Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Московский государственный технический университет имени Н.Э. Баумана» (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления(ИУ)

КАФЕДРА Теоретическая информатика и компьютерные технологии(ИУ-9)

СРЕДА МОДЕЛИРОВАНИЯ ДВУМЕРНОЙ ШКОЛЬНОЙ ФИЗИКИ

Студент Лапатин Владимир Владимирович

Группа ИУ9-51

Руководитель

от МГТУ им Н. Э. Баумана

_____ Коновалов А. В. _____

Должность Подпись

Москва, 2017

Оглавление

Введение

Данная работа посвящена математическому моделированию и касается моделирования двумерной школьной механики. Любое моделирование систем реального мира является чрезвычайно актуальным в наше время. Это область, лежащая на грани математики и информатики. В результате для работы в этой области необходимо и уметь работать с математическим аппаратом, и иметь определенные навыки в области программирования, потому что, если говорить о прикладной программе, моделирующую определенную систему, то программисту мало придумать определенную модель. Ему нужно создать стабильную, расширяемую систему, которая реализует необходимую логику и позволяет пользователю взаимодействовать с ней.

Цель данной курсовой работы заключается в изучении математического аппарата, способного смоделировать двумерную школьную механику и в получении опыта разработки крупной программы.

В первой главе рассматривается разработка интерфейса и базовой архитектуры приложения. Эта часть крайне важна, потому что в ней закладывается основа для всей будущей программы.

Во второй главе рассматривается разработка математического аппарата. На этом этапе определяется, каким способом будет решаться поставленная задача, какие методы и алгоритмы будут для этого применяться.

В третьей главе описывается процесс реализации разработанного математического аппарата непосредственно в коде программы. Также в данном разделе приводятся проблемы, с которыми пришлось столкнуться при разработке, и их решения.

В заключительной, четвертой, главе описано тестирование разработанной системы. В ней будут описаны критерии, по которым можно оценить корректность работы программы и те тесты, которые были созданы для её проверки.

Реализация интерфейса

В современном мире требования к программному обеспечению растут с каждым годом. Одним из самых важных аспектов приложения является интерфейс, а не только основная функциональность.

Даже если опускать такой немаловажный критерий, как внешний вид приложения, к любому интерфейсу имеется список требований, которым он должен удовлетворять. Перечислим некоторые из них:

1. Масштабируемость. Каждое приложение должно уметь работать с разными разрешениями экранов, разным соотношением сторон и в разных формах окна.
2. Кроссплатформенность. В идеальном случае приложение должно работать и выглядеть одинаково хорошо на всех платформах, будь то Windows, Linux или MacOS.
3. Совместимость с сочетаниями клавиш. В текущих операционных системах есть стандартные сочетания клавиш, которые во всех приложениях выполняют одно и то же предсказуемое действие. Например, сочетание клавиш Ctrl+C в операционной системе Windows выполняет копирование выделенного объекта в буфер обмена, а Ctrl+O открывает новый файл.
4. Соответствие стандартам внешнего вида. Каждая операционная система имеет определенные стандарты, которым следует каждое приложение. Например, в различных дистрибутивах Linux, например, Ubuntu, а также в MacOS, различные главные меню располагаются в строке состояния, а в Windows – внутри каждого отдельного окна приложения.

После анализа всех требований было принято решения реализовывать интерфейс приложения на библиотеке Qt для языка C++. Приведем основные преимущества данной библиотеки:

1. Это очень крупный проект, который использует огромное количество разработчиков, что снижает количество потенциальных

проблем данной библиотеки, а также риск того, что она перестанет поддерживаться авторами.

2. Возможность скомпилировать проект под основные операционные системы: Windows, Linux, MacOS с учетом вышеперечисленных требований.

3. Наличие встроенных средств, покрывающих все аспекты работы с интерфейсами, а также с многими другими вещами.

4. Наличие специализированной среды разработки под основные операционные системы.

Язык C++ был выбран по причине того, что, во-первых, библиотека Qt была создана в первую очередь для этого языка, а во-вторых, потому что в приложении будет производиться большое количество вычислений.

Теперь можно приступить к описанию внешнего вида интерфейса и его реализации.

