

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ  
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук  
Департамент программной инженерии

УТВЕРЖДАЮ  
Академический руководитель  
образовательной программы  
«Программная инженерия»,  
профессор департамента программной  
инженерии, канд. техн. наук

\_\_\_\_\_ В.В. Шилов  
« \_\_\_\_ » \_\_\_\_\_ 2018 г.

**Выпускная квалификационная работа**

на тему «Программа для дизайна поверхностей Кирхгофа-Плато»

по направлению подготовки 09.03.04 «Программная инженерия»

Научный руководитель  
Доцент департамента программной  
инженерии факультета компьютерных  
наук  
канд. техн. наук  
Римма Закиевна Ахметсафина

Выполнил  
студент группы БПИ141  
4 курса бакалавриата  
образовательной программы  
«Программная инженерия»  
Михаил Михайлович Ильченко

\_\_\_\_\_  
Подпись, Дата

\_\_\_\_\_  
Подпись, Дата

**Москва 2018**

## Реферат

Данная работа посвящена дизайну и визуализации минимальных поверхностей, ограниченных гибким каркасом, называемых поверхностями Кирхгофа-Плато.

В документе представлены теоретические основы и практическая реализация таких задач, как компьютерная визуализация трёхмерной графики, моделирование замкнутых эластичных кривых и расчёт поверхностей, имеющих минимальную площадь для заданного гибкого каркаса.

Результат работы – кроссплатформенная программа-визуализатор, созданная при помощи языка программирования Python 3 и графической библиотеки OpenGL, позволяющая создавать поверхности Кирхгофа-Плато. Для их создания пользователь рисует 2D каркасы с помощью кривых Катмулла-Рома. Затем программа изменяет координаты z контрольных точек нарисованных сплайнов и формирует поверхность минимальной площади в виде набора дополнительных вершин.

Работа содержит 42 страниц, 3 главы, 24 рисунка, 37 источников и 4 приложения.

**Ключевые слова:** задача Кирхгофа-Плато, задача Плато, компьютерная графика, минимальные поверхности, сплайны Катмулла-Рома, эластичные кривые.

### **Abstract**

This work is dedicated to design and visualization of minimal surfaces that span rigid boundary curve and named Kirchhoff-Plateau surfaces.

In this paper I present theoretical basics and implementation of problems such as computational visualization of three-dimensional objects, modelling closed elastic curves and finding surfaces of minimum area for given elastic boundary curve.

The result of this work is developed cross platform program-visualizer created with Python 3 programming language and OpenGL graphics library. This program allows to create Kirchhoff-Plateau surfaces. For creation user draws 2D boundaries using Catmull-Rom splines. Then program changes z coordinate of splines knots and forms minimal surface described with additional points.

This paper contains 42 pages, 3 chapters, 24 illustrations, 37 bibliography items, 4 appendices.

**Keywords:** *Kirchhoff-Plateau problem, Plateau problem, computer graphics, minimal surfaces, Catmull-Rom splines, elastic curves.*

## Содержание

Реферат .....	2
Abstract .....	3
Основные определения, обозначения и сокращения .....	5
Введение .....	6
Глава 1. Обзор методов построения поверхностей Кирхгофа-Плато .....	8
1.1. Задача Плато .....	8
1.2. Методы поиска минимальных поверхностей .....	8
1.3. Разбиение поверхности .....	10
1.4. Задача Кирхгофа-Плато .....	12
1.5. Компьютерный дизайн .....	13
Выводы по главе .....	13
Глава 2. Модели минимальных поверхностей и алгоритмы их построения .....	14
2.1. Базовое описание каркаса .....	14
2.2. Описание эластичного каркаса .....	17
2.3. Алгоритмы рисования .....	19
2.4. Алгоритм построения меша .....	23
2.5. Алгоритм расчёта площади поверхности .....	25
2.6. Алгоритм поиска KPS .....	26
2.7. Алгоритмы визуализации .....	26
Выводы по главе .....	28
Глава 3. Технологии и реализация программы .....	29
3.1. Функциональные требования .....	29
3.2. Выбранные инструменты .....	29
3.3. Пространство для визуализации .....	31
3.4. Классы программы и их назначение .....	32
3.5. Реализация алгоритмов .....	33
Выводы по главе .....	37
Заключение .....	38
Список использованных источников .....	40
Приложения .....	43

**Основные определения, обозначения и сокращения**

**Бишопа координаты (англ. Bishop frame)** – расслабленное состояние эластичной кривой

**Каркас** – замкнутая кривая в двухмерном или трёхмерном пространстве.

**Катмулла-Рома сплайн** – сглаженная кривая, проходящая через опорные точки ломаной кривой

**Кирхгофа-Плато поверхность (англ. Kirchhoff-Plateau surface, KPS)** - минимальная поверхность для данного эластичного каркаса.

**Контрольная точка** – опорная точка сплайнов Катмулла-Рома

**Материальные координаты (англ. material frame)** – текущее состояние эластичной кривой

**Меш** – сетка, накладываемая на поверхность

**Минимальная поверхность** – поверхность, имеющая наименьшую площадь для заданного каркаса.

**Пространство** – геометрическая модель материального мира. В данной работе может иметь 2 (2D) или 3 измерения (3D)

**Ребро** – отрезок или вектор, соединяющий две точки

**Стержень Кирхгофа** – модель эластичной кривой

**Сцена** – виртуальное пространство для моделирования

**Точка (вершина)** – точечный объект пространстве

**Эластичность (гибкость, англ. elasticity)** – характеристика модели твёрдого гибкого объекта

**ЯП** - язык программирования

## Введение

В данной выпускной квалификационной работе рассматривается визуализация минимальных поверхностей.

Задача о существовании минимальной поверхности, называемая задачей Плато, существует уже более 250 лет. При этом она до сих пор активно исследуется. Первоначально задача была сформулирована Лагранжем в 1760 году, а своё название получила в честь бельгийского физика Плато, проводившего эксперименты с натяжением мыльных плёнок. Мыльные плёнки представляют собой физическую визуализацию минимальных поверхностей, так как вследствие сил натяжения стремятся занять минимальную площадь при натяжении не некоторый каркас (Рисунок 1). Первые решения при различных ограничениях и вариациях задачи появились лишь в 1930 году [25], [9], а решения возможных модификаций задачи, таких как задача Кирхгофа-Плато, продолжают появляться и сейчас [15].



Рисунок 1. Натяжение мыльных плёнок на каркасы

Важно обратить внимание, что задача Плато и Кирхгофа-Плато заключаются в доказательстве существования минимальной поверхности в  $n$ -мерном пространстве, а для нас важна именно задача поиска, причём только в трёхмерном пространстве.

Математическое и физическое описание мыльных плёнок – не единственное, чем могут быть полезны данные поверхности. Уравнения, описывающие их, появляются и в других областях физики, а в биохимии они могут помочь понять, как форма белковой молекулы влияет на её связывание с различными поверхностями [34]. В производстве и строительстве поверхности с наименьшей площадью могут уменьшить количество использованных материалов. Кроме того, данные поверхности сами по себе обладают высокой внутренней материальной эффективностью и структурной стабильностью [23], что так же поможет при производстве различных сооружений.

В рамках данной работы была поставлена следующая **цель**: реализовать программу для персонального компьютера, которая визуализирует минимальные поверхности, существующие для заданных эластичных замкнутых кривых.

Для достижения поставленной цели необходимо решить следующие задачи:

- Изучить теоретические основы существования минимальных поверхностей в трёхмерном пространстве;
- Изучить подходы к созданию гибких каркасов и выбрать один из них для реализации;
- Изучить методы поиска минимальных поверхностей;
- Выбрать технологии для создания программы-визуализатора;
- Реализовать алгоритм создания эластичного каркаса;
- Реализовать алгоритм поиска KPS;
- Реализовать алгоритмы визуализации 3D изображений
- Спроектировать пользовательский интерфейс;
- Разработать архитектуру приложения;
- Протестировать программу;
- Написать техническую документацию.

Работа структурирована следующим образом: в первой главе описывается анализ изученных источников. Во второй главе описаны модели и алгоритмы, необходимые для реализации программной части. В третьей главе описаны средства и особенности самой реализации. В заключении работы подведены основные итоги решения поставленных задач, а также описаны возможные пути дальнейшего развития данной темы.

Также в Приложениях приведена техническая документация, составленная по ГОСТ-ЕСПД, содержащая техническое задание, программу и методику испытаний, руководство оператора и текст программы.

## Глава 1. Обзор методов построения поверхностей Кирхгофа-Плато

### 1.1. Задача Плато

Задачей Плато о существовании минимальных поверхностей занимались долгие годы, и сейчас существует множество различных источников, описывающих её решения. Одни из них используют различные топологические ограничения [25], [9], другие ограничивают размерность задачи [18], а третьи решают более общие случаи [26], [12], [37]. Так как в работе необходимо реализовать поиск минимальных поверхностей, наиболее интересны именно дискретные подходы [10], [6]. Многие авторы под дискретным описанием задачи понимают дискретизацию описания каркаса. Каркас описывается последовательностью точек в пространстве, а задача сводится к поиску точек, которые описывают минимальную поверхность, то есть принадлежат ей.

Как уже упоминалось ранее, поверхности минимальной площади имеют физический аналог – мыльные плёнки. Именно от этой аналогии и отталкиваются большинство авторов, описывающих проблему поиска. Для минимизации площади, необходимо уравновесить силы натяжения в каждой точке поверхности, как у мыльных плёнок. Исследователи сводят задачу к решению дифференциальных уравнений Лапласа или Эйлера-Лагранжа. Эти уравнения не имеют общего алгоритма решения. Поэтому для их решения будем использовать методы оптимизации.

### 1.2. Методы поиска минимальных поверхностей

Существует множество алгоритмов оптимизации, которые можно применить в нашем случае. Первым из них рассмотрим метод динамического программирования. Его использование применительно к нашей задаче хорошо и подробно описано в [19], [36]. Авторы ещё раз подчёркивают, что задачу Плато можно описать по-разному: вариационно, дифференциально-геометрически, комплексно, аналитически и другими способами. Авторы [19] описывают данную задачу с точки зрения теории графов. Различают 2 типа вершин графа: фиксированные для каркаса и свободные для поверхности, а топологию рёбер можно задать при помощи матрицы инцидентности. Далее весь граф проецируется на двумерную плоскость, как показано на Рисунок 2, после чего задача сводится к поиску функции, которая отображает высоту  $z$  точки исходного графа из точки её проекции  $A$ :

$$(A = (x, y) \in G), (z = f(A) = f(x, y): (x, y, z) \in \Omega),$$

где  $f$  – искомая функция, а остальные обозначения соответствуют Рисунок 2.

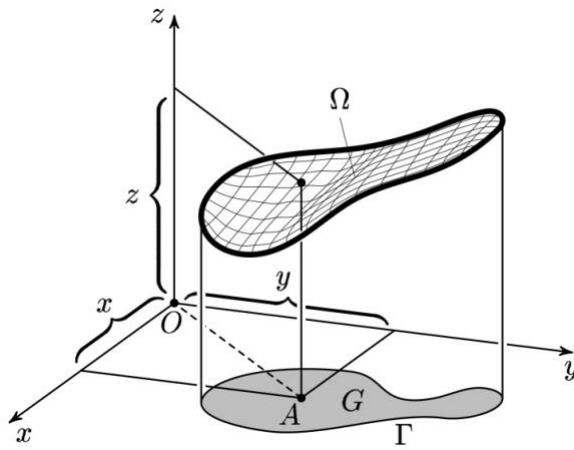


Рисунок 2. Проецирование трёхмерной плоскости на плоскость Oxy [36]

Далее необходимо понять, какие точки  $(x, y)$  будут использоваться для определения функции  $f$ . В [36] [19] авторы накладывают поверх проекции прямоугольный меш, координаты вершин которого легко рассчитываются при помощи известных координат фиксированных точек и параметров ширины и длины самого меша. Рассчитанные координаты вершин меша и будут точками, при помощи которых определяется искомая функция. Интересно, что, руководствуясь разными способами, оба источника приходят к одному и тому же заключению. Автор [36] утверждает, что физически из-за уравнивания сил натяжения, если какая-то точка находится ниже среднего уровня её соседей, то силы будут тянуть её вверх и наоборот. Авторы [19] получили аналогичный результат (1) при помощи центральных конечных разностей и уравнения Эйлера-Лагранжа.

$$f(x_i, y_j) = \frac{1}{4}(f(x_{i+1}, y_j) + f(x_i, y_{j+1}) + f(x_{i-1}, y_j) + f(x_i, y_{j-1})), \quad (1)$$

где  $i, j$  – порядковые номера вершины меша по горизонтали и вертикали соответственно. По формуле (1) итерационно вычисляется результат с использованием краевых условий – известных значений функции в точках каркаса. В начале задаётся исходное приближение, а после нескольких итераций вычисляются высоты всех вершин меша, которые и описывают минимальную поверхность.

