magnitudeShift,
```

```
782             reflectionType, fluxType, mesh, meshType, detailed, xUnits, yUnits,
783             yUnitsType, visibilityDetection, triangle, adaptiveLevel);
784     }
785
786     void Model::writePly(Mesh3& primary, Mesh3& secondary,
787         Mesh3& disc, std::wstring prefix)
788     {
789         ////determine temperature min/max values
790         double minTempPrimary, maxTempPrimary, minLogGPrimary, maxLogGPrimary,
791             minFLuxPrimary, maxFluxPrimary, minRVPrimary, maxRVPrimary;
792         double minTempSecondary, maxTempSecondary, minLogGSecondary,
793             maxLogGSecondary, minFLuxSecondary, maxFluxSecondary, minRVSecondary,
794             maxRVSecondary;
795         double minTempDisc, maxTempDisc, minLogGDisc, maxLogGDisc, minFLuxDisc,
796             maxFluxDisc, minRVDisc, maxRVDisc;
797
798         getMinMax(primary, minTempPrimary, maxTempPrimary, minLogGPrimary,
799             maxLogGPrimary, minFLuxPrimary, maxFluxPrimary, minRVPrimary,
800             maxRVPrimary);
801         getMinMax(secondary, minTempSecondary, maxTempSecondary,
802     minLogGSecondary,
803             maxLogGSecondary, minFLuxSecondary, maxFluxSecondary, minRVSecondary,
804             maxRVSecondary);
805         getMinMax(disc, minTempDisc, maxTempDisc, minLogGDisc, maxLogGDisc,
806             minFLuxDisc, maxFluxDisc, minRVDisc, maxRVDisc);
807
808         ofstream file(prefix + L".ply");
809
810         file << "ply" << endl;
811         file << "format ascii 1.0" << endl;
812
813         size_t primaryVertexCount = primary.getVertexCount();
814         size_t secondaryVertexCount = secondary.getVertexCount();
815         size_t discVertexCount = disc.getVertexCount();
816         size_t primaryTriangleCount = primary.getTriangleCount();
817         size_t secondaryTriangleCount = secondary.getTriangleCount();
818         size_t discTriangleCount = disc.getTriangleCount();
819
820
821         file << "element vertex " << primaryVertexCount + secondaryVertexCount
822             + discVertexCount << endl;
823         file << "property float x" << endl;
824         file << "property float y" << endl;
825         file << "property float z" << endl;
826
827
828         file << "element face " << primaryTriangleCount + secondaryTriangleCount
829             + discTriangleCount << endl;
830         file << "property list uchar int vertex_index" << endl;
831         file << "property uchar red" << endl;
832         file << "property uchar green" << endl;
833         file << "property uchar blue" << endl;
834         file << "end_header" <<endl;
835
836         for (size_t i = 0; i < primaryVertexCount; i++)
837             file << primary.getVertex(i)->x << " " << primary.getVertex(i)->y
838             << " " << primary.getVertex(i)->z << endl;
```

```
839
840         for (size_t i = 0; i < secondaryVertexCount; i++)
841             file << secondary.getVertex(i)->x << " " << secondary.getVertex(i)->y
842             << " " << secondary.getVertex(i)->z << endl;
843
844         for (size_t i = 0; i < discVertexCount; i++)
845             file << disc.getVertex(i)->x << " " << disc.getVertex(i)->y << " "
846             << disc.getVertex(i)->z << endl;
847
848         Palette* palette = binary.getPrimary()->getPalette();
849
850         for (size_t i = 0; i < primaryTriangleCount; i++)
851         {
852             double temperature = primary.getTemperature(i);
853
854             Color color = palette->getColor(temperature, minTempPrimary,
855                 maxTempPrimary);
856
857             if (meshTypeFlag == MESH_HIDDEN && !primary.getVisibility(i))
858                 color = Color(51, 0, 102);
859
860             file << "3 " << primary.getIndex(3*i) << " "
861                             << primary.getIndex(3*i+1) << " "
862                             << primary.getIndex(3*i+2) << " "
863                             << color.r << " "
864                             << color.g << " "
865                             << color.b << " "
866                             << endl;
867         }
868
869         palette = binary.getSecondary()->getPalette();
870         for (size_t i = 0; i < secondaryTriangleCount; i++)
871         {
872             double temperature = secondary.getTemperature(i);
873             Color color = palette->getColor(temperature, minTempSecondary,
874                 maxTempSecondary);
875
876             if (meshTypeFlag == MESH_HIDDEN && !secondary.getVisibility(i))
877                 color = Color(51, 0, 102);
878
879             file << "3 " << secondary.getIndex(3*i) + primaryVertexCount << " "
880                             << secondary.getIndex(3*i+1) + primaryVertexCount <<
881     " "
882                             << secondary.getIndex(3*i+2) + primaryVertexCount <<
883     " "
884                             << color.r << " "
885                             << color.g << " "
886                             << color.b << " "
887                             << endl;
888         }
889
890         palette = binary.getDisc()->getPalette();
891         for (size_t i = 0; i < discTriangleCount; i++)
892         {
893             double temperature = disc.getTemperature(i);
894             Color color = palette->getColor(temperature, minTempDisc,
895     maxTempDisc);
```

```
896
897         if (meshTypeFlag == MESH_HIDDEN && !disc.getVisibility(i))
898             color = Color(51, 0, 102);
899
900         file << "3 "
901             << disc.getIndex(3*i) + primaryVertexCount + secondaryVertexCount
902             << " "
903             << disc.getIndex(3*i+1) + primaryVertexCount +
904 secondaryVertexCount
905             << " "
906             << disc.getIndex(3*i+2) + primaryVertexCount +
907 secondaryVertexCount
908             << " "
909             << color.r << " "
910             << color.g << " "
911             << color.b << " "
912             << endl;
913     }
914 }
915
916 void Model::getMinMax(Mesh3& mesh, double& minTemp, double& maxTemp, double&
917 minGravity, double& maxGravity, double& minFlux, double& maxFlux, double&
918 minRV, double& maxRV)
919 {
920     minTemp = 1000000000; maxTemp = 0;
921     minGravity = 1000000000; maxGravity = 0;
922     minFlux = 1000000000; maxFlux = 0;
923     minRV   = 1000000000; maxRV = 0;
924
925     size_t triangleCount = mesh.getTriangleCount();
926     for (size_t i = 0; i < triangleCount; i++)
927     {
928         double temperature = mesh.getTemperature(i);
929         if(temperature < minTemp) minTemp = temperature;
930         if(temperature > maxTemp) maxTemp = temperature;
931
932         double g = mesh.getGravityAcceleration(i);
933         if(g < minGravity) minGravity = g;
934         if(g > maxGravity) maxGravity = g;
935
936         //Flux
937         double flux = mesh.getFlux(0, i);
938         if(flux < minFlux) minFlux = flux;
939         if(flux > maxFlux) maxFlux = flux;
940     }
941 }
942
943 void Model::writeLC(std::vector<double>& fluxes)
944 {
945     size_t pointCount = phases.size() - 1;
946     size_t lcCount = passbands.size();
947
948     for (size_t lc = 0; lc < lcCount; lc++)
949     {
950         ofstream file(prefix + L"." + passbands[lc].getID() + L".lc");
951
952         for (size_t i = 0; i < pointCount; ++i)
```

```
953             {
954                 double x, y;
955 
956                 switch (xUnitsFlag)
957                 {
958                 case XUNITS_TIME:
959                     x = phases[i] + phaseShift;
960                     if(x > 1.0) x -= 1.0;
961                     x = x * binary.getPeriod() + binary.getEpoch();
962                     break;
963                 default:
964                 case XUNITS_PHASE:
965                     x = phases[i] + phaseShift;
966                     if(x > 1.0) x -= 1.0;
967                     break;
968                 }
969 
970                 switch (yUnitsFlag)
971                 {
972                 case YUNITS_FLUX:
973                     if(yUnitsTypeFlag == YUNITS_NORMALIZED)
974                         y = fluxes[i * lcCount + lc]
975                     / fluxes[pointCount * lcCount + lc];
976                     else
977                         y = fluxes[i * lcCount + lc];
978                     break;
979                 default:
980                 case YUNITS_MAGNITUDE:
981                     if(yUnitsTypeFlag == YUNITS_NORMALIZED)
982                         y = -2.5 * log10(fluxes[i * lcCount + lc]
983                     / fluxes[pointCount * lcCount + lc]);
984                     else
985                     {
986                         y = -2.5 * log10(fluxes[i * lcCount + lc]);
987                         y += magnitudeShift;
988                     }
989                     break;
990                 }
991 
992                 file << setiosflags( ios::fixed ) << setprecision(10)
993                     << x << " " << y  << endl;
994             }
995             file.close();
996         }
997     }
998 
999     Surface* Model::getSurface(Star* star, double massRatio, double separation,
1000         Mesh3& mesh)
1001     {
1002         Surface* result;
1003 
1004         switch (star->getShape())
1005         {
1006         default:
1007         case Star::Roche:
1008             if (isEqual(binary.getEccentricity(), 0.0))
1009             {
```

```
            result = new RocheSurface(massRatio, 1.0, star->getPolarRadius(),
                star->getAsynchronicity(), star);
        }
        else
        {
            Mesh3 meshCopy = mesh;
            RocheSurface periastron(massRatio, (1.0 -
                binary.getEccentricity()),
                star->getPolarRadius(), star->getAsynchronicity(), star);
            meshCopy.pushTo(&periastron);
            meshCopy.init();
            double periastronVolume = meshCopy.getVolume();

            double radius = star->getPolarRadius();
            double corr = 0.0;
            double delR = 0.0001;

            do
            {
                radius += corr;
                RocheSurface current(massRatio, separation, radius,
                    star->getAsynchronicity(), star);
                meshCopy.pushTo(&current);
                meshCopy.init();
                double currVolume = meshCopy.getVolume();
                RocheSurface currentPlus(massRatio,
                separation, radius + delR,
                star->getAsynchronicity(), star);
                meshCopy.pushTo(&currentPlus);
                meshCopy.init();
                double currPlusVolume = meshCopy.getVolume();

                corr = (periastronVolume - currVolume)
                    / (currPlusVolume - currVolume) * delR;
            }
            while (abs(corr) > 1E-8);

            result = new RocheSurface(massRatio, separation, radius,
                star->getAsynchronicity(), star);
        }
        break;
    case Star::Sphere:
        result = new SphericalSurface(star);
        break;
    }

    return result;
}

void Model::writeOutput(Binary *binary, RocheSurface *primarySurface,
    RocheSurface *secondarySurface, Mesh3 *primary, Mesh3 *secondary,
    Mesh3* disc, std::wstring prefix)
{
    ofstream results(prefix + L".out");

    results << "Mass ratio:                          "
        << binary->getMassRatio() << endl;
```

