

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

Навчально-науковий інститут фізики, математики та інформаційних
технологій

Кафедра інформаційних технологій та систем

Цалапов Артем Євгенович

**РОЗРОБКА ДОДАТКА ДЛЯ АВТОМАТИЗАЦІЇ ПОБУДОВИ
ДИСКРЕТНОЇ (СКІНЧЕННО-ЕЛЕМЕНТНОЇ) МОДЕЛІ
КОНСТРУКЦІЙ**

кваліфікаційна робота

здобувача вищої освіти другого (магістерського) рівня

освітньої програми «Мультимедійні системи»

за спеціальністю 121 Інженерія програмного забезпечення

Особистий підпис _____ Артем ЦАЛАПОВ

Науковий керівник _____ Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій
та систем

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

Полтава – 2023

АНОТАЦІЯ

Цалапов А.Є.

Тема: Розробка додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

Спеціальність: 121 «Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2023р.

Магістерська робота містить: 61 с., 43 рис., 2 табл., 37 джерел.

Об'єкт дослідження – дискретизація геометричних областей на скінчені елементи.

Предмет дослідження – генерація розрахункових сіток для скінчено – елементного моделювання конструкцій.

Мета роботи – аналіз методів дискретизації геометричних областей на скінченні елементи заданої форми та розробка додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

Результати роботи – в роботі 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. Проведено моделювання та проаналізовано програмне забезпечення для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій. Розроблено UML- діаграми представлення статичної моделі структури, представлення моделі поведінки та фізичне представлення моделей програмної розробки. Розглянуто вибір середовища розробки додатку і мови програмування. Описано алгоритм генерації сіток. У додатку доступно створення 1- 2 - і 3-х мірних сіток, і їх оптимізація. Програма реалізується на мові програмування C++ в середовищі Visual C++ 2010. При створенні засобів відображення та візуалізації використовується графічний інтерфейс OpenGL.

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

ANNOTATION

Tsalapov Artem

Theme: Development of an application for automating the construction of a discrete (finite element) model of structures.

Speciality: 121 "Software Engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2023 year.

Master's work of: 61 p., 43 im, 37 sources.

A research object of: - discretization of geometric areas into finite elements.

The article of research- generation of calculation grids for finite element modeling of structures.

An aim of research is - analysis of methods of discretization of geometric areas into finite elements of a given shape and development of an application for automating the construction of a discrete (finite element) model of structures.

Job performanes.- the paper analyzes the main approaches used in modern CAD for solid-state modeling of geometric objects, as well as the main common methods and algorithms for the discretization of flat and spatial regions. The software for automating the construction of a discrete (finite element) model of structures was simulated and analyzed. UML diagrams of static structure model representation, behavior model representation and physical representation of software development models have been developed. The choice of application development environment and programming language is considered. The grid generation algorithm is described. Creation of 1-2- and 3-dimensional grids and their optimization is available in the application. The program is implemented in the C++ programming language in the Visual C++ 2010 environment. The OpenGL graphic interface is used to create display and visualization tools.

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

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	5
ВСТУП.....	6
РОЗДІЛ 1. ДИСКРЕТНІ (СКІНЧЕННО-ЕЛЕМЕНТНІ) МОДЕЛІ КОНТИНУАЛЬНИХ ОБЛАСТЕЙ	8
1.1. Методи геометричного моделювання складних об'єктів	9
1.1.1. Опис топології області	9
1.1.2. Дискретизація плоских областей	11
1.1.3. Дискретизація тривимірних областей	22
1.2. Об'єктна модель геометричної області	24
1.2.1. Модель дискретного представлення області	24
1.2.2. Дискретизація області на скінченні елементи	27
1.3. Висновки до розділу	29
РОЗДІЛ 2. МОДЕЛЮВАННЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДОДАТКА ДЛЯ АВТОМАТИЗАЦІЇ ПОБУДОВИ ДИСКРЕТНОЇ (СКІНЧЕННО-ЕЛЕМЕНТНОЇ) МОДЕЛІ КОНСТРУКЦІЙ	30
2.1. Логічне представлення статичної моделі структури програмної розробки	30
2.2. Логічне представлення моделі поведінки програмної розробки	32
2.3. Фізичне представлення моделі програмної розробки	38
2.4. Архітектура програмного забезпечення додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій	39
2.5. Висновки до розділу 2	40
РОЗДІЛ ІІІ. РОЗРОБКА ДОДАТКА ДЛЯ АВТОМАТИЗАЦІЇ ПОБУДОВИ ДИСКРЕТНОЇ (СКІНЧЕННО-ЕЛЕМЕНТНОЇ) МОДЕЛІ КОНСТРУКЦІЙ	41
3.1. Обґрунтування вибору середовища розробки	41
3.2. Розробка інтерфейсу	43
3.3. Висновки до розділу 3	55
ВИСНОВКИ	56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	58
ДОДАТОК.....	62

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

СЕ	Скінчений елемент
МСЕ	Метод скінчених елементів
НДС	Напружено-деформований стан
САПР	Система автоматизованого проектування
ІМ	Інформаційна модель
ООП	Об'єктно -орієнтований підхід
ППП	Пакет прикладних програм

ВСТУП

Більшість чисельних методів дослідження напружено-деформованого стану деформованого тіла базуються на ідеї переходу від континуальної задачі до дискретної, коли досліджувана суцільна область замінюється деякою скінченою дискретною моделлю. У методі скінчених елементів (МСЕ) неперервна область замінюється деякою сукупністю скінчених елементів, що заповнюють весь об'єм тіла.

Однією з головних проблем, що виникають при застосуванні МСЕ - це побудова дискретної моделі досліджуваної механічної системи. Однією з головних частин будь-якого програмного комплексу чисельного аналізу є програма, яка автоматизує побудову геометричної моделі досліджуваного об'єкта з подальшою її дискретизацією на скінчені елементи.

Проблема оптимальної дискретизації досліджуваної області на скінчені елементи в загальному вигляді є досить складною (особливо для тривимірних областей). Це обумовлено тим, що на форму скінчених елементів (СЕ) накладаються два основних обмеження: вони не повинні мати надто малих (або відповідно занадто великих) кутів і обсяг СЕ не повинен перевищувати деяку задану величину. У першому випадку при розрахунках виникають значні обчислювальні похибки. У другому з'являється ризик втрати точності обчислень при значній зміні градієнта досліджуваної функції.

Тому автоматична генерація СЕ- сітки являє собою досить складну процедуру, яка є основою будь-якого скінчено-елементного пакету програм і являється актуальною.

Об'єкт дослідження – дискретизація геометричних областей на скінчені елементи.

Предмет дослідження – генерація розрахункових сіток для скінчено – елементного моделювання конструкцій.

Мета роботи – аналіз методів дискретизації геометричних областей на скінченні елементи заданої форми та розробка додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

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

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- проаналізувати основні поширені методи й алгоритми дискретизації плоских та просторових областей;
- провести моделювання та аналіз програмного забезпечення додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій;
- розробити і реалізувати додаток для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

Практичною цінністю роботи є розроблений додаток для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

В першому розділі проаналізовано основні поширені методи й алгоритми дискретизації плоских та просторових областей. дискретизації області на скінченні елементи.

В другому розділі виконано аналіз процесу розробки додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій. Проведено моделювання та описано архітектру розробленого додатку.

У третьому розділі аргументується вибір середовища розробки додатку і мови програмування. Описано алгоритм генерації сіток. У додатку доступно створення 1- 2 - і 3-х мірних сіток, і їх оптимізація. Програма реалізується на мові програмування C++ в середовищі Visual C++ 2010. При створенні засобів відображення та візуалізації використовується графічний інтерфейс OpenGL.

РОЗДІЛ 1. ДИСКРЕТНІ (СКІНЧЕННО-ЕЛЕМЕНТНІ) МОДЕЛІ КОНТИНУАЛЬНИХ ОБЛАСТЕЙ

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. Методи геометричного моделювання складних об'єктів

1.1.1. 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:

- площина;
- 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 проектування 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:

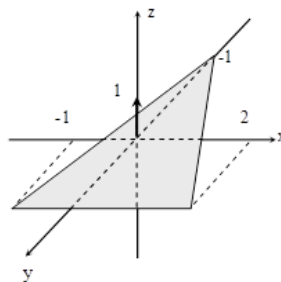


Рис.1.1. Опис трикутника в STL -форматі.

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

1.1.2. 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).

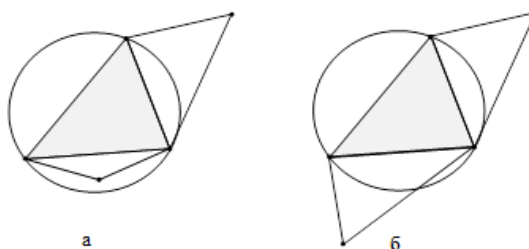


Рис. 1.2. Приклади тріангуляції (а) і тріангуляції Делоне (б).

Також базовим поняттям в тріангуляції 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 тріангуляції. 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.

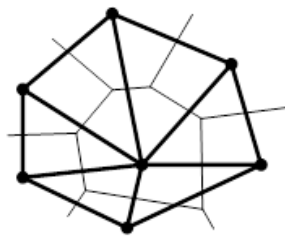


Рис. 1.3. Діаграма Воронова.

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. Триангуляційна процедура послідовно вставляє кожну наступну точку, починаючи з першої, у вже існуючу триангуляцію. Спочатку триангуляція Делоне представлена одним єдиним так званим супертрикутником, 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 супертрикутника 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 триангуляцію 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 показане, що цей ітераційний алгоритм повинен побудувати тріангуляцію Делоне і завершитися після останнього обміну. Практика показує, що для побудови Делоне-тріангуляції не потрібно велику кількість обмінів, тому цей процес є ефективним.

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

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

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 $14N$ елементів пам'яті, використовуваних для зберігання координат точок, номерів вузлів трикутників і іншої допоміжної інформації [13].

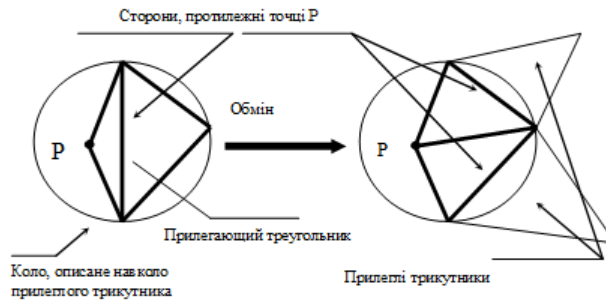


Рис. 1.4. Обмінний алгоритм Лавсона.