Описанный выше метод прост и понятен для реализации и даёт результат вне зависимости от исходного приближения, поэтому его использования будет достаточно для наших целей. Однако обратим внимание, что значение высоты на каждой итерации зависит от значений соседних точек, из-за чего они должны рассчитываться одновременно.

В [36] также предложен интересный метод, позволяющий рассчитывать значения независимо от вершин-соседей. Данный метод, называемый «блуждания пьяницы», заключается в том, что в искомую вершину меша ставится объект, который равновероятно выбирает для перехода одну из соседних вершин. Переходя по вершинам, объект рано или поздно доберётся до края – точки каркаса, после чего он штрафует на значение функции в

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

точке, принадлежащей каркасу и отправляется «блуждать» снова из исходной точки.

Математическое ожидание штрафов и будет искомым значением. У данного метода тоже есть свои проблемы, ведь если фиксированные точки очень сильно отдалены друг от друга, блуждание может быть продолжительным.

При решении задачи описанными способами, минимальная поверхность получается как следствие при решении задачи поиска точек, описывающих её. Но можно и наоборот: минимизировать именно площадь, после чего получать для неё координаты описывающих её вершин. Для этого рассмотрим другой известный метод, при помощи которого можно решать данную задачу – метод градиентного спуска. В контексте данной задачи его применили в [33], [32] для того, чтобы визуализировать кусочно-линейную поверхность с заданным каркасом в среде Blender [3]. Другое незначительное отличие данных подходов заключается в том, что используется треугольный меш.

В результате сравнения различных алгоритмов, было принято решение использовать средние (1), так как их расчёт представляется самым простым и понятным для реализации. Данный алгоритм поиска в любом случае сходится к решению, а его точность зависит только от способа разбиения поверхности.

### **1.3. Разбиение поверхности**

Для того чтобы определить, какие точки использовать для описания поверхности, были использованы наложения простейших равномерно распределённых прямоугольных или треугольных мешей. Попытаемся понять, являются ли они лучшим решением в данном случае.

Рассмотрим первую возможную проблему - равномерность: так как разница высот соседних точек в разных местах поверхности не одинакова, многоугольники меша будут равны в проекции, но сильно отличаться в трёх измерениях. Следствием этого может быть недостаточная точность в измерении площади для больших многоугольников. Но в нашем случае размеры многоугольников регулируются, поэтому точность, даже при значительных различиях высот, компенсируется уменьшением размеров многоугольников. С другой стороны, если сделать меш неравномерным в проекции, то вместо обычного среднего (1) придётся использовать веса соседей, рассчитанные в зависимости от расстояний между вершинами, что будет лишним усложнением решения и, скорее всего, не добавит точности.

Вторая возможная проблема – визуализация: визуализировать нужно именно треугольники, потому что 3 точки точно лежат в одной плоскости, а 4 и более уже нет. Но разбить многоугольники меша на треугольники не такая большая проблема. Если используется прямоугольный меш, то достаточно для каждого прямоугольника провести одну диагональ, после чего получатся треугольники для визуализации. Если же используются другие

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

многоугольники, то их можно разбить при помощи алгоритмов триангуляции, например триангуляции Делоне [35].

Кроме того, стоит обратить внимание на различные техники разбиения поверхностей (англ. subdivision surfaces), благодаря которым поверхности сглаживаются. Единственное замечание, которое здесь можно сделать – такие техники используются тогда, когда описание поверхности уже известно. Поэтому рассмотрим их как дополнение для сглаживания уже найденной поверхности с уже существующим мешем.

Первый алгоритм – алгоритм Катмулла-Кларка, разработанный как обобщение бикубических однородных В-сплайновых поверхностей [4]. Поверхности Катмулла-Кларка определяются рекурсивно и состоят из четырёхугольников, не лежащих в одной плоскости (Рисунок 3).

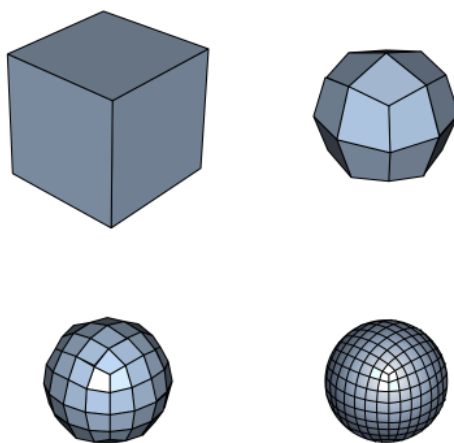


Рисунок 3. Первые 3 шага сглаживания куба методом Катмулла-Кларка

Как уже было замечено выше, визуализация требует от нас разбиения плоскости на треугольники. С этим прекрасно справляется алгоритм Чарльза Лупа [20] (Рисунок 4).

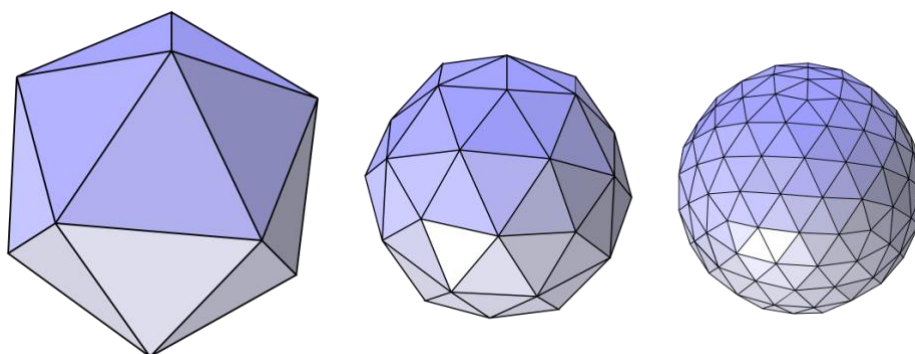


Рисунок 4. Первые 2 шага сглаживания икосаэдра методом Лупа

В данной работе будет использоваться наложение прямоугольного меша, описанные недостатки которого исправляются тонкостями реализации. Алгоритмы сглаживания для полученных минимальных поверхностей решено не использовать, так как сглаживания можно добиться и уменьшением размера четырёхугольников меша.

#### 1.4. Задача Кирхгофа-Плато

Когда задача Плато была решена, стало появляться множество модификаций задачи, в которых фигурировал специфический каркас [11], [31], [7], [14]. Конкретно задача Кирхгофа-Плато сформулирована [16] и решена [15] совсем недавно, что объясняет, почему по ней так мало информации на электронных ресурсах. Основана данная задача на теории стержней Кирхгофа и идее их использования для описания эластичных каркасов [8], [22], [28].

Поиск KPS в данном случае не отличается от поиска поверхности плато, а главное отличие заключается в более расширенной реализации каркаса. Как и с задачей Плато, для реализации нас интересуют дискретные подходы к теории стержней Кирхгофа. В данном случае были рассмотрены [29], [2]. Самая полная и простая реализация приведена в [1], благодаря расширенному описанию эластичного каркаса и достаточно компактной параметризации.

Говоря о текущей реализации и применении задачи Кирхгофа-Плато, необходимо упомянуть [23]. Авторы рассуждают о том, что KPS вследствие минимальных затрат на ресурсы и высокой структурной стабильности могут быть использованы для различных легковесных элементов, таких как акустические дефлекторы, декоративные элементы или даже небольшие крыши. Авторы на конференции SIGGRAPH в 2017 году представили разработанную программу, которая использует различные подходы для дизайна и последующей печати декоративных элементов (Рисунок 5 и Рисунок 6).

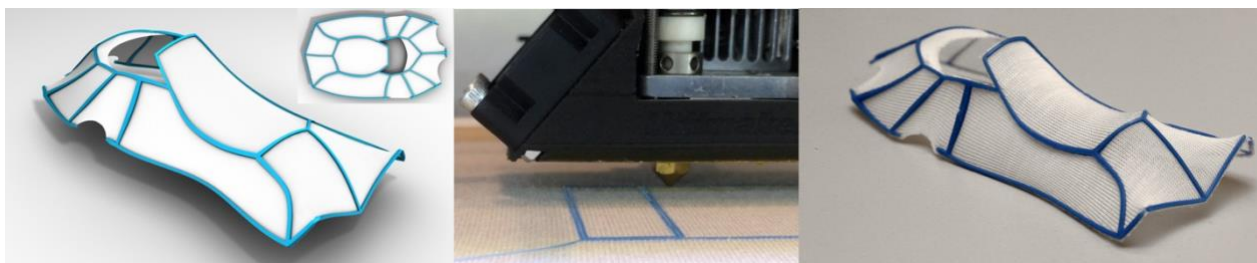


Рисунок 5. Визуализация KPS и печать с использованием ткани [23]

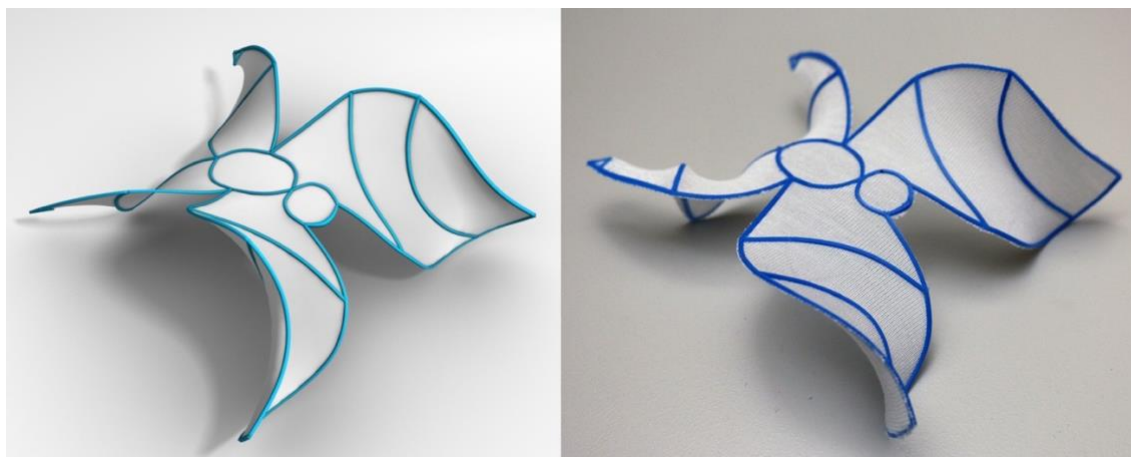


Рисунок 6. Комбинация KPS в виде бабочки [23]

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

К сожалению, опробовать данные методы дизайна и печати своими руками не удалось, так как работа защищена правами компании Disney.

Для реализации модели стержней Кирхгофа была выбрана модель [1].

