Науково-практичний звіт на тему

ТРІАНГУЛЯЦІЯ ПРОСТОГО МНОГОКУТНИКА

Єленіна Єгора Андрійовича, студента 3-го курсу, групи МІ-31

Анотація. При виконанні цієї лабораторної роботи, я розглянув метод відсікання, що використовується для тріангуляції простих многокутників. За даним методом було розроблено програмне забезпечення, що тріангулює простий многокутник, що задається множиною точок.

Abstract. In this lab, I examined the clipping method used to triangulate simple polygons. Using this method, software was developed that triangulates a simple polygon given by a set of points.

1 Вступ

Постановка задачі. У цій роботі розглядається задача тріангуляції простого многокутника — процес розбиття полігональної області на сукупність трикутників, які не перекриваються та разом утворюють вихідну фігуру. Така задача є базовою в обчислювальній геометрії та має практичне значення в численних галузях: комп’ютерній графіці, сітковій генерації для скінченноелементних методів, візуалізації сцен, 3D-моделюванні, картографії та робототехніці.

Розглядається метод "відрізання вух" (ear clipping), який дозволяє поетапно виділяти трикутники з простого многокутника. Цей метод є інтуїтивно зрозумілим, геометрично обґрунтованим та особливо зручним для реалізації в інтерактивних і графічних системах ([1], [2], [3]).

Аналіз останніх досліджень. Алгоритми тріангуляції полігонів активно вивчаються у літературі. У роботі Lazarus і Pocchiola [1] представлено формальну класифікацію задач тріангуляції та доведено їхню складність для різних випадків. Метод ear clipping, детально розглянутий у курсах обчислювальної геометрії [2], забезпечує правильність для простих полігонів, хоча має квадратичну складність у наївній реалізації. У публікації Chen et al. [3] описано вдосконалені версії цього методу з покращенням продуктивності та стабільності. У роботах [6], [7] досліджено оптимізацію алгоритмів тріангуляції для великомасштабних або паралельних обчислень. Огляд у [10] підтверджує важливість тріангуляції як ключової операції в геометричних перетвореннях. Також існує низка україномовних джерел [4], [5], [6], [8], [9], які описують приклади реалізацій та формулюють методику застосування тріангуляції в прикладних задачах — від візуалізації до аналізу просторових даних.

Новизна та ідея. Запропоновано реалізацію візуальної програмної системи, яка дозволяє користувачу вручну або автоматично вводити вершини простого полігону, а також здійснювати покрокову тріангуляцію методом відрізання вух. Особливістю підходу є інтеграція геометричного алгоритму з графічним інтерфейсом на базі Windows Forms, що забезпечує можливість візуального контролю, експериментального вивчення алгоритму та демонстрації принципу дії на практиці.

На відміну від класичних реалізацій, реалізований підхід оптимізовано для реального часу, надає миттєвий зворотний зв’язок і дозволяє аналізувати ефективність та стабільність алгоритму на прикладах довільної складності.

Мета роботи. Розробити програмну систему, яка реалізує тріангуляцію простого полігону методом відрізання вух, забезпечує візуалізацію результатів, підтримує ручний та автоматичний режим введення, і дозволяє користувачу експериментувати з різними конфігураціями вхідних даних.

Проаналізувати часову складність алгоритму, стабільність розбиття для довільних полігонів і придатність реалізації для навчальних та прикладних цілей.

2 Основна частина

2.1 Постановка задачі

Триангуляція простого многокутника. В нас заданий простий многокутник, який треба розбити на трикутники, з’єднуючи його вершини, без введення додаткових точок.

2.2 Основні означення, теореми

Теорема про два вуха

Вухо багатокутника визначається як вершина V така, що відрізок лінії між двома вершинами, які суміжні (з'єднані ребром) V, повністю лежить у внутрішній частині багатокутника. Теорема про два вуха стверджує, що кожен простий багатокутник має щонайменше два вуха. [11]

Вухом многокутника є трикутник, утворений трьома послідовними вершинами $Vi0$, $Vi1$ і $Vi2$, для якого $Vi1$ є опуклою вершиною. У такій вершині внутрішній кут менший за π радіан і відрізок $Vi0Vi2$ повністю лежить всередині багатокутника. Вухо має властивість, що всередині нього немає інших вершин многокутнику окрім вершин вуха.

