Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Новосибирский государственный технический университет»

Кафедра прикладной математики

ОТЧЕТ ПО ПРАКТИКЕ

Учебная практика: технологическая (проектно-технологическая) практика

(наименование практики в соответствии с учебным планом)

Направление подготовки: 01.04.02 Прикладная математика и информатика

Профиль: Компьютерное моделирование и наукоемкое программное обеспечение и биоинформатика

Выполнил:

Студент
Сумин Ярослав Евгеньевич
(Ф.И.О.)

Группа ПММ-42

Факультет ПМИ

_____
подпись

«___» _____ 2025 г.

Проверил:

Руководитель от НГТУ
Рояк Михаил Эммануилович
(Ф.И.О.)

Балл: _____, ECTS_____,

Оценка _____
«отлично», «хорошо», «удовлетворительно», «неуд.»,
«зачтено», «не зачтено»

_____
подпись

«___» _____ 2025 г.

Новосибирск 2025

СОДЕРЖАНИЕ

Введение 3

1 Теоретическая часть 4
1.1 Алгоритм разбиения шестигранника 4
1.2 Визуализация сетки 6

2 Тестирование 8
2.1 Тестирование на 1 шестиграннике 8
2.2 Тестирование на 2 шестигранниках 9
2.3 Тестирование на 4 шестигранниках 10
2.4 Тестирование на 4 шестигранниках 12

3 Заключение 16

Приложение А. Текст программы 18

ВВЕДЕНИЕ

Целью данной практической работы является написание программы, которая умеет разбивать сетку из шестигранников на пирамиду и тетраэдры, а так же умеющую визуализировать полученное разбиение.

Результатом работы станет программа, визуализирующая разбиение куба на пирамиду с основанием на нижней грани куба и тетраэдрами в остальном объеме.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. АЛГОРИТМ РАЗБИЕНИЯ ШЕСТИГРАННИКА

В качестве шестигранника будет использоваться куб, представленный на рисунке 1.

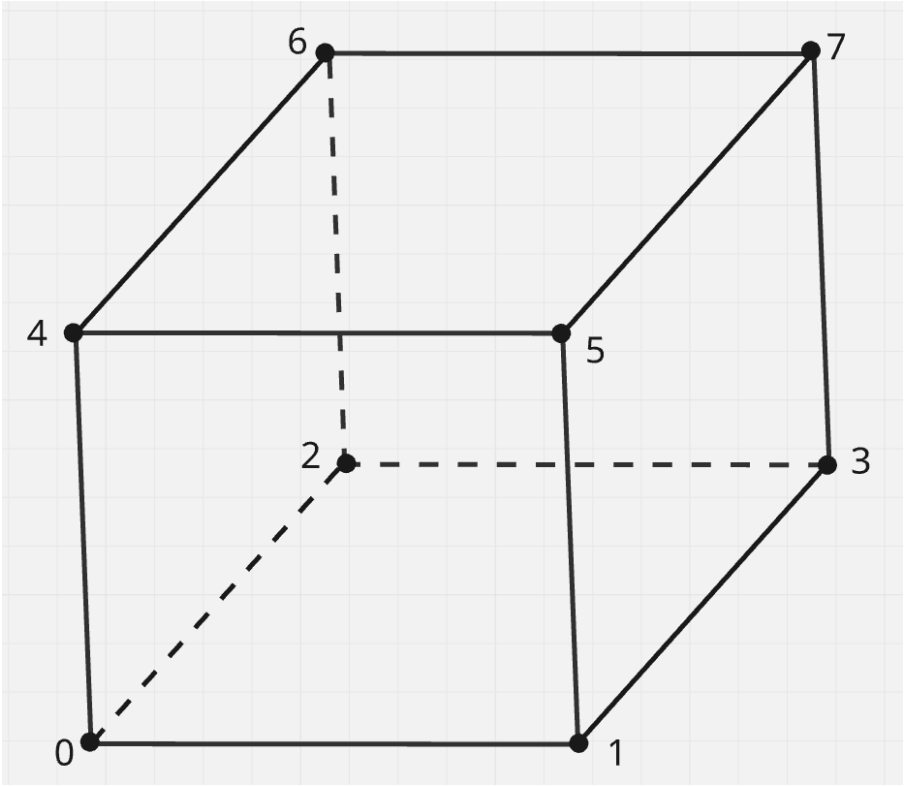

Рисунок 1 – Исходная фигура

При разбиении нижняя грань остаётся неизменной, а все остальные грани могут делиться на 2 треугольника, в зависимости от проведённой диагонали. В данной работе будет использоваться разбиение, представленное на рисунке 2.

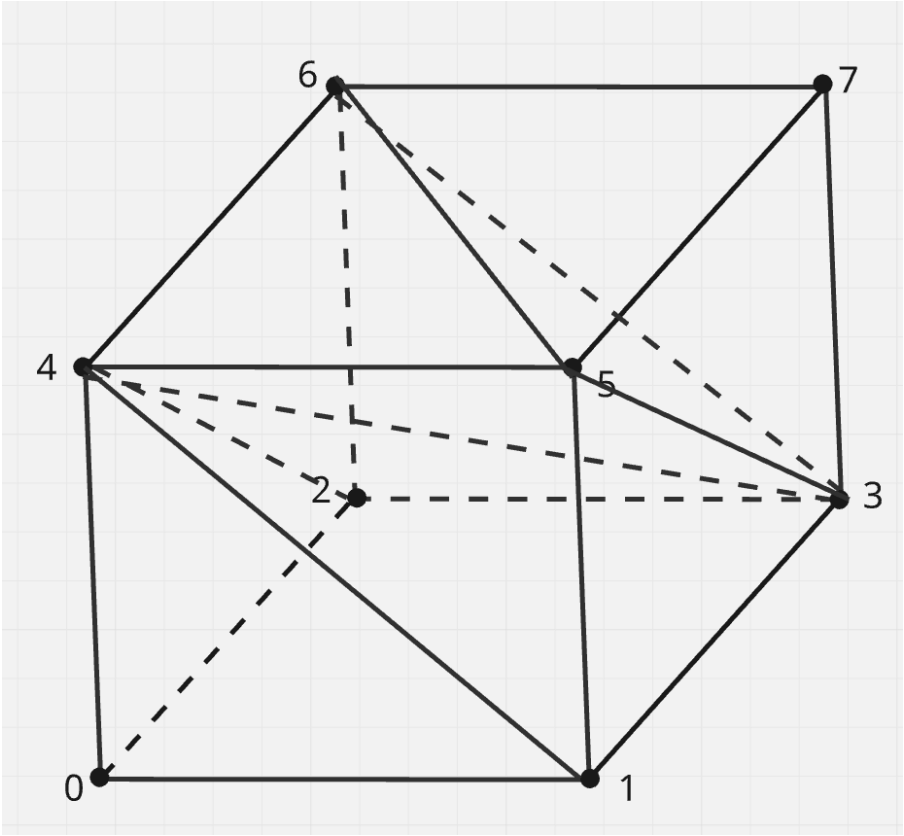

Рисунок 2 – Шаблон разбиения куба

При таком разбиении мы получаем 1 пирамиду и 4 тетраэдра с номерами узлов:

Пирамида: $[0, 1, 2, 3, 4]$;

Тетраэдр 1: $[1, 3, 4, 5]$;

Тетраэдр 2: $[2, 3, 4, 6]$;

Тетраэдр 3: $[3, 4, 5, 6]$;

Тетраэдр 4: $[3, 5, 6, 7]$.

Входные данные:

global_nodes_coordinates – координаты узлов всех шестигранников, которые учавствуют в разбиении.

global_nodes_indicies – номера узлов для каждого шестигранника, который учавствует в разбиении.

Выходные данные:

tetrahedron_indicies – вектор, содержащий номера узлов для каждого тетраэдра, полученного после разбиения.

pyramide_indicies – вектор, содержащий номера узлов для каждой пирамиды, полученной после разбиения.