Как можно видеть по рисунку 1, интерфейс имеет четыре логические области:

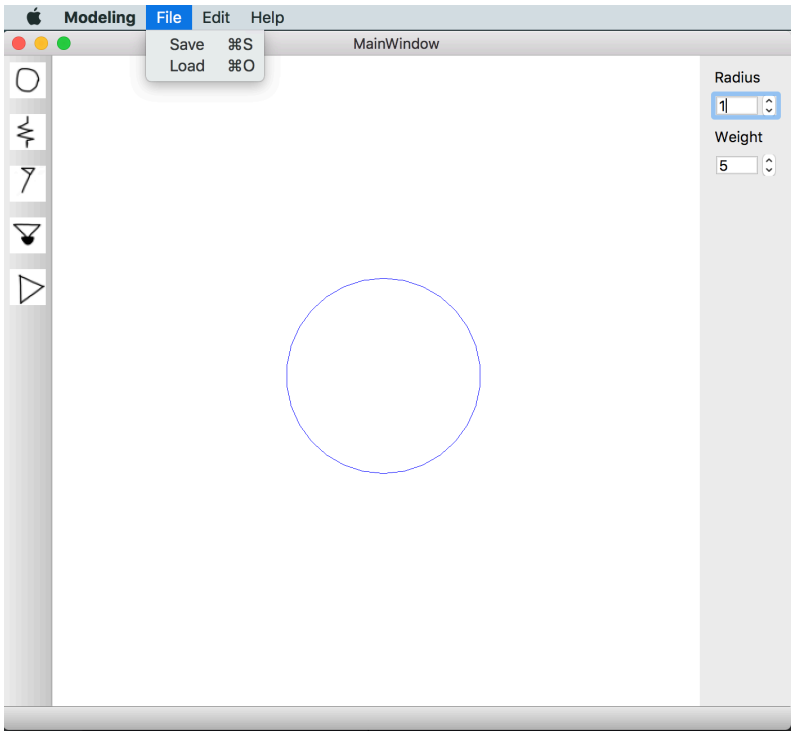

Рисунок 1 - Интерфейс приложения

1. Основная рабочая область. Это та часть интерфейса, куда добавляются элементы механической системы и где они между собой взаимодействуют.
2. Панель инструментов. Здесь собраны все самые важные элементы работы с механической моделью. Это все механические элементы: материальная точка, пружина, стержень и неподвижная точка. Также здесь есть кнопка старт/стоп.
3. Панель свойств. Здесь перечислены все свойства выбранного объекта. Например, в выбранной модели основные свойства материальной точки – это масса и радиус.
4. Панель с главными меню. В этих меню перечислены все функции приложения, которые могут быть вызваны с помощью сочетаний клавиш, например, сохранение и загрузка сцены.

Такая организация интерфейса делает его максимально наглядным и понятным.

Также важным свойством интерфейса является способность уберечь пользователя от неверных действий.

Как видно на рисунке 2, построенная модель является некорректной, так как пружина должна быть присоединена с двух концов к материальной либо к неподвижной точке, а также две материальных точки не могут быть соединены стержнем. Интерфейс показывает, что текущая модель неверна и не дает пользователю запустить некорректную модель.

Архитектура приложения построена по шаблону проектирования MVC, что расшифровывается как модель-представление-контроллер. Данный шаблон предлагает разделить приложения на три логические части:

1. Представление. Оно отвечает за получение данных из модели и отправляет их пользователю. В случае данного приложения представление отвечает за то, чтобы обновить состояние модели и вывести на экран измененную систему.