Вершину $Vi1$ називають *наконечником вуха*.

Рефлекторна вершина – це така, для якої внутрішній кут, утворений двома ребрами, що виходять з неї, перевищує π радіан. [12]

2.3 Основні етапи алгоритму "відрізання вуха"

1. Серед всіх вершин знайти "потенційні наконечники вух" (вершини, що мають кут менший за π) та рефлекторні вершини
2. Проходячись по переліку "потенційних наконечників", для кожної такої вершини визначити, чи може вона бути наконечником вуха. Для цього:

    a) Для кожної вершини Vi, знаходимо суміжні з нею, що утворюють трикутник ($Vi-1$,Vi,$Vi+1$)

    b) Проходимося по всім вершинам многокутника (окрім $Vi-1$,Vi,$Vi+1$) і визначаємо, чи лежать вони всередині даного трикутника

    c) Якщо хоча б одна вершина лежить всередині, то Vi не є наконечником вуха, інакше це наконечник вуха
3. Зберігаємо для кожного етапу списки вершин, кут яких менший за π, рефлекторних вершин та наконечників вух
4. По одному видаляємо першу вершину зі списку наконечників вух.

Якщо V є вухом, яке видаляється, то конфігурація ребер у суміжних вершинах Vi−1 та Vi+1 може змінитися. Якщо сусідня вершина є опуклою, то вона залишається опуклою. Якщо суміжна вершина є вухом, воно не обов’язково залишається вухом після видалення V. Якщо сусідня вершина рефлекторна, є можливість, що вона стане опуклою і навіть наконечником вуха. Тому, маємо наступний крок.

5. Перевіряємо суміжні вершини на опуклість та чи є вони вухом. Оновлюємо списки.
6. Повторюємо кроки 4-5, поки не залишаться невидаленими 3 вершини

Ідея алгоритму відсікання «вух» полягає в послідовному відсіканні трикутників («вуха»). Вершина v_i називається «вухом», якщо діагональ проведена з v_{i-1} до v_{i+1} лежить строго у внутрішній області багатокутника P (рисунок 2.6).

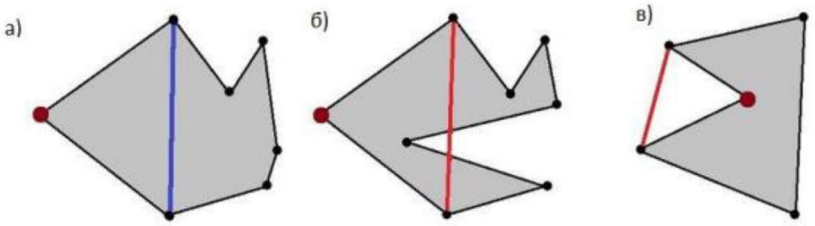


Рисунок 2.6 – Визначення вершини-вуха
(випадок а – «вухо», випадок б, в – ні)

59

Будемо розглядати вершини багатокутника в порядку обходу. Індексування вершин для зручності будемо вести по модулю n, тобто $v_{-1} = v_{n-1}$ і $v_0 = v_n$. Якщо вершина v_i є вухом, побудуємо діагональ $v_{i-1}\ v_{i+1}$ і відріжемо трикутник від $\Delta v_{i-1}\ v_i\ v_{i+1}$ від P. В іншому випадку переходимо до наступної вершині v_{i+1} в порядку обходу.

При знаходженні кожного «вуха» від багатокутника P відсікається трикутник, що складається з самого «вуха» і його двох суміжних вершин. Наприкінці алгоритму, коли всі «вуха» від P відрізані, залишається тільки один трикутник. Як нескладно бачити, триангуляція будується коректно.

2.4 Аргументація використання методу

Метод відсікання вух (Ear Clipping) має низку переваг, які обґрунтовують доцільність його використання для тріангуляції простих многокутників:

1. Легкість реалізації. Алгоритм є інтуїтивно зрозумілим і не потребує складних математичних конструкцій чи спеціалізованих структур даних, що

дозволяє легко його реалізувати на практиці навіть у навчальних чи демонстраційних проєктах. ([2], [4], [10]).

2. Прийнятна складність. Хоча базова реалізація методу має квадратичну часову складність $O(n^2)$, існують оптимізовані модифікації, які наближаються до лінійної складності для більшості вхідних даних. Це робить алгоритм достатньо ефективним для обробки полігонів з великою кількістю вершин. ([1], [3], [6]).