Загальна блок-схема комбінованого алгоритму Ватсона-Лавсона приведена на рис. 1.5. Для його застосування в першу чергу необхідно задати межу області, що підлягає тріангуляції, а потім опорні точки, на яких і буде побудована тріангуляція. 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 трудностю, 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).

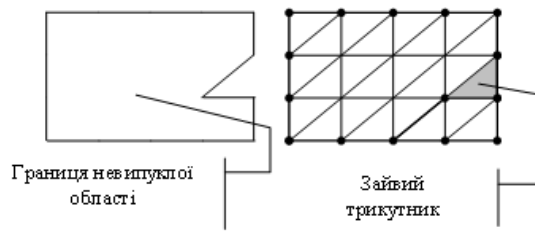


Рис. 1.6. Проблема зайвого трикутника.

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);

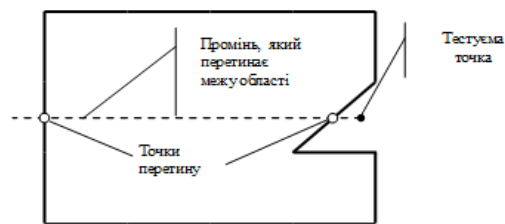


Рис. 1.7. Визначення належності точки замкнутому контуру шляхом підрахунку кількості перетинів променя, проведеного з тестованої точки, і межі області.

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

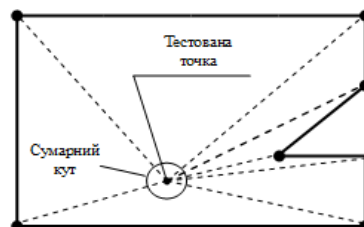


Рис. 1.8. Визначення належності точки замкнутому контуру шляхом підрахунку суми кутів.

У першому випадку, якщо точка знаходиться усередині області, то кількість перетинів променя має бути непарною. А в другому - якщо точка знаходиться усередині контуру, сумарний кут повинен

дорівнювати 360^0 якщо на межі -180^0 . Інакше точка знаходиться за межами області.

Проблема побудови безлічі опорних точок для триангуляції Делоне в загальному вигляді є досить складною, оскільки необхідно враховувати як обмеження, що накладаються на форму трикутників, так і можливу наявність в геометрії триангульованої 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 триангуляції 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).

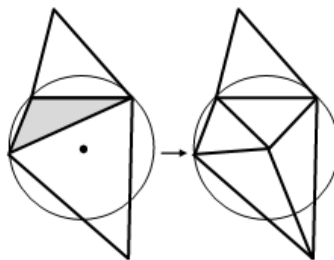


Рис. 1.9. Розбиття «неправильного» трикутника.

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.

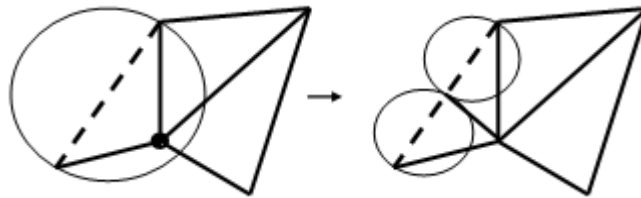


Рис. 1.10. Розбиття «неправильного» сегменту.

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 елементу. 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 напруги.

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.

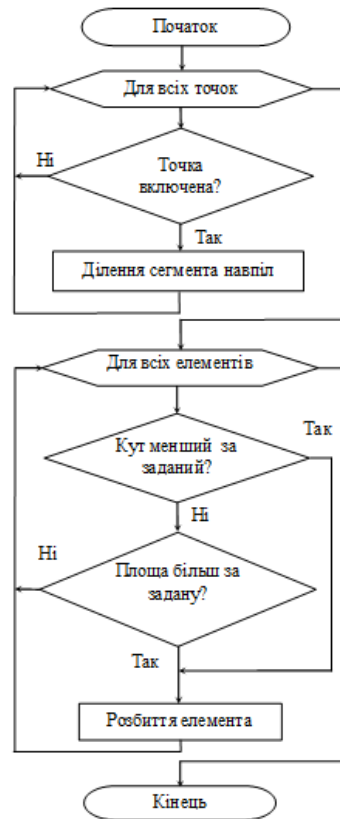


Рис. 1.11. Блок-схема алгоритму Рапперта.

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.

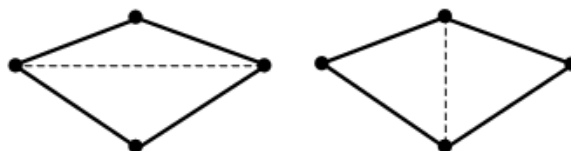


Рис. 1.12. Процедура повороту діагоналі.

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 .

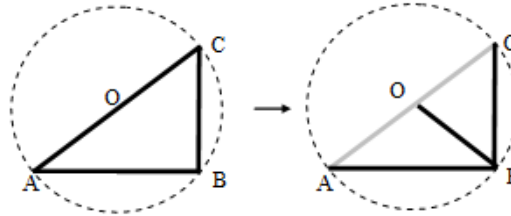


Рис. 1.13. Утворення виродженого трикутника.

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).

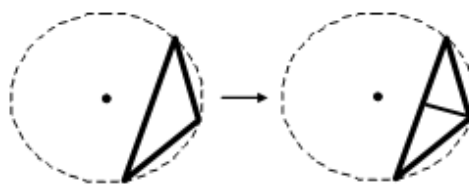


Рис. 1.14. Розбиття витягнутого трикутника.

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.

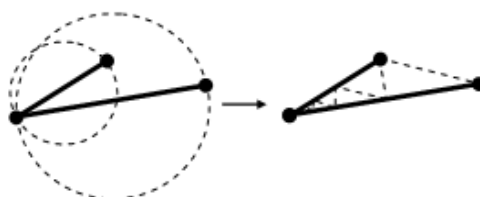


Рис. 1.15. Зациклення процедури ділення сегменту при гострому вуглі.

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.

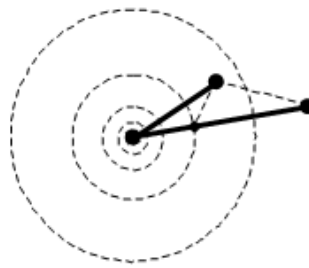


Рис. 1.16. Ділення сегменту способом концентричних кіл.

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).

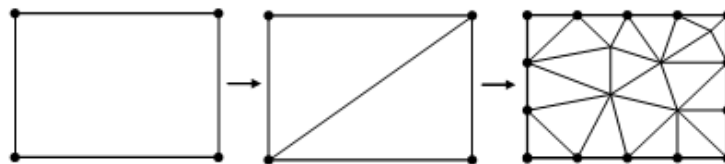


Рис. 1.17. Схема отримання початкового розбиття з подальшою триангуляцією.

1.1.3. 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 двовимірному випадку, усі вершини тріангульованої 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 тріангуляцією 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 тріангуляцію 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].

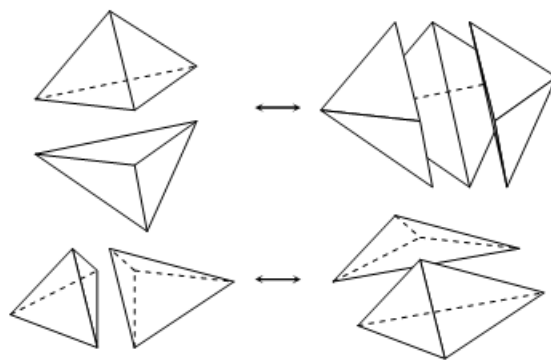


Рис. 1.18. Приклади обмінної процедури в тривимірному випадку.

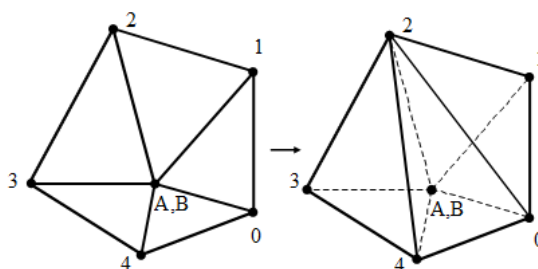


Рис. 1.19. Приклад обмінної процедури для ребер.

1.2. Об'єктна модель геометричної області

1.2.1. Модель дискретного представлення області

Після відпрацювання процедури дискретизації плоскої або тривимірної області отримане дискретне представлення геометричної області описується об'єктною моделлю, приведеною на рис. 1.20. Вона містить інформацію про загальну кількість вузлів у розбитті області на СЕ, кількості граничних вузлів, ширині стрічки, використовуваної надалі для стрічкових методів рішення СЛАР, а також інформацію про координати вузлів і зв'язків в СЕ [14].

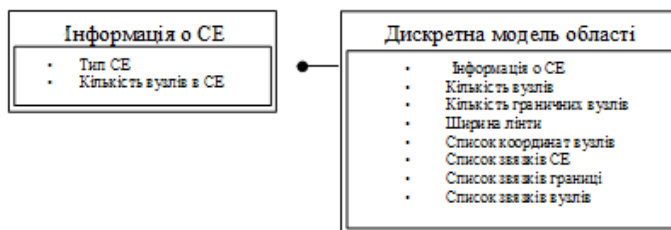


Рис. 1.20. Об'єктна модель дискретного представлення геометричної області.

Крім того, міститься додаткова інформація про графу зв'язків між вузлами області, також використовувана надалі для отримання компактного представлення матриці жорсткості.

Для генерації скінченно-елементного розбиття використовується алгоритм, що є комбінацією алгоритмів Ватсона-Лавсона і Рапперта-Шевчука. Основні етапи роботи цього алгоритму побудови кінцево-елементної моделі області наступні:

а) генерація кінцево-елементного розбиття, що включає :

- побудова точкової (каркасної) моделі області;
- початкову («грубу») дискретизацію області на СЕ;
- додавання нових вузлів в розбиття;

б) оптимізація отриманого розбиття, що складається з :

- оптимізації мінімальних кутів елементів (обмінна процедура);
- контролю максимальної площі або об'єму.

Процес дискретизації геометричної області розпочинається з побудови її точкової (каркасної) моделі. У цій моделі об'єкт представляється як сукупність вузлів, що лежать на його межі або поверхні і зв'язків між ними. На рис. 1.21 показана схема переходу від об'єкту до його каркасної моделі, а потім і до дискретної.

