

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра інформаційних технологій та систем

**Донченко Владислав Володимирович**

**ДОСЛІДЖЕННЯ МЕТОДІВ ПОБУДОВИ КОМП'ЮТЕРНИХ  
МОДЕЛЕЙ ГЕОМЕТРИЧНИХ ОБЛАСТЕЙ ТА ЇХ ДИСКРЕТИЗАЦІЇ**

**кваліфікаційна робота**

**здобувача вищої освіти другого (магістерського) рівня**

**освітньої програми «Комп'ютерні мережі»**

**за спеціальністю 123 Комп'ютерна інженерія**

Особистий підпис \_\_\_\_\_ Владислав ДОНЧЕНКО

Науковий керівник \_\_\_\_\_ Геннадій МОГИЛЬНИЙ,  
кандидат технічних наук, доцент  
кафедри інформаційних технологій  
та систем

Завідувач кафедри \_\_\_\_\_ Микола СЕМЕНОВ,  
кандидат педагогічних наук, доцент  
кафедри інформаційних технологій  
та систем

Полтава – 2025

## АНОТАЦІЯ

**Донченко В. В.**

**Тема:** Дослідження методів побудови комп'ютерних моделей геометричних областей та їх дискретизації.

**Спеціальність:** 123 «Комп'ютерна інженерія».

**Установа:** ЛНУ імені Тараса Шевченка, 2025 р.

**Магістерська робота містить:** 74 с., 37 рис., 1 табл., 1 додат., 44 джерел.

**Об'єкт дослідження** – геометричне моделювання конструкцій та дискретизація геометричних областей на скінченні елементи.

**Предмет дослідження** – побудова моделі геометричних областей конструкції та генерація розрахункових сіток для скінченно – елементного моделювання конструкцій.

**Мета роботи** – дослідження методів побудови комп'ютерних моделей геометричних областей та їх дискретизації на скінченні елементи заданої форми та розробка препроцесора для скінченно-елементного моделювання конструкцій.

**Результати роботи.** У роботі було проведено дослідження найпоширеніших програмних комплексів для моделювання та аналізу складних інженерних конструкцій, як вітчизняних, так і зарубіжних. Розглянуто основні методи побудови геометричних моделей об'єктів і поширені формати, що використовуються для їх опису. Також були проаналізовані основні підходи, які застосовуються в сучасних системах автоматизованого проектування для твердотільного моделювання геометричних об'єктів, а також досліджені поширені методи та алгоритми дискретизації плоских та просторових областей.

Було здійснено аналіз процесу розробки препроцесора з метою побудови геометрії та автоматизації створення дискретної (скінченно-елементної) моделі конструкцій. Також було проведено моделювання та надано детальний опис архітектури розробленого додатку. Виконано обґрунтування вибору середовища розробки додатку та мови програмування. Описано алгоритм побудови геометрії розрахункових областей та генерації сіток у препроцесорі. У препроцесорі було розроблено два модулі для роботи: модуль "Геометрія" для створення простих геометричних сутностей та модуль "Сітка" для створення 1-, 2- та 3-мірних сіток і їх оптимізації. Для реалізації програми була обрана мова програмування C++ у середовищі Microsoft Visual Studio 2022.

**Висновок.** В результаті роботи було розроблено препроцесор для побудови геометрії та автоматизації створення дискретної (скінченно-елементної) моделі конструкцій.

**Ключові слова.** МЕТОД СКІНЧЕНИХ ЕЛЕМЕНТІВ, СИСТЕМА АВТОМАТИЗОВАНОГО ПРОЕКТУВАННЯ, ДИСКРЕТИЗАЦІЯ, ТРІАНГУЛЯЦІЯ, АЛГОРИТМ, ОПТИМІЗАЦІЯ СІТКИ, MS VISUAL C++, OPENGL.

## ABSTRACT

**Donchenko Vladyslav**

**Theme:** Study of methods of building computer models of geometric areas and their discretization.

**Speciality:** 123 "Computer Engineering"

**Institution:** Luhansk Taras Shevchenko National University (LTSNU), 2025.

**Master's work of:** 74 pages, 37 Fig., 1 Table, 1 adj., 44 source.

**A research object is** geometric modeling of structures and discretization of geometric regions into finite elements.

**The article of research is** construction model construction and generation of calculation grids for finite element modeling of structures.

**An aim of work is** research of methods of building computer models of geometric areas and their discretization into finite elements of a given shape and development of a preprocessor for finite element modeling of structures.

**Job performances.** In the work, a study of the most common software complexes for modeling and analysis of complex engineering structures, both domestic and foreign, was conducted. The main methods of building geometric models of objects and common formats used for their description are considered. The main approaches used in modern automated design systems for solid-state modeling of geometric objects were also analyzed, as well as common methods and algorithms for discretization of flat and spatial areas were studied.

An analysis of the preprocessor development process was carried out in order to build geometry and automate the creation of a discrete (finite element) model of structures. Modeling was also carried out and a detailed description of the architecture of the developed application was provided. The justification of the choice of the application development environment and programming language has been made. The algorithm for constructing the geometry of calculation areas and generating grids in the preprocessor is described. In the preprocessor, two modules have been developed for operation: the "Geometry" module for creating simple geometric entities and the "Mesh" module for creating 1-, 2- and 3-dimensional meshes and their optimization. To implement the program, the C++ programming language was chosen in the Microsoft Visual Studio 2022 environment.

**Conclusions.** As a result of the work, a preprocessor was developed for building geometry and automating the creation of a discrete (finite element) model of structures.

**Keywords.** FINITE ELEMENT METHOD, AUTOMATED DESIGN SYSTEM, DISCRETIZATION, TRIANGULATION, ALGORITHM, MESH OPTIMIZATION, MS VISUAL C++, OPENGL.

## ЗМІСТ

<b>ВСТУП.....</b>	<b>6</b>
<b>РОЗДІЛ 1 ОГЛЯД СУЧАСНИХ МЕТОДІВ МОДЕЛЮВАННЯ ТА ДИСКРЕТИЗАЦІЇ ПРОСТОРОВИХ ОБ'ЄКТІВ.....</b>	<b>13</b>
1.1. Огляд основних способів представлення геометричних тіл.....	14
1.2. Опис топології області.....	21
1.2.1. Дискретизація плоских областей.....	23
1.2.2. Дискретизація тривимірних областей.....	34
1.3. Дослідження способів генерації сіток.....	37
1.4. Огляд існуючих програмних пакетів, призначених для дискретизації складних просторових областей.....	43
1.5. Висновки до розділу.....	48
<b>РОЗДІЛ 2. МОДЕЛЮВАННЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРЕПРОЦЕСОРА.....</b>	<b>49</b>
2.1. Логічне представлення статичної моделі структури програмної розробки.....	49
2.2. Логічне представлення моделі поведінки програмної розробки.....	58
2.3. Фізичне представлення моделі програмної розробки.....	64
2.4. Архітектура програмного забезпечення препроцесора.....	65
2.5. Висновки до розділу.....	66
<b>РОЗДІЛ 3 РОЗРОБКА ПРЕПРОЦЕСОРА ДЛЯ СКІНЧЕННО- ЕЛЕМЕНТНОГО МОДЕЛЮВАННЯ.....</b>	<b>67</b>
3.1. Обґрунтування вибору середовища розробки препроцесора.....	67
3.2. Розробка інтерфейсу.....	74
3.3. Побудова геометричних об'єктів в препроцесорі.....	82
3.4. Генерація сітки в препроцесорі.....	87
3.5. Формати файлів в препроцесорі.....	90
3.6. Тестова задача.....	91
3.7. Висновки до розділу.....	97
<b>ВИСНОВКИ .....</b>	<b>98</b>

<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....</b>	<b>100</b>
<b>ДОДАТОК А.....</b>	<b>105</b>

## ВСТУП

Дослідники та інженери стикаються з багатьма завданнями, які не можуть бути аналітично вирішені або вимагають великих витрат на експериментальне втілення. Єдиним швидким способом аналізу інженерних проблем є використання комп'ютерного математичного моделювання.

Прогрес у розробці чисельних методів значно розширив коло завдань, які можуть бути проаналізовані. Результати, отримані на основі цих методів, використовуються практично в усіх галузях науки і техніки. У аналізі конструкцій найбільше застосування має метод скінчених елементів (МСЕ).

Поява методу скінчених елементів пов'язана з неможливістю аналітичного розв'язання більшості задач механіки деформованого твердого тіла і механіки руйнування. При розв'язанні задач МСЕ досліджуваний об'єкт представляється як сукупність дискретних підобластей - скінчених елементів (СЕ). У середині кожного скінченого елемента шукана неперервна величина наближається сукупністю кусково-неперервних функцій. Шукана величина в межах СЕ визначається за допомогою значень цієї величини в кінцевому числі точок досліджуваного об'єкта - вузлових точок або вузлів, які, як правило, є характерними точками СЕ. При розв'язанні задачі шукана величина описується аналітичними залежностями, які моделюють властивості матеріалу (неперервність, ізотропність і т. д.) і закони механіки (закон рівноваги сил, варіаційні принципи). Це дозволяє отримати однозначний розв'язок задачі за заданими початковими і граничними умовами.

Розрахунок конструкцій за допомогою методу скінчених елементів можна розбити на три взаємопов'язаних послідовних процеси:

1. Підготовка початкових даних: це включає скінчено-елементну дискретизацію об'єкта, який підлягає розрахунку, визначення його топології та кінематичних і силових граничних умов, а також фізико-механічних характеристик матеріалу.
2. Чисельний розрахунок скінчено-елементної моделі: цей процес включає обчислення коефіцієнтів матриці жорсткості скінчених

елементів, формування глобальної системи розв'язуваних рівнянь і її розв'язання.

3. Обробка результатів розв'язання: на цьому етапі обчислюються параметри напружено-деформованого та температурного стану конструкції, а їх результати представляються у вигляді таблиць, графіків або двовимірних/тривимірних зображень.

Ці процеси чисельної реалізації, як це зазвичай відбувається в автоматизованих розрахунках, виконуються трьома підсистемами: препроцесором, процесором і постпроцесором.

Препроцесор є однією з головних складових будь-якого програмного комплексу чисельного аналізу. Він автоматизує процес побудови геометричної моделі об'єкта дослідження та його дискретизації на скінченні елементи. Якість препроцесора має велике значення для загальної якості програмного комплексу в цілому.

Однією з головних вимог до препроцесора є його здатність ефективно та точно моделювати геометричну форму досліджуваного об'єкта. Він повинен мати можливість створювати, редагувати та маніпулювати геометричними об'єктами, такими як точки, лінії, поверхні та об'єми. Крім того, препроцесор повинен забезпечувати можливість встановлення граничних умов, визначення фізико-механічних властивостей матеріалу та встановлення параметрів розрахункової моделі.

Препроцесор для моделювання конструкцій повинен включати наступні функції:

1. Геометричне моделювання: препроцесор повинен мати можливість створювати або імпортувати геометричну модель конструкції. Це може включати створення геометричних форм, визначення розмірів, розташування вузлів та з'єднань.
2. Матеріальні властивості: препроцесор повинен дозволяти встановлювати матеріальні властивості еластомерних матеріалів, такі як модуль пружності, міцність, деформаційні характеристики

та інші. Це дозволяє враховувати особливості матеріалу при моделюванні поведінки конструкції.

3. Дискретизація: препроцесор повинен забезпечувати можливість розбиття геометричної моделі на скінчені елементи. Він повинен дозволяти вибирати типи елементів, встановлювати їх параметри та формувати зв'язки між елементами. Це важливо для створення скінчено-елементної мережі, яка відповідає властивостям конструкції.
4. Граничні умови: препроцесор повинен дозволяти встановлювати граничні умови для конструкції, такі як закріплення, прикладення сил, температурні умови тощо. Це включає встановлення граничних умов для окремих вузлів або груп вузлів.
5. Підготовка вхідних файлів: препроцесор повинен забезпечувати генерацію вхідних файлів, які містять інформацію про геометрію, матеріальні властивості, дискретизацію, граничні умови та інші параметри. Ці файли потім можуть бути використані для чисельного розрахунку.
6. Перевірка якості моделі: препроцесор повинен мати функціонал для перевірки якості скінчено-елементної моделі. Це може включати перевірку на наявність необхідних з'єднань між елементами, перевірку граничних умов, переконання у відповідності моделі реальній геометрії та інші перевірки.
7. Візуалізація та відображення результатів: препроцесор повинен мати можливість відображати геометричну модель конструкції, скінченну елементну мережу та інші атрибути моделі. Він також повинен дозволяти відображати результати чисельного розрахунку, такі як напруження, деформації, температурний стан тощо. Це допомагає аналізувати та інтерпретувати результати моделювання.



8. Експорт та імпорт даних: препроцесор повинен мати можливість експортувати скінчено-елементну модель та інші дані у різних форматах, які можуть бути використані іншими програмами для подальшого аналізу або обробки. Він також повинен підтримувати імпорт даних з інших програм для використання в препроцесорі.
9. Зручний інтерфейс користувача: препроцесор повинен мати інтуїтивно зрозумілий та зручний інтерфейс користувача, який дозволяє легко виконувати всі необхідні функції. Це допомагає користувачеві швидко і ефективно створювати та налаштовувати скінчено-елементну модель.
10. Підтримка різних типів конструкцій: препроцесор повинен бути здатний моделювати різні типи конструкцій, включаючи тверді тіла, пластини.
11. Обробка мережевих вузлів: препроцесор повинен мати функціонал для обробки мережевих вузлів, які використовуються в певних типах конструкцій, наприклад, в електричних мережах або трубопроводах. Це може включати визначення вузлів, розташування, параметри з'єднання та інші атрибути.
12. Параметризація моделі: препроцесор повинен мати можливість параметризувати модель конструкції, що дозволяє змінювати значення різних параметрів, таких як розміри, матеріальні властивості, граничні умови і т. д. Це дозволяє проводити аналіз моделі при змінних параметрах і оптимізацію конструкції.
13. Імпорт геометрії: препроцесор повинен мати можливість імпортувати геометрію з різних форматів файлів, таких як CAD-формати (наприклад, STEP, IGES) або формати графічних об'єктів (наприклад, OBJ, STL). Це спрощує процес створення геометричної моделі конструкції і дозволяє використовувати наявну геометричну інформацію.

14. Автоматична генерація мережі: препроцесор може мати можливість автоматично генерувати скінчено-елементну мережу на основі заданої геометрії і параметрів. Це дозволяє швидко створювати деталізовану модель з необхідною кількістю елементів.

Однією з ключових проблем, що виникають при використанні методу скінчених елементів, є побудова дискретної моделі механічної системи, яка підлягає дослідженню. Використання препроцесора покращує ефективність та надійність процесу моделювання, дозволяючи швидко вносити зміни в модель, виконувати різноманітні розрахунки та аналізувати результати. Це робить його незамінним інструментом для інженерів та дослідників, зайнятих розробкою та аналізом конструкцій. Тому розробка препроцесора для скінчено-елементного моделювання конструкцій є актуальною темою.

**Об'єкт дослідження** – геометричне моделювання конструкцій та дискретизація геометричних областей на скінчені елементи.

**Предмет дослідження** – побудова моделі геометричних областей конструкції та генерація розрахункових сіток для скінчено – елементного моделювання конструкцій.

**Мета роботи** – дослідження методів побудови комп'ютерних моделей геометричних областей та їх дискретизації на скінченні елементи заданої форми та розробка препроцесора для скінчено-елементного моделювання конструкцій.

**Методи дослідження:** методи обчислювальної математики і комп'ютерної графіки.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- Провести аналіз обчислювальних комплексів на основі МСЕ;
- Проаналізувати методи побудови геометричних моделей об'єктів і поширені формати їх опису;

- Проаналізувати основні поширені методи й алгоритми дискретизації плоских та просторових областей;
- Проведення моделювання та аналізу програмного забезпечення препроцесора з метою побудови геометрії та автоматизації створення дискретної (скінчено-елементної) моделі конструкцій;
- Розробити і реалізувати препроцесор для побудови геометрії та автоматизації створення дискретної (скінчено-елементної) моделі конструкцій.

Практичною цінністю роботи є розроблений препроцесор для побудови геометрії та автоматизації створення дискретної (скінчено-елементної) моделі конструкцій. У першому розділі було проведено дослідження найпоширеніших програмних комплексів для моделювання та аналізу складних інженерних конструкцій, як вітчизняних, так і зарубіжних. Розглянуто основні методи побудови геометричних моделей об'єктів і поширені формати, що використовуються для їх опису. Також були проаналізовані основні підходи, які застосовуються в сучасних системах автоматизованого проєктування для твердотільного моделювання геометричних об'єктів, а також досліджені поширені методи та алгоритми дискретизації плоских та просторових областей. У другому розділі було здійснено аналіз процесу розробки препроцесора з метою побудови геометрії та автоматизації створення дискретної (скінчено-елементної) моделі конструкцій. Також було проведено моделювання та надано детальний опис архітектури розробленого додатку. У третьому розділі виконано обґрунтування вибору середовища розробки додатку та мови програмування. Описано алгоритм побудови геометрії розрахункових областей та генерації сіток у препроцесорі. У препроцесорі було розроблено два модулі для роботи: модуль "Геометрія" для створення простих геометричних сутностей та модуль "Сітка" для створення 1-, 2- та 3-мірних сіток і їх оптимізації. Для реалізації програми була обрана мова програмування C++ у середовищі Microsoft Visual

Studio 2022. При розробці засобів відображення та візуалізації використовується графічний інтерфейс OpenGL.

# РОЗДІЛ 1

## ОГЛЯД СУЧАСНИХ МЕТОДІВ МОДЕЛЮВАННЯ ТА ДИСКРЕТИЗАЦІЇ ПРОСТОРОВИХ ОБ'ЄКТІВ

Більшість чисельних методів дослідження напружено-деформованого стану тіла, базуються на ідеї переходу від континуального завдання до дискретного, коли досліджувана суцільна область замінюється деякою кінцевою дискретною моделлю. У МСЕ безперервна область замінюється деякою сукупністю кінцевих елементів, що заповнюють увесь об'єм тіла, що не перетинається.

Одна з головних проблем, що виникають при застосуванні МСЕ, – це побудова дискретної моделі досліджуваної механічної системи. Тому, однією з головних частин будь-якого програмного комплексу чисельного аналізу, як уже згадувалося, є препроцесор – програма, що автоматизує побудову геометричної моделі досліджуваного об'єкту з подальшою її дискретизацією на кінцеві елементи. Від якості препроцесора багато в чому залежить і якість усього програмного комплексу в цілому.

Проблема оптимальної дискретизації досліджуваної області на кінцеві елементи в загальному вигляді є дуже складною (особливо для тривимірних областей). Це обумовлено тим, що на форму СЕ накладаються два основні обмеження : вони не повинні мати занадто малих (чи відповідно занадто великих) кутів і об'єм СЕ не повинен перевищувати деяку наперед задану величину. У першому випадку при розрахунках виникають значні обчислювальні погрішності. У другому з'являється ризик втрати точності обчислень при значній зміні градієнта досліджуваної функції (наприклад, в зоні передбачуваного концентратора напруги).

Тому автоматична генерація СЕ -сітки є дуже складною процедурою, що є основою будь-якого кінцево-елементного пакету програм. На практиці частіше використовуються СЕ у формі трикутника, прямокутника, тетраедра або паралелепіпеда, оскільки вони дозволяють з високою мірою точності апроксимувати область довільної форми.

## **1.1. Огляд основних способів представлення геометричних тіл**

Загальна класифікація основних підходів до моделювання геометричних тіл була дана в роботі [21], там же були сформульовані і перераховані нижче основні вимоги до твердотілих моделей:

- Показність – модель повинна бути придатна для опису безлічі фізичних об'єктів досить широкого класу;
- Однозначність – один і тільки один об'єкт повинен відповідати кожному конкретному опису, щоб не виникало питання про те, який власне об'єкт представляється;
- Унікальність – в ідеалі бажано, щоб кожен модельований об'єкт описувався в обраній схемі подання єдиним чином. Ця властивість забезпечує легкість розрізнення двох різних об'єктів, але на практиці є важко досяжною;
- Точність – бажано, щоб подання точно описувало форму об'єкта, без апроксимації;
- Коректність – в ідеалі, схема подання повинна допускати введення тільки тих описів, які задовольняють всім критеріям визначення твердотілих моделей;
- Замкнутість – подання має бути замкнутим щодо допустимих в ньому операцій. Для геометричного моделювання велике значення мають геометричні перетворення та теоретико - множинні операції;
- Компактність – модель повинна мати компактний опис, економне з точки зору даних, необхідних для її повного завдання;
- Ефективність – модель повинна допускати ефективні алгоритми її обробки, що охоплює введення/висновки, обчислення основних відносин і операцій, редагування і модифікацію, візуалізацію, обчислення метричних характеристик і т.д.

Задовольнити всім цим вимогам одночасно важко, тому різні схеми подання будуються на основі деякого компромісу і мають свої переваги і недоліки.

В даний час можна виділити наступні найбільш поширені методи представлення геометричних тіл [4, 21, 22, 23]:

- параметризовані примітиви,
- граничне уявлення;
- конструктивна геометрія;
- кінематичний метод ("свіппінг" або замітання);
- розкладання на елементи;
- просторове перерахування;
- неявні моделі.

Моделювання за допомогою бібліотеки параметризованих примітивів застосовується зазвичай в різних прикладних областях, де набір використовуваних геометричних об'єктів обмежений і стандартизований. Такий підхід відповідає груповій технології, застосовуваний в автоматизованому проектуванні. Він часто використовується для таких складних і в той же час стандартизованих об'єктів, як болти, зубчасті шестерні і т.п., які втомливо визначати за допомогою булевих комбінацій більш простих об'єктів, але можна охарактеризувати набором високорівневих параметрів (діаметр і кількість зубів для шестерні і так далі).

При використанні граничного опису геометричне тіло задається замкнутою поверхнею, що обмежує це тіло. При цьому для опису форми області використовують як алгебраїчні поверхні 1-го і 2-го порядку, так і кусково-аналітичні поверхні. За допомогою граничного опису може бути представлений широкий клас об'єктів. У граничному поданні в явному вигляді міститься інформація про поверхні тіла, дане подання ефективно при візуалізації та чисельному моделюванні. Основним недоліком методу є громіздкість даних, що описують модель і складність обчислення теоретико - множинних операцій.

У методі конструктивної геометрії складний об'єкт формується шляхом виконання теоретико-множинних операцій та операцій геометричних перетворень над простішими об'єктами, званими базовими елементами, або

примітивами. Для того, щоб результат застосування теоретико-множинних операцій до твердотілих примітивів сам був твердим тілом, виконується його регуляризація, яка зводиться до замикання внутрішніх точок. Таким чином визначаються регуляризовані теоретико-множинні операції [4].

Модель конструктивної геометрії може бути описана за допомогою дерева побудови [24], в якому нетермінальні вузли представляють оператори, а листя – базові елементи. Метод конструктивної геометрії охоплює широке коло об'єктів, його зручно використовувати при введенні, тому що в ньому інформація про об'єкт представлена в досить структурованій формі, що забезпечує велику наочність і дозволяє уникнути помилок при описі об'єкта. Крім того, подання за допомогою дерева побудови ефективно при редагуванні. Існують алгоритми візуалізації тіл, представлених безпосередньо за допомогою методу конструктивної геометрії, проте часто на етапі отримання зображення здійснюють перехід до граничного опису.

У кінематичному методі [25] двовимірна область представляється як слід рухомої у просторі кривої, а тривимірна область – як слід рухомого двовимірного тіла або перетину. Подальшим розвитком кінематичного методу є, так званий, плазовий метод, в якому об'єкт, що рухається по складній траєкторії, не є жорстким, а деформується відповідно до залежностей тій чи іншій мірі складності. Кінематичний метод зручний при введенні, дозволяє ефективно виконувати ряд обчислювальних операцій, проте він охоплює обмежений клас об'єктів і не передбачає виконання теоретико -множинних операцій.

У методі розкладання на елементи модельований об'єкт представляється як об'єднання деякого набору неперекриваючихся елементів – примітивів. Цей метод близький до методу конструктивної геометрії, проте у ньому, в порівнянні з останнім, звужене коло перетворень, виконуваних над примітивами в процесі завдання об'єкта.



Дане обмеження значно ускладнює формування опису об'єкта, однак при цьому спрощується виконання операції розбиття, що ефективно для візуалізації та чисельного моделювання.

У методі просторового перерахування об'єкти задаються шляхом перерахування всіх тих позицій простору, які вони займають. При цьому вважають, що простір складено з елементарних осередків, що примикають один до одного, а об'єкт представлений як об'єднання деякої кінцевої безлічі таких осередків. В якості осередків використовують зазвичай замкнуті квадрати (куби) фіксованого розміру зі сторонами, паралельними координатним осям (площинам). Відомі дві різновидності методів просторового перечислення – воксельні представлення і представлення за допомогою квадратичних (вісімкових) дерев, які відрізняються способом описання сукупності осередків, які займає модельований об'єкт. У першому із зазначених методів для цих цілей використовуються матричні структури, а в другому – ієрархічні структури: квадратичні і восьмиричні дерева [26]. Розроблено спеціальні дуже ефективні алгоритми для виконання теоретико-множинних операцій і візуалізації воксельних моделей і моделей на основі квадратичних (вісімкових) дерев.

У неявних моделях тіло задається за допомогою деякої процедури, яка дозволяє для кожної точки простору моделювання визначати її приналежність описуваному об'єкту [27, 28]. Ця процедура може бути реалізована різними способами. Наприклад, у вузлах регулярної сітки, що охоплює цілком модельований об'єкт, можна задати предикат приналежності й обчислювати приналежність інших точок за допомогою інтерполяції [29]. Для неявного опису геометричних тіл використовують також функцію відстані, значення якої в кожній точці дорівнює відстані від цієї точки до модельованого об'єкта [28]. Така функція може бути також визначена на базі воксельного подання. Нарешті, існує підхід, який в загальному вигляді неявно описує довільний геометричний об'єкт в просторі  $E_n$  за допомогою функції  $f(x_1, x_2, \dots, x_n)$  координат точок  $(x_1, x_2, \dots, x_n)$  у вигляді нерівності  $f(x_1, x_2, \dots, x_n) \geq 0$ , так що

функція  $f(x)$ , що визначає об'єкт, приймає позитивні значення в точках, що лежать в середині об'єкта, нульове – в граничних точках і негативне – в точках поза об'єктом. Такий підхід отримав назву функціонального подання (F-per) [2]. Як і в методі конструктивної геометрії, складний F-per об'єкт, може формуватися з простих за допомогою теоретико-множинних операцій і геометричних перетворень.

Аналітичний вид функції, що описує результат теоретико-множинних операцій, може бути знайдений на основі теорії R- функцій [30]. Застосування R- функцій дозволяє отримувати опис поверхні складного геометричного тіла в неявному вигляді  $f(x, y, z) = 0$ . В даний час розроблені методи параметризації таких поверхонь [31, 32], що забезпечують побудову графічних відображень тіл, представлених за допомогою функціональних моделей. Функціональне уявлення узагальнює різні способи неявного завдання геометричних тіл, в рамках нього реалізуються різноманітні операції, в тому числі замітання, декартовий твір, метаморфозіс та ін.

Аналізуючи описані уявлення з точки зору зазначених вище вимог, що пред'являються до твердотілих моделей, можна відзначити наступне. Серед перерахованих уявлень низьку точність мають методи просторового перерахування, точність граничних уявлень, конструктивних моделей і розкладання на елементи залежить від форми сегментів поверхонь і об'ємних примітивів. При цьому найбільш широке коло об'єктів може бути представлене методами просторового перерахування, граничним поданням, функціональним і конструктивним методами. Методи замітання і розкладання на елементи, а також представлення екземплярами примітивів вельми обмежені в сенсі безлічі представимих тіл. Унікальність уявлення практично гарантують тільки воксельні моделі та вісімкові дерева при додатковому підрозбитті. Серед всіх уявлень, найбільш важко оцінити коректність для граничного подання, де не тільки структури даних, що представляють вершини, ребра і грані можуть бути суперечливими, але також грані або ребра можуть перетинатися. Складність аналізу модельованого об'єкта на предмет

відповідності визначенню твердого тіла характерна і для функціонального подання. Більш легко перевірити коректність для конструктивної геометрії і розкладання на елементи. Найбільш просто оцінити коректність моделей просторового перерахування. Регуляризовані теоретико - множинні операції не визначені для подання екземплярами примітивів, кінематичних моделей і розкладання на елементи. Решта моделей замкнуті щодо цих операцій.

Найбільш компактний опис мають уявлення на базі параметричних примітивів, функціональні, конструктивні та кінематичні моделі. Описи моделей просторового перерахування громіздкі, але мають просту структуру. Найбільш складно і громіздко описуються граничні подання. Ефективність реалізації операцій сильно відрізняються у різних геометричних моделей.

Візуалізація та чисельні розрахунки найбільш просто реалізуються для моделей просторового перерахування і розкладання на елементи.

Полігонізація поверхні, необхідна для відображення 3D об'єктів, найбільш просто будується для граничних моделей і кінематичних моделей. Обчислення кордону є трудомісткою операцією для конструктивної геометрії і особливо для функціонального подання.

Що ж до регуляризованих булевих операцій, то вони природним чином виконуються для конструктивної і функціональної моделей, порівняно легко обчислюються для воксельних моделей і моделей на основі вісімкових дерев і значно більш складно реалізуються для граничних моделей. Питання про приналежність точки модельованого об'єкту найбільш просто вирішується для функціонального уявлення і методу просторового перерахування. Набагато складніше це завдання вирішується для інших моделей.

Зазначена класифікація відображає принципові підходи до моделювання тіл. В рамках кожного з перерахованих типів існує безліч конкретних моделей, що відрізняються способом реалізації. Жоден із зазначених способів не може вважатися повністю універсальним, ефективність тієї чи іншої моделі залежить від операцій, які необхідно виконувати над об'єктами в процесі моделювання. Тому вибір моделі залежить від прикладної задачі.

Перераховані методи опису геометричних тіл доповнюють один одного, тому найбільш ефективним часто виявляється спільне використання декількох уявлень, що припускає їх взаємні перетворення. Проте реалізації моделей різних типів сильно відрізняються по використовуваному математичному апарату, структурам даних і алгоритмам обробки.

При вирішенні багатьох прикладних задач виникає необхідність розбиття модельованих об'єктів і побудови дискретних моделей, які з заданою точністю апроксимують форму вихідного об'єкта. У дискретному поданні важливу роль відіграють засоби опису внутрішньої структури модельованих об'єктів. Для опису дискретних моделей використовуються різні стратифікації, їх огляд подано в роботі [33].

Загалом стратифікація – це опис об'єкта у вигляді об'єднання сукупності непересічних страт, кожна страта є різноманіттям у просторі  $E_n$ , кордон кожної зв'язної компоненти страти має розмірність нижче, ніж розмірність самої страти і в кожній обмеженій множині простору моделювання міститься кінцеве число страт [34]. Окремими випадками стратифікацій є геометричні комплекси [35, 36, 37].

У рамках наведеної вище класифікації дискретні моделі можна віднести до типу розкладання на елементи. Опис просторової структури також важливий при завданні складових кривих і поверхонь, кусочно-аналітичному поданні граничних моделей і при завданні розмірно неоднорідних об'єктів. Для цих цілей також застосовуються різні стратифікації [39, 40]. Зокрема, в роботі [40] було показано, що геометричні комплекси дозволяють єдиним чином представляти граничні моделі, кінематичні моделі та моделі просторового перерахування. Відзначимо також, що в залежності від способу опису примітивів в конструктивному представленні його можна звести відповідно до граничної моделі, моделі просторового перерахування або функціональної моделі. Таким чином, в якості основної альтернативи у виборі уявлення геометричних тіл можна розглядати модель на основі комплексів, звану також клітинною, і функціональне уявлення. Перша з цих моделей задає

явний опис об'єкта, а друга - неявний. Ці моделі принципово відрізняються, однак вони не заміняють один одного, кожна з них має свої переваги і недоліки. Так функціональне уявлення забезпечує компактний опис складної, можливо багатовимірної, геометрії, воно придатне для опису об'єктів різної розмірності, включаючи розмірно неоднорідні об'єкти та об'єкти, які не є різноманітні. Однак воно не містить даних про топологічну структуру об'єкта, що викликає проблеми при виконанні багатьох чисельних процедур і операцій. У свою чергу клітинне уявлення, засноване на топологічному розбитті, дає повний опис топологічної структури об'єкта, дозволяє виділяти топологічно однорідні компоненти в складі складних об'єктів. Однак ступінь деталізації клітинного уявлення, що визначається топологічним розбиттям, часто виявляється надлишковою з точки зору опису геометричних властивостей. У додатках, які не використовують повною мірою інформацію про топологічну структуру об'єкта, така надмірна деталізація знижує ефективність роботи з геометричним об'єктом.

Враховуючи вище викладене, можна зробити висновок, що при моделюванні складних неоднорідних об'єктів різні уявлення можуть вдало доповнювати один одного. На практиці для забезпечення можливості спільного використання різних уявлень необхідно забезпечити узгодженість специфікацій різних моделей з урахуванням їх взаємних перетворень.

## **1.2. Опис топології області**

Одним з найважливіших елементів чисельного розрахунку напружено-деформованого стану (НДС) тіла, що деформується, є побудова адекватної геометричної моделі досліджуваної області. Як правило, на практиці доводиться мати справу з об'єктами дуже складної конфігурації, що істотно ускладнює побудову таких моделей. В той же час від точності побудованої геометричної моделі багато в чому залежатиме якість отриманого чисельного результату.

Нині існують різні способи опису геометрії модельованої області. Одним з найчастіше використовуваних підходів являється використання

спеціалізованих CAD-систем, що дозволяють побудувати необхідну топологічну модель, як деяку сукупність базових геометричних примітивів. Такий підхід застосовується, наприклад, в системах ANSYS, COSMOS і COSAR [12]. У препроцесорах цих систем є бібліотеки таких графічних примітивів, як точка, лінія, сплайн, ламана, коло, сфера, конус, куб та ін., над якими визначені ейлереві операції їх об'єднання, перетини і віднімання, що дозволяють задати практично довільну область. Альтернативним є підхід, що полягає в параметричному описі геометрії модельованої області за допомогою деякої мови опису топології області. Наприклад, система геометричного моделювання NETGEN [12] використовує для опису топології області мову CSG (ConstructiveSolidGeometry), що дозволяє описувати невеликі і середні плоскі і тривимірні області. У CSG в текстовому форматі ASCII можна описувати довільну просторову область, як логічну комбінацію наступних базових геометричних примітивів:

- площа;
- циліндр;
- сфера;
- еліптичний циліндр;
- еліпсоїд;
- конус;
- паралелепіпед;
- многогранник.