Сам алгоритм разбиения заключается в следующем:

1. **Сопоставление глобальных и локальных номеров для узлов куба.** Каждой вершине куба соответствует глобальный индекс из исходной сетки. Мы строим локальное упорядочение этих вершин относительно нижней фиксированной грани куба. В результате каждой вершине куба присваивается локальный номер от 0 до 7, где первые 4 номера – это вершины нижней грани, а оставшиеся 4 – это вершины верхней грани.
2. **Применение шаблона разбиения.**
3. **Проверка ориентации тетраэдров.** После построения всех тетраэдров проверяется их ориентированный объём. Если объём отрицательный у тетраэдра меняются два локальных номера вершин.
4. **Сохранение результатов.** Полученные номера вершин пирамиды и тетраэдров сохраняются в отдельные списки для последующей визуализации.

1.2. ВИЗУАЛИЗАЦИЯ СЕТКИ

Для визуализации сетки на языке программирования C++ было принято решение использовать библиотеку VTK (Visualization Toolkit). Она позволяет работать с объёмными сетками, отображать их в 3D, настраивать цвета, прозрачность и границы ячеек, а также предоставляет интерактивное вращение и масштабирование объектов.

Сам алгоритм визуализации разбиения куба на пирамиду и тетраэдры заключается в следующем:

1. **Построение сетки для VTK.** Из списков глобальных координат вершин и индексов ячеек создаётся объект vtkUnstructuredGrid. Каждой ячейке присваивается тип: пирамида (1) или тетраэдр (2).
2. **Фильтрация элементов по типу.** Используются фильтры vtkThreshold, чтобы отдельно выбрать пирамиды и тетраэдры по их типу.
3. **Создание поверхностных объектов.** Для каждой группы ячеек применяется фильтр vtkDataSetSurfaceFilter, который преобразует объёмные ячейки в поверхности для отображения.
4. **Настройка отображения.** Создаются объекты vtkActor для каждой группы, задаются цвета, прозрачность и видимость границ.
5. **Рендеринг и интерактивность.** Все акторы добавляются на vtkRenderer, который помещается в окно визуализации vtkRenderWindow. Затем за-

пускается интерактивный рендерер для вращения, масштабирования и полного обзора разбиения куба.

2. ТЕСТИРОВАНИЕ

Тестирование проводилось на сетках, содержащих 1, 2, 4 и 8 шестигранников. Целью было убедиться в корректности алгоритма разбиения кубических элементов на пирамиды и тетраэдры, а также проверить визуализацию полученной сетки.

2.1. ТЕСТИРОВАНИЕ НА 1 ШЕСТИГРАННИКЕ

Входные данные:

- global_node_coordinates:

  0.0 0.0 0.0
  1.0 0.0 0.0
  0.0 1.0 0.0
  1.0 1.0 0.0
  0.0 0.0 1.0
  1.0 0.0 1.0
  0.0 1.0 1.0
  1.0 1.0 1.0

- global_nodes_indicies:

  0 1 2 3 4 5 6 7

Выходные данные:

- **Пирамида:** 0(0,0,0) 1(1,0,0) 3(1,1,0) 2(0,1,0) 4(0,0,1)
- **Тетраэдр 1:** 1(1,0,0) 3(1,1,0) 4(0,0,1) 5(1,0,1)
- **Тетраэдр 2:** 3(1,1,0) 2(0,1,0) 4(0,0,1) 6(0,1,1)
- **Тетраэдр 3:** 3(1,1,0) 4(0,0,1) 5(1,0,1) 6(0,1,1)
- **Тетраэдр 4:** 5(1,0,1) 3(1,1,0) 6(0,1,1) 7(1,1,1)

Результат разбиения представлены на рисунке 3.

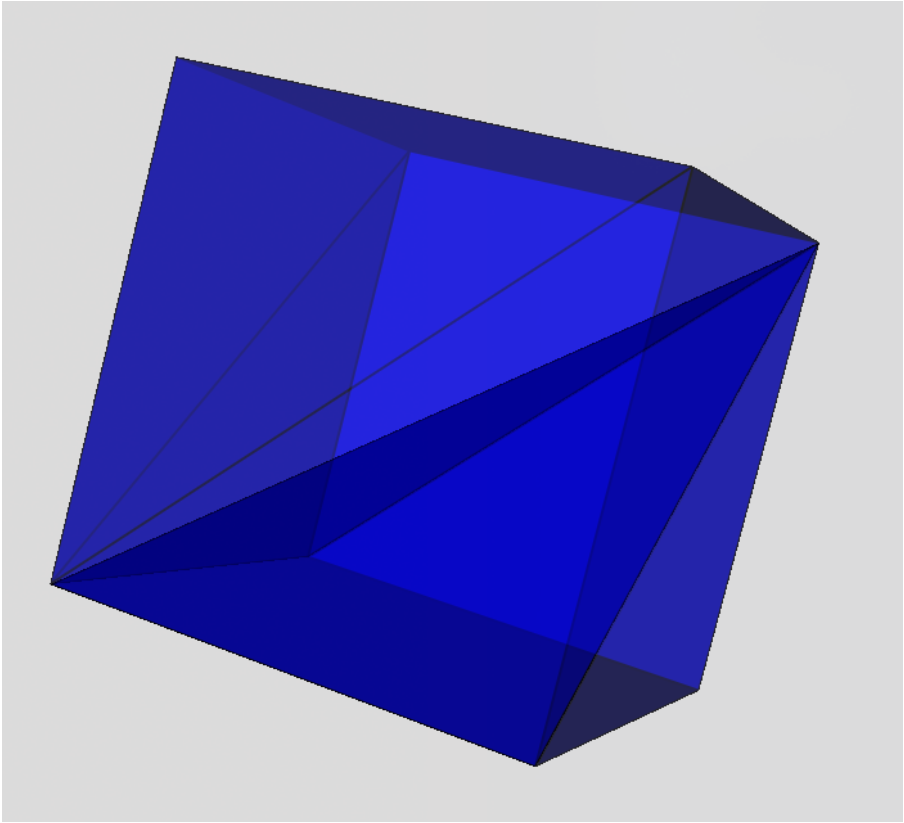

Рисунок 3 – Результат разбиения одного шестигранника

2.2. ТЕСТИРОВАНИЕ НА 2 ШЕСТИГРАННИКАХ

Входные данные:

- global_node_coordinates:

  0.0 0.0 0.0
  1.0 0.0 0.0
  2.0 0.0 0.0
  0.0 1.0 0.0
  1.0 1.0 0.0
  2.0 1.0 0.0
  0.0 0.0 1.0
  1.0 0.0 1.0
  2.0 0.0 1.0
  0.0 1.0 1.0
  1.0 1.0 1.0
  2.0 1.0 1.0

- global_nodes_indicies:

  0 1 3 4 6 7 9 10

  1 2 4 5 7 8 10 11

Выходные данные:

- **Пирамида 1:** 0(0,0,0) 1(1,0,0) 4(1,1,0) 3(0,1,0) 6(0,0,1)
- **Пирамида 2:** 1(1,0,0) 2(2,0,0) 5(2,1,0) 4(1,1,0) 7(1,0,1)
- **Тетраэдр 1:** 1(1,0,0) 4(1,1,0) 6(0,0,1) 7(1,0,1)
- **Тетраэдр 2:** 4(1,1,0) 3(0,1,0) 6(0,0,1) 9(0,1,1)
- **Тетраэдр 3:** 4(1,1,0) 6(0,0,1) 7(1,0,1) 9(0,1,1)
- **Тетраэдр 4:** 7(1,0,1) 4(1,1,0) 9(0,1,1) 10(1,1,1)
- **Тетраэдр 5:** 2(2,0,0) 5(2,1,0) 7(1,0,1) 8(2,0,1)
- **Тетраэдр 6:** 5(2,1,0) 4(1,1,0) 7(1,0,1) 10(1,1,1)
- **Тетраэдр 7:** 5(2,1,0) 7(1,0,1) 8(2,0,1) 10(1,1,1)
- **Тетраэдр 8:** 8(2,0,1) 5(2,1,0) 10(1,1,1) 11(2,1,1)

Результат разбиения представлены на рисунке 4.