```
1067        results << "Period:                              "
1068            << binary->getPeriod() << endl;
1069        results << "Semimajor axis:                      "
1070            << binary->getSemimajorAxis() << endl;
1071        results << "Inclination:                         "
1072            << binary->getInclination() << endl;
1073        results << "Epoch:                               "
1074            << binary->getEpoch() << endl;
1075        results << "Total mass:                          "
1076            << binary->getTotalMass() << endl << endl;
1077
1078        size_t minT1 = 0, maxT1 = 0, minG1 = 0, maxG1 = 0, minA1 = 0, maxA1 = 0;
1079        double avgT1 = 0.0, L1 = 0.0;
1080
1081        for (size_t i = 0; i < primary->getTriangleCount(); i++)
1082        {
1083            avgT1 += primary->getTemperature(i) * primary->getArea(i);
1084            L1 += primary->getArea(i) * pow4(primary->getTemperature(i));
1085
1086            if(primary->getTemperature(i) <
1087                primary->getTemperature(minT1)) minT1 = i;
1088            if(primary->getTemperature(i) >
1089                primary->getTemperature(maxT1)) maxT1 = i;
1090
1091            if(primary->getGravityAcceleration(i) <
1092                primary->getGravityAcceleration(minG1)) minG1 = i;
1093            if(primary->getGravityAcceleration(i) >
1094                primary->getGravityAcceleration(maxG1)) maxG1 = i;
1095
1096            if(primary->getArea(i) < primary->getArea(minA1)) minA1 = i;
1097            if(primary->getArea(i) > primary->getArea(maxA1)) maxA1 = i;
1098        }
1099        avgT1 /= primary->getMeshArea();
1100
1101        results << "Primary asynchronicity:                    "
1102            << binary->getPrimary()->getAsynchronicity() << endl;
1103        results << "Primary polar radius:                      "
1104            << binary->getPrimary()->getPolarRadius() << endl;
1105        results << "Primary critical polar radius:             "
1106            << primarySurface->getCriticalPolarRadius() << endl;
1107        results << "Primary sychronous critical polar radius:  "
1108            << primarySurface->getSychronousCriticalPolarRadius() << endl;
1109        results << "Primary front radius:                      "
1110            << primarySurface->getFrontRadius() << endl;
1111        results << "Primary critical front radius:             "
1112            << primarySurface->getCriticalFrontRadius() << endl;
1113        results << "Primary synchronous critical front radius: "
1114            << primarySurface->getSynchronousCriticalFrontRadius() << endl;
1115        results << "Primary back radius:                       "
1116            << primarySurface->getBackRadius() << endl;
1117        results << "Primary side radius:                       "
1118            << primarySurface->getSideRadius() << endl;
1119        results << "Primary mean Eggleton radius:              "
1120            << primarySurface->getMeanEggletonRadius() << endl;
1121        results << "Primary Rp/Rf:                             "
1122            << binary->getPrimary()->getPolarRadius()
1123            / primarySurface->getFrontRadius() << endl;
```

```
1124        results << "Primary filling factor:                    "
1125            << primarySurface->getFillingFactor() << endl;
1126        results << "Primary potential:                         "
1127            << primarySurface->getPotential() << endl;
1128        results << "Primary critical potential:                "
1129            << primarySurface->getCriticalPotential() << endl;
1130        results << "Primary synchronous critical potential:    "
1131            << primarySurface->getSynchronousCriticalPotential() << endl;
1132        results << "Primary effective temperature:             "
1133            << binary->getPrimary()->getTemperature() << endl;
1134        results << "Primary lowest temperature:                "
1135            << primary->getTemperature(minT1) << endl;
1136        results << "Primary highest temperature:               "
1137            << primary->getTemperature(maxT1) << endl;
1138        results << "Primary temperature range:                 "
1139            << primary->getTemperature(maxT1)
1140            - primary->getTemperature(minT1) << endl;
1141        results << "Primary average temperature:               "
1142            << avgT1 << endl;
1143        results << "Primary luminosity:                        "
1144            << 5.6704e-8 * pow2(6.955e8 * binary->getSemimajorAxis()) * L1
1145            / 3.839e26 << endl;
1146        results << "Primary temperature from luminosity:       "
1147            << root4(L1 / primary->getMeshArea()) << endl;
1148        results << "Primary gravity darkening exponent:        "
1149            << binary->getPrimary()->getGravityDarkeningExponent() << endl;
1150        results << "Primary albedo:                            "
1151            << binary->getPrimary()->getAlbedo() << endl;
1152        results << "Primary mass:                              "
1153            << binary->getPrimary()->getMass() << endl;
1154        results << "Primary effective gravity:                 "
1155            << log10(binary->getPrimary()->getEffectiveGravity()) << endl;
1156        results << "Primary lowest gravity:                    "
1157            << log10(primary->getGravityAcceleration(minG1)) << endl;
1158        results << "Primary highest gravity:                   "
1159            << log10(primary->getGravityAcceleration(maxG1)) << endl;
1160        results << "Primary gravity range:                     "
1161            << log10(primary->getGravityAcceleration(maxG1))
1162            - log10(primary->getGravityAcceleration(minG1)) << endl;
1163        results << "Primary volume:                            "
1164            << primary->getVolume() << endl;
1165        results << "Primary effective radius:                  "
1166            << pow(3.0 * primary->getVolume() / 4.0 / M_PI, 1.0/3.0) << endl;
1167        results << "Primary area:                              "
1168            << primary->getMeshArea() << endl;
1169        results << "Primary smallest area:                     "
1170            << primary->getArea(minA1) << endl;
1171        results << "Primary largest area:                      "
1172            << primary->getArea(maxA1) << endl;
1173        results << "Primary area range:                        "
1174            << primary->getArea(maxA1) - primary->getArea(minA1) << endl;
1175        results << "Primary sphericity:                        "
1176            << pow(4.0 * M_PI, 1.0/3.0) * pow(3.0 * primary->getVolume(),
1177    2.0/3.0)
1178            / primary->getMeshArea() << endl << endl;
1179
1180        size_t minT2 = 0, maxT2 = 0, minG2 = 0, maxG2 = 0, minA2 = 0, maxA2 = 0;
```

```
1181        double avgT2 = 0.0, L2 = 0.0;
1182        for (size_t i = 0; i < secondary->getTriangleCount(); i++)
1183        {
1184            avgT2 += secondary->getTemperature(i) * secondary->getArea(i);
1185            L2 += secondary->getArea(i) * pow4(secondary->getTemperature(i));
1186
1187            if(secondary->getTemperature(i) <
1188                secondary->getTemperature(minT2)) minT2 = i;
1189            if(secondary->getTemperature(i) >
1190                secondary->getTemperature(maxT2)) maxT2 = i;
1191
1192            if(secondary->getGravityAcceleration(i) <
1193                secondary->getGravityAcceleration(minG2)) minG2 = i;
1194            if(secondary->getGravityAcceleration(i) >
1195                secondary->getGravityAcceleration(maxG2)) maxG2 = i;
1196
1197            if(secondary->getArea(i) < secondary->getArea(minA2)) minA2 = i;
1198            if(secondary->getArea(i) > secondary->getArea(maxA2)) maxA2 = i;
1199        }
1200        avgT2 /= secondary->getMeshArea();
1201
1202        results << "Secondary asynchronicity:                    "
1203            << binary->getSecondary()->getAsynchronicity() << endl;
1204        results << "Secondary polar radius:                      "
1205            << binary->getSecondary()->getPolarRadius() << endl;
1206        results << "Secondary critical polar radius:             "
1207            << secondarySurface->getCriticalPolarRadius() << endl;
1208        results << "Secondary sychronous critical polar radius:  "
1209            << secondarySurface->getSychronousCriticalPolarRadius() << endl;
1210        results << "Secondary front radius:                      "
1211            << secondarySurface->getFrontRadius() << endl;
1212        results << "Secondary critical front radius:             "
1213            << secondarySurface->getCriticalFrontRadius() << endl;
1214        results << "Secondary synchronous critical front radius: "
1215            << secondarySurface->getSynchronousCriticalFrontRadius() << endl;
1216        results << "Secondary back radius:                       "
1217            << secondarySurface->getBackRadius() << endl;
1218        results << "Secondary side radius:                       "
1219            << secondarySurface->getSideRadius() << endl;
1220        results << "Secondary mean Eggleton radius:              "
1221            << secondarySurface->getMeanEggletonRadius() << endl;
1222        results << "Secondary Rp/Rf:                             "
1223            << binary->getSecondary()->getPolarRadius()
1224            / secondarySurface->getFrontRadius() << endl;
1225        results << "Secondary filling factor:                    "
1226            << secondarySurface->getFillingFactor() << endl;
1227        results << "Secondary potential:                         "
1228            << secondarySurface->getPotential() << endl;
1229        results << "Secondary critical potential:                "
1230            << secondarySurface->getCriticalPotential() << endl;
1231        results << "Secondary synchronous critical potential:    "
1232            << secondarySurface->getSynchronousCriticalPotential() << endl;
1233        results << "Secondary effective temperature:             "
1234            << binary->getSecondary()->getTemperature() << endl;
1235        results << "Secondary lowest temperature:                "
1236            << secondary->getTemperature(minT2) << endl;
1237        results << "Secondary highest temperature:               "
```

