

Дипломна робота: 44 с., рис. 5, табл. 1, джерел 10, додатків 1.

Об'єкт дослідження: рух космічного апарату та його електризація елементарними частинками плазми.

Мета роботи: розробка програмного забезпечення для моделювання електризації тіл в космічному просторі, а також обчислення значень потенціалів, що будуть накопичуватися на цих тілах або їх частинах.

Одержані висновки та їх новизна: розроблено програму, яка моделює рух частинок плазми в околиці космічного апарату, зіткнення цих частинок з апаратом, його електризацію. Новизна полягає у використанні при моделюванні такого процесу статистичного методу Монте-Карло, тобто параметри моделі задаються випадковими величинами, за розподілом наближених до реальних величин – це дозволяє говорити про відповідність отриманих результатів параметрам справжньої системи.

Результати дослідження можуть бути застосовані при конструюванні космічних апаратів; також розроблена база може бути використана при розробці схожих моделей.

Перелік ключових слів: ШТУЧНИЙ СУПУТНИК, КОСМІЧНИЙ АПАРАТ, ЕЛЕКТРИЗАЦІЯ, КОСМІЧНА ПЛАЗМА, МЕТОД МОНТЕ-КАРЛО, МОДЕЛЮВАННЯ, ЕЛЕМЕНТАРНІ ЧАСТИНКИ.

Annotation

The graduation research of the 4-year student Vsevolod Kulaga (DNU, Applied Mathematics Department, the Computer Technology Chair) deals with the development of software for calculation of spacecrafts electrification using the statistical Monte-Carlo method.

The developed software reads models of the spacecrafts from the file (many popular file formats for 3D models are supported), performs simulation of the motion of elementary particles and processes their collision with the spacecraft. At that time program also calculates the total charge and potential that appears on the spacecrafts surface. The process of modeling is visualized using the OpenGL library.

The software is developed for researching of spacecrafts electrification process to reduce its negative effects in future.

The work is interesting for spacecrafts designers and programmers who deal with similar tasks.

Bibliography 10, pictures 5, tables 1, supplement.

Зміст

Вступ

Добре відомо, що тіло, поміщене в рівноважну плазму, набуває негативний потенціал, величина якого якого близька до температури плазми. Цей факт, дослідження якого було розпочато Ленгмюром, який створив основи теорії електричних зондів, придбав нове значення у зв'язку з запусками та експлуатацією високоорбітальних космічних апаратів (КА).

Справа в тому, що якщо низькоорбітальні КА взаємодіють з плазмою, температура якої не перевищує одиниць вольт, і набувають внаслідок цього незначні негативні потенціали, то високоорбітальні КА, що потрапляють, наприклад, в плазмовий шар магнітосфери, можуть заряджатися до потенціалу 1-20 кВ. Такі різниці потенціалів між КА і навколишнього плазмою здатні значно спотворювати вимірювання потоків і спектрів заряджених частинок, що проводяться на космічних апаратах. Якщо ж врахувати, що потенціал поверхні КА залежить не тільки від параметрів навколишнього плазми, але і від умов освітленості, що більшість КА мають нееквіпотенційну поверхню, що як параметри навколишнього середовища, так і умови освітленості КА змінюються в часі, то можна уявити всю складність картини електростатичної зарядки високоорбітальних космічних апаратів. Неоднорідності електрофізичних характеристик поверхні КА і нерівномірність її освітленості призводять також до появи мінливих у часі диференційних різниць потенціалів, які можуть бути причиною розрядів, потенційно небезпечних для нормального функціонування електронних пристроїв, розміщених на космічних апаратах.

Таким чином, електростатична зарядка (електризація) високоорбітальних КА – це складний фізичний процес, необхідність дослідження якого диктується як потребами підвищення точності і якості бортових вимірювань радіаційної обстановки близько КА, так і шкідливими впливами факторів електризації, що погіршують надійність і ресурсні характеристики космічних апаратів.

Експериментальні дослідження електризації геостаціонарних КА підтверджують загальні теоретичні уявлення про те, що середній потенціал геостаціонарних КА, а також диференційні різниці потенціалів можуть

приймати значення до десятків кіловольт. Встановлено, що найбільші значення потенціалів припадають на нічні ділянки траєкторії, а екстремально великі значення реєструвалися в моменти магнітосферних суббурь, коли КА перебували в тіні від Землі. Ступінь електризації позитивно корелює з геомагнітною збуреністю, а деталі процесу електризації кожного конкретного КА виявилися залежними від його конструктивних особливостей [?].

Робота починається з розділу, котрий освітлює фізичну модель задачі. В ньому розглядається космічна плазма, її склад, характеристики і поведінка. Також в розділі наявні відомості про метод Монте-Карло, за яким проводиться моделювання.

Другий розділ містить в собі опис математичної моделі – відомості про створені структури даних, про функції обробки геометричних об'єктів моделі, дається опис функцій генерації випадкових величин з різноманітними ймовірнісними розподілами, а також функцій обробки вхідних даних та список підтримуваних форматів.

Останній розділ складається з опису процесу моделювання, інтерфейсу програми і демонстрації отриманих результатів.

Постановка задачі

Для штучних супутників Землі (ШСЗ) однією з найважливіших є проблема електризації та заняття електричного заряду з поверхні ШСЗ в процесі експлуатації.

Суть проблеми полягає в тому, що ШСЗ, які знаходяться на високих орбітах, насамперед на геостаціонарних, піддаються нерівномірній електризації швидкими електронами. При цьому ШСЗ в цілому, або окремі частини його поверхні, які знаходяться в тіні сонця, заряджаються до високого від'ємного потенціалу відносно оточуючого космічного простору. [?]. Через неоднакову освітленість діелектричних ділянок ШС виникає різниця потенціалів, яка викликає електричні пробої – вони ведуть до збоїв в роботі радіоелектронних приладів і руйнують поверхню супутника. Ефект електричного зарядження особливо посилюється в період геомагнітних бурь, пов'язаних з підвищеною сонячною активністю. У таких випадках від'ємний потенціал геостаціонарного супутника може досягати значних величин. [?] [?]

Дана робота присвячена розробці програмного забезпечення для моделювання електризації тіл в космічному просторі, а також обчислення значень потенціалів, що будуть накопичуватися на цих тілах або їх частинах. В роботі розглядається випадок струму частин плазми на поверхні незарядженого тіла, оскільки він є більш простим, але програмне забезпечення має бути розроблене таким чином, щоб мати можливість його розвинути для обробки випадку поверхні зарядженого тіла.

Для побудови математичної моделі необхідно виконати огляд наявних експериментальних даних, які б описували поведінку і параметри елементів системи, що моделюється – наприклад, щільність і температура космічної плазми, радіус екранування електричного заряду в ній, швидкості елементарних частинок і космічних апаратів тощо.

Також для більш точної відповідності побудованої моделі реальній системі, потрібно якомога точніше описати в програмі елементи цієї системи, створивши для них відповідні типи даних, класи та алгоритми, які моделювали б їх взаємодію.

Після створення програмного забезпечення, що відповідає описаним вимогам, за його допомогою має бути проведена серія стохастичних випробувань, яка, згідно зі статистичним методом Монте-Карло, дозволить прослідкувати зміну потенціалу космічного апарату і струмів на його поверхні з плином часу.

1 Огляд

1.1 Історичний огляд

Вперше ефекти електростатичної зарядки були розглянуті В.Г. Куртом і В.І. Морозом ще в 1961 році. Їх робота стала потужним поштовхом в розвитку досліджень, пов'язаних зі створенням засобів електростатичного захисту від космічних випромінювань і вивченням можливості їх використання в комплексі з іншими системами космічного апарату при тривалому знаходженні останнього в області радіаційного впливу.

В наші дні велика увага приділяється комп'ютерному моделюванню процесів електричного зарядження поверхні космічних апаратів. Комп'ютерне моделювання дозволяє отримати повну картину електризації поверхні апарату заданої геометрії в умовах космосу без значних затрат часу і коштів.

Існує декілька комп'ютерних програм, призначених для розрахунку електризації поверхні космічних апаратів. Найпершою такою програмою стала NASCAP (NASA Charging Analyzer Program), розроблена S-Cubed (System, Science and Software) для NASA і ВПС США близько 1980 року.

В даній роботі розробляється програма для розрахунку електризації космічних апаратів, в якій це робиться за допомогою статистичного методу Монте-Карло.

1.2 Фізична модель

При взаємодії космічного апарату (КА) з плазмою, що оточує його в польоті, виникають різноманітні фізичні явища, специфіка яких залежить як від параметрів плазми, так і від характеристик КА, в першу чергу – від властивостей матеріалів, що знаходяться на його поверхні, і від конфігурації апарату. До таких явищ належать: утворення електричного заряду на поверхні КА, розпилення матеріалів, світіння на поверхні і поблизу неї, збудження коливань у плазмі та деякі інші.

Найбільш значний вплив на функціонування КА може здійснювати утворення заряду на його поверхні. Знак і величина електричного заряду, що утворюється на поверхні КА, залежать від співвідношення інтенсивності

процесів, що забезпечують надходження на поверхню і видалення з неї позитивно і негативно заряджених частинок, тобто від співвідношення різних складових сумарного електричного струму, що тече через поверхню КА. Основними складовими цього струму є електронний та іонний струми навколишньої плазми, вторинно-емісійні струми, обумовлені первинними плазовими струмами, і фотоелектронний струм, що виникає під дією короткохвильового випромінювання Сонця. Додаткові складові можуть створюватися деякими видами бортового устаткування КА: електроракетними двигунами, що випускають при роботі плазові струмені, електронними та іонними прожекторами, що використовуються в наукових експериментах і т.п.

При електризації КА між його поверхнею і навколишньої плазмою виникає різниця потенціалів. Сталий потенціал поверхні, відлічуваний щодо потенціалу незбуреної плазми, визначається умовою динамічної рівноваги, при якому сумарний струм, що тече через поверхню КА, дорівнює нулю. З енергетичних співвідношень випливає, що рівноважний потенціал залежить від середньої енергії частинок плазми, тобто від її температури: чим вища температура плазми, тим більший потенціал може отримати поверхня тіла. В багатокомпонентній космічній плазмі характерне значення максимального потенціалу визначається енергією заряджених частинок, що превалюють в струмовому балансі.

Реальний КА являє собою складну конструкцію з неоднорідною структурою і великою кількістю діелектричних матеріалів на зовнішній поверхні. У зв'язку з цим потенціали окремих ділянок поверхні і елементів конструкції можуть бути різними через відмінності умов потрапляння потоків первинних частинок на ці ділянки та умов їх освітлення, а також через відмінності емісійних властивостей матеріалів поверхні. Відбувається так зване диференційне зарядження КА, при якому між окремими ділянками непровідної поверхні виникають різниці потенціалів.

Описаний вище процес зарядження КА як єдиного провідного тіла прийнято називати загальним зарядженням. Дане поняття можна застосувати і по відношенню до реального КА, проте в цьому випадку воно відноситься до середнього потенціалу КА, що визначається сукупністю всіх електричних

зарядів, які знаходяться на його поверхні і елементах конструкції.

Очевидно, що власне електричне поле зарядженого КА є збурюючим фактором, який необхідно враховувати в багатьох випадках при проведенні вимірювань параметрів космічного середовища за допомогою приладів, встановлених на КА. З цієї точки зору явище електризації КА в космічній плазмі аналізувалося ще в середині 1950-х рр. при розробці наукових приладів для перших штучних супутників Землі. Тоді приймалися до уваги потенціали з характерними величинами від часток до одиниць вольт, що, як ми побачимо далі, характерно для випадку зарядження КА на низьких навколоземних орбітах – в іоносфері.

Проте найбільший вплив на бортове обладнання КА мають електростатичні розряди (ЕСР), які можуть виникати між окремими ділянками поверхні і елементами конструкції диференційно зарядженого КА, а також між його поверхнею і навколишньої плазмою. Локальні струми і електромагнітні випромінювання, породжувані ЕСР, створюють значні перешкоди для роботи бортового обладнання КА.

Як фактор, що має серйозний несприятливий вплив на роботу бортових систем КА, явище електризації стало систематично вивчатися на початку 1970-х рр. при запусках КА на геостаціонарну орбіту (кругова екваторіальна орбіта з висотою 36000 км), де, як з'ясувалося пізніше, параметри плазми такі, що значення потенціалів на КА досягають 10-20 кВ.

Геостаціонарна орбіта (ГСО) примітна тим, що на ній кутова швидкість руху КА дорівнює швидкості обертання Землі. Внаслідок цього КА постійно знаходиться над однією точкою земної поверхні (звідси назва орбіти), забезпечуючи тим самим дуже зручні умови для трансляції через нього радіосигналів. Тому геостаціонарні КА працюють головним чином в космічних системах радіозв'язку та телебачення.

На перших геостаціонарних КА, спроектованих без урахування можливого впливу ефектів електризації, спостерігалася велика кількість неполадок в роботі бортового обладнання: відбувалися мимовільні включення і виключення різних пристроїв, змінювалася орієнтація антен, припинялася подача електроенергії від сонячних батарей і т.д., причому аномалії спостерігалися переважно в нічні та ранні ранкові години. Не відразу вдалося

зрозуміти, що всі ці ефекти пов'язані з електризацією КА.

Поступово при статистичному аналізі відмов і збоїв у роботі апаратури КА був виявлений кореляційний зв'язок між спостережуваними аномаліями і появою інтенсивних потоків гарячої плазми в області ГСО. На геостаціонарних КА були встановлені прилади для вимірювання параметрів навколишньої плазми і спеціальні датчики для реєстрації електромагнітних перешкод і вимірювання напруженості електричного поля біля поверхні КА. Дані, отримані за допомогою цих приладів, переконливо підтвердили факт виникнення ЕСР на борту КА при електризації під дією гарячої плазми. При характерних значеннях потенціалів на геостаціонарних КА, вимірюваних одиницями і навіть десятками кіловольт, рівень перешкод, створюваних ЕСР, дуже високий, а в деяких випадках ЕСР можуть призводити до руйнування компонентів апаратури та елементів конструкції.

Вжиті теоретичні і лабораторні дослідження явища електризації КА дозволили зрозуміти основні його закономірності і запропонувати методи зниження впливу ефектів електризації на функціонування бортових систем КА. Однак проблема далеко не вичерпана. Створення нових конструкцій КА, підвищення вимог до їх надійності та тривалості функціонування, оснащення КА новими видами устаткування і високочутливою науковою апаратурою – все це потребує подальшого детального вивчення особливостей електризації КА в різних умовах і вдосконалення методів їх захисту.[?]

1.2.1 Визначення плазми

Фізика плазми грає дуже важливу роль в космофізичних і астрофізичних дослідженнях, оскільки плазма є основним станом речовини у Всесвіті. Взагалі кажучи, не всякий іонізований газ може бути названий плазмою. Термін плазма використовується по відношенню до іонізованих газів, в яких можливе мимовільне розділення зарядів за рахунок хаотичного руху частинок мале в порівнянні з макроскопічної щільністю зарядів. Це обумовлено тим, що поведінка великого числа заряджених частинок в плазмі контролюється кулонівськими силами, що діють на великі відстані, порівняно з якими сили взаємодії з довколишніми зарядженими і нейтральними частками (короткодійні взаємодії) зневажливо малі. Таким чином, плазма

повинна містити досить велике число заряджених частинок – електронів та іонів, але разом з тим щільність іонізованого газу повинна бути мала. З даного визначення випливає, що в макроскопічному відношенні плазма електрично нейтральна.

У загальному випадку плазма складається з суміші заряджених і нейтральних частинок. Відношення концентрації заряджених частинок в газі до повної концентрації частинок називається ступенем іонізації. Залежно від цього параметра розрізняють слабо іонізовану плазму (ступінь іонізації порядку часток відсотка), помірно іонізовану (кілька відсотків) і повністю іонізовану.

У космічному просторі зустрічаються всі вказані види плазми. Наприклад, іоносферна плазма, про яку докладніше ми будемо говорити нижче, є слабо іонізованою, а плазма в області геостаціонарної орбіти – практично повністю іонізованою.

Кулонівських сили, що діють на великі відстані, в значній мірі визначають електричні та статистичні характеристики плазми.[?]

1.2.2 Екранування поля електричного заряду в плазмі . Дебаєвський радіус екранування

Електричне поле, яке створює пробний заряд q , внесений до плазми, виявиться на деякій відстані екранованим, так як такий заряд притягує заряджені частинки плазми протилежного знаку і відштовхує однойменно заряджені частинки, тобто в околиці пробного заряду відбувається зміна просторового розподілу електронів та іонів плазми.

Розподіл потенціалу електричного поля ϕ в околиці пробного заряду може бути знайдено за допомогою рівняння Пуассона

$$\Delta\phi = -4\pi\rho,$$

де $\rho = e(n_i - n_e)$ – щільність об'ємного заряду в плазмі; n_i – концентрація іонів, n_e – концентрація електронів, e – елементарний електричний заряд.

Тут використаний запис рівняння в системі СГСЕ для середовища з відносною діелектричною проникністю, що дорівнює одиниці.

Для точки простору з координатою r можна записати

$$\Delta\phi = -4\pi e(n_i(r) - n_e(r)).$$

Концентрації частинок плазми змінюються в електричному полі позитивного пробного заряду згідно з формулою Больцмана

$$n_i(r) = n \exp \left[-\frac{e\phi(r)}{kT_i} \right]$$

$$n_e(r) = n \exp \left[\frac{e\phi(r)}{kT_e} \right],$$

де n – концентрація заряджених частинок незбуреній області плазми, тобто в області, де електричне поле пробного заряду відсутнє; T_i , T_e – температура іонної та електронної складових плазми; k – постійна Больцмана.

Вирішуючи рівняння Пуассона з урахуванням розподілів $n_i(r)$ і $n_e(r)$ стосовно до точкового пробного заряду q і вважаючи що $T_i = T_e = T$, знайдемо вираз для потенціалу на відстані r від заряду

$$\phi(r) \approx \frac{q}{r} \exp \left(-\frac{r}{\lambda_d} \right),$$

де $\lambda_d = \left(\frac{kT}{8\pi n e^2} \right)^{\frac{1}{2}}$ – Дебаєвський радіус екранування (радіус Дебая). Можна записати також

$$\lambda_d \cong 4.9 \left(\frac{T}{n} \right)^{\frac{1}{2}},$$

де T – в кельвінах, n – в см. Якщо $T_i \neq T_e$, радіус Дебая дається виразом

$$\lambda_d = \left(\frac{k}{4\pi n e^2} \frac{T_i T_e}{T_i + T_e} \right)^{\frac{1}{2}}.$$

Аналогічним виразом можна користуватися при аналізі енергетично багатокomпонентної плазми з урахуванням концентрацій окремих складових [?].