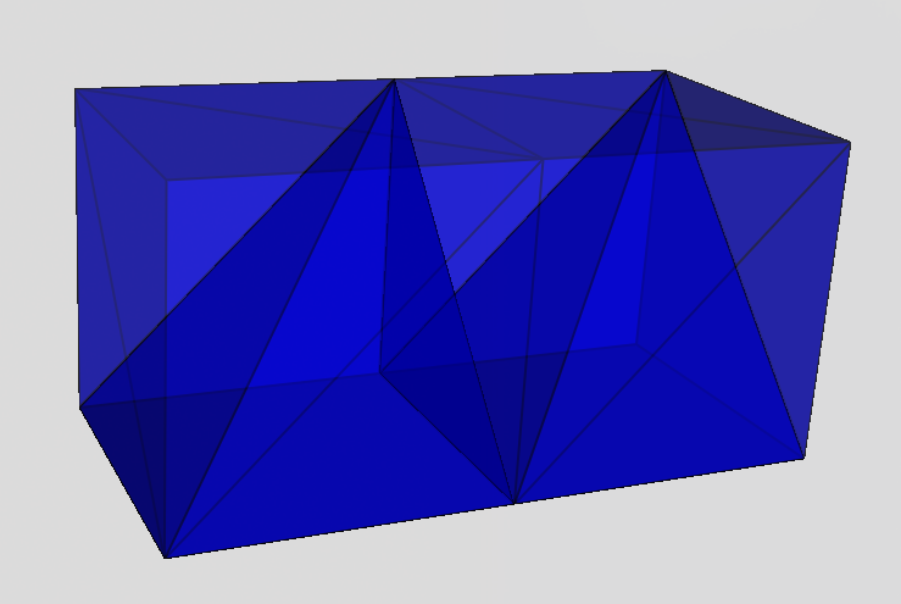

Рисунок 4 – Результат разбиения двух шестигранников

2.3. ТЕСТИРОВАНИЕ НА 4 ШЕСТИГРАННИКАХ

Входные данные:

- global_node_coordinates:

0.0 0.0 0.0
1.0 0.0 0.0
2.0 0.0 0.0
0.0 1.0 0.0
1.0 1.0 0.0
2.0 1.0 0.0
0.0 2.0 0.0
1.0 2.0 0.0
2.0 2.0 0.0
0.0 0.0 1.0
1.0 0.0 1.0
2.0 0.0 1.0
0.0 1.0 1.0
1.0 1.0 1.0
2.0 1.0 1.0
0.0 2.0 1.0
1.0 2.0 1.0
2.0 2.0 1.0

- global_nodes_indicies:

0 1 3 4 9 10 12 13
1 2 4 5 10 11 13 14
3 4 6 7 12 13 15 16
4 5 7 8 13 14 16 17

Выходные данные:

- **Пирамида 1:** 0(0,0,0) 1(1,0,0) 4(1,1,0) 3(0,1,0) 9(0,0,1)
- **Пирамида 2:** 1(1,0,0) 2(2,0,0) 5(2,1,0) 4(1,1,0) 10(1,0,1)
- **Пирамида 3:** 3(0,1,0) 4(1,1,0) 7(1,2,0) 6(0,2,0) 12(0,1,1)
- **Пирамида 4:** 4(1,1,0) 5(2,1,0) 8(2,2,0) 7(1,2,0) 13(1,1,1)
- **Тетраэдр 1:** 1(1,0,0) 4(1,1,0) 9(0,0,1) 10(1,0,1)
- **Тетраэдр 2:** 4(1,1,0) 3(0,1,0) 9(0,0,1) 12(0,1,1)
- **Тетраэдр 3:** 4(1,1,0) 9(0,0,1) 10(1,0,1) 12(0,1,1)
- **Тетраэдр 4:** 10(1,0,1) 4(1,1,0) 12(0,1,1) 13(1,1,1)

- **Тетраэдр 5:** 2(2,0,0) 5(2,1,0) 10(1,0,1) 11(2,0,1)
- **Тетраэдр 6:** 5(2,1,0) 4(1,1,0) 10(1,0,1) 13(1,1,1)
- **Тетраэдр 7:** 5(2,1,0) 10(1,0,1) 11(2,0,1) 13(1,1,1)
- **Тетраэдр 8:** 11(2,0,1) 5(2,1,0) 13(1,1,1) 14(2,1,1)
- **Тетраэдр 9:** 4(1,1,0) 7(1,2,0) 12(0,1,1) 13(1,1,1)
- **Тетраэдр 10:** 7(1,2,0) 6(0,2,0) 12(0,1,1) 15(0,2,1)
- **Тетраэдр 11:** 7(1,2,0) 12(0,1,1) 13(1,1,1) 15(0,2,1)
- **Тетраэдр 12:** 13(1,1,1) 7(1,2,0) 15(0,2,1) 16(1,2,1)
- **Тетраэдр 13:** 5(2,1,0) 8(2,2,0) 13(1,1,1) 14(2,1,1)
- **Тетраэдр 14:** 8(2,2,0) 7(1,2,0) 13(1,1,1) 16(1,2,1)
- **Тетраэдр 15:** 8(2,2,0) 13(1,1,1) 14(2,1,1) 16(1,2,1)
- **Тетраэдр 16:** 14(2,1,1) 8(2,2,0) 16(1,2,1) 17(2,2,1)

Результат разбиения представлены на рисунке 5.

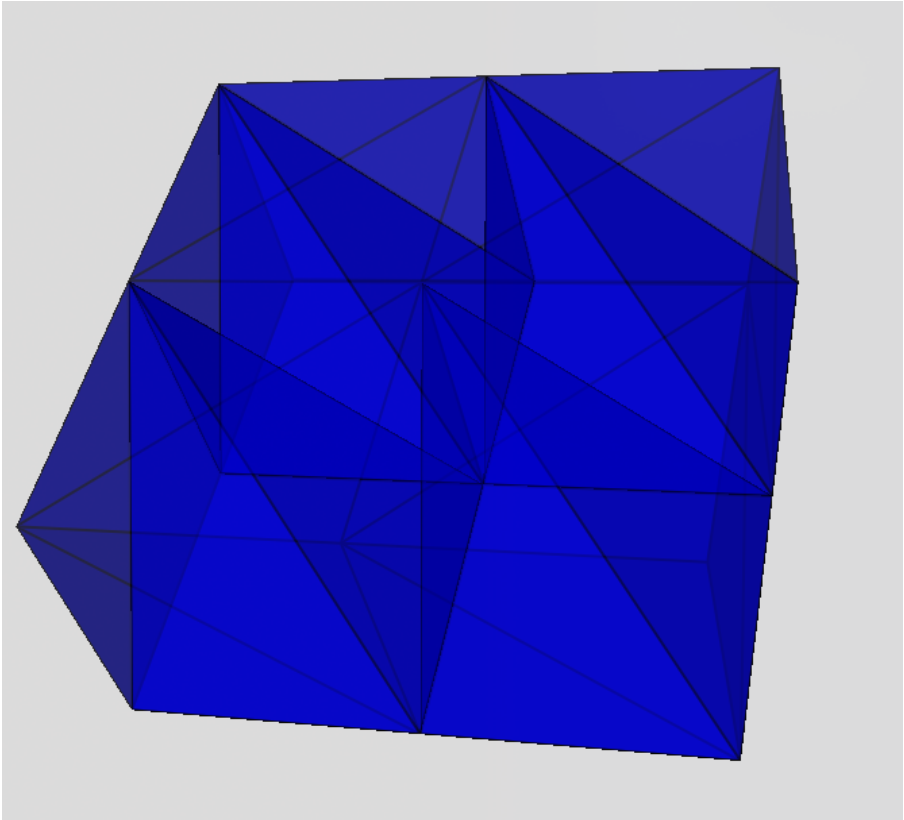

Рисунок 5 – Результат разбиения четырех шестигранников

2.4. ТЕСТИРОВАНИЕ НА 4 ШЕСТИГРАННИКАХ

Входные данные:

- global_node_coordinates:

  0.0 0.0 0.0
  1.0 0.0 0.0
  2.0 0.0 0.0
  0.0 1.0 0.0
  1.0 1.0 0.0
  2.0 1.0 0.0
  0.0 2.0 0.0
  1.0 2.0 0.0
  2.0 2.0 0.0
  0.0 0.0 1.0
  1.0 0.0 1.0
  2.0 0.0 1.0
  0.0 1.0 1.0
  1.0 1.0 1.0
  2.0 1.0 1.0
  0.0 2.0 1.0
  1.0 2.0 1.0
  2.0 2.0 1.0
  0.0 0.0 2.0
  1.0 0.0 2.0
  2.0 0.0 2.0
  0.0 1.0 2.0
  1.0 1.0 2.0
  2.0 1.0 2.0
  0.0 2.0 2.0
  1.0 2.0 2.0
  2.0 2.0 2.0

- global_nodes_indicies:

  0 1 3 4 9 10 12 13
  1 2 4 5 10 11 13 14
  3 4 6 7 12 13 15 16

4 5 7 8 13 14 16 17

9 10 12 13 18 19 21 22

10 11 13 14 19 20 22 23

12 13 15 16 21 22 24 25

13 14 16 17 22 23 25 26

Выходные данные:

- **Пирамида 1:** 0(0,0,0) 1(1,0,0) 4(1,1,0) 3(0,1,0) 9(0,0,1)
- **Пирамида 2:** 1(1,0,0) 2(2,0,0) 5(2,1,0) 4(1,1,0) 10(1,0,1)
- **Пирамида 3:** 3(0,1,0) 4(1,1,0) 7(1,2,0) 6(0,2,0) 12(0,1,1)
- **Пирамида 4:** 4(1,1,0) 5(2,1,0) 8(2,2,0) 7(1,2,0) 13(1,1,1)
- **Пирамида 5:** 9(0,0,1) 10(1,0,1) 13(1,1,1) 12(0,1,1) 18(0,0,2)
- **Пирамида 6:** 10(1,0,1) 11(2,0,1) 14(2,1,1) 13(1,1,1) 19(1,0,2)
- **Пирамида 7:** 12(0,1,1) 13(1,1,1) 16(1,2,1) 15(0,2,1) 21(0,1,2)
- **Пирамида 8:** 13(1,1,1) 14(2,1,1) 17(2,2,1) 16(1,2,1) 22(1,1,2)
- **Тетраэдр 1:** 1(1,0,0) 4(1,1,0) 9(0,0,1) 10(1,0,1)
- **Тетраэдр 2:** 4(1,1,0) 3(0,1,0) 9(0,0,1) 12(0,1,1)
- **Тетраэдр 3:** 4(1,1,0) 9(0,0,1) 10(1,0,1) 12(0,1,1)
- **Тетраэдр 4:** 10(1,0,1) 4(1,1,0) 12(0,1,1) 13(1,1,1)
- **Тетраэдр 5:** 2(2,0,0) 5(2,1,0) 10(1,0,1) 11(2,0,1)
- **Тетраэдр 6:** 5(2,1,0) 4(1,1,0) 10(1,0,1) 13(1,1,1)
- **Тетраэдр 7:** 5(2,1,0) 10(1,0,1) 11(2,0,1) 13(1,1,1)
- **Тетраэдр 8:** 11(2,0,1) 5(2,1,0) 13(1,1,1) 14(2,1,1)
- **Тетраэдр 9:** 4(1,1,0) 7(1,2,0) 12(0,1,1) 13(1,1,1)
- **Тетраэдр 10:** 7(1,2,0) 6(0,2,0) 12(0,1,1) 15(0,2,1)
- **Тетраэдр 11:** 7(1,2,0) 12(0,1,1) 13(1,1,1) 15(0,2,1)
- **Тетраэдр 12:** 13(1,1,1) 7(1,2,0) 15(0,2,1) 16(1,2,1)
- **Тетраэдр 13:** 5(2,1,0) 8(2,2,0) 13(1,1,1) 14(2,1,1)
- **Тетраэдр 14:** 8(2,2,0) 7(1,2,0) 13(1,1,1) 16(1,2,1)
- **Тетраэдр 15:** 8(2,2,0) 13(1,1,1) 14(2,1,1) 16(1,2,1)
- **Тетраэдр 16:** 14(2,1,1) 8(2,2,0) 16(1,2,1) 17(2,2,1)
- **Тетраэдр 17:** 10(1,0,1) 13(1,1,1) 18(0,0,2) 19(1,0,2)
- **Тетраэдр 18:** 13(1,1,1) 12(0,1,1) 18(0,0,2) 21(0,1,2)
- **Тетраэдр 19:** 13(1,1,1) 18(0,0,2) 19(1,0,2) 21(0,1,2)
- **Тетраэдр 20:** 19(1,0,2) 13(1,1,1) 21(0,1,2) 22(1,1,2)

- **Тетраэдр 21:** 11(2,0,1) 14(2,1,1) 19(1,0,2) 20(2,0,2)
- **Тетраэдр 22:** 14(2,1,1) 13(1,1,1) 19(1,0,2) 22(1,1,2)
- **Тетраэдр 23:** 14(2,1,1) 19(1,0,2) 20(2,0,2) 22(1,1,2)
- **Тетраэдр 24:** 20(2,0,2) 14(2,1,1) 22(1,1,2) 23(2,1,2)
- **Тетраэдр 25:** 13(1,1,1) 16(1,2,1) 21(0,1,2) 22(1,1,2)
- **Тетраэдр 26:** 16(1,2,1) 15(0,2,1) 21(0,1,2) 24(0,2,2)
- **Тетраэдр 27:** 16(1,2,1) 21(0,1,2) 22(1,1,2) 24(0,2,2)
- **Тетраэдр 28:** 22(1,1,2) 16(1,2,1) 24(0,2,2) 25(1,2,2)
- **Тетраэдр 29:** 14(2,1,1) 17(2,2,1) 22(1,1,2) 23(2,1,2)
- **Тетраэдр 30:** 17(2,2,1) 16(1,2,1) 22(1,1,2) 25(1,2,2)
- **Тетраэдр 31:** 17(2,2,1) 22(1,1,2) 23(2,1,2) 25(1,2,2)
- **Тетраэдр 32:** 23(2,1,2) 17(2,2,1) 25(1,2,2) 26(2,2,2)

Результат разбиения представлены на рисунке 6.