```
1238          << secondary->getTemperature(maxT2) << endl;
1239      results << "Secondary temperature range:                  "
1240          << secondary->getTemperature(maxT2)
1241          - secondary->getTemperature(minT2) << endl;
1242      results << "Secondary average temperature:               "
1243          << avgT2 << endl;
1244      results << "Secondary luminosity:                        "
1245          << 5.6704e-8 * pow2(6.955e8 * binary->getSemimajorAxis()) * L2
1246          / 3.839e26 << endl;
1247      results << "Secondary temperature from luminosity:       "
1248          << root4(L2 / secondary->getMeshArea()) << endl;
1249      results << "Secondary gravity darkening exponent:        "
1250          << binary->getSecondary()->getGravityDarkeningExponent() << endl;
1251      results << "Secondary albedo:                            "
1252          << binary->getSecondary()->getAlbedo() << endl;
1253      results << "Secondary mass:                              "
1254          << binary->getSecondary()->getMass() << endl;
1255      results << "Secondary effective gravity:                 "
1256          << log10(binary->getSecondary()->getEffectiveGravity()) << endl;
1257      results << "Secondary lowest gravity:                    "
1258          << log10(secondary->getGravityAcceleration(minG2)) << endl;
1259      results << "Secondary highest gravity:                   "
1260          << log10(secondary->getGravityAcceleration(maxG2)) << endl;
1261      results << "Secondary gravity range:                     "
1262          << log10(secondary->getGravityAcceleration(maxG2))
1263          - log10(secondary->getGravityAcceleration(minG2)) << endl;
1264      results << "Secondary volume:                            "
1265          << secondary->getVolume() << endl;
1266      results << "Secondary effective radius:                  "
1267          << pow(3.0 * secondary->getVolume() / 4.0 / M_PI, 1.0/3.0) << endl;
1268      results << "Secondary area:                              "
1269          << secondary->getMeshArea() << endl;
1270      results << "Secondary smallest area:                     "
1271          << secondary->getArea(minA2) << endl;
1272      results << "Secondary largest area:                      "
1273          << secondary->getArea(maxA2) << endl;
1274      results << "Secondary area range:                        "
1275          << secondary->getArea(maxA2) - secondary->getArea(minA2) << endl;
1276      results << "Secondary sphericity:                        "
1277          << pow(4.0 * M_PI, 1.0/3.0) * pow(3.0 * secondary->getVolume(),
1278  2.0/3.0)
1279          / secondary->getMeshArea() << endl << endl;
1280
1281
1282      size_t minT3 = 0, maxT3 = 0, minG3 = 0, maxG3 = 0, minA3 = 0, maxA3 = 0;
1283      double avgT3 = 0.0, L3 = 0.0;
1284      for (size_t i = 0; i < disc->getTriangleCount(); i++)
1285      {
1286          avgT3 += disc->getTemperature(i) * disc->getArea(i);
1287          L3 += disc->getArea(i) * pow4(disc->getTemperature(i));
1288
1289          if(disc->getTemperature(i) < disc->getTemperature(minT3)) minT3 = i;
1290          if(disc->getTemperature(i) > disc->getTemperature(maxT3)) maxT3 = i;
1291
1292          if(disc->getGravityAcceleration(i) <
1293              disc->getGravityAcceleration(minG3)) minG3 = i;
1294          if(disc->getGravityAcceleration(i) >
```

```
1295                disc->getGravityAcceleration(maxG3)) maxG3 = i;
1296
1297            if(disc->getArea(i) < disc->getArea(minA3)) minA3 = i;
1298            if(disc->getArea(i) > disc->getArea(maxA3)) maxA3 = i;
1299        }
1300        avgT2 /= disc->getMeshArea();
1301
1302        if (disc->getTriangleCount() > 0)
1303        {
1304            results << "Disc inner temperature:                 "
1305                << binary->getDisc()->getInnerTemperature() << endl;
1306            results << "Disc outer temperature:                 "
1307                << binary->getDisc()->getOuterTemperature() << endl;
1308            results << "Disc lowest temperature:                "
1309                << disc->getTemperature(minT3) << endl;
1310            results << "Disc highest temperature:               "
1311                << disc->getTemperature(maxT3) << endl;
1312            results << "Disc temperature range:                 "
1313                << disc->getTemperature(maxT3)
1314                - disc->getTemperature(minT3) << endl;
1315            results << "Disc average temperature:               "
1316                << avgT3 << endl;
1317            results << "Disc lowest gravity:                    "
1318                << log10(disc->getGravityAcceleration(minG3)) << endl;
1319            results << "Disc highest gravity:                   "
1320                << log10(disc->getGravityAcceleration(maxG3)) << endl;
1321            results << "Disk gravity range:                     "
1322                << log10(disc->getGravityAcceleration(maxG3))
1323                - log10(disc->getGravityAcceleration(minG3)) << endl;
1324            results << "Disc luminosity:                        "
1325                << 5.6704e-8 * pow2(6.955e8 * binary->getSemimajorAxis()) * L3
1326                / 3.839e26 << endl;
1327            results << "Disc temperature from luminosity:       "
1328                << root4(L3 / disc->getMeshArea()) << endl;
1329            results << "Disc volume:                            "
1330                << disc->getVolume() << endl;
1331            results << "Disc area:                              "
1332                << disc->getMeshArea() << endl;
1333            results << "Disc smallest area:                     "
1334                << disc->getArea(minA3) << endl;
1335            results << "Disc largest area:                      "
1336                << disc->getArea(maxA3) << endl;
1337            results << "Disc area range:                        "
1338                << disc->getArea(maxA3) - disc->getArea(minA3) << endl;
1339        }
1340
1341        results.close();
1342    }
1343
1344    void Model::writeRV(vector<double>& rv, wstring sufix)
1345    {
1346        size_t pointCount = phases.size() - 1;
1347
1348        ofstream file(prefix + sufix);
1349
1350        for (size_t i = 0; i < pointCount; ++i)
1351            file << setiosflags( ios::fixed ) << setprecision(10) << phases[i]
```

```
    << " " << rv[i]  << endl;

    file.close();
} }
```

8.7. Model akrecionog diska

Disc.h

```
#pragma once

#include "Palette.h"
#include "CrossSection.h"
#include "Calc.h"
#include <string>
#include "DiscSpot.h"

namespace Infinity {
class Disc
{
public:
    enum Shape { Conical, Toroidal };

    Disc();
    Disc(Shape shape, CrossSection cs, int segments, double innerRadius,
        double outerRadius, double innerTemperature, double outerTemperature,
        double exponent, int distributionType, std::wstring paletteFile);
    Disc(const Disc& source);
    Disc& operator=(const Disc& right);
    ~Disc(void);

    int getSegments();
    Palette* getPalette();
    CrossSection* getCrossSection();

    double getTemperatureAt(float radius);
    double getInnerTemperature();
    double getOuterTemperature();
    double getInnerRadius();
    double getOuterRadius();
    Disc::Shape getShape();

    size_t getSpotCount();
    DiscSpot* getSpot(size_t index);
    void addSpot(double phi, double size, double temperatureRatio);
private:
    Shape shape;
    CrossSection crossSection;
    Palette palette;
    double innerRadius, outerRadius;
    double innerTemperature, outerTemperature;
    double exponent;
    int distributionType;

    int segments;
```

```
47
48      std::vector<DiscSpot> spots;
49  };
50
51  inline int Disc::getSegments() { return segments; }
52  inline Palette* Disc::getPalette() { return &palette; }
53  inline CrossSection* Disc::getCrossSection() { return &crossSection; }
54  inline double Disc::getInnerTemperature() { return innerTemperature; }
55  inline double Disc::getOuterTemperature() { return outerTemperature; }
56  inline double Disc::getInnerRadius() { return innerRadius; }
57  inline double Disc::getOuterRadius() { return outerRadius; }
58  inline Disc::Shape Disc::getShape() { return shape; }
59
60  inline double Disc::getTemperatureAt(float radius)
61  {
62      double result = 0;
63      switch (distributionType)
64      {
65      case DISC_TEMP_DIST_ALPHA:
66          result = outerTemperature * pow(outerRadius / radius, exponent)
67          break;
68      case DISC_TEMP_DIST_DJURASEVIC:
69          result = outerTemperature + (innerTemperature - outerTemperature)
70              * (1 - pow((radius - innerRadius) / (outerRadius - innerRadius),
71              exponent));
72          break;
73      default:
74      case DISC_TEMP_DIST_ZOLA:
75          result = outerTemperature + (innerTemperature - outerTemperature)
76              * pow(1 - (radius - innerRadius) / (outerRadius - innerRadius),
77              exponent);
78          break;
79      }
80
81      return result;
82  }
83
84  inline size_t Disc::getSpotCount()
85  { return spots.size(); }
86
87  inline DiscSpot* Disc::getSpot(size_t index)
88  { return &spots[index]; }
89
90  inline void Disc::addSpot(double phi, double size, double temperatureRatio)
91  { spots.push_back(DiscSpot(phi, size, temperatureRatio)); } }
```

Disc.cpp

```
1   #include "stdafx.h"
2   #include "Disc.h"
3
4   using namespace std;
5
6   namespace Infinity {
7   Disc::Disc() :
8       palette(L"red"),
```

```
    crossSection(0) { }

Disc::Disc(Shape shape, CrossSection cs, int segments, double innerRadius,
    double outerRadius, double innerTemperature, double outerTemperature,
    double exponent, int distributionType, std::wstring paletteFile) :
    shape(shape),
    crossSection(cs),
    segments(segments),
    innerRadius(innerRadius),
    outerRadius(outerRadius),
    innerTemperature(innerTemperature),
    outerTemperature(outerTemperature),
    exponent(exponent),
    distributionType(distributionType),
    palette(paletteFile)
{ }

Disc::Disc(const Disc& source) :
    shape(source.shape),
    crossSection(source.crossSection),
    segments(source.segments),
    innerRadius(source.innerRadius),
    outerRadius(source.outerRadius),
    innerTemperature(source.innerTemperature),
    outerTemperature(source.outerTemperature),
    exponent(source.exponent),
    distributionType(source.distributionType),
    palette(source.palette),
    spots(source.spots)
{ }

Disc& Disc::operator=(const Disc& source)
{
    shape = source.shape;
    crossSection = source.crossSection;
    segments = source.segments;
    innerRadius = source.innerRadius;
    outerRadius = source.outerRadius;
    innerTemperature = source.innerTemperature;
    outerTemperature = source.outerTemperature;
    exponent = source.exponent;
    distributionType = source.distributionType;
    palette = source.palette;
    spots = source.spots;

    return *this;
}

Disc::~Disc(void)
{ } }
```

DiscEffect.h

```
#pragma once

#include <cmath>
```

```
#include "Calc.h"
#include "Mesh3.h"
#include "Binary.h"

namespace Infinity {
class DiscEffect
{
public:
    DiscEffect(Binary* binary, Disc* disc, Mesh3* mesh);
    ~DiscEffect(void);

    virtual void run(double time = 0) = 0;
protected:
    Binary* binary;
    Mesh3* mesh;
    Disc* disc;
}; }
```

DiscEffect.cpp

```
#include "stdafx.h"
#include "DiscEffect.h"

namespace Infinity {
DiscEffect::DiscEffect(Binary* binary, Disc* disc, Mesh3* mesh)
    : binary(binary), disc(disc), mesh(mesh)
{ }

DiscEffect::~DiscEffect(void)
{ } }
```

DiscGravity.h

```
#pragma once
#include "DiscEffect.h"

namespace Infinity {
class DiscGravity : public DiscEffect
{
public:
    DiscGravity(Binary* binary, Disc* disc, Mesh3* mesh);
    ~DiscGravity(void);

    void run(double time = 0);
}; }
```

DiscGravity.cpp

```
#include "stdafx.h"
#include "DiscGravity.h"

namespace Infinity {
DiscGravity::DiscGravity(Binary* binary, Disc* disc, Mesh3* mesh)
    : DiscEffect(binary, disc, mesh) { }

```