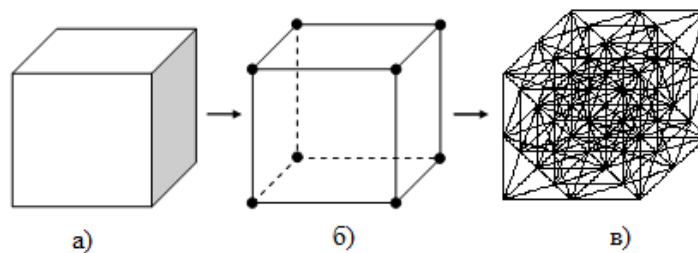


Рис. 1.21. Етапи дискретизації області : а) об'єкт; б) каркасна модель; в) дискретна модель.

У плоскому випадку побудова каркасної моделі розпочинається з процесу «інтерполяції» - заміни криволінійних граничних сегментів ламаними прямими (рис. 1.22). При цьому обчислюється кількість необхідних вузлів так, щоб забезпечити їх мінімальну кількість при заданому параметрі розміру елементу.



Рис. 1.22. Дискретизація криволінійного граничного сегменту.

При заміні криволінійного сегменту ламаною прямою шляхом вставки нових вузлів важливо контролювати довжину сегментів ламаної, щоб не втратити точність при моделюванні граничних особливостей топології початкової області. При цьому чим більше кривизна криволінійної граничної ділянки, тим більше вимагається для його інтерполяції сегментів ламаної.

Для реалізації процедури «інтерполяції» криволінійних граничних сегментів потрібна наявність алгоритму визначення координат чергової точки, що вставляється на межу. Одним з можливих варіантів реалізації такого алгоритму є обчислення кута між дотичною до граничного сегменту і прямої, що містить передбачуваний сегмент ламаної, що починається у вже наявній точці і закінчується у вузлі (рис. 1.23), що знову вставляється. При цьому величина кута (обчислюється на початку роботи препроцесора і залежить від геометрії початкової області і параметра, що визначає максимальну величину елементів. Обчислити цей кут можна, знаючи величину розміру місцевої особливості $lfs(localfeaturesize)$ в початковій точці $p(x_0, y_0)$, визначеною Раппертом як радіус мінімального кола з центром в точці p , яка торкається двох сегментів межі області, що не перетинаються. Для кожного типу криволінійних сегментів межі плоскої області обчислення lfs не представляє складності [10].

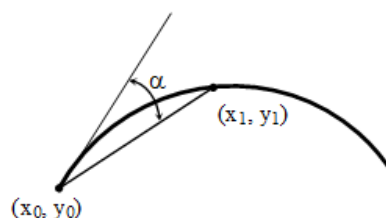


Рис. 1.23. Обчислення координат вузла, що вставляється на граничний сегмент.

Недоліком цього алгоритму є те, що при побудові точкових моделей областей з граничними сегментами, що мають малий радіус кривизни,

можливі значні спотворення його топології, що виникають за рахунок того, що за умовчанням розмір елементів приймається максимально можливим (рис 1.24).

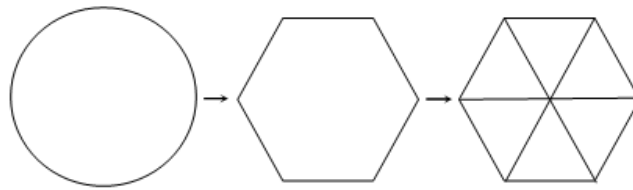


Рис. 1.24. Спотворення геометрії області при великих значеннях кута α .

Цю проблему можна розв'язати тільки за рахунок згущування сітки, тобто зменшення значення кута α .

Таким чином, блок-схему алгоритму побудови каркасної моделі плоскої геометричної області можна зображувати таким чином (рис. 1.25).

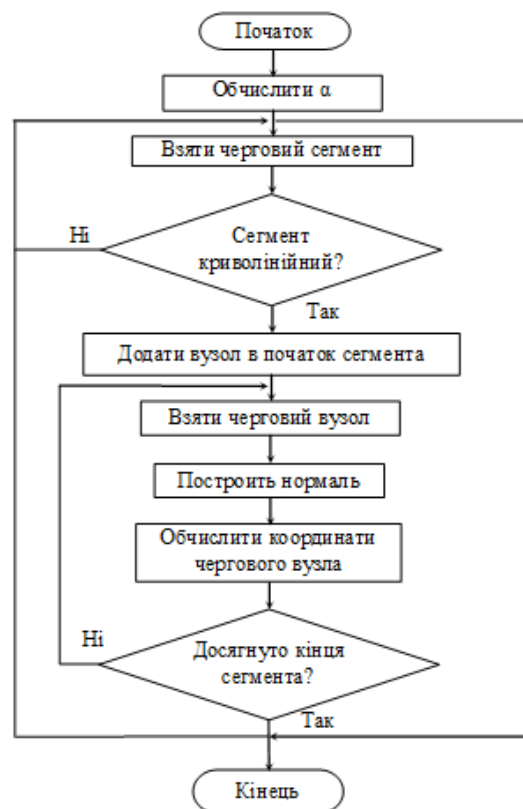


Рис. 1.25. Блок-схема алгоритму початкової дискретизації межі області.

1.2.2. Дискретизація області на скінченні елементи

Як вже відзначалося, в якості вхідної інформації алгоритму Рапперта потрібно початкове «грубе» розбиття області на елементи. Для попереднього

розбиття плоскої або об'ємної геометричній області використовується вищеописаний модифікований алгоритм Ватсона-Лавсона.

Цьому алгоритму в якості вхідної інформації потрібний набір вузлових точок, на базі яких і буде побудована початкова тріангуляція або тетраедризація. В якості такого набору точок безпосередньо використовуються вузли, що утворюють каркасну модель. Для оптимізації початкового розбиття можна використовувати модифікований алгоритм Рапперта-Шевчука.

Визначення вузлів, що належать межі плоскої або поверхні об'ємної області, відбувається в процесі роботи алгоритму оптимізації початкового розбиття і додавання нових вузлів в дискретизацію. Проте на практиці часто виникає необхідність визначення граничних вузлів і сегментів для вже існуючого розбиття (наприклад, при використанні геометрії, побудованої іншим препроцесором). В цьому випадку виділити поверхневі грані (у тривимірному випадку) можна виходячи з того міркування, що внутрішня грань завжди належить двом СЕ (рис. 1.26). У плоскому випадку побудувати межу можна виходячи з аналогічних міркувань.

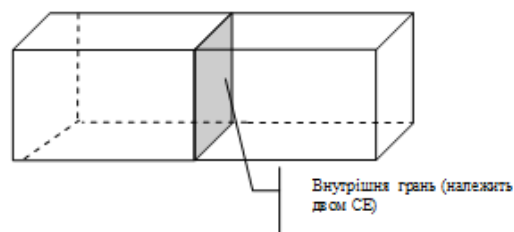


Рис. 1.26. Схема виділення поверхневих граней СЕ-об'єкту.

Багато вживаних спільно з МСЕ методи рішення СЛАР вимагають наявності додаткової інформації про зв'язки між вузлами кінцево-елементного розбиття. Така інформація дозволяє оптимізувати потрібний для зберігання коефіцієнтів матриці жорсткості об'єм оперативної пам'яті ЕОМ.

Для побудови списку зв'язків кожного вузла сітки з іншими на етапі дискретизації області необхідно побудувати модель розбиття у вигляді графа, а потім її досліджувати.

1.3. Висновки до розділу

В цьому розділі 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 додатку автоматичної генерації розрахункових сіток для скінченно- елементного моделювання конструкцій при первинній дискретизації області на скінченні елементи прийнято застосувати модифікований алгоритм Ватсона-Лавсона. Оптимізацію отриманої скінченно-елементної сітки робити шляхом застосування алгоритму Рапперта в плоскому випадку та Шевчука – в тривимірному.

РОЗДІЛ 2. МОДЕЛЮВАННЯ ТА АНАЛІЗ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДОДАТКА ДЛЯ АВТОМАТИЗАЦІЇ ПОБУДОВИ ДИСКРЕТНОЇ (СКІНЧЕННО-ЕЛЕМЕНТНОЇ) МОДЕЛІ КОНСТРУКЦІЙ

Підходом до проектування 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 проектуванні об'єктів як компонентів.

Проектування 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 проект 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 сіткового генератора.

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.

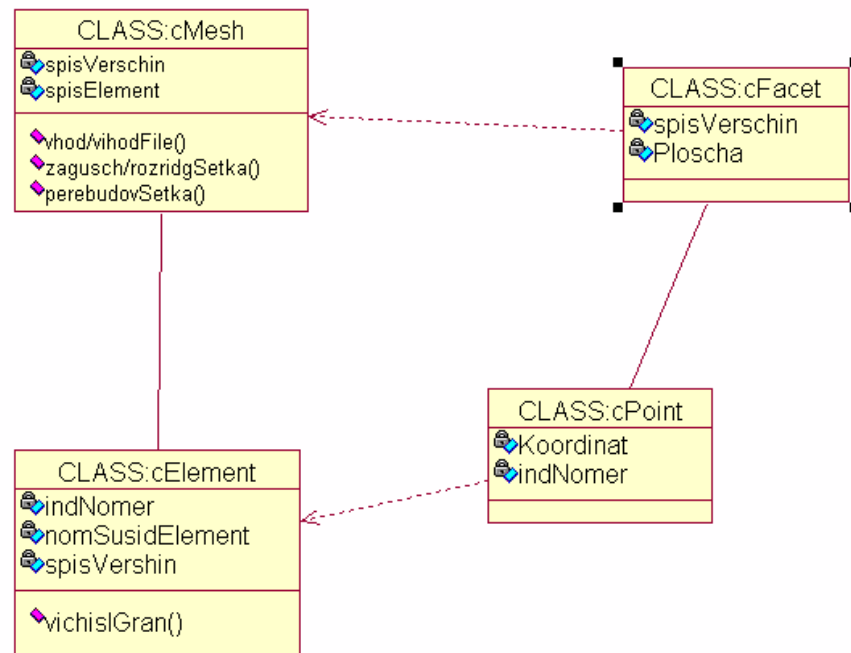


Рис.2.1. Діаграма класів сіткового генератора

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.2.2.) 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.2.2.)

Коротке описання:

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»

Коротке описання :

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»

Коротке описання:

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»

Коротке описання:

Даний варіант використання описує запуск програми для дискретизації.

Основні події:

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

Варіант використання «Вибір фігури»

Коротке описання:

Даний варіант використання описує вибір оператором фігури.

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

Варіант використання «Встановити масштаб»

Коротке описання:

Даний варіант використання описує завдання оператором масштабу фігури.