У випадку електронейтральної системи дебаєвський радіус також зна-

ходитьсь з рівняння Дебая-Хюкеля

$$\lambda_d = \left(\frac{\varepsilon_r \varepsilon_0 k_B T}{\sum_{j=1}^N n_j^0 q_j^2} \right)^{\frac{1}{2}}, \quad (1)$$

де ε_r – відносна діелектрична провідність, яку, як вже зазначалось, приймаємо рівній одиниці, ε_r – діелектрична постійна, $\varepsilon_r \approx 8.854187817 \cdot 10^{-12}$ Ф/м, n_j^0 – середня концентрація зарядів типу j з величиною заряду q_j .

Дебаєвський радіус екранування λ_d визначає характерні розміри сфери, в межах якої в плазмі виявляється дія електричного поля пробного заряду.

1.2.3 Загальна характеристика і математичний опис гарячої магнітосферної плазми

В магнітосфері Землі гаряча плазма присутня переважно на висотах, що вимірюються тисячами кілометрів – в плазмовому шарі і в області авроральної радіації, яка проектується уздовж геомагнітних силових ліній на іоносферні висоти, утворюючи авроральні овали, усередині яких електрони з енергіями 1-50 кеВ проникають в атмосферу до висот 100 км, викликаючи полярні сйва.

Екваторіальний кордон авроральної овалу лежить на широті 68° на нічній стороні і на широті 75° – на денній стороні. Ширина зони висипань становить $2 - 3^\circ$. Обидва зазначених параметра залежать від рівня геомагнітної активності: зі збільшенням активності відбувається розширення овалу і зміщення його зовнішнього кордону до екватора. В умовах сильних геомагнітних збурень зовнішня межа овалу може розташовуватися на широті 55° , а внутрішня - на широті 80° . Вплив на КА авроральних електронів має спорадичний характер.

Таким чином, в високоширотних областях низькоорбітальні КА можуть піддаватися одночасному впливу холодної іоносферної плазми і потоків авроральних електронів. В таких умовах потенціал КА досягає 1-5 кВ. Відразу ж відзначимо, що цей випадок електризації найбільш важкий для аналізу.

При розгляді зовнішніх факторів, що викликають електризацію КА, найбільшу увагу приділяють аналізу характеристик гарячої магнітосферної плазми, оскільки саме вона викликає появу на КА найбільш високих потенціалів.

До теперішнього часу характеристики гарячої магнітосферної плазми досить добре вивчені, чому неабиякою мірою сприяли дослідження електризації геостаціонарних КА. Енергетичні спектри електронів та іонів гарячої магнітосферної плазми, зокрема в області ГСО, займають діапазон енергій від 0,05 до 100 кеВ. Проведені дослідження показали, що функція розподілу часток гарячої магнітосферної плазми досить точно апроксимується суперпозицією двох максвеллівських розподіл з характерними енергіями $kT_1 \cong 0,2 - 0,4$ кеВ і $kT_2 \cong 5 - 10$ кеВ

$$f_j(v_j) = n_{1j} \left(\frac{m_j}{2\pi kT_{1j}} \right)^{\frac{3}{2}} \exp \left(-\frac{m_j v_j^2}{2\pi kT_{1j}} \right) + n_{2j} \left(\frac{m_j}{2\pi kT_{2j}} \right)^{\frac{3}{2}} \exp \left(-\frac{m_j v_j^2}{2\pi kT_{2j}} \right),$$

де n_j - концентрація часток j -ого сорту (електрони, протони або інші іони) відповідно для складових з температурами T_1 і T_2 ; m_j , v_j – маса і швидкість частинок.

Характеристики гарячої магнітосферної плазми в області ГСО вимірювалися за допомогою апаратури багатьох КА. В таблиці ?? наведені параметри двухтемпературних максвеллівських функцій розподілу для електронів і протонів на ГСО, які запропоновано розглядати як «найгірших умов» функціонування КА з точки зору його електризації. Реально параметри плазми можуть змінюватися в досить широких межах залежно від рівня сонячної та геомагнітної активності.

Експериментально встановлено, що в області ГСО крім іонів H^+ , які є зазвичай основним іонним компонентом плазми, можуть бути присутніми іони He^+ , O^+ , He^{2+} , O^{2+} іоносферного походження. Зміст іонів в плазмі зазвичай зростає з підвищенням рівня геомагнітної активності, а при дуже високій збуреності концентрація іонів O^+ може в деяких випадках перевищувати концентрацію іонів H^+ . Тому іонний склад плазми бажано враховувати в моделі електризації геостаціонарних КА. Однак поки в більшості випадків при аналізі електризації розглядають електроннопротонну

плазму [?].

Табл. 1: Параметри двухтемпературних максвеллівських функцій розподілу для електронів і протонів на ГСО «для найгіршого випадку»

Параметр	Електрони	Протони
$n_1, \text{см}^{-3}$	0.2	0.6
$kT_1, \text{кеВ}$	0.4	0.2
$n_2, \text{см}^{-3}$	1.2	1.3
$kT_2, \text{кеВ}$	27.5	28

1.2.4 Струми частинок плазми на поверхні незарядженого тіла

На поверхню тіла, внесеного в плазму, надходять потоки електронів та іонів, зумовлені тепловим рухом частинок. При однаковій енергії електронів та іонів, яка визначається температурою плазми, електрони мають значно більш високу швидкість в порівнянні з іонами через різницю мас частинок. Тому спочатку, поки внесене в плазму тіло не заряджена, потік електронів, що падає на поверхню, перевищує потік позитивних іонів, і тіло заряджається негативно. Далі надходження заряджених частинок на поверхню відбувається в умовах дії на них електричного поля, яке по відношенню до електронів є гальмуючим, а по відношенню до позитивних іонів – прискорюючим. Це в результаті призводить до рівності потоків електронів та іонів при деякому негативному потенціалі поверхні. Такий простий випадок зарядження тіла в двокомпонентній плазмі розглядається в теорії плазмового зонда, відомого у фізиці як зонд Ленгмюра.

Як уже зазначалося, якщо внесене в плазму тіло не заряджена, тобто знаходиться при потенціалі плазми, струми, що течуть на поверхні, обумовлені тільки тепловим рухом частинок.

В цьому випадку щільність струму частинок плазми одного виду з концентрацією n визначається виразом

$$j = -en \int (sv) f(v) dv,$$

де s – нормаль до поверхні, що розглядається, а добуток (sv) – нормальна складова швидкості. Інтегрування ведеться по зовнішній відносно поверхні півсфері.

Для незарядженої поверхні при максвелівському розподілі частинок за швидкостями отримаємо

$$j_0 = en \left(\frac{kT}{2\pi m} \right)^{\frac{1}{2}}. \quad (2)$$

У ізотермічній електронно-протонній плазмі з однаковою концентрацією часток ($T_e = T_p, n_e = n_p$) відношення щільності електронного струму на незарядженій поверхні до щільності протонного струму визначається виразом [?]

$$\frac{j_{e0}}{j_{p0}} = \left(\frac{m_p}{m_e} \right)^{\frac{1}{2}} = \sqrt{1836} \approx 43.$$

1.3 Метод Монте-Карло

Прояв методів статистичного моделювання (Монте-Карло) в різних областях прикладної математики, як правило, пов'язаний з необхідністю вирішення якісно нових завдань, що виникають з потреб практики. Так було при створенні атомної зброї, на першому етапі освоєння космосу, дослідженні явищ атмосферної оптики, фізичної хімії, моделюванні турбулентності. В якості одного з більш-менш вдалих визначень методів Монте-Карло можна привести наступне:

Методи Монте-Карло – це чисельні методи розв'язання математичних задач (систем алгебраїчних, диференціальних, інтегральних рівнянь) і пряме статистичне моделювання (фізичних, хімічних, біологічних, економічних, соціальних процесів) за допомогою отримання і перетворення випадкових чисел.

Перша робота по використанню методу Монте-Карло була опублікована Холлом в 1873 році саме при організації стохастичного процесу експериментального визначення числа шляхом кидання голки на лист лінійованого паперу. Яскравий приклад використання методів Монте-Карло – використання ідеї Дж. фон Неймана при моделюванні траєкторій нейтронів в лабораторії Лос Аламоса в сорокових роках минулого століття. Хоча методи Монте-Карло пов'язані з великою кількістю обчислень, відсутність електронної обчислювальної техніки ні в тому ні в іншому випадку не збен-

тежила дослідників при застосуванні цих методів, оскільки в тому і в іншому випадку мова йшла про моделювання випадкових процесів. І свою назву вони отримали по імені столиці князівства Монако, знаменитої своїми гральними домами, основу яких складає рулетка – досконалий інструмент для отримання випадкових чисел. А перша робота, де це питання викладався систематично, опублікована в 1949 році Метрополіс і Уламом [?], де метод Монте-Карло застосовувався для вирішення лінійних інтегральних рівнянь, де явно вгадувалось завдання про проходження нейтронів через речовину. [?].

У 1950-х роках метод використовувався для розрахунків при розробці водневої бомби. Основні заслуги у розвитку методу в цей час належать співробітникам лабораторій ВПС США і корпорації RAND.

У 1970-х роках в новій галузі математики – теорії обчислювальної складності було показано, що існує клас задач, складність (кількість обчислень, необхідних для отримання точної відповіді) яких зростає з розмірністю задачі експоненційно. Іноді можна, пожертвувавши точністю, знайти алгоритм, складність якого зростає повільніше, але є велика кількість задач, для яких цього не можна зробити (наприклад, задача визначення обсягу опуклого тіла в n -мірному евклідовому просторі) і метод Монте-Карло є єдиною можливістю для отримання достатньо точної відповіді за прийнятний час.

В даний час основні зусилля дослідників спрямовані на створення ефективних Монте-Карло алгоритмів різних фізичних, хімічних і соціальних процесів для паралельних обчислювальних систем.

1.3.1 Пряме моделювання методом Монте-Карло

Пряме моделювання методом Монте-Карло деякого фізичного процесу передбачає моделювання поведінки окремих елементарних частин фізичної системи. По суті це пряме моделювання близьке до вирішення завдання з перших принципів, проте зазвичай для прискорення розрахунків допускається застосування деяких фізичних наближень. Прикладом можуть служити розрахунки різних процесів методом молекулярної динаміки: з одного боку система описується через поведінку її елементарних складо-

вих частин, з іншого боку, використаний потенціал взаємодії найчастіше є емпіричним.

Приклади прямого моделювання методом Монте-Карло:

- Моделювання опромінення твердих тіл іонами в наближенні бінарних зіткнень.
- Пряме Монте-Карло моделювання розріджених газів.

Більшість кінетичних Монте-Карло моделей належать до прямих (зокрема, дослідження молекулярно-пучкової епітаксії).

2 Математична модель

Для моделювання руху і зіткнень частинок та космічного апарату було введено низку структур даних, що представляють як геометричні абстракції (точка, пряма, площина, вектор, сфера), так і реальні об'єкти (елементарна частинка, полігональний тривимірний об'єкт). Для роботи з цими структурами було, окрім методів самих структур, розроблено модуль з окремими функціями – їх опис, а також опис структур даних подано нижче.

2.1 Опис структур даних

Для представлення чисел з плаваючою крапкою введено тип `real`, який є альтернативним іменем для типу `float` (одинарна точність), а при недостатці точності чи інших потребах може бути легко замінений на `double` (подвійна точність).

2.1.1 Point

Тип потрібен для представлення точки в тривимірному просторі – в кожному об'єкті зберігаються три координати типу `float`. Також наявні методи для порівняння з іншими об'єктами цього ж типу і методи для додавання або віднімання вектора (виконується зсув точки на заданий вектор).

2.1.2 Vector

Тип введено для представлення вектора в тривимірному просторі. Клас `Vector` успадковано від класу `Point`, оскільки вектор теж однозначно задається трьома координатами – при необхідності може перетворений до батьківського класу. Серед методів можна назвати порівняння з об'єктами цього ж типу, множення вектора на константу і на вектор (в наявності як скалярний добуток, так і векторний, знаходження суми і різниці векторів, довжини вектора, косинуса кута між векторами, а також нормалізація вектора і приведення його до заданої довжини.

2.1.3 Locus

Базовий шаблонний клас для всіх підкласів, які представляють собою геометричний об'єкт, що задається набором точок. В якості параметра класу виступає кількість точок. Об'єкт класу містить лише масив заданої параметром довжини і має метод для виводу цього масиву в зручній для сприйняття формі. В перспективі до класу можуть бути додані методи, що працюють відразу з усіма точками, незалежно від їх кількості (наприклад, афінні перетворення).

2.1.4 Line

Клас успадковано від Locus з параметром 2, тобто містить в собі дві точки, а також для зручності направляючий вектор, котрий обчислюється при конструюванні об'єкта. Серед методів можна відмітити метод отримання точки на прямій за коефіцієнтом, яким ця точка визначається в параметричних рівняннях прямої.

2.1.5 ThreePoints

Базовий клас для всіх класів, що представляють об'єкти, які можуть бути задані трьома точками (трикутник, площа, орієнтована площа). Клас успадковано від Locus з параметром 3, тобто містить в собі три точки. Окрім оператора присвоювання має ще метод для визначення нормалі (за допомогою векторного добутку векторів, утворених двома різними парами точок), а також методи для знаходження центру мас і площі трикутника, утвореного цими трьома точками (при умові, що три точки не лежать на одній прямій).

2.1.6 Plane

Клас для представлення площини, успадкований від класу ThreePoints. Додано метод для визначення чи належить точка даній площині.

2.1.7 OrientedPlane

Клас для представлення орієнтованої площини, успадкований від класу Plane. Містить вектор нормалі, який тепер повертається перевантаже-

ним метод отримання нормалі, який було визначено в класі `ThreePoints`. Нормаль обчислюється при конструюванні об'єкта як векторний добуток двох векторів: один з початком в першій точці, кінцем в другій, інший з початком в першій точці, кінцем в третій – тобто перший, другий вектори і нормаль утворюють праву трійку векторів; іншими словами з кінця нормалі, початок якої лежить в площині, видно три точки, якими задається площина, в порядку руху годинникової стрілки. Конструктор класу, що описується, приймає також логічний параметр, який визначає, чи буде видно точки в напрямі руху годинникової стрілки (за замовчуванням цей параметр істинний).

2.1.8 Particle

Клас, що описує елементарну частинку. Є успадкованим від класу `Point`. Додано поля для збереження вектора руху, а також поле типу `real`, що визначає час існування частинки в секундах (якщо значення від'ємне, час вважається не заданим).

2.1.9 Sphere

Клас для представлення сфери. Має поля для збереження точки – центру сфери, а також числа типу `real` – радіуса сфери.

2.1.10 Object3D

Клас, призначений для збереження координат полігонів тривимірних тіл. При конструюванні кожного об'єкта обчислюються і пишуться у дві відповідні точки (об'єкти `Point`) максимальні та мінімальні координати тіла за всіма осями. Клас успадковано від класу `Sphere` – центр шукається як середина відрізка, що сполучає дві вищеописані точки, радіус – як половина довжини цього відрізка; таким чином виходить, що ці параметри задають сферу, описану навколо тіла. Це наслідування є корисним при перевірці, чи перетинає траєкторія частинки тіло – спочатку перевіряється, чи перетинає пряма траєкторія сферу, якщо так – виконується перевірка для кожного полігону циклічно. Як параметр конструктора об'єкта приймається також вектор, що задає напрям руху тіла (за замовчуванням співпадає з віссю

ОХ). Також в класі визначено метод для пошуку повної поверхні тіла – вона обчислюється як сума площ всіх полігонів, з яких це тіло складається.

2.1.11 GenerativeSphere

Клас, призначений для генерації елементарних частинок. Є нащадком класу Sphere. По суті являє собою сферу, центр якої співпадає з центром тіла, а радіус має перевищувати радіус тіла. Полями класу є генератори випадкових чисел, за допомогою яких для кожної частинки генерується випадковим чином швидкість (один генератор для кожного типу частинок – іонів та електронів), а також посилання на об'єкт, що представляє собою власне тіло (типу Object3D). В класі наявні методи для генерації частинок наступним чином:

1. частинки, що в початковий момент часу свого існування лежать всередині сфери і траєкторія яких не обов'язково перетинає тіло;
2. частинки, що в початковий момент часу свого існування лежать на сфері і траєкторія яких перетинає тіло;

2.2 Опис геометричних функцій

2.2.1 Перевірка, чи знаходиться точка всередині трикутника

Перевірка виконується для точки, що знаходиться в площині трикутника.

Спосіб 1

Виконується перевірка, чи не співпадає точка з однією з вершин трикутника. Після цього шукаються кути між всіма парами векторів, початок яких знаходиться в даній точці, а кінець співпадає з вершиною трикутника. Очевидно, що якщо всі кути будуть тупими, тобто всі косинуси від'ємними, то точка лежатиме всередині трикутника. Також очевидно, що якщо три або два кути виявляться гострими, то точка лежить за межами трикутника. Інакше за допомогою стандартної функції знаходження арккосинуса шукаємо суму всіх кутів. Неважко побачити, що для точок, які не належать трикутнику, ця сума буде менше за 2π .

Спосіб 2

Для кожної пари вершин трикутника виконаємо перевірку: проведемо через них пряму і визначимо, чи лежить точка і третя вершина в одній і тій самій півплощині відносно цієї прямої. Для цього знайдемо проекцію третьої вершини на пряму і косинус кута, вершиною якого є дана проекція, а сторони проходять через задану точку і третю вершину трикутника. Якщо синус від’ємний, то кут є тупим, тобто точка і третя вершина трикутника лежить в різних півплощинах – перевірка не виконалась. Якщо для кожної пари вершин ця перевірка виконається, то точка лежатиме всередині трикутника. Інакше – ні.

2.2.2 Пошук точки перетину прямої і площини

Як відомо, пряма в просторі (тут і далі мається на увазі тривимірний Евклідов простір) може бути задана трьома параметричними рівняннями

$$\begin{cases} x = A_x - k(B_x - A_x) \\ y = A_y - k(B_y - A_y) \\ z = A_z - k(B_z - A_z) \end{cases}, \quad (3)$$

де A, B – точки, через які проведено пряму. В цьому випадку кожна її точка задається єдиним значенням коефіцієнта. В обох описаних нижче способах ми знаходимо коефіцієнт точки перетину прямої і площини.

Спосіб 1

В канонічне рівняння площини, проведеної через 3 точки A, B і C

$$\begin{vmatrix} x - A_x & y - A_y & z - A_z \\ B_x - A_x & B_y - A_y & B_z - A_z \\ C_x - A_x & C_y - A_y & C_z - A_z \end{vmatrix} = 0 \quad (4)$$

підставимо координати з рівнянь ???. Отримаємо лінійне рівняння відносно коефіцієнта k . Розв’язавши його, отримаємо коефіцієнт шуканої точки перетину.