```
8   DiscGravity::~DiscGravity(void) { }
9
10  void DiscGravity::run(double time)
11  {
12      double gravityConstant = 6.67428E-8;
13      double solarRadius = 6.955E10;
14
15      double scalingFactor = gravityConstant * binary->getPrimary()->getMass()
16          / pow2(binary->getSemimajorAxis() * solarRadius);
17
18      size_t triangleCount = mesh->getTriangleCount();
19      for (size_t i = 0; i < triangleCount; ++i)
20          mesh->setGravityAcceleration(i, scalingFactor
21          / mesh->getCentroid(i)->LengthSquared());
22  } }
```

DiscSpot.h

```
1   #pragma once
2
3   namespace Infinity {
4   class DiscSpot
5   {
6   public:
7       DiscSpot(double phi, double size, double temperautreRatio);
8       ~DiscSpot();
9
10      double getPhi();
11      double getSize();
12      double getTemperatureRatio();
13
14  private:
15      double phi, size, temperatureRatio;
16  };
17
18  inline double DiscSpot::getPhi() { return phi; }
19  inline double DiscSpot::getSize() { return size; }
20  inline double DiscSpot::getTemperatureRatio() { return temperatureRatio; }
```

DiscSpot.cpp

```
1   #include "stdafx.h"
2   #include "DiscSpot.h"
3
4   namespace Infinity {
5   DiscSpot::DiscSpot(double phi, double size, double temperatureRatio):
6       phi(phi),
7       size(size),
8       temperatureRatio(temperatureRatio)
9   { }
10
11  DiscSpot::~DiscSpot() { } }
```

DiscSpots.h

```
1   #pragma once
2
3   #include "DiscEffect.h"
4   #include <cmath>
5   #include "Calc.h"
6   #include "Mesh3.h"
7   #include "Binary.h"
8   #include "Spot.h"
9   #include <vector>
10
11  namespace Infinity {
12  class DiscSpots : public DiscEffect
13  {
14  public:
15      DiscSpots(Binary* binary, Disc* disc, Mesh3* mesh);
16      ~DiscSpots();
17
18      void run(double time = 0);
19  private:
20      void getPolar(size_t index, float* rho, float* phi);
21  };
22
23  inline void DiscSpots::getPolar(size_t index, float* rho, float* phi)
24  {
25      float x = mesh->getPoint(index)->x;
26      float y = mesh->getPoint(index)->y;
27      float z = mesh->getPoint(index)->z;
28
29      *phi = atan2(y, x);
30      *phi = (*phi >= 0) ? *phi : *phi + TWO_PI_F;
31
32      *rho = sqrt(pow2(x) + pow2(y));
```

DiscSpots.cpp

```
1   #include "stdafx.h"
2   #include "DiscSpots.h"
3
4   using namespace std;
5   using namespace DirectX::SimpleMath;
6
7   namespace Infinity {
8   DiscSpots::DiscSpots(Binary* binary, Disc* disc, Mesh3* mesh)
9       : DiscEffect(binary, disc, mesh)
10  { }
11
12  DiscSpots::~DiscSpots(void)
13  { }
14
15  void DiscSpots::run(double time)
16  {
17      size_t spotCount = disc->getSpotCount();
18      size_t triangleCount =  mesh->getTriangleCount();
19
```

```
20      for(size_t j = 0; j < spotCount; j++)
21      {
22          double spotPhi = disc->getSpot(j)->getPhi();
23          double spotSize = disc->getSpot(j)->getSize();
24          double temperatureRatio = disc->getSpot(j)->getTemperatureRatio();
25
26          for (size_t i = 0; i < triangleCount; i++)
27          {
28              if (disc->getShape() == Disc::Shape::Conical)
29              {
30                  float rho1 = 0, rho2 = 0, rho3 = 0,
31                      phi1 = 0, phi2 = 0, phi3 = 0;
32
33                  getPolar(3*i,   &rho1, &phi1);
34                  getPolar(3*i+1, &rho2, &phi2);
35                  getPolar(3*i+2, &rho3, &phi3);
36
37                  float minPhi = min(min(phi1, phi2), phi3);
38                  float maxPhi = max(max(phi1, phi2), phi3);
39
40                  float radius = static_cast<float>
41                   (disc->getOuterRadius());
42
43                  bool onEdge = isEqual(rho1, radius)
44                  && isEqual(rho2, radius) && isEqual(rho3, radius);
45
46                  if (onEdge && minPhi > spotPhi - spotSize / 2.0
47                      && maxPhi < spotPhi + spotSize / 2.0)
48                      mesh->setTemperature(i,
49                      mesh->getTemperature(i) * temperatureRatio);
50              }
51              else //Disc::Shape::Toroidal
52              {
53                  double x = mesh->getCentroid(i)->x;
54                  double y = mesh->getCentroid(i)->y;
55                  double z = mesh->getCentroid(i)->z;
56
57                  double phi = atan2(y, x);
58                  phi = (phi >= 0) ? phi : phi + 2 * M_PI;
59                  double theta = acos(z / sqrt(x*x + y*y + z*z));
60
61                  double rho = sqrt(pow2(x) + pow2(y));
62
63                  double cosL = sin(theta) * cos(phi - spotPhi);
64
65                  if (cos(spotSize) < cosL &&
66                      rho > 0.5 * (disc->getInnerRadius()
67                      + disc->getOuterRadius()))
68                      mesh->setTemperature(i,
69                      mesh->getTemperature(i) * temperatureRatio);
70              }
71          }
72      }
73  } }
```

DiscTemperature.h

```
#pragma once

#include "DiscEffect.h"

namespace Infinity {
class DiscTemperature : public DiscEffect
{
public:
    DiscTemperature(Binary* binary, Disc* disc, Mesh3* mesh);
    ~DiscTemperature();

    void run(double time = 0);
}; }
```

DiscTemperature.cpp

```
#include "stdafx.h"
#include "DiscTemperature.h"

namespace Infinity {
DiscTemperature::DiscTemperature(Binary* binary, Disc* disc, Mesh3* mesh) :
    DiscEffect(binary, disc, mesh)
{ }

DiscTemperature::~DiscTemperature(void)
{ }

void DiscTemperature::run(double time)
{
    size_t triangleCount =  mesh->getTriangleCount();
    for (size_t i = 0; i < triangleCount; i++)
    {
        float x = mesh->getCentroid(i)->x;
        float y = mesh->getCentroid(i)->y;

        float r = sqrt(pow2(x) + pow2(y));

        double temperature = disc->getTemperatureAt(r);
        mesh->setTemperature(i, temperature);
    }
} }
```

SystemEffect.h

```
#pragma once

#include "Mesh3.h"
#include "Binary.h"

namespace Infinity {
class SystemEffect
{
public:
```

```
10      SystemEffect(Binary* binary, Mesh3* primary, Mesh3* secondary, Mesh3*
11  disc);
12      ~SystemEffect(void);
13
14      virtual void run() = 0;
15  protected:
16      Binary* binary;
17      Mesh3* primary;
18      Mesh3* secondary;
19      Mesh3* disc;
20  }; }
```

SystemEffect.cpp

```
1   #include "stdafx.h"
2   #include "SystemEffect.h"
3
4   namespace Infinity {
5   SystemEffect::SystemEffect(Binary* bin, Mesh3* prim, Mesh3* sec, Mesh3* d)
6       : binary(bin), primary(prim), secondary(sec), disc(d)
7   { }
8
9   SystemEffect::~SystemEffect(void)
10  { } }
```

8.8. Model refleksije

DjurasevicReflection.h

```
1   #pragma once
2
3   #include "SystemEffect.h"
4   #include "SimpleMath.h"
5
6   namespace Infinity {
7   class DjurasevicReflection : public SystemEffect
8   {
9   public:
10      DjurasevicReflection(Binary* binary, Mesh3* primary,
11          Mesh3* secondary, Mesh3* disc);
12      ~DjurasevicReflection(void);
13
14      void run();
15  }; }
```

DjurasevicReflection.cpp

```
1   #include "stdafx.h"
2   #include "DjurasevicReflection.h"
3
4   using namespace DirectX::SimpleMath;
5
6   namespace Infinity {
7   DjurasevicReflection::DjurasevicReflection(Binary* bin, Mesh3* prim,
```

```
8       Mesh3* sec, Mesh3* disc) : SystemEffect(bin, prim, sec, disc)
9   { }
10
11  DjurasevicReflection::~DjurasevicReflection(void)
12  { }
13
14  void DjurasevicReflection::run()
15  {
16      //Primary
17      for (size_t i = 0; i < primary->getTriangleCount(); i++)
18      {
19          Vector3 rho(1.0f - primary->getCentroid(i)->x,
20              -primary->getCentroid(i)->y, -primary->getCentroid(i)->z);
21          float rhoNormSquared = rho.LengthSquared();
22          rho.Normalize();
23
24          double cosAlpha = rho.Dot(*primary->getNormal(i));
25          if (cosAlpha > 0)
26          {
27              double correction = sqrt(sqrt(
28                  1.0 + binary->getPrimary()->getAlbedo() * cosAlpha
29                  * (1.0 - sqrt(1.0 - pow2(secondary->getInnerRadius())
30                  / rhoNormSquared))
31                  * pow4(binary->getSecondary()->getTemperature()
32                  / binary->getPrimary()->getTemperature())));
33              primary->setTemperature(i,
34                  primary->getTemperature(i) * correction);
35          }
36      }
37
38      //Secondary
39      for (size_t i = 0; i < secondary->getTriangleCount(); i++)
40      {
41          Vector3 rho(1.0f - secondary->getCentroid(i)->x,
42              -secondary->getCentroid(i)->y, -secondary->getCentroid(i)->z);
43          float rhoNormSquared = rho.LengthSquared();
44          rho.Normalize();
45
46          double cosAlpha = rho.Dot(*secondary->getNormal(i));
47          if (cosAlpha > 0)
48          {
49              double correction = sqrt(sqrt(
50                  1.0 + binary->getSecondary()->getAlbedo() * cosAlpha
51                  * (1.0 - sqrt( 1.0 - pow2(primary->getInnerRadius())
52                  / rhoNormSquared))
53                  * pow4(binary->getPrimary()->getTemperature()
54                  / binary->getSecondary()->getTemperature())));
55              secondary->setTemperature(i,
56                  secondary->getTemperature(i) * correction);
57          }
58      }
59  } }
```

KhruzinaReflection.h

```
#pragma once
#include "SystemEffect.h"
#include "SimpleMath.h"

namespace Infinity {
class KhruzinaReflection :
    public SystemEffect
{
public:
    KhruzinaReflection(Binary* binary, Mesh3* primary,
        Mesh3* secondary, Mesh3* disc);
    ~KhruzinaReflection(void);

    void run();

private:
    double primLuminosity;
    double secLuminosity;
}; }
```

KhruzinaReflection.cpp