3. Гарантована коректність. У випадку простого многокутника без самоперетинів метод завжди завершується успішно, гарантуючи отримання правильної тріангуляції без накладень або пропусків. ([5], [11]).

4. Застосовність у реальному часі. Завдяки своїй конструктивній простоті, метод широко використовується у задачах реального часу – зокрема в комп'ютерній графіці, CAD-системах та інтерактивних додатках, де швидкість виконання має вирішальне значення. ([3], [7], [10]).

5. Можливість розширення на складніші випадки. Метод легко адаптується для обробки полігонів з отворами шляхом попереднього розбиття їх на множину простих полігонів, кожен з яких тріангується незалежно. ([1], [8], [9]).

2.5 Оцінка складності алгоритму

Просторова складність. Алгоритм відсікання вух потребує зберігання структури самого многокутника, а також допоміжних списків, зокрема списку вершин, які ще не були оброблені (вушок). Оскільки кількість таких вершин не перевищує кількість вершин вихідного многокутника, просторові витрати зростають лінійно від кількості елементів. Таким чином, просторова складність алгоритму становить $O(n)$, де n — кількість вершин вхідного полігону. [13]

Часова складність. На кожній ітерації алгоритм шукає вушко серед залишених вершин, перевіряючи його коректність (опуклість і відсутність інших вершин всередині). У гіршому випадку така перевірка виконується за O(n) часу. Оскільки кількість вушок (а отже, кроків тріангуляції) також дорівнює O(n), то загальна кількість операцій дорівнює $O(n^2)$. Таким чином, часова складність алгоритму оцінюється як $O(n^2)$. [13]

3 Практична частина

3.1 Особливості програмної реалізації

Програмна реалізація побудована на основі всіх кроків, описаних в пункті 2.3. Зберігаються списки всіх вершин з кутом, меншим за π, рефлекторних множин та наконечників вух. Метод, що використовується для тріангуляції завершує свою роботу, коли залишиться всього 3 точки. Таким чином, якщо початковий многокутник – трикутник, то ми повертаємо цей же трикутник, що є цілком логічним.

3.2 Опис основних функцій

Всі основні функції для проведення тріангуляції знаходяться в класі SimplePolygon. Серед них:

1) Функції для створення списку опуклих вершин FindConvexVerticles та функція для визначення чи є вершина опуклою IsConvex

```
1 reference
private void FindConvexVerticles()
{
    for (int i = 0; i < points.Count; i++)
    {
        if (IsConvex(points[i],points[Mod(i - 1)], points[Mod(i + 1)]))
        {
            convexVerticles.Add(points[i]);
        }
    }
}

2 references
private bool IsConvex(Point pointExtensions, Point adjP1, Point adjP2)
{
    return pointExtensions.GetAngleBetweenTwoPoints(adjP1, adjP2) < 180;
}
```

Рис. 1

2) Функція для створення списку рефлекторних вершин на основі вже збудованого списку опуклих вершин FindReflexVerticles

```
1 reference
private void FindReflexVerticles() => reflexVerticles = points.Except(convexVerticles).ToList();

1 reference
```

Рис. 2

3) Функція для створення списку накінечників вух FindEars та метод визначення чи є опукла вершина накінечником вуза IsEar

```
1 reference
private void FindEars()
{
    for (int i = 0; i < points.Count; i++)
    {
        if (convexVerticles.Contains(points[i]) &&
            IsEar(points[i], points[Mod(i - 1)], points[Mod(i + 1)]))
        {
            ears.Add(points[i]);
        }
    }
}

3 references
private bool IsEar(Point pointExtensions, Point adjP1, Point adjP2)
{
    var suspectedEarTriangle = new Triangle(pointExtensions, adjP1, adjP2);
    foreach (var p in points)
    {
        if (p != pointExtensions && p != adjP1 && p != adjP2 && suspectedEarTriangle.IsPointInside(p))
        {
            return false;
        }
    }
    return true;
}
```