Спосіб 2

У відоме векторне рівняння площини

$$\bar{n} \cdot \overline{AX} = 0 \quad (5)$$

(де \bar{n} – нормаль площини, P – задана точка на площині, а X – довільна точка площини) підставляємо замість X точку з координатами, взятими з параметричного рівняння прямої ???. Отримаємо лінійне рівняння відносно коефіцієнта k . Розв'язавши його, отримаємо коефіцієнт шуканої точки перетину.

2.2.3 Пошук проекцій

Пошук проекції точки на пряму

Щоб точка на прямій була проекцією заданої точки, необхідно виконання двох умов:

1. Її координати мають задовольняти рівняння прямої ??;
2. Вектор з точки до її проекції має бути перпендикулярним направляючому вектору прямої: $\overline{PP'} \cdot \bar{n} = 0$.

Підставляючи замість координат проекції P' координати з рівнянь ??, отримуємо коефіцієнт точки перетину.

Пошук проекції точки на площину

Шукана точка – точка перетину прямої (направляючий вектор якої дорівнює нормалі площини; пряма проходить через задану точку) та заданої площини. Пошук перетину прямої і площини описаний в ??.

2.2.4 Пошук відстаней

Відстань між двома точками

Знаходиться за відомою формулою

$$\rho(x, y) = \sqrt{\sum_{i=1}^3 (x_i - y_i)^2} \quad (6)$$

Відстань між точкою та площиною

Знаходимо проекцію точки на площину і підставляємо в формулу ??.

2.2.5 Перевірка, чи перетинає пряма сферу

Неважко побачити, що у випадку, коли пряма перетинає сферу, проекція центру сфери на пряму не буде лежати зовні сфери. Тобто, необхідно лише знайти відстань (див. ??) між центром сфери і його проекцією (див. ??) на задану пряму та переконатись, що ця відстань не перевищує радіус сфери.

2.2.6 Поворот точки навколо прямої

Знайдемо проекцію точки на пряму – точку P' . Введемо двовимірну систему координат: одиничний вектор (орт) осі абсцис \bar{i} співпадатиме з вектором $\overline{P'P}$, а для осі ординат \bar{j} буде одночасно перпендикулярним до осі абсцис та направляючого вектора прямої (знайдемо його як векторний добуток направляючого вектора і першої осі); також вісь ординат нормалізуємо і домножимо на довжину осі абсцис, щоб система була декартовою. Очевидно, що шукана точка знаходиться саме в отриманій площині. Як відомо, на площині координати повороту точки навколо початку координат задаються наступною формулою:

$$\begin{cases} x' = x \cdot \cos\alpha - y \cdot \sin\alpha \\ y' = x \cdot \sin\alpha + y \cdot \cos\alpha \end{cases}$$

Також очевидно, що початкова точка в новій системі матиме координати (0;1), отже формула повороту для неї перепишеться як

$$\begin{cases} x' = -y \cdot \sin\alpha \\ y' = y \cdot \cos\alpha \end{cases}$$

Тепер залишилось перейти від двовимірних координат назад до трьохмірних – для цього необхідно до координат точки P' додати зміщення $x'\bar{i} + y'\bar{j}$.

2.2.7 Геометричні функції, в яких використовуються випадкові величини

Отримання випадкової точки всередині сфери

Обираємо випадковий вектор (кожна з його координат являє собою рівномірно розподілену випадкову величину з проміжку $[-0.5, 0.5]$), нормалізуємо, домножуємо на випадкову величину, що приймає значення $[0, R]$ (R – радіус сфери) і додаємо отриманий вектор до центру сфери. Випадкову величину, на яку домножується вектор, обираємо $\sqrt{R * \xi}$, де ξ – рівномірно розподілена випадкова величина з проміжку $[0, R]$. Це робиться для того, щоб отримані таким чином випадкові точки були рівномірно розподілені всередині сфери, а не розташовувались більш щільно до центру.

Отримання випадкової точки, що лежить на сфері

Обираємо випадковий вектор (кожна з його координат являє собою рівномірно розподілену випадкову величину з проміжку $[-0.5, 0.5]$), нормалізуємо, домножуємо на радіус сфери і додаємо отриманий вектор до центру сфери.

2.3 Опис типів даних і функцій для роботи з часом і випадковими величинами

Для генерації випадкових величин, що мають визначені розподіли з заданими параметрами, було написано декілька класів-обгорток навколо класів стандартної бібліотеки мови C++.

2.3.1 UniformDistributionGenerator

Клас, що представляє рівномірний розподіл. За допомогою функції `getUniformDistributionGenerator`, яка приймає в якості параметрів максимальне і мінімальне можливі значення випадкової величини, можна отримати об'єкт цього класу з заданими параметрами. Також наявна функція `getRandom`, яка не має аргументів і повертає рівномірно розподілену величину з проміжку $[0, 1]$.

2.3.2 GaussianDistributionGenerator

Клас, що представляє нормальний розподіл. За допомогою функції `getGaussianDistributionGenerator`, яка приймає в якості параметрів математичне сподівання і дисперсію випадкової величини, можна отримати об'єкт цього класу з заданими параметрами. Для цього і попереднього класу визначено єдиний метод – оператор "круглі дужки при виклику якого повертається щойно-згенероване випадкове число.

2.3.3 MaxwellDistributionSpeedGenerator

Аліас для типу "функція, що повертає значення типу `float` і не приймає жодного параметра" (`function<float ()>`). За допомогою функції `getMaxwellDistributionSpeedGenerator`, яка приймає в якості параметрів математичне сподівання і дисперсію випадкової величини, є можливість отримати об'єкт типу `MaxwellDistributionSpeedGenerator`. Оскільки випадкову величину, розподіл за максвелівським розподілом за модулем швидкостей, її отримати з трьох нормально розподілених випадкових величин ($\sqrt{x_1^2 + x_2^2 + x_3^2}$), для генерації яких в тілі вищезгаданої функції використовується статичний об'єкт `GaussianDistributionGenerator`.

2.3.4 Функції для роботи з часом

При необхідності визначити час, за який виконалась та чи інша операція, використовується POSIX функція `clock_gettime`, яка зберігає в передачу структуру час вказаного таймера з точністю до наносекунд. В програмі використовується таймер, що задається константою `CLOCK_THREAD_CPUTIME_ID` і якому відповідає таймер центрального процесора, специфічний для даного потоку. Для роботи зі структурою, в яку записує результат вищезгадана функція, в програмі наявні функції `printTimespec` і `getTimespecDelta` – для зручного виводу і знаходження різниці в часі двох структур відповідно.

2.4 Опис функцій для роботи з масивами даних

Функції для роботи з даними є шаблонними. Вони потребують для об'єкту даних лише його розмір і наявність оператора "квадратні дужки" для

отримання елемента за індексом, а отже можуть працювати зі звичайними масивами, які відомі ще з мови С. В цьому відмінність даних функцій від стандартних алгоритмів `stl`, які працюють з ітераторами.

2.4.1 reduce

Функція приймає в якості аргументів функцію, масив з даними та його розмір. Функція, що є першим аргументом, спочатку застосовується до перших двох елементів масиву, потім до результату і третього елементу, і так далі. Дійшовши до останнього елементу, отримане значення повертається як результат роботи функції.

2.4.2 map

Функція приймає в якості аргументів функцію, масив з даними та його розмір. Функція, що є першим аргументом, має приймати в якості єдиного аргументу посилання на той тип даних, з елементів якого складається масив. Ця функція циклічно викликається для кожного елементу масива, змінюючи, якщо потрібно, його значення.

2.5 функції для роботи з файлами

Тіло (космічний апарат) для роботи програми має уявляти собою набір трикутних орієнтованих полігонів. Для зчитування даних про тіло з файлу в програмі наявні описані далі функції.

2.5.1 getCoordinatesFromPlainFile

Функція приймає в якості єдиного аргументу масив символів, що представляє собою назву вхідного файлу. Координати полігонів в файлі мають розміщатись по 9 на строку, тобто 3 точки або один полігон. Розділені між собою координати мають бути пробілами або знаками табуляції. Рядки, з яких не вдається прочитати описані дані, пропускаються.

2.5.2 getCoordinatesFromSpecialFile

Функція приймає в якості єдиного аргументу масив символів, що представляє собою назву вхідного файлу. Ця функція використовує бібліотеку

ASSIMP, що позиціюється як бібліотека для завантаження і обробки різноманітних форматів даних геометричних сцен, і переводить дані, отримані за допомогою функцій бібліотеки у внутрішній формат програми.

На офіційному сайті бібліотеки зазначено підтримку наступних форматів даних [?]:

- textbfCollada (*.dae;*.xml)
- textbfBlender (*.blend) 3
- textbfBiovision BVH (*.bvh)
- textbf3D Studio Max 3DS (*.3ds)
- textbf3D Studio Max ASE (*.ase)
- textbfWavefront Object (*.obj)
- textbfStanford Polygon Library (*.ply)
- textbfAutoCAD DXF (*.dxf)
- textbfNeutral File Format (*.nff)
- textbfSense8 WorldToolkit (*.nff)
- textbfValve Model (*.smd,*.vta)
- textbfQuake I (*.mdl)
- textbfQuake II (*.md2)
- textbfQuake III (*.md3)
- textbfQuake 3 BSP (*.pk3)
- textbfRtCW (*.mdc)
- textbfDoom 3 (*.md5mesh;*.md5anim;*.md5camera)
- textbfDirectX X (*.x).
- textbfQuick3D (*.q3o;*.q3s).

- textbfRaw Triangles (*.raw).
- textbfAC3D (*.ac).
- textbfStereolithography (*.stl).
- textbfAutodesk DXF (*.dxf).
- textbfIrrlicht Mesh (*.irmesh;*.xml).
- textbfIrrlicht Scene (*.irr;*.xml).
- textbfObject File Format (*.off).
- textbfTerragen Terrain (*.ter)
- textbf3D GameStudio Model (*.mdl)
- textbf3D GameStudio Terrain (*.hmp)
- textbfOgre (*.mesh.xml, *.skeleton.xml, *.material)3
- textbfMilkshape 3D (*.ms3d)
- textbfLightWave Model (*.lwo)
- textbfLightWave Scene (*.lws)
- textbfModo Model (*.lxo)
- textbfCharacterStudio Motion (*.csm)
- textbfStanford Ply (*.ply)
- textbfTrueSpace (*.cob, *.scn)

Також в модулі з функціями для обробки файлів присутня змінна, що задає масштаб (за замовчуванням дорівнює одиниці) – наприклад, на випадок, якщо розміри тіла вимірюються в міліметрах.

3 Математичний і програмний опис алгоритму

3.1 Опис розв'язання

В даній роботі проводиться пряме моделювання методом Монте-Карло. При такому підході виконується моделювання окремих частин фізичної системи, для прискорення розрахунків допускається застосування деяких фізичних наближень. В нашому випадку елементи фізичної системи – це штучний супутник і велика кількість елементарних частинок, для яких реалізується стохастичний процес їх руху і зіткнення із літальним апаратом.

Для моделювання електризації були взяті характеристики навколишнього середовища геостаціонарної орбіти [?]:

- модель плазми проста Максвелівська:
- Стан системи рівноважний, тому розподіл частинок плазми за абсолютним значенням швидкості відбувається за розподілом Максвела, функція розподілу якого має вигляд

$$F(v) = \left(\frac{m}{2\pi kT} \right)^{\frac{3}{2}} e^{-\frac{mv^2}{2kT}} 4\pi v^2$$

де v – швидкість частинки, m – маса частинки, $k = 1.38 \cdot 10^{-23}$ Дж/к – постійна Больца, T – температура в градусах Кельвіна.

- зовнішні поля відсутні. Доведення: на частинки плазми діє магнітне поле Землі, яке змушує їх рухатись по кільцевим траєкторіям з т.зв. Ларморівським радіусом. Але цим полем можна знехтувати, так як Ларморівський радіус для електронів і іонів (протонів) високотемпературної плазми значно перевищує характерний розмір ШСЗ:

$$R_{le} = \frac{v_e \perp m_e}{eB} = \frac{3.37\sqrt{E_e}}{B} = \frac{3.37\sqrt{0.4 \cdot 10^3}}{3.1 \cdot 10^{-5}} = 2.1 \cdot 10^6 m \quad (7)$$

$$R_{li} = \frac{v_i \perp m_i}{eB} = \frac{145\sqrt{AE_i}}{B} = \frac{145\sqrt{1 \cdot 0.2 \cdot 10^3}}{3.1 \cdot 10^{-5}} = 6.6 \cdot 10^7 m \quad (8)$$

де A – заряд іона.

- рух частинок системи відбувається за напрямками швидкості і є рівномірним.

– частинки рухаються без зіткнень між собою, так як довжина вільного пробігу на геостаціонарній орбіті більша за характерні розміри ШСЗ: на висоті 300 км дорівнює декільком кілометрам.

Отже, рух частинок розглядається як рівномірний прямолінійний, а зіткнення частинок між собою не враховуються, оскільки не мають значення для розв’язання задачі і не впливають на результати.

Тіло (космічний апарат) в програмі представляється як об’єкт типу Object3D. При цьому найважливішим полем цього об’єкту є поле polygons, що являє собою масив об’єктів-полігонів типу OrientedPlane – об’єкти цього типу відрізняються від об’єктів класу Plane тим, що зберігають для в собі вектор нормалі до площини, яку представляють. Нормаль необхідна полігонам для чіткого визначення зовнішньої і внутрішньої сторони.

Для генерації частинок використовується декілька об’єктів типу GenerativeSphere – по одній сфері для кожного типу елементарних частинок. Сфера містить генератори типу MaxwellDistributionSpeedGenerator, за допомогою яких генерується швидкості елементарних частинок. Також у сфери є метод populateArray для заповнення масиву частинок.

За формулою ?? для обчислення щільності струму отримаємо струм, що проходить через одиницю площі поверхні генеруючої сфери. При цьому значення температури будемо брати з таблиці ?. Отримаємо наступну щільність для електронного і протонного струму відповідно:

$$1.6 \cdot 10^{-19} \cdot 1.2 \cdot 10^6 \cdot \sqrt{\frac{27.5 \cdot 10^3 \cdot 1.6 \cdot 10^{-19}}{2 \cdot \pi \cdot 9.1093829140 \cdot 10^{-31}}} = -0.000005334 A/m^2$$

$$1.6 \cdot 10^{-19} \cdot 1.3 \cdot 10^6 \cdot \sqrt{\frac{28 \cdot 10^3 \cdot 1.6 \cdot 10^{-19}}{2 \cdot \pi \cdot 1.67 \cdot 10^{-27}}} = 0.000000136 A/m^2$$

Знаючи радіус генеруючої сфери, маємо можливість знайти сумарний струм, що проходить через її поверхню. Далі для кожної елементарної частинки будемо розглядати її траєкторію як частину загального струму. Для цього обчислені раніше загальний протонний і електронний струм розділимо на кількість протонів і електронів нашої моделі.

Далі починаємо моделювання – циклічно змінюємо траєкторії частинок згідно їх вектору швидкості, але щоб переміщення на один крок було

співвідносним з розмірами космічного апарату. Якщо частинка покинула межі генеруючої сфери, видаляємо її з масиву і на її місце генеруємо нову. Якщо ж частинка зіштовхнулася з космічним апаратом, змінюємо сумарний струм плазми на поверхні апарату на величину, яку позначає одна траєкторія. При цьому для електронів ця величина буде від'ємною, а для протонів – додатною.

Дебаєвський радіус (радіус генеруючих сфер) обчислимо за формулою ??:

$$\sqrt{\frac{8.8541 \cdot 10^{-12} \cdot 0.25 \cdot 10^3 \cdot 1.6 \cdot 10^{-19}}{0.8 \cdot 10^6 \cdot (1.6 \cdot 10^{-19})^2}} = 131.414769518$$

(дані взято з таблиці ??).

В кожний момент часу ми можемо знайти заряд на поверхні космічного апарату за формулою

$$Q = I \cdot t,$$

де t – час, що пройшов з початку спостереження, I – сумарний струм плазми на поверхні космічного апарату.

Також ми можемо знайти потенціал космічного апарату за формулою

$$U = \frac{Q}{C},$$

де C – електрична ємність космічного апарату. В розрахунках беремо її приблизне значення, що дорівнює електричній ємності сфери з тією ж площею поверхні (формула для сфери: $C = 4\pi\epsilon_0 R$).

При цьому система має такі параметри: n – кількість частинок, присутніх в системі в кожну одиницю часу, R – радіус сфери, на якій будуть генеруватись частинки (між n і R існує залежність, оскільки кількість частинок визначається об'ємом сфери, всередині якої вони знаходяться) і Δt – часовий крок.

3.2 Опис інтерфейсу програми

Параметри запуску програми наступні:

```
program [-m][-v][-d][-x][-t NUMBER][-r RADIUS][-s TIME][-n N][-f SF][-i INTERVAL][-p STEP] <filename>
```

Тут:

-s TIME – час паузи в мікросекундах після кожної ітерації; за замовчуванням 0

-m – індикатор, що вказує на необхідність моделювати частинки (без нього буде виводитись або лише модель апарату, або взагалі нічого, залежно від інших опцій)

-v – режим більш детального виводу

-d – індикатор, що вказує на необхідність малювати (використовуючи виклики бібліотеки OpenGL) космічний апарат і елементарні частинки (якщо заданий індикатор -m)

-f SF – коефіцієнт масштабу для файлу даних; за замовчуванням 1 (якщо, наприклад, координати задано в міліметрах, необхідно виставити даний параметр 0.001)

-n N – загальна кількість частинок в кожний момент часу

-i INTERVAL – інтервал, з яким будуть виводитись дані спостережень (час, потенціал, заряд); якщо вказаний з префіксом 's', інтервал інтерпретується як час в секундах, з індексом 'i' – як кількість ітерацій.

-p STEP – крок однієї частинки (середньої), заданий в довжинах космічного апарату; за замовчуванням 0.25

-x - індикатор, який показує, що зміст файлу даних з даними слід інтерпретувати як складний формат даних (детальніше – у секції ??)

Програма через задані проміжки часу виводить в строку: заряд на поверхні космічного апарату, потенціал, час і кількість зіткнень частинок з КА.

3.3 Результати роботи програми

3.3.1 Зображення космічних апаратів

Побудова моделей апаратів з файлів даних [?].