```
#include "stdafx.h"
#include "KhruzinaReflection.h"

using namespace DirectX::SimpleMath;

namespace Infinity {
KhruzinaReflection::KhruzinaReflection(Binary* bin, Mesh3* prim, Mesh3* sec,
    Mesh3* disc) : SystemEffect(bin, prim, sec, disc)
{
    //Primary
    primLuminosity = 0.0;

    for (size_t i = 0; i < primary->getTriangleCount(); i++)
        primLuminosity += primary->getArea(i)
        * pow4(primary->getTemperature(i));

    //Secondary
    secLuminosity = 0.0;

    for (size_t i = 0; i < secondary->getTriangleCount(); i++)
        secLuminosity += secondary->getArea(i)
        * pow4(secondary->getTemperature(i));
}

KhruzinaReflection::~KhruzinaReflection(void)
{ }

void KhruzinaReflection::run()
{
    //Primary
    for (size_t i = 0; i < primary->getTriangleCount(); i++)
    {
```

```
33          Vector3 rho(1.0f - primary->getCentroid(i)->x,
34              -primary->getCentroid(i)->y, -primary->getCentroid(i)->z);
35          rho.Normalize();
36
37          double cosAlpha = rho.Dot(*primary->getNormal(i));
38          if (cosAlpha > 0)
39          {
40              double newTemp = sqrt(sqrt(pow4(primary->getTemperature(i))
41                  + binary->getPrimary()->getAlbedo() * cosAlpha
42                  * secLuminosity / 4.0 / M_PI));
43              primary->setTemperature(i, newTemp);
44          }
45      }
46
47      //Secondary
48      for (size_t i = 0; i < secondary->getTriangleCount(); i++)
49      {
50          Vector3 rho(1.0f - secondary->getCentroid(i)->x,
51              -secondary->getCentroid(i)->y, -secondary->getCentroid(i)->z);
52          rho.Normalize();
53
54          double cosAlpha = rho.Dot(*secondary->getNormal(i));
55          if (cosAlpha > 0)
56          {
57              double newTemp = sqrt(sqrt(pow4(secondary->getTemperature(i))
58                  + binary->getSecondary()->getAlbedo() * cosAlpha
59                  * primLuminosity / 4.0 / M_PI));
60              secondary->setTemperature(i, newTemp);
61          }
62      }
63  } }
```

NoReflection.h

```
1   #pragma once
2   #include "SystemEffect.h"
3
4   namespace Infinity {
5   class NoReflection : public SystemEffect
6   {
7   public:
8       NoReflection(Binary* binary, Mesh3* primary, Mesh3* secondary, Mesh3*
9   disc);
10      ~NoReflection(void);
11
12      void run();
13  }; }
```

NoReflection.cpp

```
1   #include "stdafx.h"
2   #include "NoReflection.h"
3
```

```
4   namespace Infinity {
5   NoReflection::NoReflection(Binary* bin, Mesh3* prim, Mesh3* sec, Mesh3* disc)
6       : SystemEffect(bin, prim, sec, disc)
7   { }
8
9   NoReflection::~NoReflection(void)
10  { }
11
12  void NoReflection::run()
13  { } }
```

8.9. Detekcija vidljivosti

GridVisibility.h

```
1   #pragma once
2
3   #include "Mesh3.h"
4   #include "VisibilityDetection.h"
5   #include <vector>
6   #include <algorithm>
7
8   #include "Triangle2.h"
9
10  namespace Infinity {
11  class GridVisibility : public VisibilityDetection
12  {
13  public:
14      GridVisibility(Mesh3* primary, Mesh3* secondary, Mesh3* disk, int
15  adaptive,
16          int triangleIntersection, bool convexMesh = true);
17      ~GridVisibility();
18
19      void run();
20      static bool distanceSorter(const Triangle2& left, const Triangle2&
21  right);
22  private:
23      bool hasDisk;
24      void optimizedCheck();
25      void fullCheck();
26
27      bool checkVisibility(const std::vector<Triangle2*>* grid, size_t row,
28          size_t column, size_t columnCount, Triangle2* triangle,
29          int method) const;
30      void projectMesh(Mesh3* mesh, std::vector<Triangle2>& faces,
31          float& maxRadius, float obsPosition = 5.0);
32  };
```

GridVisibility.cpp

```
1   #include "stdafx.h"
2   #include "GridVisibility.h"
3
4   using namespace std;
5   using namespace DirectX::SimpleMath;
```

```
6
7   namespace Infinity {
8   GridVisibility::GridVisibility(Mesh3* primary, Mesh3* secondary, Mesh3* disc,
9       int adaptive, int triangleIntersection, bool convexMesh)
10      : VisibilityDetection(primary, secondary, disc, adaptive,
11      triangleIntersection, true)
12  { hasDisk = (disc->getTriangleCount() != 0); }
13
14  GridVisibility::~GridVisibility() { }
15
16  void GridVisibility::run()
17  {
18      if (!hasDisk && convexMesh) optimizedCheck();
19      else fullCheck();
20  }
21
22  void GridVisibility::optimizedCheck()
23  {
24      //Determine the distance of the components
25      float obsPosition = 5.0;
26      double distanceSquaredPrimary =
27          pow2(primary->getCenter()->x - obsPosition) +
28          pow2(primary->getCenter()->y) +
29          pow2(primary->getCenter()->z);
30
31      double distanceSquaredSecondary =
32          pow2(secondary->getCenter()->x - obsPosition) +
33          pow2(secondary->getCenter()->y) +
34          pow2(secondary->getCenter()->z);
35
36      Mesh3* nearMesh;
37      Mesh3* farMesh;
38
39      if (distanceSquaredPrimary <= distanceSquaredSecondary)
40      {
41          nearMesh = primary;
42          farMesh = secondary;
43      }
44      else
45      {
46          nearMesh = secondary;
47          farMesh = primary;
48      }
49
50      //Check wether there is a posibility for the eclipse.
51      bool eclipsePossible = true;
52
53      double separation =
54          pow2(nearMesh->getCenter()->y - farMesh->getCenter()->y) +
55          pow2(nearMesh->getCenter()->z - farMesh->getCenter()->z);
56      if (separation > pow2(nearMesh->getOuterRadius()
57          + farMesh->getOuterRadius()))
58          eclipsePossible = false;
59
60      //Adaptive
61      if (eclipsePossible && adaptive > 0)
62          farMesh->adaptive(adaptive, nearMesh->getCenter(),
```

```
63          0.9 * nearMesh->getInnerRadius(), 1.1 * nearMesh->getOuterRadius());
64
65      //Project & colect triangles
66      vector<Triangle2> nearFaces, farFaces;
67      nearFaces.reserve(nearMesh->getTriangleCount());
68      farFaces.reserve(farMesh->getTriangleCount());
69
70      float nearRadius = -1;
71      float farRadius = -1;
72      projectMesh(nearMesh, nearFaces, nearRadius, obsPosition);
73      projectMesh(farMesh, farFaces, farRadius, obsPosition);
74
75      //If there is no chance for eclipse, we are done.
76      if (!eclipsePossible)
77          return;
78
79      //Try to guess the optimal size of the cell
80      double cellSizeGuess = 1.4 * max(nearRadius, farRadius);
81
82      Vector2 primaryCenter(primary->getCenter()->y, primary->getCenter()->z);
83      Vector2 secondaryCenter(secondary->getCenter()->y,
84          secondary->getCenter()->z);
85
86      double xMax = max(primaryCenter.x + primary->getOuterRadius(),
87          secondaryCenter.x + secondary->getOuterRadius()) + cellSizeGuess;
88      double xMin = min(primaryCenter.x - primary->getOuterRadius(),
89          secondaryCenter.x - secondary->getOuterRadius()) - cellSizeGuess;
90      double xSize = xMax - xMin;
91
92      double yMax = max(primaryCenter.y + primary->getOuterRadius(),
93          secondaryCenter.y + secondary->getOuterRadius()) + cellSizeGuess;
94      double yMin = min(primaryCenter.y - primary->getOuterRadius(),
95          secondaryCenter.y - secondary->getOuterRadius()) - cellSizeGuess;
96      double ySize = yMax - yMin;
97
98
99      size_t rowCount = static_cast<size_t>(ceil(ySize / cellSizeGuess));
100     size_t columnCount = static_cast<size_t>(ceil(xSize / cellSizeGuess));
101
102     double cellSizeX = xSize / columnCount;
103     double cellSizeY = ySize / rowCount;
104
105     size_t cellCount = rowCount * columnCount;
106     vector<Triangle2*>* grid = new vector<Triangle2*>[cellCount];
107
108     //Add nearMesh to the grid
109     Circle nearBoundingCircle(nearMesh->getCenter()->y,
110         nearMesh->getCenter()->z, nearMesh->getOuterRadius() + 3 *
111 nearRadius);
112     Circle farBoundingCircle(farMesh->getCenter()->y,
113         farMesh->getCenter()->z, farMesh->getOuterRadius() + 3 * farRadius);
114
115     size_t row, column;
116
117     for (size_t i = 0; i < nearFaces.size(); i++)
118     {
119         if (Circle::Intersects(&farBoundingCircle,
```

```
            nearFaces[i].getCircumCircle()))
        {
            row = static_cast<size_t>
                ((nearFaces[i].getCentroid()->y - yMin) / cellSizeY);
            column = static_cast<size_t>
                ((nearFaces[i].getCentroid()->x - xMin) / cellSizeX);

            grid[row * columnCount + column].push_back(&nearFaces[i]);
        }
    }

    //Check the visibility of the farMesh
    sort(farFaces.begin(), farFaces.end(), GridVisibility::distanceSorter);

    for (size_t i = 0; i < farFaces.size(); i++)
    {
        if (Circle::Intersects(&nearBoundingCircle,
            farFaces[i].getCircumCircle()))
        {
            row = static_cast<size_t>
                ((farFaces[i].getCentroid()->y - yMin) / cellSizeY);
            column = static_cast<size_t>
                ((farFaces[i].getCentroid()->x - xMin) / cellSizeX);

            //Check for the current cell
            bool visible = checkVisibility(grid, row, column, columnCount,
                &farFaces[i], triangleIntersection);