Рис. 3

4) Функція IsPointInside класу Triangle для визначення чи лежить точка всередині трикутника. Для цього вимірюється сума площин трикутників, що утворюються вершинами початкового трикутника та точкою і площина самого трикутника. Якщо вони рівні, точка лежить всередині.

```
1 reference
public bool IsPointInside(Point p)
{
    var thisSquare = FindSquare();
    var squareP1 = (new Triangle(P1,P2,p)).FindSquare();
    var squareP2 = (new Triangle(P1, P3, p)).FindSquare();
    var squareP3 = (new Triangle(P3, P2, p)).FindSquare();
    return thisSquare == squareP1 + squareP2 + squareP3;
}
```

Рис. 4

5) Функція RecalculateEars для визначення чи стала опукла точка накінечником вуха після видалення вуха і оновлення списку накінечників вух

```
1 reference
private void RecalculateEars((Point, Point) adjPoints)
{
    var adjPointsOfAdj1 = GetAdjPoints(adjPoints.Item1);
    var adjPointsOfAdj2 = GetAdjPoints(adjPoints.Item2);

    if (convexVerticles.Contains(adjPoints.Item1))
    {
        AddIfEar(adjPoints.Item1, adjPointsOfAdj1.Item1, adjPointsOfAdj1.Item2);
    }
    if (convexVerticles.Contains(adjPoints.Item2))
    {
        AddIfEar(adjPoints.Item2, adjPointsOfAdj2.Item1, adjPointsOfAdj2.Item2);
    }

    void AddIfEar(Point p, Point adj1, Point adj2)
    {
        if (IsEar(p, adj1, adj2))
        {
            if (!ears.Contains(p))
            {
                ears.Add(p);
            }
        }
        else
        {
            ears.Remove(p);
        }
    }
}
```

Рис. 5

6) Функція RecalculateConvexVerticles для визначення чи стала рефлекторна вершина опуклою після видалення вуха. Якщо стала опуклою, чи стала накінечником вуха. Оновлюються списки опуклих вершин та накінечників вух

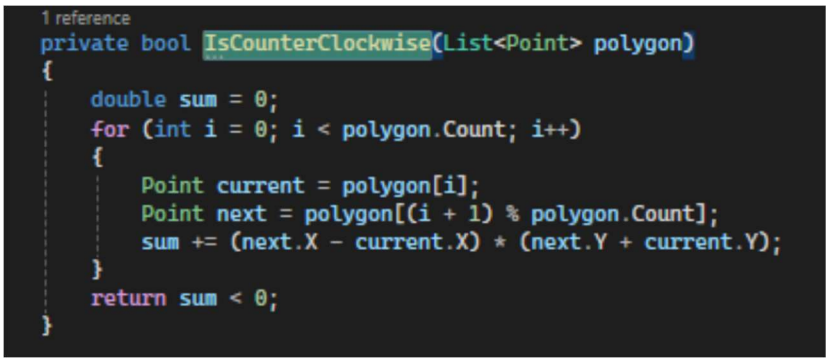
```
1 reference
private void RecalculateConvexVerticles((Point, Point) adjPoints)
{
    var adjPointsOfAdj1 = GetAdjPoints(adjPoints.Item1);
    var adjPointsOfAdj2 = GetAdjPoints(adjPoints.Item2);

    if (!convexVerticles.Contains(adjPoints.Item1))
    {
        AddIfConvexIfEar(adjPoints.Item1, adjPointsOfAdj1.Item1, adjPointsOfAdj1.Item2);
    }
    if (!convexVerticles.Contains(adjPoints.Item2))
    {
        AddIfConvexIfEar(adjPoints.Item2, adjPointsOfAdj2.Item1, adjPointsOfAdj2.Item2);
    }

    void AddIfConvexIfEar(Point p, Point adj1, Point adj2)
    {
        if (IsConvex(p, adj1, adj2))
        {
            convexVerticles = convexVerticles.Prepend(adj2).ToList();
            reflexVerticles.Remove(p);
            if (IsEar(p, adj1, adj2))
            {
                ears = ears.Prepend(p).ToList();
            }
        }
    }
}
```

Рис. 6

7) Функція IsCounterClockwise на перевірку напрямку обходу в коді, щоб уникнути крашу при неправильному введені порядку точок. (бо це користувач і він може не задумуватись про точний порядок за годинниковою стрілкою або проти)



```
1 reference
private bool IsCounterClockwise(List<Point> polygon)
{
    double sum = 0;
    for (int i = 0; i < polygon.Count; i++)
    {
        Point current = polygon[i];
        Point next = polygon[(i + 1) % polygon.Count];
        sum += (next.X - current.X) * (next.Y + current.Y);
    }
    return sum < 0;
}
```