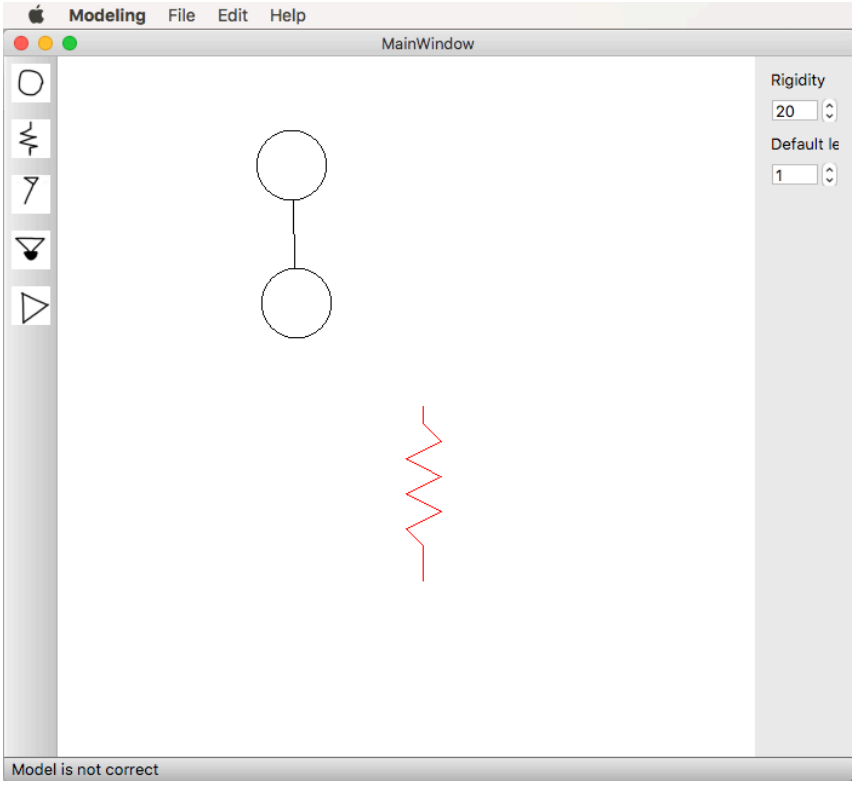

Рисунок 2 - Реакция интерфейса на некорректную модель

2. Контроллер. Обеспечивает взаимодействие между пользователем и моделью. Например, обрабатывает нажатия кнопок и движения мыши, чтобы модифицировать модель.

3. Модель. Она реализует основную логику и хранит в себе данные. В описываемом приложении модель обеспечивает создание, хранение и модификацию систем из различных точек, пружин и стержней, а также выполняет все вычисления с ними.

Основная идея данного шаблона заключается независимости представления от модели. Правильно реализованный шаблон MVC позволит без особого труда портировать программу в браузер, в консоль и в мобильное приложение.

Теперь можно перейти к приложению шаблона MVC на данную программу.

Роль модели выполняют классы ModelingModel, RungeCutta, а также все наследники класса DrawableObject. Представление — это классы MainWindow и OpenGLWidget, а контроллер — MainWindow. В архитектуре Qt-приложения

оказалось проще и удобнее совместить логику представления и контроллера в одном классе.

Теперь перейдем к краткому описанию функций каждого перечисленного класса.

ModelingModel. Этот класс хранит в себе данные о всех объектах математической модели и обеспечивает вычисления над ними.

RungeCutta — отдельный класс, который реализует непосредственно вычисления над данными. Выделение для этого отдельного класса необходимо для обеспечения модульности приложения. Например, если появится необходимость добавить альтернативный вид вычислений, то необходимо будет в новом классе реализовать тот же интерфейс и просто заменить один класс другим.

DrawableObject. Это абстрактный родительский класс, который представляет любой объект, который может быть отображен. Он имеет несколько методов, которые должен имплементировать любой его наследник:

1. draw() — возвращает массив точек, который будет отрисован в классе OpenGLWidget.
2. moveTo(Point point) — передвигает объект на плоскости к выбранной точке.
3. checkCursorInObject(Point point) — проверяет, лежит ли данная точка в площади объекта.
4. write(), read() — методы, записывающие объекты в JSON файл и считывающие данные из него.

Рисунок 3 показывает структуру классов наследников от DrawableObject. Абстрактные подклассы PointableObject и ConnectableObject обозначают объекты, которые соединяют точки и объекты, которые могут быть соединены точками соответственно. Соединять точки могут пружина и стержень, а быть соединенными — материальная и неподвижная точки.

Несколько отрешенно от этой структуры отстоит класс Arrow, который необходим, но который не включен в диаграмму по причине того, что он не относится к элементам математической системы. Этот класс нужен для того,