### **1.5. Компьютерный дизайн**

Для того чтобы визуализировать трёхмерные поверхности, необходимо создать каркас. Есть 2 основных способа: загрузить каркас в программу или создать его собственноручно. Для нашей задачи дизайна подходит именно второй способ, который и нужно продумать.

Авторы [23] в своей работе для создания KPS осуществляют дизайн и преобразование обычных двухмерных объектов, что логично, учитывая, что дизайн трёхмерных каркасов не будет простым и понятным для пользователя программы. Таким образом, подзадача реализации создания каркаса сводится к обычному рисованию на полотне.

Каркас – это набор точек, которые соединены между собой. Поэтому при рисовании надо создавать и соединять контрольные точки, что будет подробнее описано во второй главе.

### **Выводы по главе**

В данной главе кратко описаны основные алгоритмы, используемые для построения минимальных поверхностей. Также выбраны алгоритмы и подходы для реализации.

## Глава 2. Модели минимальных поверхностей и алгоритмы их построения

### 2.1. Базовое описание каркаса

Каркас - это входные данные для программы и алгоритма поиска KPS. Последовательно опишем то, как он представляется в рамках реализации, добавляя всё больше и больше параметров.

Начать стоит с простейшего объекта – точки. Точка представляет собой кортеж трёх координат, определяющих её положение в пространстве:

$$p = (x, y, z)$$

Для описания каркаса используются  $N$  точек, которые называются контрольными и могут храниться в массиве (2) или матрице (3)

$$V = [pc_1, pc_2, \dots, pc_L], \quad (2)$$

$$V = \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_L & y_L & z_L \end{pmatrix} \quad (3)$$

где  $p_i$  –  $i$ -я точка в пространстве,  $i \in [1, L]$ ,  $L$  – общее количество точек

Точки могут быть соединены между собой рёбрами, причём каждое ребро соединяет только 2 точки. Хранить описание рёбер будем двумя способами. Во-первых, для каждого индекса из  $[1, L]$  создадим список, хранящий все индексы связанных точек. Во-вторых, добавим к описанию матрицу инцидентности. Для удобства будущего описания будем использовать ориентированные рёбра. Тогда матрица инцидентности описывается:

$$C = (c_{i,j})_{i=1, j=1}^{L,M}, (c_{i,j}) = \begin{cases} 1 & \text{если ребро } j \text{ направлено в точку } i \\ 0 & \text{если ребро } j \text{ не связывает точку } i, \\ -1 & \text{если ребро } j \text{ исходит из точки } i \end{cases} \quad (4)$$

где  $M$  – количество рёбер

Чтобы получить координаты векторов, описывающих рёбра, нужно выполнить операцию умножения матриц:

$$E = C^T V \quad (5)$$

Из полученной матрицы легко посчитать длины рёбер, используя Эвклидово расстояние:

$$E\_L_i = (E_i E_i^T)^{\frac{1}{2}},$$

Последовательно соединяя контрольные точки, получим линию:

$$line = \{control\_V\},$$

$$control\_V = [pc_1, pc_2, \dots, pc_N],$$

где  $N$  – количество контрольных точек, и  $\forall i \in [1, N - 1] \exists$  ребро  $pc_i, pc_{i+1}$

Ломаная, состоящая из контрольных точек, может быть незамкнутой и замкнутой (Рисунок 7).

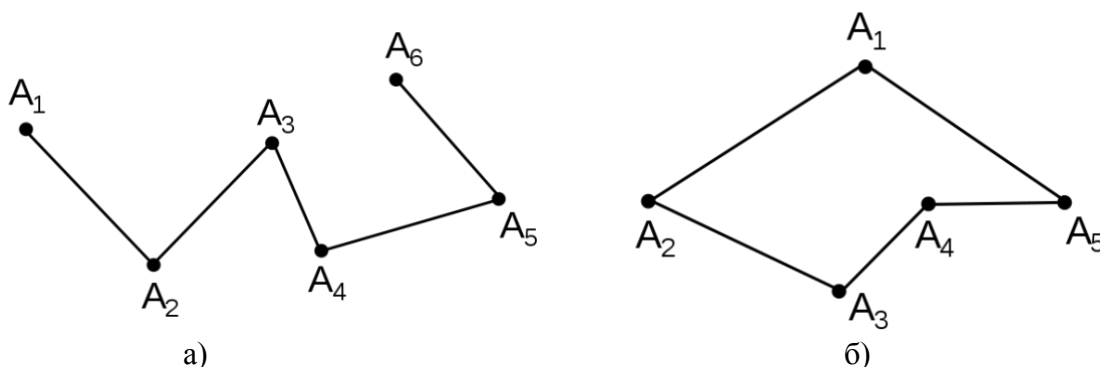


Рисунок 7. Ломаные линии с контрольными точками: а) незамкнутая; б) замкнутая

Каркас для минимальной поверхности должен быть замкнутым, поэтому линии необходимо добавить это свойство. После этого описание линии становится следующим:

$$line = \{control\_V, is\_closed\} \quad (6)$$

где  $is\_closed$  – признак замкнутости.

Такие линии нельзя использовать в качестве эластичных кривых, так как они имеют нулевую производную в большинстве точек. Будем использовать сплайны. Кривые Безье или В-сплайны широко используются для построения непрерывных, гладких кривых. При этом они ещё и достаточно простые для реализации, но возникнет одна проблема: эти сплайновые кривые не проходят через все контрольные заданные точки (Рисунок 8).

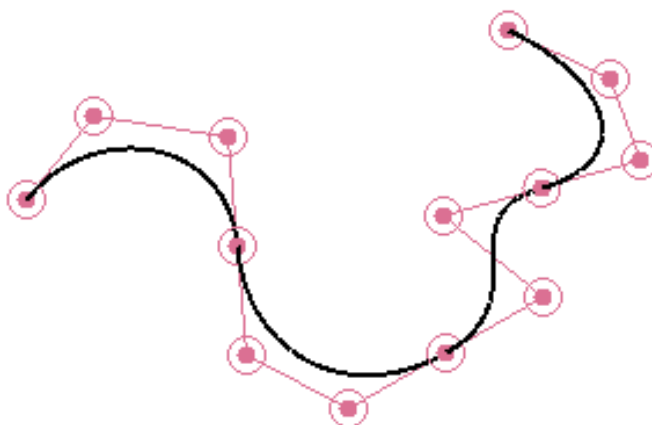


Рисунок 8. Составная кубическая кривая Безье не проходит через все контрольные точки

Сплайновые кривые, проходящие через все заданные точки, описаны Катмуллом и Ромом в 1974 году [5]. Сплайны Катмулла-Рома, как и другие из семейства кубических, описываются кубическим полиномом:

$$p(s) = c_0 + c_1s + c_2s^2 + c_3s^3, \quad s \in [0,1], \quad (7)$$

где  $c_i$  – коэффициенты в уравнении.

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

В общем случае они задаются с помощью четырёх точек, производная в одной из них зависит от предыдущей и следующей за ней вершин, поэтому кривую можно построить только между второй и третьей точками (Рисунок 9). Производная в точке  $p_i$  параллельна вектору  $(p_{i-1}, p_{i+1})$

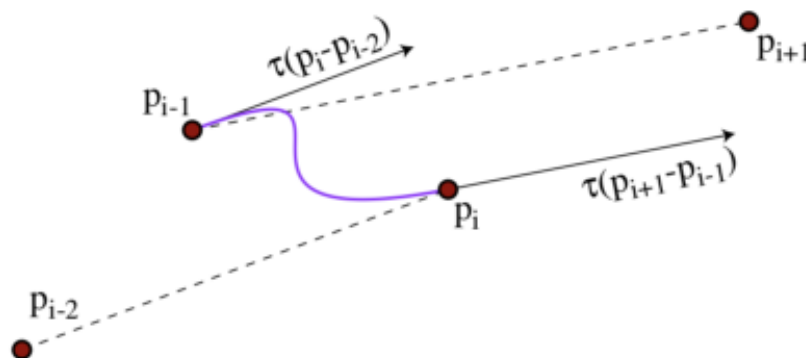


Рисунок 9. Кривая Катмулла-Рома (фиолетовым) для точек  $p_{i-2}, p_{i-1}, p_i, p_{i+1}$

Если для замкнутой линии можно рассчитать каждый фрагмент кривой, то для незамкнутой линии фрагменты, проходящие через первую и последнюю точки не определены. Бороться с этим можно задав крайним точкам направление производной, совпадающее с прилежащим ребром (Рисунок 10)

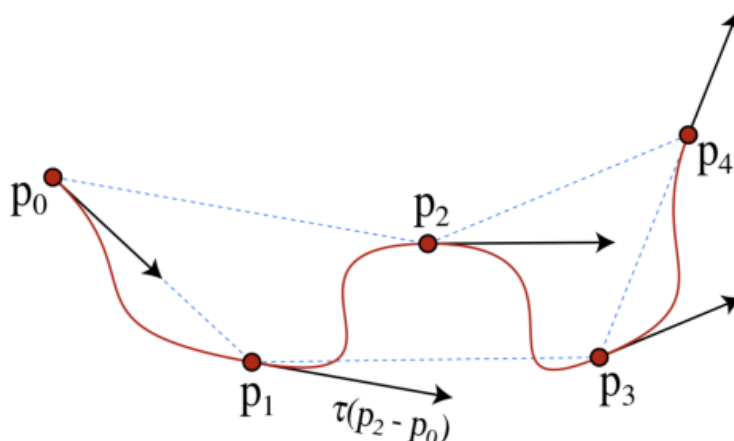


Рисунок 10. Кривая Катмулла-Рома (красным) для кривой из пяти точек

Параметр  $\tau$ , используемый при расчёте производной, авторы называют натянутостью и используют для того, чтобы определить, насколько «остро» кривая проходит через контрольную точку (Рисунок 11).

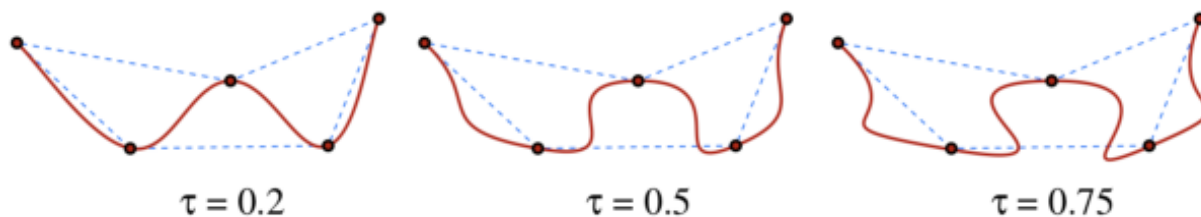


Рисунок 11. Кривая Катмулла-Рома в зависимости от  $\tau$

Перейдём к расчётам коэффициентов  $c_1, c_2, c_3, c_4$

Для четырёх точек  $p_{i-1}, p_i, p_{i+1}, p_{i+2}$  можно составить систему уравнений:

$$\begin{cases} p(0) = c_0 = p_i \\ p(1) = c_0 + c_1 + c_2 + c_3 = p_{i+1} \\ \dot{p}(0) = c_1 = \tau(p_{i+1} - p_{i-1}) \\ \dot{p}(1) = c_1 + 2c_2 + 3c_3 = \tau(p_{i+2} - p_i) \end{cases} \quad (8)$$

Решив систему (8) получим:

$$\begin{cases} c_0 = p_i \\ c_1 = -\tau p_{i-1} + \tau p_{i+1} \\ c_2 = 2\tau p_{i-1} + (\tau - 3)p_i + (3 - 2\tau)p_{i+1} - \tau p_{i+2} \\ c_3 = -\tau p_{i-1} + (2 - \tau)p_i + (\tau - 2)p_{i+1} + \tau p_{i+2} \end{cases}$$

В матричном виде решение (7) имеет вид:

$$p(s) = \begin{pmatrix} 1 & s & s^2 & s^3 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{pmatrix} \begin{pmatrix} p_{i-1} \\ p_i \\ p_{i+1} \\ p_{i+2} \end{pmatrix} \quad (9)$$

Для случая, когда необходимо построить фрагменты возле крайних точек, необходимо всего 3 точки для расчёта. Путём аналогичных вычислений получаем:

$$p(s) = \begin{pmatrix} s & s & s^2 & s^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ -\tau & \tau & 0 \\ 3\tau - 3 & 3 - 2\tau & -\tau \\ 2 - 2\tau & \tau - 2 & \tau \end{pmatrix} \begin{pmatrix} p_i \\ p_{i+1} \\ p_{i+2} \end{pmatrix} \quad (10)$$

Теперь из описания линии (6) получаем

$$\begin{aligned} line &= \{control\_V, all\_V, is\_closed\}, \\ all\_V &= [p_1, p_2, \dots, p_T], \end{aligned}$$

где  $T$  – количество вычисленных точек сплайна Катмулла-Рома, и  $\forall i \in [1, T -$

$1] \exists$  ребро  $p_i, p_{i+1}$ .