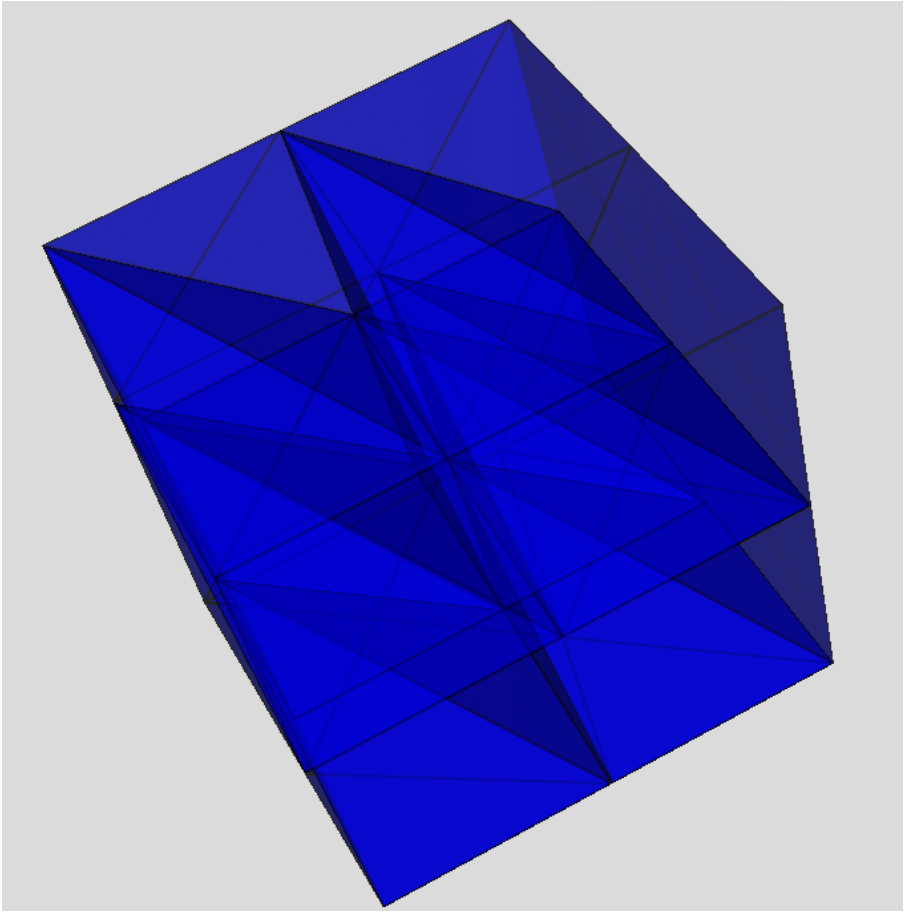

Рисунок 6 – Результат разбиения восьми шестигранников

3. ЗАКЛЮЧЕНИЕ

В результате практической работы был изучен метод визуализации данных на C++ через VTK. Реализована программа, успешно разбивающая шестигранники на пирамиду и тетраэдры по заданному шаблону.

СПИСОК ЛИТЕРАТУРЫ

1. Метод конечных элементов для решения скалярных и векторных задач : учеб. Пособие / Ю.Г. Соловейчик, М.Э. Рояк, М.Г. Персова – Новосибирск: Изд-во НГТУ, 2007.– 896 с. ("Учебники НГТУ").

2. Зенкевич О., Морган К. Конечные элементы и аппроксимация. – М.: Мир, 1986.

3. Schroeder W., Martin K., Lorensen B. The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics. Edition 4.1. July 2018.

ПРИЛОЖЕНИЕ А. ТЕКСТ ПРОГРАММЫ

Листинг 1: main.cpp

```
#include <array>
#include <fstream>
#include <iostream>
#include <vector>

#include "../include/grid.h"

constexpr auto GLOBAL_NODE_INDICIES_INPUT_FILE_NAME =
    std::to_array("./tests/5_test/global_node_indicies.txt");

constexpr auto GLOBAL_NODE_COORDINATES_INPUT_FILE_NAME =
    std::to_array("./tests/5_test/global_node_coordinates.txt");

int main() {
    Grid grid;

    std::ifstream global_indicies_input_file(
        GLOBAL_NODE_INDICIES_INPUT_FILE_NAME.data());

    int count_node_indicies = 0;
    int count_node_coordinates = 0;

    global_indicies_input_file >> count_node_indicies;
    std::vector<std::array<int, 8>> global_nodes_indicies(count_node_indicies /
                                                          8);

    for (int i = 0; i < count_node_indicies / 8; i++) {
        std::array<int, 8> temp{};
#pragma unroll 4
        for (int j = 0; j < 8; j++) {
            global_indicies_input_file >> temp.at(j);
        }
        global_nodes_indicies[i] = temp;
    }

    global_indicies_input_file.close();

    std::ifstream global_coordinates_input_file(
        GLOBAL_NODE_COORDINATES_INPUT_FILE_NAME.data());

    global_coordinates_input_file >> count_node_coordinates;

    std::map<int, std::vector<double>> global_node_coordinates;

#pragma unroll 4
    for (int i = 0; i < count_node_coordinates; i++) {
        double x = 0.0;
        double y = 0.0;
        double z = 0.0;
        global_coordinates_input_file >> x >> y >> z;
        global_node_coordinates[i] = {x, y, z};
    }
```

```
    global_coordinates_input_file.close();

    std::vector<int> fixed_face_local_indicies = {0, 1, 2, 3};

#pragma unroll 4
    for (auto& global_nodes_index : global_nodes_indicies) {
        std::vector<int> nodes8(8);

#pragma unroll 4
        for (int k = 0; k < 8; ++k) {
            nodes8[k] = global_nodes_index.at(k);
        }

        std::vector<int> fixed_face_local_indicies = {nodes8[0], nodes8[1],
                                                      nodes8[2], nodes8[3]};

        grid.splitHexAnyFixedFace(nodes8, global_node_coordinates,
                                  fixed_face_local_indicies);
    }

    // grid.printElements();
    grid.visualizeSplittedGrid(global_node_coordinates);
    return 0;
}
```

Листинг 2: grid.h

```
#ifndef GRID_H
#define GRID_H

#include <map>
#include <vector>

#include "../include/visualize_split.h"

class Grid {
   private:
    VTKVisualizer visualizer;

    std::vector<std::vector<int>> tetrahedron_indicies;

    std::vector<std::vector<int>> pyramide_indicies;

   public:
    static std::vector<int> buildLocalMappingForFixedFace(
        const std::vector<int>& global_nodes_indicies,
        const std::map<int, std::vector<double>>& global_nodes_coordinates,
        const std::vector<int>& fixed_base_global_indicies);

    static std::vector<std::vector<int>> applyTemplate(
        const std::vector<int>& local_mapping);

    static void ensurePositiveTets(
        std::vector<std::vector<int>>& elems,
        const std::map<int, std::vector<double>>& global_nodes_coordinates);

    void splitHexAnyFixedFace(
        const std::vector<int>& global_nodes_indicies,
        const std::map<int, std::vector<double>>& global_nodes_coordinates,
```

```
        const std::vector<int>& fixed_base_global_indicies);

    void printElements();

    void visualizeSplittedGrid(
        const std::map<int, std::vector<double>>& global_nodes_coordinates);
};

#endif
```

Листинг 3: math_utils.h

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

#include <vector>

class MathHelper {
   public:
    static double calculateScalarProductOfVectors(
        const std::vector<double>& vec1, const std::vector<double>& vec2);

    static std::vector<double> subtractVectors(const std::vector<double>& vec1,
                                               const std::vector<double>& vec2);

    static std::vector<double> additionVectors(const std::vector<double>& vec1,
                                               const std::vector<double>& vec2);

    static std::vector<double> multiplyVectorByMultiplier(
        const std::vector<double>& vec1, double multiplier);

    static std::vector<double> calculateCrossProductOfVectors(
        const std::vector<double>& vec1, const std::vector<double>& vec2);

    static double calculateNormOfVector(const std::vector<double>& vec1);

    static std::vector<double> calculateNormalizedVector(
        const std::vector<double>& vec1);

    static double calculateOrientedTetrahedronVolume6(
        const std::vector<double>& a, const std::vector<double>& b,
        const std::vector<double>& c, const std::vector<double>& d);
};

#endif
```

Листинг 4: visualize_split.h

```
#ifndef VISUALIZE_SPLIT_H
#define VISUALIZE_SPLIT_H

#include <vtkActor.h>
#include <vtkCamera.h>
#include <vtkCellData.h>
#include <vtkDataSetMapper.h>
#include <vtkDataSetSurfaceFilter.h>
#include <vtkExtractEdges.h>
#include <vtkIntArray.h>
```

```
#include <vtkLookupTable.h>
#include <vtkPoints.h>
#include <vtkPolyDataMapper.h>
#include <vtkProperty.h>
#include <vtkPyramid.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderer.h>
#include <vtkSmartPointer.h>
#include <vtkTetra.h>
#include <vtkThreshold.h>
#include <vtkUnstructuredGrid.h>
#include <vtkXMLUnstructuredGridWriter.h>