Геометрія об'єкту визначається як деяка сукупність ейлеревих операцій (об'єднання, перетин і доповнення) над вищеописаними примітивами.

Іншим поширеним способом опису топології тривимірних об'єктів є так званий формат стерео літографії (STL - формат). Цей формат застосовується в автоматизованих системах проєктування для опису тривимірних моделей і є для них найбільш часто-використовуваним стандартним форматом.

Інформація про об'єкт в STL – файлі включає список трикутних граней, які описують поверхню його твердотілої моделі із заданою точністю, і може

бути представлена у вигляді текстового (ASCII) або бінарного файлу. Текстове представлення STL - файлу повинне починатися ключовим словом SOLID і закінчуватися ENDSOLID. Між цими програмними дужками наводиться опис трикутників. Опис кожного трикутника включає завдання одиничного вектору нормалі, спрямованого від його поверхні, після чого слідує список тривимірних координат усіх вершин. Усі координати представлені в ортогональній декартовій системі координат і записані у вигляді дійсних чисел.

На рис. 1.1. наведений приклад опису одного трикутника в STL - форматі:

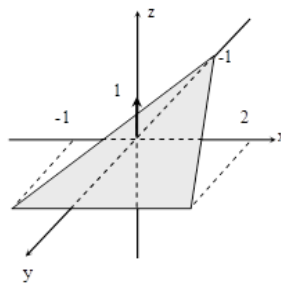


Рис. 1.1. Опис трикутника в STL -форматі

При правильному описі тріангульованої поверхні усі сусідні трикутники повинні мати по дві загальні вершини.

#### **1.2.1. Дискретизація плоских областей**

Тріангуляцією плоскої області називається її розбиття на деяку сукупність трикутників, що не перетинаються. Усі поширені алгоритми автоматичної дискретизації областей на кінцеві елементи оперують поняттям тріангуляції Делоне [15]. Тріангуляцією Делоне називається безліч трикутників, що не перетинаються, для яких виконується умова: в коло, описане біля довільного трикутника, не потрапляє жодна вершина, що належить будь-якому іншому трикутнику (рис. 1.2.).

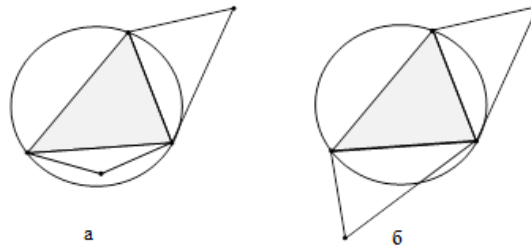


Рис. 1.2. Приклади триангуляції (а) і триангуляції Делоне (б)

Також базовим поняттям в триангуляції плоских областей є діаграма Воронова. Діаграмою Воронова для деякої безлічі точок на площині називається сукупність полігональних (многокутних) фігур, утворених лініями, які перпендикулярні відрізкам, що сполучають задані точки (рис. 1.3).

Нині розроблена велика кількість алгоритмів автоматичної генерації триангуляції. Огляд найбільш поширених алгоритмів приведений в роботі. Серед них можна виділити такі:

- алгоритм Ватсона;
- алгоритм Лавсона;
- комбінований алгоритм Ватсона і Лавсона;
- алгоритм послідовного розбиття;
- алгоритм ділення і включення;
- покроковий алгоритм;
- модифікований ієрархічний алгоритм;
- алгоритм Рапперта.

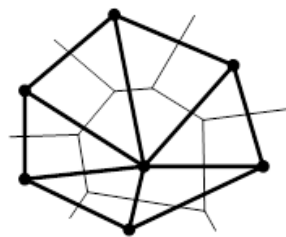


Рис. 1.3. Діаграма Воронова

Більшість з перерахованих алгоритмів базуються на ідеї побудови триангуляції Делоне для заданої на площині сукупності точок. Одним з найбільш ефективних і легких в реалізації являється комбінований алгоритм



Ватсона і Лавсона. Його суть полягає в наступному. Нехай на площині задана деяка сукупність точок. Тріангуляційна процедура послідовно вставляє кожен наступну точку, починаючи з першої, у вже існуючу тріангуляцію. Спочатку тріангуляція Делоне представлена одним єдиним так званим супер трикутником, усередині якого на початковому етапі розташовується уся задана сукупність точок. Для цього, наприклад, координати точок нормуються так, щоб вони лежали на інтервалі від 0 до 1, а координати вершин супер трикутника приймаються рівними  $(-100, -100)$ ,  $(100, -100)$  і  $(0, 100)$ .

При розгляді чергової точки  $P$ , в першу чергу, знаходиться трикутник, що містить  $P$ , а потім будуються три нові трикутники шляхом з'єднання точки  $P$  з вершинами трикутника, що обгороджує її. Після цього початковий трикутник, що обгороджує точку  $P$  видаляється, і загальне число побудованих трикутників збільшується на два.

Після обробки точки  $P$  отримане для неї розбиття перетворюється в тріангуляцію Делоне за допомогою обмінного алгоритму Лавсона. У цій процедурі всі трикутники, суміжні з протилежними до точки  $P$  ребрами, поміщаються в стек (максимальний трикутник поміщається в стек першим). Кожен поміщений в стек трикутник, піддається перевірці, в ході якої визначається, чи лежить  $P$  за межами кола, описаного біля тестованого трикутника. Якщо це випадок, коли  $P$  є вершиною трикутника, і суміжний трикутник утворює з даним опуклий чотирикутник, в якому діагональ проведена неправильно, то діагональ проводиться по-іншому. В результаті обмінної процедури Лавсона і робиться перетворення отриманої тріангуляції в тріангуляцію Делоне.

Обмінна процедура міняє два старі трикутники на два нових. Після одного обміну усі протилежні до точки  $P$  трикутники додаються в стек (це максимум два трикутники). Наступний трикутник виштовхується із стека, і увесь процес повторюється, поки стек не стане порожнім. Після цієї фази для точки  $P$  виходить нова тріангуляція Делоне. Суть обмінної процедури Лавсона

приведена на рис. 1.4. Тут слід зауважити, що якщо точка  $P$  лежить поза межами кола, то необхідно перейти до наступного трикутника в стеку.

Лавсоном було показано, що цей ітераційний алгоритм повинен побудувати триангуляцію Делоне і завершитися після останнього обміну. Практика показує, що для побудови Делоне-триангуляції не потрібно велику кількість обмінів, тому цей процес є ефективним.

Після того, як до триангуляції будуть додані усі точки, підсумкова триангуляція Делоне виходить шляхом видалення усіх трикутників, що мають в якості вершин вершини супер трикутника. Будь-яка вершина трикутника, що видаляється, не співпадаюча з вершиною супер трикутника, повинна лежати на межі триангуляції.

Оскільки вставка кожної нової точки в триангуляцію створює два нові трикутники, загальне число отриманих трикутників в триангуляції має дорівнювати  $2N+1$ , де  $N$ - число вершин, що беруть участь в триангуляції.

Тести показують, що для  $N$  довільно розташованих на площині точок розрахунковий час алгоритму складає  $O(N^{5/4})$ . Крім того, для роботи алгоритму потрібно близько  $14 N$  елементів пам'яті, використовуваних для зберігання координат точок, номерів вузлів трикутників і іншої допоміжної інформації [13].

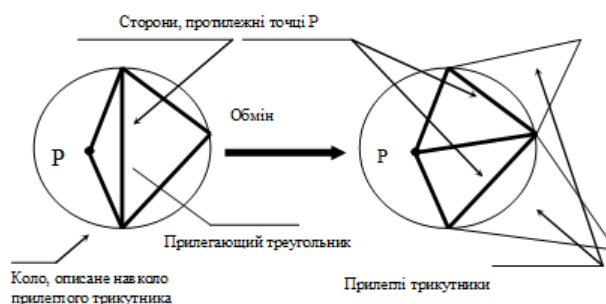


Рис. 1.4. Обмінний алгоритм Лавсона

Загальна блок-схема комбінованого алгоритму Ватсона-Лавсона приведена на рис. 1.5. Для його застосування в першу чергу необхідно задати межу області, що підлягає триангуляції, а потім опорні точки, на яких і буде побудована триангуляція. Після чого, з метою підвищення ефективності

роботи алгоритму, координати опорних вузлів нормуються і сортуються. Після тріангуляції початкові координати вузлів відновлюються.

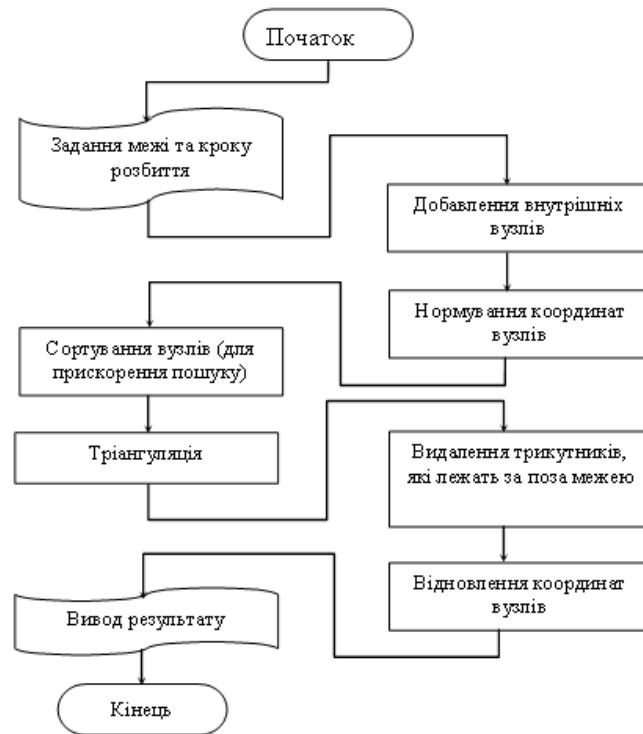


Рис. 1.5. Загальна блок-схема комбінованого алгоритму Ватсона-Лавсона

Головною труднощію, що виникає при використанні цього алгоритму, є пошук і видалення зайвих трикутників при тріангуляції багатозв'язкових або неопуклих областей (рис. 1.6.).

Ця проблема зводиться до рішення задачі про належність точки (наприклад, геометричного центру трикутника) заданому багатокутнику (межі області).

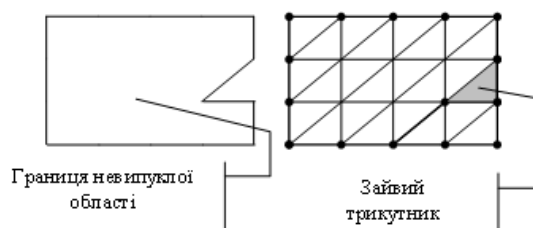


Рис. 1.6. Проблема зайвого трикутника

Відома велика кількість методів і алгоритмів рішення цієї задачі. Найбільш ефективними серед них є наступні:

- підрахунок кількості перетинів межі області променем, проведеним з тестованої точки (рис. 1.7.);

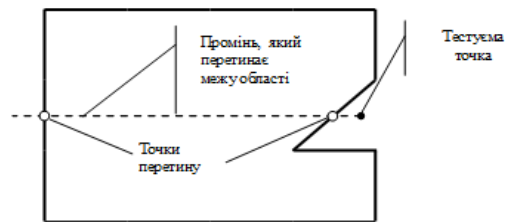


Рис. 1.7. Визначення належності точки замкнутому контуру шляхом підрахунку кількості перетинів променя, проведеного з тестованої точки, і межі області

– визначення величини кута, утвореного відрізками, що сполучають тестовану точку і сусідні вершини контура (рис. 1.8).

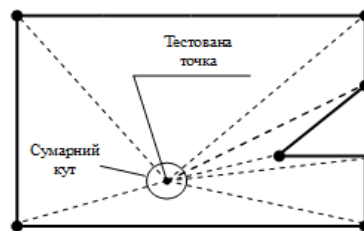


Рис. 1.8. Визначення належності точки замкнутому контуру шляхом підрахунку суми кутів

У першому випадку, якщо точка знаходиться у середині області, то кількість перетинів променя має бути непарною. А в другому - якщо точка знаходиться у середині контура, сумарний кут повинен дорівнювати  $360^0$  якщо на межі  $-180^0$ . Інакше точка знаходиться за межами області.

Проблема побудови безлічі опорних точок для тріангуляції Делоне в загальному вигляді є досить складною, оскільки необхідно враховувати як обмеження, що накладаються на форму трикутників, так і можливу наявність в геометрії тріангульованої фігури так званих сингулярностей: тріщин, розрізів, отворів, гострих кутів і тому подібне, що вимагає значного згущування сітки в їх області.

Одним з найбільш ефективних алгоритмів побудови тріангуляції Делоне, що дозволяють здолати описані вище проблеми, являється алгоритм Рапперта. Фактично цей алгоритм оптимізаційний. Він дозволяє поліпшити якість вже наявного первинного розбиття.

Ідея алгоритму Рапперта полягає в наступних двох кроках:

- а) отримання первинної («грубої») триангуляції плоскої області шляхом завдання деякої сукупності точок на її межі;
- б) оптимізація цієї триангуляції шляхом введення в сітку нових вузлів з подальшим перетворенням розбиття в триангуляцію Делоне.

Поліпшення якості кінцево-елементної мережі досягається за рахунок розбиття трикутників, що мають неправильну форму (гострі кути або велика площа) шляхом введення нових точок. При цьому величини кутів і площа елементів є параметрами алгоритму, що дозволяють управляти процесом розбиття.

При описі алгоритму Рапперт ввів терміни[16], що фактично стали в теорії триангуляції загальноприйнятими :

- елемент (element) - трикутник;
- сегмент (segment) - відрізок, що сполучає сусідні точки, що лежать на межі області;
- вузол (node) - точка, в якій сходяться ребра дотичних елементів. Вузлам відповідають точки на діаграмі Воронова (рис. 1.3.);
- ребро (edge) - відрізок, по якому граничать сусідні елементи;
- включена точка (encroachedpoint) - довільна точка поточного розбиття, що знаходиться усередині кола, радіусом якого є довільний сегмент;
- «неправильний» трикутник (badtriangle) — елемент з характеристиками (кути, площа) що не задовольняють заданим обмеженням на триангуляцію.

Алгоритм Рапперта складається з двох базових процедур:

- 1) розбиття «неправильного» трикутника шляхом введення нового вузла;
- 2) розбиття сегментів шляхом введення нового вузла.

Розбиття «неправильного» трикутника відбувається таким чином:

- а) обчислюються координати кола, описаного біля елемента, який підлягає розбиттю;
- б) у центр кола додається новий вузол;
- в) початковий елемент віддаляється і замінюється знову утвореними (рис. 1.9).

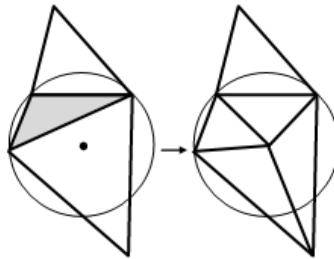


Рис. 1.9. Розбиття «неправильного» трикутника

Розбиття сегменту відбувається у тому випадку, якщо в коло, діаметром якого він є, потрапляє вузол (рис. 1.10), що не належить йому. Тоді такий «неправильний» сегмент ділиться таким чином:

- сегмент ділиться навпіл;
- у середину сегменту додається новий вузол;
- видаляється трикутник, ребром якого був початковий граничний сегмент;
- будуються нові трикутники.

Таким чином, побудова тріангуляції Делоне з використанням алгоритму Рапперта полягає в послідовному переборі усіх елементів і їх оптимізації із застосуванням двох базових процедур.

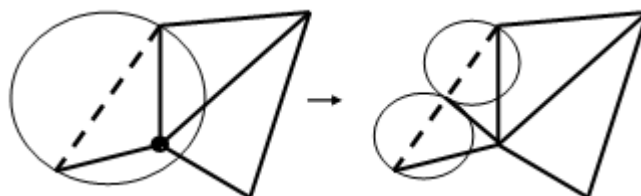


Рис. 1.10. Розбиття «неправильного» сегменту

Оригінальний алгоритм Рапперта містить тільки один параметр - критерій якості згенерованої сітки (мінімальний кут сітки). Він визначався як

мінімальний по усіх елементах кут між ребрами. Проте, при використанні МКЕ потрібно введення ще одного критерію якості сітки - максимальної площі елемента. Тому усі сучасні реалізації і модифікації алгоритму Рапперта використовують ці два критерії. Раппертом було показано, що критерій мінімального кута розбиття автоматично забезпечує згущування сітки поблизу сингулярностей, які, як правило, є концентраторами напружки.

Таким чином, загальну блок-схему застосування алгоритму Рапперта для автоматичної оптимізації тріангуляції можна зображувати таким чином (рис. 1.11).

Раппертом було також показано, що алгоритм буде стійкий і правильно працювати для мінімальних кутів  $\alpha \approx 20^\circ$ .

У оригінальному алгоритмі Рапперта явно не є присутньою процедура оптимізації якості сітки, що базується на повороті діагоналі. Суть цієї операції, як і в алгоритмі Ватсона-Лавсона, полягає в тому, що два суміжні трикутники, що мають загальну грань, утворюють чотирикутник, діагональ в якому можна провести різними способами, як це показано на рис. 1.12.

Поворот діагоналі можна використовувати для локальної оптимізації отриманого звичайно-елементного розбиття. Проте його використання вимагає наявності критерію, згідно з яким необхідно проводити цю процедуру. Найчастіше в якості такого критерію розглядаються площі елементів до повороту діагоналі і після, а також критерій мінімального кута в елементах.

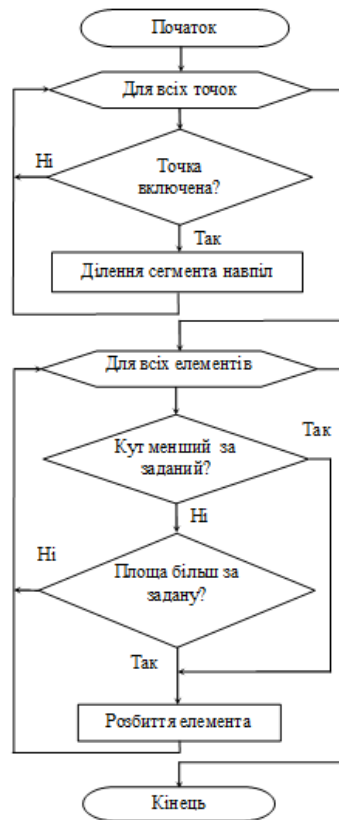


Рис. 1.11. Блок-схема алгоритму Рапперта

Поворот діагоналі зручно проводити відразу ж після виконання однієї з двох базових процедур алгоритму Рапперта. В цьому випадку точно відома область, для якої були проведені зміни в сітці, що дозволяє швидко знайти потрібні елементи і при необхідності виконати поворот діагоналі.

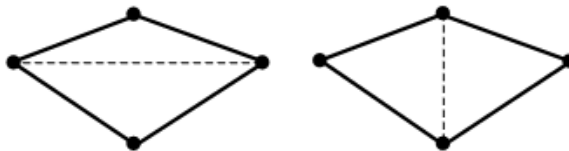


Рис. 1.12. Процедура повороту діагоналі

При застосуванні алгоритму Рапперта можливе виникнення наступних проблем. При виконанні розбиття рівнобедрених трикутників можуть виникати нові елементи з нульовою площею. Це пояснюється тим, що при попаданні центру описаного навколо трикутника кола на одну з його сторін площа одного з знову утворених трьох трикутників дорівнюватиме нулю. З рис. 1.13 видно, що після розбиття трикутника  $ABC$  утворюються три нових:  $AOB$ ,  $BCO$  і вироджений  $AOC$ .



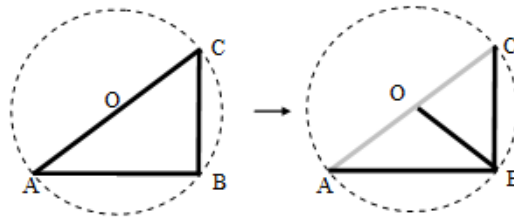


Рис. 1.13. Утворення виродженого трикутника

Одним із способів подолання цієї проблеми є додавання нової точки при розбитті трикутника не строго в центр описаного кола, а в деяку її околицю. Іншим поширеним способом є виключення виродженого трикутника з мережі.

Другою проблемою є розбиття трикутника, центр описаного біля якого кола лежить за його межами. В цьому випадку виникає проблема зациклення, оскільки точка, що додається, не змінює форму трикутника. Одним з можливих варіантів вирішення цієї проблеми є перевірка на попадання точки, що вставляється, в трикутник. Якщо точка лежить за його межами, то трикутник розбивається, наприклад, шляхом ділення навпіл його найбільшої сторони (рис. 1.14.).

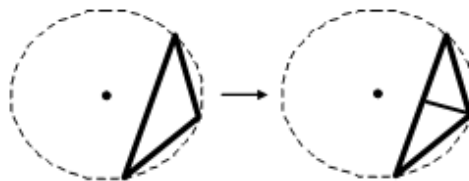


Рис. 1.14. Розбиття витягнутого трикутника

Наступною відомою проблемою, являється дискретизація областей, в геометрії яких є присутніми гострі кути. В цьому випадку можливе зациклення процедури ділення граничних сегментів, як показано на рис 1.15.

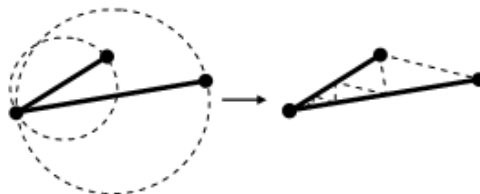


Рис. 1.15. Зациклення процедури ділення сегменту при гострому вуглі

Раппертом був запропонований спосіб вирішення цієї проблеми, що полягає в тому, що «неправильний» сегмент при гострому вуглі ділиться не посередині, а в найближчій до середини точці перетину цього сегменту і

концентричних кіл з центрами у вершині кута (рис. 1.16.). Причому радіус кожного кола в два рази більший за попередній, а радіус першого довільний, але береться досить малим відносно довжини «неправильного» сегменту.

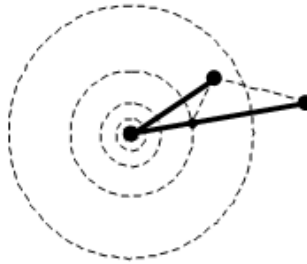


Рис. 1.16. Ділення сегменту способом концентричних кіл

Нині алгоритм Рапперта є однією з найбільш ефективних і часто використовуваних для Делоне-триангуляції плоских областей. Хоча, як було відмічено вище, формально він є усього лише алгоритмом оптимізації вже існуючого розбиття області. Тому для його застосування необхідно отримати первинну дискретизацію області на кінцеві елементи. Зробити це можна, наприклад, таким чином. На межі області з наперед заданим кроком задаються вузли, які будуть використані для побудови первинного «грубого» розбиття. Потім, використовуючи як опорні отримані вузли і застосовуючи послідовно модифікований алгоритм Ватсона-Лавсона і алгоритм Рапперта, можна безпосередньо отримати необхідну триангуляцію (рис. 1.17.).

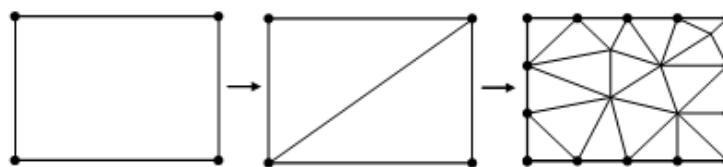


Рис. 1.17. Схема отримання початкового розбиття з подальшою триангуляцією

### 1.2.2. Дискретизація тривимірних областей

Як і в плоскому випадку, дискретизація тривимірної області на кінцеві елементи розпочинається з початкового «грубого» розбиття. Поверхня тривимірної області представляється як сукупність багатокутних областей – полігонів. Іншими словами, спочатку відбувається дискретизація поверхні тривимірної області.

Потім, практично так само, як і в двомірному випадку, усі вершини триангульованої поверхні тривимірної області вставляються всередину супертетраедра.

Дж. Шевчук запропонував модифікацію алгоритму Рапперта для тривимірного випадку. Ним, по аналогії з Раппертом, була введена наступна термінологія:

- елемент - тетраедр;
- граничний сегмент (boundarysegment) - це ребро, що належить поверхні початкового об'єкту, який підлягає дискретизації, і задане у вхідному описі геометрії області; якщо граничний сегмент розділений на частини послідовною вставкою точок, кожна його частина називається граничним підсегментом;
- гранична грань (boundaryfacet) - це плоский багатокутник (полігон), що лежить на поверхні початкового об'єкту і обмежений граничними сегментами; Якщо гранична грань розділена триангуляцією або додаванням всередину нових точок, то її внутрішні частини називаються граничними під гранями;
- екваторіальна сфера (equatorialsphere) трикутної граничної грані або підграні - це єдина сфера, екватором якої є коло, описане біля заданого трикутника.

Алгоритм Дж. Шевчука у загальних рисах містить наступні кроки:

- 1) якщо тетраедр має «неправильну» форму, тобто його найкоротша грань занадто мала в порівнянні з радіусом описаної біля нього сфери, то вставляється новий вузол в центр цієї сфери, і елемент ділиться по аналогії з «неправильним» трикутником в алгоритмі Рапперта;
- 2) якщо вершина, що вставляється, потрапляє на граничну підгрань, то вона розділяється; підграні діляться шляхом вставки вузла в центр описаних біля них екваторіальних сфер;

3) якщо вузол, що вставляється, потрапляє на граничний підсегмент, то він також розділяється.

Шевчук визначив наступний пріоритет виконання цих операцій :

- а) ділення підсегменту;
- б) ділення підграні;
- в) ділення елементів, що мають «погану» форму.

По аналогії з двовимірним випадком оптимізації отриманої сітки, коли виконується процедура повороту діагоналі, в тривимірному випадку також виконується обмінна процедура реконфігурації розбиття.

У тривимірному випадку ця процедура є складнішою. У канонічному випадку відбувається заміна двох суміжних тетраедрів на три або двох тетраедрів на два. Можлива і зворотна процедура (рис. 1.18.).

Обмінна процедура для ребер є складнішою. У ній  $N$  тетраедрів, інцидентних єдиному ребру, замінюються новим набором з  $2N - 4$  тетраедра. На рис. 1.19 наведений приклад ребра  $AB$ , перпендикулярного площині сторінки. П'ять тетраедрів, спочатку інцидентних йому ( $01AB$ ,  $12AB$ ,  $23AB$ ,  $34AB$  і  $40AB$ ), в результаті обміну були замінені шістьма новими тетраедрами, два для кожного з трикутників, що утворюють триангуляцію полігону  $01234$ :  $012A$ ,  $024A$ ,  $234A$ ,  $021B$ ,  $042B$  і  $324B$ . Після завершення цих процедур (як і в плоскому випадку), з отриманого кінцево-елементного розбиття видаляються усі елементи, що мають в якості вершин вершини супертетраедра [13].

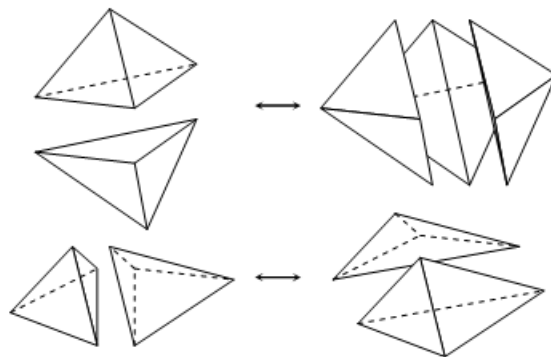


Рис. 1.18. Приклади обмінної процедури в тривимірному випадку

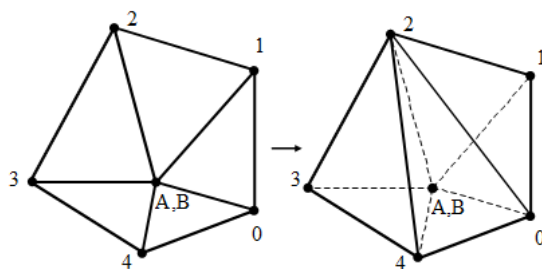


Рис. 1.19. Приклад обмінної процедури для ребер

### 1.3. Дослідження способів генерації сіток

Процес побудови сіток може розглядатися як перетворення геометричних моделей. Методи дискретизації істотно залежать від способу подання об'єкту, що розбивається, і від обмежень, накладених прикладним завданням на дискретну модель.

Огляд алгоритмів побудови сіток можна знайти зокрема в роботах [40, 41]. Серед різних способів генерації сіток можна виділити наступні групи найбільш поширених методів:

- методи тетраедрізації (тріангуляції);
- методи побудови неструктурованих гексагональних (чотирикутних) сіток;
- кінематичні методи або методи об'єднання перетинів;
- методи накладення неузгоджених з формою області сіток;
- методи відображень геометрично нерегулярних областей в області правильної форми;
- методи оптимізації.

Аналіз різних методів тріангуляції і тетраедрізації дається в роботах [42, 43]. Серед методів тріангуляції можна виділити дві основні групи. У методах першої групи вихідною інформацією для генерації сіток є набір вузлів, які потім з'єднуються, утворюючи трикутні елементи або тетраедри. Найбільш часто при цьому використовується метод Делоне, в якому вузли з'єднуються таким чином, що всередину кола (кулі), описаного навколо формованого трикутника (тетраедра), не потрапляють ніякі інші вузли сітки, відмінні від вершин даного елемента.

Існують різні способи початкового вибору вузлів. Одні автори пропонують спочатку розміщувати вузли на межі області, а потім генерувати внутрішні вузли випадковим чином, залишаючи тільки ті з них, які задовольняють заданій щільності сітки. У ряді робіт спочатку вводиться набір горизонтальних прямих, що покриває всю область рішення. Потім вузли рівномірно розподіляються на відрізках цих прямих, що лежать всередині області. У тривимірному випадку для автоматичного завдання вузлів пропонується також використовувати представлення області з допомогою вісімкових дерев.

До другої групи методів тріангуляції відносяться так звані методи декомпозиції. У методах декомпозиції можна виділити рекурсивні та ітераційні методи.

У рекурсивних методах вихідна область спочатку розбивається на підобласті, які потім діляться в відповідності з тим же самим алгоритмом розбиття. Процес продовжується до тих пір, поки розміри отриманих підобластей не відповідатимуть необхідної щільності сітки. Цей метод дозволяє будувати сітки в областях, які є багатокутниками, які в загальному випадку можуть бути і криволінійними. Вихідна груба сітка виходить шляхом побудови діагоналей, що з'єднують несуміжні вершини багатокутника. Елементи грубої сітки потім подрібнюються, для цього вводяться додаткові вузли, що розміщуються на кордонах елементів, ці вузли з'єднуються один з одним відрізками, що лежать всередині елементів. Даний метод допускає узагальнення й на тривимірний випадок.

В ітераційних методах декомпозиції розбиття здійснюється таким чином, що на кожному кроці будується один або кілька елементів результуючої сітки. Процес продовжується до тих пір, поки елементи сітки не заповнять всю розрахункову область. Методи цієї групи дозволяють будувати, як правило, трикутні і тетраедральні сітки. В ітераційних методах існує можливість управління розмірами і формою елементів безпосередньо в

процесі побудови сітки, що є їх безперечною перевагою порівняно з рекурсивними методами.

Ітераційні методи універсальні і, як правило, застосовуються для областей досить довільної форми. Саме тому ітераційні методи в основному і використовуються в автоматичних програмних комплексах. Недоліком цього класу методів є ресурсомісткість, істотно більш повільна швидкість роботи (у порівнянні з прямими методами) і менша надійність.

Ітераційні методи через свою універсальність отримали найбільший розвиток. Розроблено кілька різних підходів, які можна розділити на три підкласи: методи граничної корекції, методи на основі критерію Делоне і методи вичерпання.

Методи граничної корекції є найшвидшими з ітераційних методів, але, на жаль, мають ряд невикорінних недоліків. Побудова сіток в цих методах здійснюється в два етапи. На першому етапі проводиться триангуляція якоїсь простої "супер- області", повністю включає в себе задану область. Як правило, ця супер- область являє собою паралелепіпед (через простоту триангуляції), триангуляція якого здійснюється на основі одного з численних шаблонів. На другому етапі всі вузли отриманої сітки, що лежать поблизу кордону заданій області, проєктуються на поверхню кордону; а вузли, що лежать поза заданій області – видаляються. Щоб компенсувати неминучі геометричні спотворення елементів сітки поблизу кордонів, часто додатково проводять ще один етап – етап оптимізації сітки, що в підсумку дозволяє отримати досить хороші результати.

Очевидно, що даний метод не можна застосовувати для дискретизації областей із заданою триангуляцією кордонів. Це істотне обмеження, а також інші складності знижують популярність методу, зводячи нанівець його основну перевагу – високу швидкість роботи.

Сутність методів вичерпання полягає в послідовному "вирізанні" із заданої області фрагментів тетраедричної форми доти, поки вся область не опиниться "вичерпана". В англomовній літературі цей метод отримав назву

"advancing front", що також добре відображає ідею методу. Вихідними даними на кожній ітерації є "фронт", тобто триангуляція кордону ще не "вичерпана" частина області. Кожен трикутник цієї триангуляції є основою області тетраедра, який вилучається, причому на кожній ітерації може вилучатися або один тетраедр, або відразу цілий шар тетраедрів. Після вилучення тетраедра "фронт" оновлюється, після чого відбувається перехід до наступної ітерації.

Методи вичерпання універсальні і можуть бути використані для областей довільної форми та конфігурації (навіть для незв'язних областей), що пояснює їх популярність. Зокрема, саме ці методи використовуються в програмному комплексі ANSYS. Разом з тим слід відзначити їх високу ресурсомісткість і низьку швидкість роботи.

Методи на основі критерію Делоне часто називають просто методами Делоне, хоча це не зовсім коректно, оскільки сам Б.Н. Делоне ніяких методів не розробляв, а лише запропонував простий і ефективний критерій, що використовується при установці зв'язків між вузлами. Відповідно, ідеєю цього класу методів є розміщення в заданій області вузлів і подальша розстановка між ними зв'язків згідно з критерієм Делоне (чи іншому схожому критерію).