Несмотря на то, что  $all\_V$  содержит контрольные точки, они всё ещё нужны в нашем описании, так как они могут быть изменены, из-за чего потребуется пересчитать все остальные точки.

## 2.2. Описание эластичного каркаса

Переходим к дискретной модели эластичных стержней Кирхгофа, описанной в [1].

При помощи матрицы инцидентности (4) для вершин  $all\_V$  рассчитаем по формуле (5) значения координат векторов, описывающих рёбра линии.

Для каждого ребра вводим материальные координаты (от англ. material frame)  $M = \{t, m_1, m_2\}$ , где  $t$  – единичные векторы касательных, а  $m_1, m_2$  – ортогональные им векторы (Рисунок 12).

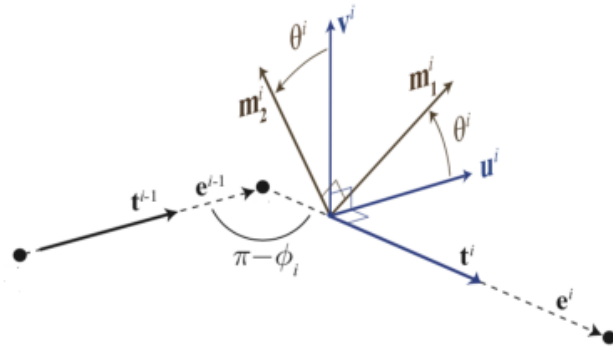


Рисунок 12. Основные параметры дискретной модели [1]

Далее введём понятие кривизны  $k$  и бинормальной кривизны  $kb$ . Фактически кривизна – это вторая производная, а бинормаль является вектором, ортогональным касательной и кривизне. В дискретном виде:

$$k = 2tg\left(\frac{\phi}{2}\right),$$

$$kb_i = \frac{2 * e^{i-1} \times e^i}{|e^{i-1}| * |e^i| + e^{i-1} e^i}, \quad (11)$$

где  $\phi$  - угол при вершине, изображённый на Рисунок 12, который можно рассчитать при помощи теоремы косинусов,  $i \in [2, T - 1]$  – индекс вершины, имеющей рёбра  $e^{i-1}, e^i$ .

Обратим внимание, что в отличие от материальных координат, кривизна и бинормальная кривизна связаны с вершиной, а не ребром.

Далее рассмотрим, как определяется потенциальная энергия эластичной кривой. Её можно подразделить на энергию сгиба и энергию вращения:

$$E = E_{bend} + E_{twist} \quad (12)$$

Энергия сгиба зависит от бинормальной кривизны (11) и рассчитывается:

$$E_{bend} = \frac{1}{2} \alpha \sum_{i=2}^{T-1} \frac{|kb_i|^2}{|e^{i-1}| + |e^i|}, \quad (13)$$

где  $\alpha$  – гиперпараметр.

Теперь необходимо разобраться со второй частью – энергией вращения. Взглянув на Рисунок 12, можно увидеть, что векторы  $m_1, m_2$  материальных координат отклонены на угол  $\theta$  от векторов  $u, v$  соответственно. Чтобы найти энергию вращения, надо понять, при каких векторах  $u, v$  кривая находилась в покое.

Введём понятие координат Бишоп (англ. Bishop frame)  $B = \{t, u, v\}$ , которая и определяет самое расслабленное состояние эластичной кривой. Точного определения состояния расслабленности нет, поэтому его объясняют при помощи примера. Если взять вытянутый в длину шнурок и провернуть один его край, остальная часть шнурка будет стремиться «расслабиться» и тоже будет поворачиваться. Однако важно обратить внимание, что остальная часть шнурка будет вращаться не до исходного вытянутого состояния и остановится раньше.

Исходя из связи материальных координат и координат Бишопа, можно сказать, как определяются  $m_1, m_2$  в зависимости от угла поворота:

$$m_1^i = \cos \theta^i u^i + \sin \theta^i v^i,$$

$$m_2^i = -\sin \theta^i u^i + \cos \theta^i v^i,$$

где  $i \in [1, T - 1]$  – индекс соответствующего ребра.

Энергия вращения зависит от того, на какой угол повернуто каждое ребро:

$$E_{twist} = \beta \sum_{i=2}^{T-1} \frac{(\theta^i - \theta^{i-1})^2}{|e^{i-1}| + |e^i|}, \quad (14)$$

где  $\beta$  – гиперпараметр.

Теперь, когда известно, как рассчитывается энергия (12), (13), (14) и материальные координаты, необходимо понять, каким образом искать координаты Бишопа. Как и в примере со шнурком, нужно задать исходное положение, инициализировав  $u^0$ , что достаточно просто при известном значении  $t^0$  и ортогональности  $u^0 \perp t^0$ . Оставшийся вектор, как бинормаль, рассчитывается:

$$v^i = u^i \times t^i$$

Для того чтобы найти остальные векторы координат Бишопа, необходимо сформировать матрицу переноса  $P$ . Если осуществлять простые переносы без дополнительных вращений, то векторы будут меняться наиболее расслабленным способом. Сформируем правила:

$$P_i(t^{i-1}) = t^i \quad (15)$$

$$P_i(t^{i-1} \times t^i) = t^{i-1} \times t^i \quad (16)$$

$$P_i(u^{i-1}) = u^i \quad (17)$$

$$u^i t^i = 0 \quad (18)$$

Уравнения (15) и (17) определяют матрицу как матрицу переноса, (16) показывает, что бинормаль меняться не должна, а (18) равнозначно  $u^i \perp t^i$ . Решив систему из уравнений (15) - (18), получим  $P$ .

### 2.3. Алгоритмы рисования

В данном разделе рассмотрим те методы, при помощи которых осуществляется рисование каркаса.

Оговорим некоторые правила, которые существуют при рисовании:

- можно поставить новую точку или передвинуть существующую;
- после создания или передвижения точка считается активной, то есть от неё может быть проведено ребро;
- при соединении активной точки с другой образуется ребро;

- в любой момент времени либо активных точек нет, либо она ровно одна;
- при нажатии на активную точку все точки становятся неактивными;
- при отсутствии активной точки и нажатии на любую точку, она становится активной;
- при наличии активной точки и создании новой, они соединяются ребром.

Казалось бы, задача не такая сложная, ведь если её сильно упростить, то для неё просто нужно соединять точки. Однако при построении нескольких каркасов возникают следующие вопросы:

- какие рёбра становятся частью линий;
- если ребро принадлежит разным линиям, то по точкам какой из них его необходимо сглаживать;
- как понять, что линию нужно замкнуть;
- все ли точки можно соединять;
- ограничено ли пространство, куда можно перетащить точку;
- что делать если пересекаются множества точек, ограниченные двумя замкнутыми линиями;

и другие.

Договоримся, что ребро двух контрольных точек может принадлежать только одной линии. В таком случае сглаживать его будем только один раз. Рёбра, которые получаются при сглаживании, могут принадлежать уже нескольким линиям. Исходя из этой договорённости, становится понятно, что для незамкнутых линий не должно быть ветвления, так как невозможно определить, к какой именно линии отнести ребро. Ветвление же от замкнутой линии быть может, так как ребро будет принадлежать именно замкнутой линии (Рисунок 13).

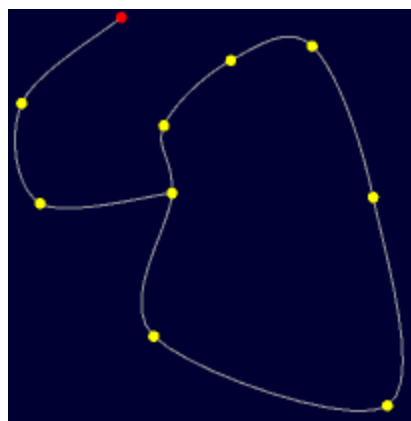


Рисунок 13. Замкнутая линия и отходящая от неё незамкнутая

Скажем, что каждое ребро принадлежит какой-то линии. Тогда есть четыре возможных случая задания ребра линии.

Первый из них, самый простой, когда ребро создаёт новую линию. Новая линия создаётся, если соединяемые точки не принадлежат ни одной линии, то есть не имеют рёбер.

Второй случай, когда ребро продолжает линию. Во-первых, такое происходит, если одна точка является концом незамкнутой линии, а вторая не имеет рёбер. Во-вторых, такое происходит, если одна точка является концом незамкнутой линии, а вторая лежит на замкнутой. Во втором случае как раз и получается ветвление, показанное на Рисунок 13.

Третий случай - ребро соединяет две линии и становится частью одной большой линии. Такое происходит только в том случае, если обе точки являются концами двух разных линий (Рисунок 14).

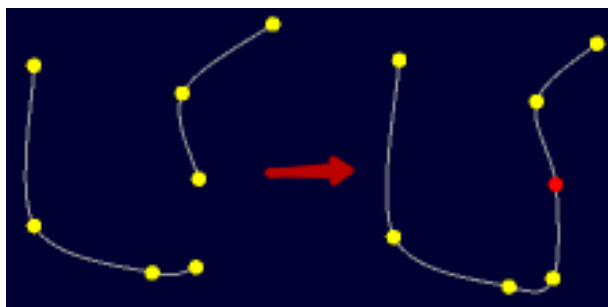


Рисунок 14. Соединение двух линий в одну

Четвёртый случай - новое ребро замыкает линию. Такое может произойти только если обе точки являются разными концами одной и той же незамкнутой линии (Рисунок 15).

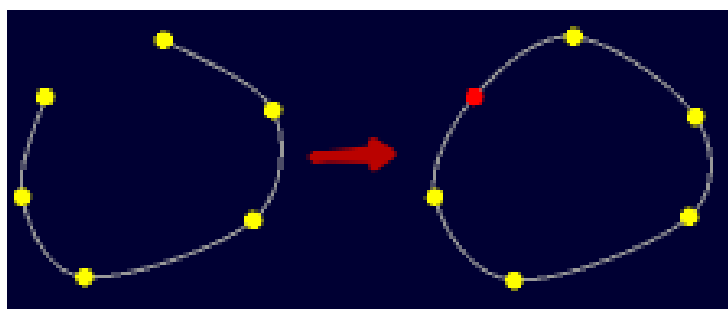


Рисунок 15. Замыкание линии

Подумаем, что должно произойти, если красную точку на Рисунок 13 соединить с точкой замкнутой линии. По логике у нас получается новая замкнутая линия, так как она ограничивает некое множество точек, но часть этой замкнутой линии на самом деле принадлежит другой замкнутой линии. Если замыкать так, как это делается обычно, получится следующее (Рисунок 16), что конечно же неверно.