#include <map>
#include <vector>

class VTKVisualizer {
   public:
    static std::array<int, 4> orderPyramidBase(
        const std::array<int, 4>& base,
        const std::map<int, std::vector<double>>& coords);

    static std::array<int, 4> orderQuadPerPlane(
        const std::array<int, 4>& quad,
        const std::map<int, std::vector<double>>& coords);

    static void dumpUgridInfo(vtkUnstructuredGrid* ugrid);

    static vtkSmartPointer<vtkUnstructuredGrid> buildVTKGrid(
        const std::map<int, std::vector<double>>& global_node_coordinates,
        const std::vector<std::vector<int>>& pyramide_indicies,
        const std::vector<std::vector<int>>& tetrahedron_indicies);
};

#endif
```

Листинг 5: grid.cpp

```
#include "../include/grid.h"

#include <algorithm>
#include <array>
#include <cstdlib>
#include <iostream>
#include <set>
#include <vector>

#include "../include/math_utils.h"

static constexpr std::array<int, 5> PYRAMID_TEMPLATE = {0, 1, 2, 3, 4};

static constexpr std::array<std::array<int, 4>, 4> TETRAHEDRON_TEMPLATE{
    {{{1, 3, 4, 5}}, {{2, 3, 4, 6}}, {{3, 4, 5, 6}}, {{3, 5, 6, 7}}}};

constexpr double EPS = 1e-9;

std::vector<int> Grid::buildLocalMappingForFixedFace(
    const std::vector<int>& global_nodes_indicies,
```

```
    const std::map<int, std::vector<double>>& global_nodes_coordinates,
    const std::vector<int>& fixed_base_global_indicies) {
    std::vector<std::pair<int, std::vector<double>>> base;
    base.reserve(4);

    std::set<int> base_set(fixed_base_global_indicies.begin(),
                           fixed_base_global_indicies.end());

#pragma unroll 4
    for (int g : fixed_base_global_indicies) {
        auto it = global_nodes_coordinates.find(g);
        base.emplace_back(g, it->second);
    }

    std::vector<double> a = base[0].second;
    std::vector<double> b = base[1].second;
    std::vector<double> c = base[2].second;

    std::vector<double> n = MathHelper::calculateCrossProductOfVectors(
        MathHelper::subtractVectors(b, a), MathHelper::subtractVectors(c, a));

    if (MathHelper::calculateNormOfVector(n) < EPS) {
        c = base[3].second;
        n = MathHelper::calculateCrossProductOfVectors(
            MathHelper::subtractVectors(b, a),
            MathHelper::subtractVectors(c, a));
        if (MathHelper::calculateNormOfVector(n) < EPS) {
            n = {0, 0, 0};
#pragma unroll 4
            for (int i = 0; i < 4; ++i) {
                std::vector<double> p0 = base[i].second;
                std::vector<double> p1 = base[(i + 1) % 4].second;
                std::vector<double> p2 = base[(i + 2) % 4].second;
                std::vector<double> c =
                    MathHelper::calculateCrossProductOfVectors(
                        MathHelper::subtractVectors(p1, p0),
                        MathHelper::subtractVectors(p2, p0));
                n = MathHelper::additionVectors(n, c);
            }
        }
    }
    n = MathHelper::calculateNormalizedVector(n);

    std::vector<double> u =
        MathHelper::subtractVectors(base[1].second, base[0].second);
    if (MathHelper::calculateNormOfVector(u) < EPS) {
        u = MathHelper::subtractVectors(base[2].second, base[0].second);
    }
    u = MathHelper::calculateNormalizedVector(u);
    double proj = MathHelper::calculateScalarProductOfVectors(u, n);
    u = MathHelper::subtractVectors(
        u, MathHelper::multiplyVectorByMultiplier(n, proj));
    u = MathHelper::calculateNormalizedVector(u);
    std::vector<double> v = MathHelper::calculateCrossProductOfVectors(n, u);
    v = MathHelper::calculateNormalizedVector(v);

    std::vector<double> centroid = {0, 0, 0};
#pragma unroll 4
    for (auto& p : base) {
        centroid = MathHelper::additionVectors(centroid, p.second);
```

```
    }
    centroid = MathHelper::multiplyVectorByMultiplier(centroid, 1.0 / 4.0);

    struct Item {
        int g;
        double uc;
        double vc;
        std::vector<double> pos;
    };
    std::vector<Item> items;
    items.reserve(4);
#pragma unroll 4
    for (auto& p : base) {
        std::vector<double> r = MathHelper::subtractVectors(p.second, centroid);
        double uc = MathHelper::calculateScalarProductOfVectors(r, u);
        double vc = MathHelper::calculateScalarProductOfVectors(r, v);
        items.push_back({p.first, uc, vc, p.second});
    }

    std::ranges::sort(items, [](const Item& a, const Item& b) {
        if (std::abs(a.vc - b.vc) > 1e-8) {
            return a.vc < b.vc;
        }
        return a.uc < b.uc;
    });

    std::vector<int> others;
    others.reserve(4);
#pragma unroll 4
    for (int i = 0; i < 8; ++i) {
        int g = global_nodes_indicies[i];
        if (!base_set.contains(g)) {
            others.push_back(g);
        }
    }

    auto find_opposite = [&](const std::vector<double>& base_pos) -> int {
        double best = -1e308;
        int bestg = -1;
#pragma unroll 4
        for (int g : others) {
            std::vector<double> r = MathHelper::subtractVectors(
                global_nodes_coordinates.at(g), base_pos);
            double val = MathHelper::calculateScalarProductOfVectors(r, n);
            if (val > best) {
                best = val;
                bestg = g;
            }
        }
        return bestg;
    };

    std::vector<int> used_opp(others.size(), 0);
    std::vector<int> local(8);
#pragma unroll 4
    for (int i = 0; i < 4; ++i) {
        local[i] = items[i].g;
    }

    for (int i = 0; i < 4; ++i) {
```

```
        const std::vector<double>& base_pos =
            global_nodes_coordinates.at(local[i]);
        double best = -1e308;
        int bestj = -1;
#pragma unroll 4
        for (size_t j = 0; j < others.size(); ++j) {
            if (used_opp[j] != 0) {
                continue;
            }
            std::vector<double> r = MathHelper::subtractVectors(
                global_nodes_coordinates.at(others[j]), base_pos);
            double val = MathHelper::calculateScalarProductOfVectors(r, n);
            if (val > best) {
                best = val;
                bestj = static_cast<int>(j);
            }
        }
        used_opp[bestj] = 1;
        local[4 + i] = others[bestj];
    }

    return local;
}

std::vector<std::vector<int>> Grid::applyTemplate(
    const std::vector<int>& local_mapping) {
    std::vector<std::vector<int>> elems;

    elems.reserve(5);
    elems.emplace_back();

#pragma unroll 4
    for (int k = 0; k < 5; ++k) {
        elems.back().push_back(local_mapping[PYRAMID_TEMPLATE.at(k)]);
    }

    for (int t = 0; t < 4; ++t) {
        elems.emplace_back();
#pragma unroll 4
        for (int k = 0; k < 4; ++k) {
            elems.back().push_back(
                local_mapping[TETRAHEDRON_TEMPLATE.at(t).at(k)]);
        }
    }

    return elems;
}

void Grid::ensurePositiveTets(
    std::vector<std::vector<int>>& elems,
    const std::map<int, std::vector<double>>& global_nodes_coordinates) {
#pragma unroll 4
    for (size_t e = 1; e < elems.size(); ++e) {
        if (elems[e].size() != 4) {
            continue;
        }
        const std::vector<double>& a = global_nodes_coordinates.at(elems[e][0]);
        const std::vector<double>& b = global_nodes_coordinates.at(elems[e][1]);
        const std::vector<double>& c = global_nodes_coordinates.at(elems[e][2]);
        const std::vector<double>& d = global_nodes_coordinates.at(elems[e][3]);
```