Неструктуровані гексагональні сітки будуються зазвичай на основі симпліціальних сіток шляхом об'єднання сусідніх елементів [40].

Кінематичний метод дозволяє будувати сітки в областях, заданих методом замітання [45]. При дискретизації області спочатку розбивається рухома крива (двовимірний розтин), а потім відповідні один одному точки (відрізки) на сусідніх за часом кривих (перетинах) з'єднуються один з одним, утворюючи елементи двовимірної (тривимірної) сітки. Кінематичний метод є окремим випадком загального методу об'єднання перерізів. У методі об'єднання перерізів передбачається відома довільна послідовність отриманих яким-небудь способом перерізів об'єкта, який розбивається. Тоді, якщо на перетинах вдається побудувати ізоморфні сітки, то з'єднуючи відповідні один одному елементи перерізів, можна отримати чотирикутну або призматичну сітку, яка є дискретизацією розглянутого об'єкта. Даний спосіб є досить



ефективним, однак він застосується до обмеженого кола об'єктів. У деяких випадках, коли сітки на сусідніх перерізу не є ізоморфними, для дискретизації простору між такими перерізами використовуються методи тріангуляції.

При використанні неузгоджених сіток вихідна область покривається деякою регулярною сіткою. Сіткові елементи всередині області залишаються без змін, а елементи, що лежать на кордоні і поза області ігноруються. У результаті виходить сітка зі ступінчастим або ламаним кордоном, для згладжування якої можуть застосовуватися різні підходи. Зокрема, проєціювання точок ступінчастого кордону на межу області, або зсув прикордонних вузлів уздовж ліній сітки до перетину з кордоном.

Ефективний підхід до побудови неузгоджених сіток пов'язаний з використанням представлення області з допомогою вісімкових дерев. Цей метод дозволяє в автоматичному режимі здійснювати дискретизацію складних тривимірних об'єктів [44].

Для генерації сіток активно застосовуються також різні відображення [45, 46], що дозволяють перетворювати сітки, поставлені на областях правильної форми, сітки, що покривають області складної форми. В даний час найбільше поширення одержали конформні відображення і трансфінитні перетворення. Методи, засновані на використанні різних відображень, що забезпечують побудову, як правило, якісних розрахункових сіток. Однак застосування подібних алгоритмів пов'язана з істотними обмеженнями на форму вихідної області і на спосіб подання цієї області, що призводить до необхідності індивідуального підходу до вирішення кожної нової задачі.

Методи відображень дозволяють будувати блочно-структуровані сітки в галузях, визначених методом розкладання на елементи, що припускає завдання галузі у вигляді об'єднання макроблоків певної форми. Для автоматичного розбиття складних областей на макроблоки застосовується також перетворення, яке дозволяє виділити кістяк об'єкта, що складається з безлічі точок, кожна з яких є рівно віддаленою від найближчих до неї точок межі об'єкта [30].

Методи оптимізації сіток використовуються для покращення та адаптації до умов поставленої задачі вже сформованих яким-небудь способом сіток [31, 32, 33]. У загальному вигляді суть цих методів можна сформулювати наступним чином:

$$\text{необхідно знайти } E = \min \left( \max_i E_i \right), i=1,2,\dots,N,$$

де  $E_i$  - якась міра помилки, що дозволяє кількісно оцінити якість кінцево-різницевої або скінчено-елементної апроксимації -  $E_i$ -помилка на  $i$ -му елементі,  $N$  - загальна кількість елементів. Міра  $E$  може бути обрана самими різними способами, вона може залежати тільки від геометричних характеристик елементів або ж враховувати характер прикладної задачі. При оптимізації сіток або варіюють тільки координатами вузлів, залишаючи топологію без зміни, або допускають зміну топології, в цьому випадку можливе розбиття або навпаки укрупнення деяких елементів, зміна конфігурації зв'язків між вузлами або введення додаткових вузлів на кордонах і всередині елементів.

Кожен з перерахованих методів побудови сіток розрахований на певну модель подання розрахункової області [34]. Так при тріангуляції, заснованої на методах декомпозиції, використовується граничний опис області. При побудові неузгоджених сіток застосовується завдання області методом просторового перерахування або граничним методом. Кінематичний метод придатний для розбиття тільки тих областей, опис яких також будується на основі кінематичного методу. Методи відображень припускають завдання розрахункових областей або методом розкладання на елементи, або граничним методом.

Слід зазначити, що застосування тих або інших способів побудови сіток істотний вплив мають обмеження, що накладаються методами, використовуваними для вирішення прикладних завдань.

Таким чином, вибір типу розрахункових сіток і алгоритмів їх побудови з одного боку залежить від прикладної задачі і методів її чисельного рішення,

а з іншого боку визначається способом завдання геометричної моделі розрахункової області.

#### **1.4. Огляд існуючих програмних пакетів, призначених для дискретизації складних просторових областей**

Сітка робить досить істотний вплив на точність рішення проекційно-сітковими методами (а в деяких випадках і на збіжність методів). Якість сітки з точки зору точності рішення визначається трьома обставинами: розмірами елементів, їх формою і тим, наскільки добре сітка апроксимує вихідну область. Таким чином, можна сформулювати такі вимоги до будь-якої системи автоматичної тріангуляції (САТ).

1. Максимально точна апроксимація кордонів області та її внутрішніх обмежень: лінії з'єднань поверхонь повинні бути апроксимовані ланцюжками ребер сітки, а самі поверхні - безліччю плоских трикутних граней.
2. Форма елементів повинна бути по можливості близька до форми правильного симплекса.
3. САТ повинна дозволяти контролювати розміри елементів сітки, щоб можна було забезпечити згущення сітки в потрібних областях.

Важливою також є можливість перевірки якості та коректності побудованої сітки.

Нижче наводиться огляд різних програмних пакетів, призначених для дискретизації складних просторових областей (вибірка обмежена програмами з відкритим кодом, повністю безкоштовними програмами та програмами, безкоштовними для академічного використання).

Пакет Geompack розроблений доктором Баррі Джо, відомим своїми роботами в області дискретизації методами на основі критерію Делоне. Перші версії пакету з'явилися більше 20 років тому і мали вид програмної бібліотеки на мові FORTRAN77 (старі версії пакету поширюються з відкритим кодом). Тепер Geompack є потужною консольною програмою для виконання різних операцій з сітками – побудови, згущення, оптимізації, перебудови і т.п. В

основі алгоритму використовується метод корекції спільно з методом на основі критерію Делоне.

Пакет Geompack безкоштовний для академічного використання. У комплекті з пакетом поставляється докладна документація англійською мовою і безліч прикладів. Автор здійснює обмежену підтримку користувачів по електронній пошті. Візуальний інтерфейс і можливості по візуалізації відсутні. Імпорт та експорт даних здійснюється через текстові та бінарні файли спеціальних (нестандартних) форматів. Пакет може бути використаний під операційними системами сімейств Unix і Windows.

До недоліків пакета слід віднести неможливість прямого контролю над розмірами елементів.

Домашня сторінка пакета в Інтернеті: <http://members.shaw.ca/bjoe/>

TetGen – програма, розроблена доктором Хан Сі з інституту Вейерштраса (WIAS). В основі алгоритму лежить метод на основі критерію Делоне. Для обліку внутрішніх обмежень використовується метод перебудови. TetGen володіє візуальним інтерфейсом (може запускатися і як консольна програма), дозволяє будувати та оптимізувати сітки, оцінювати їх якість, а також виробляти локальне згущення.

Пакет TetGen безкоштовний для академічного використання. У комплекті з пакетом поставляється докладна документація англійською мовою і безліч прикладів. Імпорт та експорт даних здійснюються через текстові файли різних (у тому числі і стандартних) форматів. Розміри елементів контролюються глобальним чином.

Пакет може бути використаний під будь-якими операційними системами, для яких існує компілятор C ++.

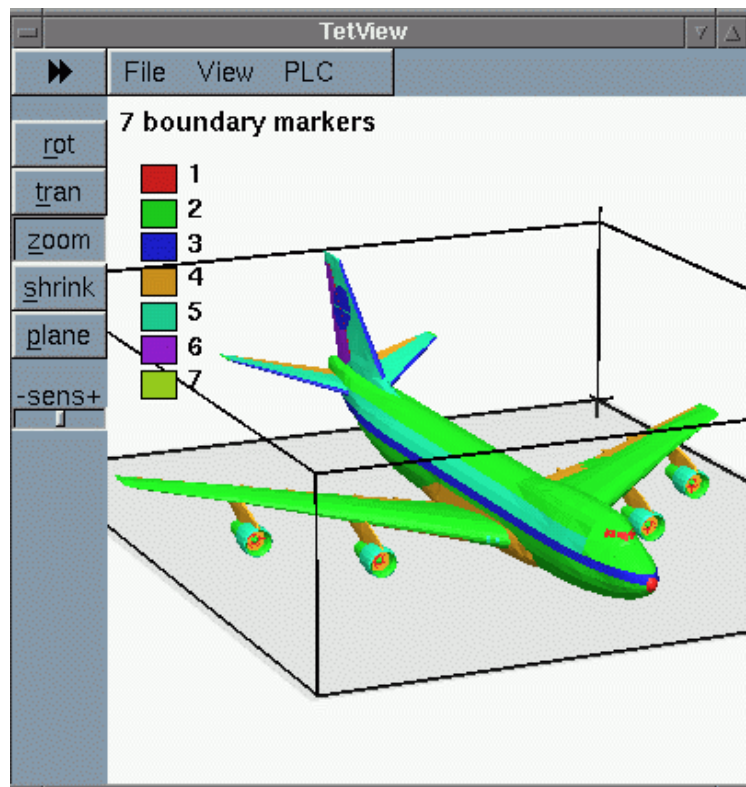


Рис. 1.20. Вікно програми TetGen

Домашня сторінка пакета в Інтернеті: <http://tetgen.berlios.de/>

Програмний комплекс "Mefisto", розроблений колективом французьких авторів під керівництвом Алана Перрона, призначений для вирішення різних завдань математичної фізики, і включає в себе модуль дискретизації, постпроцесингу, вирішувача і візуалізації. Модуль для дискретизації областей (Mefisto - maillages ) можна використовувати і окремо.

Програмний комплекс Mefisto призначений для роботи під управлінням операційних систем сімейства Unix. Поширюється разом з вихідними кодами (написаний на мовах FORTRAN77 і C) і демонстраційними прикладами. Документація французькою мовою.

Для тріангуляції поверхні і внутрішній області використовується комбінований метод на основі критерію Делоне та алгоритму Quad - tree. Управління розмірами елементів здійснюється за допомогою спеціальних функцій (тобто згущення сітки можна отримати в процесі побудови). Область для тріангуляції можна задавати за допомогою текстових файлів спеціального

формату або візуального інтерфейсу. Експорт сітки проводиться в текстові файли спеціального формату.

Домашня сторінка програмного комплексу в Інтернеті:  
<http://www.ann.jussieu.fr/~perronnet/mefistoa.gene.html>

NetGen – один з найпопулярніших безкоштовних пакетів для дискретизації складних областей. Поширюється з відкритим кодом, супроводжується документацією англійською мовою, безліччю прикладів, а також має власний Інтернет -форум. Може бути використаний під будь-якими операційними системами, для яких існує компілятор C++. Володіє візуальним інтерфейсом, але може бути запущений і як консольна програма. Дозволяє згущувати та оптимізувати сітки.

NetGen призначений для генерації і відображення 3D- сіток і складається з декількох бібліотек. Основна бібліотека `nglib` реалізує безпосередньо генерацію сіток. Є два способи завдання простору, яке має бути заповнене тетраедральними елементами:

Завдання простору за допомогою операторів конструктивної блокової геометрії (Constructive Solid Geometr, CSG). У цій технології шуканий простір описується сукупністю примітивних об'єктів, над якими задаються такі операції як об'єднання, перетин, різниця.

Опис поверхні, що є кордоном шуканого простору, у файлі формату STL. У цьому форматі інформація про поверхні представляється у вигляді координат трикутних граней поверхні і їх нормалей.

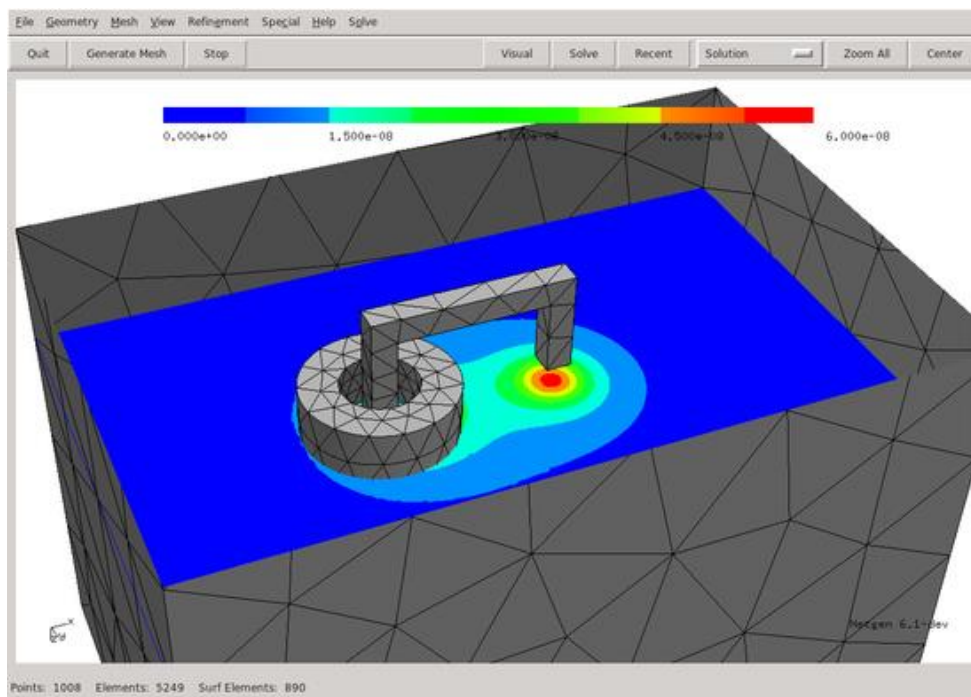


Рис. 1.21. Вікно програми NetGen

У пакеті використовується алгоритм на основі комбінованого методу вичерпання і методу на основі критерію Делоне. Імпорт та експорт даних може бути здійснений через найрізноманітніші формати, в тому числі і через стандартні. Зокрема, NetGen "знає" багато форматів CAD-систем.

Ймовірно, єдиний недолік пакета - неможливість прямого контролю над розмірами елементів.

Автор програми – Joachim Schöberl. Сторінка в Інтернеті: <http://www.hpfem.jku.at/netgen/>.

T3D – програма для побудови якісних сіток методом вичерпування. Складні області дискретизуються методом узгодження по кордону. Сама область задається як сукупність поверхонь кордону і обмежень. Дані імпортуються та експортуються через текстові файли спеціального формату. Крім того, пакет вміє працювати безпосередньо з системами CAD.

Існують версії пакета для ОС Windows і Linux. Програма володіє консольним інтерфейсом. Для ОС сімейства Unix передбачена можливість візуалізації сіток за допомогою сторонньої програми (Elixir). Також можливий запис сіток в графічних форматах VRML 2.0 і VTK.

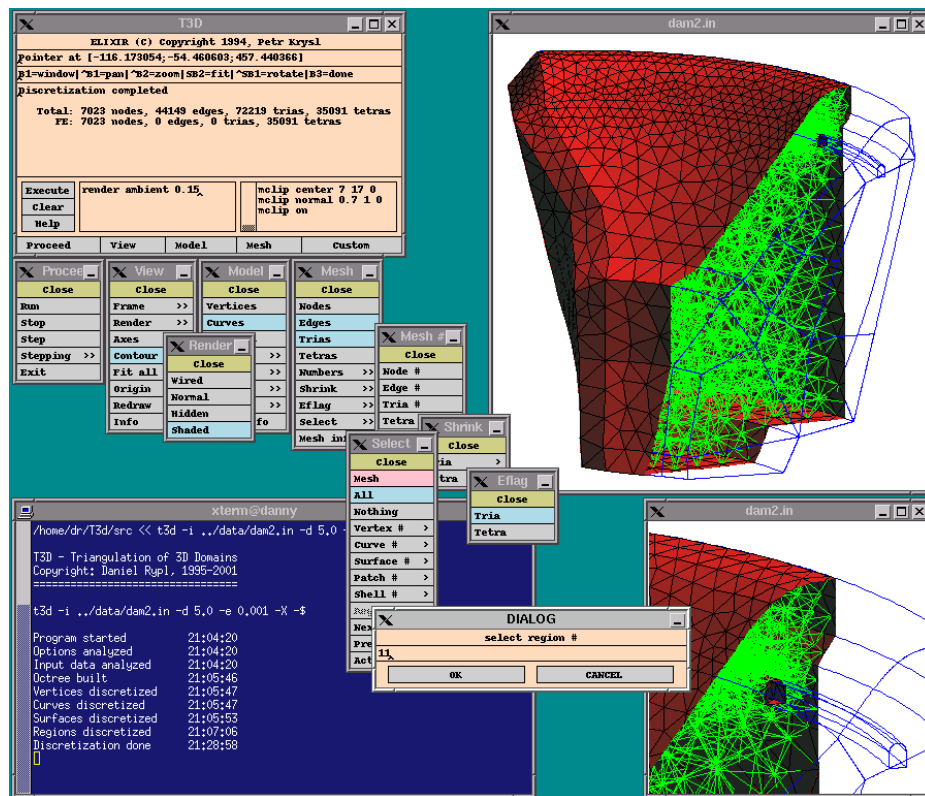


Рис. 1.22. Вікно програми T3D

Пакет безкоштовний для академічного використання, але автором підтримується дуже обмежено. Автор пакета – Daniel Ryp1.

### 1.5. Висновки до розділу

В результаті огляду літературних джерел проаналізовано способи подання геометричних тіл та методи побудови сіток. Найпоширеніші алгоритми дискретизації можна розбити на дві частини: побудова первинної дискретизації (найбільш складний етап), та її оптимізація. У додатку автоматичної генерації розрахункових сіток для скінчено- елементного моделювання конструкцій при первинній дискретизації області на скінченні елементи прийнято застосувати модифікований алгоритм Ватсона-Лавсона. Оптимізацію отриманої скінчено-елементної сітки робити шляхом застосування алгоритму Рапперта в плоскому випадку та Шевчука – в тривимірному. Розглянуто та проаналізовано критерії якості побудованої сітки. Зроблено огляд сучасних програм для дискретизації просторових конструкцій.



## **РОЗДІЛ 2.**

### **МОДЕЛЮВАННЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРЕПРОЦЕСОРА**

Підходом до проєктування різних систем, шляхом подання у вигляді діаграм їхніх статичних і динамічних моделей на всіх процесах життєвого циклу, являється мова візуального моделювання UML (Unified Modeling Language).

В основу методу покладено парадигму об'єктного підходу, при якій концептуальне моделювання проблеми полягає у побудові [17]:

- онтології домену, яка визначає склад та ієрархію класів об'єктів домену, їх атрибутів і взаємозв'язків, а також операцій, які можуть виконувати об'єкти класів;
- моделі поведінки, яка задає можливі стани об'єктів, інцидентів, що ініціюють переходи з одного стану до іншого, а також повідомлення, якими обмінюються об'єкти;
- моделі процесів, що визначає дії, які виконуються при проєктуванні об'єктів як компонентів.

Проєктування в UML починається з побудови сукупності діаграм, які візуалізують основні елементи структури системи.

Мова моделювання UML підтримує статичні і динамічні моделі, зокрема модель послідовностей – одну з найкорисніших і наочних моделей, в кожному вузлі якої є взаємодіючі об'єкти. Всі моделі зображаються діаграмами, коротка характеристика яких дається нижче.

#### **2.1. Логічне представлення статичної моделі структури програмної розробки**

Повний проєкт програмної системи являє собою сукупність моделей логічного і фізичного уявлень, які повинні бути узгоджені між собою. У мові UML для статичного представлення моделей систем використовуються діаграми класів. Вони визначають склад класів об'єктів і їх зв'язків.

На діаграмі класів (рис. 2.1.) прямокутники, поділені на три частини – це класи, лінії – зв'язки між ними. Ім'я класу записано в верхній частині прямокутника. В другій і третій частині – відповідно список атрибутів і операції, що мають специфікатори доступу.

Специфікація класів препроцесора.

**cPoint** – клас вершин елементів в  $n$ -мірному просторі, який містить координати точки, її індивідуальний номер в списку вершин усіх точок.

**cElement** – клас елементів в просторі  $R_n$ . Цей клас зберігає в собі список номерів вершин, їх кількість, індивідуальний номер елементу, координати його центру мас, об'єм, номери сусідніх елементів, що мають з ним загальну грань. Для моделювання завдань, що мають об'єкти з різними фізичними властивостями використовується змінна, що містить номер підобласті, якій належить елемент. Крім того клас елементу містить методи для динамічного обчислення граней.

**cFacet** – клас граней елементу, який містить список номерів вершин, що утворюють грань; їх кількість, площу, нормаль, орієнтовану з елементу хазяїна грані, номер елементу сусіда хазяїна грані, і номер елементу хазяїна грані. Грань обчислюється динамічно у міру використання для економії пам'яті.

**cMesh** – клас для зберігання і доступу до сітки в тілі програми. Цей клас містить список точок і елементів сітки. Іноді такі класи містять список граней. Проте, як показує практика, зберігання списків граней призводить до значної перевитрати пам'яті. Цей клас містить у тому числі методи для: читання і запису сіток в різних сіткових форматах, методи для реалізації згущення і розрідження сіток, які використовуються для адаптивних сіткових методів, методи по перебудові сіток.

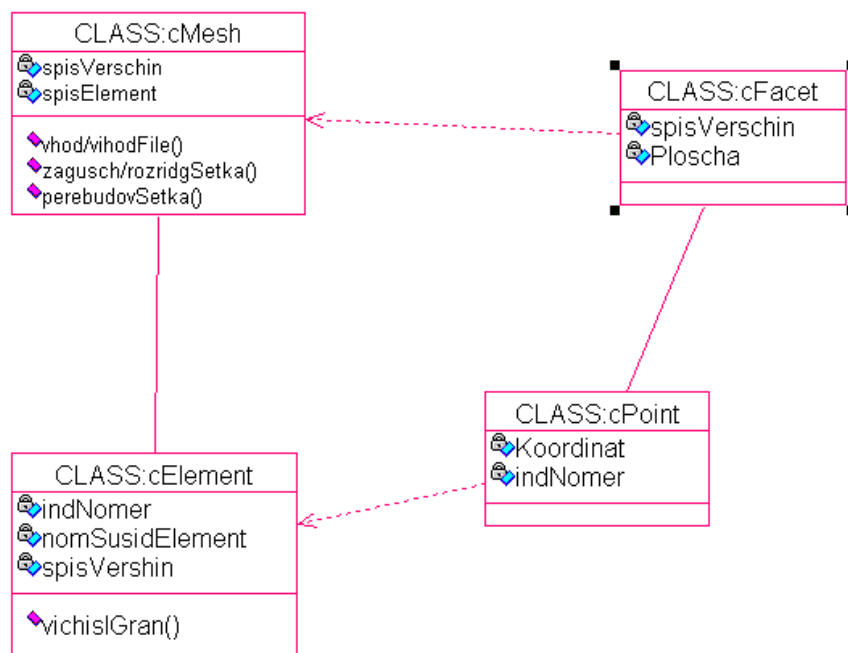
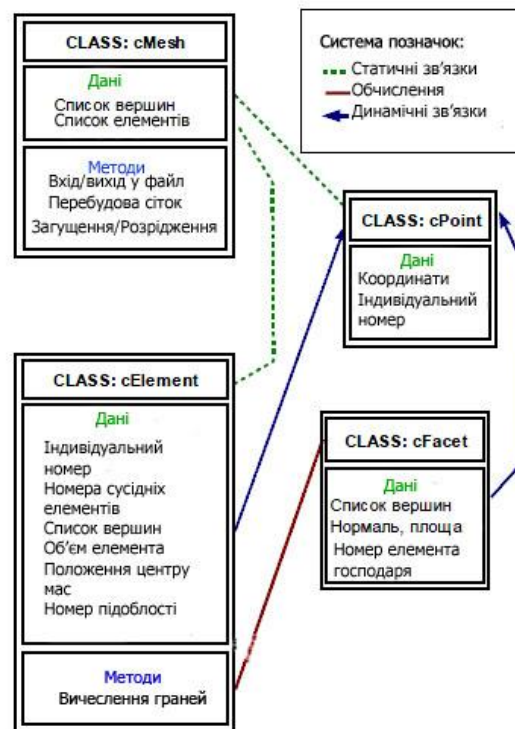


Рис. 2.1. Діаграма класів препроцесора

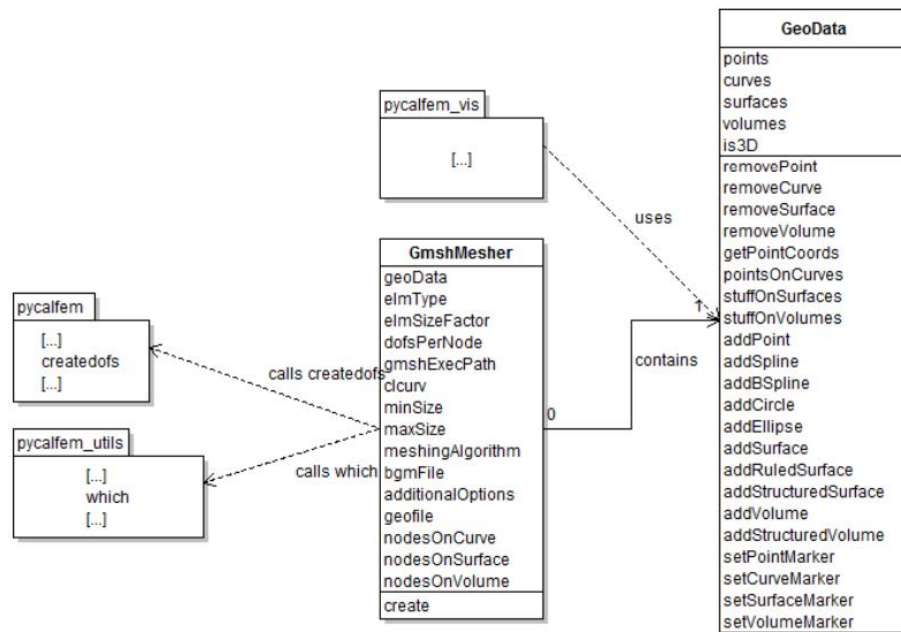


Рис. 2.2. Діаграма класів нових класів

Квадрати з вкладками вгорі є модулями. Показано «публічні» методи, атрибути та функції, які викликаються за класами.

Обчислювальна сітка міститься в об'єкті `chi_mesh::MeshContinuum`, який можна отримати за допомогою:

```
auto grid_ptr = cur_handler->GetGrid();
```

Детальніше про структури даних Mesh

### **chi\_mesh::MeshHandler**

Меші та операції з сітками обробляються `chi_mesh::MeshHandler`. Обробники сітки завантажуються в глобальну змінну `chi_meshhandler_stack`, а поточний обробник відстежується інший глобальної змінної, `chi_current_mesh_handler`. Якщо виконується будь-яка операція з сіткою, вона повинна поміщати новостворені об'єкти у стек в обробнику. На рисунку 2.3 показано узагальнену архітектуру.

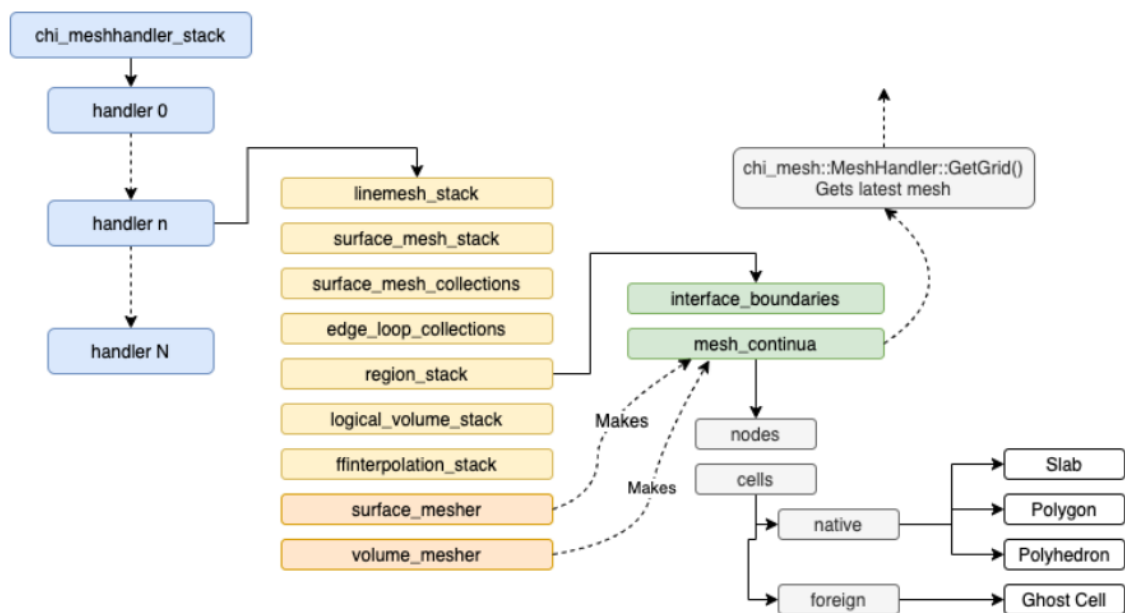


Рис. 2.3. Огляд ієрархії сітки

Кожен обробник сітки обладнано однією сіткою для поверхні та однією сіткою для об'єму, обидва спочатку невизначені. Етап сітки поверхні можна розглядати як етап попередньої обробки.

Види поверхневих сіток:

**chi\_mesh::SurfaceMesherPredefined.** Пройти через препроцесор. Не виконує жодного зчеплення.

**chi\_mesh::SurfaceMesherDelaunay.** Змінює поверхневу сітку за допомогою тріангуляції Делоне.

**chi\_mesh::SurfaceMesherTriangle.** Змінює сітку поверхні за допомогою Triangle 1.6.

Так само існують різні типи об'ємних сіток:

**chi\_mesh::VolumeMesherLinemesh1D.** Перетворює лінійні сітки на сляби.

**chi\_mesh::VolumeMesherPredefined2D.** Перетворює попередньо визначені сітки поверхонь у двовимірні трикутники, чотирикутники або багатокутники.

**chi\_mesh::VolumeMesherExtruder.** Екстрадує попередньо визначені поверхневі сітки в тривимірні трикутні призми, шестигранники або багатогранники.

**chi\_mesh::VolumeMesherPredefined3D.** Перетворює завантажені 3D-сітки на 3D-тетраедри, шестигранники або багатогранники.

**chi\_mesh::VolumeMesherPredefinedUnpartitioned.** Перетворює легку нерозділену сітку на правильну розділену сітку з повними деталями. Цей мешер забезпечує велику гнучкість для читання сіток із зовнішніх джерел.