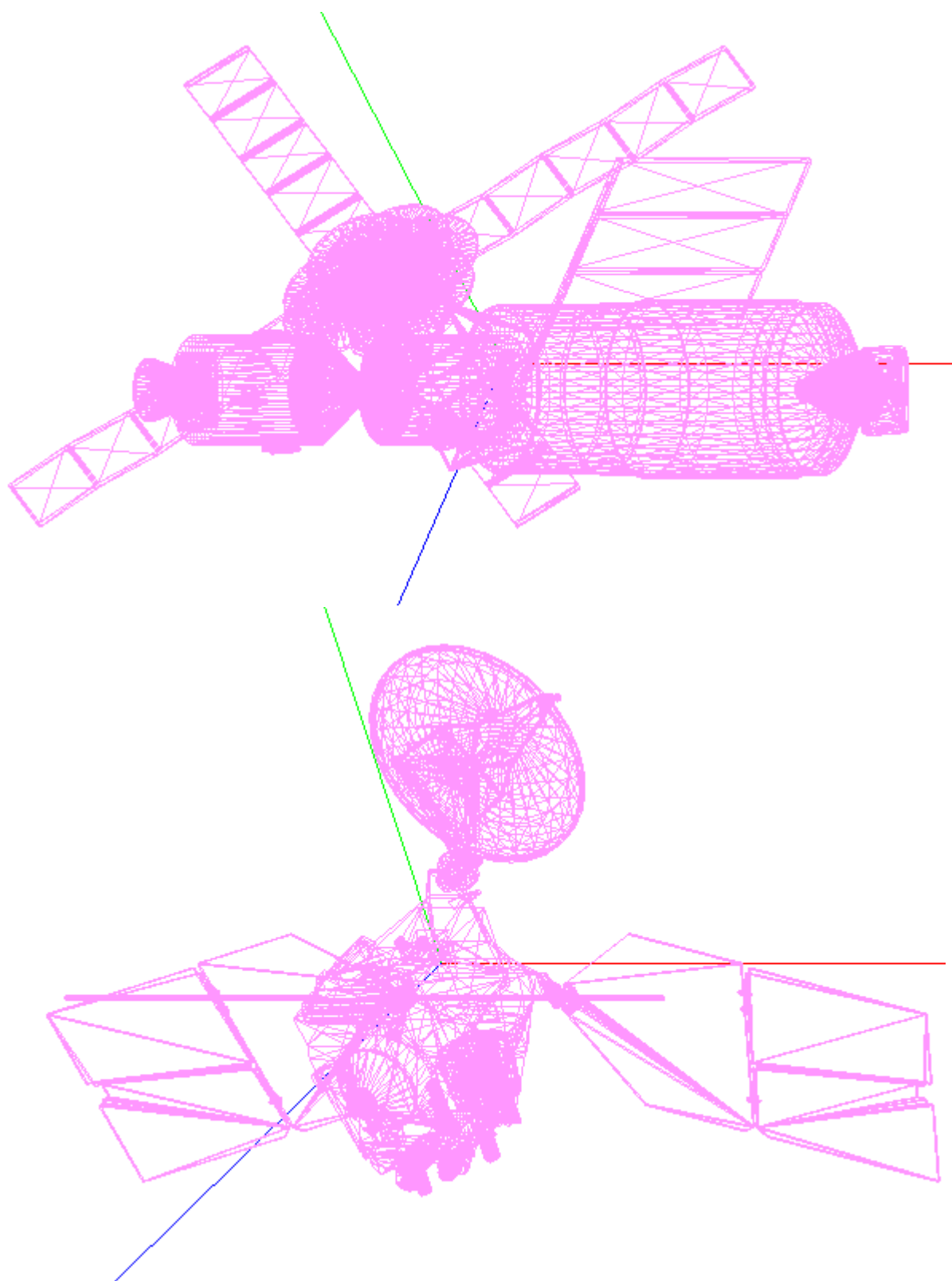


Рис. 1: Моделі космічних апаратів

3.3.2 Зображення частинок, що моделюються

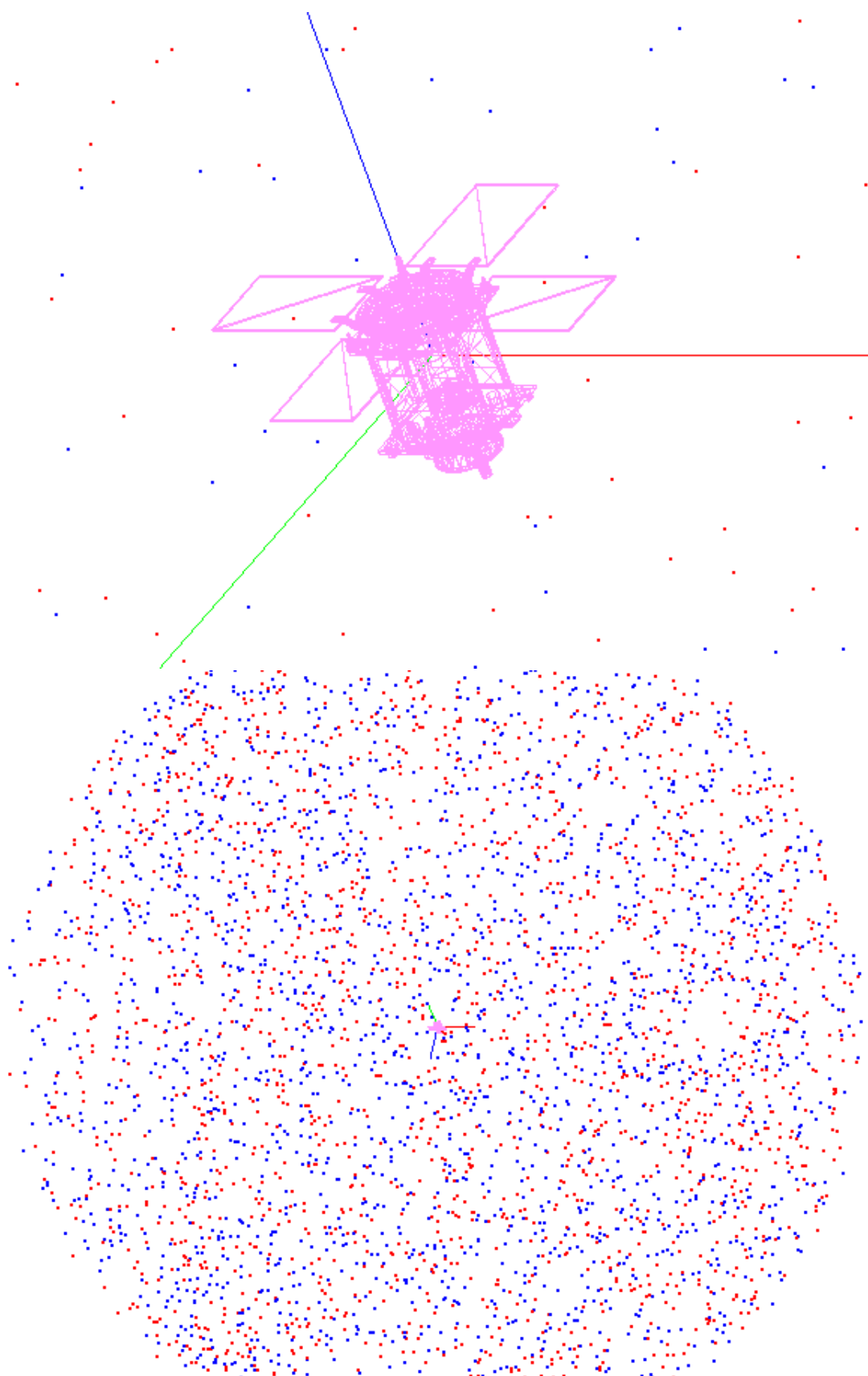


Рис. 2: Моделі космічних апаратів та елементарних частинок

На рисунку ?? зверху показано космічний апарат у наближенні. В нижній частині камеру віддалено достатньо для того, щоб побачити, що все моделювання відбувається в межах сфери.

Також слід зазначити, що сині точки позначають електрони, а червоні – іони.

3.3.3 Результати спостережень

Зміна заряду (за час 0.00025 секунди)

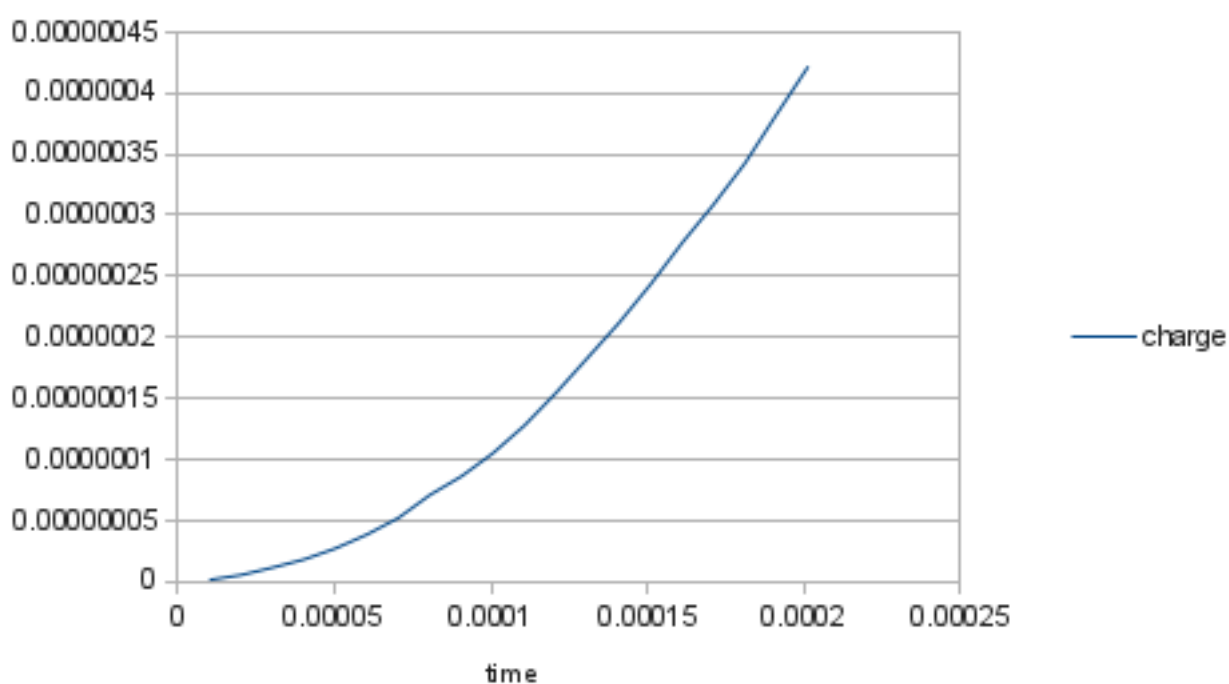


Рис. 3: Зміна заряду

Зміна потенціалу (за час 0.00025 секунди)

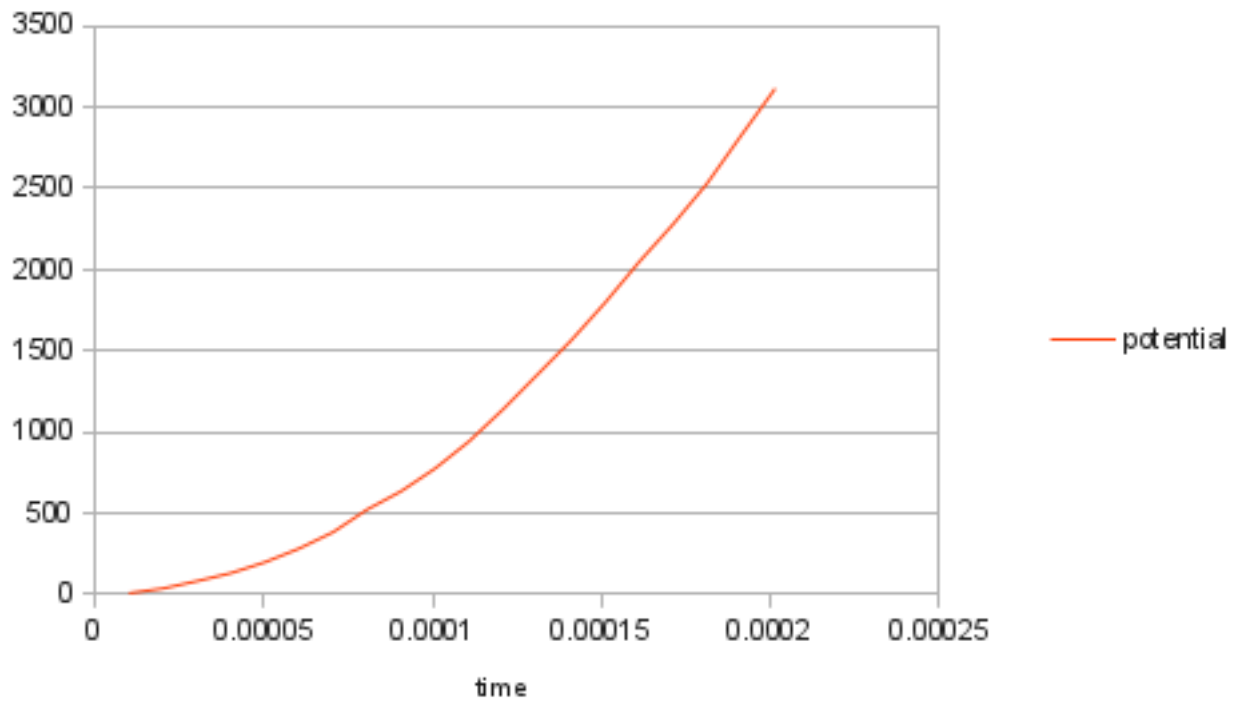


Рис. 4: Зміна потенціалу

Сумарна кількість зіткнень частинок (в цифрах реальної системи) з космічним апаратом (за час 0.00025 секунди)

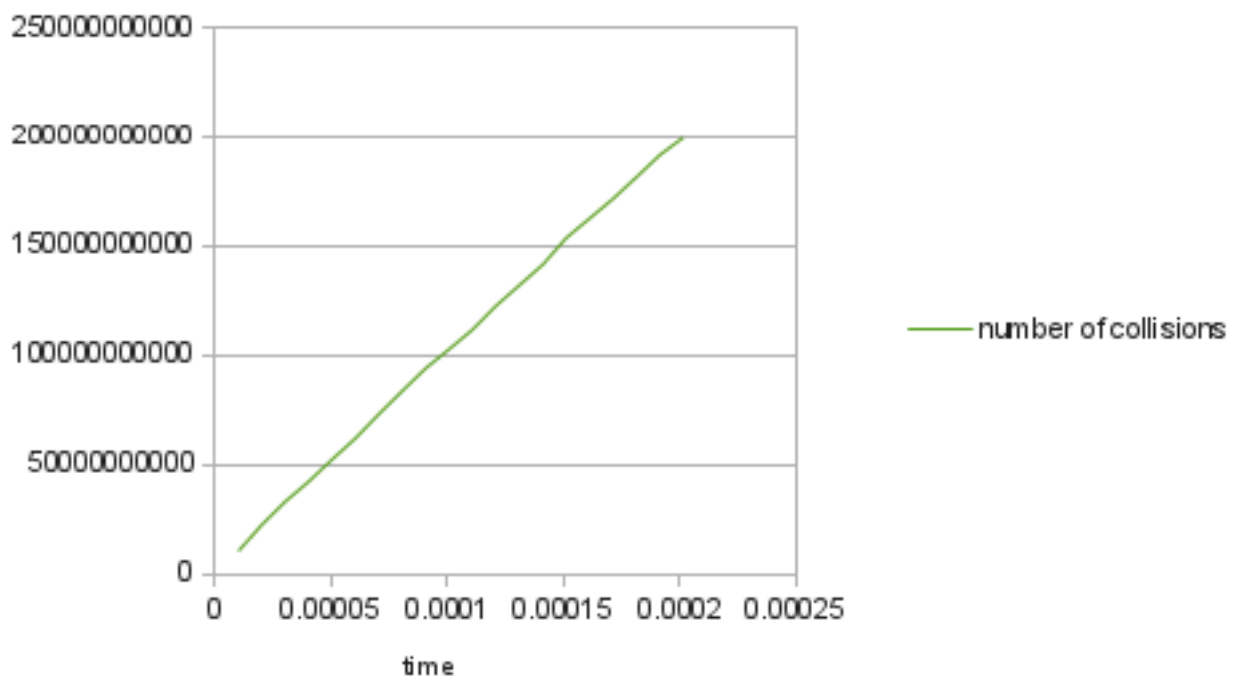


Рис. 5: Сумарна кількість зіткнень

Висновок

Було розроблено програмне забезпечення для моделювання руху літального апарату та зіткнення з ним елементарних частинок, що складають космічну плазму.

Отримана програма має зрозумілий інтерфейс командного рядка та формат вихідних даних. Завдяки використанню бібліотеки ASSIMP і власних функцій зчитування здатна працювати з вхідними файлами різноманітних форматів.

В програмі використано також наявні експериментальні дані, що були взяті з відповідної літератури.

За допомогою розробленого програмного забезпечення було проведено моделювання за статистичним методом Монте-Карло, було прослідковано зміну потенціалу космічного апарату і струмів на його поверхні з плином часу, а також кількість зіткнень елементарних частинок з космічним апаратом.

Також розроблена програма здатна візуалізувати процес моделювання, використовуючи графічну бібліотеку OpenGL та бібліотеку роботи з мультимедіа SDL.

Окрім того, розроблені в рамках програми модулі можуть використовуватись окремо від неї, оскільки кожен з них містить завершений функціонал деякого напрямку – це

- модуль з описом структур даних, що представляють геометричні примітиви і фізичні об'єкти
- модуль алгоритмів, що дозволяють змоделювати взаємодію цих об'єктів
- модуль з функціями для зручної генерації випадкових величин із заданим розподілом і параметрами
- невеликий, але зручний модуль обробки масивів даних
- модуль зчитування моделей апаратів і збереження їх в описану в програмі структуру

- о модуль зображення частинок і полігональних об'єктів засобами OpenGL

З використанням розробленого програмного забезпечення було змодельовано і досліджено процес електризації космічних апаратів. На основі отриманих даних було побудовано графіки, що відображають залежність різних параметрів системи від часу.

На даному етапі було розглянуто дещо спрощену модель – струми частинок плазми на поверхні незарядженого тіла, але розроблене програмне забезпечення може слугувати базою для подальших досліджень – моделюванню руху частинок поблизу тіла, поверхня якого вже має деякий заряд.

Література

- [1] Ю.И. Вакулин, О.С. Графодатский, В.И. Гусельников, В.И. Дегтярев, Г.А. Жеребцов, Ш.Н. Исляев, А.А. Кочеев, О.И. Платонов, Г.В. Попов, Л.Л. Фрумин *Основные геофизические закономерности электризации геостационарных спутников связи «Горизонт»*
- [2] А. М. Капулкин, В. Г. Труш, Д. В. Красношайка: *Исследование плазменных нейтрализаторов для снятия электростатических зарядов с поверхности высокоорбитальных космических аппаратов*. ДНУ, 1994.
- [3] Sharp R.D., Shelley E.G., Johnson K.G., Paschmann G. *Preliminary Results of a Low Energy Particle Survey at Synchronous Altitude* JGR, 1970, Volume 75, P. 6092
- [4] DeForest S.E. *Spacecraft Charging at Synchronous Altitudes* JGR, 1972, Volume 77, P. 651-659
- [5] Новиков Л.С. *Взаимодействие космических аппаратов с окружающей плазмой* Учебное пособие. – М.: Университетская книга, 2006. – 120 с.
- [6] Metropolis N., Ulam S. *The Monte-Carlo method* J. Amer. Stat. Assos. 44, № 247, 1949
- [7] О.М. Белоцерковский, Ю.И. Хлопков: *Методы Монте-Карло в прикладной математике и вычислительной аэродинамике*.
- [8] http://assimp.sourceforge.net/lib_html/index.html *ASSIMP - Open Asset Import Library*
- [9] http://www.nasa.gov/multimedia/3d_resources/models.html *National Aeronautics and Space Administration*
- [10] И. М. Соболев: *Метод Монте-Карло*. «Наука», Москва, 1968.