```
        double s6 = MathHelper::calculateOrientedTetrahedronVolume6(a, b, c, d);
        if (s6 < 0.0) {
            std::swap(elems[e][0], elems[e][1]);
        }
    }
}

void Grid::splitHexAnyFixedFace(
    const std::vector<int>& global_nodes_indicies,
    const std::map<int, std::vector<double>>& global_nodes_coordinates,
    const std::vector<int>& fixed_base_global_indicies) {
    auto local_mapping = buildLocalMappingForFixedFace(
        global_nodes_indicies, global_nodes_coordinates,
        fixed_base_global_indicies);

    auto elems = applyTemplate(local_mapping);

    ensurePositiveTets(elems, global_nodes_coordinates);

    pyramide_indicies.push_back(elems[0]);

#pragma unroll 4
    for (size_t i = 1; i < elems.size(); i++) {
        tetrahedron_indicies.push_back(elems[i]);
    }
}

void Grid::printElements() {
    std::cout << "Pyramids: \n";

    for (const auto& pyramide_index : pyramide_indicies) {
        std::cout << "\t[";
#pragma unroll 4
        for (size_t i = 0; i < pyramide_index.size(); ++i) {
            if (i != 0U) {
                std::cout << ", ";
            }
            std::cout << pyramide_index[i];
        }
        std::cout << "]\n";
    }

    std::cout << "Tetrahedrons: \n";

    for (const auto& tetrahedron_index : tetrahedron_indicies) {
        std::cout << "\t[";
#pragma unroll 4
        for (size_t i = 0; i < tetrahedron_index.size(); ++i) {
            if (i != 0U) {
                std::cout << ", ";
            }
            std::cout << tetrahedron_index[i];
        }
        std::cout << "]\n";
    }
}

void Grid::visualizeSplittedGrid(
    const std::map<int, std::vector<double>>& global_nodes_coordinates) {
    auto ugrid = VTKVisualizer::buildVTKGrid(
```

```
    global_nodes_coordinates, pyramide_indicies, tetrahedron_indicies);

auto renderer = vtkSmartPointer<vtkRenderer>::New();
renderer->SetBackground(0.9, 0.9, 0.9);

auto pyr_filter = vtkSmartPointer<vtkThreshold>::New();
pyr_filter->SetInputData(ugrid);
pyr_filter->SetInputArrayToProcess(
    0, 0, 0, vtkDataObject::FIELD_ASSOCIATION_CELLS, "elemType");
pyr_filter->SetLowerThreshold(1);
pyr_filter->SetUpperThreshold(1);
pyr_filter->Update();

auto pyr_surface = vtkSmartPointer<vtkDataSetSurfaceFilter>::New();
pyr_surface->SetInputConnection(pyr_filter->GetOutputPort());
pyr_surface->Update();

auto pyr_mapper = vtkSmartPointer<vtkDataSetMapper>::New();
pyr_mapper->SetInputConnection(pyr_surface->GetOutputPort());

auto pyr_actor = vtkSmartPointer<vtkActor>::New();
pyr_actor->SetMapper(pyr_mapper);

pyr_actor->GetProperty()->SetColor(1.0, 0.6, 0.2);
pyr_actor->GetProperty()->SetEdgeColor(0.0, 0.0, 0.0);
pyr_actor->GetProperty()->SetOpacity(0.6);
pyr_actor->GetProperty()->EdgeVisibilityOn();
pyr_actor->GetProperty()->SetLineWidth(2.0);

renderer->AddActor(pyr_actor);

auto tet_filter = vtkSmartPointer<vtkThreshold>::New();
tet_filter->SetInputData(ugrid);
tet_filter->SetInputArrayToProcess(
    0, 0, 0, vtkDataObject::FIELD_ASSOCIATION_CELLS, "elemType");
tet_filter->SetLowerThreshold(2);
tet_filter->SetUpperThreshold(2);
tet_filter->Update();

auto tet_surface = vtkSmartPointer<vtkDataSetSurfaceFilter>::New();
tet_surface->SetInputConnection(tet_filter->GetOutputPort());
tet_surface->Update();

auto tet_mapper = vtkSmartPointer<vtkDataSetMapper>::New();
tet_mapper->SetInputConnection(tet_surface->GetOutputPort());

auto tet_actor = vtkSmartPointer<vtkActor>::New();
tet_actor->SetMapper(tet_mapper);
tet_actor->GetProperty()->SetColor(0.2, 1.0, 0.2);
tet_actor->GetProperty()->SetEdgeColor(0.0, 0.0, 0.0);
tet_actor->GetProperty()->SetOpacity(0.6);
tet_actor->GetProperty()->EdgeVisibilityOn();
tet_actor->GetProperty()->SetLineWidth(2.0);

renderer->AddActor(tet_actor);

auto ren_win = vtkSmartPointer<vtkRenderWindow>::New();
ren_win->AddRenderer(renderer);
ren_win->SetSize(900, 700);
ren_win->SetWindowName("Hex split visualization");
```

```
    auto iren = vtkSmartPointer<vtkRenderWindowInteractor>::New();
    iren->SetRenderWindow(ren_win);

    ren_win->Render();
    iren->Start();
}
```

Листинг 6: math_utils.cpp

```
#include "../include/math_utils.h"

#include <cmath>

double MathHelper::calculateScalarProductOfVectors(
    const std::vector<double>& vec1, const std::vector<double>& vec2) {
    double result = 0.0;

#pragma unroll 4
    for (int i = 0; i < vec1.size(); i++) {
        result += vec1[i] * vec2[i];
    }

    return result;
}

std::vector<double> MathHelper::subtractVectors(
    const std::vector<double>& vec1, const std::vector<double>& vec2) {
    std::vector<double> result(vec1.size());

#pragma unroll 4
    for (size_t i = 0; i < vec1.size(); ++i) {
        result[i] = vec1[i] - vec2[i];
    }

    return result;
}

std::vector<double> MathHelper::additionVectors(
    const std::vector<double>& vec1, const std::vector<double>& vec2) {
    std::vector<double> result(vec1.size());

#pragma unroll 4
    for (size_t i = 0; i < vec1.size(); ++i) {
        result[i] = vec1[i] + vec2[i];
    }

    return result;
}

std::vector<double> MathHelper::multiplyVectorByMultiplier(
    const std::vector<double>& vec1, double multiplier) {
    std::vector<double> result(vec1.size());

#pragma unroll 4
    for (size_t i = 0; i < vec1.size(); ++i) {
        result[i] = vec1[i] * multiplier;
    }
```

```
    return result;
}

std::vector<double> MathHelper::calculateCrossProductOfVectors(
    const std::vector<double>& vec1, const std::vector<double>& vec2) {
    return {(vec1[1] * vec2[2]) - (vec1[2] * vec2[1]),
            (vec1[2] * vec2[0]) - (vec1[0] * vec2[2]),
            (vec1[0] * vec2[1]) - (vec1[1] * vec2[0])};
}

double MathHelper::calculateNormOfVector(const std::vector<double>& vec1) {
    return std::sqrt(calculateScalarProductOfVectors(vec1, vec1));
}

std::vector<double> MathHelper::calculateNormalizedVector(
    const std::vector<double>& vec1) {
    double norm_of_vector = calculateNormOfVector(vec1);
    std::vector<double> result(vec1.size());

#pragma unroll 4
    for (size_t i = 0; i < vec1.size(); ++i) {
        result[i] = vec1[i] / norm_of_vector;
    }

    return result;
}

double MathHelper::calculateOrientedTetrahedronVolume6(
    const std::vector<double>& a, const std::vector<double>& b,
    const std::vector<double>& c, const std::vector<double>& d) {
    std::vector<double> u = subtractVectors(b, a);
    std::vector<double> v = subtractVectors(c, a);
    std::vector<double> w = subtractVectors(d, a);

    return (u[0] * (v[1] * w[2] - v[2] * w[1])) -
           (u[1] * (v[0] * w[2] - v[2] * w[0])) +
           (u[2] * (v[0] * w[1] - v[1] * w[0]));
}
```

Листинг 7: visualize_split.cpp

```
#include "../include/visualize_split.h"

#include <algorithm>
#include <set>
#include <unordered_map>

std::array<int, 4> VTKVisualizer::orderPyramidBase(
    const std::array<int, 4>& base,
    const std::map<int, std::vector<double>>& coords) {
    double cx = 0;
    double cy = 0;