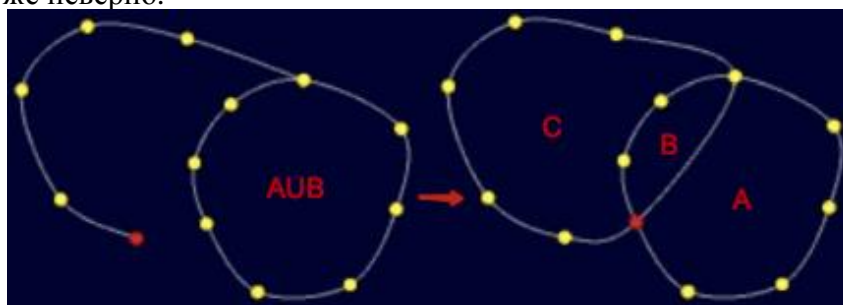


Рисунок 16. Неправильное замыкание второй линии на первой

Получается, что объединение множеств  $A$  и  $B$  – множество, ограниченное первой замкнутой линией, а объединение множеств  $B$  и  $C$  – множество, ограниченное второй замкнутой линией.

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

При этом для второй линии необходимо получить только множество  $C$ . Объединения  $A$  и  $B$ ,  $B$  и  $C$  известны, тогда  $C$  можно получить как разницу множеств:

$$C = (B \cup C) - (A \cup B),$$

Для того чтобы определить, какая линия замыкается при помощи одной или двух других замкнутых, необходимо хранить ссылки на них.

Остался не пояснённым вопрос передвижения точек. Единственное ограничение, которое здесь необходимо поставить: рёбра могут пересекаться только своими точками. Иными словами, пересечений рёбер не должно быть, могут быть только соединения. Данное ограничение делает недопустимой ситуацию на Рисунок 17.

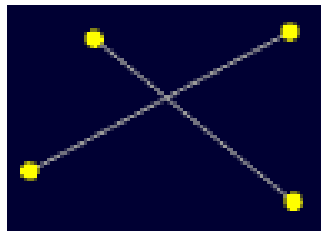


Рисунок 17. Недопустимое пересечение рёбер

Теперь рассмотрим удаление точек. Нередко происходит так, что случайно была поставлена лишняя точка, поэтому необходимо предусмотреть возможность её удаления. Как и соединение точек, процесс удаления влияет на рёбра и линии, которым принадлежала точка. Разберём различные случаи, которые могут возникнуть.

1. Разделение незамкнутой линии на две новые, когда удаляется одна из не концевых точек (Рисунок 18).

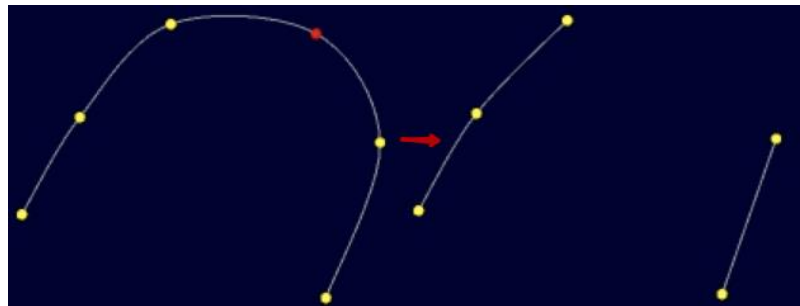


Рисунок 18. Удаление точки в середине линии

2. Удаление линии. Это происходит, когда после удаления точки у линии остаётся только одна точка. Иногда в эту ситуацию могут попадать и разделяемые линии (Рисунок 19).

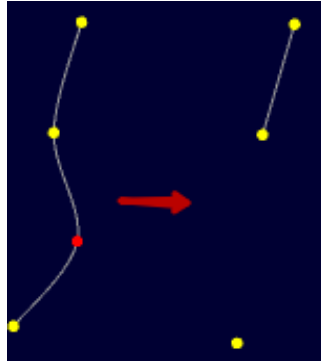


Рисунок 19. Разделение линии на 2 части, при котором одна удаляется

3. Удаление точки замкнутой линии. Во-первых, сама замкнутая линия становится незамкнутой. Во-вторых, другие линии, примыкающие к ней, должны быть отделены, чтобы не было ветвления (Рисунок 20).

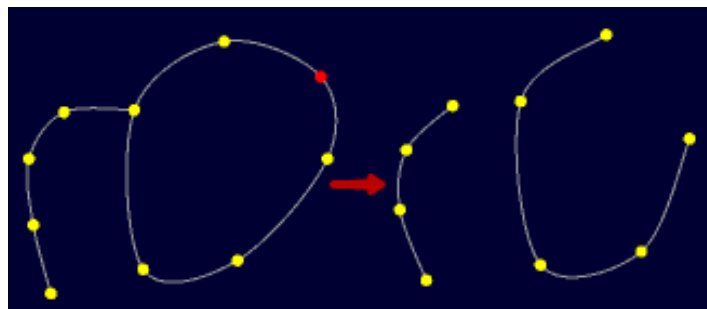


Рисунок 20. Результат удаления красной точки

## 2.4. Алгоритм построения меша

Перед этапом построения меша у нас есть трёхмерный каркас и его проекция  $\Gamma$  (Рисунок 2):

$$\Gamma = \{pr, fr(pr)\}, \quad (19)$$

где  $pr$  – массив точек с координатами  $(x, y)$ ,  $fr(pr)$  – координаты  $z$  точек  $pr$ .

Будем использовать квадратный меш, который для визуализации можно разбить на треугольники диагоналями.

Если не обращать особого внимания на то, как накладывается меш, точки  $pr$  могут не попасть в его вершины. Данные несовпадения вызовут проблемы при поиске KPS, так как если точка лежит не в вершине, вместо обычного среднего (1) придётся использовать взвешенное. Чтобы избежать данной проблемы, необходимо распределить меш таким образом, чтобы все точки попали в его вершины.

Первое, что необходимо понять, как располагается меш. Очевидно, что он должен покрывать все существующие точки. Для этого опишем ограничения на координаты вершин меша:

$$\begin{aligned} \min_{1 \leq i \leq N} x_i &\leq x \leq \max_{1 \leq i \leq N} x_i, \\ \min_{1 \leq i \leq N} y_i &\leq y \leq \max_{1 \leq i \leq N} y_i, \end{aligned}$$

где  $N$  – количество точек  $pr$ .

Далее, какой именно должен быть шаг. Для использования обычного среднего необходимо, чтобы шаг был одинаковым как для  $x$ , так и для  $y$ . Для того чтобы найти шаг, при помощи которого можно попасть вершинами меша во все точки, необходимо выполнить следующий алгоритм:

1. Отсортировать точки  $pr$  по координате  $x$  по возрастанию
2. Посчитать разницы в координатах для соседних точек:

$$dx_j = x_{j+1} - x_j,$$

где  $j \in [1, N - 1]$  – индекс координаты в отсортированном массиве

3. Далее нужно будет использовать наибольший общий делитель, но он считается для целых чисел, а  $dx_j$  может быть и дробным. Тогда вычисляем степень десятки, на которую умножим все числа  $dx_j$ , чтобы они стали целыми:

$$pow = \max_{1 \leq j \leq N-1} fl_j,$$

где  $fl_j$  – количество знаков после запятой для числа  $dx_j$

4. Найти разницу  $dx$ , которая покрывает все точки по  $x$ :

$$dx = \frac{\text{НОД}(dx_1 * 10^{pow}, dx_2 * 10^{pow}, \dots, dx_{N-1} * 10^{pow})}{10^{pow}},$$

где  $\text{НОД}$  – наибольший общий делитель.

5. Выполнить пункты 1-4 для  $y$
6. Найти разницу  $d$ , как наибольший общий делитель для  $dx$ ,  $dy$  с учётом того, что они могут быть дробными

Таким образом получаем дискретное описание для меша:

$$mesh = \{pm, fm(pm)\}, \quad (20)$$

$$M = \left( \frac{\max_{1 \leq i \leq N} x_i - \min_{1 \leq i \leq N} x_i}{d} + 1 \right) * \left( \frac{\max_{1 \leq i \leq N} y_i - \min_{1 \leq i \leq N} y_i}{d} + 1 \right),$$

где  $M$  – количество вершин  $pm$  меша, а  $fm(pm)$  – высота точки, которую и необходимо найти.

Взглянув на результат (Рисунок 21), становится понятно, что какое-то количество вершин меша не находится в пределах, ограниченных каркасом.

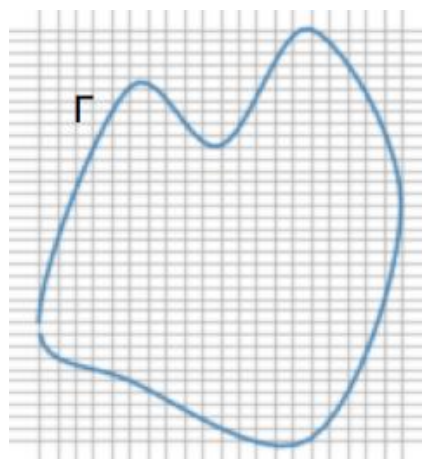


Рисунок 21. Меш, полученный для заданной линии Г

Если вершина  $i$  не принадлежит множеству точек, ограниченному каркасом  $\Gamma$ , необходимо убрать её из *mesh*. После данной фильтрации лишних точек не останется.

## 2.5. Алгоритм расчёта площади поверхности

Из-за того, что Рисунок 21 двухмерный, кажется, что меш образует своими вершинами много маленьких квадратов. Так и есть для проекции, но на самом деле высоты вершин меша могут сильно отличаться, из-за чего 4 точки какого-нибудь «квадрата» могут не лежать в одной плоскости.

Именно поэтому каждый квадрат необходимо разбить на 2 треугольника для более точной визуализации. Площадь вычисляется как сумма площадей треугольников:

$$S = \sum_{i=1}^K S_{\Delta_i},$$

где  $S_{\Delta_i}$  – площадь  $i$ -го треугольника,  $K$  – количество треугольников в меше

Проведение диагонали в четырёхугольном фрагменте поверхности чаще всего не вызывает проблем, но могут быть квадраты, находящиеся рядом с границей каркаса, у которых отсутствует одна вершина, если она отфильтрована из меша. (Рисунок 22).

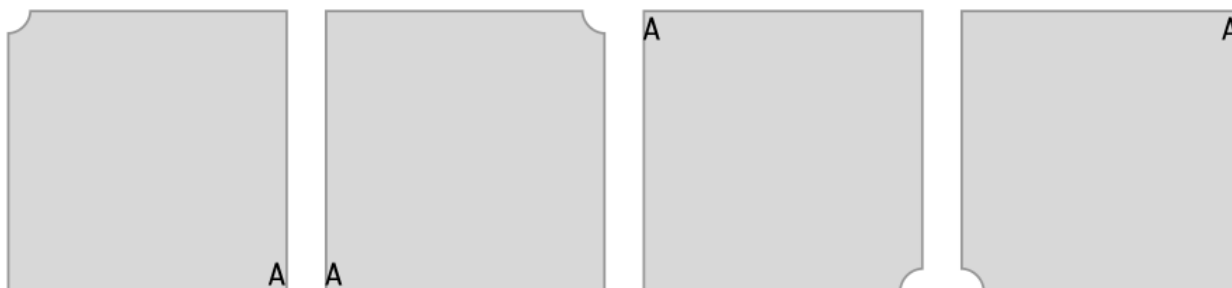


Рисунок 22. Фрагменты меша с отфильтрованной вершиной

В таких случаях необходимо посчитать площадь только одного треугольника. Например, рассмотрим квадрат, у которого отсутствует верхний левый угол. Тогда для вычисления площади треугольника будут использованы вершины  $A(x, y, fm(x, y))$ ,  $B(x, y + d, fm(x, y + d))$  и  $C(x - d, y, fm(x - d, y))$ . Площадь треугольника:

$$S_{\Delta ABC} = \frac{1}{2} * \rho(A, B) * \rho(A, C) * \sin A,$$

где  $\rho$  – Эвклидово расстояние между точками.