Поверхневі сітки та об'ємні сітки призначаються обробнику як:

```
cur_handler->surface_mesher = new chi_mesh::SurfaceMesherPredefined;  
cur_handler->volume_mesher = new chi_mesh::VolumeMesherPredefined3D;
```

Щоб виконати поверхневі сітки, просто виконайте:

```
cur_handler->surface_mesher->Execute();  
cur_handler->volume_mesher->Execute();
```

Комірки в Chi-Tech є основними будівельними блоками для наукових обчислень на основі сітки. Деякі типи сіток показані на рисунку 2.4 та визначаються переліком, який вони містять, як визначено `chi_mesh::CellType`. Наразі підтримуються такі типи комірок:

- `chi_mesh::CellType::SLAB`
- `chi_mesh::CellType::TRIANGLE`
- `chi_mesh::CellType::QUADRILATERAL`
- `chi_mesh::CellType::POLYGON`
- `chi_mesh::CellType::TETRAHEDRON`
- `chi_mesh::CellType::HEXAHEDRON`
- `chi_mesh::CellType::POLYHEDRON`

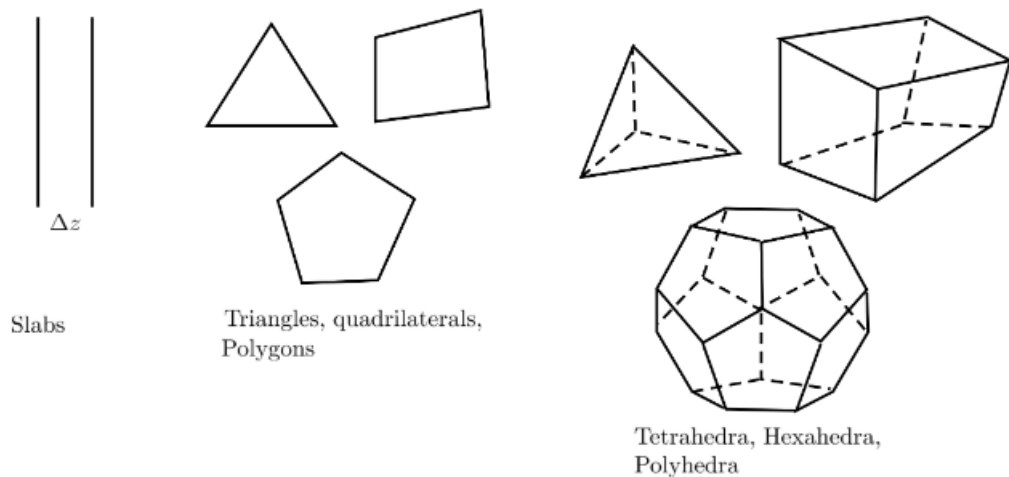


Рис. 2.4. Типи клітин

Клітини живуть у члені `local_cells` сітки, яка має тип об'єкта `chi_mesh::MeshContinuum::LocalCells` під егідою `native_cells` або `external_cells`. Гарантовано, що локальні клітини є повністю визначеними, тоді як нелокальні клітини, швидше за все, будуть клітинами-привидами. Найшвидший спосіб отримати доступ до комірки за допомогою її локального ідентифікатора.

```
auto cell = grid->local_cells[cell_local_id];
```

Об'єкт `local_cells` також сприяє ітератору.

```
for (auto cell = grid->local_cells.begin();
```

```
cell != grid->local_cells.end();
```

```
++cell)
```

```
{//do stuff}
```

а також ітератор на основі діапазону

```
for (auto& cell : grid->local_cells)
```

```
{ //do stuff}
```

Крім того, більш дорогий спосіб отримати доступ до комірки за допомогою її глобального індексу через елемент клітинки сітки. Цей член є допоміжним об'єктом, який шукатиме в `native_cells` і `external_cells` клітинку з пов'язаним глобальним ідентифікатором

```
auto cell = grid->cells[cell_global_id];
```

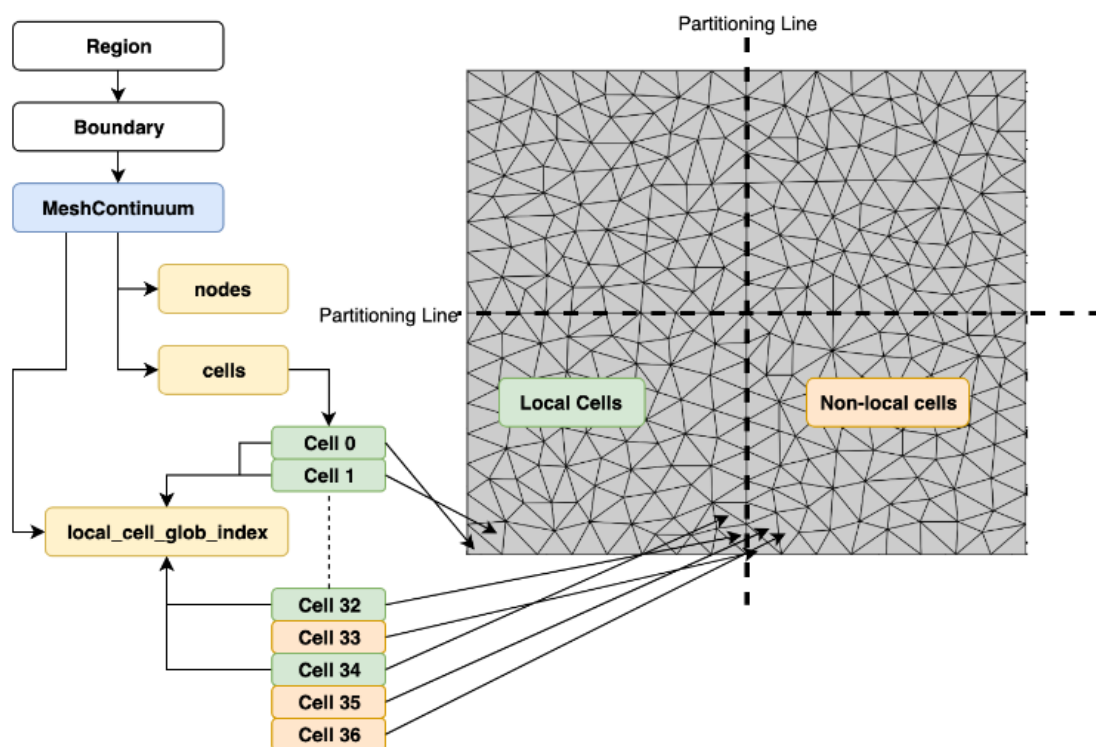
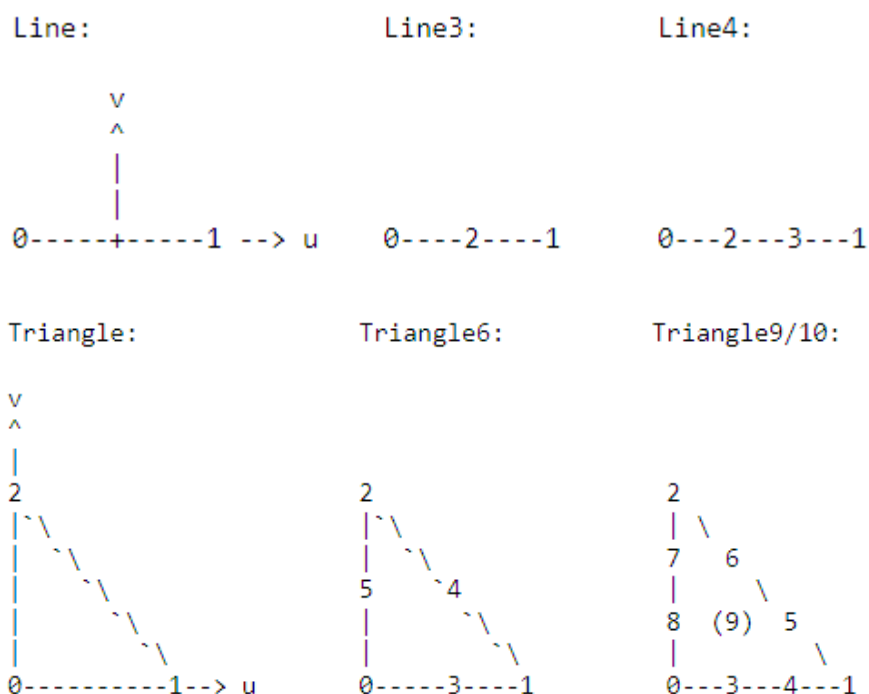


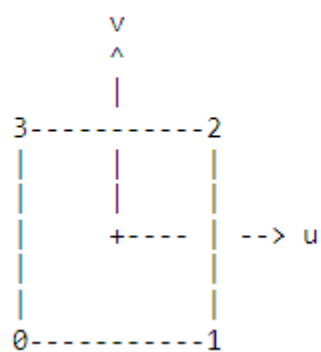
Рис. 2.5. Логіка відображення клітинок

Для всіх форматів сітки та файлів постобробки еталонні елементи визначаються наступним чином:

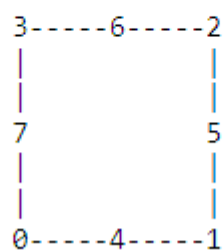




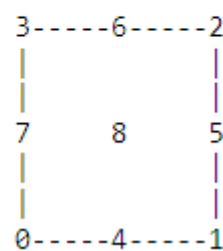
Quadrangle:



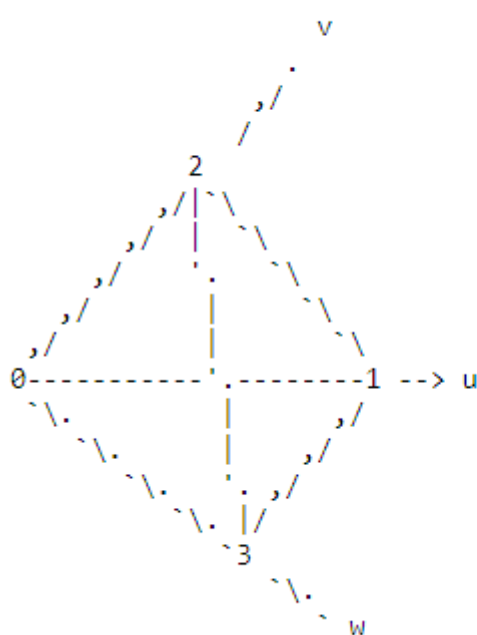
Quadrangle8:



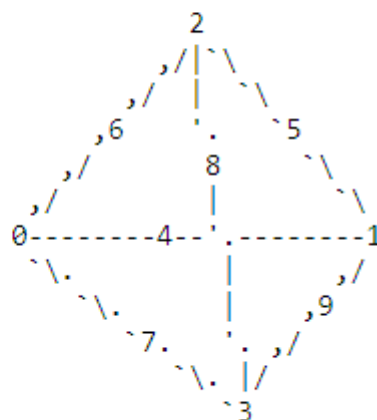
Quadrangle9:



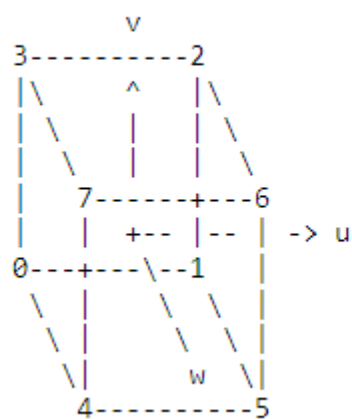
Tetrahedron:



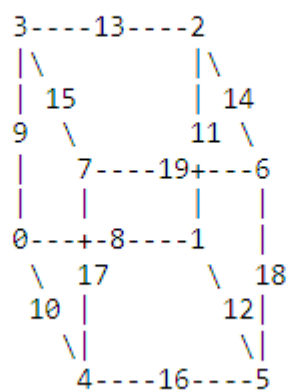
Tetrahedron10:



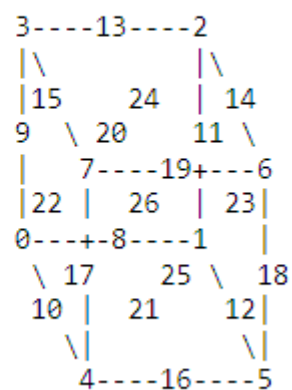
Hexahedron:



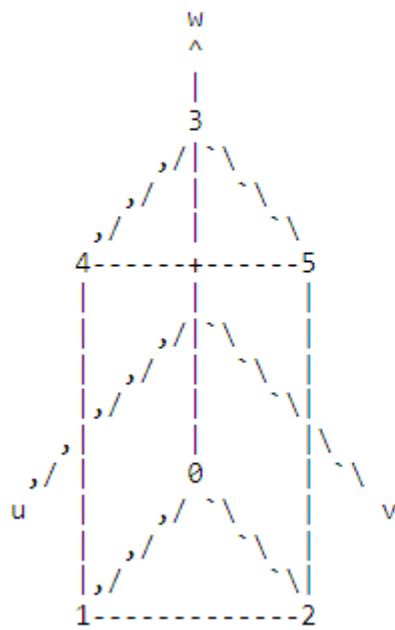
Hexahedron20:



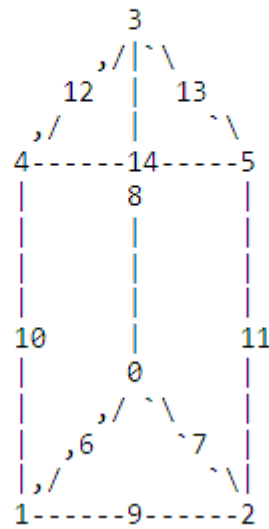
Hexahedron27:



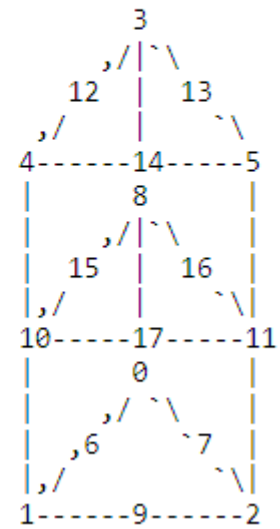
Prism:



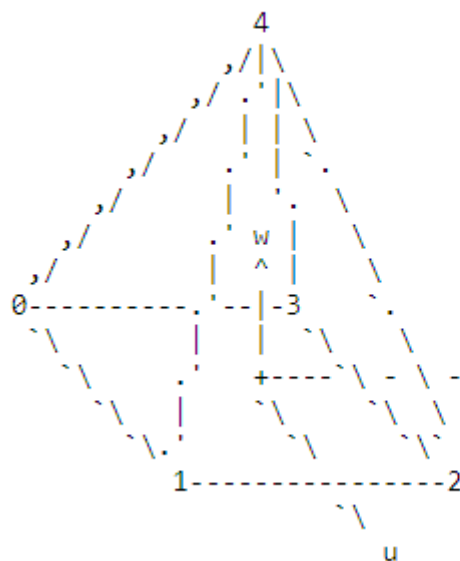
Prism15:



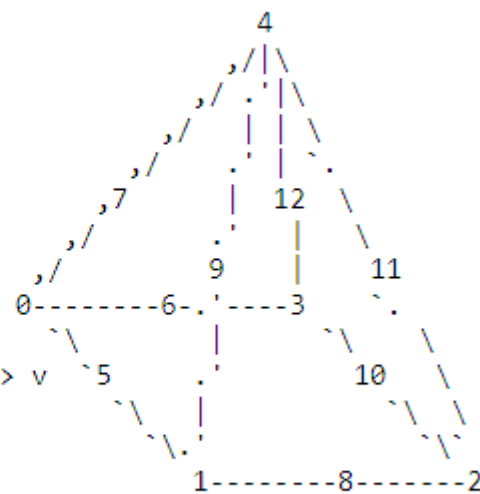
Prism18:



Pyramid:



Pyramid13:



## 2.2. Логічне представлення моделі поведінки програмної розробки

Поведінка програмної розробки визначається множиною об'єктів, що обмінюються повідомленнями.

На UML діаграмі варіантів використання (Use Case Diagram) (рис. 2.6.) показано взаємодію між варіантами використання і діючими особами. Вона відображає вимоги до системи з точки зору користувача. В нашому випадку варіанти використання – це функції, виконувані програмним комплексом, а

дійові особи - це користувачі створюваного програмного забезпечення. Такі діаграми показують, які діючі особи ініціюють варіанти використання. З них також видно, коли дійова особа отримує інформацію від варіанту використання.

Специфікація Use Case Diagram «Дискретизація фігури» (рис. 2.6.)

Короткий опис:

Use Case Diagram дозволяє користувачу дискретизувати фігуру на 1D, 2D, 3D - скінчені елементи.

Користувачі системи:

1. Оператор- задає команди для побудови моделі.
2. Програмний комплекс-виконує команди оператора.

Варіанти використання:

1. Запуск програми.
2. Встановити масштаб.
3. Вибір моделі.
4. Задати вид моделі.
5. Задати тип дискретизації.
6. Розбиття фігури на скінчені елементи.
7. 1D- розбиття.
8. 2D- розбиття.
9. 3D- розбиття.
10. Дискретизована фігура.
11. Оптимізація сітки скінчених елементів.
12. Збереження сітки скінчених елементів.

Користувач системи «Оператор»

Короткий опис :

Даний користувач системи описує дії оператора для побудови моделі.

Основні події:

1. Оператор запускає програму.
2. Програма завантажується.

3. Оператор вибирає фігуру.
4. Програмний комплекс завантажує обрану фігуру.
5. Оператор задає вид моделі.
6. Оператор задає масштаб
7. Оператор задає тип дискретизації.
8. Програмний комплекс автоматизує послідовність дій оператора.
9. Програмний комплекс виконує поставлені задачі оператора.

Користувач системи «Програмний комплекс»

Короткий опис:

Даний користувач системи описує дії програмного комплексу щодо побудови моделі.

Основні події:

1. Отримав запит оператора, завантажується для роботи.
2. Оператор задає тип дискретизації.
3. Програмний комплекс розбиває фігури на скінчені елементи.
4. Оператор прагне покращити якість сітки.
5. Програма оптимізує сітку скінчених елементів.
6. Оператор зберігає побудовану модель.
7. Програмний комплекс зберігає сітку скінчених елементів.

Варіант використання «Запуск програми»

Короткий опис:

Даний варіант використання описує запуск програми для дискретизації.

Основні події:

1. Оператор відкриває програму .
2. Програмний комплекс завантажується.

Варіант використання «Вибір фігури»

Короткий опис:

Даний варіант використання описує вибір оператором фігури.

1. Оператор вибирає потрібну фігуру.
2. Програма відображає вибрану фігуру.

Варіант використання «Встановити масштаб»

Короткий опис:

Даний варіант використання описує завдання оператором масштабу фігури.

1. Оператор обирає потрібний масштаб.
2. Програма відображає фігуру.

Варіант використання «Задати тип дискретизації»

Короткий опис:

Даний варіант використання описує вибір типу дискретизації для розбиття фігури.

1. Оператор задає тип дискретизації (1D,2D,3D)
2. Програмний комплекс дискретизує фігуру на скінчені елементи.

Варіант використання «1D- розбиття» , «2D-розбиття», «3D-розбиття»

Короткий опис:

Дані варіанти використання описують розбиття фігури для 1D,2D,3D- площини.

1. Оператор задає команду для розбиття фігур, а саме для 1D- площини.
2. Програмний комплекс виконує розбиття 1D,2D,3D- розбиття.

Варіант використання «Дискретизована фігура»

Короткий опис:

Варіант використання описує автоматичне виконання дискретизації на 1D, 2D чи на 3D скінчені елементи.

1. Програма отримує запит на розбиття фігури.
2. Програма дискретизує фігуру.

Варіант використання «Оптимізація сітки скінчених елементів»

Короткий опис:

Варіант використання описує покращення топологічної якості сітки.

1. Програма отримує запит від оператора на покращення якості сітки.

2. Програма оптимізує сітку.

Варіант використання «Збереження сітки скінчених елементів»

Короткий опис:

Варіант використання описує процес збереження побудованої моделі.

1. Програмний комплекс отримує запит на збереження моделі.
2. Програма зберігає модель по заданому шляху.

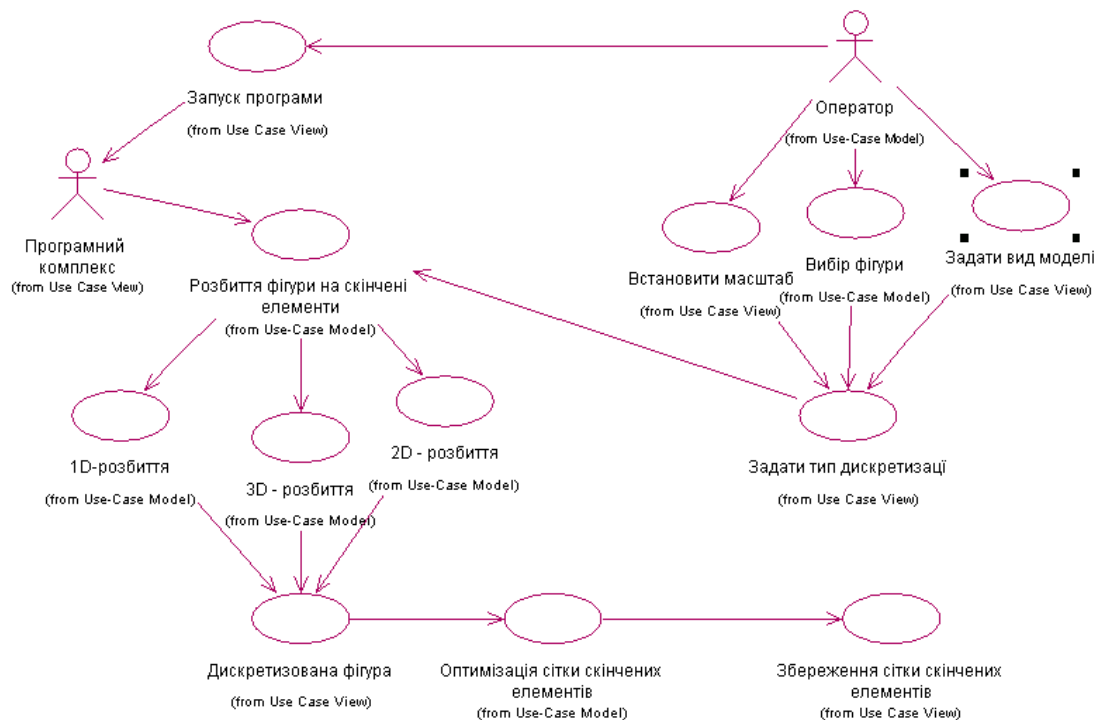


Рис. 2.6. Use Case Diagram «Дискретизація фігури»

Діаграма послідовності дій (рис. 2.7.) відображає взаємодію об'єктів, впорядковану за часом. На ній показані об'єкти і класи, використовувані в сценарії, і послідовність повідомлень, якими обмінюються об'єкти, для виконання сценарію. Діаграми послідовності дій зазвичай відповідають реалізаціям прецедентів в логічному представленні системи.

Діаграма відображає послідовність повідомлень між об'єктами.

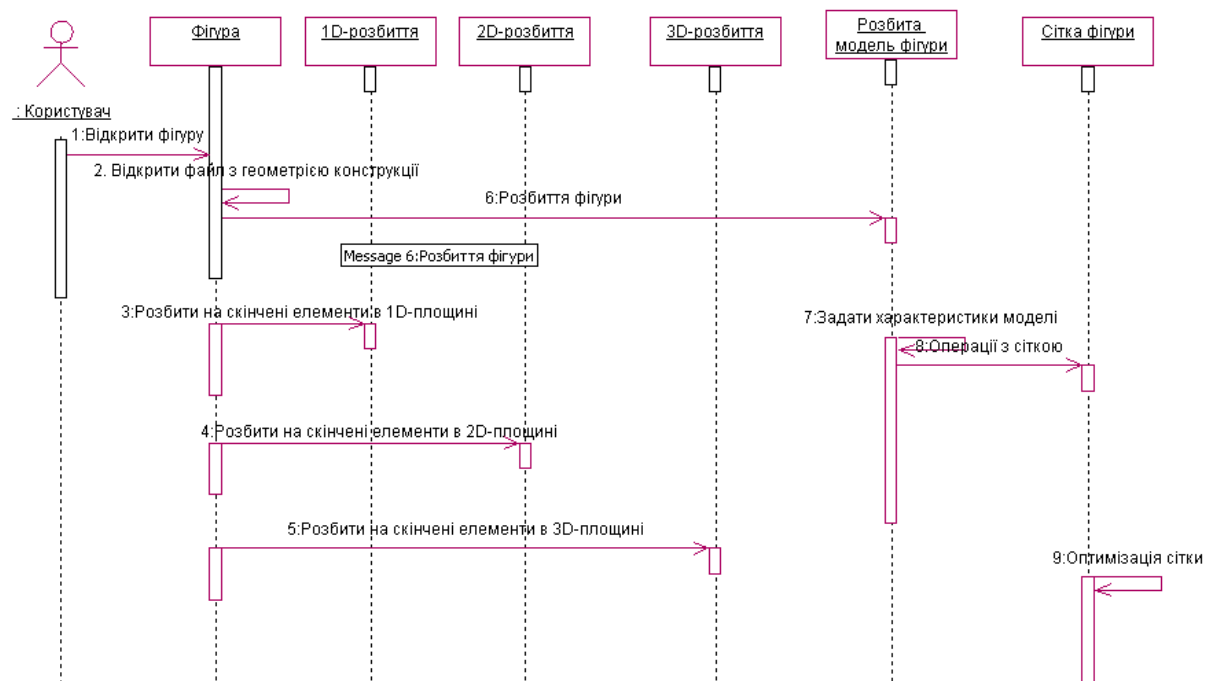


Рис. 2.7. Діаграма послідовності дій

Кооперативна діаграма відображає потік подій через сценарій варіанта використання. Діаграма (рис. 2.8.) загострює увагу на зв'язках між об'єктами.

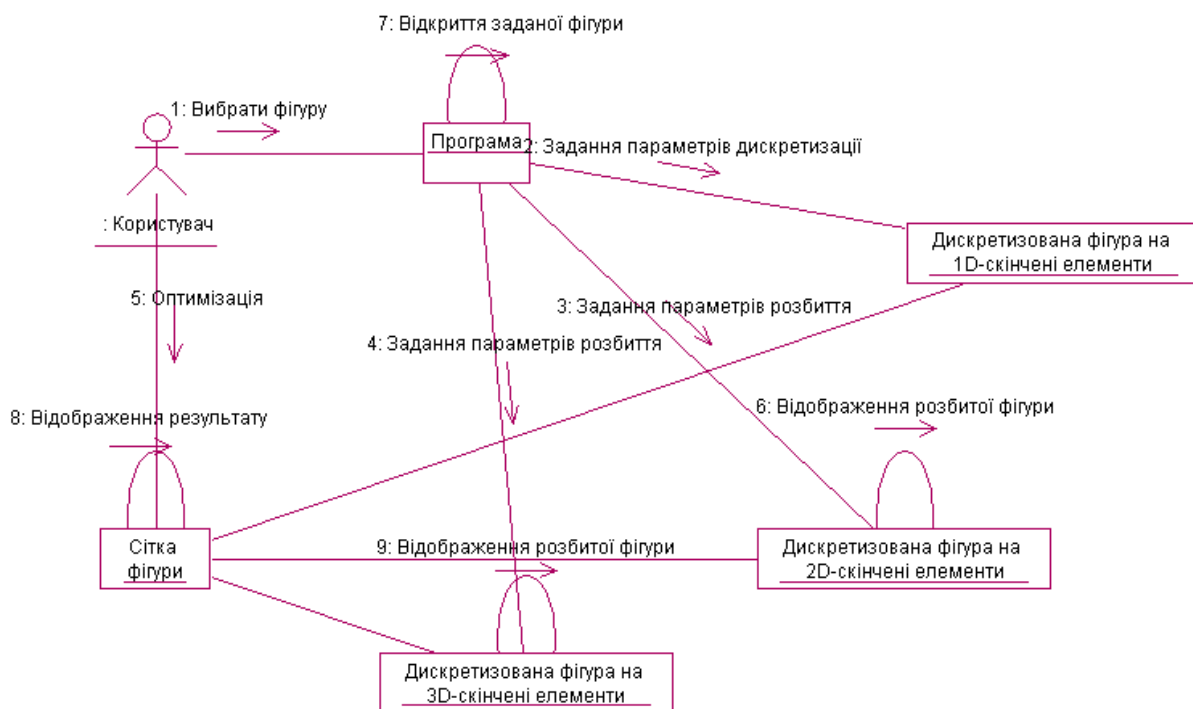


Рис. 2.8. Діаграма кооперації

### 2.3. Фізичне представлення моделі програмної розробки

Діаграми станів визначають всі можливі стани, в яких може перебувати конкретний об'єкт, а також процес зміни станів об'єкта в результаті настання деяких подій.

На діаграмі є два спеціальних стану – початкове (start) і кінцеве (stop). Початковий стан виділено чорною точкою, він відповідає стану об'єкта, коли він тільки що був створений. Кінцевий стан позначається чорною точкою в білому кружку, він відповідає стану об'єкта перед його знищенням. На діаграмі станів може бути один і тільки один початковий стан.

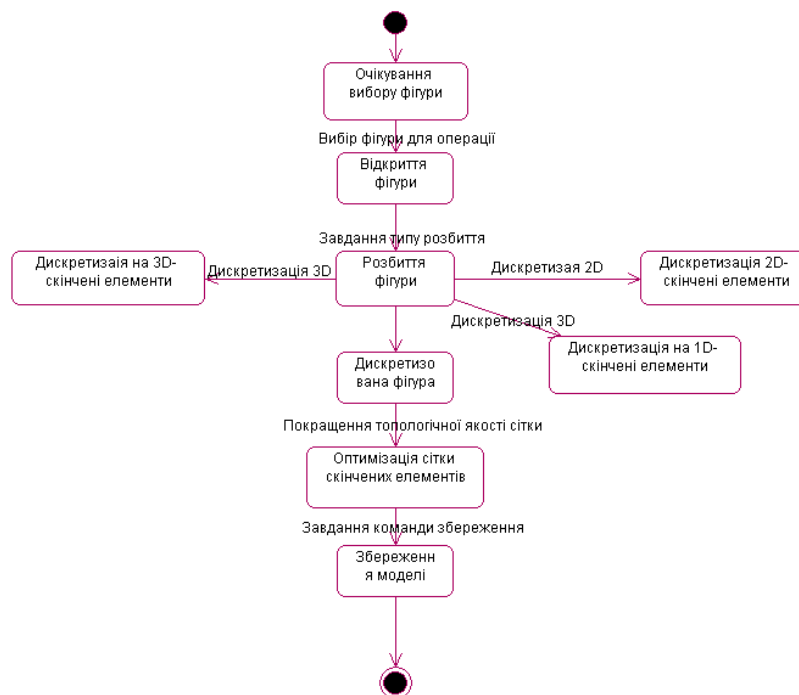


Рис. 2.9. Діаграма стану (Statechart diagrams)

Діаграма діяльності (Activity diagrams) деталізує особливості алгоритмічної реалізації виконання програмним комплексом операцій.



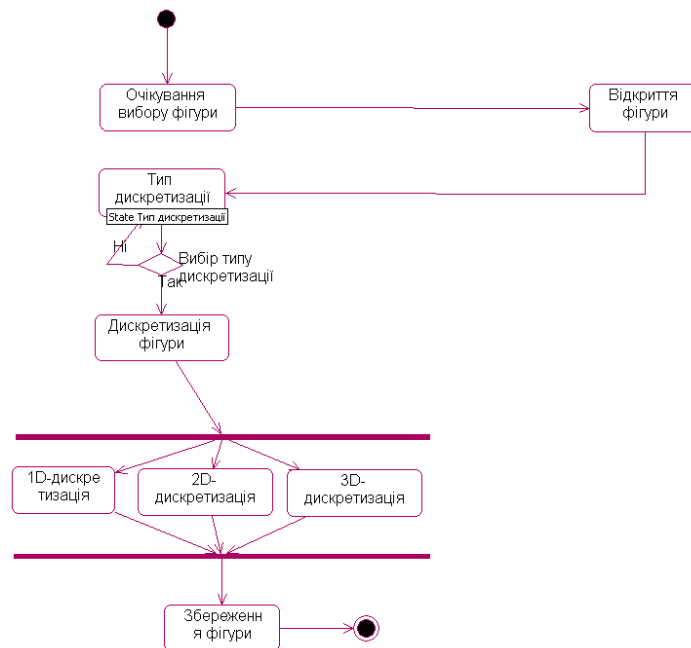


Рис. 2.10. Діаграма діяльності (Activity diagrams)

## 2.4. Архітектура програмного забезпечення препроцесора

Архітектурою програмного забезпечення є структура системи, яка представляє набір компонентів, що виконують певну функцію або набір функцій. Основне призначення архітектури – організація компонентів з метою забезпечення певної функціональності.

Розглядається архітектура програмного забезпечення препроцесора. Розроблена схема (рис. 2.11.) з основними функціональними компонентами, які входять до складу додатку автоматичної генерації скінчено-елементного моделювання.

Архітектура програмного комплексу складається з:

- підсистеми скінчено-елементної дискретизації;
- підсистеми моделювання програмного забезпечення.

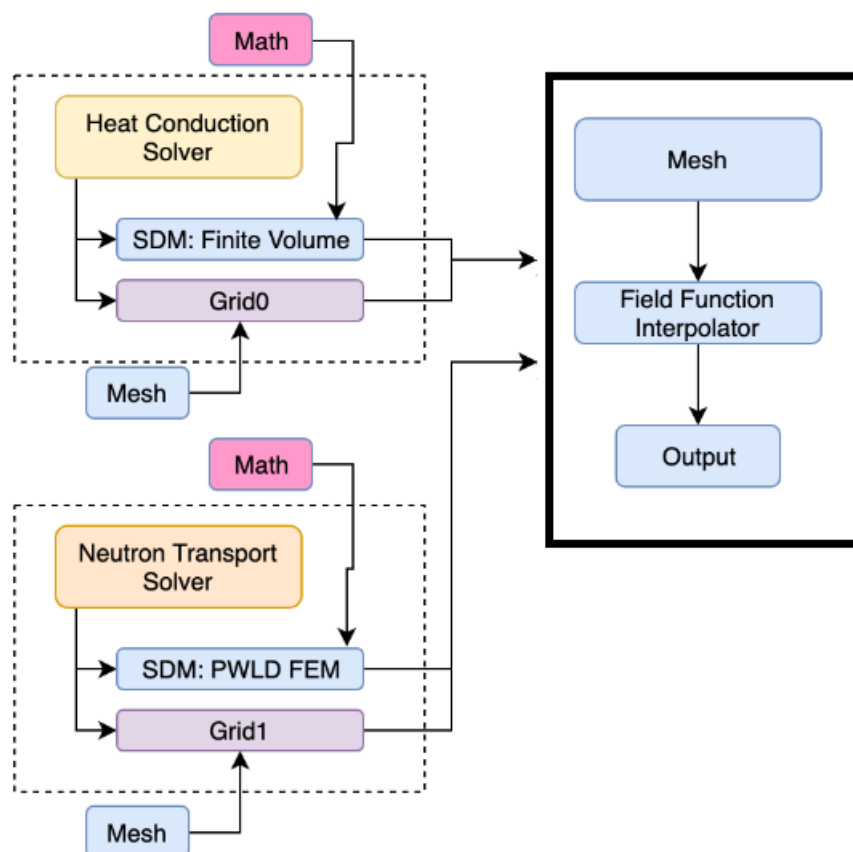


Рис. 2.11. Схема компонентів програмного комплексу

## 2.5. Висновки до розділу

В цьому розділі представлено статичну модель та модель поведінки програмної розробки. А також виконано фізичне представлення моделі.

Описано архітектуру програмного забезпечення додатка для автоматизації побудови дискретної (скінчено-елементної) моделі конструкцій.

## РОЗДІЛ 3

### РОЗРОБКА ПРЕПРОЦЕСОРА ДЛЯ СКІНЧЕННО-ЕЛЕМЕНТНОГО МОДЕЛЮВАННЯ

#### 3.1. Обґрунтування вибору середовища розробки препроцесора

Microsoft Visual C++ (MSVC) — інтегроване середовище розробки додатків на мові C++, що розроблене фірмою Microsoft і поставляється як частина комплексу Microsoft Visual Studio 2022.

Visual C++ 2022 надає потужне і гнучке середовище розробки, що дозволяє створювати додатки для Microsoft Windows і додатки, засновані на Microsoft .NET. Це середовище можна використовувати як інтегроване середовище розробки, так і в якості окремих засобів. Visual C++ складається з наступних компонентів:

- **Компілятор C++:** Visual C++ постачається з компілятором, який перетворює код на мові C++ у машинний код, який може бути виконаний операційною системою.
- **Інтегроване середовище розробки (IDE):** Visual C++ має потужне інтегроване середовище розробки (IDE) під назвою Visual Studio. Це середовище надає розширені можливості для розробки, налагодження, тестування та керування проєктами на мові C++. Воно має інтерфейс користувача з багатьма функціями, такими як редактор коду, відладчик, система контролю версій, інструменти тестування та багато іншого.
- **Бібліотеки Windows API:** Visual C++ надає доступ до багатьох бібліотек Windows API, які дозволяють розробникам взаємодіяти з операційною системою Windows. Ці бібліотеки надають функції для керування вікнами, обробки подій, мережевого взаємодії, роботи з файлами та багато іншого.
- **Бібліотеки стандарту C++:** Visual C++ включає бібліотеки стандарту C++, такі як STL (Standard Template Library), які надають реалізації різних алгоритмів, контейнерів та інших

корисних компонентів. Ці бібліотеки полегшують розробку програм на мові C++ і забезпечують переносимість коду між різними платформами.