Додаток

dr_program/constants.h

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#include <cmath>

#define EXIT_ERR(msg) { cerr << msg << "\nerrno: " << errno << endl; Graphics::quitGraphics(1); }
#define PRINTLN(arg) cout << arg << endl
#define PRINT(arg) cout << arg && cout.flush()
#define COUT(args) cout << args << endl

// Particle types
#define PTYPE_ELECTRON 1
#define PTYPE_ION 0

// charges
extern double ELECTRON_ELECTRIC_CHARGE;
extern double ION_ELECTRIC_CHARGE;

// charge to mass ratio
extern double ELECTRON_CHARGE_TO_MASS;
extern double ION_CHARGE_TO_MASS;

// current density
extern double ELECTRON_CURRENT_DENSITY;
extern double ION_CURRENT_DENSITY;

#define PARTICLE_CHARGE(_particle_type) \
    ((_particle_type == PTYPE_ELECTRON)? ELECTRON_ELECTRIC_CHARGE: ION_ELECTRIC_CHARGE)

#define PARTICLE_CHARGE_TO_MASS(_particle_type) \
    ((_particle_type == PTYPE_ELECTRON)? ELECTRON_CHARGE_TO_MASS: ION_CHARGE_TO_MASS)

#define PARTICLE_CURRENT_DENSITY(_particle_type) \
    ((_particle_type == PTYPE_ELECTRON)? ELECTRON_CURRENT_DENSITY: ION_CURRENT_DENSITY)

typedef double real;
typedef float velocity;

extern velocity ORBITAL_VELOCITY;
//extern velocity ION_VELOCITY;
//extern velocity ELECTRON_VELOCITY;

extern real ELECTRON_VELOCITY_M;
extern real ELECTRON_VELOCITY_D;
extern real ION_VELOCITY_M;
extern real ION_VELOCITY_D;

// Debye radius
extern real ELECTRONS_GENERATIVE_SPHERE_RADIUS;
extern real IONS_GENERATIVE_SPHERE_RADIUS;

// particles density
extern int ELECTRONS_CONSISTENCE;
extern int IONS_CONSISTENCE;

extern double VACUUM_PERMITTIVITY;
// e0 ~ 8.854187817620*10^-12 F/m

#endif // CONSTANTS_H
```

dr_program/constants.cpp

```
#include "constants.h"

velocity ORBITAL_VELOCITY = 7907.343098064;

real ELECTRONS_GENERATIVE_SPHERE_RADIUS = 131.414769518;
real IONS_GENERATIVE_SPHERE_RADIUS = ELECTRONS_GENERATIVE_SPHERE_RADIUS;

real ELECTRON_VELOCITY_M = 110901445.521404395; // 2*sqrt(2*1.6*10^-19*27.5*10^3/(9.11*10^-31*pi))
real ELECTRON_VELOCITY_D = 69497174.760350449; // sqrt(1.6*10^-19*27.5*10^3/(9.11*10^-31))
real ION_VELOCITY_M = 2613670.454744482; // 2*sqrt(2*1.6*10^-19*28*10^3/(1.67*10^-27*pi))
real ION_VELOCITY_D = 1637875.065607546; // sqrt(1.6*10^-19*28*10^3/(1.67*10^-27))
```

```

#ifndef DATA_UTILS_H
#define DATA_UTILS_H

// methods to be used on ranges of elements

namespace Data {
    // C - container type (vector or pointer to array)
    // T - type of values in container
    template <typename C, typename T>
    static T reduce(function <T& (T&,T&)> f, C container, int size) {
        T result = container[0];
        for(int i = 1; i < size; i++)
            result = f(result, container[i]);
        return result;
    }
}

#endif // DATA_UTILS_H

```

```
#ifndef READ_FILE_H
#define READ_FILE_H

#include <assimp/scene.h>
#include <assimp/Importer.hpp>
#include <assimp/postprocess.h>

#include <iostream>
#include <fstream>
#include <vector>
#include "types.h"

using namespace std;

typedef OrientedPlane PlaneType;

namespace File {
    extern float scaleFactor;

    vector<PlaneType>* getCoordinatesFromPlainFile(char*);
    vector<PlaneType>* getCoordinatesFromSpecialFile(char*);
}

#endif // READ_FILE_H
```

```
#include "file_utils.h"

float File::scaleFactor = 1; // by default coordinates are given in meters
```

```

vector<PlaneType>* File::getCoordinatesFromPlainFile(char *filename) {
    filebuf fb;
    if (!fb.open(filename, ios::in)) {
        cerr << "An error occurred while opening file" << endl;
        return NULL;
    }
    istream fileInputStream(&fb);
    vector<PlaneType>* coordinatesList = new vector<PlaneType>();
    ThreePoints *tempThreePoints;

    int i;
    while(!fileInputStream.eof()) {
        tempThreePoints = new ThreePoints();

        // read 3 points
        for (i = 0; i < 3; i++) {
            fileInputStream >> tempThreePoints->set[i].x
                           >> tempThreePoints->set[i].y
                           >> tempThreePoints->set[i].z;
            if (fileInputStream.fail()) {
                if (!fileInputStream.eof())
                    fileInputStream.clear();
                break;
            }
        }

        // if all values have been read successfully then push array to result list
        if (i == 3) {
            try {
                PlaneType *pt = new PlaneType(*tempThreePoints);
                coordinatesList->push_back(*pt);
                for (i = 0; i < 3; i++) {
                    coordinatesList->back().set[i].x *= scaleFactor;
                    coordinatesList->back().set[i].y *= scaleFactor;
                    coordinatesList->back().set[i].z *= scaleFactor;
                }
            } catch (ZeroNormal zn) {}

            // then delete it
            delete tempThreePoints;
            // skipping remaining characters in current string
            while (!fileInputStream.eof() && fileInputStream.get() != '\n');
        }

        fb.close();
        return coordinatesList;
    }
}

vector<PlaneType>* File::getCoordinatesFromSpecialFile(char *filename) {
    Assimp::Importer importer;
    aiScene *aiscene = (aiScene*)importer
        .ReadFile(filename, aiProcess_Triangulate|aiProcess_FixInfacingNormals|
            aiProcess_FindDegenerates
            |aiProcess_PreTransformVertices|aiProcess_OptimizeMeshes|
            aiProcess_FindInvalidData|aiProcess_RemoveRedundantMaterials);
    if (aiscene == NULL) {
        cerr << "An error occurred while opening file" << endl;
        return NULL;
    }
    vector<PlaneType>* coordinatesList = new vector<PlaneType>();
    ThreePoints *tempThreePoints;
    int failedNumber = 0;

    for(unsigned int i = 0; i < aiscene->mNumMeshes; ++i) {
        if (aiscene->mMeshes[i]->HasFaces()) {
            for(unsigned int j = 0; j < aiscene->mMeshes[i]->mNumFaces; ++j) {
                tempThreePoints = new ThreePoints();

                for(int k = 0; k < 3; k++) {
                    tempThreePoints->set[k].x = aiscene->mMeshes[i]->mVertices[ aiscene->
                        mMeshes[i]->mFaces[j].mIndices[k]].x;
                    tempThreePoints->set[k].y = aiscene->mMeshes[i]->mVertices[ aiscene->
                        mMeshes[i]->mFaces[j].mIndices[k]].y;
                    tempThreePoints->set[k].z = aiscene->mMeshes[i]->mVertices[ aiscene->
                        mMeshes[i]->mFaces[j].mIndices[k]].z;
                }

                try {
                    PlaneType *pt = new PlaneType(*tempThreePoints);

```

```

        coordinatesList->push_back(*pt);
        for (int g = 0; g < 3; g++) {
            coordinatesList->back().set[g].x *= scaleFactor;
            coordinatesList->back().set[g].y *= scaleFactor;
            coordinatesList->back().set[g].z *= scaleFactor;
        }
    } catch (ZeroNormal zn) {
        ++failedNumber;
    }

    delete tempThreePoints;
}

failedNumber && cerr << "failed: " << failedNumber << " polygons" << endl;
return coordinatesList;
}

```

dr_program/geometry_utils.h

```

#ifndef GEOMETRY_UTILS_H
#define GEOMETRY_UTILS_H

#include "types.h"
#include <algorithm>
#include <iostream>
#include <cmath>

#ifdef __GNUC__
#define DEPRECATED(func) func __attribute__((deprecated))
#elif defined(_MSC_VER)
#define DEPRECATED(func) __declspec(deprecated) func
#else
#pragma message("WARNING: You need to implement DEPRECATED for this compiler")
#define DEPRECATED(func) func
#endif

using namespace std;

struct ThreePoints;
struct Line;
struct Point;
struct Plane;
struct Sphere;
struct Vector;
struct Triangle;
struct Particle;
struct Object3D;

namespace Geometry {
    // geometry utils with random
    DEPRECATED(Point getRandomPointFromSphere(Sphere));
    Point getRandomPointFromSphere2(Sphere);
    Point getRandomPointOnSphere(Sphere);
    Vector getRandomOrthogonalVector(Vector);
    Point getRandomPointFromTriangle(ThreePoints&);

    // geometry utils returning distances and lengths
    real getDistanceBetweenPoints(Point, Point);
    real getDistanceBetweenPointAndPlane(ThreePoints&, Point);
    real getDistanceBetweenPointAndSphere(Sphere&, Point);
    real getChordLength(Sphere, Line);

    // geometry utils for intersections and projections calculation
    Point getPointOnPlaneProjection(ThreePoints&, Point);
    Point getPointOnLineProjection(Line, Point);
    DEPRECATED(Point getPlaneAndLineIntersection(ThreePoints&, Line));
    Point getPlaneAndLineIntersection2(ThreePoints&, Line);
    // Point getNearestObject3DAndParticleTrajectoryIntersection(Object3D&, Particle);
    int getIndexOfPolygonThatParicleIntersects(Object3D&, Particle&);

    // geometry utils for conditions checking
    DEPRECATED(bool isPointInsideTriangle(ThreePoints&, Point));
    bool isPointInsideTriangle2(ThreePoints&, Point);
    bool doesLineIntersectTriangle(ThreePoints&, Line);
    bool isPointInsideParallelepiped(Point, Point, Point);
    bool doesParticlesTrajectoryIntersectObject(Particle&, Object3D&);
    bool doesLineIntersectParallelepiped(Line, Point, Point);
    bool doesLineIntersectSphere(Line, Sphere);
}

```

```

    // other geometry utils
    Point rotatePointAroundLine(Point, Line, double);
}

#endif // GEOMETRY_UTILS_H

dr_program/geometry_utils.cpp

#include "geometry_utils.h"

using namespace std;

bool Geometry::isPointInsideTriangle(ThreePoints &t, Point k) {
    Vector v0(Point(0,0,0)), v1(k,t.a), v2(k,t.b), v3(k,t.c);
    if (v1 == v0 || v2 == v0 || v3 == v0)
        return true;
    real cos1 = v1.cos(v2), cos2 = v1.cos(v3), cos3 = v2.cos(v3);
    switch( sign(cos1) + sign(cos2) + sign(cos3) ) {
        case -3:
            return true;
        case 3:
        case 1:
            return false;
        default:
            // delta is required to prevent mashine imprecision
            real delta = 0.0001;
            return !(acos(cos1) + acos(cos2) + acos(cos3) < M_PI*2 - delta);
    }
}

bool Geometry::isPointInsideTriangle2(ThreePoints &t, Point k) {
    Point p = getPointOnLineProjection(Line(t.a,t.b),t.c);
    if (Vector(p,t.c).cos(Vector(p,k)) < 0)
        return false;
    p = getPointOnLineProjection(Line(t.c,t.b),t.a);
    if (Vector(p,t.a).cos(Vector(p,k)) < 0)
        return false;
    p = getPointOnLineProjection(Line(t.c,t.a),t.b);
    if (Vector(p,t.b).cos(Vector(p,k)) < 0)
        return false;
    return true;
}

// koefitsiyent tochki peresecheniya nakhodim podstavlyaya parametriceskiye uravneniya
// pryamoy
// v vektornoye uravneniye ploskosti
Point Geometry::getPlaneAndLineIntersection2(ThreePoints &plane, Line line) {
    real coef = plane.getNormal()*Vector(line.a, plane.a) /
        (plane.getNormal()*line.directionVector);
    return line.pointByCoef(coef);
}

Point Geometry::getPointOnPlaneProjection(ThreePoints& plane, Point p) {
    return getPlaneAndLineIntersection2(plane, Line(p, plane.getNormal()));
}

// this method is deprecated. use getPlaneAndLineIntersection2 instead.
// koefitsiyent tochki peresecheniya nakhodim podstavlyaya parametriceskiye uravneniya
// pryamoy
// v kanonicheskoye uravneniye ploskosti
Point Geometry::getPlaneAndLineIntersection(ThreePoints &plane, Line line) {
    real coef1 = (plane.b.y - plane.a.y)*(plane.c.z - plane.a.z) -
        (plane.c.y - plane.a.y)*(plane.b.z - plane.a.z);
    real coef2 = (plane.b.x - plane.a.x)*(plane.c.z - plane.a.z) -
        (plane.c.x - plane.a.x)*(plane.b.z - plane.a.z);
    real coef3 = (plane.b.x - plane.a.x)*(plane.c.y - plane.a.y) -
        (plane.c.x - plane.a.x)*(plane.b.y - plane.a.y);
    real coef = ((line.a.x - plane.a.x)*coef1 -
        (line.a.y - plane.a.y)*coef2 +
        (line.a.z - plane.a.z)*coef3) /
        (line.directionVector.x*coef1 - line.directionVector.y*coef2 +
        line.directionVector.z*coef3);

    // inf means that the line is parallel towards plane
    // nan means that the line belongs to plane
    if(std::isinf(coef) || std::isnan(coef)) {
        return Point(coef, coef, coef);
    }

    return line.pointByCoef(-coef);
}

```

```

// koeffitsiyent tochki peresecheniya nakhodim iz usloviya perpendikulyarnosti
// napravlyayushchego vektora
// pryamoy i vektora, obrazovannogo zadannoy tochkoy i yeye proyektiyey (v kachestve
// koordinat posledney
// berem parametriceskiye uravneniya pryamoy)
Point Geometry::getPointOnLineProjection(Line line, Point point) {
    real coef = line.directionVector*Vector(line.a, point) /
        (line.directionVector*line.directionVector);
    return line.pointByCoef(coef);
}

bool Geometry::doesLineIntersectTriangle(ThreePoints &triangle, Line line) {
    Point intersection = getPlaneAndLineIntersection2(triangle, line);
    if (std::isinf(intersection.x)) {
        cerr << "INF" << endl;
        //TODO: here should be checking whether the line intersects
        /// at least one of the triangles side
        /// for this function for finding two lines intersection should be
        /// implemented
        return false;
    }
    if (std::isnan(intersection.x)) {
        cerr << "NAN " << endl;
        return false;
    }
    bool retVal = isPointInsideTriangle2(triangle, intersection);
    return retVal;
}

Point Geometry::getRandomPointFromSphere(Sphere s) {
    Point randPoint;
    do {
        randPoint.x = (Time::getRandom()*2 - 1)*s.radius;
        randPoint.y = (Time::getRandom()*2 - 1)*s.radius;
        randPoint.z = (Time::getRandom()*2 - 1)*s.radius;
    } while (getDistanceBetweenPoints(randPoint, POINT_OF_ORIGIN) > s.radius);
    return s.center + Vector(POINT_OF_ORIGIN, randPoint);
}

Point Geometry::getRandomPointFromSphere2(Sphere s) {
    return s.center + Vector(Time::getRandom() - 0.5, Time::getRandom() - 0.5, Time::getRandom() - 0.5)
        .resized(sqrt(s.radius*Time::getRandom(0, s.radius)));
}

Point Geometry::getRandomPointOnSphere(Sphere s) {
    return s.center + Vector(Time::getRandom() - 0.5, Time::getRandom() - 0.5, Time::getRandom() - 0.5)
        .resized(s.radius);
}

Vector Geometry::getRandomOrthogonalVector(Vector v) {
    Vector a;
    if (v.x != 0) {
        a.y = Time::getRandom();
        a.z = Time::getRandom();
        a.x = (-v.y*a.y - v.z*a.z)/v.x;
    } else if (v.y != 0) {
        a.x = Time::getRandom();
        a.z = Time::getRandom();
        a.y = (-v.x*a.x - v.z*a.z)/v.y;
    } else if (v.z != 0) {
        a.y = Time::getRandom();
        a.x = Time::getRandom();
        a.z = (-v.y*a.y - v.x*a.x)/v.z;
    }
    return a;
}

real Geometry::getDistanceBetweenPoints(Point a, Point b) {
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2) + pow(a.z - b.z, 2));
}

real Geometry::getDistanceBetweenPointAndPlane(ThreePoints& plane, Point p) {
    return getDistanceBetweenPoints(getPointOnPlaneProjection(plane, p), p);
}

real Geometry::getDistanceBetweenPointAndSphere(Sphere& s, Point p) {
    return max<real>(getDistanceBetweenPoints(p, s.center) - s.radius, 0);
}

```



```

}

bool Geometry::isPointInsideParallelepiped(Point a, Point v1, Point v2) {
    return a.x <= max(v1.x, v2.x) && a.x >= min(v1.x, v2.x) &&
           a.y <= max(v1.y, v2.y) && a.y >= min(v1.y, v2.y) &&
           a.z <= max(v1.z, v2.z) && a.z >= min(v1.z, v2.z);
}

bool Geometry::doesLineIntersectParallelepiped(Line l, Point p1, Point p2) {
    Plane planePencularToOX(p1, Vector(1, 0, 0));
    Plane planePencularToOY(p1, Vector(0, 1, 0));
    Plane planePencularToOZ(p1, Vector(0, 0, 1));

    Point pX = getPlaneAndLineIntersection2(planePencularToOX, l);
    Point pY = getPlaneAndLineIntersection2(planePencularToOY, l);
    Point pZ = getPlaneAndLineIntersection2(planePencularToOZ, l);

    return (inInterval(pZ.x, p1.x, p2.x) && inInterval(pZ.y, p1.y, p2.y)) ||
           (inInterval(pX.y, p1.y, p2.y) && inInterval(pX.z, p1.z, p2.z)) ||
           (inInterval(pY.x, p1.x, p2.x) && inInterval(pY.z, p1.z, p2.z));
}

bool Geometry::doesParticlesTrajectoryIntersectObject(Particle& p, Object3D &obj) {
    Line line(p, p.speed);
    if (!doesLineIntersectSphere(line, obj))
        return false;
    for (unsigned int i = 0; i < obj.polygons->size(); i++)
        if (doesLineIntersectTriangle(obj.polygons->at(i), line))
            return true;
    return false;
}