Рис. 7

8) Головна функція самої тріангуляції Triangulate

```
1 reference
public List<Triangle> Triangulate()
{
    List<Triangle> result = new List<Triangle>();

    if (points.Count < 3)
        throw new InvalidOperationException("Для тріангуляції потрібно щонайменше 3 вершини.");

    if (!IsCounterClockwise(points))
        points.Reverse();

    FindConvexVerticles();
    FindReflexVerticles();
    FindEars();
    while(points.Count > 3)
    {
        Point ear = ears[0];
        var adjPoints = GetAdjPoints(ear);
        result.Add(new Triangle(ears[0], adjPoints.Item1, adjPoints.Item2));
        points.Remove(ear);

        RecalculateEars(adjPoints);
        RecalculateConvexVerticles(adjPoints);
        convexVerticles.Remove(ear);
        ears.Remove(ear);
    }
    result.Add(new Triangle(points[0], points[1], points[2]));
    return result;

}
```

Рис. 8

3.3 Користувацький інтерфейс

У програмному забезпеченні передбачено два способи введення даних:

1. Автоматична генерація опуклого многокутника, де користувач задає кількість вершин;
2. Ручне введення, при якому користувач самостійно вводить координати кожної вершини.

На початку роботи програма виводить список вершин побудованого многокутника. Після ініціації процесу тріангуляції користувачем, на екран виводиться вже тріангульований многокутник.

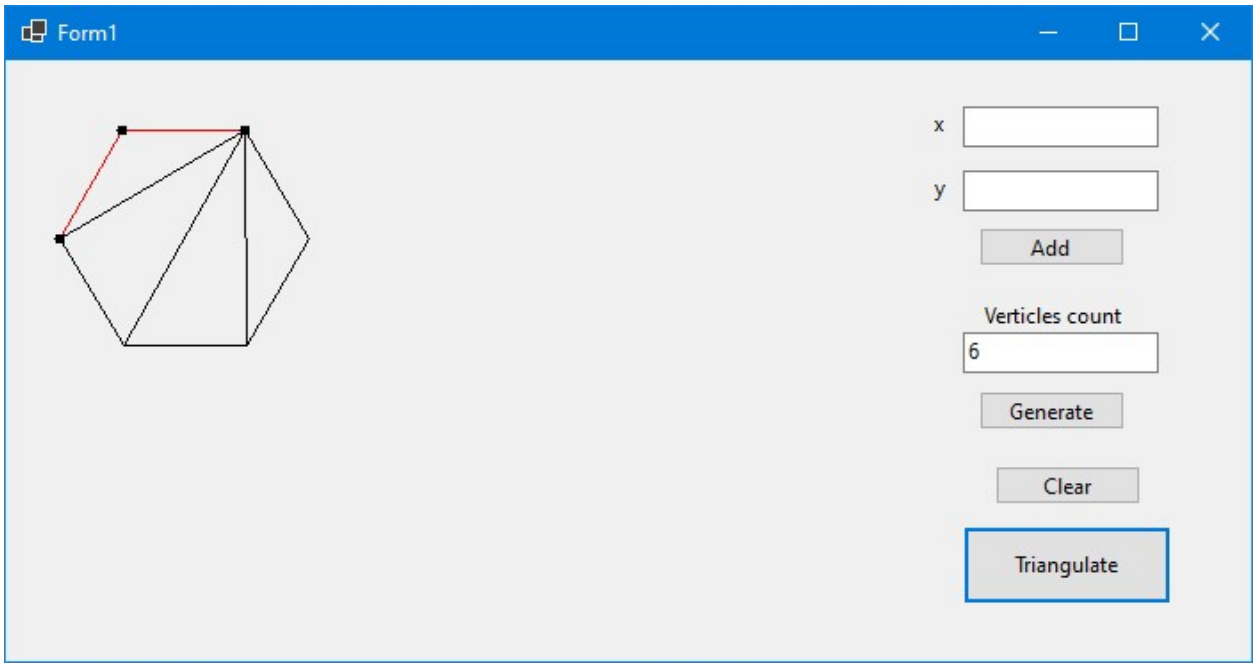


Рис. 9 – Інтерфейс програми

3.4 Технічні вимоги

Програмне забезпечення було реалізовано мовою програмування C# з використанням інтегрованого середовища розробки Visual Studio 2022. Основна логіка програми була винесена до бібліотеки класів, а користувацький інтерфейс створено за допомогою технології Windows Forms.