- **Інструменти для налагодження та профілювання:** Visual C++ надає різноманітні інструменти для налагодження та профілювання програм. Це включає точковий налагоджувач (debugger) для виявлення та виправлення помилок, аналізатор пам'яті для виявлення витоків пам'яті.
- **Пакети розробки (SDK):** Visual C++ має різноманітні пакети розробки (SDK), які дозволяють розробникам створювати програми для конкретних платформ або функціональностей. Наприклад, Windows SDK надає набір інструментів та бібліотек для розробки програм під операційну систему Windows.
- **Інструменти для розробки графічного інтерфейсу:** Visual C++ має набір інструментів для розробки графічного інтерфейсу користувача. Включаючи дизайнер форм, який дозволяє створювати і налаштовувати вікна, кнопки, поля введення та інші елементи інтерфейсу.

Ці компоненти разом створюють потужне середовище розробки для програм на мові C++, дозволяючи розробникам створювати різні типи програм з використанням широкого набору інструментів та бібліотек.

Мова C++, що є найпопулярнішою у світі мовою рівня системи, і Visual C++ разом надають розробникові висококласний засіб світового рівня для побудови програмного забезпечення.

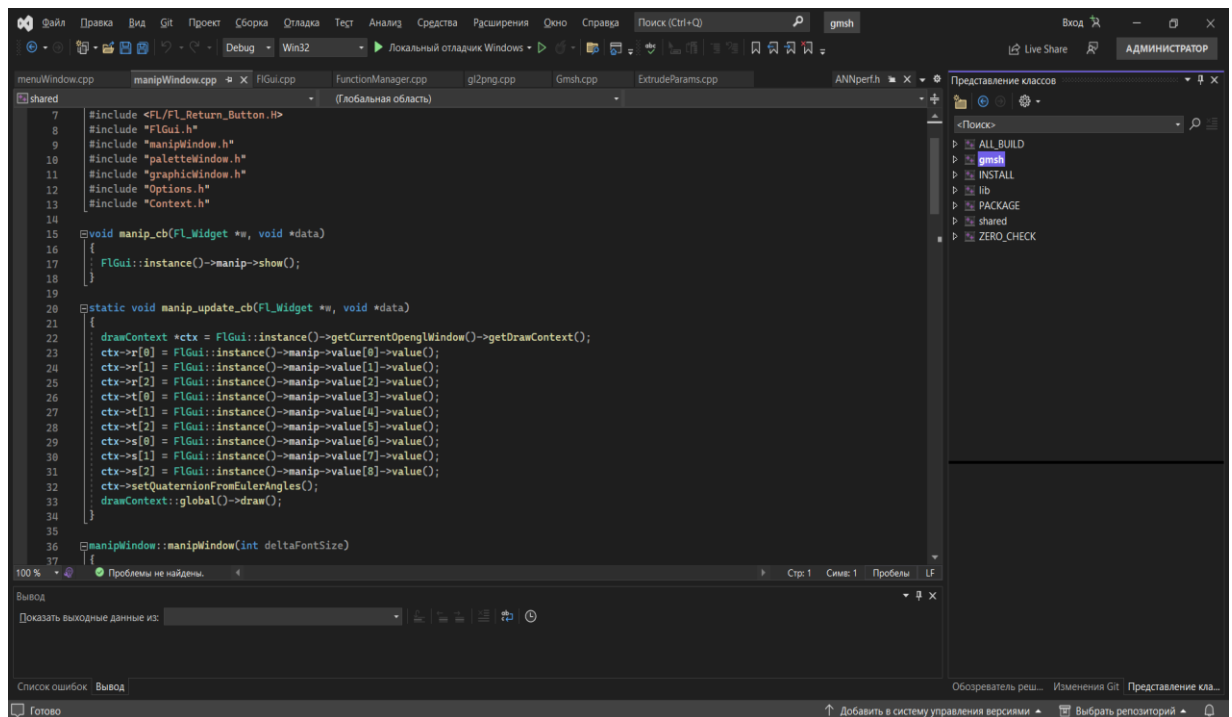


Рис. 3.1. Microsoft Visual Studio 2022

Програма реалізована на мові програмування C++ в середовищі Visual C++ 2022. Вибір середовища програмування обумовлений відносною простотою реалізації графічного інтерфейсу користувача (GraphicUserInterface – GUI).

При створенні засобів відображення і візуалізації використаний графічний інтерфейс OpenGL.

OpenGL має основну мету - відображення статичних та динамічних сцен з двовимірними та тривимірними об'єктами. У цьому контексті, об'єкти можуть бути представлені як набір вершин (для геометричних фігур) або пікселів (для растрових зображень). Починаючи з вихідних даних, OpenGL виконує перетворення, щоб перетворити примітиви та зображення в піксельне представлення. Кожен сформований піксель асоціюється з необхідними даними для його відображення та подальшої обробки, а результат перетворення розміщується в буфері кадру. Реалізація OpenGL для Windows включає понад 400 функцій, з яких 368 є базовими функціями основної бібліотеки opengl32.dll, а ще 52 функції входять до складу бібліотеки утиліт glu32.dll.

Базові функції OpenGL забезпечують можливість побудови зображень графічних примітивів, таких як точки, лінії, багатокутники та растрові зображення. Вони також включають функції для перетворення координат, обмеження області видимості, керування кольором, освітленням, текстурою та туманом.

Функції бібліотеки утиліт OpenGL є розширенням базового набору функцій і служать для створення зображень складніших об'єктів, таких як сфери, диски, конічні циліндри. Вони також дозволяють керувати текстурою та виконувати перетворення координат, проводити тріангуляцію багатокутників та побудову кривих та поверхонь на нерегулярній сітці контрольних точок з використанням форм Без'є та раціональних B-сплайнів.

Всі базові функції OpenGL можна розділити на п'ять категорій:

- Функції опису примітивів визначають нижній рівень ієрархії об'єктів, які можуть бути відображені графічною системою. У OpenGL такими примітивами є точки, лінії, багатокутники і т.д.
- Функції опису джерел світла використовуються для визначення положення та параметрів джерел світла, які знаходяться у тривимірній сцені.
- Функції завдання атрибутів дозволяють програмісту визначати зовнішній вигляд відображуваних об'єктів. Ці атрибути включають колір, характеристики матеріалу, текстури, параметри освітлення.
- Функції візуалізації дозволяють задавати положення спостерігача у віртуальному просторі та параметри об'єктива камери. Це дозволяє системі правильно побудувати зображення і відсікти об'єкти, які не потрапляють в поле зору.

Набір функцій геометричних перетворень дозволяє програмісту здійснювати різноманітні трансформації об'єктів, такі як повороти, зсуви, масштабування.

OpenGL складається з набору бібліотек, і основні функції знаходяться в основній бібліотеці, яку позначають аббревіатурою GL. Крім основної бібліотеки, OpenGL включає кілька додаткових бібліотек. Перша з них – бібліотека утиліт GL (GLU – GL Utility). Усі функції цієї бібліотеки визначаються за допомогою базових функцій GL. Бібліотека GLU містить реалізацію складніших функцій, таких як набір популярних геометричних примітивів (куб, куля, циліндр, диск), функції побудови сплайнів та додаткові операції над матрицями.

OpenGL сам по собі не включає спеціальних команд для роботи з вікнами або отримання інформації від користувача. Тому були створені спеціальні бібліотеки, які забезпечують такі функції, які часто використовуються при взаємодії з користувачем та для відображення інформації за допомогою віконної підсистеми. Однією з найпопулярніших таких бібліотек є GLUT (GL Utility Toolkit). Формально GLUT не є частиною OpenGL, але практично включається в багато дистрибутивів OpenGL і має реалізації для різних платформ. GLUT надає мінімальний набір функцій, необхідних для створення OpenGL-додатків. Існує також менш популярна бібліотека GLX, яка має аналогічні функції.

Крім того, функції, специфічні для конкретної віконної підсистеми, зазвичай входять до її прикладного програмного інтерфейсу (API). Наприклад, функції, специфічні для виконання OpenGL, можуть бути включені в Win32 API для системи Windows або в X Window для системи X Window.

На схемі 3.2, яка представлена нижче, зображена організація системи бібліотек у версії, що працює під управлінням системи Windows.

(Тут не вдається відобразити схему, але можна навести опис зображеного на схемі.)

На схемі видно, що основна бібліотека GL містить базові функції OpenGL. Бібліотека GLU є додатковою і включає більш складні функції. Бібліотека GLUT, хоча не є частиною OpenGL, забезпечує мінімальний набір

функцій для створення OpenGL-застосувань. Інші функції, пов'язані з віконною підсистемою, можуть бути включені в Win32 API.

Отже, OpenGL складається з основної бібліотеки GL, додаткової бібліотеки GLU і, за необхідності, використовується разом з бібліотекою GLUT або іншими віконними підсистемами для роботи з вікнами та отримання інформації від користувача.

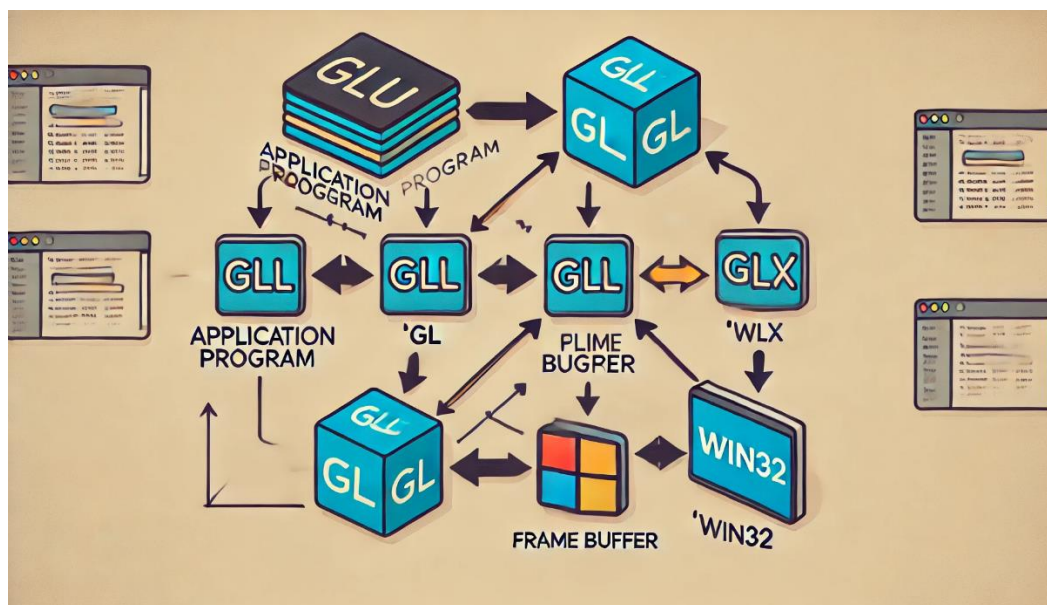


Рис. 3.2. Організація бібліотеки OpenGL

OpenGL використовує модель клієнт-сервер для реалізації своїх функцій. У цій моделі прикладна програма виступає в ролі клієнта, що генерує команди, а сервер OpenGL інтерпретує та виконує ці команди. Сервер може бути розташований на тому ж комп'ютері, що й клієнт (наприклад, у вигляді динамічної бібліотеки - DLL), або на іншому комп'ютері, використовуючи спеціальний протокол передачі даних між машинами.

OpenGL обробляє та формує зображення графічних примітивів в буфері кадру з урахуванням вибраних режимів. Примітив може бути точкою, відрізком, багатокутником тощо. Кожен режим може бути змінений незалежно від інших. Визначення примітивів, вибір режимів та інші операції виконуються за допомогою команд, які викликаються у вигляді функцій прикладної бібліотеки.



Примітиви визначаються набором однієї чи декількох вершин (vertex). Кожна вершина визначає точку, кінець відрізка або кут багатокутника. Кожній вершині приписуються певні дані, такі як координати, колір, нормалі, текстурні координати і т.д., що називаються атрибутами. У більшості випадків кожна вершина обробляється незалежно від інших.

Архітектура OpenGL реалізує конвеєрну схему обробки графічних даних, яка складається з кількох послідовних етапів обробки. На рисунку 3.3 показана ця схема.

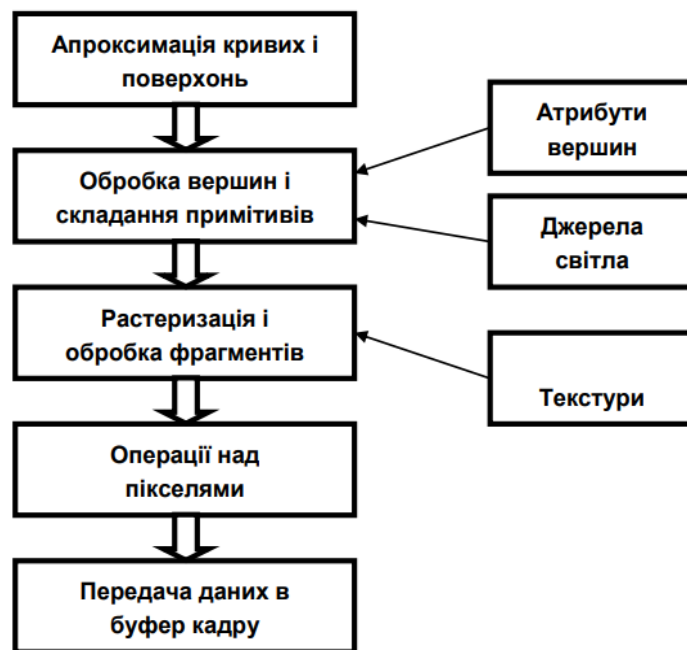


Рис. 3.3. Схема функціонування конвеєра OpenGL

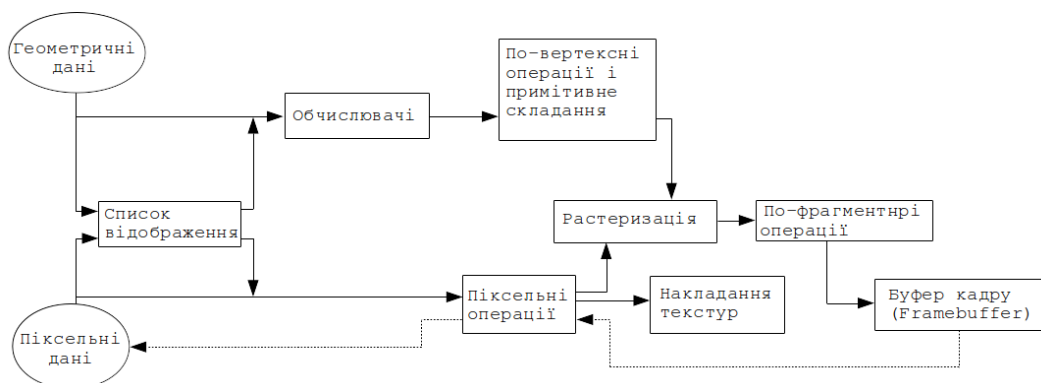


Рис. 3.4. Конвеєр відмальовування OpenGL

Команди OpenGL завжди обробляються в порядку їх надходження, але можуть відбуватися затримки, перед тим як проявиться ефект від їх виконання. Зазвичай OpenGL реалізує прямий інтерфейс, що означає, що визначення

об'єкта призводить до його візуалізації в буфері кадру. Для розробників OpenGL - це набір команд, які керують використанням графічного апарату.

Якщо графічний апарат складається тільки з адресованого буфера кадру, то функції OpenGL повністю реалізуються за допомогою ресурсів центрального процесора. Зазвичай графічний апарат надає різні рівні прискорення, починаючи від апаратної реалізації виводу ліній та багатокутників до високопродуктивних графічних процесорів з підтримкою різних операцій над геометричними даними.

OpenGL є прошарком між апаратним рівнем та рівнем користувача, що дозволяє забезпечити єдиний інтерфейс на різних платформах, використовуючи можливості апаратної підтримки.

### **3.2. Розробка інтерфейсу**

Дуже важливу роль в ефективності роботи програми грає правильно розроблений інтерфейс. Саме від цього буде залежати виконання декількох з основних вимог - зручність і простота в освоєнні. Складний, перевантажений інтерфейс може зменшити ефективність роботи з препроцесором. Головне правило, від якого варто відштовхуватися - інтерфейс не повинен заважати роботі користувача й не повинен відволікати його увагу від роботи, одночасно не втрачаючи при цьому функціональності.

Виходячи з цього, було вирішено використовувати типові для windows-програм візуальні компоненти. Це дасть можливість більшості користувачів швидко розібратися й включитися в роботу. Інтерфейс прямо залежить від функцій, що виконуються програмою.

Розроблений програмний комплекс виконує створення простих геометричних сутностей та дискретизацію на скінчені елементи досліджуваної області в конструкціях для елементів чисельного розрахунку напружено-деформованого стану деформівного тіла.

Має змогу:

- відкривати файли: Gmsh geometry .geo - файл геометрії GMSH; STEP і IGES - відомі формати файлів геометрія, підтримувані багатьма CAD –пакетами;
- будує точки, відрізки, сплайни, B-сплайн, дугу кола, еліптичну дугу, плоску фігуру, обтягуючу поверхню, об’ємне тіло;
- візуалізує нумерацію точок;
- візуалізує точки, відрізки, сплайни, B-сплайни, дуги кола, еліптичні дуги, плоскі фігури, обтягуючі поверхні, об’ємні тіла;
- автоматично виконувати дискретизацію на 1D, 2D чи на 3D скінчені елементи;
- візуалізує дискретизацію на 1D, 2D чи на 3D скінчені елементи;
- візуалізує нумерацію скінчених елементів;
- автоматизує рекомбінацію сітки скінчених елементів;
- виконувати оптимізацію сітки скінчених елементів.

Програмний комплекс зберігає в файл геометрію з дискретизацією 1D, 2D чи 3D скінчені елементи.

Програмний комплекс має досить зручний інтерфейс. В ньому передбачена можливість виконувати дискретизацією на 1D, 2D чи 3D скінчені елементи для розбиття криволінійних, просторових поверхонь і поверхонь, що обмежують об’ємні тіла та об’ємні тіла;

Програма має змогу виконувати оптимізацію скінчених елементів та забезпечує збереження в файл сітку скінчених елементів конструкцій різних форм.

Інтерфейс програми представлений в вигляді поєднання двох вікон (Рис.3.5.). Одне з них являється Графічним Вікном, де створюється модель і редагування моделі. А інше містить в собі основне Меню Програми та набір команд для роботи з фігурами.

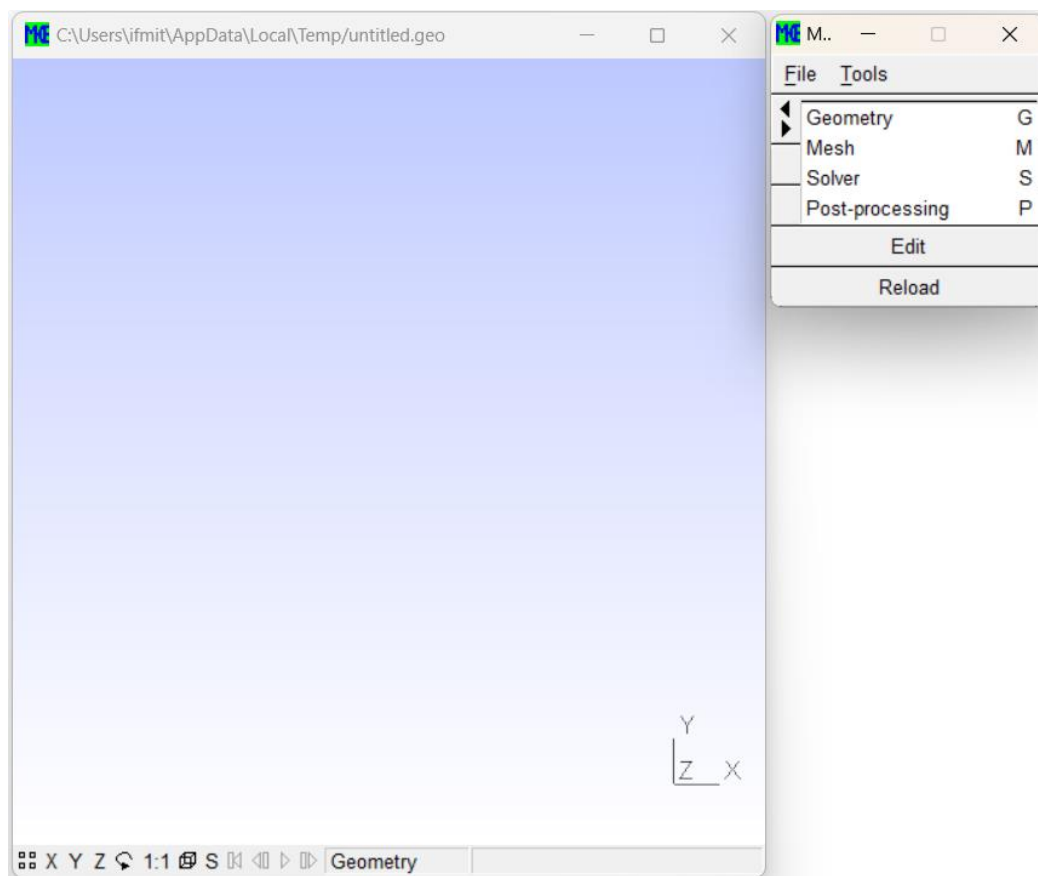


Рис. 3.5. Інтерфейс препроцесора

В нижній частині Графічного Вікна знаходиться Рядок Стану (рис.3.6).

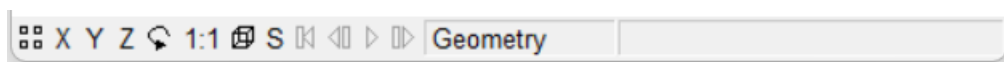









Рис. 3.5. Рядок Стану

Він містить команди для керування зображенням і інформацію про дію активної команди. (Табл.3.1)

Таблиця 3.1

### Команди Рядка Стану

Зображення піктограми	Комбінація клавіш	Дія
	Alt+x, Alt+Shift+x	Вид моделі відносно осі X.
	Alt+y, Alt+Shift+y	Вид моделі відносно осі Y.
	Alt+z, Alt+Shift+z	Вид моделі відносно осі Z.
	Shift Alt	Поворот моделі на 90° за часовою стрілкою та проти часової. Синхронізація повороту.

Зображення піктограми	Комбінація клавіш	Дія
	Alt	Встановлення масштабу. Синхронізація масштабу.
	Alt+o, Alt+Shift+o	Перемикання режиму проекції.
	Esc	Перемикач миші Вкл/Викл.

Вікно Меню Програми містить в собі основне меню програми та набір команд для роботи з фігурами. У верхній ділянці розташований рядок падаючих меню. В нього входять File, Tools. (рис.3.6).

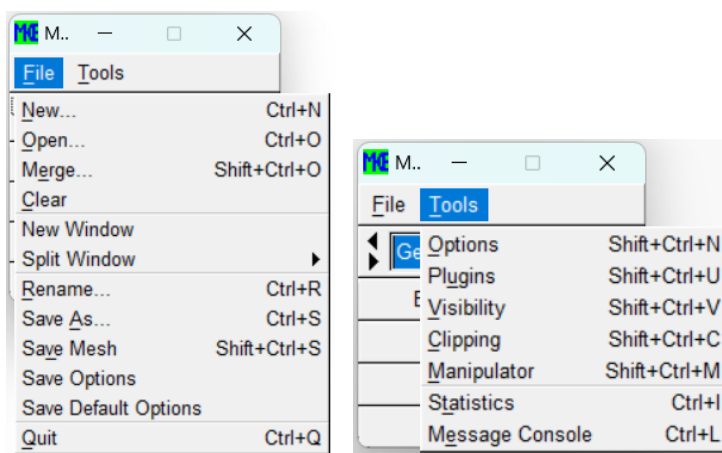


Рис. 3.6. Меню File, Tools

## Опис змісту діалогу меню File

### Відкриття фігури

1. Натискаємо на меню **File**.
2. З випадаючого меню обираємо **Open** або користуємося комбінацією клавіш **Ctrl+O**.
3. В списку знаходимо папку, в якій зберігаються фігури.
4. Подвійним натисканням відкриваємо папку.
5. Подвійним натисканням обираємо фігуру

### Об'єднання декількох елементів фігур

1. Натискаємо на меню File.
2. З випадаючого меню обираємо **Merge**, **Shift+Ctrl+O**.
3. Вибираємо спочатку одну фігуру, а потім таким чином обираємо ще одну .

### *Відкриття нового вікна*

1. Натискаємо на меню File.
2. З випадаючого меню обираємо New Window.

### *Розділення вікна*

1. Натискаємо на вікно File.
2. З випадаючого вікна обираємо Split Window => Horizontally для горизонтального розділення, Split Window =>Vertically для вертикального розділення.

### *Збереження проєкту*

1. Натискаємо на меню **File**.
2. З випадаючого меню обираємо **Save As** або **Ctrl+S**.
3. Вибираємо папку, в яку виконається зберігання .
4. В поле **Ім'я файлу** задаємо ім'я моделі.
5. Натиснути кнопку Сохранить.

### *Закриття проєкту*

1. Натискаємо на меню File.
2. З випадаючого меню обираємо Quit.

### **Опис змісту діалогу меню Tools**

#### *Опції налаштування відображення*

1. Натискаємо на меню **Tools**.
2. З випадаючого меню обираємо Options.
3. З'являється вікно Options General (рис. 3.7).

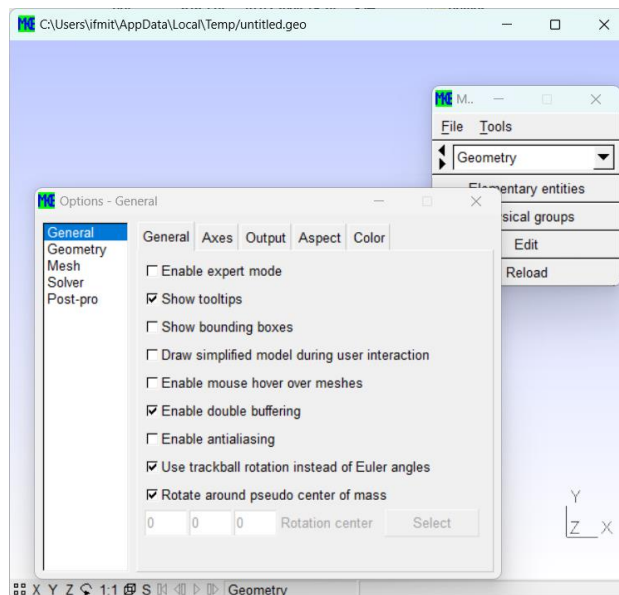


Рис. 3.7. Вікно Options General

4. При виборі Mesh на вкладці та Visibility можливо обирати відображення на геометричній фігурі точок, ліній, поверхонь та об'ємів та нумерацію точок, ліній, поверхонь та об'ємів (рис.3.8).

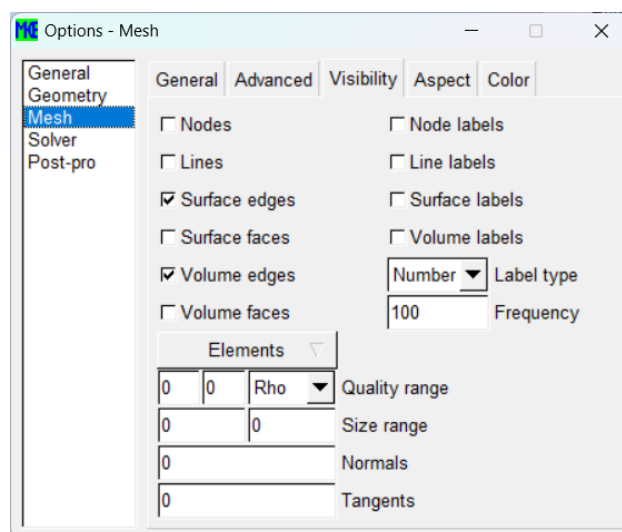


Рис. 3.8. Вікно Options Mesh

Visibility – дуже корисний інструмент, яким ви користуватиметеся дуже часто. Після побудови тетраедральної сітки та увімкнення Surface faces та Volume faces (якщо ці налаштування ще не включені, див. вище) ви побачите лише трикутну сітку на гранях паралелепіпеда.

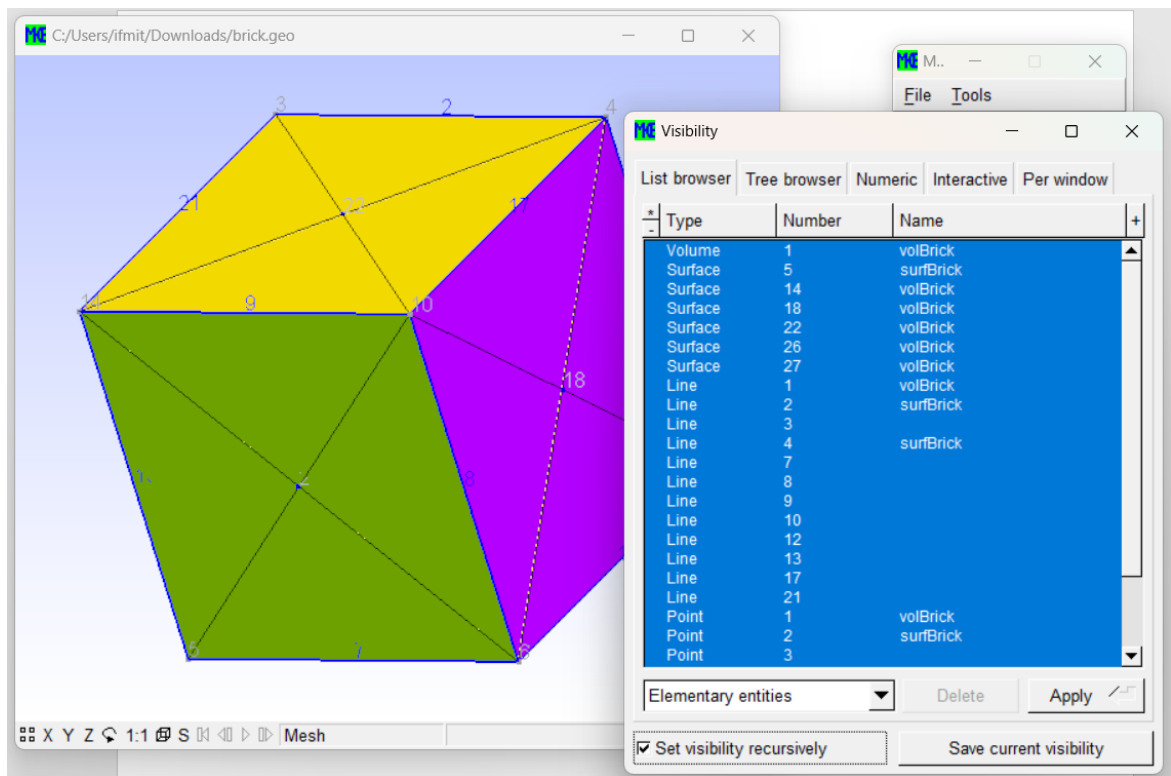


Рис. 3.9. Вікно Visibility

### Onyji Statistics

1. Натискаємо на меню **Tools**.
2. З випадаючого меню обираємо *Statistics*.
3. З'являється вікно *Statistics* (рис. 3.9.).

Показує інформацію про кількість елементів геометрії, сітки, а також деякі дані про якість сітки.

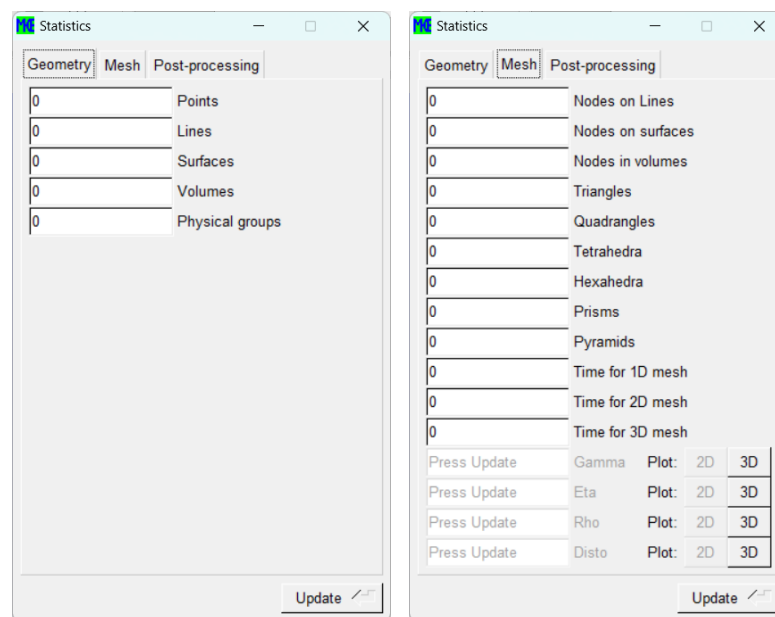


Рис. 3.10. Вікно Statistics



### Опції *Clipping*

1. Натискаємо на меню **Tools**.
2. З випадаючого меню обираємо *Clipping*.
3. З'являється вікно *Clipping*.