            //Check for the surrounding cells if we use the full
            //triangle-triangle intersection
            if (visible)
            visible = (visible && checkVisibility(grid, row, column + 1,
                columnCount, &farFaces[i], triangleIntersection));
            else
            { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
            columnCount + column].push_back(&farFaces[i]); continue; }
             if (visible)
                visible = (visible && checkVisibility(grid,row,column-1,
                columnCount, &farFaces[i], triangleIntersection));
            else
            { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
            columnCount + column].push_back(&farFaces[i]); continue; }
             if (visible)
            visible = (visible && checkVisibility(grid,row + 1,column + 1,
                columnCount, &farFaces[i], triangleIntersection));
            else
            { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
            columnCount + column].push_back(&farFaces[i]); continue; }
             if (visible)
            visible = (visible && checkVisibility(grid, row + 1, column,
                columnCount, &farFaces[i], triangleIntersection));
            else
            { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
            columnCount + column].push_back(&farFaces[i]); continue; }
             if (visible)
            visible = (visible && checkVisibility(grid,row + 1,column - 1,
                columnCount, &farFaces[i], triangleIntersection));
```

```
177                 else
178                 { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
179                 columnCount + column].push_back(&farFaces[i]); continue; }
180                  if (visible)
181                 visible = (visible && checkVisibility(grid,row - 1,column + 1,
182                     columnCount, &farFaces[i], triangleIntersection));
183                 else
184                 { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
185                 columnCount + column].push_back(&farFaces[i]); continue; }
186                  if (visible)
187                 visible = (visible && checkVisibility(grid, row - 1, column,
188                     columnCount, &farFaces[i], triangleIntersection));
189                 else
190                 { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
191                 columnCount + column].push_back(&farFaces[i]); continue; }
192
193                 if (visible)
194                 visible = (visible && checkVisibility(grid,row - 1,column - 1,
195                     columnCount, &farFaces[i], triangleIntersection));
196                 else
197                 { farMesh->setVisibility(farFaces[i].getIndex(), 0.0); grid[row *
198                 columnCount + column].push_back(&farFaces[i]); continue; }
199                  if (!visible)
200                 farMesh->setVisibility(farFaces[i].getIndex(), 0.0);
201
202                 grid[row * columnCount + column].push_back(&farFaces[i]);
203             }
204         }
205         delete []grid;
206     }
207
208     void GridVisibility::fullCheck()
209     {
210         //Determine the distance of the components
211         float obsPosition = 5.0;
212
213         //Project & colect triangles
214         vector<Triangle2> faces;
215         faces.reserve(primary->getTriangleCount() + secondary->getTriangleCount()
216             + (hasDisk ? disk->getTriangleCount() : 0));
217
218         float primaryRadius = -1;
219         float secondaryRadius = -1;
220         float diskRadius = -1;
221         projectMesh(primary, faces, primaryRadius, obsPosition);
222         projectMesh(secondary, faces, secondaryRadius, obsPosition);
223         if (hasDisk)
224             projectMesh(disk, faces, diskRadius, obsPosition);
225
226         //Check wether there is a posibility for the eclipse.
227         double separation = pow2(
228             primary->getCenter()->y - secondary->getCenter()->y)
229             + pow2(primary->getCenter()->z - secondary->getCenter()->z);
230
231         double primaryOuterRadius = hasDisk ? disk->getOuterRadius()
232             : primary->getOuterRadius();
233         double secondaryOuterRadius = secondary->getOuterRadius();
```

```
234
235         if (separation > pow2(primaryOuterRadius + secondaryOuterRadius))
236             return;
237
238         //Try to guess the optimal size of the cell
239         double cellSizeGuess = 1.4 * max(max(primaryRadius, secondaryRadius),
240             diskRadius);
241
242         Vector2 primaryCenter(primary->getCenter()->y,
243             primary->getCenter()->z);
244         Vector2 secondaryCenter(secondary->getCenter()->y,
245             secondary->getCenter()->z);
246
247         double xMax = max(primaryCenter.x + primaryOuterRadius,
248             secondaryCenter.x + secondaryOuterRadius) + cellSizeGuess;
249         double xMin = min(primaryCenter.x - primaryOuterRadius,
250             secondaryCenter.x - secondaryOuterRadius) - cellSizeGuess;
251         double xSize = xMax - xMin;
252
253         double yMax = max(primaryCenter.y + primaryOuterRadius,
254             secondaryCenter.y + secondaryOuterRadius) + cellSizeGuess;
255         double yMin = min(primaryCenter.y - primaryOuterRadius,
256             secondaryCenter.y - secondaryOuterRadius) - cellSizeGuess;
257         double ySize = yMax - yMin;
258
259             size_t rowCount = static_cast<size_t>(ceil(ySize / cellSizeGuess));
260             size_t columnCount = static_cast<size_t>(ceil(xSize /
261     cellSizeGuess));
262
263         double cellSizeX = xSize / columnCount;
264         double cellSizeY = ySize / rowCount;
265
266         size_t cellCount = rowCount * columnCount;
267         vector<Triangle2*>* grid = new vector<Triangle2*>[cellCount];
268
269         //Collect the triangles
270         sort(faces.begin(), faces.end(), GridVisibility::distanceSorter);
271
272         Circle primaryBoundingCircle(primary->getCenter()->y,
273             primary->getCenter()->z, static_cast<float>(primaryOuterRadius) + 3
274             * max(primaryRadius, diskRadius));
275         Circle secondaryBoundingCircle(secondary->getCenter()->y,
276             secondary->getCenter()->z, static_cast<float>(secondaryOuterRadius)
277             + 3 * secondaryRadius);
278
279         size_t row, column;
280
281         for (size_t i = 0; i < faces.size(); i++)
282         {
283             if (Circle::Intersects(&primaryBoundingCircle,
284                 faces[i].getCircumCircle()))
285             {
286                 row = static_cast<size_t>
287                     ((faces[i].getCentroid()->y - yMin) / cellSizeY);
288                 column = static_cast<size_t>
289                     ((faces[i].getCentroid()->x - xMin) / cellSizeX);
290
```

```
291             //Check for the current cell
292             bool visible = checkVisibility(grid, row, column, columnCount,
293                 &faces[i], triangleIntersection);
294
295             //Check for the surrounding cells if we use the full
296             //triangle-triangle intersection
297             if (visible)
298             visible = (visible && checkVisibility(grid, row, column + 1,
299                 columnCount, &faces[i], triangleIntersection));
300             else
301             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
302             grid[row * columnCount + column].push_back(&faces[i]); continue;
303 }
304              if (visible)
305                 visible = (visible && checkVisibility(grid,row,column-1,
306                 columnCount, &faces[i], triangleIntersection));
307             else
308             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
309             grid[row * columnCount + column].push_back(&faces[i]); continue;
310 }
311              if (visible)
312             visible = (visible && checkVisibility(grid,row + 1,column + 1,
313                 columnCount, &faces[i], triangleIntersection));
314             else
315             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
316             grid[row * columnCount + column].push_back(&faces[i]); continue;
317 }
318              if (visible)
319             visible = (visible && checkVisibility(grid,row + 1,column,
320                 columnCount, &faces[i], triangleIntersection));
321             else
322             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
323             grid[row * columnCount + column].push_back(&faces[i]); continue;
324 }
325              if (visible)
326             visible = (visible && checkVisibility(grid,row + 1,column - 1,
327                 columnCount, &faces[i], triangleIntersection));
328             else
329             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
330             grid[row * columnCount + column].push_back(&faces[i]); continue;
331 }
332              if (visible)
333             visible = (visible && checkVisibility(grid,row-1,column + 1,
334                 columnCount, &faces[i], triangleIntersection));
335             else
336             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
337             grid[row * columnCount + column].push_back(&faces[i]); continue;
338 }
339              if (visible)
340             visible = (visible && checkVisibility(grid, row - 1, column,
341                 columnCount, &faces[i], triangleIntersection));
342             else
343             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
344             grid[row * columnCount + column].push_back(&faces[i]); continue;
345 }
346              if (visible)
347             visible = (visible && checkVisibility(grid,row - 1,column - 1,
```

```
348                 columnCount, &faces[i], triangleIntersection));
349             else
350             { faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
351             grid[row * columnCount + column].push_back(&faces[i]); continue;
352 }
353
354
355             if (!visible)
356             faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
357              grid[row * columnCount + column].push_back(&faces[i]);
358         }
359     }
360
361     delete []grid;
362 }
363
364 bool GridVisibility::checkVisibility(const vector<Triangle2*>* grid, size_t
365 row,
366     size_t column, size_t columnCount, Triangle2* triangle, int method) const
367 {
368     if (grid[row * columnCount + column].size() == 0)
369     return true;
370
371     bool visible = true;
372
373     for (size_t k = 0; k < grid[row * columnCount + column].size(); k++)
374     {
375     if (triangle->intersects(grid[row * columnCount + column][k], method))
376         {
377             visible = false;
378             break;
379         }
380     }
381     return visible;
382 }
383
384 void GridVisibility::projectMesh(Mesh3* mesh, std::vector<Triangle2>& faces,
385     float& maxRadius, float obsPosition)
386 {
387     double cosGamma;
388     size_t triangleCount = mesh->getTriangleCount();
389
390     for (size_t j = 0; j < triangleCount; ++j)
391     {
392         //cosGamma = *mesh->getTriangle(j)->getNormal() * obsVector;
393         //if triangle normal has the following components (x, y, z) and the
394         //observer vector (1, 0, 0), then
395         //(x, y, z) * (1, 0, 0) = x * 1 + y * 0 + z * 0 = x
396         cosGamma = mesh->getNormal(j)->x;
397
398         if (cosGamma > 0)
399         {
400             Triangle2 face(mesh, j, obsPosition - mesh->getCentroid(j)->x);
401
402             mesh->setProjectedArea(j, face.getArea());
403             mesh->setVisibility(j, 1.0);
404             float radius = face.getCircumCircle()->getRadius();
```

```
405                 if (radius > maxRadius)
406                     maxRadius = radius;
407 
408                 faces.push_back(face);
409             }
410             else
411                 mesh->setVisibility(j, 0.0);
412         }
413     }
414 
415     bool GridVisibility::distanceSorter(const Triangle2& left,
416                                         const Triangle2& right)
417     {
418         return left.getDistance() < right.getDistance();
419     }
```

PaintersVisibility.h

```
 1   #pragma once
 2 
 3   #include "Mesh3.h"
 4   #include "VisibilityDetection.h"
 5   #include <vector>
 6   #include <algorithm>
 7 
 8   #include "Triangle2.h"
 9 
10   namespace Infinity {
11   class PaintersVisibility : public VisibilityDetection
12   {
13   public:
14       PaintersVisibility(Mesh3* primary, Mesh3* secondary, Mesh3* disk,
15           int adaptive, int triangleIntersection, bool convexMesh = true);
16       ~PaintersVisibility();
17 
18       void run();
19       static bool distanceSorter(const Triangle2& left, const Triangle2&
20   right);
21   private:
22       bool hasDisk;
23 
24       bool checkVisibility(std::vector<Triangle2*>& checkedFaces,
25           Triangle2* triangle, int method) const;
26       void projectMesh(Mesh3* mesh, std::vector<Triangle2>& faces,
27           float& maxRadius, float obsPosition = 5.0);
28   };
```

PaintersVisibility.cpp