## 2.6. Алгоритм поиска KPS

Теперь, когда все входные данные готовы (каркас (19) и меш (20)), можно приступить к поиску высот вершин меша для расчёта KPS. Для этого будем использовать следующий алгоритм:

1. Инициализируем  $fm(pm)$  для каждой точки  $mp$  меша средним значением известных точек каркаса;
2. Инициализируем булевый параметр  $optimizing=True$ . От данного параметра зависит выполнение цикла;
3. Инициализируем параметр точности  $\varepsilon$  для определения, что цикл пора остановить;
4. Пока  $optimizing==True$ 
  - а. Для каждой точки  $(x,y)$  меша
    - i. Инициализируем  $z=0$
    - ii. Для каждой соседней точки  $(x-d,y)$ ,  $(x,y-d)$ ,  $(x+d,y)$ ,  $(x,y+d)$ 
      1. Если точка принадлежит каркасу,  $z=z+fr(pm)$
      2. Иначе  $z=z+fm(pm)$
    - iii. Обновляем значение  $fm(x,y) = \frac{z}{4}$
  - б. Если модуль разности площадей нового и старого мешей меньше  $\varepsilon$ , обновить значение  $optimizing=False$

## 2.7. Алгоритмы визуализации

Визуализация и её алгоритмы больше привязаны к инструментам реализации, которые будут описаны в третьей главе. В данном пункте опишем некоторые из них.

Начнём с двумерной визуализации. Она нужна для рисования каркаса, является статичной (то есть её нельзя передвигать или вращать) и использует самую базовую ортогональную проекцию. Ортогональная проекция задаётся при помощи матрицы:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{b-t} \\ 0 & 0 & \frac{2}{n-f} & \frac{f+n}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где  $l, r$  определяют левую и правую границу для координаты  $y$ ;  $b, t$  – левая и правая граница для  $x$ ;  $n, f$  – соответствующие границы для  $z$

Точка в модели нашего мира является точечным объектом, но в программе точку необходимо видеть, выделять для передвижения или удаления. Для того чтобы такое можно было осуществить, точка визуализируется как круг с заданным радиусом, центр которого находится в координатах точки. Тогда необходимо оговорить, что в рамках визуализации точки считаются одинаковыми, если расстояние между ними меньше суммы радиусов.

Теперь перейдём к трёхмерной визуализации. В ней, в отличие от двухмерной, наоборот, изменения каркаса не происходит, а все манипуляции с изменением положения и направления камеры. Камера всегда определяется тремя векторами:

- *eye* – положение камеры в пространстве
- *center* – положение, куда камера должна быть направлена
- *up* – вектор определяющий куда направлен верх камеры.

В рамках нашей модели вектор *up* всегда направлен вверх (21), положение *center* всегда находится в середине сцены (22), а положение *eye* всегда на сфере с центром в *center* и радиусом *R* (23)

$$up = (0,1,0), \quad (21)$$

$$center = (center_x, center_y, center_z), \quad (22)$$

$$eye = center + (\sin \alpha \cos \beta * R, \sin \alpha \sin \beta * R, \cos \alpha * R), \quad (23)$$

где  $\alpha, \beta$  – углы поворота камеры

Теперь соберём эти три вектора в так называемую матрицу *lookAt*. Для этого необходимо рассчитать 3 вспомогательных вектора:

$$f = \frac{center-eye}{|center-eye|}, \quad (24)$$

$$s = f \times \frac{up}{|up|}, \quad (25)$$

$$u = \frac{s}{|s|} \times f, \quad (26)$$

LookAt преобразование выполняет матрица, представляющая собой произведение матрицы, содержащей вспомогательные векторы (24), (25), (26), на матрицу переноса:

$$\begin{pmatrix} s_x & s_y & s_z & 0 \\ u_x & u_y & u_z & 0 \\ -f_x & -f_y & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Важно, что в трёхмерной визуализации незамкнутые линии, которые есть в двухмерной, не нужны, потому что их можно считать недорисованными каркасами.

Говоря про замкнутые линии, в двухмерном пространстве их можно визуализировать как многоугольники, которые заданы каркасом. В трёхмерном же пространстве необходимо использовать разделённый на треугольники меш.

### **Выводы по главе**

В данной главе описано, каким именно образом дискретно определяются каркас, линия и меш, которые будут использоваться. Также описаны необходимые для задачи алгоритмы и формулы, при помощи которых могут быть рассчитаны площадь поверхности, энергия эластичного стержня или, точки, описывающие искомую KPS. Теперь, когда все алгоритмы рассмотрены, можно перейти к реализации, которая тоже имеет свои тонкости, связанные с выбранными инструментами разработки.

### **Глава 3. Технологии и реализация программы**

В данной главе описаны функциональные требования, сформированные в техническом задании (Приложение А), а также самые важные детали реализации программы.

#### **3.1. Функциональные требования**

В рамках технического задания, был поставлен ряд требования к функциональным характеристикам.

Программа-визуализатор должна обеспечить возможность выполнения следующих функций:

- Создание каркаса;
- Деформация каркаса путём добавления, передвижения или удаления контрольных точек;
- Преобразование двухмерного каркаса в трёхмерный;
- Визуализация двухмерных и трёхмерных объектов на сцене;
- Движение камеры и объекта по сцене;
- Расчёт точек, принадлежащих минимальной поверхности;

#### **3.2. Выбранные инструменты**

Одним из главных компонентов работы данной программы является компьютерная визуализация KPS, поэтому необходимо использовать библиотеки для создания графики. В качестве такой библиотеки можно использовать OpenGL, Vulkan или DirectX, но была выбрана именно первая – Open Graphics Library, которая признаётся стандартом в области компьютерной графики на протяжении многих лет.

OpenGL представляет независимую от платформы и языка спецификацию, что позволяет выбрать для реализации любой ЯП, реализующий привязку функций графической библиотеки. Однако стоит обратить внимание, что сам OpenGL не предоставляет инструментов создания графического интерфейса, а содержит только функции для работы с графикой. Именно поэтому разработка интерфейса и различных функций программы выполняется средствами выбранного языка.

Несмотря на то, что C++ наиболее популярен у разработчиков для работы с OpenGL, в рамках данной реализации был выбран ЯП Python 3.

Теперь, когда основное средство выбрано, рассмотрим библиотеки, позволяющие создавать простейшую программу с оконным интерфейсом. В результате поиска были выделены такие библиотеки, как Tkinter, PyQt, kivy, pygame, pygamelet, GLFW. Проведём небольшое сравнение:

Сравнение библиотек для python

	Tkinter	PyQT	kivy	pygame	pyglet	GLFW
Встроенные виджеты для интерфейса	+	+	+	-	-	-
Обработка действий клавиатуры и мыши	+	+	+	+	+	+
Собственная обёртка для функций OpenGL	-	-	+	-	+	-
Кроссплатформенность	+	+	+	+	+	+

С библиотеками, у которых нет собственной gl обёртки, придётся использовать стороннюю библиотеку PyOpenGL. Несмотря на то, что она реализует весь функционал OpenGL, с ней всё-таки может быть проблема при совместном использовании с другими библиотеками.

Библиотека kivy очень большая, и для создания интерфейса требует написания кода на собственном Kv языке, что необоснованно усложнит реализацию. Совместного использования сравниваемых библиотек быть не может, так как каждая из них реализует собственный цикл для выполнения. Таким образом, несложная библиотека pyglet с хорошей, понятной документацией [24]. Как уже видно из Таблица 1, pyglet не имеет собственных виджетов, и, по сути, просто реализует обработку входных сигналов мыши, клавиатуры или джойстика и обёртку для OpenGL. Отсутствие инструментов для создания интерфейса может быть восполнено обычным рисованием при помощи графической библиотеки, с чем прекрасно справляется написанная специально для pyglet библиотека glooeu [17].

Для базовых математических функций, использовался модуль math, встроенный в Python.

Так как во многих описанных во второй главе формулах фигурировали матричные вычисления, было принято решение использовать вычислительную библиотеку NumPy, реализующую многомерные массивы, линейную алгебру и другие полезные математические функции [21].

Для решения уравнений была выбрана библиотека символьных вычислений SymPy [30].

Для топологических вычислений была выбрана библиотека shapely [27], реализующая для python популярный проект GEOS [13].

### 3.3. Пространство для визуализации

Необходимо отдельно визуализировать двухмерное пространство, в котором происходит дизайн каркаса, и трёхмерное, в котором происходит взаимодействие с самими KPS. Так как двухмерный случай отличается от трёхмерного только отсутствием координаты  $z$ , принято решение визуализировать их одновременно в одном окне, разделённом примерно пополам.

В используемой библиотеке `pyglet` контекст OpenGL целиком связывается с окном программы. Для того чтобы выбрать конкретную часть окна, нужно использовать функцию `glViewport(x,y,w,h)`, которая устанавливает левый нижний угол поля для рисования в позиции  $(x, y)$  окна, а само поле делает шириной  $w$  и высотой  $h$ . Единственное, что стоит оговорить, Retina дисплеи имеют в 2 раза больше пикселей на дюйм, поэтому для них в функцию необходимо передавать в 2 раза большие ширину и высоту.

Теперь рассчитаем позицию контейнера и его вьюпорта так, чтобы они расположились как показано на Рисунок 23.



Рисунок 23. Разделение окна приложения на контейнеры

На Рисунок 23 красными линиями показаны отступы.

Пусть позиция окна имеет координаты  $(pos_x, pos_y)$ , ширину и высоту  $w, h$  соответственно. Отступ обозначим  $margin$ . Тогда контейнер для 2D имеем описание:

$$left_w = pos_x + margin, \quad (27)$$

$$bottom_w = pos_y + margin,$$

$$width_w = \frac{w - 3 * margin}{2},$$

$$height_w = h - 2 * margin,$$

$$left_v = pos_x + margin, \quad (28)$$

$$bottom_v = pos_x + margin,$$

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато

$$width_v = \frac{w * coef - 3 * margin}{2},$$

$$height_v = h * coef - 2 * margin,$$

где  $(left\_*, bottom\_*)$  – позиция контейнера,  $width\_*$  и  $height\_*$  – его ширина и высота соответственно,  $coef$  – коэффициент, равный двум для Retina дисплея, причём  $*_v$  для вьюпорта, а  $*_w$  для самого окна

Для описания контейнера для 3D вместо (27) и (28) используются (29) и (30) соответственно, которые определяют позицию  $x$  контейнера

$$left_w = pos_x + \frac{w + margin}{2}, \quad (29)$$

$$left_v = pos_x + \frac{w * coef + margin}{2} \quad (30)$$

Параметры вьюпорта используются только для контекста OpenGL, а параметры самого контейнера нужны для того, чтобы определять, к чему именно относятся действия мыши. Pyglet всегда даёт координаты мыши относительно всего окна, поэтому понять, что курсор мыши находится в пределах контейнера можно проверкой условий:

$$left_w \leq x \leq left_w + width_w,$$

$$bottom_w \leq y \leq bottom_w + height_w,$$

где  $(x, y)$  – координаты курсора.