Перетин - напевно, ще більш важливий інструмент перегляду сітки, ніж Visibility. Адже як інакше заглянути всередину моделі, якщо не зробити розтин? Gmsh дозволяє робити розтин геометрії і сітки. Інтерфейс наступний:

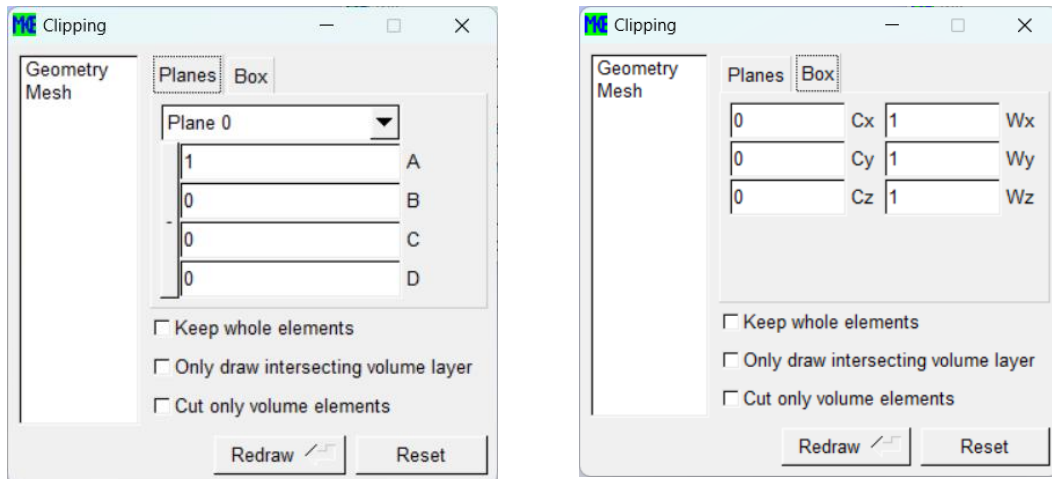


Рис. 3.11. Вікно Clipping

І для геометрії і для сітки можна зробити переріз або площинами (Planes) або паралелепіпедом (Box). Для цього потрібно вибрати відповідну вкладку. Planes. Можна задати 6 секучих площин коефіцієнтами A, B, C, D з рівняння площини  $A * x + B * y + C * z + D = 0$ . Можливо також інтерактивний рух площин за допомогою миші. Для цього посуньте мишею з лівою затиснутою кнопкою в поле введення числа для коефіцієнта. Ви побачите, як змінюється площина перерізу і, відповідно, сам переріз сітки.

Box. Cx, Cy, Cz - координати центру січного паралелепіпеда. Wx, Wy, Wz – довжини відповідних сторін симетрично щодо центру.

Приблизно такий перетин сітки площиною ви отримаєте, визначивши коефіцієнт D:

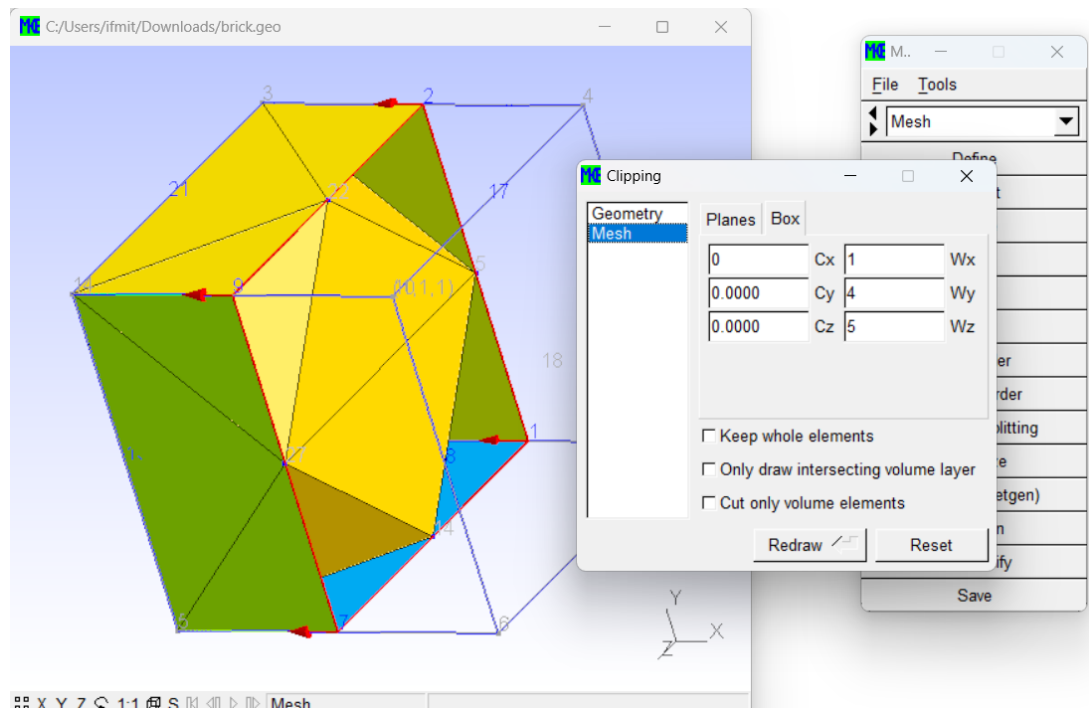


Рис. 3.12. Зміна в налаштуваннях Clipping

### 3.3. Побудова геометричних об'єктів в препроцесорі

Розглянемо роботу модулю геометрія на прикладі побудови прямокутника розмірами 5 на 3 одиниці та побудову об'ємної фігури, яку отримуємо витягуванням нашого прямокутника.

У Вікні Меню вибираємо: Geometry => Elementary Entities => Add => New => Point. Виникне вікно "Contextual Geometry Definitions" з активною закладкою "Point" (рис. 3.13.).

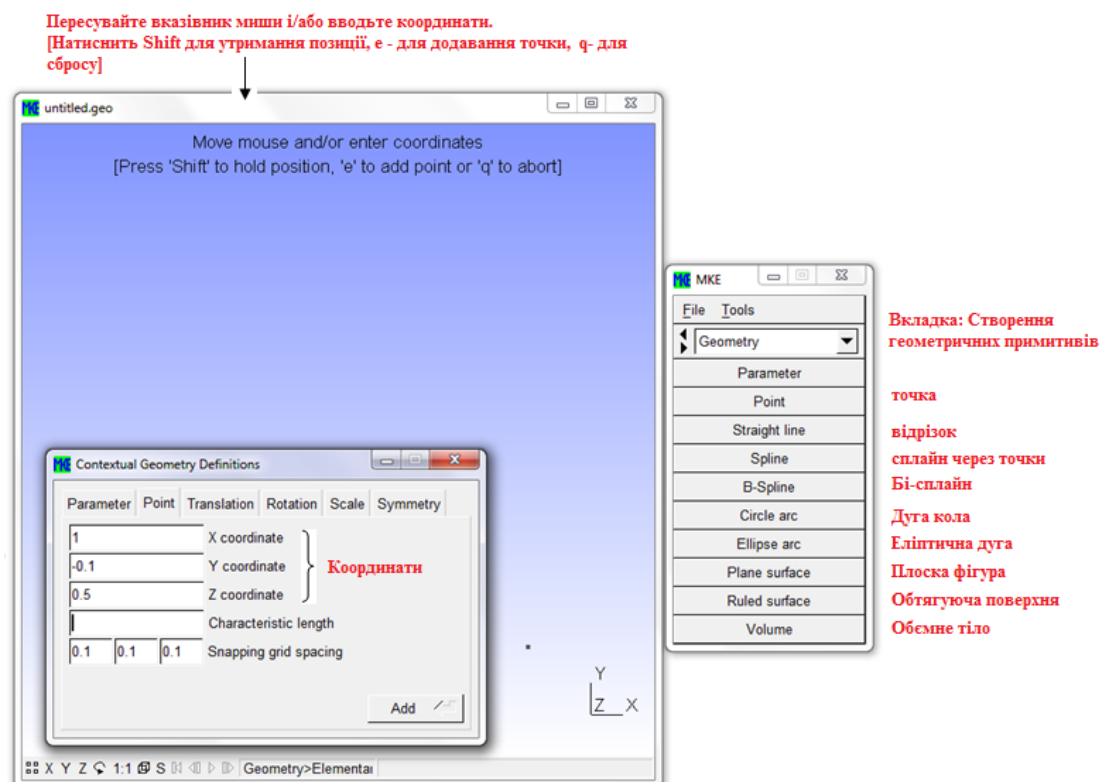


Рис. 3.13. Введення координат точок

Вводимо в "текстбокси"  $X=0$ ,  $Y=0$ ,  $Z=0$  – значення координат першої точки і тиснемо Add. Також, є можливість вводити точки покажчиком миші безпосередньо в Графічному Вікні (для цього переміщаємо покажчик і тиснемо "e" на клавіатурі).

В процесі введення точок вони відображаються в Графічному Вікні. Звернемо увагу на "текстбокс" з назвою "Characteristic length" ("Характеристична довжина"), в якій вже вказано за умовчанням значення 0.1. Ця величина пов'язана з розміром SE сітки що примикає до точки, яка вводиться. Закриваємо вікно "Contextual Geometry Definitions".

Відобразимо номери введенних точок. Tools => Options => Geometry на закладці Visibility ставимо галочку "Point Numbers". Результат показаний на рис. 3.14.

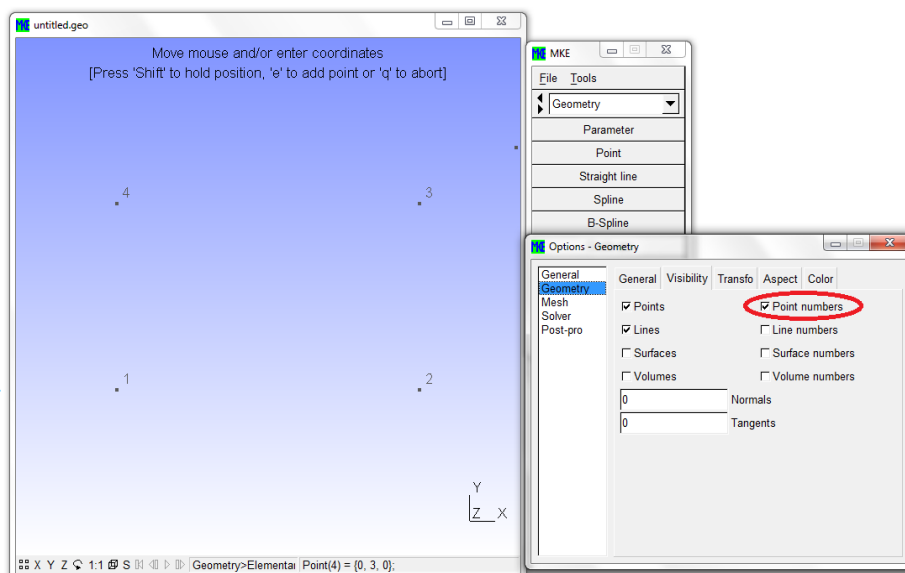


Рис. 3.14. Відображення номерів точок

Вводимо лінії (відрізки ліній). Для цього у Вікні Меню обираємо: Geometry => Elementary Entities => Add => New => Straight Line. І здійснюємо введення, послідовно кликаючи покажчиком миші по точках в Графічному вікні (рис. 3.15.).

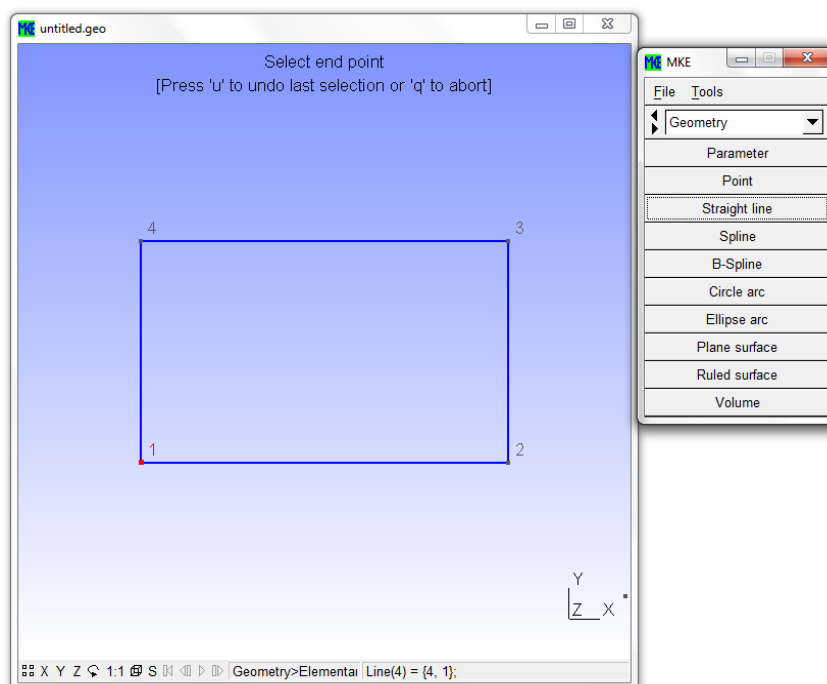


Рис. 3.15. Відображення ліній

Залишилося побудувати прямокутник (плоску фігуру) обмежений введеними лініями. Вибираємо:

Geometry =>ElementaryEntities =>Add =>New =>PlaneSurface

У заголовку Графічного вікна з'являється напис Select Boundary (Обери межу). Клікаємо по будь-якій лінії, що обмежує замкнутий контур і спостерігаємо "почервоніння". У заголовку Графічного вікна з'являється напис Select hole boundaries (Вибери межі отвору або порожнини усередині прямокутника). Оскільки ніяких отворів у нас спочатку не планувалося тиснемо "е" на клавіатурі для закінчення виділення.

У Графічному Вікні усередині прямокутної області позначилися пунктирні лінії, що утворюють хрест. Це - поточне позначення поверхні в програмі. Повертаємося в Geometry => Edit. Відкривається програма "Блокнот" і перед нами предстають команди, записані в процесі виконання дій в інтерактивному режимі.

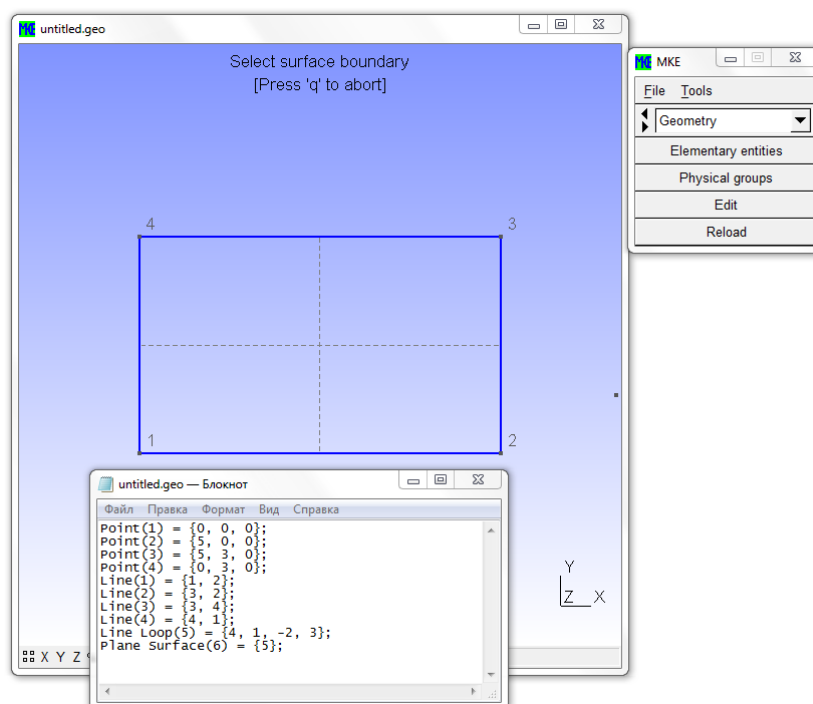


Рис. 3.16. Твердотіла модель прямокутника

Якщо ви хочете подивитися і підправити вміст файлу в процесі роботи, то обираєте Edit у вікні Меню (рис. 3.17.). Geo файл із записаними в ньому командами відкриється в текстовому редакторі. Після правки і збереження, вам треба виконати Reload, тобто перезавантажити модель.

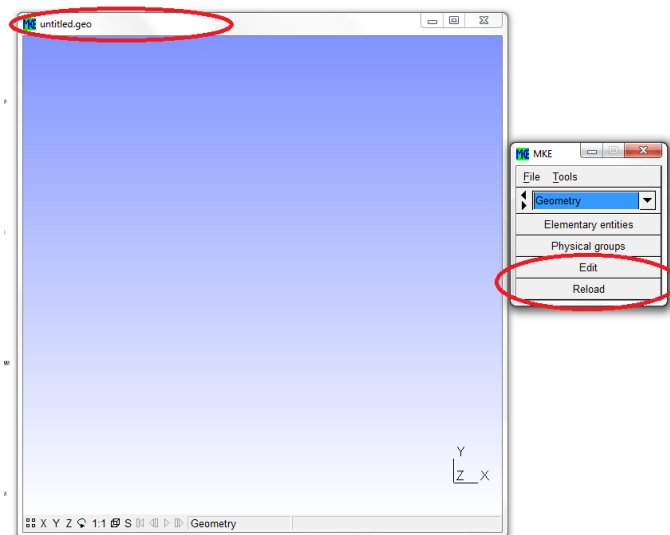


Рис. 3.17. Правка моделі

Побудуємо 3D - тіло витягуванням нашого прямокутника. Geometry => Elementary Entities => Extrude => Rotate => Surface.

Виникне вікно "Contextual Geometry Definitions" з активною вкладкою "Translation". Вводимо в нього значення, як показано на рис. 3.18, виділяємо показчиком миші нашу єдину поверхню і натискаємо клавішу "e" на клавіатурі.

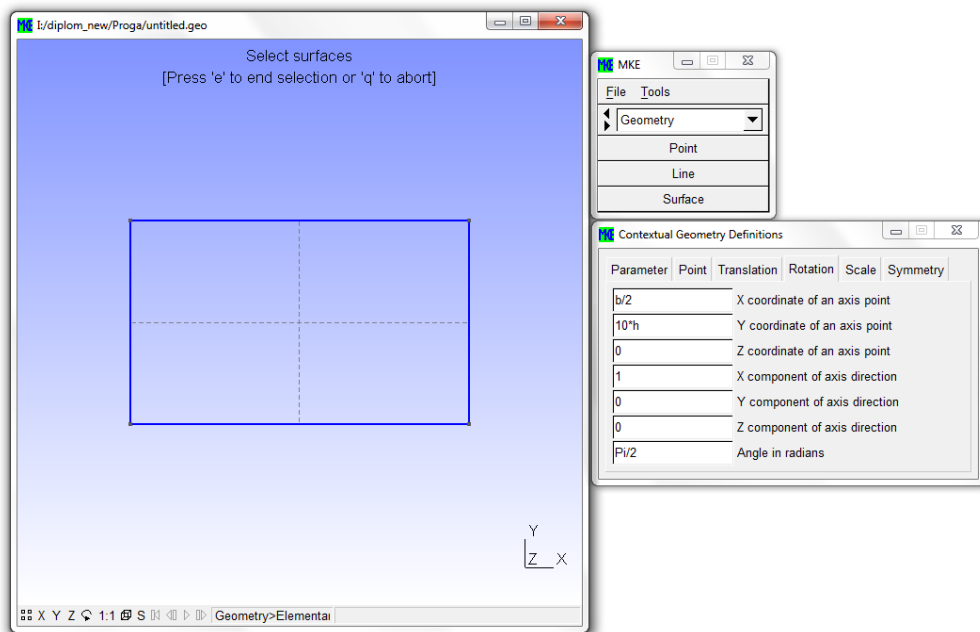


Рис. 3.18. Вікно "Contextual Geometry Definitions"

В результаті в гео -файл додається вираз:

```
Extrude {{1,0,0}, {b/2,10*h, 0}, Pi/2} {
    Surface{6};
}
```

А створене обертанням об'ємне тіло матиме вигляд як на рис. 3.19.

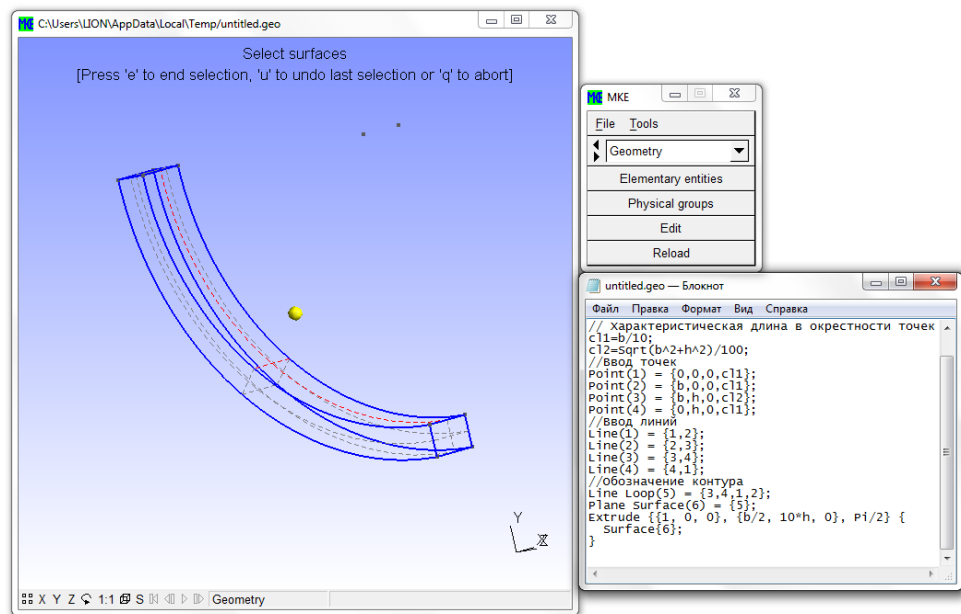


Рис. 3.19. Отримання об'ємного тіла обертанням

### 3.4. Генерація сітки в препроцесорі

Розглянемо дискретизацію плоских та тривимірних об'єктів на прикладі геометричних тіл описаних в пункті 3.3.

Виправимо і закоментуємо команди в geo – файлі прямокутниками таким чином:

```
//Початкові дані:
//Розміри
b=5;
h=3;
// Характеристична довжина в околиці точок
c11=b/10;
c12=Sqrt(b^2+h^2)/100;
//Ввід точок
Point(1) = {0,0,0,c11};
Point(2) = {b,0,0,c11};
Point(3) = {b,h,0,c12};
Point(4) = {0,h,0,c11};
//Ввод ліній
Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,1};
//Позначення контура
Line Loop(5) = {3,4,1,2};
//Поверхня прямокутника
Plane Surface(6) = {5};
```

Вибираємо в "Комбобоксі" Вікна Меню "Mesh" ("мэш" – тобто сітка) і натискаємо на кнопку "2D". Результат показаний на рис. 3.20. Вийшло те, що називають "тріангуляція", причому обертає на себе увагу місцеве згущування сітки в районі точки №3 внаслідок прийняття для неї відповідного значення характеристичної довжини.

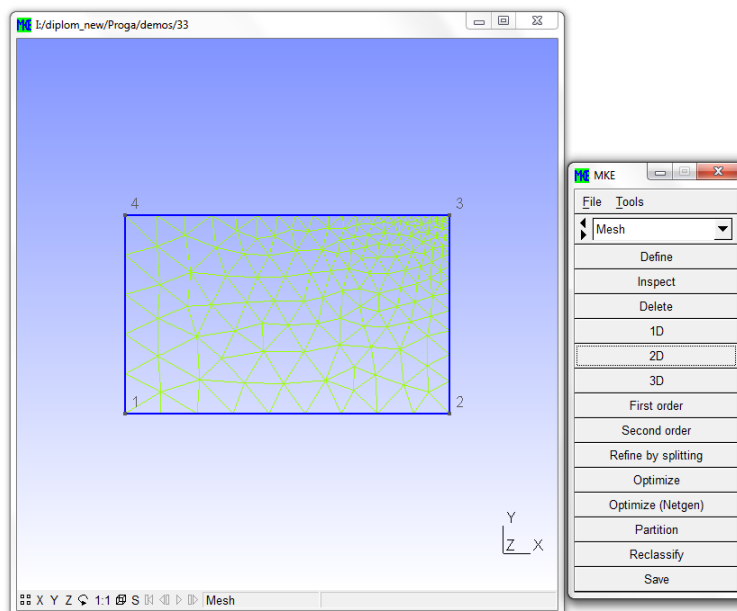


Рис. 3.20. Розбиття прямокутника на СЕ

Отже, у нас вийшли трьохвузлові елементи першого порядку. А якщо ми натиснемо на кнопку "Second Order" на екрані Графічного Вікна будуть вже шести-вузлові елементи (трикутні, з проміжними вузлами), як відомо - на порядок точніші з розрахункової точки зору, а отже що не вимагають великого подрібнення для отримання заданої точності. На практиці сітка згущується в зонах концентрації напруги, там де напруга змінюється різко на невеликому відрізку. Там, де градієнт напруги не високий, можна обійтися грубішою сіткою, а дрібнити усе однаково - непродуктивно з точки зору точності і економії машинних ресурсів.

Якщо ми зайдемо на вкладку: Mesh => Define => Recombine, виділимо поверхню, після закінчення виділення натиснемо "е" і розіб'ємо заново, то отримаємо результат, показаний на рис. 3.21, а в список команд додасться рядок:

Recombine Surface {6};



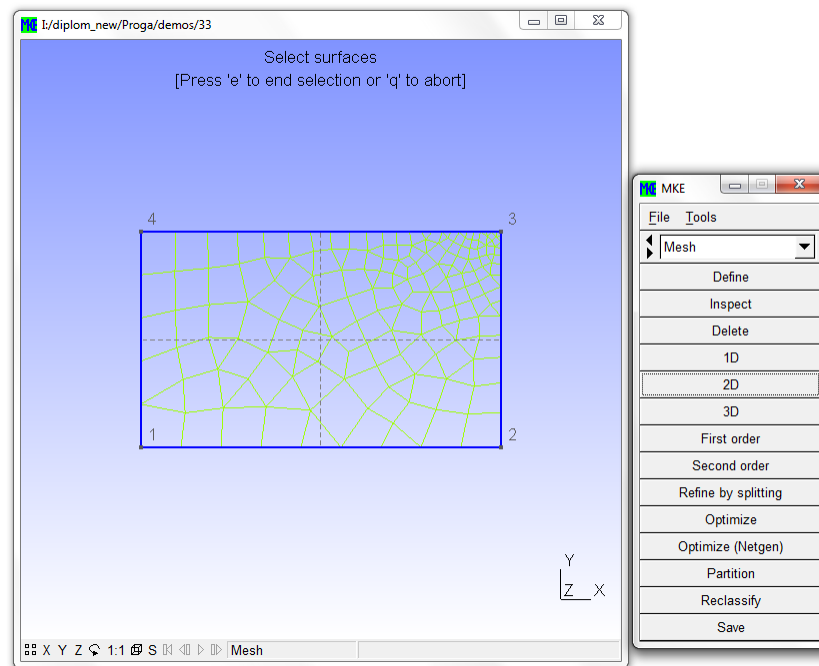


Рис. 3.21. Рекомбінована сітка

Отже, ми скористалися для розбиття на СЕ кнопкою "2D". "2D" - не обов'язково означає плоску сітку, нею треба користуватися і для розбиття криволінійних, просторових поверхонь і поверхонь, що обмежують об'ємні тіла.

Для розбиття об'ємних тіл на об'ємні елементи треба користуватися кнопкою "3D".

Для розбиття на кінцеві елементи встановимо нові значення параметра сітки, тобто перепризначаємо характеристичну довжину (при занадто великому числі елементів програма може вилетіти через нестачу пам'яті в Windows).

Заходимо на Mesh => Define => Characteristic length, що вводиться значення  $b/6$  замість 0,1, виділяємо точки в Графічному вікні і після закінчення виділення (усі точки почервоніють) даємо "е", як пропонує нам підказка в заголовку Графічного вікна (для виділення точок не поодиночці, а за допомогою ласо користуємося мишею з натиснутою і утримуваною клавішею "Ctrl"). В результаті в geo -файл додається вираз:

$$\text{Characteristic Length } \{26,28,3,2,4,1,5,14,6,10\} = b/6;$$

Заходимо на вкладку Mesh, і натискаємо кнопку 3D. Розбите на СЕ тіло показане на рис. 3.22.

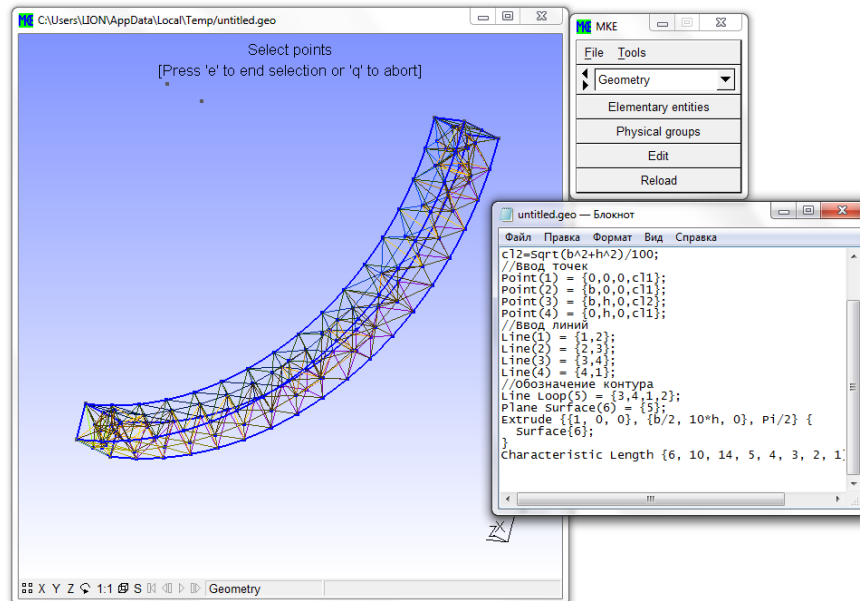


Рис. 3.20. Об'ємне тіло, розбите на скінченні елементи

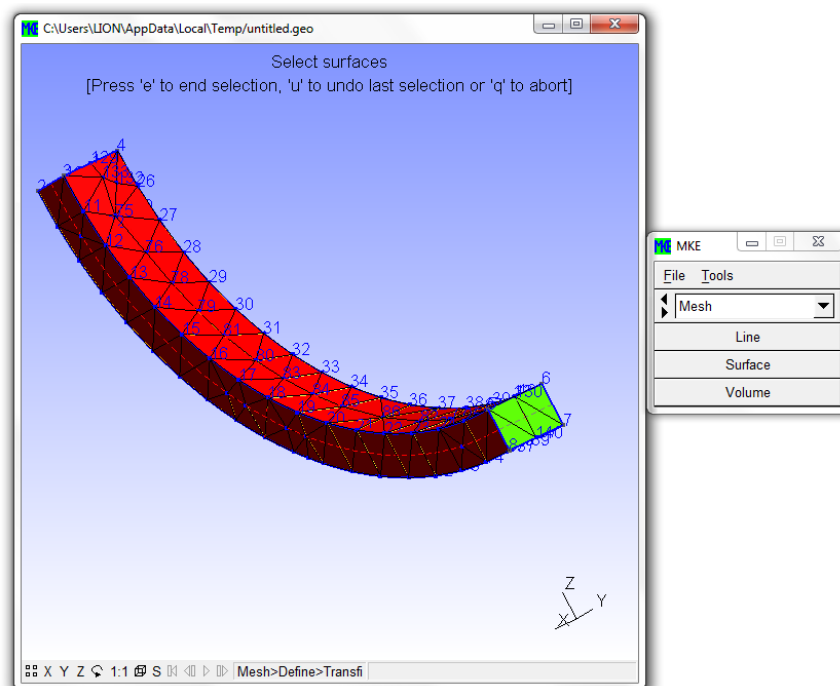


Рис. 3.23. Об'ємне тіло, розбите на скінченні елементи

### 3.5. Формати файлів в препроцесорі

Препроцесор нічого сам не обчислює, тому він має бути пов'язаний з іншими програмами за допомогою створюваних і прочитуваних ним файлів.

Зробити це можна через випадне меню "File" Вікна Меню, вибравши "Open" або "Save As" (рис. 3.24).

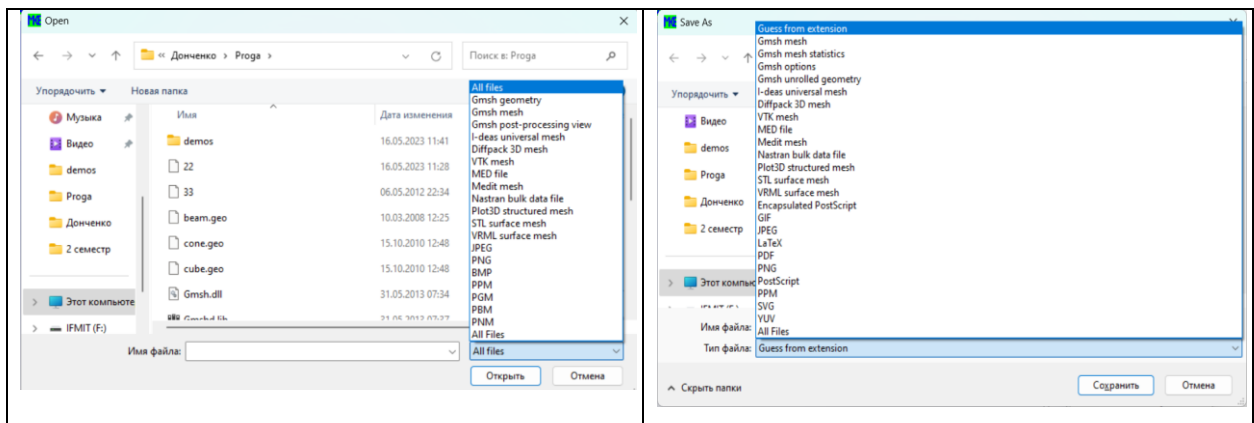


Рис. 3.24. Випадаюче меню "File": "Open" та "Save As"

Розглянемо деякі формати файлів по їх розширеннях:

- Gmsh geometry .geo - файл геометрії GMSH.
- STEP і IGES - відомі формати файлів геометрія, підтримувана багатьма CAD -пакетами. Імпорт STEP і IGES, розроблений на основі відкритої бібліотеки OpenCascade дозволяє завантажувати в програму моделі, побудовані в сторонніх CAD -ах (зворотне вивантаження моделі з .geo в ці формати доки не передбачена), в т.ч. у відкритому пакеті Salome.
- STL surface mesh - поширений формат для передачі поверхонь через трикутні сітки.
- .msh файл - файл сітки препроцесора.

### 3.6. Тестова задача

Сконструювати код, яким можна користуватися для визначення геометрії об'єктів зі складною структурою мікровключень, таких як композити, геофізичні середовища, різні суміші, бетон та інші, що в даний час дуже популярні у моделюванні процесів.

Як зразок матеріалу, що розглядається, виступає паралелепіпед. Тому поставимо його першим. Наведемо тут найкоротший спосіб визначення паралелепіпеда. Заодно скористаємося принципом модульного програмування і кожен значний фрагмент моделі оформлятимемо в окремому файлі.