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

Варіант використання «Задати тип дискретизації»

Коротке описання:

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»

Коротке описання:

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»

Коротке описання:

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»

Коротке описання:

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»

Коротке описання:

Варіант використання описує процес збереження побудованої моделі.

1. Програмний комплекс отримує запит на збереження моделі.

2. Програма зберігає модель по заданому шляху.

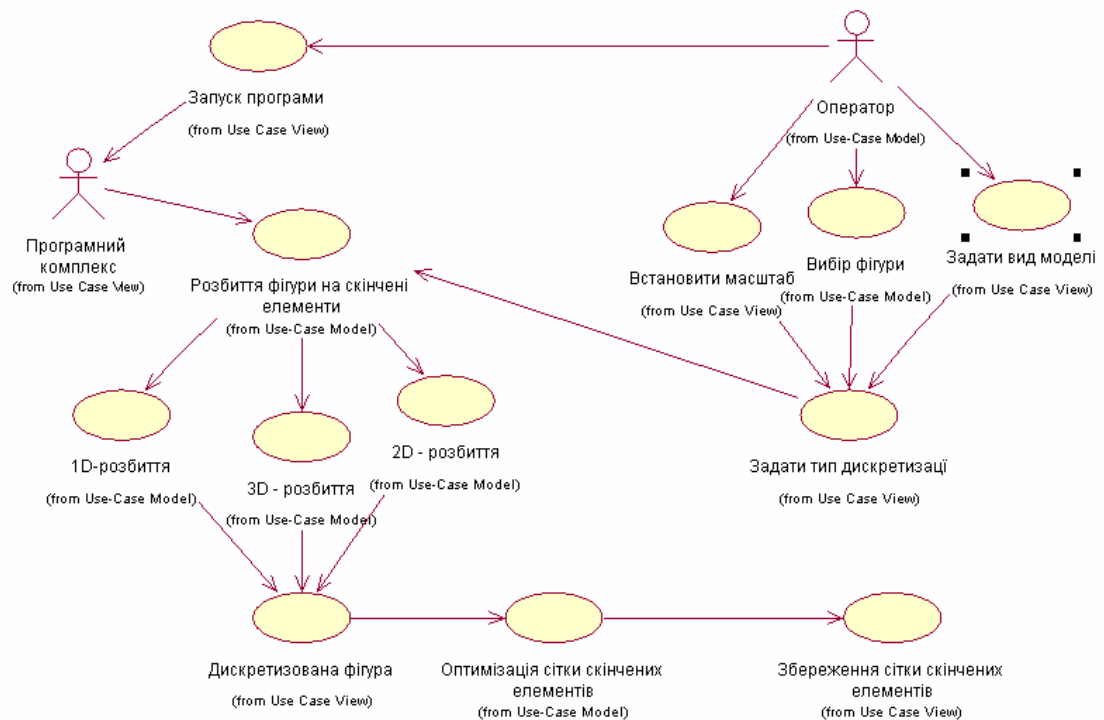


Рис.2.2. Use Case Diagram «Дискретизація фігури»

Діаграма послідовності дій (Рис.2.3.) 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.

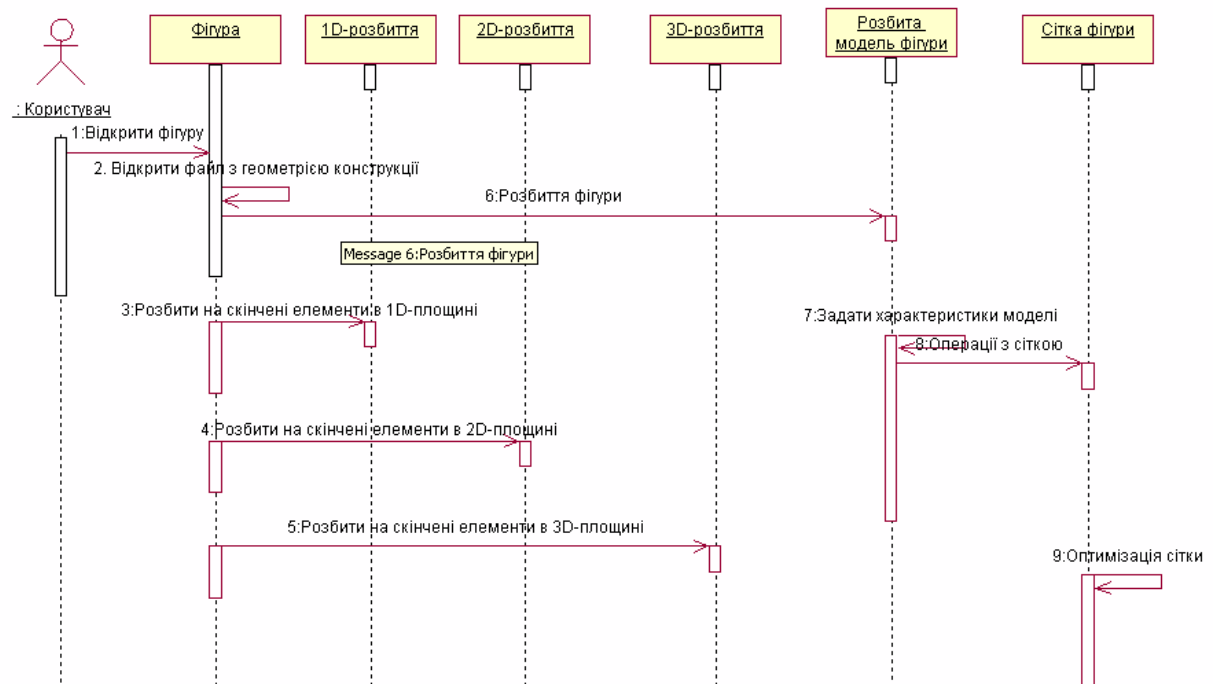


Рис.2.3. Діаграма послідовності дій

Кооперативна діаграма відображає потік подій через сценарій варіанта використання. Діаграма (Рис.2.4.) загострює увагу на зв'язках між об'єктами.

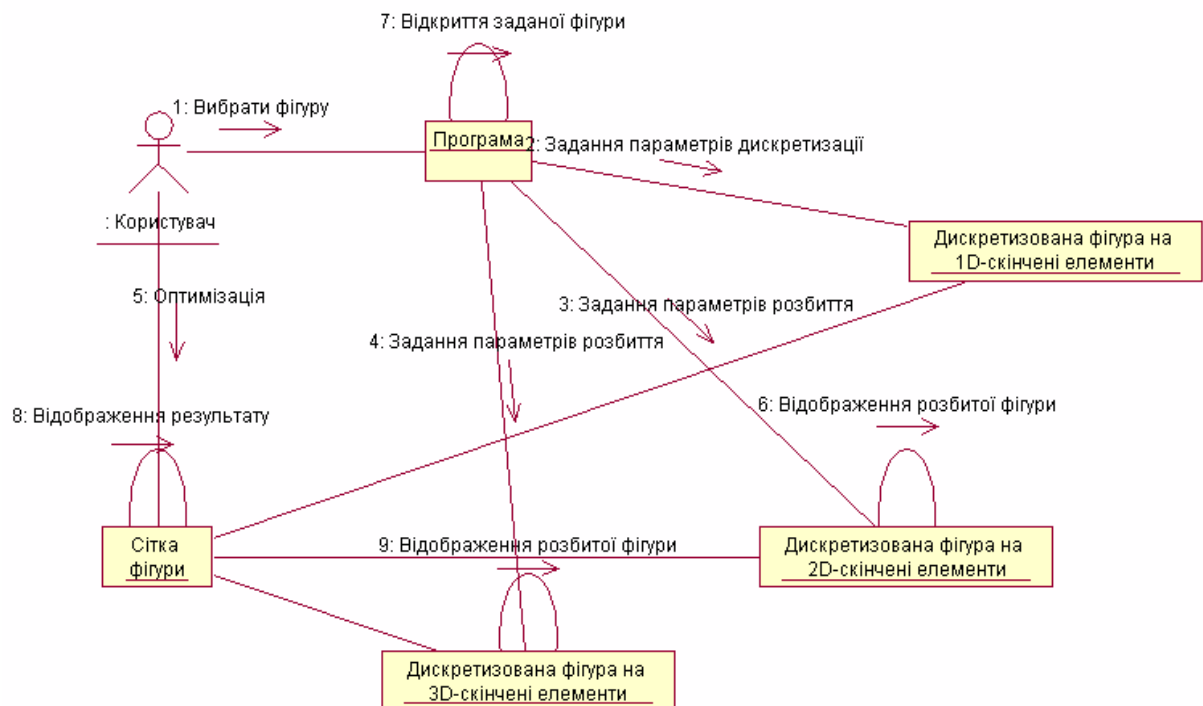


Рис.2.4. Діаграма кооперації

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.

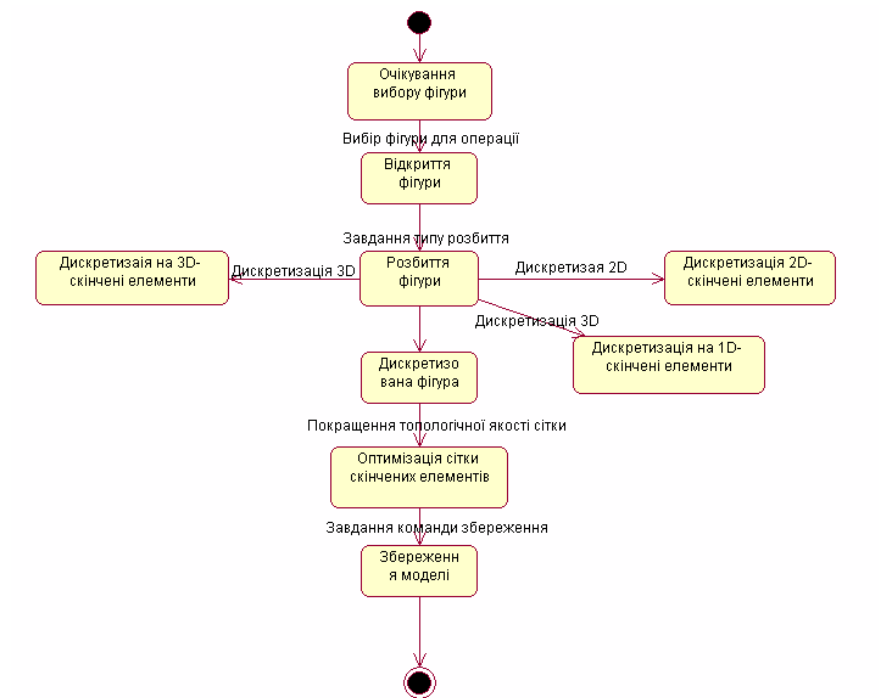


Рис.2.5. Діаграма стану (Statechart diagrams)