```
 1   #include "stdafx.h"
 2   #include "PaintersVisibility.h"
 3 
 4   using namespace std;
 5   using namespace DirectX::SimpleMath;
 6 
 7   namespace Infinity {
```

```
 8  PaintersVisibility::PaintersVisibility(Mesh3* primary, Mesh3* secondary,
 9      Mesh3* disc, int adaptive, int triangleIntersection, bool convexMesh)
10      : VisibilityDetection(primary, secondary, disc, adaptive,
11  triangleIntersection, true)
12  { hasDisk = (disc->getTriangleCount() != 0); }
13
14  PaintersVisibility::~PaintersVisibility() { }
15
16  void PaintersVisibility::run()
17  {
18      //Determine the distance of the components
19      float obsPosition = 5.0;
20
21      //Project & colect triangles
22      vector<Triangle2> faces;
23      vector<Triangle2*> checkedFaces;
24      faces.reserve(primary->getTriangleCount() + secondary->getTriangleCount()
25          + (hasDisk ? disk->getTriangleCount() : 0));
26      checkedFaces.reserve(primary->getTriangleCount()
27          + secondary->getTriangleCount() + (hasDisk ?
28          disk->getTriangleCount() : 0));
29
30      float primaryRadius = -1;
31      float secondaryRadius = -1;
32      float diskRadius = -1;
33      projectMesh(primary, faces, primaryRadius, obsPosition);
34      projectMesh(secondary, faces, secondaryRadius, obsPosition);
35      if (hasDisk)
36          projectMesh(disk, faces, diskRadius, obsPosition);
37
38      //Check wether there is a posibility for the eclipse.
39      double separation = pow2(primary->getCenter()->y
40          - secondary->getCenter()->y) + pow2(primary->getCenter()->z
41          - secondary->getCenter()->z);
42
43      double primaryOuterRadius = hasDisk ? disk->getOuterRadius()
44          : primary->getOuterRadius();
45      double secondaryOuterRadius = secondary->getOuterRadius();
46
47      if (separation > pow2(primaryOuterRadius + secondaryOuterRadius))
48          return;
49
50      //Collect the triangles
51      sort(faces.begin(), faces.end(), PaintersVisibility::distanceSorter);
52
53      Circle primaryBoundingCircle(primary->getCenter()->y,
54          primary->getCenter()->z, static_cast<float>(primaryOuterRadius)
55          + 3 * max(primaryRadius, diskRadius));
56      Circle secondaryBoundingCircle(secondary->getCenter()->y,
57          secondary->getCenter()->z, static_cast<float>(secondaryOuterRadius)
58          + 3 * secondaryRadius);
59
60      for (size_t i = 0; i < faces.size(); i++)
61      {
62          if (Circle::Intersects(&primaryBoundingCircle,
63              faces[i].getCircumCircle()))
64          {
```

```
65              //Check for the current cell
66              bool visible = checkVisibility(checkedFaces, &faces[i],
67                  triangleIntersection);
68
69              if (!visible)
70              faces[i].getParent()->setVisibility(faces[i].getIndex(), 0.0);
71
72              checkedFaces.push_back(&faces[i]);
73          }
74      }
75  }
76
77  bool PaintersVisibility::checkVisibility(vector<Triangle2*>& checkedFaces,
78                                           Triangle2* triangle, int method)
79  const
80  {
81      if (checkedFaces.size() == 0)
82          return true;
83
84      bool visible = true;
85
86      for (size_t k = 0; k < checkedFaces.size(); k++)
87      {
88          if (triangle->intersects(checkedFaces[k], method))
89          {
90              visible = false;
91              break;
92          }
93      }
94      return visible;
95  }
96
97  void PaintersVisibility::projectMesh(Mesh3* mesh, std::vector<Triangle2>&
98  faces,
99      float& maxRadius, float obsPosition)
100 {
101     double cosGamma;
102     size_t triangleCount = mesh->getTriangleCount();
103
104     for (size_t j = 0; j < triangleCount; ++j)
105     {
106         //cosGamma = *mesh->getTriangle(j)->getNormal() * obsVector;
107         //if triangle normal has the following components (x, y, z) and the
108         //observer vector (1, 0, 0), then
109         // (x, y, z) * (1, 0, 0) = x * 1 + y * 0 + z * 0 = x
110         cosGamma = mesh->getNormal(j)->x;
111
112         if (cosGamma > 0)
113         {
114             Triangle2 face(mesh, j, obsPosition - mesh->getCentroid(j)->x);
115
116             mesh->setProjectedArea(j, face.getArea());
117             mesh->setVisibility(j, 1.0);
118             float radius = face.getCircumCircle()->getRadius();
119             if (radius > maxRadius)
120                 maxRadius = radius;
121
```

```
122             faces.push_back(face);
123         }
124         else
125             mesh->setVisibility(j, 0.0);
126     }
127 }
128
129 bool PaintersVisibility::distanceSorter(const Triangle2& left,
130     const Triangle2& right)
131 {
132     return left.getDistance() < right.getDistance();
133 } }
```

VisibilityDetection.h

```
1  #pragma once
2
3  #include "Mesh3.h"
4
5  namespace Infinity {
6  class VisibilityDetection
7  {
8  public:
9      VisibilityDetection(Mesh3* primary, Mesh3* secondary,
10         Mesh3* disk = nullptr, int adaptive = 0,
11         int triangleIntersection = TRIANGLE_FULL, bool convexMesh = true);
12     ~VisibilityDetection(void);
13
14     virtual void run() = 0;
15 protected:
16     Mesh3* primary;
17     Mesh3* secondary;
18     Mesh3* disk;
19
20     int adaptive;
21     int triangleIntersection;
22     bool convexMesh;
23 }; }
```

VisibilityDetection.cpp