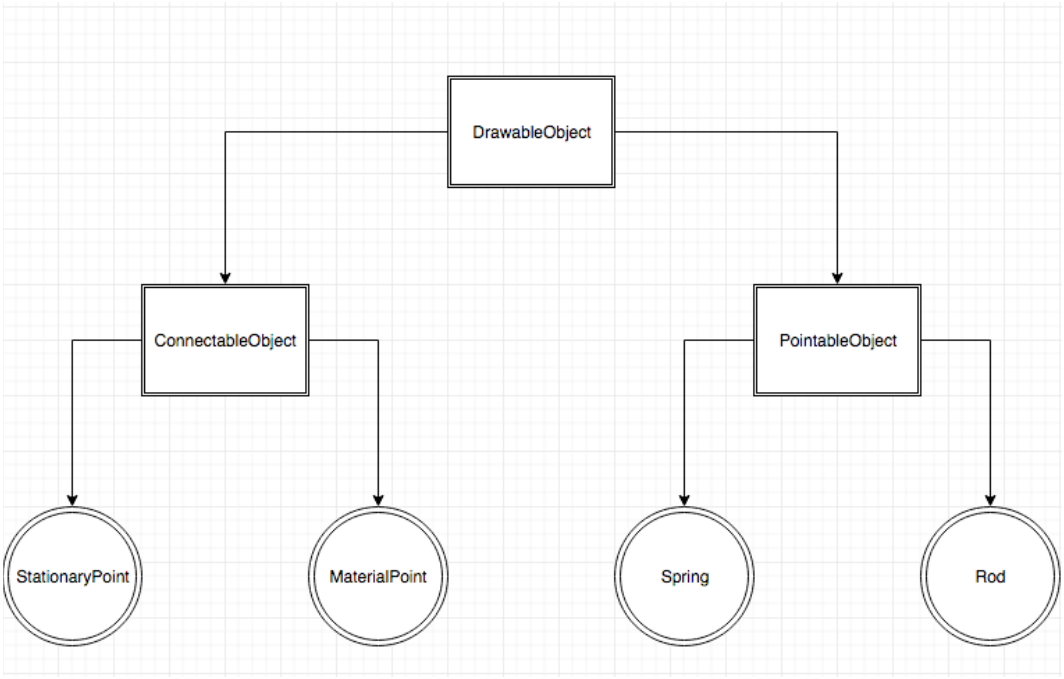

Рисунок 3 - диаграмма классов хранения данных модели

чтобы задавать начальную скорость материальной точки. Общий принцип работы схож с пружиной и стержнем за исключением того, что стержни и пружины соединяют только точки, а стрелка — точку и курсор мыши.

Класс MainWindow соответствует окну, которое и является приложением. Именно в этом классе происходит формирование интерфейса и обработка всех событий.

OpenGLWidget — класс работы с OpenGL. Его задача заключается в том, чтобы получить из ModelingModel массив объектов DrawableObject, получить из них с помощью функции draw массивы точек и отрисовать их.

Также, говоря об интерфейсе, стоит отметить то, как реализовано сохранение модели в файл. В качестве формата используется JSON, главным преимуществом которого является удобство и читаемость.

Разработка математического аппарата для моделирования механики

Прежде чем приступать к описанию решения задачи, необходимо её формализовать. Модель может состоять из четырех элементов: закрепленная точка, материальная точка, стержень и пружина. Перейдем к описанию каждого элемента системы:

1. Стержень в модели может существовать только в рамках математического маятника. Это означает, что с одного конца стержень всегда должен быть присоединен к закрепленной точке, а с другого конца может быть только материальная точка.
2. Пружина. Может соединять между собой закрепленные и материальные точки в любой конфигурации. Является бесконечно растяжимой. Свойства: жесткость, начальная длина.
3. Закрепленная точка. Не может обладать скоростью и передвигаться. К ней может быть присоединено любое количество стержней и пружин.
4. Материальная точка. Может обладать скоростью и передвигаться. К ней может быть присоединено любое количество пружин и только один стержень. Свойства: масса.

Также стоит описать всю систему в целом:

1. Система может состоять из одной и более независимых частей.
2. В системе не предусмотрено столкновение объектов.
3. Система должна моделировать физически корректные взаимодействия между всеми элементами системы.
4. В системе может быть перенастроена сила тяжести, но при этом она всегда будет направленной вертикально.

Теперь стал возможен переход к описанию решения данной задачи.

Для этого было решено использовать уравнение Лагранжа второго рода, связывающее кинетическую и потенциальную энергию системы с помощью системы дифференциальных уравнений.

Уравнение Лагранжа второго рода выглядит следующим образом:

$$\frac{\partial}{\partial t}(\frac{\partial L}{\dot{q}_i}) - \frac{\partial L}{q_i} = 0$$

где L — лагранжиан, T – кинетическая энергия системы, U – потенциальная, q – обобщенные координаты.

Обобщенные координаты – это параметры, описывающие конфигурацию динамической системы относительно некоторой эталонной конфигурации.

В рамках задачи было решено рассмотреть два варианта выбора обобщенных координат.