// line should intersect sphere
real Geometry::getChordLength(Sphere sphere, Line line) {
    return 2*sqrt(sphere.radius*sphere.radius -
                 pow(getDistanceBetweenPoints(sphere.center,
                                                getPointOnLineProjection(line, sphere.center)),
                    2));
}

Point Geometry::getRandomPointFromTriangle(ThreePoints& tp) {
    return tp.a + ( Vector(tp.a, tp.b)*Time::getRandom() + Vector(tp.a, tp.c)*Time::getRandom() ) * 0.5;
}

bool Geometry::doesLineIntersectSphere(Line l, Sphere s) {
    return getDistanceBetweenPoints(getPointOnLineProjection(l, s.center), s.center) <= s.radius;
}

int Geometry::getIndexOfPolygonThatParicleIntersects(Object3D& obj, Particle& p) {
    Line line(p, p.speed);
    if (!doesLineIntersectSphere(line, obj))
        return -1;
    for (unsigned int i = 0; i < obj.polygons->size(); i++)
        if (doesLineIntersectTriangle(obj.polygons->at(i), line))
            return i;
    return -1;
}

// see explanation at pages 9-10 of draft
Point Geometry::rotatePointAroundLine(Point p, Line l, double angle) {
    Point projection = getPointOnLineProjection(l, p);
    Vector j(projection, p);
    double length = j.length();
    Vector i = j.vectorProduct(l.directionVector).resized(length);
    return projection - i*length*sin(angle) + j*length*cos(angle);
}

```

dr_program/graphics_utils.h

```

#ifndef GRAPHICS_UTILS_H
#define GRAPHICS_UTILS_H

#include <SDL/SDL.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <cstdlib>
#include "types.h"
#include <utility>

```

```

namespace Graphics {
    extern Point viewerPosition;

    extern int width;
    extern int height;

    extern bool isLMousePressed;
    extern bool drawAxes;
    extern double rotationAngles[2];
    extern float zoomFactor;

    void initGraphics(int, int);
    void draw(Object3D&, Particle*, int);
    void quitGraphics(int);
}

#endif // GRAPHICS_UTILS_H

```

dr_program/graphics_utils.cpp

```

#include "graphics_utils.h"

Point Graphics::viewerPosition(0,0,0);
int Graphics::width = 0;
int Graphics::height = 0;
float Graphics::zoomFactor = 1.0;
bool Graphics::isLMousePressed = false;
double Graphics::rotationAngles[2] = {0,0};
bool Graphics::drawAxes = false;

void Graphics::initGraphics(int _width, int _height) {
    height = _height;
    width = _width;
    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
        cerr << "Video initialization failed: " << SDL_GetError() << endl;
        quitGraphics(1);
    }

    const SDL_VideoInfo* info = NULL;
    info = SDL_GetVideoInfo();
    if(!info) {
        cerr << "getting video info failed: " << SDL_GetError() << endl;
        quitGraphics(1);
    }

    SDL_GL_SetAttribute(SDL_GL_RED_SIZE,5);
    SDL_GL_SetAttribute(SDL_GL_GREEN_SIZE,5);
    SDL_GL_SetAttribute(SDL_GL_BLUE_SIZE,5);
    SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE,16);
    SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER,1);

    int bitsPerPixel = info->vfmt->BitsPerPixel;
    int flags = SDL_OPENGL | SDL_HWSURFACE | SDL_ASYNCBLIT | SDL_RESIZABLE;
    if(SDL_SetVideoMode(width,height,bitsPerPixel,flags) == 0) {
        cerr << "setting video mode failed: " << SDL_GetError() << endl;
        quitGraphics(1);
    }

    glShadeModel(GL_SMOOTH);

    glCullFace(GL_BACK);
    glFrontFace(GL_CCW);
    glEnable(GL_CULL_FACE);

    glViewport(0, 0, width, height);

    glPointSize(1.5);
}

void Graphics::quitGraphics(int code) {
    SDL_Quit();
    exit(code);
}

void Graphics::draw(Object3D &satelliteObj, Particle* particlesArray = NULL, int
particlesNumber = 0)
{
    // colors
    static GLubyte purple[] = {255,150,255,0};
    // static GLubyte grey[] = {150,150,150,0};
    static GLubyte red[] = {255,0,0,0};
    static GLubyte green[] = {0,255,0,0};
}

```

```

static GLubyte blue[] = {0,0,255,0};

glClearColor(255,255,255,0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Projections matrix processing
static float ratio = (float)width/(float)height;
static double diameter = satelliteObj.radius*2;
static GLdouble zNear = ELECTRONS_GENERATIVE_SPHERE_RADIUS*2;
static GLdouble zFar = -ELECTRONS_GENERATIVE_SPHERE_RADIUS*2;
static GLdouble left = satelliteObj.center.x - diameter;
static GLdouble right = satelliteObj.center.x + diameter;
static GLdouble bottom = satelliteObj.center.y - diameter;
static GLdouble top = satelliteObj.center.y + diameter;
static bool firstTimeCall = true;

if (firstTimeCall) {
    firstTimeCall = false;
    viewerPosition.z = -1.5*diameter;
    if (ratio < 1.0) { // width < height
        bottom /= ratio;
        top /= ratio;
    } else {
        left *= ratio;
        right *= ratio;
    }
}

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left*zoomFactor, right*zoomFactor, bottom*zoomFactor, top*zoomFactor, zNear, zFar);

// Modelview matrix processing
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
/* Move down the z-axis. */
gluLookAt(viewerPosition.x, viewerPosition.y, viewerPosition.z,
           satelliteObj.center.x, satelliteObj.center.y, satelliteObj.center.z,
           0.0, 1.0, 0.0);

/* Rotation by mouse */
if (rotationAngles[0])
    glRotated(rotationAngles[0], 0, 1, 0);
if (rotationAngles[1])
    glRotated(rotationAngles[1], 1, 0, 0);

// draw axes:
if (drawAxes) {
    glBegin(GL_LINES);
    glColor4ubv(red); glVertex3d(0,0,0); glVertex3d(1.5*diameter,0,0);
    glColor4ubv(green); glVertex3d(0,0,0); glVertex3d(0,1.5*diameter,0);
    glColor4ubv(blue); glVertex3d(0,0,0); glVertex3d(0,0,1.5*diameter);
    glEnd();
}

// draw the particles
if (particlesArray != NULL) {
    glBegin(GL_POINTS);
    for(int i = 0; i < particlesNumber; ++i) {
        if (particlesArray[i].behaviour == PARTICLE_WILL_INTERSECT_OBJ) {
            glColor4ubv(green);
        } else {
            glColor4ubv((particlesArray[i].type == PTYPE_ELECTRON)? blue: red); // this
                will colorize electrons to blue, ions to red
        }

        if (particlesArray[i].getPreviousStates()->size() != 0) {
            glBegin(GL_LINE_STRIP);
            for(auto it = particlesArray[i].getPreviousStates()->begin();
                it != particlesArray[i].getPreviousStates()->end(); ++it)
                glVertex3f((*it).x, (*it).y, (*it).z);
            glVertex3f(particlesArray[i].x, particlesArray[i].y, particlesArray[i].z);
            glEnd();
        } else {
            glVertex3f(particlesArray[i].x, particlesArray[i].y, particlesArray[i].z);
        }
    }
    glEnd();
}
}

```

```

    // draw the object
    glColor4ubv(purple);
    vector<PlaneType> *coords = satelliteObj.polygons;
    for(vector<PlaneType>::iterator it = coords->begin(); it != coords->end(); it++) {
        glBegin(GL_LINE_LOOP);
        for(int i = 0; i < 3; i++)
            glVertex3d((*it).set[i].x, (*it).set[i].y, (*it).set[i].z);
        glEnd();
    }
    SDL_GL_SwapBuffers();
}

```

dr_program/time_utils.h

```

#ifndef TIME_UTILS_H
#define TIME_UTILS_H

#include <unistd.h>
#include <ctime>
#include <cstdlib>
#include <iostream>
#include <random>
#include <functional>
#include <iomanip>
#include "constants.h"

using namespace std;

#define RAND(n) rand() % n

extern int CLOCK_ID;

typedef uniform_real_distribution<double> UniformDistribution;
typedef normal_distribution<double> GaussianDistribution;

typedef mt19937 Engine;

template <typename Eng, typename Distrib>
class Generator {
private:
    Eng engine;
    Distrib distribution;
public:
    Generator(Eng e, Distrib d): engine(e), distribution(d) {}
    double operator() () {
        return distribution(engine);
    }
};

typedef Generator<Engine, UniformDistribution> UniformDistributionGenerator;
typedef Generator<Engine, GaussianDistribution> GaussianDistributionGenerator;
typedef function<velocity ()> MaxwellDistributionSpeedGenerator;

namespace Time {
    void printTimespec(timespec*);

    UniformDistributionGenerator getUniformDistributionGenerator(double, double);
    GaussianDistributionGenerator getGaussianDistributionGenerator(double, double);
    MaxwellDistributionSpeedGenerator getMaxwellDistributionSpeedGenerator(double, double);

    timespec* getTimespecDelta(timespec*, timespec*);

    double getRandom();

    double getRandom(double, double);
}

#endif // TIME_UTILS_H

```

dr_program/time_utils.cpp

```

#include "time_utils.h"

int CLOCK_ID = CLOCK_THREAD_CPUTIME_ID;

void Time::printTimespec(timespec *ts) {
    cout << ts->tv_sec << '.' << std::setfill('0') << std::setw(9) << ts->tv_nsec << " sec"
        << endl;
}

timespec* Time::getTimespecDelta(timespec *older, timespec *newer) {
    newer->tv_nsec -= older->tv_nsec;
}

```

```

    newer->tv_sec -= older->tv_sec;
    if (newer->tv_nsec < 0) {
        newer->tv_sec--;
        newer->tv_nsec += 1000000000;
    }
    return newer;
}

double Time::getRandom() {
    static random_device seed;
    static UniformDistributionGenerator* generator =
        new UniformDistributionGenerator(Engine(seed()), UniformDistribution(0,1));
    return (*generator)();
}

double Time::getRandom(double left, double right) {
    return getRandom()*(right - left) + left;
}

UniformDistributionGenerator Time::getUniformDistributionGenerator(double min, double max) {
    static random_device seed;
    return UniformDistributionGenerator(Engine(seed()), UniformDistribution(min,max));
}

GaussianDistributionGenerator Time::getGaussianDistributionGenerator(double M, double D) {
    static random_device seed;
    return GaussianDistributionGenerator(Engine(seed()), GaussianDistribution(M,D));
}

MaxwellDistributionSpeedGenerator Time::getMaxwellDistributionSpeedGenerator(double M, double
D) {
    return [M,D]() -> velocity {
        //TODO check for M, D
        static GaussianDistributionGenerator gdg = getGaussianDistributionGenerator(M/
sqrt(3.0), D/sqrt(3.0));
        return sqrt(pow(gdg(),2) + pow(gdg(),2) + pow(gdg(),2));
    };
}

```

dr_program/types.h

```

#ifndef TYPES_H
#define TYPES_H

#include <iostream>
#include <string>
#include <stdexcept>
#include <cmath>
#include <vector>
#include <list>
#include <cstring>
#include <assert.h>

#include "types.h"
#include "constants.h"
#include "geometry_utils.h"
#include "time_utils.h"

using namespace std;

struct Point;
struct Plane;
struct Vector;
struct OrientedPlane;
struct Object3D;
struct Particle;

typedef OrientedPlane PlaneType;

extern Point POINT_OF_ORIGIN; // (0,0,0)

// flags for particle states
const int PARTICLE_WILL_INTERSECT_OBJ = 1;
const int PARTICLE_WILL_NOT_INTERSECT_OBJ = 2;
const int PARTICLE_HAS_UNDEFINED_BEHAVIOUR = 4;
//extern unsigned int PARTICLE_WILL_;

// system of coordinates orientation
#define ORIENT_RIGHT_HANDED 1
#define ORIENT_LEFT_HANDED 0

// order - clockwise or counterclockwise
#define ORDER_CW 1

```

```

#define ORDER_CCW 0

// generation flags
enum genFlags {GEN_ON_SPHERE = 1, GEN_IN_SPHERE = 2, GEN_RANDOM = 4, GEN_INTERSECT_OBJ = 8};

void setOrientation(bool);
bool getOrientation();

template <typename T>
inline char sign(T t) {
    return (t > 0)? 1: (t < 0)? -1: 0;
}

template <typename T>
inline bool inInterval(T x,T a,T b) {
    return x <= max(a,b) && x >= min(a,b);
}

class ZeroNormal: public runtime_error {
public:
    ZeroNormal(char *msg): runtime_error(string(msg)) {}
    ZeroNormal(string &msg): runtime_error(msg) {}
};

struct Point {
    real x;
    real y;
    real z;

    Point(): x(0),y(0),z(0) {}
    Point(real _x, real _y, real _z): x(_x),y(_y),z(_z) {}
    Point(const Point &p) {x = p.x;y = p.y;z = p.z;}

    Point operator+(Vector v);
    Point operator-(Vector v);
    friend ostream& operator<<(ostream &os, const Point &p) {
        os << '(' << p.x << ', ' << p.y << ', ' << p.z << ')';
        return os;
    }
    bool operator==(Point a) const {
        return (x == a.x && y == a.y && z == a.z);
    }
    bool operator!=(Point b) const {
        return !(*this == b);
    }
    Point rotateAroundZ(double cos,double sin) {
        return Point(x*cos - y*sin,x*sin + y*cos,z);
    }
    Point rotateAroundY(double cos,double sin) {
        return Point(x*cos + z*sin,y,-x*sin + z*cos);
    }
    real& operator[](int i) {
        switch(i % 3) {
            case 0: return x;
            case 1: return y;
            case 2: return z;
            default: assert(false);
        }
    }
};

struct Vector: public Point {
    Vector(): Point() {}
    Vector(Point p): Point(p) {}
    Vector(real _x, real _y, real _z): Point(_x,_y,_z) {}
    Vector(Point b,Point a): Point(a.x - b.x,a.y - b.y,a.z - b.z) {}

    real operator*(Vector right) {
        return x*right.x + y*right.y + z*right.z;
    }
    Vector operator*(double k) {
        return Vector(k*x,k*y,k*z);
    }
    Vector operator/(double k) {
        return Vector(x/k,y/k,z/k);
    }
    Vector operator+(Vector v) {
        return Vector(x + v.x,y + v.y, z + v.z);
    }
    Vector operator-(Vector v) {
        return Vector(x - v.x,y - v.y, z - v.z);
    }
};

```

```

    }
    real length() {
        return sqrt(x*x + y*y + z*z);
    }
    double cos(Vector right) {
        return ((*this)*right)/((this->length()*right.length()));
    }
    Vector vectorProduct(Vector left) {
        return Vector(y*left.z - z*left.y, -x*left.z +
                      z*left.x, x*left.y - y*left.x);
    }
    Vector normalized() {
        double len = length();
        return Vector(x/len, y/len, z/len);
    }
    Vector resized(real _length) {
        double coef = _length/length();
        if (std::isnan(coef))
            return Vector(x, y, z);
        return Vector(x*coef, y*coef, z*coef);
    }
    void resize(real _length) {
        double coef = _length/length();
        if (!std::isnan(coef)) {
            x *= coef;
            y *= coef;
            z *= coef;
        }
    }
};

template <unsigned int T>
struct Locus { // collection of points
    Point set[T];
    friend ostream& operator<<(ostream &os, const Locus &l) {
        for(unsigned int i = 0; i < T - 1; i++)
            os << l.set[i] << ", ";
        os << l.set[T - 1];
        return os;
    }

    virtual ~Locus() {
    }
};

struct Line: public Locus<2> {
    Vector directionVector;
    Line(Point _a, Point _b): a(set[0]), b(set[1]) {
        set[0] = _a; set[1] = _b;
        directionVector = Vector(_a, _b);
    }
    Line(Point _a, Vector v): a(set[0]), b(set[1]) {
        directionVector = v;
        set[0] = _a;
        set[1] = _a + v;
    }
    Point& a;
    Point& b;
    Point pointByCoef(real coef) {
        return Point(set[0].x + coef*directionVector.x,
                     set[0].y + coef*directionVector.y,
                     set[0].z + coef*directionVector.z);
    }
};

struct ThreePoints : public Locus<3> {
    ThreePoints(): a(set[0]), b(set[1]), c(set[2]) {}
    ThreePoints(const ThreePoints &tP): a(set[0]), b(set[1]), c(set[2]) {
        set[0] = tP.set[0]; set[1] = tP.set[1]; set[2] = tP.set[2];
    }
    ThreePoints(Point _a, Point _b, Point _c): a(set[0]), b(set[1]), c(set[2]) {
        set[0] = _a; set[1] = _b; set[2] = _c;
    }
    ThreePoints& operator=(const ThreePoints& right) {
        if (this != &right) {
            memcpy(set, right.set, 3*sizeof(Point));
        }
        return *this;
    }
};

```

```

    Point& a;
    Point& b;
    Point& c;
    virtual Vector getNormal() {
        return Vector(a,b).vectorProduct(Vector(a,c));
    }

    Point centerOfMass() {
        return Point( (a.x + b.x + c.x) / 3.0,
                      (a.y + b.y + c.y) / 3.0,
                      (a.z + b.z + c.z) / 3.0);
    }

    double area() {
        Vector ab(a,b), ac(a,c);
        return 0.5*sqrt(1 - pow(ab.cos(ac),2))*ab.length()*ac.length();
    }
};

struct Triangle: public ThreePoints {
    Triangle(Point _a,Point _b,Point _c): ThreePoints(_a,_b,_c) {}
    Triangle(ThreePoints &tP): ThreePoints(tP) {}
};

struct Plane: public ThreePoints {
    Plane(): ThreePoints() {}
    Plane(Point _a,Point _b,Point _c): ThreePoints(_a,_b,_c) {}
    Plane(ThreePoints &tP): ThreePoints(tP) {}
    Plane(Point, Vector);
    bool doesPointBelongPlane(Point p) {
        /// TODO fix possible error because of mashine precision
        return Vector(a,p)*getNormal() == 0;
    }
};

struct OrientedPlane: public Plane {
    Vector normal;
    OrientedPlane(): Plane(), normal() {}
    OrientedPlane(Point _a,Point _b,Point _c, bool _pointsOrder = ORDER_CCW):
        Plane(_a,_b,_c), _pointsOrder(_pointsOrder) {
        initNormal();
    }
    OrientedPlane(ThreePoints &tP, bool _pointsOrder = ORDER_CCW):
        Plane(tP), _pointsOrder(_pointsOrder) {
        initNormal();
    }
    OrientedPlane(Plane p, Vector v): Plane(p), normal(v) {}
    OrientedPlane(Point p, Vector v): Plane(p,v), normal(v) {}
    OrientedPlane(const OrientedPlane &op): Plane((ThreePoints&)op),
        normal(op.normal) {}
    Vector getNormal() {
        return normal;
    }
};

private:
    bool pointsOrder;
    void initNormal() {
        Vector ab(a,b);
        Vector ac(a,c);
        // get cross product
        normal = ab.vectorProduct(ac);
        if (pointsOrder == ORDER_CW) {
            normal = normal*(-1);
        }
        double test1 = normal.length(), test2 = area();
        if (std::isnan(test1) || std::isnan(test2) || test1 < 0.0000001 || test2 < 0.0000001) {
            throw ZeroNormal("zero-normal");
        }
    }
};

struct Particle: public Point {
public:
    list<Point> *previousStates;
    static double electronTrajectoryCurrent;
    static double ionTrajectoryCurrent;
    char type;
    Vector speed;
};

```