### 3.4. Классы программы и их назначение

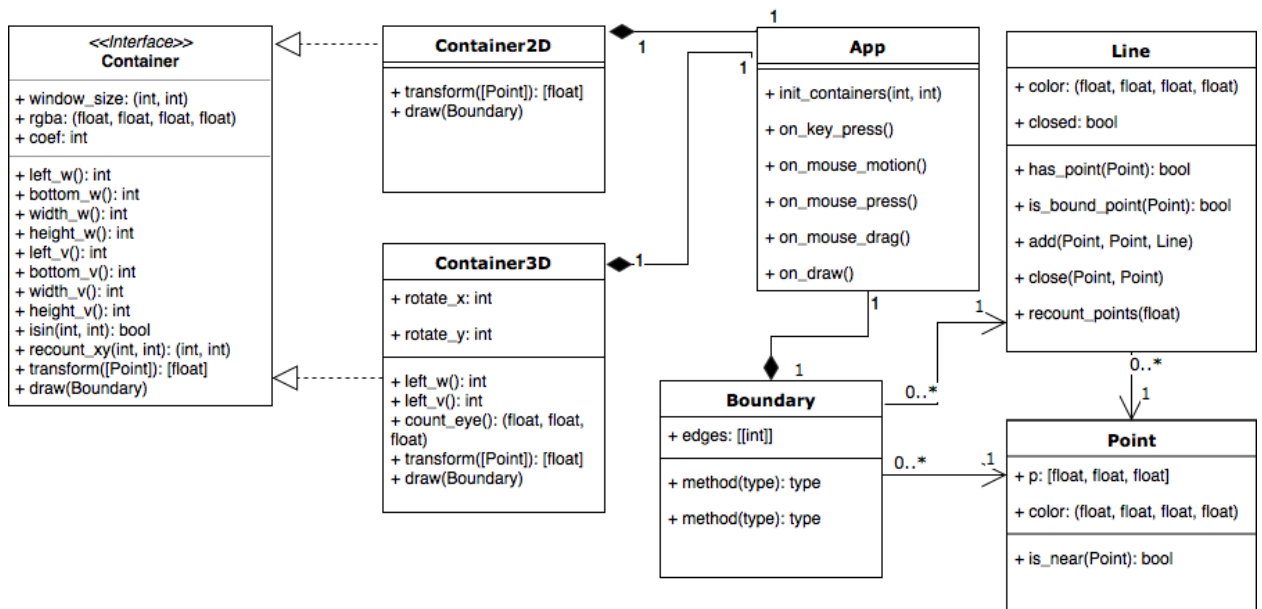


Рисунок 24. Диаграмма классов

В программе есть основной класс App, который реализует pyglet окно. Окно делится при помощи Container2D и Container3D на две части, как описано в предыдущем разделе. App отлавливает нажатия мыши и клавиатуры, а затем пересылает их для интерпретации

Ильченко М.М. Программа для дизайна поверхностей Кирхгофа-Плато контейнеру, в котором произошло действие. Программа имеет одно единственное Boundary, которое является контейнером для описания всех точек, линий, рёбер. Контейнеры при помощи функции transform по-разному используют и интерпретируют информацию из Boundary для того, чтобы визуализировать объекты.

### 3.5. Реализация алгоритмов

Некоторые алгоритмы реализуются не точь-в-точь, как это описано во второй главе, поэтому в данном разделе описаны тонкости программной реализации некоторых из них. Все схемы приведены на языке Python 3 с комментариями кода.

#### 3.5.1. Построение сплайнов Катмулла-Рома

Опишем функции, определяющие точки сплайна линии по её контрольным точкам:

```
1 def catmul_rom4(s, p0, p1, p2, p3, tau=0.5):
2     """
3     Функция расчёта координат точки по четырём опорным точкам
4     """
5     S = np.array([s ** i for i in range(0, 4)])
6     M = np.array([[0, 1, 0, 0],
7                  [-tau, 0, tau, 0],
8                  [2 * tau, tau - 3, 3 - 2 * tau, -tau],
9                  [-tau, 2 - tau, tau - 2, tau]])
10    # p.data == [p.x, p.y, p.z]
11    P = np.array([p0.data, p1.data, p2.data, p3.data])
12    return Point(*S.dot(M).dot(P))
```

Схема 1. Значение точки p(s) по четырём точкам (9)

```
1 def catmul_rom3(s, p0, p1, p2, tau=0.5):
2     """
3     Функция расчёта координат точки по трём опорным точкам
4     """
5     S = np.array([s ** i for i in range(0, 4)])
6     M = np.array([[1, 0, 0],
7                  [-tau, tau, 0],
8                  [3 * tau - 3, 3 - 2 * tau, -tau],
9                  [2 - 2 * tau, tau - 2, tau]])
10    # p.data == [p.x, p.y, p.z]
11    P = np.array([p0.data, p1.data, p2.data])
12    return Point(*S.dot(M).dot(P))
```

Схема 2. Значение точки p(s) по трём точкам (10)

```
1 def recount_points(self, step=0.1):
2     """
3     Функция расчёта точек сплайна для линии
4     type(self) == Line
5     """
6     self.points = []
7     # если линия замкнутая
8     if self.closed:
9         # для каждой точки pi
10        for i in range(self.length):
11            # строим точки [pi, pi+1) и последовательно добавляем их в points
```

```

12         self.points.extend([catmul_rom4(s,
13                                 self.control_points[i - 1],
14                                 self.control_points[i],
15                                 self.control_points[(i + 1) % self.length],
16                                 self.control_points[(i + 2) % self.length])
17         for s in np.arange(0, 1, step)])
18     # если линия из одного ребра, ничего достраивать не нужно
19     elif self.length == 2:
20         self.points = self.control_points[:]
21     # если линия не замкнута и содержит больше двух рёбер
22     else:
23         # строим [p0, p1] с производной, совпадающей с ребром
24         self.points.extend([catmul_rom3(s,
25                                     *self.control_points[:3])
26         for s in np.arange(0, 1, step)])
27         # строим [p1, p-2]
28         for i in range(1, self.length - 2):
29             self.points.extend([catmul_rom4(s,
30                                     *self.control_points[i-1:i+3])
31             for s in np.arange(0, 1, step)])
32         # строим [p-1, p-2] с производной, совпадающей с ребром
33         # разворачиваем при добавлении в points
34         self.points.extend([catmul_rom3(s,
35                                     *self.control_points[:-4:-1])
36         for s in np.arange(0, 1 + step, step)][:-1])

```

Схема 3. Перерасчёт сплайновых точек линии

### 3.5.2. Рисование каркаса

В пункте 5 второй главы были оговорены правила для построения линий. В результате получился алгоритм с большим количеством вложенных условий ветвления. Принято решение описывать отдельные случаи предикатами, что увеличивает количество повторений проверок условий, но сильно облегчает простоту и читаемость кода.

```

1 def build_lines(self, ai, active, pi, point):
2     """
3     Функция построения линий для текущей конфигурации Boundary
4     :param ai: либо индекс линии, которой принадлежит active,
5                либо None если active не имеет рёбер
6     :param active: активная точка
7     :param pi: либо индекс линии, которой принадлежит point,
8                либо None если point не имеет рёбер
9     :param point: точка, к которой перейдёт активность
10    """
11    # просто переключаем точку если
12    # - хотя бы одна точка не является крайней для своей незамкнутой линии при том,
13    # что линии разные (2 случая)
14    predicate = ai is not None and ai != pi and \
15                not self.lines[ai].closed and \
16                not self.lines[ai].is_bound_point(active)
17    predicate |= pi is not None and ai != pi and \
18                not self.lines[pi].closed and \
19                not self.lines[pi].is_bound_point(point)
20    if predicate:
21        return
22    # новая линия если
23    # - точки не имеют линий

```

```

24     # - обе точки лежат на замкнутых линиях (могут на одной, а могут на разных
25     predicate = ai is None and pi is None
26     predicate |= ai is not None and pi is not None and \
27         self.lines[ai].closed and self.lines[pi].closed
28     if predicate:
29         self.lines.append(Line(active, point))
30         return
31     # новая линия с частичным замыканием если
32     # - одна точка лежит на замкнутой линии, а другая не имеет линии (2 случая)
33     predicate = ai is None and pi is not None and self.lines[pi].closed
34     predicate |= pi is None and ai is not None and self.lines[ai].closed
35     if predicate:
36         self.lines.append(Line(active, point))
37         if ai is not None:
38             self.lines[-1].closed_with[0] = self.lines[ai]
39         else:
40             self.lines[-1].closed_with[1] = self.lines[pi]
41     # продолжение линии если
42     # - одна из точек является крайней для своей линии, а вторая не имеет линии (2)
43     predicate = pi is None and ai is not None and self.lines[ai].is_bound_point(active)
44     predicate |= ai is None and pi is not None and self.lines[pi].is_bound_point(point)
45     if predicate:
46         if pi is None:
47             self.lines[ai].add(point, active)
48         else:
49             self.lines[pi].add(active, point)
50         return
51     # продолжение линии с частичным замыканием если
52     # - одна из точек является крайней для своей линии, а вторая лежит на замкнутой (2)
53     predicate = pi is not None and ai is not None and \
54         self.lines[pi].closed and self.lines[ai].is_bound_point(active)
55     predicate |= ai is not None and pi is not None and \
56         self.lines[ai].closed and self.lines[pi].is_bound_point(point)
57     if predicate:
58         if self.lines[pi].closed:
59             self.lines[ai].add(point, active, self.lines[pi])
60         else:
61             self.lines[pi].add(active, point, self.lines[ai])
62         return
63     # соединение линий если
64     # - обе точки лежат на краях разных линий
65     predicate = ai is not None and pi is not None and ai != pi and \
66         self.lines[ai].is_bound_point(active) and \
67         self.lines[pi].is_bound_point(point)
68     if predicate:
69         self.lines[ai].connect_line(self.lines.pop(pi), active, point)
70         return
71     # замыкание линии если
72     # - обе точки являются краями одной линии
73     predicate = ai is not None and pi is not None and ai == pi and \
74         self.lines[ai].is_bound_point(active) and \
75         self.lines[pi].is_bound_point(point)
76     if predicate:
77         self.lines[ai].close()
78         return
79     # замыкание линии и создание линии от остатка если
80     # - обе точки на одной линии и только одна из них крайняя (2 случая)
81     predicate = ai is not None and ai == pi and self.lines[ai].is_bound_point(active)
82     predicate |= pi is not None and pi == ai and self.lines[pi].is_bound_point(point)

```

```
83     if predicate:
84         line = self.lines[ai].close(active, point)
85         self.lines.append(line)
86         return
```

Схема 4. Добавление, соединение или замыкание линий при переключении точек

### 3.5.3. Эластичный каркас

Большая часть дискретного описания эластичности каркаса приведено в пункте 2 второй главы и заключается в добавлении обычному каркасу разных векторов.