#### **brick.geo**

```
//початкова точка паралелепіпеда
x0 = 0;
y0 = 0;
```

```

z0 = 0;
// розміри паралелепіпеда
lenX = 1;
lenY = 1;
lenZ = 1;
// характеристична довжина елементів розбиття паралелепіпеда
clBrick = lenX / 2;
// задаємо крапку
Point(1) = { x0, y0, z0, clBrick };
// отримуємо лінію
lineBrick[] = Extrude { lenX, 0, 0 } { Point{1}; };
// отримуємо поверхню
surfBrick[] = Extrude { 0, lenY, 0 } { Line{lineBrick[1]}; };
// отримуємо об'єм
volBrick[] = Extrude { 0, 0, lenZ } { Surface{surfBrick[1]}; };
// зберігаємо поверхню паралелепіпеда у вигляді масиву
brickSurface[] = Boundary { Volume{volBrick[1]}; };
// зберігаємо також номер замкнутого контуру поверхонь
sl = newreg;
Surface Loop(sl) = { brickSurface[] };
brickSurfaceLoop = sl;

```

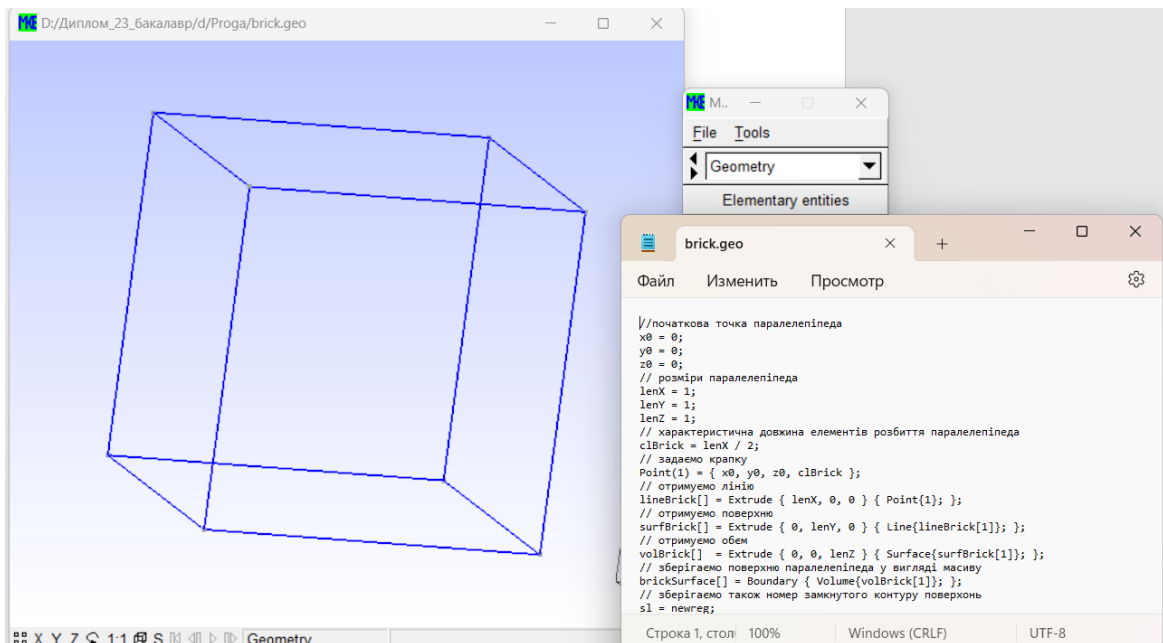


Рис. 3.25. brick.geo побудована модель

Тепер створимо кулю, що символізує мікровключення. Оформимо його як функції. Як і для паралелепіпеда, створимо новий файл.

#### sphere.geo

```

// центр кулі - (x_c, y_c, z_c)
// радіус кулі - rad
// характеристична довжина елементів розбиття кулі - clSphere
// порядковий номер кулі - number

Function SphereTemplate
// задаємо 3 точки сектора сфери
p1 = newpr; // команда newpr визначає новий унікальний номер точки
Point(p1) = { x_c, y_c, z_c, clSphere };
p2 = newpr;
Point(p2) = { x_c - rad, y_c, z_c, clSphere };

```

```

p3 = newp;
Point(p3) = { x_c, y_c - rad, z_c, clSphere };

// визначаємо одну лінію - півколо
line = newl; // команда newl визначає новий унікальний номер лінії
Circle(line) = { p2, p1, p3 };

// отримуємо поверхню - 1/8 частина всієї поверхні кулі
surfSphere1[] = Extrude { { rad, 0, 0 }, { x_c, y_c, z_c }, Pi / 2 }
                    { Line{line}; };
// отримуємо інші частини поверхні, копіюючи вихідну поверхню
// із дзеркальним відображенням по трьох площинах
surfSphere2[] = Symmetry { 1, 0, 0, -x_c }
                { Duplicata { Surface{surfSphere1[1]}; } };
surfSphere3[] = Symmetry { 0, 1, 0, -y_c }
                { Duplicata { Surface{surfSphere1[1],
                                     surfSphere2[0]}; } };
surfSphere4[] = Symmetry { 0, 0, 1, -z_c }
                { Duplicata { Surface{surfSphere1[1],
                                     surfSphere2[0],
                                     surfSphere3[0],
                                     surfSphere3[1]}; } };

// замкнутий контур поверхонь кулі
s11 = newsl; // команда newsl визначає новий унікальний номер
            // контура поверхонь
Surface Loop(s11) = { surfSphere1[1], surfSphere2[0],
                    surfSphere3[0], surfSphere3[1],
                    surfSphere4[0], surfSphere4[1],
                    surfSphere4[2], surfSphere4[3] };
surfaceLoops[number] = s11; // масив номерів контурів поверхонь

// задаємо кулю
v1 = newv; // команда newv визначає новий унікальний номер об'єму
Volume(v1) = { s11 };
sphereVolumes[number] = v1; // масив номерів об'ємів
Return

```

Починається функція з рядка `Function funcName` Закінчується словом `Return` (без ';' наприкінці). Жодних вхідних та вихідних параметрів немає. Просто ті змінні, які ви використовуєте в функції, повинні бути ініціалізовані до моменту виклику функції, що здійснюється командою `Call funcName`; (Тут вже ';' присутній). У нашій функції визначення кулі це змінні `x_c`, `y_c`, `z_c`, `rad`, `clSphere`, `number`.

Наведемо також список усіх команд типу `newp`, `newl`, що використовуються в нашому коді:

- `newp` визначає новий унікальний номер точки (point);
- `newl` визначає новий унікальний номер лінії (line);
- `newll` визначає новий унікальний номер контуру ліній (line loop);

- news визначає новий унікальний номер поверхні (surface);
- newsl визначає новий унікальний номер контуру поверхонь (surface loop);
- newv визначає новий унікальний номер об'єму (volume);
- newreg визначає новий унікальний номер регіону (region) - може використовуватись замість будь-якої з вищезгаданих команд.

Зберемо тепер наші готові геометричні елементи в одному файлі, який і визначить геометрію досліджуваного об'єкта.

### **model.geo**

```

Include "brick.geo";
Include "sphere.geo";

// Нехай центри включень розташовані не випадково,
// а у вузлах деякої тривимірної решітки.
// підхід, заснований на окремому завданні координат центрів включень,
// дозволить без великої зміни коду перейти
// наразі, коли включення розкидані паралелепіпедом випадковим чином

// кількість включень щодо кожного виміру
nInclusionsX = 2;
nInclusionsY = 2;
nInclusionsZ = 2;

// Радіус сферичного включення
radIncl = lenX/10; // величина lenX взята з "brick.geo"

// загальна кількість включень
nIncl = nInclusionsX * nInclusionsY * nInclusionsZ;

// координати центрів включень
n = 0;
For i In { 1:nInclusionsZ }
  For j In { 1:nInclusionsY }
    For k In { 1:nInclusionsX }
      // x0, y0, z0, lenX, lenY, lenZ взяті з "brick.geo"
      x[n] = x0 + k * lenX / (nInclusionsX + 1);
      y[n] = y0 + j * lenY / (nInclusionsY + 1);
      z[n] = z0 + i * lenZ / (nInclusionsZ + 1);
      n = n + 1;
    EndFor
  EndFor
EndFor

// Створення включень
For i In { 1:nIncl }
  x_c = x[i-1];
  y_c = y[i-1];
  z_c = z[i-1];
  rad = radIncl; // тут можна варіювати радіус куль,
                  // Створюючи різномасштабні включення
  clSphere = rad; // "Дрібність" сітки в кожній кулі також можна
                  варіювати

```

```

        number = i; // номер включення
        Call SphereTemplate; // виклик функції
    EndFor

    // остаточно оформляємо масив номерів контурів поверхонь
    surfaceLoops[0] = brickSurfaceLoop;

    // створюємо обсяг з урахуванням масиву контурів.
    // цей обсяг - те, що не зайняте включеннями
    v = newreg;
    Volume(v) = { surfaceLoops[] };

    Physical Volume(1001) = {v}; // скелет (матриця)
    Physical Volume(1002) = { sphereVolumes[] }; // пори (включення)

```

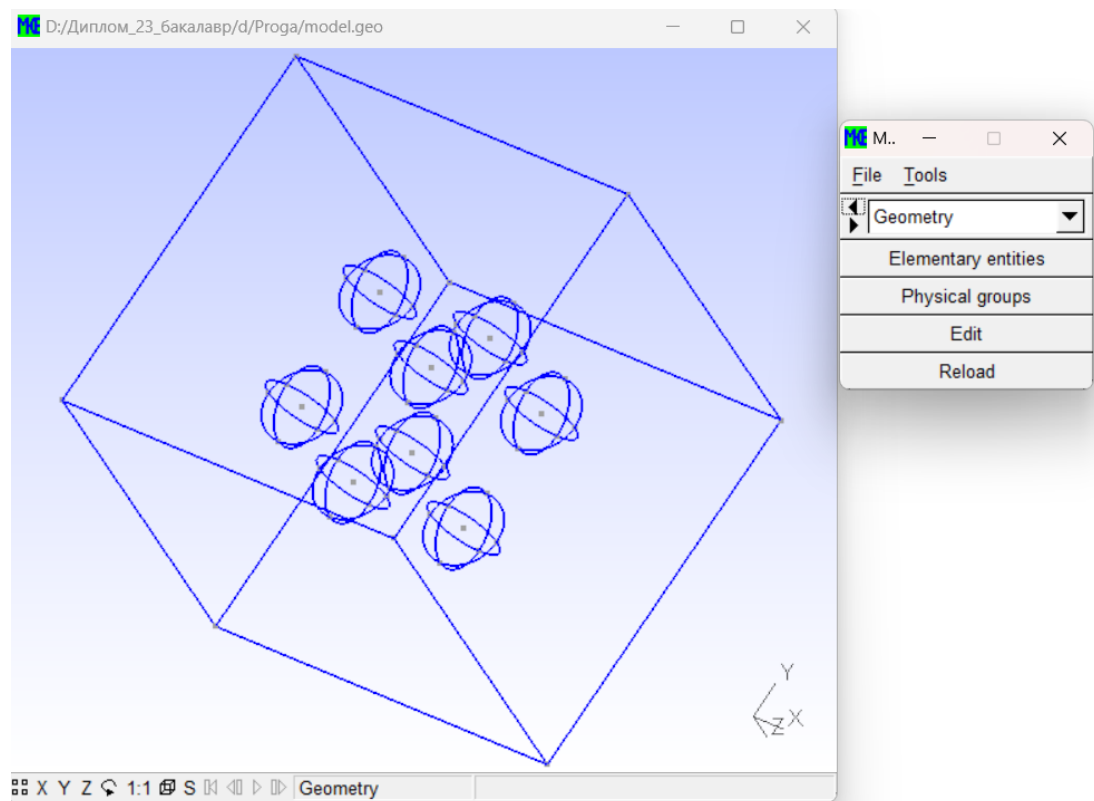


Рис. 3.26. model.geo побудована модель

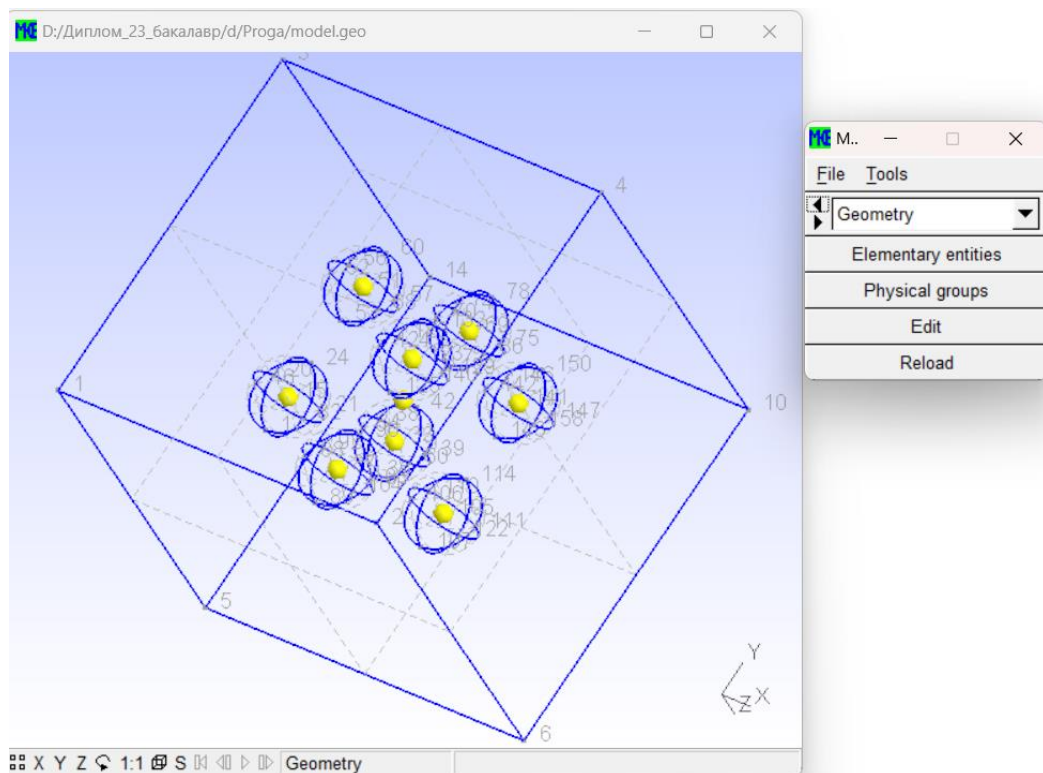


Рис. 3.27. model.geo побудована модель

Проведемо дискретизацію на 1D, 2D та на 3D скінчені елементи.

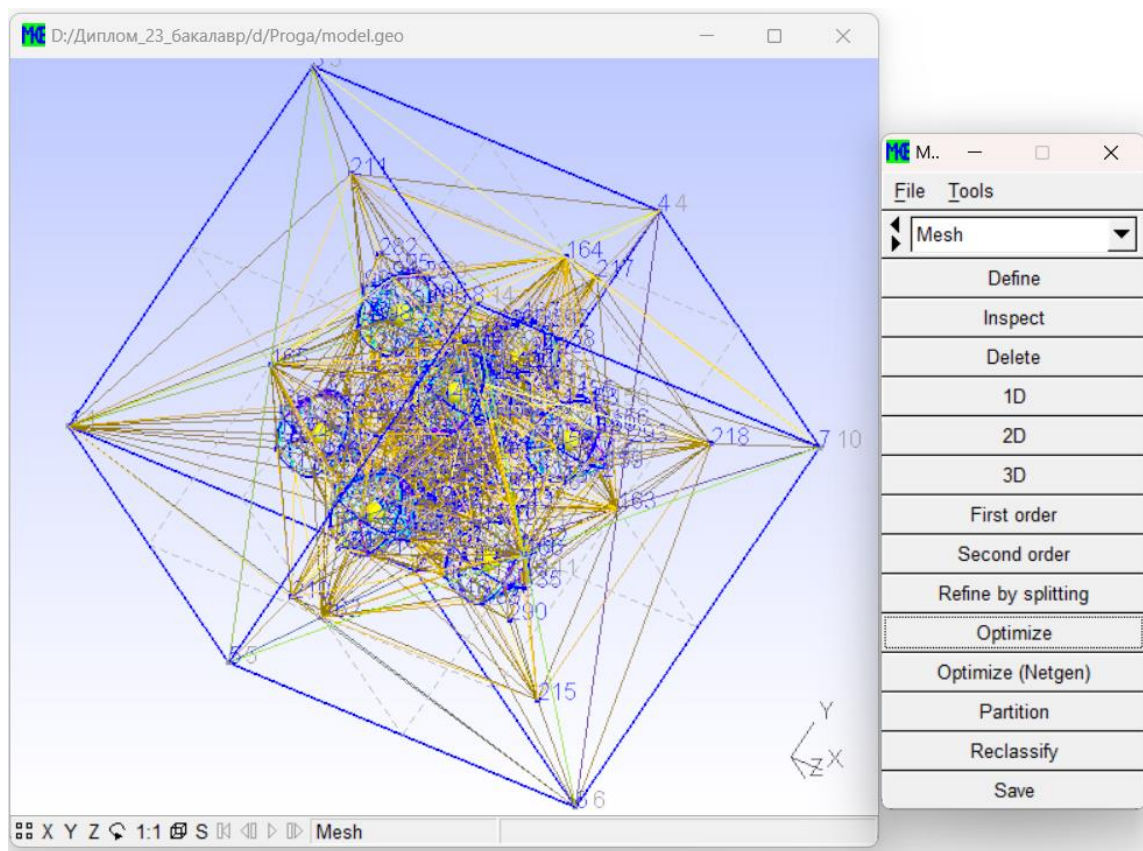


Рис. 3.28. Дискретизація на 3D скінчені елементи



### **3.7. Висновки до розділу**

В цьому розділі було розроблено загальний алгоритм роботи програми й інтерфейс на основі передбачуваної функціональності, були обрані програмні засоби для реалізації поставленого завдання. Програма реалізована на мові програмування C++ в середовищі Visual C++ 2022. При створенні засобів відображення і візуалізації використаний графічний інтерфейс OpenGL.

У препроцесорі реалізовано два модулі для роботи: геометрія, сітка. У модулі геометрія доступно створення простих геометричних сутностей, в модулі сітка доступно створення 1- 2 - і 3-х мірних сіток, і їх оптимізація.

## ВИСНОВКИ

У магістерській роботі було досліджено методи побудови комп'ютерних моделей геометричних областей та їх дискретизації на скінченні елементи заданої форми та розробка препроцесора для скінчено-елементного моделювання конструкцій.

Препроцесор є однією з найважливіших складових частин будь-якого програмного комплексу для чисельних розрахунків. Від якості реалізації препроцесора та побудованих за його допомогою геометричних та дискретних моделей проєктованих конструкцій залежить і якість всього програмного комплексу в цілому.

Для досягнення поставленої мети були вирішені наступні завдання:

- проведено аналіз найпоширеніших в наш час вітчизняних й зарубіжних програмних комплексів моделювання та аналізу напружено-деформованого стану складних інженерних конструкцій і споруд на основі МСЕ. Відзначено, що сучасні програмні комплекси автоматизації проєктування можна умовно розділити на три підсистеми: препроцесор, процесор і постпроцесор. Препроцесор відповідає за підготовку вихідних даних, яка включає такі етапи роботи, як опис геометричної моделі проєктованого об'єкта, та його дискретизацію на заданий тип скінченних елементів. Процесор виконує всі необхідні для конкретного типу задачі розрахунки: формує матриці мас, жорсткостей і демпфірування; будує систему лінійних алгебраїчних рівнянь; враховує початкові й граничні умови; розв'язує систему рівнянь і виводить результати розрахунку. Постпроцесор автоматизує процес аналізу результатів та генерацію документації.
- розглянуто основні методи побудови геометричних моделей об'єктів і поширені формати їх опису. Проаналізовано основні підходи, що застосовуються в сучасних САПР для

твердотільного моделювання геометричних об'єктів, а також основні поширені методи й алгоритми дискретизації плоских та просторових областей. Найпоширеніші алгоритми дискретизації можна розбити на дві частини: побудова первинної дискретизації (найбільш складний етап), та її оптимізація. У препроцесорі для скінчено- елементного моделювання конструкцій для первинної дискретизації області на скінченні елементи прийнято застосувати модифікований алгоритм Ватсона-Лавсона. Оптимізацію отриманої скінчено-елементної сітки робити шляхом застосування алгоритму Рапперта в плоскому випадку та Шевчука – в тривимірному.

- розроблено препроцесор для скінчено-елементного моделювання конструкцій. У препроцесорі реалізовано два модулі для роботи: геометрія, сітка. У модулі геометрія доступно створення простих геометричних сутностей, в модулі сітка доступно створення 1- 2 - і 3-х мірних сіток, і їх оптимізація.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities // <http://gmsh.info>
2. Arantes, E.R., Oliveira, E., Babuska, I., Zienkiewicz, O.C., Gado J.P. Proceedings International Conference on Accuracy Estimates and Adaptive Refinement in Finite Element Computation, Lisbon, 1984.
3. Baldwin K.H., Schreyer H.L. Automatic generation of quadrilateral elements by a conformal mapping. Eng. Comput. 1985, V.2, p. 187-194.
4. Blum H A transformation for extracting new descriptors of shape. In Models for the perception of speech and visual formed. By W.Wathen-Dunn, Cambridge, MA, the MIT Press, 1967, p.326-380
5. Brawn P.R. A non-interactive method for the automatic generation of finite element meshes using Schwarz-Christoffel transformation. Comp. Methods in Appl. Mech. Eng., 1981, V. 25, p. 101-126.
6. Cook W.A. Body oriented (natural) coordinates for generating three-dimensional meshes. Int. J. Num. Meth. Eng., 1974, V.8, p. 27-43.
7. D.A. Field The legacy of automatic mesh generation from solid modeling, Computer-Aided Geometric Des, V12, 1995, 651-674
8. Fleischmann P. Mesh Generation for Technology CAD in Three Dimensions [Электронный ресурс] / P. Fleischmann // Dissertation. - 1999. - Режим доступа: <http://www.iue.tuwien.ac.at/phd/fleischmann/diss.html>.
9. Foley J, A.van Dam, S.Feiner, J.Hughes, R.Philips Introduction to Computer Graphics, - Addison Wesley, 1994
10. Fomenko A.T., Kunii T.L. Topological modeling for visualization, Springer-Verlag, Tokyo and Heidelberg, 1997
11. Frey, P.J., George, P.-L. Mesh Generation: Application to Finite Elements - HERMES Science Europe, OXFORD & PARIS, 2000, 814p.
12. Fritsch F, Piccinini R.A. Cellular structures in topology. Cambridge University Press, Cambridge, 1990

13. George P.-L., Borouchaki H., Saltel E. 'Ultimate' robustness in meshing an arbitrary polyhedron // Int. J. Numer. Meth. Eng. 2003. Vol. 58. Pp. 1061–1089.
14. Geuzaine C., Remacle J.F. Gmsh: a three-dimensional finite element mesh generator with builtin pre- and post-processing facilities // International Journal for Numerical Methods in Engineering, 2009, Vol. 79, No 11, pp. 1309–1331.
15. Ho-Le, K. Finite. Element mesh generation methods: a review and classification. / CAD, 1988, V.20, N 1, p. 27-38.
16. I. Babushka, W.C. Rheinboldt. A-posteriori Error Estimates for Finite Element Method // Int. J. Numer. Meth. Eng., Vol. 12, p.p. 1597-1615, 1978.
17. Marot C., Pellerin J., Remacle J.F. One machine, one minute, three billion tetrahedral // International Journal for Numerical Methods in Engineering, 2019, Vol. 117, No 9, pp. 967–990.
18. Schöberl J. NETGEN. An advancing front 2d/3d mesh generator based on abstract rules // Computing and Visualization in Science, 1997, Vol. 1, No 1, pp. 41–52.
19. Ermakov M.K., Kryukov I.A. Supercomputer modeling of flow past hypersonic flight vehicles // Journal of Physics: Conference Series, 2017, Vol. 815, 012016
19. Si H. TetGen, A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator, 2006 // [www.wias-berlin.de/software/tetgen/files/tetgenmanual.pdf](http://www.wias-berlin.de/software/tetgen/files/tetgenmanual.pdf)
20. Si H. TetGen. A delaunay-based tetrahedral mesh generator // ACM Transactions on Mathematical Software, 2015, Vol. 41, No 2, Article 11, pp. 1–36.
21. Александров П.С. Комбинаторная топология. ОГИЗ, Гостехиздат, 1947
22. Галанин М.П. Разработка и реализация алгоритмов трехмерной триангуляции сложных пространственных областей: итерационные методы / Галанин М.П., Щеглов И.А. - М.: ИПМ им. М.В. Келдыша РАН,

2006. – № 9. – 32 с. - (Препринт / РАН, ИПМ им. М.В. Келдыша; 06-01-00421).
23. Галанин М.П. Разработка и реализация алгоритмов трехмерной триангуляции сложных пространственных областей: прямые методы / М.П. Галанин, И.А. Щеглов. – М.: ИПМ им. М.В. Келдыша РАН, 2006. – №10. – 32 с. – (Препринт / РАН, ИПМ им. М.В. Келдыша; 06-01-00421).
24. Голованов Н.Н. Геометрическое моделирование / Н.Н. Голованов. – М.: Издво Физ.-мат. лит., 2002. – 472 с.
25. Дижевский, А. Ю. Общий подход к реализации методов построения триангуляции неявно заданных поверхностей, использующих разбиение пространства на ячейки / А. Ю. Дижевский // Вычислительные методы и программирование. – 2007. – Т. 8. – С. 286–296.
26. Карташева Е.Л. Инструментальные средства подготовки и анализа данных для решения трехмерных задач математической физики - Математическое моделирование. 1997. Т. 9. №6. С. 20.
27. Ласло М. Вычислительная геометрия и компьютерная графика на C++ / Ласло М. Пер. с англ. М.: БИНОМ, 1997. 304 с.
28. Математическое моделирование: Проблемы и результаты, -М. Наука, 2003
29. Новые возможности Visual Studio 2022 [Электронный ресурс] // Microsoft Learn: [web-сайт]. – Режим доступа: <https://learn.microsoft.com/ruru/visualstudio/ide/whats-new-visual-studio-2022?view=vs-2022>
30. Пасько А.А., Пилюгин В.В., Покровский В.Н. Геометрическое моделирование в задаче анализа функций трех переменных, Сообщение ОИЯИ P10-86-310, Дубна, 1986. Publication in English: Computers and Graphics, vol.12, # 3/4, 1988, pp. 457-465.
31. Построение расчетных сеток: Теория и приложения – В сб. под ред. Иваненко С.А., Гаранжа В.А., ВЦ РАН, М. 2002

- 32.Препарата Ф. Вычислительная геометрия: Введение /Препарата Ф., Шеймос М. Пер. с англ. М.: Мир, 1989. 478 с.
- 33.Рвачев Л. Методы логической алгебры в математической физике. Наукова думка, Киев, 1974г.
- 34.Роджерс Д. Математические основы машинной графики / Роджерс Д., Адаме Дж. Пер. с англ. М.: Машиностроение, 1980. 204 с.
- 35.Скворцов А.В. Обзор алгоритмов построения триангуляции Делоне / А.В. Скворцов // Вычислительные методы и программирование. – 2002. – Т.3. – С. 14-39.
- 36.Скворцов А.В. Применение триангуляции для решения задач вычислительной геометрии / Скворцов А.В., Костюк Ю.Л. Геоинформатика: Теория и практика. Вып. 1. Томск: Изд-во Томск, ун-та, 1998. С. 127-138.
- 37.Скворцов А.В. Триангуляция Делоне и ее применение / А.В. Скворцов // Томск, Изд-во Том. ун-та, 2002. С. 128.
- 38.Соллогуб А.И., Вальшин А.Т. Система трехмерного геометрического моделирования пространственных тел с использованием характеристических и R-функций. Программирование, 1991, N3, с.86-96
- 39.Сьярле Ф. Метод конечных элементов для эллиптических задач / Ф. Сьярле; пер. с англ. Б.И. Квасова. – М.: Изд-во «Мир», 1980. – 512 с.
- 40.Химушин Ф.Ф. Конструктивное геометрическое моделирование. –В. сб. Программное обеспечение САПР. ВЦАН СССР,М., 1987, с.102-121
- 41.Химушин Ф.Ф. Обзор методов геометрического моделирования для САПР.- В сб. Программное обеспечение САПР, ВЦ АН СССР, М., 1987, с.78-101
- 42.ЧельцовА.Ю., ДавыдченкоЭ.А. Представление данных и алгоритмы для системы геометрического моделирования, основанной на заметании. - В сб. Программное обеспечение САПР, ВЦАН СССР, М., 1987, с.121-133.
- 43.Шайдуров В.В. Многосеточные методы конечных элементов / В.В. Шайдуров. – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 288 с.

44.Шайдуров В.В. Многосеточные методы конечных элементов. - М.,  
Наука, 1989. - 288с.



## ДОДАТОК А

### Generator.h

```
#ifndef _GENERATOR_H_
#define _GENERATOR_H_
class GModel;
void GetStatistics(double stat[50], double quality[4][100]=0);
void AdaptMesh(GModel *m);
void GenerateMesh(GModel *m, int dimension);
void OptimizeMesh(GModel *m);
void OptimizeMeshNetgen(GModel *m);
void RefineMesh(GModel *m, bool linear, bool splitIntoQuads=false,
                bool splitIntoHexas=false);
#endif
```

### Generator.cpp

```
#include <stdlib.h>
#include "GmshConfig.h"
#include "GmshMessage.h"
#include "Numeric.h"
#include "Context.h"
#include "OS.h"
#include "GModel.h"
#include "MLine.h"
#include "MTriangle.h"
#include "MQuadrangle.h"
#include "MTetrahedron.h"
#include "MHexahedron.h"
#include "MPrism.h"
#include "MPyramid.h"
#include "meshGEdge.h"
#include "meshGFace.h"
#include "meshGFaceBDS.h"
#include "meshGRegion.h"
#include "BackgroundMesh.h"
#include "BoundaryLayers.h"
#include "HighOrder.h"
#include "Generator.h"
#if !defined(HAVE_NO_POST)
#include "PView.h"
#include "PViewData.h"
#endif
static MVertex* isEquivalentTo(std::multimap<MVertex*, MVertex*>&m, MVertex *v)
{
    std::multimap<MVertex*, MVertex*>::iterator it = m.lower_bound(v);
    std::multimap<MVertex*, MVertex*>::iterator ite = m.upper_bound(v);
    if (it == ite) return v;
    MVertex *res = it->second; ++it;
    while (it != ite){
        res = std::min(res, it->second); ++it;
    }
    if (res < v) return isEquivalentTo(m, res);
    return res;
}
```

```

}
static void buildASetOfEquivalentMeshVertices(GFace *gf,
                                              std::multimap<MVertex*, MVertex*>&equivalent,
                                              std::map<GVertex*, MVertex*>&bm)
{
    // an edge is degenerated when its length is considered to be
    // zero. In some cases, a model edge can be considered as too
    // small and is ignored.
    // for taking that into account, we loop over the edges
    // and create pairs of MVertices that are considered as
    // equal.
    std::
    <GEdge*> edges = gf->edges();
    std::list<GEdge*> emb_edges = gf->embeddedEdges();
    std::list<GEdge*>::iterator it = edges.begin();
    while(it != edges.end()){
        if((*it)->isMeshDegenerated()){
            MVertex *va = ((*it)->getBeginVertex()->mesh_vertices.begin());
            MVertex *vb = ((*it)->getEndVertex()->mesh_vertices.begin());
            if (va != vb){
                equivalent.insert(std::make_pair(va, vb));
                equivalent.insert(std::make_pair(vb, va));
                bm[(*it)->getBeginVertex()] = va;
                bm[(*it)->getEndVertex()] = vb;
                printf("%d equivalent to %d\n", va->getNum(), vb->getNum());
            }
        }
        ++it;
    }
    it = emb_edges.begin();
    while(it != emb_edges.end()){
        if((*it)->isMeshDegenerated()){
            MVertex *va = ((*it)->getBeginVertex()->mesh_vertices.begin());
            MVertex *vb = ((*it)->getEndVertex()->mesh_vertices.begin());
            if (va != vb){
                equivalent.insert(std::make_pair(va, vb));
                equivalent.insert(std::make_pair(vb, va));
                bm[(*it)->getBeginVertex()] = va;
                bm[(*it)->getEndVertex()] = vb;
            }
        }
        ++it;
    }
}

struct geomTresholdVertexEquivalence
{
    // Initial MVertex associated to one given MVertex
    std::map<GVertex*, MVertex*> backward_map;
    // initiate the forward and backward maps
    geomTresholdVertexEquivalence(GModel *g);
    // restores the initial state
    ~geomTresholdVertexEquivalence ();
}

```

```

};
geomTresholdVertexEquivalence::geomTresholdVertexEquivalence(GModel *g)
{
    std::multimap<MVertex*, MVertex*> equivalenceMap;
    for (GModel::fiter it = g->firstFace(); it != g->lastFace(); ++it)
        buildASetOfEquivalentMeshVertices(*it, equivalenceMap, backward_map);
    // build the structure that identifiates geometrically equivalent
    // mesh vertices.
    for (std::map<GVertex*, MVertex*>::iterator it = backward_map.begin();
        it != backward_map.end(); ++it){
        GVertex *g = it->first;
        MVertex *v = it->second;
        MVertex *other = isEquivalentTo(equivalenceMap, v);
        if (v != other){
            printf("Finally : %d equivalent to %d\n", v->getNum(), other->getNum());
            g->mesh_vertices.clear();
            g->mesh_vertices.push_back(other);
            std::list<GEdge*> ed = g->edges();
            for (std::list<GEdge*>::iterator ite = ed.begin() ; ite != ed.end() ; ++ite){
                std::vector<MLine*> newl;
                for (unsigned int i = 0; i < (*ite)->lines.size(); ++i){
                    MLine *l = (*ite)->lines[i];
                    MVertex *v1 = l->getVertex(0);
                    MVertex *v2 = l->getVertex(1);
                    if (v1 == v && v2 != other){
                        delete l;
                        l = new MLine(other,v2);
                        newl.push_back(l);
                    }
                    else if (v1 != other && v2 == v){
                        delete l;
                        l = new MLine(v1,other);
                        newl.push_back(l);
                    }
                    else if (v1 != v && v2 != v)
                        newl.push_back(l);
                    else
                        delete l;
                }
                (*ite)->lines = newl;
            }
        }
    }
}

geomTresholdVertexEquivalence::~geomTresholdVertexEquivalence()
{
    // restore the initial data
    for (std::map<GVertex*, MVertex*>::iterator it = backward_map.begin();
        it != backward_map.end() ; ++it){
        GVertex *g = it->first;
        MVertex *v = it->second;
        g->mesh_vertices.clear();
    }
}