1. Случай математического маятника. Здесь в качестве обобщенной координаты будет выбираться угол φ отклонения стержня от положения равновесия. В данном случае кинетическую энергию можно записать как $T = \frac{ml^2\dot{\varphi}^2}{2}$, а потенциальную как $U = mglcos\varphi$, где m – масса материальной точки, ℓ – длина стержня, g – сила тяжести, φ– угол между стержнем и осью y по часовой стрелке.
2. Случай свободной материальной точки. В общем случае в качестве обобщенных координат будут приниматься положения материальной точки по вертикали и горизонтали. Тогда кинетическая энергия всегда будет одинаковой: $T = \frac{m(\dot{x}^2 + \dot{y}^2)}{2}$, а потенциальная будет равна: $U = mgy$
3. Случай пружины. В данной модели пружина является невесомой, так что может обладать только потенциальной энергией $U = \frac{k\Delta^2}{2}$, k – жесткость пружины, Δ- растяжение пружины.

На основе этих обобщенных координат становится возможным вывод $\ddot{x}, \ddot{y}, \ddot{\varphi}$.

В рамках выбранной модели кинетическая энергия элемента системы может быть записана в виде $T = \frac{M\dot{q}^2}{2}$, где

$$M = \begin{cases} m_i \text{ в случае материальной точки} \\ m_i l_i^2 \text{ в случае математического маятника} \end{cases}$$

Отсюда видно, что кинетическая энергия системы не зависит от q_i.

Перейдем к рассмотрению потенциальной энергии системы. В случае выбранной модели возможен случай $U = m_i g y_i$ при материальной точке, либо $U = \frac{k\Delta q_i^2}{2}$ в случае пружины, где k = жесткость, Δq_i — растяжение. Значит, потенциальная энергия не зависит от $\dot{q}_i$.

На основе данных наблюдений можно переписать уравнение Лагранжа второго рода:

$$\frac{d}{dt}\left(\frac{\partial(T-U)}{\partial \dot{q}_i}\right) - \frac{\partial(T-U)}{\partial q_i} = 0$$

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_i}\right) + \frac{\partial U}{\partial q_i} = 0$$

$$\frac{d}{dt}\left(\frac{\partial}{\partial \dot{q}_i}\left(\frac{M\dot{q}_i^{\,2}}{2}\right)\right) = M\ddot{q}_i$$

Отсюда, пользуясь уравнением Лагранжа, получаем

$$M\ddot{q}_i = -\frac{\partial U}{\partial q_i}$$

$\ddot{q}_i = -\frac{1}{M}\left(\frac{\partial U}{\partial q_i}\right)$, где U — сумма всех потенциальных энергий, действующих на систему.

Для нашей системы сумма всех потенциальных энергий имеет следующий вид: $U = \sum_{i=1}^{n} \frac{k_i \Delta L_i^2}{2} + \sum_{i=1}^{r} m_i g y_i + \sum_{i=1}^{p} m_i g l_i sin\varphi_i$, где n — количество пружин в системе, r — количество материальных точек, p — количество маятников.

$$\Delta L_i = (L_i - L_0) = \begin{cases} \sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2} - L_{i0} \\ \sqrt{(x_{stat} - l_i sin\varphi_i - x_{mat})^2 + (y_{stat} - l_i cos\varphi_i - y_{mat})^2} - L_{i0} \end{cases}$$

где L_i — длина пружины, L_{i0} — длина пружины в состоянии спокойствия, x_{i2}, y_{i2} — координаты конца пружины, x_{i1}, y_{i1} — координаты начала пружины, x_{stat}, y_{stat} — координаты стационарной точки закрепления стержня, x_{mat}, y_{mat} — координаты материальной точки.

Первая формула используется при использовании в качестве обобщенной координаты x_i, y_i, а вторая — в случае φ_i.