3.5 Лістинг основних модулів програми

Основна логіка програми складається з 4 класів: Triangle, SimplePolygon (тут знаходиться метод для тріангуляції), Generator (генерує вершини опуклого багатокутника за заданою кількістю вершин) та Extensions (розширення з доданим методом для визначення кута між трьома точками).

Висновки

У межах даної лабораторної роботи було досліджено метод тріангуляції простого многокутника шляхом відсікання вух (Ear Clipping). Алгоритм продемонстрував свою ефективність у випадках з відносно невеликою кількістю вершин і простими геометричними конфігураціями. Реалізація методу в рамках програмного забезпечення дозволила переконатися в його працездатності та точності.

Перспективним напрямом подальшого розвитку є вдосконалення алгоритму за допомогою використання структур даних (наприклад, двозв'язного списку для представлення контурів), а також паралелізація обчислень для підвищення продуктивності. Ще одним напрямом є розробка гібридних методів, які комбінують відсікання вух з іншими алгоритмами тріангуляції, що дозволить підвищити стабільність та ефективність для складніших випадків.

У цілому, отримані результати підтверджують доцільність застосування методу відсікання вух для базової тріангуляції, а проведене програмне моделювання створює надійну основу для подальших розробок та експериментів у галузі обробки геометричних структур.

Список літератури

1. Lazarus F., Pocchiola M. Triangulating simple polygons and equivalent problems // *arXiv preprint arXiv:1212.6038*, 2012. – Режим доступу: https://arxiv.org/pdf/1212.6038

2. Цветков С. *Triangulation By Ear Clipping* // Курс лекцій з обчислювальної геометрії.

3. Chen Z., et al. Efficient algorithms for ear clipping based polygon triangulation // *Journal of Visual Communication and Image Representation*. – 2018. – Vol. 55.

– Режим доступу:
https://www.sciencedirect.com/science/article/pii/S092577211830004X

4. Вікіпедія. Тріангуляція многокутника // *Вікіпедія – вільна енциклопедія*. – Режим доступу: https://uk.wikipedia.org/wiki/Тріангуляція_многокутника

5. Вікіпедія. Тріангуляція (геометрія) // *Вікіпедія – вільна енциклопедія*. – Режим доступу: https://uk.wikipedia.org/wiki/Тріангуляція_(геометрія)

6. Гаврилюк В.І., Мельничук І.В. Задача триангуляції багатокутника // *Інформаційні технології і моделювання – ВНТУ*. – Режим доступу: https://inmad.vntu.edu.ua/portal/static/8E47BDCF-556C-4C49-A065-18B05801E40E.pdf

7. Eder M., Held M., Palfrader P. Experiments on Parallel Polygon Triangulation Using Ear Clipping // *EuroCG 2015, Extended Abstract*.

8. Старовойтов О.В. Геометричні алгоритми в задачах тріангуляції багатокутників // *Магістерська робота – ЗНУ*, 2021. – Режим доступу: https://dspace.znu.edu.ua/xmlui/bitstream/handle/12345/25233/StarovoitovOleksandrVictorovich.pdf

9. Збірник ІТтаМС 2023 // *Житомирська політехніка*. – Режим доступу: http://eprints.zu.edu.ua/37568/1/Збірник_ІТтаМС_2023_друк-90-91.pdf

10. Mark de Berg, Marc van Kreveld, Mark Overmars та Otfried Schwarzkopf (2000). *Computational Geometry* (вид. 2nd revised). Springer-Verlag. ISBN 3-540-65620-0. Chapter 3: Polygon Triangulation: pp.45–61.

11. Вікіпедія**.** Теорема про два вуха // *Вікіпедія – вільна енциклопедія*. – Режим доступу: https://uk.wikipedia.org/wiki/Теорема_про_два_вуха

12. David Eberly, *Geometric Tools*, Redmond WA 98052 (2022)

13. Дисертація С. В. Цюнь**.** *Алгоритмічне забезпечення для задач комп'ютерної геометрії:* дисертація на здобуття наук. ступеня канд. техн. наук: 05.13.06 / Національний університет «Львівська політехніка». – Львів, 2020.