```
1  #include "stdafx.h"
2  #include "VisibilityDetection.h"
3
4  namespace Infinity {
5  VisibilityDetection::VisibilityDetection(Mesh3* primary, Mesh3* secondary,
6      Mesh3* disk, int adaptive , int triangleIntersection, bool convexMesh)
7      : primary(primary), secondary(secondary), disk(disk), adaptive(adaptive),
8      triangleIntersection(triangleIntersection), convexMesh(convexMesh)
9  { }
10
11 VisibilityDetection::~VisibilityDetection(void)
12 { } }
```

Literatura

Abt, H. A., 1983. Normal and abnormal binary frequencies. *Annual review of astronomy and astrophysics,* Volume 21, 343-372.

Atwood-Stone, C. et al., 2012. Modeling the Accretion Structure of AU Mon. *The Astrophysical Journal,* 760(2), AID 134, 1-16.

Bisikalo, D. V. et al., 2000. Circumstellar structures in the eclipsing binary Beta Lyr A - Gasdynamical modelling confronted with observations. *Astronomy and Astrophysics,* Volume 353, 1009-1015.

Castelli, F. & Kurucz, R. L., 2004. New Grids of ATLAS9 Model Atmospheres. *eprint arXiv:astro-ph/0405087.*

Claret, A. & Bloemen, S., 2011. Gravity and limb-darkening coefficients for the Kepler, CoRoT, Spitzer, uvby, UBVRIJHK, and Sloan photometric systems. *Astronomy & Astrophysics,* Volume 529, A75, 1-5.

Cozic, L., 2006. *2D Polygon Collision Detection.* [Online]
Available at: http://www.codeproject.com/Articles/15573/2D-Polygon-Collision-Detection
[Accessed 1 May 2013].

Cséki, A. & Prša, A., 2010. Calculation of Emergent Intensities for the Wilson-Devinney Code with New Kurucz Stellar Atmosphere Models. *ASP Conf. Ser. 434, Binaries: Key to Comprehension of the Universe, Edited by Andrej Prša and Miloslav Zejda,* Volume 438, 81-82.

Desmet, M. et al., 2010. CoRoT photometry and high-resolution spectroscopy of the interacting eclipsing binary AU Monocerotis. *Monthly Notices of the Royal Astronomical Society,* 401(1), 418-432.

Devor, J., 2005. Solutions for 10,000 Eclipsing Binaries in the Bulge Fields of OGLE II Using DEBiL. *The Astrophysical Journal,* 628(1), 411-425.

Drechsel, H., Haas, S., Lorenz, R. & Gayler, S., 1995. Radiation pressure effects in early-type close binaries and implications for the solution of eclipse light curves. *Astronomy and Astrophysics,* 294(3), 723-743.

Đurašević, G., 1991. Ispitivanje aktivnih tesnih dvojnih sistema na osnovu fotometrijskih merenja. *Publication of the Astronomical Observatory of Belgrade,* Volume 42, 1-215.

Đurašević, G., 1992a. An analysis of active close binaries (CB) based on photometric measurements. I - A model of active CB with spots on the components. *Astrophysics and Space Science,* 196(2), 241-265.

Đurašević, G., 1992b. An Analysis of Active Close Binaries / CB / Based on Photometric Measurements - Part Two - Active CB with Accretion Discs. *Astrophysics and Space Science,* 196(2), 267-282.

Đurašević, G. et al., 2013. Photometric Analysis of HS Aqr, EG Cep, VW LMi, and DU Boo. *The Astronomical Journal,* 145(3), AID 80, 1-10.

Đurašević, G., Latković, O., Vince, I. & Cséki, A., 2010. Accretion disc in the eclipsing binary AU Mon. *Monthly Notices of the Royal Astronomical Society,* 409(1), 329-336.

Đurašević, G. et al., 2012. A study of the interacting binary system V455 Cygni. *Monthly Notices of the Royal Astronomical Society,* 420(4), 3081-3090.

Đurašević, G., Vince, I. & Atanacković, O., 2008. Accretion Disk in the Massive Binary RY Scuti. *Astronomical Journal,* 136(2), 767-772.

Đurašević, G., Zakirov, M., Hojaev, A. & Arzumanyants, G., 1998. Analysis of the activity of the eclipsing binary WZ Cephei. *Astronomy and Astrophysics Supplement,* Volume 131, 17-23.

Eberly, D., 2008. *Platonic Solids.* [Online]
Available at: http://www.geometrictools.com/Documentation/PlatonicSolids.pdf
[Accessed 24 April 2013].

Etzel, P. B., 1981. A Simple Synthesis Method for Solving the Elements of Well-Detached Eclipsing Systems. *Photometric and Spectroscopic Binary Systems, Proceedings of the NATO Advanced Study Institute,* 111.

Foley, J., van Dam, A., Feiner, S. K. & Hughes, J. F., 1995. *Computer Graphics: Principles and Practice.* 1st ed., Addison-Wesley.

Goodricke, J., 1783. a Series of Observations on, and a Discovery of, the Period of the Variation of the Light of the Bright Star in the Head of Medusa, Called Algol. *Philosophical Transactions of the Royal Society of London,* Volume 73, 474-482.

Gray, R. O. & Corbally, C. J., 1994. The calibration of MK spectral classes using spectral synthesis. 1: The effective temperature calibration of dwarf stars. *Astronomical Journal,* 107(2), 742-746.

Harmanec, P., Bisikalo, D. V., Boyarchuk, A. A. & Kuznetsov, O. A., 2002. On the role of duplicity in the Be phenomenon. I. General considerations and the first attempt at a 3-D gas-dynamical modelling of gas outflow from hot and rapidly rotating OB stars in binaries. *Astronomy and Astrophysics,* Volume 396, 937-948.

Heemskerk, M. H. M., 1994. Hydrodynamic calculations of accretion discs in close binaries. The superhump phenomenon. *Astronomy and Astrophysics,* Volume 288, 807-818.

Hendry, P. D. & Mochnacki, S. W., 1992. The GDDSYN light curve synthesis method. *Astrophysical Journal,* Volume 338, 603-613.

Herschel, W., 1803. Account of the Changes That Have Happened, during the Last Twenty-Five Years, in the Relative Situation of Double-Stars; With an Investigation of the Cause to Which They Are Owing. *Philosophical Transactions of the Royal Society of London,* Volume 93, 339-382.

Hilditch, R. W., 2001. *An Introduction to Close Binary Stars.* Cambridge University Press.

Hut, P., 1981. Tidal evolution in close binary systems. *Astronomy and Astrophysics,* 99(1), pp. 126-140.

Kallrath, J. & Milone, E. F., 2009. *Eclipsing Binary Stars: Modeling and Analysis.* Second Edition ed. New York, Springer.

Kambe, E. A. H. S. B. I. H. et al., 2008. Development of Iodine Cells for Subaru HDS and Okayama HIDES. III. An Improvement on the Radial-Velocity Measurement Technique. *Publications of the Astronomical Society of Japan,* 60(1), 45-53.

Khruzina, T., 1985. Synthesis of the Lightcurves of X-Ray Binary Systems with Eccentric Orbits. *Soviet Astronomy,* Volume 29, 55-59.

Kopal, Z., 1959. *Close Binary Stars.* London, Chapman & Hall.

Latković, O., 2013, u pripremi. *Modeliranje zvezdanih oscilacija u tesno dvojnim sistemima.* u pripremi.

Lubow, S. H. & Shu, F. H., 1975. Gas dynamics of semidetached binaries. *Astrophysical Journal,* 198(1), 383-405.

Lucy, L. B., 1967. Gravity-Darkening for Stars with Convective Envelopes. *Zeitschrift für Astrophysik,* Volume 65, 89-92.

Lucy, L. B. & Sweeney, M. A., 1971. Spectroscopic binaries with circular orbits. *Astronomical Journal,* Volume 76, 544-556.

Mennickent, R. E., Pietrzyński, G., Diaz, M. & Gieren, W., 2003. Double-periodic blue variables in the Magellanic Clouds. *Astronomy and Astrophysics,* Volume 399, L47-L50.

Metcalfe, T. S., Mathieu, R. D., Latham, D. W. & Torres, G., 1996. The Low-Mass Double-lined Eclipsing Binary CM Draconis: A Test of the Primordial Helium Abundance and the Mass-Radius Relation near the Bottom of the Main Sequence. *Astrophysical Journal,* Volume 456, 356-364.

Mochnacki, S. W. & Doughty, N. A., 1972. A model for the totally eclipsing W Ursae Majoris system AW UMa. *Monthly Notices of the Royal Astronomical Society,* Volume 156, 51-56.

Moro, D. & Munari, U., 2000. The Asiago Database on Photometric Systems (ADPS). I. Census parameters for 167 photometric systems. *Astronomy and Astrophysics Supplement,* Volume 147, 361-628.

Nelder, J. A. & Mead, R., 1965. A Simplex Method for Function Minimization. *The Computer Journal,* Volume 7, 308–313.

Nelson, B. & Davis, W. D., 1972. Eclipsing-Binary Solutions by Sequential Optimization of the Parameters. *Astrophysical Journal,* Volume 174, 617-628.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P., 2007. *Numerical Recipes - The Art of Scientific Computing.* 3rd ed. Cambridge, Cambridge University Press.

Rafert, J. B. & Twigg, L. W., 1980. Observational determination of the gravity darkening exponent and bolometric albedo for close binary star systems. *Monthly Notices of the Royal Astronomical Society,* Volume 193, 79-86.

Rossiter, R. A., 1924. On the detection of an effect of rotation during eclipse in the velocity of the brigher component of beta Lyrae, and on the constancy of velocity of this system.. *Astrophysical Journal,* Volume 60, 15-21.

Rucinski, S. M., 1969. The Proximity Effects in Close Binary Systems. II. The Bolometric Reflection Effect for Stars with Deep Convective Envelopes. *Acta Astronomica,* Volume 19, 245-255.

Rucinski, S. M. et al., 2008. Radial Velocity Studies of Close Binary Stars. XIII. *The Astronomical Journal,* 136(2), 586-593.

Russell, H. N., 1912a. On the Determination of the Orbital Elements of Eclipsing Variable Stars. I.. *Astrophysical Journal,* Volume 35, 315-340.

Russell, H. N., 1912b. On the Determination of the Orbital Elements of Eclipsing Variable Stars. II. *Astrophysical Journal,* Volume 36, 54-74.

Sasselov, D. D., 1998. Surface Imaging by Microlensing. *ASP Conf. Ser. 154 - The Tenth Cambridge Workshop on Cool Stars, Stellar Systems and the Sun, Edited by R. A. Donahue and J. A. Bookbinder,* 383-391.

Southworth, J., 2008. Homogeneous studies of transiting extrasolar planets - I. Light-curve analyses. *Monthly Notices of the Royal Astronomical Society,* 386(3), 1644-1666.

Strohmeier, W., 1958. Neue veranderliche Sterne.. *Kl. Veröff. Bamberg,* Volume 21, 22.

Takahama, T. & Sakai, S., 2005. Constrained Optimization by Applying the Alpha Constrained Method to the Nonlinear Simplex Method With Mutations. *IEEE Transactions on Evolutionary Computation,* 9(5), 437-451.

Tamuz, O., Mazeh, T. & North, P., 2006. Automated analysis of eclipsing binary light curves - I. EBAS - a new Eclipsing Binary Automated Solver with EBOP. *Monthly Notices of the Royal Astronomical Society,* 367(4), 1521-1530.

Tassoul, J.-L., 2000. *Stellar Rotatio,* Cambridge University Press .

Van Hamme, W. & Wilson, R. E., 2003. Stellar atmospheres in eclipsing binary models. *ASP Conf. Ser. 298 - GAIA Spectroscopy - Science and Technology, Edited by Ulisse Munari,* Volume 298, 323-328.

Vogel, H. C., 1890. Orbit and Mass of Algol. *Publications of the Astronomical Society of the Pacific,* 2(6), 27.

von Zeipel, H., 1924. The radiative equilibrium of a rotating system of gaseous masses. *Monthly Notices of the Royal Astronomical Society,* Volume 84, 665-683.

Weisstein, E. W., 2013. *"Rotation Matrix." From MathWorld--A Wolfram Web Resource.* [Online]
Available at: http://mathworld.wolfram.com/RotationMatrix.html
[Accessed 24 April 2013].

Wilson, R. E., 1979. Eccentric orbit generalization and simultaneous solution of binary star light and velocity curves. *Astrophysical Journal,* Volume 234, 1054-1066.

Wilson, R. E. & Devinney, E. J., 1971. Realization of Accurate Close-Binary Light Curves: Application to MR Cygni. *Astrophysical Journal,* Volume 166, 605-619.

Wilson, R. E. & Sofia, S., 1976. Effects of tidal distortion on binary-star velocity curves and ellipsoidal variation. *Astrophysical Journal,* 203(1), 182-186.

Wilson, R. E. & Van Hamme, W., 2013. *Computing Binary Star Observables.* [Online]
Available at: ftp://ftp.astro.ufl.edu/pub/wilson/lcdc2013/
[Accessed Jun 2013].

Yamasaki, A., 1982. A SPOT model for VW Cephei. *Astrophysics and Space Science,* 85(1-2), 43-58.

Zola, S., 1991. RZ OPH - The Algol-type, long-period binary star with a thick accretion disk. *Acta Astronomica,* 41(3), 213-230.

Biografija autora

Čeki Atila rođen je 7. jula 1980. godine u Pančevu. Završio je Gimnaziju „Uroš Predić“ u Pančevu 1999. godine i iste godine je upisao smer Astrofizika na Matematičkom fakultetu u Beogradu. Diplomirao je u oktobru 2006. godine sa prosečnom ocenom 9.58. U martu 2007. upisao je doktorske studije na Matematičkom fakultetu. Zaposlen je na Astronomskoj opservatoriji od januara 2007. godine i trenutno ima zvanje istraživač-saradnik.

Od 2007. do 2010. godine je bio angažovan na projektu 146003 – „Fizika Sunca i zvezda“ pod rukovodstvom dr Gojka Đuraševića sa punim istraživačkih vremenom. Od 2010. godine učestvuje na projektima 176004 – „Fizika zvezda“ pod rukovodstvom dr Gojka Đuraševića (sa jedanaest istraživačkih meseci) i 176021 – „Vidljiva i nevidljiva materija u bliskim galaksijama: teorija i posmatranja“ pod rukovodstvom dr Srđana Samurovića (sa jednim istraživačkim mesecom).

Istraživačka aktivnost kandidata je usmerena na razvoj programa za modeliranje dvojnih sistema, kao i na analizu pojedinačnih dvojnih sistema. Koautor je dva rada objavljena u vrhunskim međunarodnim časopisima, tri rada u nacionalnim časopisima od međunarodnog značaja i sedam radova sa konferencija štampanih u celini. Ima jedan samostalni rad.

Изјава о ауторству

Потписани Атила Чеки

број уписа _____

Изјављујем

да је докторска дисертација под насловом

Усавршавање модела тесно двојних система сагласно са резултатима посматрања високе прецизности

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 11.09.2013.

_____

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Атила Чеки

Број уписа _____

Студијски програм Астрономија и астрофизика

Наслов рада Усавршавање модела тесно двојних система сагласно са резултатима посматрања високе прецизности

Ментор проф. др Олга Атанацковић

Потписани Атила Чеки

изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 11.09.2013.

_____

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Усавршавање модела тесно двојних система сагласно са резултатима посматрања високе прецизности

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

(1.) Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 11.09.2013.

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.