```

real ttl;
int polygonIndex;
int behaviour;
Particle operator+(Vector v);
Particle operator-(Vector v);
Particle(char _type = PTYPE_ELECTRON, int _flags = PARTICLE_HAS_UNDEFINED_BEHAVIOUR):
    Point(), type(_type), speed(), ttl(-1), polygonIndex(-1), behaviour(_flags) {
    previousStates = NULL;
}
Particle(Point p, Vector s, real ttl_ = -1, char _type = PTYPE_ELECTRON, int _pi = -1,
    int _flags = PARTICLE_HAS_UNDEFINED_BEHAVIOUR):
    Point(p), type(_type), speed(s), ttl(ttl_), polygonIndex(_pi), behaviour(_flags) {
    previousStates = NULL;
}
void affectField(Vector fieldGrad, double timeStep) {
    Vector distance = speed*timeStep + fieldGrad*(timeStep*timeStep*
        PARTICLE_CHARGE_TO_MASS(type)/2); ///!! +, not -
    Point newPosition = *(Point*)this + distance;
    x = newPosition.x;
    y = newPosition.y;
    z = newPosition.z;
    Vector acceleration = fieldGrad*(PARTICLE_CHARGE_TO_MASS(type)); ///!! +, not -
    speed = speed + acceleration*timeStep;
}

void addPreviousStates(Point p) {
    if (previousStates == NULL)
        initPreviousStates();
    previousStates->push_back(p);
}

list<Point*> getPreviousStates() {
    if (previousStates == NULL)
        initPreviousStates();
    return previousStates;
}

void finalize() {
    if (previousStates != NULL)
        delete previousStates;
}

private:
    void initPreviousStates() {
        previousStates = new list<Point*>();
    }
};

struct Sphere {
    Point center;
    real radius;
    Sphere(): center(), radius(0) {}
    Sphere(Point _p, real _r): center(_p), radius(_r) {}
    Sphere(const Sphere &_s): center(_s.center), radius(_s.radius) {}
};

struct Object3D: public Sphere {
    double totalPlasmaCurrent;
    long double totalCharge;
    Vector front;
    Point maxCoords, minCoords;
    Point nearestPoint, furthestmostPoint; // relatively to front of the object
    vector<PlaneType> *polygons;
    velocity speed;
    // double *polygonsCurrents;

    Object3D(int polygonsNumber, Vector _front = Vector(100,0,0)): front(_front) {
        polygons = new vector<PlaneType>(polygonsNumber);
        init();
    }
    Object3D(vector<PlaneType> *_polygons, Vector _front = Vector(100,0,0)):
        front(_front), polygons(_polygons) {
        init();
    }

    void init();

    PlaneType& operator[(int i)] {
        return polygons->at(i);
    }
}

```

```

Vector step() {
    return front.normalized()*speed;
}

double surfaceArea() {
    double sA = 0.0;
    for(vector<PlaneType>::iterator it = polygons->begin(); it != polygons->end(); ++it) {
        sA += (*it).area();
    }
    return sA;
}

double capacitance() {
    // calculating capacitance as for sphere with the same radius
    return 4*M_PI*VACUUM_PERMITTIVITY*sqrt(surfaceArea()/(4*M_PI));
}

void changePlasmaCurrents(double change) {
    // polygonsCurrents[polygonIndex] += change;
    totalPlasmaCurrent += change;
}

~Object3D() {
    // delete polygonsCurrents;
}

};

struct GenerativeSphere: public Sphere {
private:
    void checkForIntersectionsAndSetTtl(Particle&);
    velocity electronVelocityGenerator() {
        static MaxwellDistributionSpeedGenerator generator =
            Time::getMaxwellDistributionSpeedGenerator(ELECTRON_VELOCITY_M,
                ELECTRON_VELOCITY_D);
        return generator();
    }
    velocity ionVelocityGenerator() {
        static MaxwellDistributionSpeedGenerator generator =
            Time::getMaxwellDistributionSpeedGenerator(ION_VELOCITY_M, ION_VELOCITY_D);
        return generator();
    }
    Object3D &object;
    Vector objectStep;

public:
    GenerativeSphere(Point _p, real _r, Object3D &_object):
        Sphere(_p, _r), object(_object), objectStep(object.step()) {}
    GenerativeSphere(const Sphere &_s, Object3D &_object):
        Sphere(_s), object(_object), objectStep(object.step()) {}

    void generateParticleInSphere(Particle *,int);
    void generateParticleWhichIntersectsObject(Particle *,int,bool);
    void generateParticleOnSphere(Particle *,int);
    void populateArray(Particle*,int,int,int);
};

#endif // TYPES_H

```

dr_program/types.cpp

```

#include "types.h"
#include <assert.h>

Point POINT_OF_ORIGIN = Point(0,0,0);

double Particle::electronTrajectoryCurrent;
double Particle::ionTrajectoryCurrent;

Point Point::operator+(Vector v) {
    return Point(x + v.x, y + v.y, z + v.z);
}

Point Point::operator-(Vector v) {
    return Point(x - v.x, y - v.y, z - v.z);
}

Particle Particle::operator+(Vector v) {
    Point newPosition = (Point)(*this) + v;
    x = newPosition.x;
    y = newPosition.y;
    z = newPosition.z;
}

```

```

    return (*this);
}

Particle Particle::operator-(Vector v) {
    Point newPosition = (Point)(*this) - v;
    x = newPosition.x;
    y = newPosition.y;
    z = newPosition.z;
    return (*this);
}

Plane::Plane(Point p, Vector v):
    ThreePoints(p, p + Geometry::getRandomOrthogonalVector(v),
                p + Geometry::getRandomOrthogonalVector(v)) {}

void GenerativeSphere::checkForIntersectionsAndSetTtl(Particle &p) {
    p.polygonIndex = Geometry::getIndexOfPolygonThatParicleIntersects(object, p);
    if (p.polygonIndex != -1) {
        p.ttl = Geometry::getDistanceBetweenPoints(p,
            Geometry::getPlaneAndLineIntersection2(object.polygons->at(p.polygonIndex), Line(
                p, p.speed)))/p.speed.length();
    } else { // see description at page 11 of the draft
        p.ttl = 2*radius*p.speed.cos(Vector(p, center)) / p.speed.length();
    }
}

void GenerativeSphere::generateParticleInSphere(Particle *p, int type) {
    Point initialPosition = Geometry::getRandomPointFromSphere2(*this);
    Vector step(Time::getRandom() - 0.5, Time::getRandom() - 0.5, Time::getRandom() - 0.5);
    switch(type) {
        case PTYPE_ELECTRON:
            step = step.resized(electronVelocityGenerator()) - objectStep;
            break;
        case PTYPE_ION:
            step = step.resized(ionVelocityGenerator()) - objectStep;
            break;
    }

    *p = Particle(initialPosition, step, 0, type);
    checkForIntersectionsAndSetTtl(*p);
}

void GenerativeSphere::generateParticleWhichIntersectsObject(Particle *pt, int type, bool
isParticleOnSphere) {
    int polygonIndex = RAND(object.polygons->size());
    Vector n = object.polygons->at(polygonIndex).getNormal().normalized();
    velocity particleSpeed;

    switch(type) {
        case PTYPE_ELECTRON:
            particleSpeed = electronVelocityGenerator();
            break;
        case PTYPE_ION:
            particleSpeed = ionVelocityGenerator();
            break;
    }

    Point p = Geometry::getRandomPointFromTriangle(object.polygons->at(polygonIndex));
    Line auxLine(p, (object.polygons->at(polygonIndex).a != p)?
        object.polygons->at(polygonIndex).a : object.polygons->at(polygonIndex).
        b);

    // see explanation at page 2 of draft
    if (particleSpeed <= -object.speed*n.cos(objectStep)) {
        particleSpeed = Time::getRandom(max<velocity>(0.1, -object.speed*n.cos(objectStep))
            , 2.*ELECTRON_VELOCITY_M); // TODO fix me
    }

    double cos = Time::getRandom(-1, min<velocity>(1, object.speed*n.cos(objectStep))/
        particleSpeed);
    // see explanation at page 8 of draft
    Vector s(p, Geometry::rotatePointAroundLine(p + n, auxLine, acos(cos)));
    s = s.resized(particleSpeed) - objectStep;

    real distanceBetweenParticleAndPolygon;
    if (isParticleOnSphere) {
        // now we should calculate point on sphere were particle will be initially placed
        // see explanation at page 1 of draft
        real a = Geometry::getDistanceBetweenPoints(center, p);
        cos = Vector(center, p).cos(s);
        distanceBetweenParticleAndPolygon = sqrt(a*a*(cos*cos - 1) + radius*radius) + a*cos;
    }
}

```

```

        // now p is point on sphere
        p = p - s.resized(distanceBetweenParticleAndPolygon);

        //assert (abs(radius - GU::getDistanceBetweenPoints(p,center)) <= 0.00001);
    } else {
        // see explanation at page 5 of draft
        distanceBetweenParticleAndPolygon = sqrt(Time::getRandom(object.radius, radius)*
            radius);
        p = p - s.resized(distanceBetweenParticleAndPolygon);
    }

    *pt = Particle(p,s,distanceBetweenParticleAndPolygon/particleSpeed,type,polygonIndex);
}

void GenerativeSphere::generateParticleOnSphere(Particle *p,int type) {
    Point initialPosition = Geometry::getRandomPointOnSphere(*this);
    Vector step(initialPosition,Geometry::getRandomPointOnSphere(*this));

    switch(type) {
    case PTYPE_ELECTRON:
        step = step.resized(electronVelocityGenerator()) - objectStep;
        break;
    case PTYPE_ION:
        step = step.resized(ionVelocityGenerator()) - objectStep;
        break;
    }

    *p = Particle(initialPosition,step,0,type);
    checkForIntersectionsAndSetTtl(*p);
}

void GenerativeSphere::populateArray(Particle *particles,int number,int type,int FLAGS) {
    int n = 0;
    bool isOnSphere = FLAGS & GEN_ON_SPHERE;
    if (FLAGS & GEN_INTERSECT_OBJ)
        for (;n < number;++n) {
            generateParticleWhichIntersectsObject(particles + n,type,isOnSphere);
        }
    else if (FLAGS & GEN_RANDOM)
        for (;n < number;++n) {
            generateParticleInSphere(particles + n,type);
        }
    else if (isOnSphere)
        for (;n < number;++n) {
            generateParticleOnSphere(particles + n,type);
        }
}

void Object3D::init() {
    totalPlasmaCurrent = 0;
    totalCharge = 0;
    speed = ORBITAL_VELOCITY;
    assert(polygons->size() > 0);
    nearestPoint = furthestmostPoint = maxCoords = minCoords = polygons->at(0).set[0];
    for(vector<PlaneType>::iterator it = polygons->begin(); it != polygons->end(); it++)
        for(int i = 0; i < 3; i++) {
            if (Vector(nearestPoint,(*it).set[i]).cos(front) > 0)
                nearestPoint = (*it).set[i];
            if (Vector(furthestmostPoint,(*it).set[i]).cos(front) < 0)
                furthestmostPoint = (*it).set[i];

            if (maxCoords.x < (*it).set[i].x) maxCoords.x = (*it).set[i].x;
            if (maxCoords.y < (*it).set[i].y) maxCoords.y = (*it).set[i].y;
            if (maxCoords.z < (*it).set[i].z) maxCoords.z = (*it).set[i].z;
            if (minCoords.x > (*it).set[i].x) minCoords.x = (*it).set[i].x;
            if (minCoords.y > (*it).set[i].y) minCoords.y = (*it).set[i].y;
            if (minCoords.z > (*it).set[i].z) minCoords.z = (*it).set[i].z;
        }
    center = Point((maxCoords.x + minCoords.x)/2,
        (maxCoords.y + minCoords.y)/2,
        (maxCoords.z + minCoords.z)/2);
    radius = Geometry::getDistanceBetweenPoints(center,maxCoords);
    // polygonsCurrents = new double[polygons->size()];
    // for(unsigned int i = 0; i < polygons->size(); ++i) {
    //     polygonsCurrents[i] = 0;
    // }

    // move the spacecraft to the point of origin

```

```

Vector shift(center, POINT_OF_ORIGIN);
if (shift.length() > radius) {
    for(vector<PlaneType>::iterator it = polygons->begin(); it != polygons->end(); it++)
        for(int i = 0; i < 3; i++) {
            (*it).set[i] = (*it).set[i] + shift;
        }
    // and reinit
    init();
}
}

```

dr_program/main.cpp

```

#include <unistd.h>
#include <cmath>
#include <algorithm>
#include <cstdio>
#include <vector>
#include <errno.h>
#include <unistd.h>
#include <assert.h>
#include <limits>

#include "types.h"
#include "file_utils.h"
#include "geometry_utils.h"
#include "constants.h"
#include "data_utils.h"
#include "graphics_utils.h"
#include "fortran_modules.h"

#define rand(max) rand()%max

const char usage[] = "Usage:\n\nprogram [-m][-v][-d][-x][-g][-t NUMBER]\n\n[-r RADIUS][-s TIME][-n N][-f SF][-i INTERVAL][-p STEP] <filename>\n\n-t NUMBER - test probability with number of particles NUMBER\n\n-r RADIUS - radius of generative sphere [not used]\n\n-s TIME - time to sleep in microseconds\n\n-m - model particles\n\n-o NUM - modeling type; 1, 2 or 3\n\n    1 - modeling without field affecting\n\n    2 - (default) most optimized modeling\n\n    3 - modeling best applicable for drawing\n\n-v - verbose mode\n\n-d - draw\n\n-j - draw trajectories of particles\n\n-a - draw axes\n\n-c CHARGE - initial charge of spacecraft (default -0.0000005)\n\n-f SF - scale factor for coordinates in file to reduce them to SI\n\n    (default 1)\n\n-n N - total number of particles at time moment\n\n-i INTERVAL - interval to print measurings\n\n    (use 'i' prefix for number of steps or 's' for seconds)\n\n-p STEP - step of particle measured in spacecrafts length\n\n    (default 0.25)\n\n-x - file with complex data format";

namespace Globals {
    unsigned long long realToModelNumber;
    long double initialCharge = -0.0000005; // -0.00000445;
    bool debug = true;
    bool pause = false;
    bool drawTrajectories = false;
    int modelingType = 2;
}

static void handleKeyDown(SDL_keysym* keysym)
{
    switch(keysym->sym) {
        case SDLK_ESCAPE:
            Graphics::quitGraphics(0);
            break;
        case SDLK_p:
            Globals::pause = !Globals::pause;
            break;
        default:
            break;
    }
}

```

```

void processEvents(void)
{
    /* Our SDL event placeholder. */
    SDL_Event event;
    float zoomDelta = 0.05;
    float zoomDelta2 = 0.5;

    /* Grab all the events off the queue. */
    while(SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_KEYDOWN:
                handleKeyDown( &event.key.keysym );
                break;
            case SDL_QUIT:
                /* Handle quit requests (like Ctrl-c). */
                Graphics::quitGraphics(0);
                break;
            case SDL_MOUSEBUTTONDOWN:
                switch(event.button.button) {
                    case SDL_BUTTON_LEFT:
                        Graphics::isLMousePressed = true;
                        break;
                    case SDL_BUTTON_WHEELDOWN:
                        if (Graphics::zoomFactor > 2*zoomDelta)
                            Graphics::zoomFactor -= (Graphics::zoomFactor > 2)? zoomDelta2:
                                zoomDelta;
                        break;
                    case SDL_BUTTON_WHEELUP:
                        Graphics::zoomFactor += (Graphics::zoomFactor > 2)? zoomDelta2: zoomDelta;
                        break;
                }
                break;
            case SDL_MOUSEBUTTONUP:
                if (event.button.button == SDL_BUTTON_LEFT)
                    Graphics::isLMousePressed = false;
                break;
            case SDL_MOUSEMOTION:
                if (Graphics::isLMousePressed) {
                    double coef = 0.5;
                    Graphics::rotationAngles[0] += -event.motion.xrel*coef;
                    Graphics::rotationAngles[0] -= 360*int(Graphics::rotationAngles[0]/360);
                    Graphics::rotationAngles[1] += -event.motion.yrel*coef;
                    Graphics::rotationAngles[1] -= 360*int(Graphics::rotationAngles[1]/360);
                }
        }
    }
}

// for internal usage only
inline void finalizeParticle(Object3D &satelliteObj, Particle* particles,
                            unsigned long long &electronsNumber, unsigned long long &
                            ionsNumber, int i) {
    if (particles[i].behaviour == PARTICLE_WILL_INTERSECT_OBJ) {
        satelliteObj.totalCharge += PARTICLE_CHARGE(particles[i].type)*Globals::
            realToModelNumber;
        Globals::debug && COUT("charge delta = " << PARTICLE_CHARGE(particles[i].type)*
            Globals::realToModelNumber << ", totalCharge = " << satelliteObj.totalCharge);
        satelliteObj.changePlasmaCurrents((particles[i].type == PTYPE_ELECTRON)?
            Particle::electronTrajectoryCurrent :
            Particle::ionTrajectoryCurrent);
    }

    switch(particles[i].type) {
        case PTYPE_ELECTRON:
            electronsNumber--;
            break;
        case PTYPE_ION:
            ionsNumber--;
            break;
    }

    particles[i].finalize();
}

int processParticles(Object3D &satelliteObj, Particle* particles,
                    unsigned long long &electronsNumber, unsigned long long &ionsNumber,
                    double timeStep, GenerativeSphere electronsGenerativeSphere,
                    GenerativeSphere ionsGenerativeSphere) {
    int finalizedNumber = 0;
    Vector fieldGrad;
    real fieldPot;

```