Отсюда, продифференцировав потенциальную энергию системы, можно получить:

$$\ddot{x}_i = -\frac{1}{m_i} \sum_{j=1}^{n} \frac{k_i(\sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2} - L_{i0})(x_{i2} - x_{i1})}{\sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2}}$$

$$\ddot{y}_i = -\frac{1}{m_i}(m_i g + \sum_{j=1}^{n} \frac{k_i(\sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2} - L_{i0})(y_{i2} - y_{i1})}{\sqrt{(x_{i2} - x_{i1})^2 + (y_{i2} - y_{i1})^2}})$$

$$\ddot{\varphi}_i = -\frac{1}{m_i l_i^2}(m_g l_i sin\varphi_i + \sum_{j=0}^{n} \frac{k_i(\sqrt{(x_{stat} - l_i sin\varphi_i - x_{mat})^2 + (y_{stat} - l_i cos\varphi_i - y_{mat})^2} - L_{i0})(-l_i cos\varphi_i(x_{stat} - l_i sin\varphi_i - x_{mat}) + l_i sin\varphi_i(y_{stat} - l_i cos\varphi_i - y_{mat}))}{\sqrt{(x_{stat} - l_i sin\varphi_i - x_{mat})^2 + (y_{stat} - l_i cos\varphi_i - y_{mat})^2}})$$

Следующий шаг — составление системы дифференциальных уравнений размера 3n, где n — количество материальных точек системы. Эта система состоит из приведенных выше $\ddot{x}_i, \ddot{y}_i, \ddot{\varphi}_i$ для каждой материальной точки системы.

После получения системы дифференциальных уравнений, необходимо проинтегрировать её для получения скоростей и координат из выведенных сил,

действующих на каждый элемент системы. Для этого было решено использовать метод Рунге-Кутты четвертого порядка, так как это известный метод численного интегрирования дифференциальных уравнений, дающий приемлемую точность для решения данной задачи.

Перейдем к описанию данного метода.

Пусть $Q, Q_i, f, k_i \in \mathbb{R}^n; x, h \in \mathbb{R}$; n — количество обобщенных координат.

Q — массив, составленный по следующему принципу:

$$Q[6i] = x_i$$
$$Q[6i+1] = \dot{x}_i$$
$$Q[6i+2] = y_i$$
$$Q[6i+3] = \dot{y}_i$$
$$Q[6i+4] = \varphi_i$$
$$Q[6i+5] = \dot{\varphi}_i$$

При этом $Q = f(x, y), Q(x_0) = y_0$.

Тогда каждое следующее состояние системы можно вычислить по формулам:

$$Q_{n+1} = Q_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \text{где}$$
$$k_1 = f(x_n, Q_n)$$
$$k_2 = f(x_n + \frac{h}{2}, Q_n + \frac{h}{2}k_1)$$
$$k_3 = f(x_n + \frac{h}{2}, Q_n + \frac{h}{2}k_2)$$
$$k_4 = f(x_n + h, Q_n + hk_3)$$

где h — величина шага интегрирования по x.

Таким образом становится возможным на основе сил, действующих в системе, скоростей и координат материальных точек посчитать состояние всей системы в любой момент времени.

Реализация интерактивной модели механики

После формального решения задачи можно приступить к её практической реализации. Для этого в коде программы предусмотрено два класса: RungeCutta и ModelingModel, общее назначение которых уже описывалось ранее.

Но перед тем, как приступать к описанию реализации каждого класса, необходимо пояснить несколько общих идей, которые были использованы для решения поставленной задачи. В первую очередь стоит проблема составления и хранения сил, действующих в системе. Сложность заключается в том, что система никак не ограничена в размерах, а значит может содержать произвольное количество элементов, связанных друг с другом произвольным образом. Это означает, что на каждую материальную точку может действовать любое количество сил. Для решения данной задачи было решено использовать замыкания языка C++. Такое решение обусловлено тем, что оно позволяет итеративно составлять силы, действующие на каждую материальную точку. Это достигается за счет возможности замыкания захватывать переменные окружения. В данном случае каждое замыкание захватывает необходимый набор констант, таких как жесткости, веса и т.д., а также составленное ранее замыкание, после чего новая формула суммируется с результатом работы захваченного замыкания. Такой подход позволяет удобно составлять силы, действующие на каждую точку. Также важно отметить, что каждое замыкание имеет аргумент — контейнер systemState, хранящий координаты и скорости системы, составленный по принципу массива Q, описанного в предыдущем разделе. Ниже это решение будет рассмотрено более подробно.

Класс RungeCutta инкапсулирует в себе численное интегрирование массива сил системы. Для работы в класс необходимо передать два контейнера с данными: контейнер systemState и контейнер с замыканиями. Важно отметить, что для хранения и передачи координат и скоростей системы было решено использовать класс стандартной библиотеки языка C++ std::valarray. Его преимущество заключается в том, что для него перегружены арифметические

операции, такие, как сложение контейнеров и умножение контейнера на константу. Подобная реализация позволяет добиться большего удобства и выразительности класса RungeCutta. Более подробную реализацию данного класса можно видеть в Листинге 1.