```

```

    g->mesh_vertices.push_back(v);
}
}

template<class T>
static void GetQualityMeasure(std::vector<T*>&ele,
                             double &gamma, double &gammaMin, double &gammaMax,
                             double &eta, double &etaMin, double &etaMax,
                             double &rho, double &rhoMin, double &rhoMax,
                             double &disto, double &distoMin, double &distoMax,
                             double quality[4][100])
{
    for(unsigned int i = 0; i < ele.size(); i++){
        double g = ele[i]->gammaShapeMeasure();
        gamma += g;
        gammaMin = std::min(gammaMin, g);
        gammaMax = std::max(gammaMax, g);
        double e = ele[i]->etaShapeMeasure();
        eta += e;
        etaMin = std::min(etaMin, e);
        etaMax = std::max(etaMax, e);
        double r = ele[i]->rhoShapeMeasure();
        rho += r;
        rhoMin = std::min(rhoMin, r);
        rhoMax = std::max(rhoMax, r);
        double d = ele[i]->distoShapeMeasure();
        disto += d;
        distoMin = std::min(distoMin, d);
        distoMax = std::max(distoMax, d);
        for(int j = 0; j < 100; j++){
            if(g > j / 100. && g <= (j + 1) / 100.) quality[0][j]++;
            if(e > j / 100. && e <= (j + 1) / 100.) quality[1][j]++;
            if(r > j / 100. && r <= (j + 1) / 100.) quality[2][j]++;
            if(d > j / 100. && d <= (j + 1) / 100.) quality[3][j]++;
        }
    }
}

void GetStatistics(double stat[50], double quality[4][100])
{
    for(int i = 0; i < 50; i++) stat[i] = 0.;
    GModel *m = GModel::current();
    if(!m) return;
    stat[0] = m->getNumVertices();
    stat[1] = m->getNumEdges();
    stat[2] = m->getNumFaces();
    stat[3] = m->getNumRegions();
    std::map<int, std::vector<GEntity*>> physicals[4];
    m->getPhysicalGroups(physicals);
    stat[45] = physicals[0].size() + physicals[1].size() +
        physicals[2].size() + physicals[3].size();
    for(GModel::eiter it = m->firstEdge(); it != m->lastEdge(); ++it)
        stat[4] += (*it)->mesh_vertices.size();
}

```

```

for(GModel::fiter it = m->firstFace(); it != m->lastFace(); ++it){
stat[5] += (*it)->mesh_vertices.size();
stat[7] += (*it)->triangles.size();
stat[8] += (*it)->quadrangles.size();
}
for(GModel::riter it = m->firstRegion(); it != m->lastRegion(); ++it){
stat[6] += (*it)->mesh_vertices.size();
stat[9] += (*it)->tetrahedra.size();
stat[10] += (*it)->hexahedra.size();
stat[11] += (*it)->prisms.size();
stat[12] += (*it)->pyramids.size();
}
stat[13] = CTX::instance()->meshTimer[0];
stat[14] = CTX::instance()->meshTimer[1];
stat[15] = CTX::instance()->meshTimer[2];
if(quality){
for(int i = 0; i < 3; i++){
for(int j = 0; j < 100; j++){
quality[i][j] = 0.;
double gamma=0., gammaMin=1., gammaMax=0.;
double eta=0., etaMin=1., etaMax=0.;
double rho=0., rhoMin=1., rhoMax=0.;
double disto=0., distoMin=1., distoMax=0.;
if (m->firstRegion() == m->lastRegion()){
for(GModel::fiter it = m->firstFace(); it != m->lastFace(); ++it){
GetQualityMeasure((*it)->quadrangles, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
GetQualityMeasure((*it)->triangles, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
}
}
else{
for(GModel::riter it = m->firstRegion(); it != m->lastRegion(); ++it){
GetQualityMeasure((*it)->tetrahedra, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
GetQualityMeasure((*it)->hexahedra, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
GetQualityMeasure((*it)->prisms, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
GetQualityMeasure((*it)->pyramids, gamma, gammaMin, gammaMax,
eta, etaMin, etaMax, rho, rhoMin, rhoMax,
disto, distoMin, distoMax, quality);
}
}
double N = stat[9] + stat[10] + stat[11] + stat[12];
stat[17] = N ? gamma / N : 0.;
stat[18] = gammaMin;

```

```

    stat[19] = gammaMax;
    stat[20] = N ? eta / N : 0.;
    stat[21] = etaMin;
    stat[22] = etaMax;
    stat[23] = N ? rho / N : 0;
    stat[24] = rhoMin;
    stat[25] = rhoMax;
    stat[46] = N ? disto / N : 0;
    stat[47] = distoMin;
    stat[48] = distoMax;
}
#ifdef HAVE_NO_POST
stat[26] = PView::list.size();
for(unsigned int i = 0; i < PView::list.size(); i++) {
    PViewData *data = PView::list[i]->getData(true);
    stat[27] += data->getNumPoints();
    stat[28] += data->getNumLines();
    stat[29] += data->getNumTriangles();
    stat[30] += data->getNumQuadrangles();
    stat[31] += data->getNumTetrahedra();
    stat[32] += data->getNumHexahedra();
    stat[33] += data->getNumPrisms();
    stat[34] += data->getNumPyramids();
    stat[35] += data->getNumStrings2D() + data->getNumStrings3D();
}
#endif
}
static bool TooManyElements(GModel *m, int dim)
{
    if(CTX::instance()->expertMode || !m->getNumVertices()) return false;
    // try to detect obvious mistakes in characteristic lengths (one of
    // the most common cause for erroneous bug reports on the mailing
    // list)
    double sumAllLc = 0.;
    for(GModel::viter it = m->firstVertex(); it != m->lastVertex(); ++it)
        sumAllLc += (*it)->prescribedMeshSizeAtVertex() * CTX::instance()->mesh.lcFactor;
    sumAllLc /= (double)m->getNumVertices();
    if(!sumAllLc || pow(CTX::instance()->lc / sumAllLc, dim) > 1.e10)
        return !Msg::GetBinaryAnswer
            ("Your choice of characteristic lengths will likely produce a very\n"
             "large mesh. Do you really want to continue?\n\n"
             "(To disable this warning in the future, select `Enable expert mode`\n"
             "in the option dialog.)",
             "Continue", "Cancel");
    return false;
}
static bool CancelDelaunayHybrid(GModel *m)
{
    if(CTX::instance()->expertMode) return false;
    int n = 0;
    for(GModel::riter it = m->firstRegion(); it != m->lastRegion(); ++it)
        n += (*it)->getNumMeshElements();
}

```



```

    nTotGoodLength += (*it)->meshStatistics.nbGoodLength;
    nTotGoodQuality += (*it)->meshStatistics.nbGoodQuality;
    numFaces++;
}
fprintf(statreport, "\\t%16s\\t%d\\t%d\\t", m->getName().c_str(), numFaces, nUnmeshed);
fprintf(statreport, "%d\\t%8.7f\\t%8.7f\\t%8.7f\\t%8.7f\\t",
        nTotT, avg / (double)nTotT, best, worst, nTotGoodQuality,
        (double)nTotGoodQuality / nTotT);
fprintf(statreport, "%d\\t%8.7f\\t%8.7f\\t%8.1f\\n",
        nTotE, exp(e_avg / (double)nTotE), nTotGoodLength,
        (double)nTotGoodLength / nTotE, CTX::instance()->meshTimer[1]);
fclose(statreport);
}
static void Mesh2D(GModel *m)
{
    if(TooManyElements(m, 2)) return;
    Msg::StatusBar(1, true, "Meshing 2D...");
    double t1 = Cpu();
    // skip short mesh edges
    geomTresholdVertexEquivalence inst(m);

    if(!Mesh2DWithBoundaryLayers(m)){
        std::for_each(m->firstFace(), m->lastFace(), meshGFace());
        int nIter = 0;
        while(1){
            meshGFace mesher;
            int nbPending = 0;
            for(GModel::fiter it = m->firstFace(); it != m->lastFace(); ++it){
                if ((*it)->meshStatistics.status == GFace::PENDING){
                    mesher(*it);
                    nbPending++;
                }
            }
            if(!nbPending) break;
            if(nIter++ > 10) break;
        }
    }
    // collapseSmallEdges(*m);
    double t2 = Cpu();
    CTX::instance()->meshTimer[1] = t2 - t1;
    Msg::Info("Mesh 2D complete (%g s)", CTX::instance()->meshTimer[1]);
    Msg::StatusBar(1, false, "Mesh");
    PrintMesh2dStatistics(m);
}
static void FindConnectedRegions(std::vector<GRegion*>&delaunay,
                                std::vector<std::vector<GRegion*>>&connected)
{
    // FIXME: need to split region vector into connected components here!
    connected.push_back(delaunay);
}
static void Mesh3D(GModel *m)
{

```



```

if(TooManyElements(m, 3)) return;
Msg::StatusBar(1, true, "Meshing 3D...");
double t1 = Cpu();
// mesh the extruded volumes first
std::for_each(m->firstRegion(), m->lastRegion(), meshGRegionExtruded());
// then subdivide if necessary (unfortunately the subdivision is a
// global operation, which can require changing the surface mesh!)
SubdivideExtrudedMesh(m);
// then mesh all the non-delaunay regions
std::vector<GRegion*> delaunay;
std::for_each(m->firstRegion(), m->lastRegion(), meshGRegion(delaunay));
// warn if attempting to use Delaunay for mixed meshes
if(delaunay.size() && CancelDelaunayHybrid(m)) return;
// and finally mesh the delaunay regions (again, this is global; but
// we mesh each connected part separately for performance and mesh
// quality reasons)
std::vector<std::vector<GRegion*>> connected;
FindConnectedRegions(delaunay, connected);
for(unsigned int i = 0; i < connected.size(); i++){
    MeshDelaunayVolume(connected[i]);
}
double t2 = Cpu();
CTX::instance()->meshTimer[2] = t2 - t1;
Msg::Info("Mesh 3D complete (%g s)", CTX::instance()->meshTimer[2]);
Msg::StatusBar(1, false, "Mesh");
}

void OptimizeMeshNetgen(GModel *m)
{
    Msg::StatusBar(1, true, "Optimizing 3D with Netgen...");
    double t1 = Cpu();
    std::for_each(m->firstRegion(), m->lastRegion(), optimizeMeshGRegionNetgen());
    double t2 = Cpu();
    Msg::Info("Mesh 3D optimization with Netgen complete (%g s)", t2 - t1);
    Msg::StatusBar(1, false, "Mesh");
}

void OptimizeMesh(GModel *m)
{
    Msg::StatusBar(1, true, "Optimizing 3D...");
    double t1 = Cpu();
    std::for_each(m->firstRegion(), m->lastRegion(), optimizeMeshGRegionGmsh());
    double t2 = Cpu();
    Msg::Info("Mesh 3D optimization complete (%g s)", t2 - t1);
    Msg::StatusBar(1, false, "Mesh");
}

void AdaptMesh(GModel *m)
{
    Msg::StatusBar(1, true, "Adapting 3D Mesh...");
    double t1 = Cpu();
    if(CTX::instance()->lock) {
        Msg::Info("I'm busy! Ask me that later...");
        return;
    }
}

```

```

CTX::instance()->lock = 1;
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
std::for_each(m->firstRegion(), m->lastRegion(), adaptMeshGRegion());
double t2 = Cpu();
Msg::Info("Mesh Adaptation complete (%g s)", t2 - t1);
Msg::StatusBar(1, false, "Mesh");
}
void GenerateMesh(GModel *m, int ask)
{
    if(CTX::instance()->lock) {

        Msg::Info("I'm busy! Ask me that later...");
        return;
    }
    CTX::instance()->lock = 1;
    Msg::ResetErrorCounter();
    int old = m->getMeshStatus(false);
    // Initialize pseudo random mesh generator with the same seed
    srand(1);
    // Change any high order elements back into first order ones
    SetOrder1(m);
    // 1D mesh
    if(ask == 1 || (ask > 1 && old < 1)) {
        std::for_each(m->firstRegion(), m->lastRegion(), deMeshGRegion());
        std::for_each(m->firstFace(), m->lastFace(), deMeshGFace());
        Mesh1D(m);
    }
    // 2D mesh
    if(ask == 2 || (ask > 2 && old < 2)) {
        std::for_each(m->firstRegion(), m->lastRegion(), deMeshGRegion());
        Mesh2D(m);
    }
    // 3D mesh
    if(ask == 3) {
        Mesh3D(m);
    }
    // Orient the surface mesh so that it matches the geometry
    if(m->getMeshStatus() >= 2)
        std::for_each(m->firstFace(), m->lastFace(), orientMeshGFace());
    // Optimize quality of 3D tet mesh
    if(m->getMeshStatus() == 3){
        for(int i = 0; i < std::max(CTX::instance()->mesh.optimize,
                                   CTX::instance()->mesh.optimizeNetgen); i++){
            if(CTX::instance()->mesh.optimize > i) OptimizeMesh(m);
        }
    }
}

```

```

    if(CTX::instance()->mesh.optimizeNetgen > i) OptimizeMeshNetgen(m);
}
}
// Subdivide into quads or hexas
if(m->getMeshStatus() == 2 && CTX::instance()->mesh.algoSubdivide == 1)
    RefineMesh(m, CTX::instance()->mesh.secondOrderLinear, true);
else if(m->getMeshStatus() == 3 && CTX::instance()->mesh.algoSubdivide == 2)
    RefineMesh(m, CTX::instance()->mesh.secondOrderLinear, false, true);
// Create high order elements
if(m->getMeshStatus() && CTX::instance()->mesh.order > 1)
    SetOrderN(m, CTX::instance()->mesh.order, CTX::instance()->mesh.secondOrderLinear,
        CTX::instance()->mesh.secondOrderIncomplete);
Msg::Info("%d vertices %d elements",
    m->getNumMeshVertices(), m->getNumMeshElements());
Msg::PrintErrorCounter("Mesh generation error summary");
CTX::instance()->lock = 0;
CTX::instance()->mesh.changed = ENT_ALL;
}
MTriangle.h
#ifndef _MTRIANGLE_H_
#define _MTRIANGLE_H_
#include "MElement.h"
/*
* MTriangle
*
* v
* ^
* |
* 2
* | \
* |  \
* |   \
* |    \
* |     \
* 0-----1 --> u
*
*/
class MTriangle : public MElement {
protected:
    MVertex *_v[3];
    void _getEdgeVertices(const int num, std::vector<MVertex*> &v) const
    {
        v[0] = _v[edges_tri(num, 0)];
        v[1] = _v[edges_tri(num, 1)];
    }
    void _getFaceVertices(std::vector<MVertex*> &v) const
    {
        v[0] = _v[0];
        v[1] = _v[1];
        v[2] = _v[2];
    }
public :

```

```

MTriangle(MVertex *v0, MVertex *v1, MVertex *v2, int num=0, int part=0)
: MElement(num, part)
{
    _v[0] = v0; _v[1] = v1; _v[2] = v2;
}
MTriangle(std::vector<MVertex*> &v, int num=0, int part=0)
: MElement(num, part)
{
    for(int i = 0; i < 3; i++) _v[i] = v[i];
}
~MTriangle(){}
virtual int getDim(){ return 2; }
virtual double gammaShapeMeasure();
virtual double distoShapeMeasure();
virtual int getNumVertices() const { return 3; }
virtual MVertex *getVertex(int num){ return _v[num]; }
virtual MVertex *getVertexMED(int num)
{
    static const int map[3] = {0, 2, 1};
    return getVertex(map[num]);
}
virtual MVertex *getOtherVertex(MVertex *v1, MVertex *v2)
{
    if(_v[0] != v1 && _v[0] != v2) return _v[0];
    if(_v[1] != v1 && _v[1] != v2) return _v[1];
    if(_v[2] != v1 && _v[2] != v2) return _v[2];
    return 0;
}
virtual int getNumEdges(){ return 3; }
virtual MEdge getEdge(int num)
{
    return MEdge(_v[edges_tri(num, 0)], _v[edges_tri(num, 1)]);
}
virtual int getNumEdgesRep(){ return 3; }
virtual void getEdgeRep(int num, double *x, double *y, double *z, SVector3 *n)
{
    MEdge e(getEdge(num));
    _getEdgeRep(e.getVertex(0), e.getVertex(1), x, y, z, n, 0);
}
virtual void getEdgeVertices(const int num, std::vector<MVertex*> &v) const
{
    v.resize(2);
    _getEdgeVertices(num, v);
}
virtual int getNumFaces(){ return 1; }
virtual MFace getFace(int num)
{
    return MFace(_v[0], _v[1], _v[2]);
}
virtual int getNumFacesRep(){ return 1; }
virtual void getFaceRep(int num, double *x, double *y, double *z, SVector3 *n)
{

```

```

    _getFaceRep(_v[0], _v[1], _v[2], x, y, z, n);
}
virtual void getFaceVertices(const int num, std::vector<MVertex*> &v) const
{
    v.resize(3);
    _getFaceVertices(v);
}
virtual int getType() const { return TYPE_TRI; }
virtual int getTypeForMSH() const { return MSH_TRI_3; }
virtual int getTypeForUNV() const { return 91; } // thin shell linear triangle
virtual int getTypeForVTK() const { return 5; }
virtual const char *getStringForPOS() const { return "ST"; }
virtual const char *getStringForBDF() const { return "CTRIA3"; }
virtual const char *getStringForDIFF() const { return "ElmT3n2D"; }
virtual void revert()
{
    MVertex *tmp = _v[1]; _v[1] = _v[2]; _v[2] = tmp;
}
virtual const functionSpace* getFunctionSpace(int o=-1) const;
virtual bool isInside(double u, double v, double w)
{
    double tol = _isInsideTolerance;
    if(u < (-tol) || v < (-tol) || u > ((1. + tol) - v))
        return false;
    return true;
}
virtual void getIntegrationPoints(int pOrder, int *npts, IntPt **pts) const;
virtual SPoint3 circumcenter();
private:
int edges_tri(const int edge, const int vert) const
{
    static const int e[3][2] = {
        {0, 1},
        {1, 2},
        {2, 0}
    };
    return e[edge][vert];
}
};

/*
 * MTriangle6
 *
 * 2
 * | \
 * |  \
 * 5   `4
 * |   \
 * |   \
 * 0-----3-----1
 *
 */

```

```

class MTriangle6 : public MTriangle {
protected:
    MVertex *_vs[3];
public :
    MTriangle6(MVertex *v0, MVertex *v1, MVertex *v2, MVertex *v3, MVertex *v4,
        MVertex *v5, int num=0, int part=0)
        : MTriangle(v0, v1, v2, num, part)
    {
        _vs[0] = v3; _vs[1] = v4; _vs[2] = v5;
        for(int i = 0; i < 3; i++) _vs[i]->setPolynomialOrder(2);
    }
    MTriangle6(std::vector<MVertex*> &v, int num=0, int part=0)
        : MTriangle(v, num, part)
    {
        for(int i = 0; i < 3; i++) _vs[i] = v[3 + i];
        for(int i = 0; i < 3; i++) _vs[i]->setPolynomialOrder(2);
    }
    ~MTriangle6(){}
    virtual int getPolynomialOrder() const { return 2; }
    virtual int getNumVertices() const { return 6; }
    virtual MVertex *getVertex(int num){ return num < 3 ? _v[num] : _vs[num - 3]; }
    virtual MVertex *getVertexUNV(int num)
    {
        static const int map[6] = {0, 3, 1, 4, 2, 5};
        return getVertex(map[num]);
    }
    virtual MVertex *getVertexMED(int num)
    {
        static const int map[6] = {0, 2, 1, 5, 4, 3};
        return getVertex(map[num]);
    }
    virtual int getNumEdgeVertices() const { return 3; }
    virtual int getNumEdgesRep();
    virtual void getEdgeRep(int num, double *x, double *y, double *z, SVector3 *n);
    virtual void getEdgeVertices(const int num, std::vector<MVertex*> &v) const
    {
        v.resize(3);
        MTriangle::_getEdgeVertices(num, v);
        v[2] = _vs[num];
    }
    virtual int getNumFacesRep();
    virtual void getFaceRep(int num, double *x, double *y, double *z, SVector3 *n);
    virtual void getFaceVertices(const int num, std::vector<MVertex*> &v) const
    {
        v.resize(6);
        MTriangle::_getFaceVertices(v);
        v[3] = _vs[0];
        v[4] = _vs[1];
        v[5] = _vs[2];
    }
    virtual int getTypeForMSH() const { return MSH_TRI_6; }
    virtual int getTypeForUNV() const { return 92; } // thin shell parabolic triangle

```

```

//virtual int getTypeForVTK() const { return 22; }
virtual const char *getStringForPOS() const { return "ST2"; }
virtual const char *getStringForBDF() const { return "CTRIA6"; }
virtual const char *getStringForDIFF() const { return "ElmT6n2D"; }
virtual void revert()
{
    MVertex *tmp;
    tmp = _v[1]; _v[1] = _v[2]; _v[2] = tmp;
    tmp = _vs[0]; _vs[0] = _vs[2]; _vs[2] = tmp;
}
};

/*
 * MTriangleN  FIXME: check the plot
 *
 * 2
 * | \      E = order - 1;
 * |  \     N = total number of vertices
 * 3+2E  2+2E
 * |  \     Interior vertex numbers
 * ...   ...   for edge 0 <= i <= 2: 3+i*E to 2+(i+1)*E
 * |  \     in volume      : 3+3*E to N-1
 * 2+3E    3+E
 * | 3+3E to N-1 \
 * |  \
 * 0---3---...---2+E---1
 *
 */
class MTriangleN : public MTriangle {
protected:
    std::vector<MVertex *> _vs;
    const char _order;
public:
    MTriangleN(MVertex *v0, MVertex *v1, MVertex *v2,
               std::vector<MVertex *> &v, char order, int num=0, int part=0)
        : MTriangle(v0, v1, v2, num, part), _vs(v), _order(order)
    {
        for(unsigned int i = 0; i < _vs.size(); i++) _vs[i]->setPolynomialOrder(_order);
    }
    MTriangleN(std::vector<MVertex *> &v, char order, int num=0, int part=0)
        : MTriangle(v[0], v[1], v[2], num, part), _order(order)
    {
        for(unsigned int i = 3; i < v.size(); i++) _vs.push_back(v[i]);
        for(unsigned int i = 0; i < _vs.size(); i++) _vs[i]->setPolynomialOrder(_order);
    }
    ~MTriangleN(){}
    virtual int getPolynomialOrder() const { return _order; }
    virtual int getNumVertices() const { return 3 + _vs.size(); }
    virtual MVertex *getVertex(int num){ return num < 3 ? _v[num] : _vs[num - 3]; }
    virtual int getNumFaceVertices() const
    {
        if(_order == 3 && _vs.size() == 6) return 0;

```

```

    if(_order == 3 && _vs.size() == 7) return 1;
    if(_order == 4 && _vs.size() == 9) return 0;
    if(_order == 4 && _vs.size() == 12) return 3;
    if(_order == 5 && _vs.size() == 12) return 0;
    if(_order == 5 && _vs.size() == 18) return 6;
    return 0;
}
virtual int getNumEdgeVertices() const { return 3 * (_order - 1); }
virtual int getNumEdgesRep();
virtual int getNumFacesRep();
virtual void getEdgeRep(int num, double *x, double *y, double *z, SVector3 *n);
virtual void getEdgeVertices(const int num, std::vector<MVertex*> &v) const
{
    v.resize(_order + 1);
    MTriangle::_getEdgeVertices(num, v);
    int j = 2;
    const int ie = (num + 1) * (_order - 1);
    for(int i = num * (_order - 1); i != ie; ++i) v[j++] = _vs[i];
}
virtual void getFaceRep(int num, double *x, double *y, double *z, SVector3 *n);
virtual void getFaceVertices(const int num, std::vector<MVertex*> &v) const
{
    v.resize(3 + _vs.size());
    MTriangle::_getFaceVertices(v);
    for(unsigned int i = 0; i != _vs.size(); ++i) v[i + 3] = _vs[i];
}
virtual int getTypeForMSH() const
{
    if(_order == 2 && _vs.size() == 3) return MSH_TRI_6;
    if(_order == 3 && _vs.size() == 6) return MSH_TRI_9;
    if(_order == 3 && _vs.size() == 7) return MSH_TRI_10;
    if(_order == 4 && _vs.size() == 9) return MSH_TRI_12;
    if(_order == 4 && _vs.size() == 12) return MSH_TRI_15;
    if(_order == 5 && _vs.size() == 12) return MSH_TRI_15I;
    if(_order == 5 && _vs.size() == 18) return MSH_TRI_21;
    return 0;
}
virtual void revert()
{
    MVertex *tmp;
    tmp = _v[1]; _v[1] = _v[2]; _v[2] = tmp;
    std::vector<MVertex*> inv;
    inv.insert(inv.begin(), _vs.rbegin(), _vs.rend());
    _vs = inv;
}
};
template <class T>
void sort3(T *t[3])
{
    T *temp;
    if(t[0] > t[1]){
        temp = t[1];

```



```

    t[1] = t[0];
    t[0] = temp;
}
if(t[1] > t[2]){
    temp = t[2];
    t[2] = t[1];
    t[1] = temp;
}
if(t[0] > t[1]){
    temp = t[1];
    t[1] = t[0];
    t[0] = temp;
}
}
struct compareMTriangleLexicographic
{
    bool operator () (MTriangle *t1, MTriangle *t2) const
    {
        MVertex *_v1[3] = {t1->getVertex(0), t1->getVertex(1), t1->getVertex(2)};
        MVertex *_v2[3] = {t2->getVertex(0), t2->getVertex(1), t2->getVertex(2)};
        sort3(_v1);
        sort3(_v2);
        if(_v1[0] < _v2[0]) return true;
        if(_v1[0] > _v2[0]) return false;
        if(_v1[1] < _v2[1]) return true;
        if(_v1[1] > _v2[1]) return false;
        if(_v1[2] < _v2[2]) return true;
        return false;
    }
};
#endif
MTriangle.cpp
#include "MTriangle.h"
#include "Numeric.h"
#include "Context.h"
#include "qualityMeasures.h"
#define SQU(a) ((a)*(a))
SPoint3 MTriangle::circumcenter()
{
    double p1[3] = {_v[0]->x(), _v[0]->y(), _v[0]->z()};
    double p2[3] = {_v[1]->x(), _v[1]->y(), _v[1]->z()};
    double p3[3] = {_v[2]->x(), _v[2]->y(), _v[2]->z()};
    double res[3];
    circumCenterXYZ(p1, p2, p3, res);
    return SPoint3(res[0], res[1], res[2]);
}
double MTriangle::distoShapeMeasure()
{
    return qmDistorsionOfMapping(this);
}

double MTriangle::gammaShapeMeasure()

```

```

{
    return qmTriangle(this, QMTRI_RHO);
}
const functionSpace* MTriangle::getFunctionSpace(int o) const
{
    int order = (o == -1) ? getPolynomialOrder() : o;
    int nf = getNumFaceVertices();
    if ((nf == 0) && (o == -1)) {
        switch (order) {
            case 1: return &functionSpaces::find(MSH_TRI_3);
            case 2: return &functionSpaces::find(MSH_TRI_6);
            case 3: return &functionSpaces::find(MSH_TRI_9);
            case 4: return &functionSpaces::find(MSH_TRI_12);
            case 5: return &functionSpaces::find(MSH_TRI_15I);
            default: Msg::Error("Order %d triangle function space not implemented", order);
        }
    }
    else {
        switch (order) {
            case 1: return &functionSpaces::find(MSH_TRI_3);
            case 2: return &functionSpaces::find(MSH_TRI_6);
            case 3: return &functionSpaces::find(MSH_TRI_10);
            case 4: return &functionSpaces::find(MSH_TRI_15);
            case 5: return &functionSpaces::find(MSH_TRI_21);
            default: Msg::Error("Order %d triangle function space not implemented", order);
        }
    }
    return 0;
}
int MTriangleN::getNumEdgesRep(){ return 3 * CTX::instance()->mesh.numSubEdges; }
int MTriangle6::getNumEdgesRep(){ return 3 * CTX::instance()->mesh.numSubEdges; }

static void _myGetEdgeRep(MTriangle *t, int num, double *x, double *y, double *z,
                          SVector3 *n, int numSubEdges)
{
    n[0] = n[1] = n[2] = t->getFace(0).normal();

    if (num < numSubEdges){
        SPoint3 pnt1, pnt2;
        t->pnt((double)num / numSubEdges, 0., 0., pnt1);
        t->pnt((double)(num + 1) / numSubEdges, 0., 0, pnt2);
        x[0] = pnt1.x(); x[1] = pnt2.x();
        y[0] = pnt1.y(); y[1] = pnt2.y();
        z[0] = pnt1.z(); z[1] = pnt2.z();
        return;
    }
    if (num < 2 * numSubEdges){
        SPoint3 pnt1, pnt2;
        num -= numSubEdges;
        t->pnt(1. - (double)num / numSubEdges, (double)num / numSubEdges, 0, pnt1);
        t->pnt(1. - (double)(num + 1) / numSubEdges, (double)(num + 1) / numSubEdges, 0, pnt2);
        x[0] = pnt1.x(); x[1] = pnt2.x();
    }
}

```

```

    y[0] = pnt1.y(); y[1] = pnt2.y();
    z[0] = pnt1.z(); z[1] = pnt2.z();
    return ;
}
{
    SPoint3 pnt1, pnt2;
    num -= 2 * numSubEdges;
    t->pnt(0, (double)num / numSubEdges, 0, pnt1);
    t->pnt(0, (double)(num + 1) / numSubEdges, 0, pnt2);
    x[0] = pnt1.x(); x[1] = pnt2.x();
    y[0] = pnt1.y(); y[1] = pnt2.y();
    z[0] = pnt1.z(); z[1] = pnt2.z();
}
}
void MTriangleN::getEdgeRep(int num, double *x, double *y, double *z, SVector3 *n)
{
    _myGetEdgeRep(this, num, x, y, z, n, CTX::instance()->mesh.numSubEdges);
}
void MTriangle6::getEdgeRep(int num, double *x, double *y, double *z, SVector3 *n)
{
    _myGetEdgeRep(this, num, x, y, z, n, CTX::instance()->mesh.numSubEdges);
}
int MTriangle6::getNumFacesRep(){ return SQU(CTX::instance()->mesh.numSubEdges); }
int MTriangleN::getNumFacesRep(){ return SQU(CTX::instance()->mesh.numSubEdges); }

static void _myGetFaceRep(MTriangle *t, int num, double *x, double *y, double *z,
                        SVector3 *n, int numSubEdges)
{
    // on the first layer, we have (numSubEdges-1) * 2 + 1 triangles
    // on the second layer, we have (numSubEdges-2) * 2 + 1 triangles
    // on the ith layer, we have (numSubEdges-1-i) * 2 + 1 triangles
    int ix = 0, iy = 0;
    int nbt = 0;
    for (int i = 0; i < numSubEdges; i++){
        int nbl = (numSubEdges - i - 1) * 2 + 1;
        nbt += nbl;
        if (nbt > num){
            iy = i;
            ix = nbl - (nbt - num);
            break;
        }
    }
}

const double d = 1. / numSubEdges;

SPoint3 pnt1, pnt2, pnt3;
double J1[3][3], J2[3][3], J3[3][3];
if (ix % 2 == 0){
    t->pnt(ix / 2 * d, iy * d, 0, pnt1);
    t->pnt((ix / 2 + 1) * d, iy * d, 0, pnt2);
    t->pnt(ix / 2 * d, (iy + 1) * d, 0, pnt3);
    t->getJacobian(ix / 2 * d, iy * d, 0, J1);
}

```

```

    t->getJacobian((ix / 2 + 1) * d, iy * d, 0, J2);
    t->getJacobian(ix / 2 * d, (iy + 1) * d, 0, J3);
}
else{
    t->pnt((ix / 2 + 1) * d, iy * d, 0, pnt1);
    t->pnt((ix / 2 + 1) * d, (iy + 1) * d, 0, pnt2);
    t->pnt(ix / 2 * d, (iy + 1) * d, 0, pnt3);
    t->getJacobian((ix / 2 + 1) * d, iy * d, 0, J1);
    t->getJacobian((ix / 2 + 1) * d, (iy + 1) * d, 0, J2);
    t->getJacobian(ix / 2 * d, (iy + 1) * d, 0, J3);
}
{
    SVector3 d1(J1[0][0], J1[0][1], J1[0][2]);
    SVector3 d2(J1[1][0], J1[1][1], J1[1][2]);
    n[0] = crossprod(d1, d2);
    n[0].normalize();
}
{
    SVector3 d1(J2[0][0], J2[0][1], J2[0][2]);
    SVector3 d2(J2[1][0], J2[1][1], J2[1][2]);
    n[1] = crossprod(d1, d2);
    n[1].normalize();
}
{
    SVector3 d1(J3[0][0], J3[0][1], J3[0][2]);
    SVector3 d2(J3[1][0], J3[1][1], J3[1][2]);
    n[2] = crossprod(d1, d2);
    n[2].normalize();
}
x[0] = pnt1.x(); x[1] = pnt2.x(); x[2] = pnt3.x();
y[0] = pnt1.y(); y[1] = pnt2.y(); y[2] = pnt3.y();
z[0] = pnt1.z(); z[1] = pnt2.z(); z[2] = pnt3.z();
}
void MTriangleN::getFaceRep(int num, double *x, double *y, double *z, SVector3 *n)
{
    _myGetFaceRep(this, num, x, y, z, n, CTX::instance()->mesh.numSubEdges);
}
void MTriangle6::getFaceRep(int num, double *x, double *y, double *z, SVector3 *n)
{
    _myGetFaceRep(this, num, x, y, z, n, CTX::instance()->mesh.numSubEdges);
}
void MTriangle::getIntegrationPoints(int pOrder, int *npts, IntPt **pts) const
{
    *npts = getNGQTPts(pOrder);
    *pts = getGQTPts(pOrder);
}

```