```

for(unsigned long long i = 0; i < electronsNumber + ionsNumber; ++i) {
    GenerativeSphere &gs = (particles[i].type == PTYPE_ELECTRON)?
        electronsGenerativeSphere: ionsGenerativeSphere;
    if (Geometry::getDistanceBetweenPoints(gs.center, particles[i]) > gs.radius) { //
        kick particle if it has left the modeling sphere
        particles[i].ttl = -1;
        continue;
    }

    if (Globals::drawTrajectories)
        particles[i].addPreviousStates(particles[i]);

    //
    _____

    if (Globals::modelingType == 1) { // modeling without affecting of field
        //
        _____

        // if for some reason particle has left generative sphere - remove it
        if (particles[i].behaviour == PARTICLE_HAS_UNDEFINED_BEHAVIOUR) {
            real index = Geometry::getIndexOfPolygonThatParicleIntersects(satelliteObj,
                particles[i]);
            if (index == -1) {
                particles[i].behaviour = PARTICLE_WILL_NOT_INTERSECT_OBJ;
                particles[i].ttl = 100; // particle will be kicked
            } else {
                particles[i].behaviour = PARTICLE_WILL_INTERSECT_OBJ;
                particles[i].ttl = Geometry::getDistanceBetweenPointAndPlane(
                    satelliteObj.polygons->at(index), particles[i]) /
                    particles[i].speed.length();
            }
        }
        particles[i] = particles[i] + particles[i].speed*timeStep;
        particles[i].ttl -= timeStep;
        //
        _____

    } else if (Globals::modelingType == 2) { // most optimized modeling
        //
        _____

        if (particles[i].behaviour == PARTICLE_WILL_INTERSECT_OBJ || particles[i].
            behaviour == PARTICLE_WILL_NOT_INTERSECT_OBJ) {
            particles[i] = particles[i] + particles[i].speed*timeStep;
            particles[i].ttl -= timeStep;
        } else { // particles[i].behaviour == PARTICLE_HAS_UNDEFINED_BEHAVIOUR
            real distanceToSatellite = Geometry::getDistanceBetweenPointAndSphere(
                satelliteObj, particles[i]);
            int index;

            if (distanceToSatellite == 0 || // if particle is inside satellite's sphere
                or too close to sphere and will be inside it soon
                (distanceToSatellite < particles[i].speed.length()*timeStep &&
                    Geometry::doesLineIntersectSphere(Line(particles[i], particles[i].
                        speed), satelliteObj))) {
                index = Geometry::getIndexOfPolygonThatParicleIntersects(satelliteObj,
                    particles[i]);
                if (index != -1) { // then particle will intersect object
                    particles[i].behaviour = PARTICLE_WILL_INTERSECT_OBJ;
                    particles[i].ttl = Geometry::getDistanceBetweenPointAndPlane(
                        satelliteObj.polygons->at(index), particles[i]) /
                        particles[i].speed.length();
                    Globals::debug && COUT("particle will intersect object, ttl = " <<
                        particles[i].ttl << ", timestep = " << timeStep << ", steps = "
                        << particles[i].ttl/timeStep << ", i = " << i << ", behaviour = "
                        << particles[i].behaviour);
                    continue;
                } else { // then particle will not intersect object
                    particles[i].behaviour = PARTICLE_WILL_NOT_INTERSECT_OBJ;
                    particles[i].ttl = 100; // particle will be kicked
                }
            } else {
                real distanceToCenterOfSatellite = Geometry::getDistanceBetweenPoints(
                    satelliteObj.center, particles[i]);
                resultf_(particles + i, &fieldPot, &fieldGrad); // get gradient of field
in the current point
                real electricField = satelliteObj.totalCharge/(4*M_PI*
                    VACUUM_PERMITTIVITY*distanceToCenterOfSatellite*

```

```

        distanceToCenterOfSatellite);
fieldGrad.resize(electricField); // resize gradient vector according to
current satellite charge by formula 1 in the draft
particles[i].affectField(fieldGrad,timeStep);
index = Geometry::getIndexOfPolygonThatParicleIntersects(satelliteObj ,
particles[i]);
if (index == -1 && sign(PARTICLE_CHARGE(particles[i].type)) == sign(
satelliteObj.totalCharge)) {
// particle will be repeled by satellite
particles[i].behaviour = PARTICLE_WILL_NOT_INTERSECT_OBJ;
particles[i].ttl = 100; // particle will be kicked
} else if (index != -1 && sign(PARTICLE_CHARGE(particles[i].type)) == -
sign(satelliteObj.totalCharge)) {
// particle will intersect satellite
particles[i].behaviour = PARTICLE_WILL_INTERSECT_OBJ;
particles[i].ttl = Geometry::getDistanceBetweenPointAndPlane(
satelliteObj.polygons->at(index),particles[i]) /
particles[i].speed.length();
Globals::debug && COUT("particle will intersect satellite , ttl = "
<< particles[i].ttl <<" , timestep = " << timeStep << " , steps = "
<< particles[i].ttl/timeStep <<" , i = " << i <<" , behaviour = "
<< particles[i].behaviour);
    }
}
}
//

} else { // modeling best applicable for drawing
//

if (particles[i].behaviour == PARTICLE_WILL_INTERSECT_OBJ) {
particles[i] = particles[i] + particles[i].speed*timeStep;
particles[i].ttl -= timeStep;
} else { // particles[i].behaviour == PARTICLE_HAS_UNDEFINED_BEHAVIOUR
real distanceToSatellite = Geometry::getDistanceBetweenPointAndSphere(
satelliteObj , particles[i]);
int index;
if (distanceToSatellite == 0 || // if particle is inside satellite's sphere
or too close to sphere and will be inside it soon
(distanceToSatellite < particles[i].speed.length()*timeStep &&
Geometry::doesLineIntersectSphere(Line(particles[i],particles[i].
speed),satelliteObj))) {
index = Geometry::getIndexOfPolygonThatParicleIntersects(satelliteObj ,
particles[i]);
if (index != -1) { // then particle will intersect object
particles[i].behaviour = PARTICLE_WILL_INTERSECT_OBJ;
particles[i].ttl = Geometry::getDistanceBetweenPointAndPlane(
satelliteObj.polygons->at(index),particles[i]) /
particles[i].speed.length();
Globals::debug && COUT("particle will intersect object , ttl = " <<
particles[i].ttl <<" , timestep = " << timeStep << " , steps = "
<< particles[i].ttl/timeStep <<" , i = " << i <<" , behaviour = "
<< particles[i].behaviour);
        }
} else {
real distanceToCenterOfSatellite = Geometry::getDistanceBetweenPoints(
satelliteObj.center , particles[i]);
resultf_(particles + i,&fieldPot,&fieldGrad); // get gradient of field
in the current point
real electricField = satelliteObj.totalCharge/(4*M_PI*
VACUUM_PERMITTIVITY*distanceToCenterOfSatellite*
distanceToCenterOfSatellite);
fieldGrad.resize(electricField); // resize gradient vector according to
current satellite charge by formula 1 in the draft
particles[i].affectField(fieldGrad,timeStep);
    }
}
//

} // end of modelling branch
} // end of loop

// checking all particles excluding the last one
unsigned long long end = electronsNumber + ionsNumber;
for(unsigned long long i = 0; i < end;) {

```



```

        if (particles[i].ttl <= 0) {
            if (particles[i].behaviour == PARTICLE_WILL_INTERSECT_OBJ)
                ++finalizedNumber;
            finalizeParticle(satelliteObj, particles, electronsNumber, ionsNumber, i);
            if (i != end - 1)
                memcpy(particles + i, particles + end - 1, sizeof(Particle));
            end--;
        } else {
            ++i;
        }
    }
    return finalizedNumber;
}

int main(int argc, char** argv) {
    srand(time(NULL));
    cout.precision(16);
    cout.setf(ios::fixed, ios::floatfield);

    // process arguments
    int c;
    bool modelingFlag = false;
    bool verboseFlag = false;
    bool drawFlag = false;
    char *filename = NULL;
    int testProbabilityCount = -1;
    int generativeSphereRadius = -1;
    int sleepTime = 0; //microsecond
    double printInterval = 10000.0;
    double intervalInSteps = true;
    double distanceStepCoef = 0.25;
    unsigned long long averageParticlesNumber = 10000;
    float complexDataFileFlag = false;

    while ((c = getopt(argc, argv, ":vdjamxgt:r:s:f:t:n:i:p:o:c")) != -1) {
        switch(c) {
            case 'a':
                Graphics::drawAxes = true;
                break;
            case 't':
                testProbabilityCount = atoi(optarg);
                break;
            case 'r':
                generativeSphereRadius = atoi(optarg);
                break;
            case 'f':
                File::scaleFactor = atof(optarg);
                break;
            case 's':
                sleepTime = atoi(optarg);
                break;
            case 'n':
                averageParticlesNumber = atoll(optarg);
                break;
            case 'd':
                drawFlag = true;
                break;
            case 'j':
                Globals::drawTrajectories = true;
                break;
            case 'v':
                verboseFlag = true;
                break;
            case 'm':
                modelingFlag = true;
                break;
            case 'x':
                complexDataFileFlag = true;
                break;
            case 'p':
                distanceStepCoef = atof(optarg);
                break;
            case 'o':
                Globals::modelingType = atoi(optarg);
                break;
            case 'c':
                Globals::initialCharge = strtold(optarg, NULL);
                break;
            case 'i':
                if(optarg[0] == 'i')

```

```

        { printInterval = atof(optarg + 1); intervalInSteps = true; }
        else if (optarg[0] == 's')
        { printInterval = atof(optarg + 1); intervalInSteps = false; }
        break;
    case '?':
    default:
        EXIT_ERR(usage);
    }
}
if (optind == argc) {
    EXIT_ERR(usage);
}
filename = argv[optind];
//if (generativeSphereRadius < 0) generativeSphereRadius =
    DEFAULT_GENERATIVE_SPHERE_RADIUS;

/*-----*/
// getting coordinates from file
vector<PlaneType> *coordinatesList;
if (complexDataFileFlag) {
    coordinatesList = File::getCoordinatesFromSpecialFile(filename);
} else {
    coordinatesList = File::getCoordinatesFromPlainFile(filename);
}
assert(coordinatesList != NULL);

// creating object using coordinates
Object3D satelliteObj(coordinatesList);
satelliteObj.totalCharge = Globals::initialCharge;

GenerativeSphere electronsGenerativeSphere(satelliteObj.center,
                                              ELECTRONS_GENERATIVE_SPHERE_RADIUS,
                                              satelliteObj);
GenerativeSphere ionsGenerativeSphere(satelliteObj.center,
                                        IONS_GENERATIVE_SPHERE_RADIUS,
                                        satelliteObj);

double electronsToIonsRatio = 1.*pow(ELECTRONS_GENERATIVE_SPHERE_RADIUS,3)*
    ELECTRONS_CONSISTENCE/
    (pow(IONS_GENERATIVE_SPHERE_RADIUS,3)*IONS_CONSISTENCE);
unsigned long long averageElectronsNumber = electronsToIonsRatio*averageParticlesNumber
    /(electronsToIonsRatio + 1);
unsigned long long averageIonsNumber = averageParticlesNumber/(electronsToIonsRatio +
    1);
Particle::electronTrajectoryCurrent =
    4*M_PI*pow(electronsGenerativeSphere.radius,2)*ELECTRON_CURRENT_DENSITY /
    averageElectronsNumber;
Particle::ionTrajectoryCurrent =
    4*M_PI*pow(ionsGenerativeSphere.radius,2)*ION_CURRENT_DENSITY /
    averageIonsNumber;

verboseFlag && COUT("electron trajectory current: " << Particle::
    electronTrajectoryCurrent);
verboseFlag && COUT("ion trajectory current: " << Particle::ionTrajectoryCurrent);
Globals::realToModelNumber = 4.0/3.0*M_PI*pow(ELECTRONS_GENERATIVE_SPHERE_RADIUS,3)
    *ELECTRONS_CONSISTENCE/averageElectronsNumber;

verboseFlag && COUT("real number/modeln number: " << Globals::realToModelNumber);

if (testProbabilityCount > 0) {
    // allocating memory for particles array
    verboseFlag && PRINTLN("memory allocation");
    Particle *particlesArray = (Particle*)calloc(testProbabilityCount, sizeof(Particle));

    verboseFlag && COUT("memory usage: " << testProbabilityCount*sizeof(Particle)
        /(1024*1024.0) << " MB");
    verboseFlag && PRINTLN("particles generation");
    electronsGenerativeSphere.populateArray(particlesArray, testProbabilityCount,
        PTYPE_ELECTRON, GEN_RANDOM);

    int intersectionsCounter = 0;
    verboseFlag && PRINTLN("checking for intersections");
    for(int j = 0; j < testProbabilityCount; ++j) {
        if (Geometry::doesParticlesTrajectoryIntersectObject(particlesArray[j],
            satelliteObj))
            ++intersectionsCounter;
        verboseFlag && (!(j%(testProbabilityCount/20 + 1))) && PRINT('.');
    }
    verboseFlag && PRINTLN("");
    if (verboseFlag) {

```

```

        COUT("percentage: " << intersectionsCounter << "/" << testProbabilityCount
        << " = " << intersectionsCounter/double(testProbabilityCount)*100 << '%');
    } else {
        cout << intersectionsCounter/double(testProbabilityCount) << endl;
    }
    free(particlesArray);
}

Particle *particlesArray = NULL;
double timeStep = 0;
unsigned long long maxParticlesNumber = averageParticlesNumber*1.5;
unsigned long long maxElectronsNumber = electronsToIonsRatio*maxParticlesNumber/(
    electronsToIonsRatio + 1);
unsigned long long maxIonsNumber = maxParticlesNumber/(electronsToIonsRatio + 1);
unsigned long long electronsNumber;
unsigned long long ionsNumber;

GaussianDistributionGenerator electronsNumberGenerator =
    Time::getGaussianDistributionGenerator(averageElectronsNumber,
        averageElectronsNumber*0.05);
GaussianDistributionGenerator ionsNumberGenerator =
    Time::getGaussianDistributionGenerator(averageIonsNumber, averageIonsNumber*0.05);

if (modelingFlag) {
    verboseFlag && PRINTLN("particles array initialization...");
    verboseFlag && COUT("memory will be allocated: " << maxParticlesNumber*sizeof(
        Particle)/pow(1024.,2) << " MB");
    electronsNumber = averageElectronsNumber;
    ionsNumber = averageIonsNumber;
    particlesArray = (Particle*)malloc(maxParticlesNumber*sizeof(Particle));
    verboseFlag && COUT("average number of electrons: " << electronsNumber << ", ions: "
        << ionsNumber);
    verboseFlag && COUT("number of particles: " << electronsNumber + ionsNumber << endl
        << " initialization ...");

    electronsGenerativeSphere.populateArray(particlesArray, electronsNumber,
        PTYPE_ELECTRON, GEN_ON_SPHERE);
    ionsGenerativeSphere.populateArray(particlesArray + electronsNumber, ionsNumber,
        PTYPE_ION, GEN_ON_SPHERE);

    double distanceStep = satelliteObj.radius*2.0*distanceStepCoef;
    timeStep = distanceStep/ELECTRON_VELOCITY_M; // time to do step for particle with
        average velocity
    verboseFlag && COUT("distanceStep: " << distanceStep << "; timeStep: " << timeStep);
}

verboseFlag && COUT("polygons: " << satelliteObj.polygons->size());
verboseFlag && COUT("center: " << satelliteObj.center);
verboseFlag && COUT("radius: " << satelliteObj.radius);
verboseFlag && COUT("capacitance: " << satelliteObj.capacitance());

// video mode initialization
if (drawFlag) {
    // set appropriate OpenGL & properties SDL
    int width = 1200;
    int height = 900;
    Graphics::initGraphics(width, height);
}

timespec start, stop, *delta;
int framesCount = 0;
double seconds = 0;
int frames = 0;

// main program loop
unsigned long long newElectronsNumber = min<unsigned long long>(electronsNumberGenerator
    (), maxElectronsNumber);
unsigned long long newIonsNumber = min<unsigned long long>(ionsNumberGenerator(),
    maxIonsNumber);
double elapsedTime = 0.0;
double timeToPrint = printInterval;
double spacecraftCapacitance = satelliteObj.capacitance();
double surfaceCharge;
unsigned long long numberOfIntersections = 0;
if (drawFlag || modelingFlag) {
    if (modelingFlag && Globals::modelingType != 1) {
        solveBoundaryProblem(coordinatesList, verboseFlag); // solve using fortran module
    }
    while(true) {
        if (drawFlag) {

```

```

processEvents();

clock_gettime(CLOCK_ID,&start);
Graphics::draw(satelliteObj,particlesArray,electronsNumber + ionsNumber);
clock_gettime(CLOCK_ID,&stop);

delta = Time::getTimeSpecDelta(&start,&stop);
++frames;
seconds += delta->tv_sec + delta->tv_nsec/pow(10,9);
if (seconds >= 1) {
    framesCount += frames;
    verboseFlag && COUT(frames/seconds << " fps; frames drawn: " <<
        framesCount);
    seconds = frames = 0;
}

}

if (Globals::pause)
    continue;

if (modelingFlag) {
    numberOfIntersections += processParticles(satelliteObj,particlesArray,
        electronsNumber,
        ionsNumber,timeStep,
        electronsGenerativeSphere,
        ionsGenerativeSphere);

    elapsedTime += timeStep;
    timeToPrint -= ((intervalInSteps)? 1: timeStep);
    surfaceCharge = satelliteObj.totalPlasmaCurrent*elapsedTime;
    if (timeToPrint <= 0) {
        cout << satelliteObj.totalPlasmaCurrent << " " << surfaceCharge << " "
            << surfaceCharge/spacecraftCapacitance
            << " " << elapsedTime << " " << numberOfIntersections*Globals::
                realToModelNumber << " " << satelliteObj.totalCharge
            << " " << electronsNumber << " " << ionsNumber << endl; // "
            " << realEN << " " << realIN << endl;
        (timeToPrint = printInterval);
    }
    // producing new particles if necessary
    if (electronsNumber < newElectronsNumber) {
        electronsGenerativeSphere.populateArray(particlesArray + electronsNumber
            + ionsNumber,
            newElectronsNumber - electronsNumber,
            PTYPE_ELECTRON,GEN_ON_SPHERE);
        electronsNumber = newElectronsNumber;
        newElectronsNumber = min<unsigned long long>(electronsNumberGenerator(),
            maxElectronsNumber);
    }
    if (ionsNumber < newIonsNumber) {
        ionsGenerativeSphere.populateArray(particlesArray + electronsNumber +
            ionsNumber,
            newIonsNumber - ionsNumber,PTYPE_ION,
            GEN_ON_SPHERE);
        ionsNumber = newIonsNumber;
        newIonsNumber = min<unsigned long long>(ionsNumberGenerator(),
            maxIonsNumber);
    }
}

sleepTime && usleep(sleepTime);
}

}

if (particlesArray != NULL) {
    free(particlesArray);
}

Graphics::quitGraphics(0);

return 0;
}

```