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

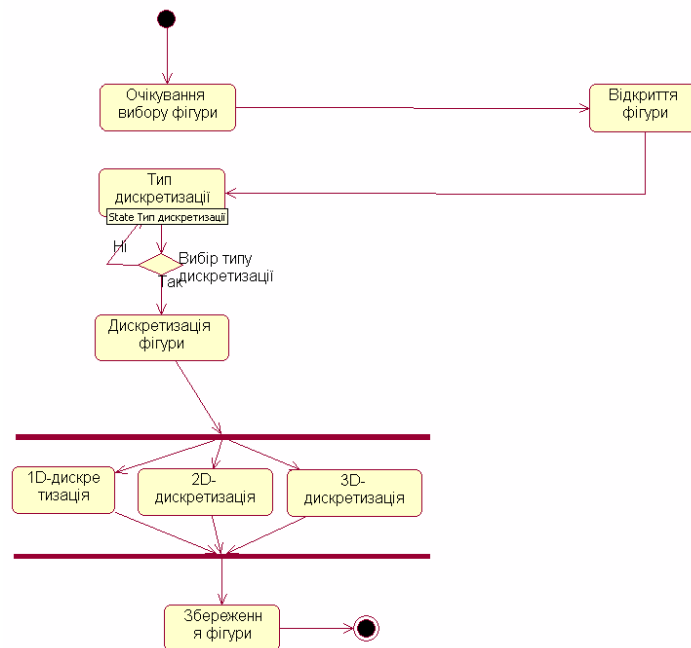


Рис.2.5. Діаграма діяльності (Activity diagrams)

2.4. Архітектура програмного забезпечення додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій

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 для автоматичної генерації розрахункових сіток для скінчено – елементного моделювання конструкцій. Розроблена схема (Рис.2.6.) 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.

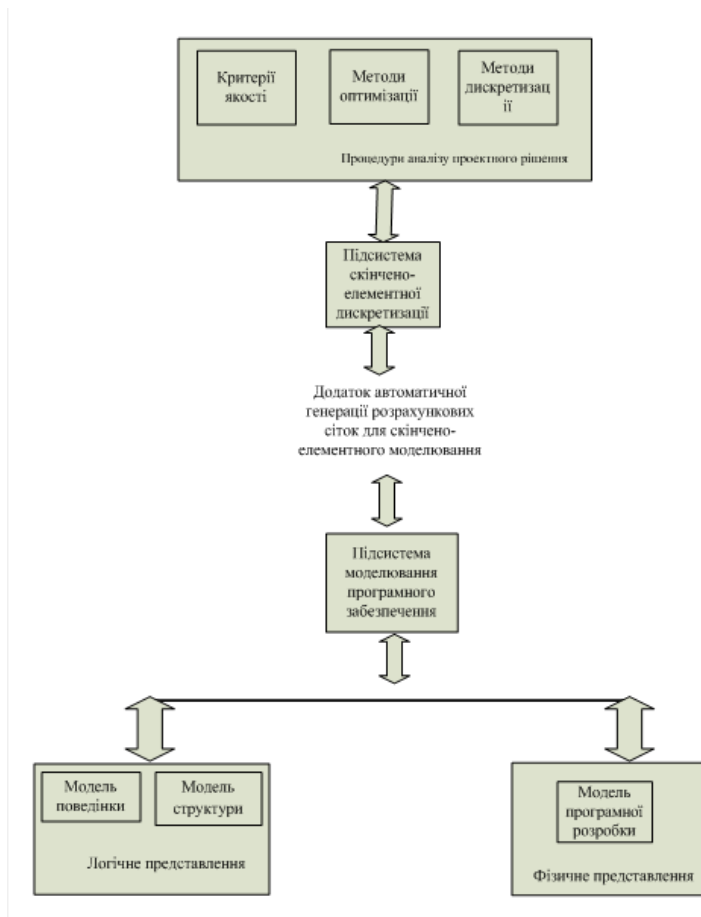


Рис.2.6. Схема компонентів програмного комплексу

2.5. Висновки до розділу 2

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 (скінченно-елементної) моделі конструкцій.

РОЗДІЛ III. РОЗРОБКА ДОДАТКА ДЛЯ АВТОМАТИЗАЦІЇ ПОБУДОВИ ДИСКРЕТНОЇ (СКІНЧЕННО-ЕЛЕМЕНТНОЇ) МОДЕЛІ КОНСТРУКЦІЙ

3.1. Обґрунтування вибору середовища розробки

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

Visual C++ 2010 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 [11]. Visual C++ складається з наступних компонентів:

- Засоби компілятора Visual C++ 2010. Компілятор підтримує як традиційну розробку з використанням машинного коду, так і розробку з використанням платформ віртуальних машин, таких як середовище CLR. Visual C++ 2010 містить компілятори для цільового об'єкту x64 і Itanium. Компілятор продовжує безпосередньо підтримувати архітектуру x86 і оптимізує продуктивність коду для обох платформ.
- Бібліотеки Visual C++. Містять загальновизнану бібліотеку шаблонних класів (ATL), бібліотеки Microsoft Foundation Class (MFC) і стандартні бібліотеки, такі як стандартна бібліотека C++, яка складається з бібліотеки iostreams, бібліотеки стандартних шаблонів (STL) і бібліотеки часу виконання мови C (CRT). Бібліотека CRT включає альтернативні функції з поліпшеною безпекою для функцій з відомими проблемами безпеки. Бібліотека STL/CLR дозволяє розробникам, що використовують керований код, використовувати також і можливості бібліотеки STL. Бібліотека підтримки C++ надає нові можливості для маршалінгу

даних і спрощує написання програм, що використовують середовище CLR.

- Середовище розробки Visual C++. Середовище розробки надає всебічну підтримку при управлінні проектами та їх налаштуванні (включаючи поліпшену підтримку великих проектів), редагуванні початкового коду, перегляді початкового коду, а також потужні засоби відладки. Середовище розробки також підтримує технологію IntelliSense, яка надає при написанні коду детальні підказки, що враховують контекст.

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

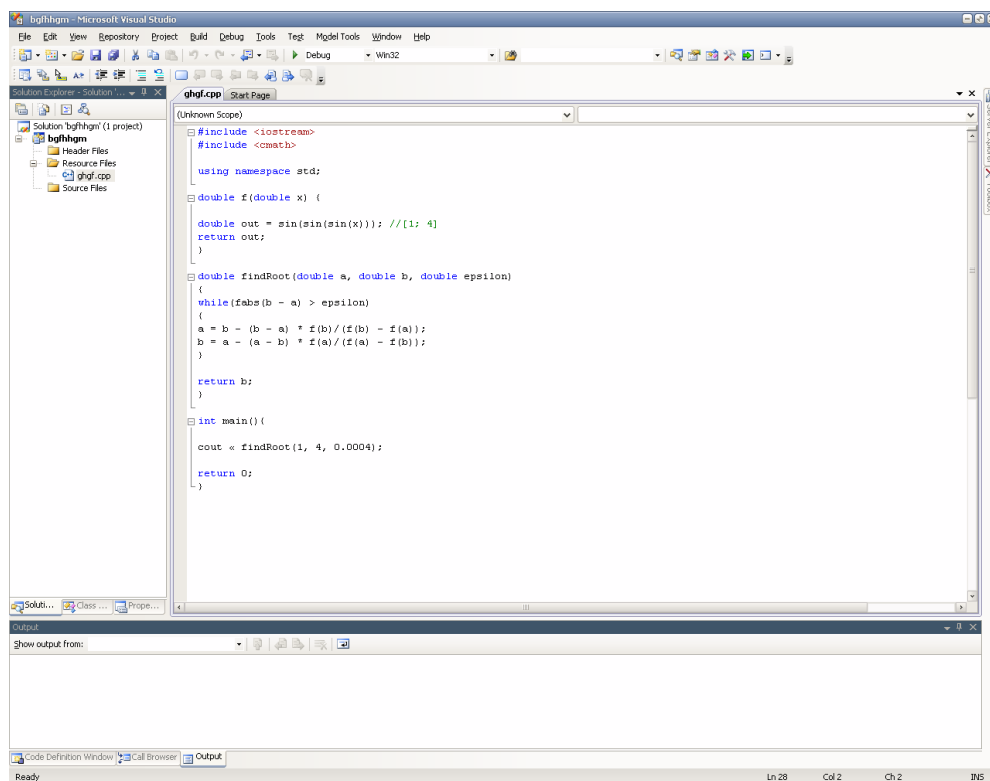


Рис. 3.1. Microsoft Visual Studio 2010

Програма реалізована на мові програмування C++ в середовищі Visual C++ 2010. Вибір середовища програмування обумовлений відносною простотою реалізації графічного інтерфейсу користувача (GraphicUserInterface – GUI).

При створенні засобів відображення і візуалізації використаний графічний інтерфейс OpenGL.

Бібліотека OpenGL являє собою інтерфейс програмування тривимірної графіки. Одиницею інформації є вершина, з яких створюються складніші об'єкти. Програміст створює вершини, вказує яким чином їх сполучати (лініями або багатокутниками), встановлює координати і параметри камери та ламп, а бібліотека OpenGL бере на себе роботу по створенню зображення на екрані. OpenGL ідеально підходить для програмістів, яким необхідно створити невелику тривимірну сцену і не замислюватися про деталі реалізації алгоритмів тривимірної графіки.

З погляду архітектури, графічна система OpenGL є конвеєром, що складається з декількох етапів обробки даних:

- Апроксимація кривих і поверхонь;
- Обробка вершин і збірка примітивів;
- Растеризація і обробка фрагментів;
- Операції над пікселями;
- Підготовка текстури;
- Передача даних в буфер кадру.

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.

Розроблений програмний комплекс виконує дискретизацію на скінченні 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;
- 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.3.3.). Одне з них являється Графічним Вікном, де створюється модель і редагується. А інше містить в собі основне Меню Програми та набір команд для роботи з фігурами.