Самой сложной задачей представляется именно расчёт координат Бишопа, для которых нужно искать матрицу переноса. В нашей реализации для решения системы из уравнений (15) - (18) была использована библиотека символьных вычислений.

```
1  import sympy as sp
2  import numpy as np
3  from sklearn.preprocessing import normalize
4
5  # составляем матрицу параметров 3x3
6  P00, P01, P02, P10, P11, P12, P20, P21, P22 = sp.symbols('P:3:3')
7  P = np.array([[P00, P01, P02],
8               [P10, P11, P12],
9               [P20, P21, P22]])
10 # касательная в предыдущем ребре
11 t_prev = normalize(e_prev)
12 # касательная в текущем ребре
13 t = normalize(e)
14 # t_prev x t
15 ort = np.cross(t_prev, t)
16 # составляем систему уравнений
17 # для этого организуем матрицу, где каждый элемент равен нулю
18 to_solve = np.append(P.dot(t_prev) - t, # (15)
19                      P.dot(ort) - ort) # (16)
20 to_solve = np.append(to_solve,
21                      [P.dot(u_prev).dot(t), # (17, 18)
22                      P02, P12]) # подходящих матриц несколько
23 # sympy решает систему уравнений и выдаёт словарь с ключами Pij
24 solved = sp.solve(to_solve)
25 # преобразуем словарь обратно в матрицу
26 P = np.array(list(map(lambda x: list(map(lambda y: solved[y], x)), P)))
```

Схема 5. Расчёт матрицы переноса

### 3.5.4. Создание меша

При расчёте координат вершин меша используется размер шага  $d$ . Сложность расчёта шага заключается в сравнении расстояний между точками и поиске наибольшего общего делителя для дробных чисел.

```
1  def gcd(*arr):
2      """
3      Рассчёт НОД для len(arr) дробных чисел
4      :param arr: список чисел
5      :return: наибольший общий делитель
6      """
7      # приводим float числа к str и считаем количество знаков после точки
```

```
8 fl = map(lambda a: 0 if '.' not in str(a) else len(str(a).split('.')[1]),
9          np.array(arr))
10 # чтобы все числа стали целыми нужно умножить на 10^max(fl)
11 mul = 10 ** max(fl)
12 # приводим к целым и сортируем по возрастанию
13 arr = sorted(list(map(lambda x: int(x * mul), arr)), reverse=True)
14 # если всего 2 числа, реализация алгоритма Эвклида
15 if len(arr) == 2:
16     # важно поделить на mul, чтобы привести число к исходному виду
17     return (arr[0] if arr[1] == 0 else gcd(arr[1], arr[0] % arr[1])) / mul
18 # НОД(a, b, c) == НОД(НОД(a, b), c), поэтому итеративно сводим к двум числам
19 res = arr[0]
20 for i in range(1, len(arr)):
21     res = gcd(res, arr[i])
22 # важно поделить на mul, чтобы привести число к исходному виду
23 return res / mul
```

Схема 6. Модифицированный алгоритм Эвклида

```
1 import numpy as np
2
3 # сортированный массив координат x
4 xs = np.array([p.x for p in sorted(pr, key=lambda p: p.x)])
5 # разницы соседних
6 dx = xs[1:] - xs[:-1]
7 # сортированный массив координат y
8 ys = np.array([p.y for p in sorted(pr, key=lambda p: p.y)])
9 # разницы соседних
10 dy = ys[1:] - ys[:-1]
11 # НОД из всех разниц
12 d = gcd(*dx, *dy)
```

Схема 7. Расчёт шага d для каркаса с точками *pr*

## Выводы по главе

В данной главе было описано, каким функционалом должна обладать программа и каким образом она его реализует. Описан способ разделения программы на две части при помощи контейнеров. Приведён код реализации основных алгоритмов и

### Заключение

В рамках выпускной квалификационной работы были изучены задачи существования и поиска минимальных поверхностей, а также различные подходы к созданию гибких каркасов и их преобразования.

Для изучения теоретических основ предметной области были использованы более 30 источников, одни из которых представляют собой современные интерпретации и возможности использования задачи [23], а другие являются первыми доказательствами задачи Плато, существующей уже более 250 лет [25].

Задачам Плато и поиска минимальных поверхностей посвящено большое количество работ, в которых приводятся разные способы описания, доказательства и решения этих задач. В то же время, если искать информацию о задаче Кирхгофа-Плато, находится очень мало источников, так как эта модификация задачи использует один из современных подходов к использованию эластичных каркасов, по которым тоже не так много источников.

В результате проведённого анализа источников, посвящённых созданию трёхмерных гибких каркасов, не было выявлено лучшего алгоритма. Поэтому было принято решение создавать двумерные ломаные, которые можно сгладить и преобразовать в 3D. Для сглаживания заданной опорной ломаной, были выбраны сплайны Катмулла-Рома, которые проходят через все опорные точки ломаной и имеют непрерывную производную первого порядка. Для преобразования в 3D используется простейшая рандомизация координаты  $z$  контрольных точек.

Для расчёта минимальной поверхности решено использовать итерационные приближённые вычисления на основе физических свойств мыльных плёнок и описывающих их дифференциальных уравнений.

На основе проделанной работы было разработано техническое задание (Приложение А), выбраны технологии реализации, спроектирован интерфейс программы и создана программа-визуализатор.

Программа реализована на языке Python с использованием OpenGL. При выборе технологий реализации было проведено сравнение различных библиотек, в том числе графических. В итоге была выбрана именно `pyglet`, в частности из-за своей простоты, но такие библиотеки, как `kivy` и `PyQT` тоже очень интересны и способны на большее, чем сам `pyglet`. Единственная по-настоящему серьёзная проблема возникла из-за Retina дисплея, так как функция `glViewport` неправильно делит экран, и это нигде не документировано.

В программе были реализованы алгоритмы создания эластичного каркаса, поиска минимальных поверхностей и визуализации каркаса и соответствующей ему трёхмерной поверхности в пространстве.

Кроме технического задания, в рамках работы к программному продукту была разработана техническая документация, состоящая из руководства оператора (Приложение Б), программы и методики испытаний (Приложение В), и текста программы (Приложение Г).

Путей для продолжения исследований и работ по данной теме много. Не первый раз сказано, что тема достаточно новая, несмотря на богатую историю. Будет появляться всё больше новых моделей эластичных кривых и вариаций задачи Плато. Даже если взять только текущие конфигурации и развивать их в сторону трёхмерной печати, получаются очень интересные результаты, как показали авторы [23]. Кроме того, развивать работу можно не только расширением теории, на которую она опирается, но и увеличением функционала самой программы и её инструментов дизайна.

**Список использованных источников**

1. Bergou M., Wardetzky M., Robinson S., Audoly B., Grinspun E. Discrete elastic rods // ACM Transactions on Graphics, Vol. 27, No. 3, August 2008. pp. 63:1-63:12.
2. Bertails F., Audoly B., Cani M., Querleux B., Leroy F., Lévêque J. Super-helices for predicting the dynamics of natural hair // ACM Transactions on Graphics, Vol. 25, No. 3, July 2006. pp. 1180-1187.
3. Blender Foundation web-site [Электронный ресурс] // Blender Foundation web-site: [сайт]. URL: <https://www.blender.org> (дата обращения: 20.03.2018).
4. Catmull E., Clark J. Recursively generated B-spline surfaces on arbitrary topological meshes // Computer-Aided Design, Vol. 10, No. 6, November 1978. pp. 350-355.
5. Catmull E., Rom R. A class of local interpolating splines // In: Computer Aided Geometric Design / Ed. by BARNHILL R.E., RIESENFELD R.F. Salt Lake City: Academic Print, 1974. pp. 317–326.
6. Concus P. Numerical Solution of the Minimal Surface Equation // Mathematics of Computation, Vol. 21, No. 99, July 1967. pp. 340-350.
7. Courant R. The existence of minimal surfaces of given topological structure under prescribed boundary conditions // Acta Mathematica, Vol. 72, 1940. pp. 51-98.
8. Dill E.H. Kirchhoff's theory of rods // Archive for History of Exact Sciences, Vol. 44, No. 1, March 1991. pp. 1-23.
9. Douglas J. Solution of the problem of Plateau // Transactions of the American Mathematical Society, Vol. 33, No. 1, 1931. pp. 263-321.
10. Dziuk G., Hutchinson J.E. The Discrete Plateau Problem: Algorithm and Numerics // Mathematics of Computatiun, Vol. 68, No. 225, January 1999. pp. 1-23.
11. Felicia B. R.Y. Minimal Surfaces with an Elastic Boundary // Annals of Global Analysis and Geometry, Vol. 19, No. 1, March 2001. pp. 1-9.
12. Fomenko A.T. The Plateau Problem: Historical Survey. Williston: Gordon and Breach Science Publishers, 1989.
13. Geometry Engine Open Source [Электронный ресурс] // Geometry Engine Open Source: [сайт]. URL: <http://trac.osgeo.org/geos/> (дата обращения: 25.03.2018).
14. Giomi L., Mahadevan L. Minimal surfaces bounded by elastic lines // The Royal Society, March 2012.

15. Giulio G. Giusteri L.L.E.F. Solution of the Kirchhoff–Plateau Problem // Journal of Nonlinear Science, Vol. 27, No. 3, June 2017. pp. 1043–1063.
16. Giulio G. Giusteri P.F.E.F. Instability Paths in the Kirchhoff–Plateau Problem // Journal of Nonlinear Science, Vol. 25, No. 4, August 2016. pp. 1097–1132.
17. Glooeey documentation [Электронный ресурс] // readthedocs: [сайт]. URL: [http://glooeey.readthedocs.io/en/latest/getting\\_started.html](http://glooeey.readthedocs.io/en/latest/getting_started.html) (дата обращения: 25.03.2018).
18. Harrison J. Soap Film Solutions to Plateau’s Problem // Journal of Geometric Analysis, Vol. 24, No. 1, January 2014. pp. 271-297.
19. Lipkovski J.A., Lipkovski A.T. Form-Finding Software and Minimal Surface Equation: a Comparative Approach // Filomat, Vol. 29, No. 10, 2015. pp. 2447-2455.
20. Loop C. Smooth Subdivision Surfaces Based on Triangles // Mas-ter’s thesis, Department of Mathematics, University of Utah. August 1987.
21. NumPy web-site [Электронный ресурс] // NumPy: [сайт]. URL: <http://www.numpy.org> (дата обращения: 15.03.2018).
22. O’Reilly O. Kirchhoff’s Rod Theory // In: Interaction of Mechanics and Mathematics. Springer, Cham, 2017. pp. 187-268.
23. Perez J., Otaduy M.A., Thomaszewski B. Computational design and automated fabrication of kirchhoff-plateau surfaces // ACM Transactions on Graphics, Vol. 36, No. 4, July 2017. pp. 62:1-62:12.
24. Pyglet documentation [Электронный ресурс] // readthedocs: [сайт]. URL: <http://pyglet.readthedocs.io/en/pyglet-1.3-maintenance/> (дата обращения: 20.03.2018).
25. Rado T. On Plateau's problem // Annals of Mathematics, Vol. 31, No. 3, July 1930. pp. 457-469.
26. Reifenberg E.R. Solution of the Plateau problem for m-dimensional surfaces of varying topological type // Acta Mathematica, Vol. 104, No. 1-2, 1960. pp. 1-92.
27. Shapely documentation [Электронный ресурс] // readthedocs: [сайт]. URL: <http://shapely.readthedocs.io/> (дата обращения: 20.03.2018).
28. Shi Y., Hearst J.E. The Kirchhoff elastic rod, the nonlinear Schrödinger equation, and DNA supercoiling // The Journal of Chemical Physics, Vol. 101, No. 6, June 1994. pp. 5186–5200.
29. Spillmann J., Teschner M. Cosserat Rod Elements for the Dynamic Simulation of One-Dimensional Elastic Objects // SCA '07 Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation. San Diego, California. 2007. pp. 63-72.

30. SymPy documentation [Электронный ресурс] // SymPy: [сайт]. URL: <http://docs.sympy.org/latest/index.html> (дата обращения: 20.03.2018).
31. Yi-chao Chen E.F. Stability and Bifurcation of a Soap Film Spanning a Flexible Loop // Journal of Elasticity, Vol. 116, No. 1, June 2014. pp. 75-100.
32. Григорьева Е., Клячин А., Клячин В. Алгоритмы идентификации границы и визуализация задачи Плато в среде Blender // Научная визуализация, Vol. 9, No. 4, 2017. pp. 13-25.
33. Клячин А., Клячин В., Григорьева Е. Визуализация расчёта формы поверхностей минимальной площади // Научная визуализация, Vol. 6, No. 2, 2014. pp. 34-42.
34. Математики решили задачу о мыльных пленках на леске [Электронный ресурс] // N+1: [сайт]. [2017]. URL: <https://nplus1.ru/news/2017/04/05/soap-film-problem> (дата обращения: 20.03.2018).
35. Скворцов А. Триангуляция Делоне и её применение. Томск: Издательство Томского университета, 2002. 1-128 pp.
36. Сосинский А. Мыльные пленки и случайные блуждания. Москва: МЦНМО, 2012. 1-20 pp.
37. Тхи Д.Ч. Мультиварианты и классические многомерные задачи Плато // Известия РАН. Серия математическая, Vol. 44, No. 5, 1980. pp. 1031-1065.