#pragma unroll 4
    for (int i : base) {
        cx += coords.at(i)[0];
        cy += coords.at(i)[1];
    }
    cx /= 4.0;
```

```
    cy /= 4.0;

    struct Item {
        int id;
        double ang;
    };

    std::array<Item, 4> items{};

#pragma unroll 4
    for (int i = 0; i < 4; ++i) {
        const auto& p = coords.at(base.at(i));
        items.at(i) = {.id = base.at(i),
                       .ang = std::atan2(p[1] - cy, p[0] - cx)};
    }

    std::ranges::sort(items, [](auto& a, auto& b) { return a.ang < b.ang; });

    return {items[0].id, items[1].id, items[2].id, items[3].id};
}

std::array<int, 4> VTKVisualizer::orderQuadPerPlane(
    const std::array<int, 4>& quad,
    const std::map<int, std::vector<double>>& coords) {
    std::array<double, 3> centroid{0, 0, 0};

#pragma unroll 4
    for (int i = 0; i < 4; ++i) {
        const auto& c = coords.at(quad.at(i));
        centroid[0] += c[0];
        centroid[1] += c[1];
        centroid[2] += c[2];
    }

#pragma unroll 4
    for (int k = 0; k < 3; ++k) {
        centroid.at(k) *= 0.25;
    }

    const auto& p0 = coords.at(quad[0]);
    const auto& p1 = coords.at(quad[1]);
    const auto& p2 = coords.at(quad[2]);

    std::array<double, 3> u{p1[0] - p0[0], p1[1] - p0[1], p1[2] - p0[2]};
    std::array<double, 3> v{p2[0] - p0[0], p2[1] - p0[1], p2[2] - p0[2]};

    std::array<double, 3> n{(u[1] * v[2]) - (u[2] * v[1]),
                            (u[2] * v[0]) - (u[0] * v[2]),
                            (u[0] * v[1]) - (u[1] * v[0])};

    double nn = std::sqrt((n[0] * n[0]) + (n[1] * n[1]) + (n[2] * n[2]));

    if (nn < 1e-12) {
        return orderPyramidBase(quad, coords);
    }

#pragma unroll 4
    for (int k = 0; k < 3; ++k) {
        n.at(k) /= nn;
    }
```

```
    std::array<double, 3> ux = u;
    double proj = (ux[0] * n[0]) + (ux[1] * n[1]) + (ux[2] * n[2]);
    ux = {ux[0] - (proj * n[0]), ux[1] - (proj * n[1]), ux[2] - (proj * n[2])};
    double ulen =
        std::sqrt((ux[0] * ux[0]) + (ux[1] * ux[1]) + (ux[2] * ux[2]));

    if (ulen < 1e-12) {
        return orderPyramidBase(quad, coords);
    }

#pragma unroll 4
    for (int k = 0; k < 3; ++k) {
        ux.at(k) /= ulen;
    }

    std::array<double, 3> uy = {(n[1] * ux[2]) - (n[2] * ux[1]),
                                (n[2] * ux[0]) - (n[0] * ux[2]),
                                (n[0] * ux[1]) - (n[1] * ux[0])};

    struct Item {
        int g;
        double ang;
        double r2;
    };

    std::array<Item, 4> items{};

#pragma unroll 4
    for (int i = 0; i < 4; ++i) {
        const auto& p = coords.at(quad.at(i));
        std::array<double, 3> r{p[0] - centroid[0], p[1] - centroid[1],
                                p[2] - centroid[2]};

        double uxv = (r[0] * ux[0]) + (r[1] * ux[1]) + (r[2] * ux[2]);
        double uyv = (r[0] * uy[0]) + (r[1] * uy[1]) + (r[2] * uy[2]);

        items.at(i) = {.g = quad.at(i),
                       .ang = std::atan2(uyv, uxv),
                       .r2 = (uxv * uxv) + (uyv * uyv)};
    }

    std::ranges::sort(items, [](const Item& a, const Item& b) {
        if (std::abs(a.ang - b.ang) > 1e-9) {
            return a.ang < b.ang;
        }
        return a.r2 < b.r2;
    });

    return {items[0].g, items[1].g, items[2].g, items[3].g};
}

void VTKVisualizer::dumpUgridInfo(vtkUnstructuredGrid* ugrid) {
    std::cout << "UGRID: points=" << ugrid->GetNumberOfPoints()
              << " cells=" << ugrid->GetNumberOfCells() << "\n";

    for (vtkIdType cid = 0; cid < ugrid->GetNumberOfCells(); ++cid) {
        vtkCell* cell = ugrid->GetCell(cid);
        int ctype = cell->GetCellType();
        std::cout << "Cell " << cid << " type=" << ctype
```

```
                << " npts=" << cell->GetNumberOfPoints() << " pts: ";

#pragma unroll 4
        for (vtkIdType i = 0; i < cell->GetNumberOfPoints(); ++i) {
            vtkIdType pid = cell->GetPointId(i);
            std::array<double, 3> p{};
            ugrid->GetPoint(pid, p.data());
            std::cout << pid << "(" << p[0] << "," << p[1] << "," << p[2]
                      << ") ";
        }
        std::cout << "\n";
    }
}

vtkSmartPointer<vtkUnstructuredGrid> VTKVisualizer::buildVTKGrid(
    const std::map<int, std::vector<double>>& global_node_coordinates,
    const std::vector<std::vector<int>>& pyramide_indicies,
    const std::vector<std::vector<int>>& tetrahedron_indicies) {
    std::set<int> used_indices;

    for (const auto& cell : pyramide_indicies) {
#pragma unroll 4
        for (int gi : cell) {
            used_indices.insert(gi);
        }
    }

    for (const auto& cell : tetrahedron_indicies) {
#pragma unroll 4
        for (int gi : cell) {
            used_indices.insert(gi);
        }
    }

    auto points = vtkSmartPointer<vtkPoints>::New();
    std::unordered_map<int, vtkIdType> global_to_vtk_id;
    global_to_vtk_id.reserve(used_indices.size());

#pragma unroll 4
    for (int gi : used_indices) {
        auto it = global_node_coordinates.find(gi);

        const std::vector<double>& c = it->second;
        vtkIdType pid = points->InsertNextPoint(c[0], c[1], c[2]);
        global_to_vtk_id[gi] = pid;
    }

    auto ugrid = vtkSmartPointer<vtkUnstructuredGrid>::New();
    ugrid->SetPoints(points);

    auto elem_type = vtkSmartPointer<vtkIntArray>::New();
    elem_type->SetName("elemType");

    for (const auto& cell : pyramide_indicies) {
        auto pyr = vtkSmartPointer<vtkPyramid>::New();

        std::array<int, 4> base = {cell[0], cell[1], cell[2], cell[3]};

        auto ordered = orderQuadPerPlane(base, global_node_coordinates);
```

```
#pragma unroll 4
        for (int i = 0; i < 4; ++i) {
            pyr->GetPointIds()->SetId(i, global_to_vtk_id.at(ordered.at(i)));
        }

        pyr->GetPointIds()->SetId(4, global_to_vtk_id.at(cell[4]));

        ugrid->InsertNextCell(pyr->GetCellType(), pyr->GetPointIds());
        elem_type->InsertNextValue(1);
    }

    for (const auto& cell : tetrahedron_indicies) {
        auto tet = vtkSmartPointer<vtkTetra>::New();
#pragma unroll 4
        for (int i = 0; i < 4; ++i) {
            int g = cell[i];
            tet->GetPointIds()->SetId(i, global_to_vtk_id.at(g));
        }
        ugrid->InsertNextCell(tet->GetCellType(), tet->GetPointIds());
        elem_type->InsertNextValue(2);
    }

    ugrid->GetCellData()->AddArray(elem_type);
    ugrid->GetCellData()->SetActiveScalars("elemType");

    dumpUgridInfo(ugrid);
    return ugrid;
}
```