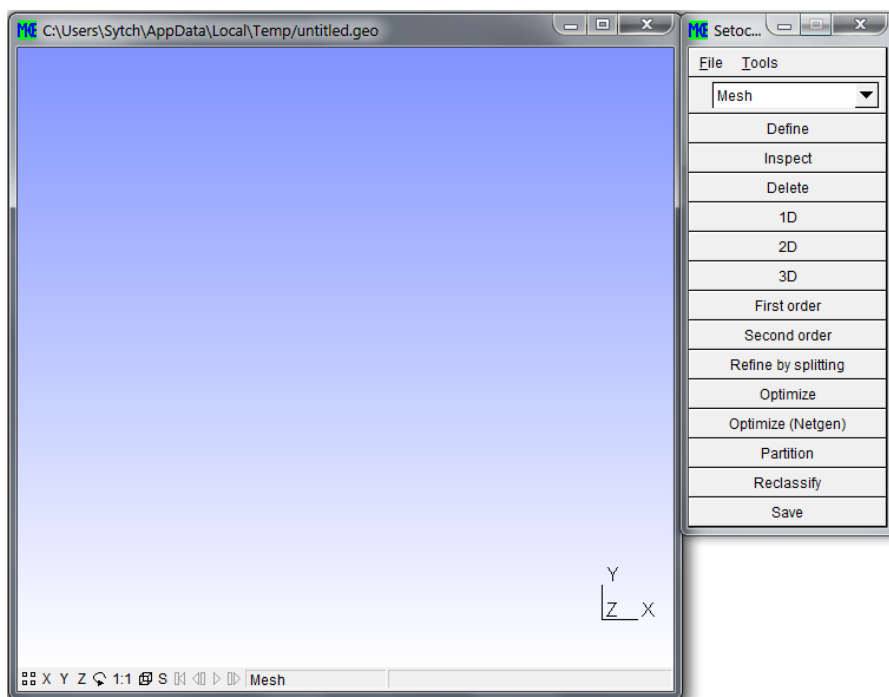


Рис.3.2. Інтерфейс програми

В нижній частині Графічного Вікна знаходиться Рядок Стану (Рис.3.3.).

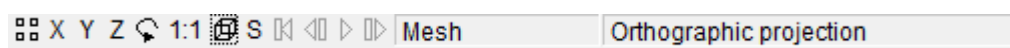


Рис.3.3. Рядок Стану

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.
	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.

За допомогою меню Files можна :

- відкрити потрібний файл ;
- об'єднати декілька фігур;
- видалити непотрібні елементи;
- відкрити одразу декілька вікон для роботи чи розділити активоване;
- зберегти побудовану модель;
- зберегти її безпосередньо отриману сітку;
- зберегти опції;
- вийти з програми.

За допомогою меню Tools можна :

- задати потрібні параметри моделі;
- зробити видимою будь-яку частину фігури для зручності роботи;
- задати координати обертання.

1D, 2D, 3D-розбиття виконується за допомогою команд з Головного меню.

Після відкриття фігури натискаємо на кнопку "2D". Результат показаний на рис. 3.1. 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.

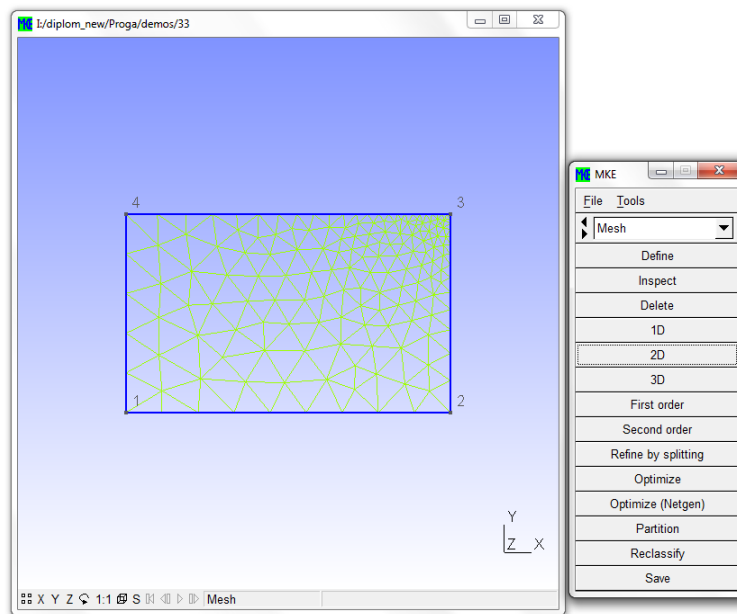


Рис. 3.4. Розбиття прямокутника на СЕ

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 "e" і розіб'ємо заново, то отримаємо результат, показаний на рис. 3.2, а в список команд додасться рядок: Recombine Surface {6}

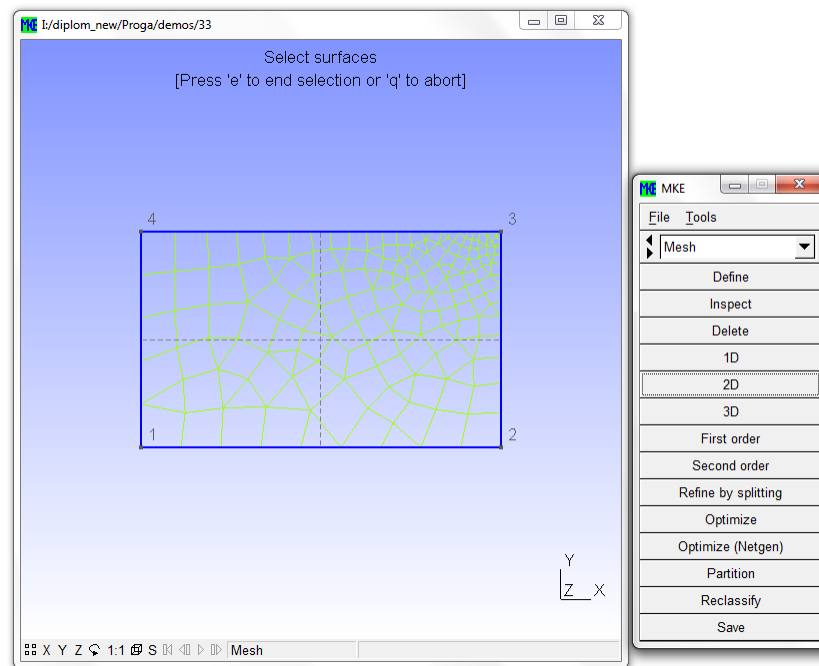


Рис. 3.5. Рекомбінована сітка

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'ємної фігури потрібно її вибрати за допомогою меню File. (Рис.3.5.)

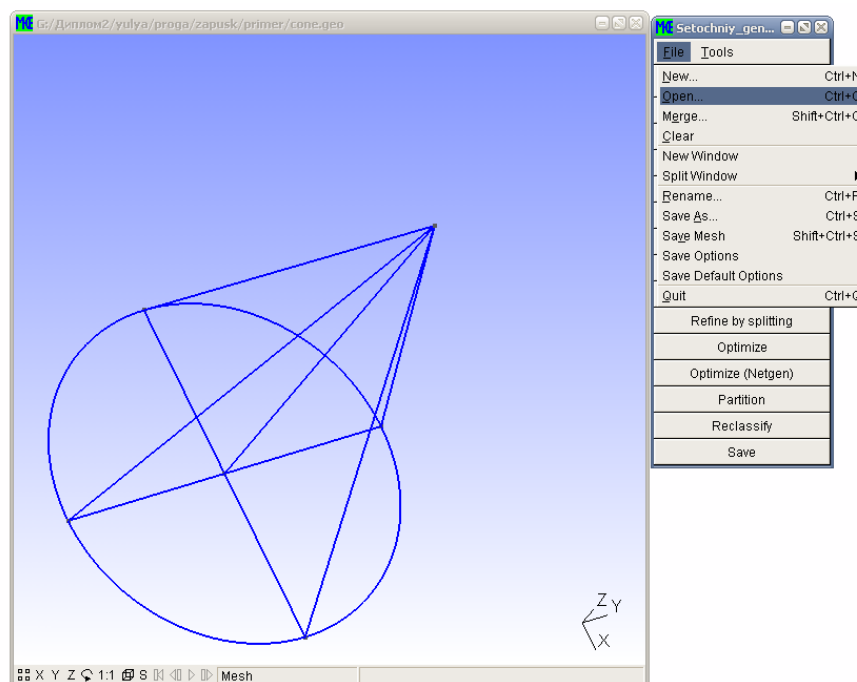


Рис.3.6. Вибір фігури для розбиття

За допомогою кнопки 2D фігура розбивається на скінченні елементи в 2D-площині (Рис.3.6.).

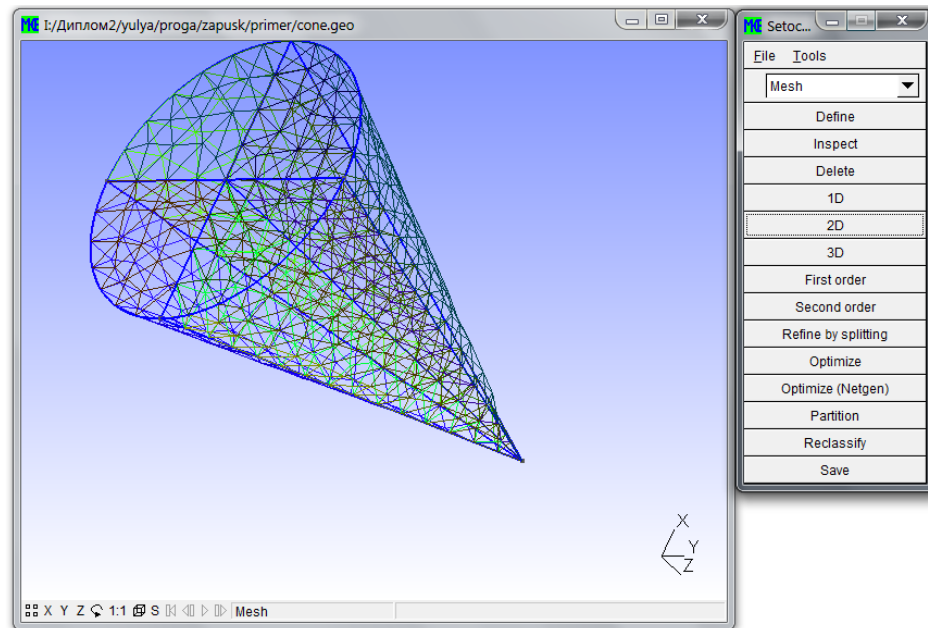


Рис.3.6. 2D-розбиття

При натисканні кнопки 3D виконується розбиття на елементи в 3D-площині (Рис.3.7.).

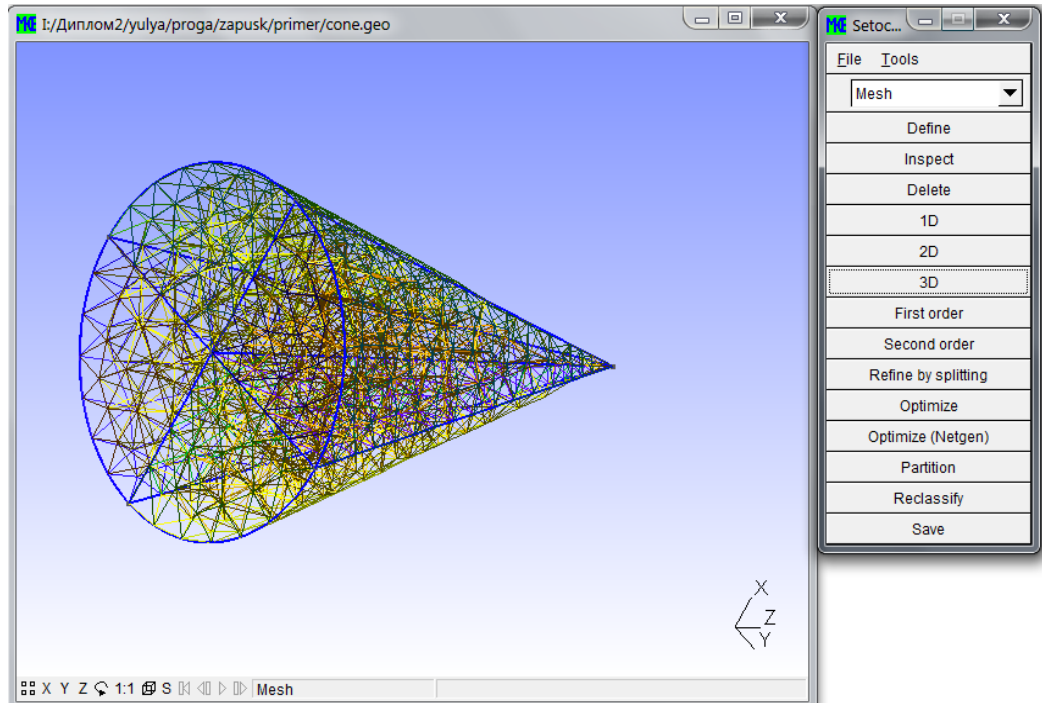


Рис.3.7. 3D-розбиття

Розглянемо опції програмного додатку які принципово впливають на побудову сітки, якість її елементів, а також їх збереження. У дужках після назви опції стоїть значення за замовчуванням.

Geometry.ExtrudeReturnLateralEntities (=1) - чи потрібно додавати додаткові номери в масив, що повертається командою Extrude.

Geometry.ExtrudeSplinePoints (=5) - кількість контрольних точок для сплайнів, створених внаслідок витягування.

Geometry.NumSubEdges (=20) - кількість відрізків між контрольними точками при відображенні кривих (ця опція не стосується побудови сітки, проте дозволяє сильно поліпшити відображення складних кривих).

Geometry.OCCFixSmallEdges (=1) - фіксувати малі ребра в IGES, STEP і BRep моделях (важлива опція при експорті моделей з інших CAD програм).

Geometry.OCCFixSmallFaces (=1) - фіксувати малі грані в IGES, STEP і BRep моделях (важлива опція при експорті моделей з інших CAD програм).

Geometry.Tolerance (=1e-06) - геометрична точність (допустиме відхилення).

Mesh.Algorithm3D (=1 - алгоритм побудови тетраедральної сітки (1-Delaunay, 2-Frontal). Алгоритм Delaunay найбільш стійкий і швидкий, однак іноді він змінює поверхневу сітку (спочатку будує одновимірну сітку, потім засновану на ній двовимірну, а потім засновану на двовимірної тривимірну сітку) і не підходить для створення змішаної структурованої/неструктурованою сітки. Для цих випадків підходить Frontal алгоритм. Якість елементів, вироблених цими алгоритмами, приблизно однакове.

Mesh.Binary (=0) - записувати сітку в бінарному форматі.

Mesh.CharacteristicLengthFactor (=1) - множник характеристичної довжини (застосовується, щоб подрібнити/згрубіть всі елементи сітки в рівній мірі).

Mesh.CharacteristicLengthMin (=0) - мінімальний розмір елемента сітки.

Mesh.CharacteristicLengthMax (=1e+22) - максимальний розмір елемента сітки.

Mesh.CpuTime (=0) - час (у секундах), відведений для побудови сітки.

Mesh.ElementOrder (=1) - порядок елементів сітки (1 - лінійні елементи; доступні також 2, 3, 4 і 5 порядки).

Mesh.MinimumCirclePoints (=7) - мінімальна кількість точок для апроксимації окружності.

Mesh.MinimumCurvePoints (=3) - мінімальна кількість точок для апроксимації кривої лінії.

Mesh.NumSubEdges (=2) - кількість ребер розбиття при відображенні елементів високих порядків. Ця опція важлива тільки з точки зору візуалізації. Проілюструємо це на прикладі розбиття кола грубої сіткою другого порядку (зліва - кількість ребер = 2, праворуч - кількість ребер = 5):

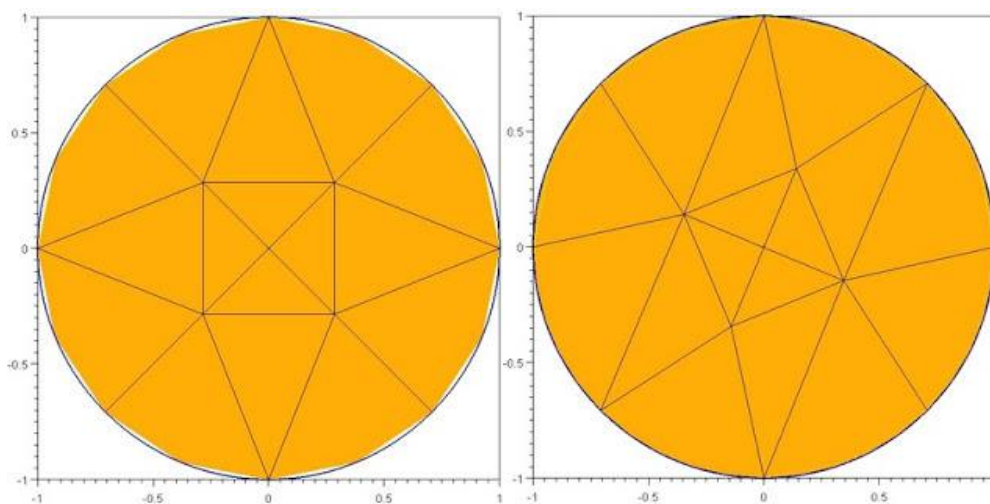


Рис. 3.8. Розбиття кола грубої сіткою

Mesh.SaveAll (=0) - зберігати всі елементи сітки, незалежно від фізичних об'єктів.

Mesh.SecondOrderIncomplete (=0) - створювати елементи неповного другого порядку (8 - вузлові чотирикутники, 20 - вузлові шестигранники і т.д.).

Mesh.SecondOrderLinear (=0) - чи повинні вершини елементів другого порядку бути отримані лінійною інтерполяцією (у цьому випадку, додаткові вузли ставляться строго на середину ребра елемента. Інакше, ребра ламаються

на два, а додаткові вузли зміщуються для більш точної апроксимації). Проілюструємо це (ліворуч опція відключена, праворуч - включена):

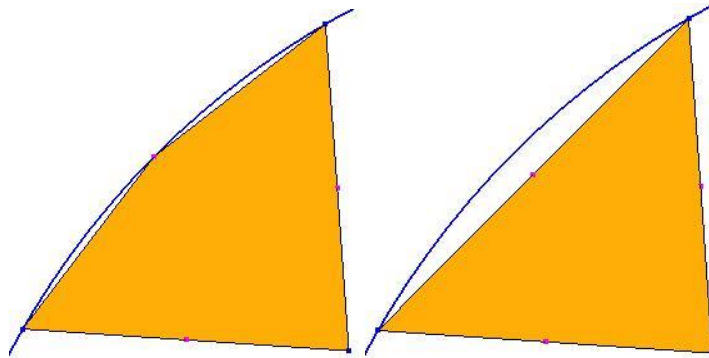


Рис. 3.9. Лінійна інтерполяція

За допомогою команди Optimize відбувається оптимізація сітки, тобто поліпшення топологічної якості сітки (Рис.3.10).

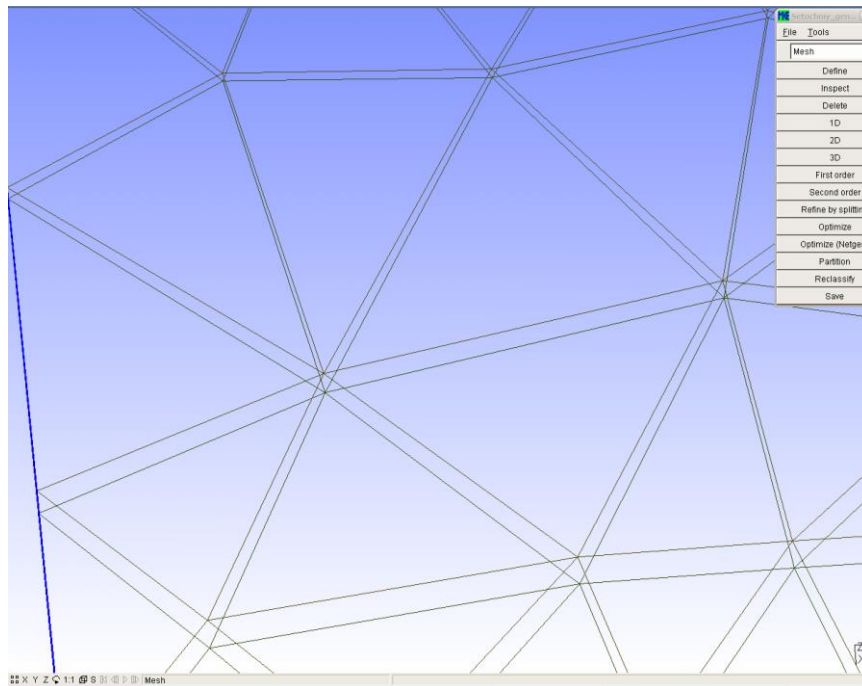


Рис.3.10. Оптимізація сітки

Зупинимося на опціях, які принципово впливають на якість сітки, її елементів, а також їх збереження. У дужках після назви опції стоїть значення за замовчуванням.

Mesh.Optimize (=0) - оптимізувати сітку для підвищення якості тетраедральних елементів.

Mesh.OptimizeNetgen (=0) - оптимізувати сітку для підвищення якості тетраедральних елементів, використовуючи алгоритм Netgen'a.

Mesh.RecombinationAlgorithm (= 0) - алгоритм рекомбінування (0 - стандартний , 1 - blossom).

Mesh.Smoothing (=1) - кількість кроків згладжування, що застосовується до підсумкової сітці.

Програмний додаток 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. 3.11).

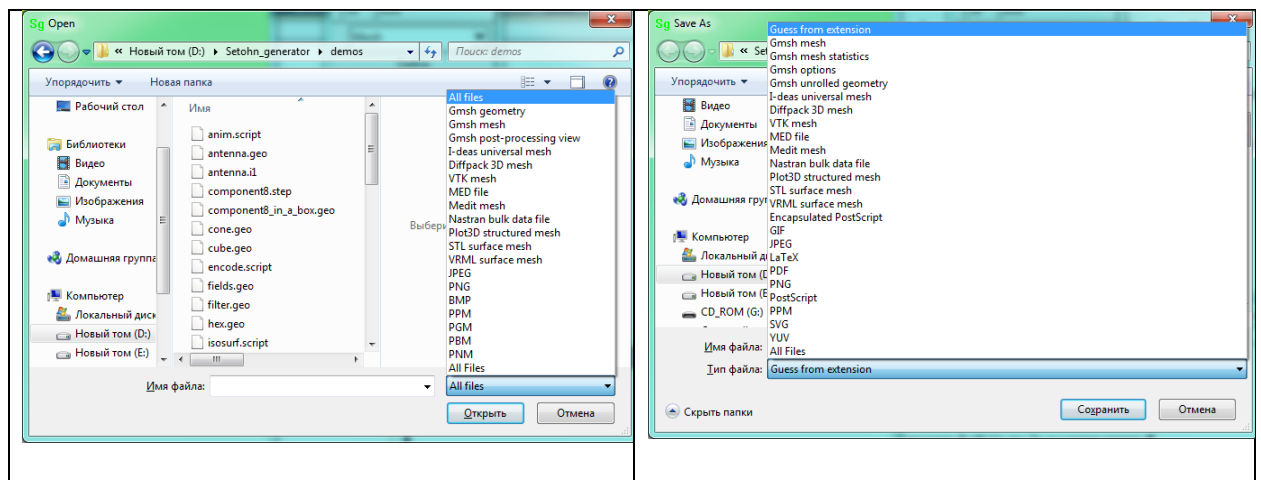


Рис. 3.11. Випадне меню "File": "Open" та "Save As"

Розглянемо деякі формати файлів по їх розширеннях:

- Gmsh geometry .geo - файл геометрії GMSH.
- STEP і IGES - відомі формати файлів геометрії, 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 генератора.

Msh-файл складається з секції з інформацією про формат файлу (MeshFormat) і секцій. Дана інформація описує вузли (Nodes) й елементи (Elements) сітки:

```
$MeshFormat
$EndMeshFormat
$Nodes
nNodes
iNode x y z
...
$EndNodes
$Elements
nElements
iElement type nTags tags nodes
...
$EndElements
```

В даному прикладі:

nNodes – кількість вузлів сітки,

i – індекс вузла,

x, y, z – координати вузла,

nElements – кількість елементів,

type – геометричний тип елемента (таблиця. 3.2),

nTags – кількість тегів,

tags – список тегів,

nodes – список вузлів з яких складається елемент.

За замовчуванням, у msh-файлі задається три теги: перший вказує номер фізичної сутності, до якої даний елемент належить, другий геометричну область, третє - номер частини сітки, до якої належить елемент.

Таблиця 3.2. – Геометричні типи елементів

Номер	Геометричний тип
1	Лінія
2	Трикутник
3	Чотирикутник
4	Тетраедр
5	Гексаедр

3.3. Висновки до розділу 3

Програма реалізована на мові програмування C++ в середовищі Visual C++ 2010, при створенні засобів відображення і візуалізації. Описано алгоритм генерації сіток. У додатку доступно створення 1- 2 - і 3-х мірних сіток, і їх оптимізація. Програма реалізується на мові програмування C++ в середовищі Visual C++ 2010. При створенні засобів відображення та візуалізації використовується графічний інтерфейс OpenGL.

ВИСНОВКИ

В магістерській роботі було розглянуто розробку додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

Для досягнення поставленої мети були вирішені наступні задачі:

- Проаналізовано основні поширені методи й алгоритми дискретизації плоских та просторових областей;
- Проведено моделювання та аналіз програмного забезпечення додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій, розроблено UML- моделі логічного і фізичного уявлень;
- Розглянуто програмну реалізацію додатка для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій.

В першому розділі 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 2 частини: побудову первинної дискретизації та її оптимізації. У додатку автоматичної генерації розрахункових сіток для скінчено-елементного моделювання для первинної дискретизації області прийнято застосовувати модифікований алгоритм Ватсона-Лавсона. Для оптимізації отриманої скінчено-елементної сітки - алгоритм Рапперта в пласкому випадку та Шевчука в тривимірному.

В другому розділі промодельоване та проаналізовано програмне забезпечення для автоматизації побудови дискретної (скінченно-елементної) моделі конструкцій. Розроблено UML- діаграми представлення статичної моделі структури, представлення моделі поведінки та фізичне представлення моделей програмної розробки.

У третьому розділі аргументується вибір середовища розробки додатку і мови програмування. Описано алгоритм генерації сіток. У додатку доступно

створення 1- 2 - і 3-х мірних сіток, і їх оптимізація. Програма реалізується на мові програмування C + + в середовищі Visual C + + 2010. При створенні засобів відображення та візуалізації використовується графічний інтерфейс OpenGL.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Александров П.С. Комбинаторная топология. ОГИЗ, Гостехиздат, 1947
2. 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).
3. 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).
4. Голованов Н.Н. Геометрическое моделирование / Н.Н. Голованов. – М.: Издво Физ.-мат. лит., 2002. – 472 с.
5. 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.
6. Карташева Е.Л. Инструментальные средства подготовки и анализа данных для решения трехмерных задач математической физики - Математическое моделирование. 1997. Т. 9. №6. С. 20.
7. Ласло М. Вычислительная геометрия и компьютерная графика на C++ / Ласло М. Пер. с англ. М.: БИНОМ, 1997. 304 с.
8. Математическое моделирование: Проблемы и результаты,-М. Наука, 2003
9. 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.

10. Построение расчетных сеток: Теория и приложения – В сб. под ред. Иваненко С.А., Гаранжа В.А., ВЦ РАН, М. 2002
11. Препарата Ф. Вычислительная геометрия: Введение /Препарата Ф., Шеймос М. Пер. с англ. М.: Мир, 1989. 478 с.
12. Рвачев Л. Методы логической алгебры в математической физике. Наукова думка, Киев, 1974г.
13. Роджерс Д. Математические основы машинной графики / Роджерс Д., Адаме Дж. Пер. с англ. М.: Машиностроение, 1980. 204 с.
14. Скворцов А.В. Обзор алгоритмов построения триангуляции Делоне / А.В. Скворцов // Вычислительные методы и программирование. – 2002. – Т.3. – С. 14-39.
15. 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.
16. Скворцов А.В. Триангуляция Делоне и ее применение / А.В. Скворцов // Томск, Изд-во Том. ун-та, 2002. С. 128.
17. Соллогуб А.И., Вальшин А.Т. Система трехмерного геометрического моделирования пространственных тел с использованием характеристических и R-функций. Программирование, 1991, N3, с.86-96
18. Сьярле Ф. Метод конечных элементов для эллиптических задач / Ф. Сьярле; пер. с англ. Б.И. Квасова. – М.: Изд-во «Мир», 1980. – 512 с.
19. Химушин Ф.Ф. Конструктивное геометрическое моделирование. –В. сб. Программное обеспечение САПР. ВЦАН СССР, М., 1987, с.102-121
20. Химушин Ф.Ф. Обзор методов геометрического моделирования для САПР.- В сб. Программное обеспечение САПР, ВЦ АН СССР, М., 1987, с.78-101
21. Чельцов А.Ю., Давыдченко Э.А. Представление данных и алгоритмы для системы геометрического моделирования, основанной на заметании. - В сб. Программное обеспечение САПР, ВЦАН СССР, М., 1987, с.121-133.

- 22.Шайдуrow В.В. Многосеточные методы конечных элементов / В.В. Шайдуrow. – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 288 с.
- 23.Шайдуrow В.В. Многосеточные методы конечных элементов. - М., Наука, 1989. - 288с.
- 24.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.
- 25.Baldwin K.H., Schreyer H.L. Automatic generation of quadrilateral elements by a conformal mapping. Eng. Comput. 1985, V.2, p. 187-194.
- 26.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
- 27.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.
- 28.Cook W.A. Body oriented (natural) coordinates for generating three-dimensional meshes. Int. J. Num. Meth. Eng., 1974, V.8, p. 27-43.
- 29.D.A. Field The legacy of automatic mesh generation from solid modeling, Computer-Aided Geometric Des,V12, 1995,651-674
- 30.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>.
- 31.Foley J, A.van Dam, S.Feiner, J.Hughes, R.Philips Introduction to Computer Graphics, - Addison Wesley, 1994
- 32.Fomenko A.T., Kunii T.L. Topological modeling for visualization, Springer-Verlag, Tokyo and Heidelberg, 1997
- 33.Frey, P.J., George, P.-L. Mesh Generation: Application to Finite Elements - HERMES Science Europe, OXFORD & PARIS, 2000, 814p.
- 34.Fritsch F, Piccinini R.A. Cellular structures in topology. Cambridge University Press, Cambridge, 1990

35. 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.
36. Ho-Le, K Finite. Element mesh generation methods: a review and classification. / CAD, 1988, V.20, N 1, p. 27-38.
37. 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.

ДОДАТОК

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"
#ifdef 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 is length is considered to be
    // zero. In some cases, a model edge can be considered as too
    // small an 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%.7f\t%.7f\t%.7f\t%.7f\t",
        nTotT, avg / (double)nTotT, best, worst, nTotGoodQuality,
        (double)nTotGoodQuality / nTotT);
fprintf(statreport, "%d\t%.7f\t%.7f\t%.7f\t%.7f\t%.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);
}

```