Для произведения вычислений используется метод rungeCutta(), возвращающий столбец std::valarray с новыми скоростями и координатами системы.

Теперь можно приступить к описанию класса ModelingModel. Краткое описание значения этого класса также было выше.

В первую очередь для понимания работы этого класса стоит описать, как и в каком виде он хранит данные. Как уже описывалось, все объекты математической модели являются наследниками класса DrawableObject.

```
std::valarray<double> RungeCutta::applyPositionsToAccelerations(
     std::valarray<double> args)
{
    std::valarray<double> result(modelState.size());
    for (int i = 0; i < accelerations.size(); i++)
    {
        result[2*i + 1] = accelerations[i](args);
        result[2*i] = args[2*i + 1];
    }
    return result;
}

std::valarray<double> RungeCutta::rungeCutta()
{
    double h = 1.0 / 60.0;
    std::valarray<double> k1 =
         applyPositionsToAccelerations(modelState);
    std::valarray<double> k2 =
         applyPositionsToAccelerations(modelState + h / 2 * k1);
    std::valarray<double> k3 =
         applyPositionsToAccelerations(modelState + h / 2 * k2);
    std::valarray<double> k4 =
         applyPositionsToAccelerations(modelState + h * k3);

    return modelState + h / 6 * (k1 + 2.0 * k2 + 2.0 * k3 + k4);
}
```

Листинг 1 - реализация класса RungeCutta

Структура данного наследования такова, что можно эффективно использовать приведение типов для повышения или понижения уровня абстракции. Другими словами, каждая точка хранит массив объектов типа PointableObject, за которыми скрываются объекты типов Spring и Rod. Но с точки зрения хранения удобнее хранить их вместе, чтобы не писать лишний код. Это лишь один из примеров удобства данного подхода.

Но чрезмерный уровень абстракции также вредит, поэтому все объекты системы хранятся отдельно. Сделано это по причине того, что зачастую необходим доступ только к объектам одного типа. Таким образом, в классе ModelingModel есть четыре массива: QVector<MaterialPoint> matPoints, QVector<Spring> springs, QVector<Rod> rods, QVector<StationaryPoint> statPoints. Использование класса QVector обусловлено тем, что оно обеспечивает приемлемую эффективность и высокое удобство работы с данными.

Работу с математической моделью можно разделить на две логических части.

1. Формирование замыканий. Сюда входят методы createAccelerations, createSpringAccelerations, createRodAccelerations, createSpringAndRodAccelerations, а также вспомогательный метод findIndexOfDrawableByHash.
2. Сбор и применение координат. Если говорить более конкретно, то к этому разделу можно отнести метод getConnectablesPosition, который из всех элементов системы формирует контейнер std::valarray со скоростями и координатами материальных точек, а также метод applySpeedsAndCoordinatesToModel, который применяет вычисленные значения к модели.

Перейдем к описанию работы каждой функции.

Функция createAcceleration занимается тем, что проходит по всем точкам, определяет случай и, в зависимости от этого, выбирают нужный метод.

Пример этой функции в псевдокоде можно увидеть в листинге 2.

```
func createAcceleration:
  for point in matPoints:
    for conn in point.getPointableObjects():
      if conn.getType == SPRING:
        if point.isConnectedToRod():
          createSpringAndRodAccelerations
        else:
          createSpringAccelerations()
      else:
        createRodAccelerations()
```

Листинг 2 - функция createAcceleration в псевдокоде

Далее стоит описать функции создания отдельных формул: createSpringAccelerations, createRodAccelerations и createSpringAndRodAccelerations. Но описывать их отдельно и детально либо же приводить код функций целиком нецелесообразно, так как эти методы довольно громоздкие и невыразительные. Поэтому здесь будет приведено описание общей логики работы, а также листинги с замыканиями, так как это самая сложная часть всей программы.

Не вдаваясь в детали реализации, все три функции состоят из сбора констант, таких как веса, жесткости, длины, индексы в массивах и т.д., из выбора актуальной обобщенной координаты и реализация искомой формулы.

Отдельно стоит отметить, насколько важен выбор актуальной обобщенной координаты. Для правильной реализации формулы необходимо знать, какие координаты описывают текущее состояние материальной точки. Например, если к материальной точке присоединен стержень, то есть имеется математический маятник, то его положение однозначно определяется длиной стержня и углом отклонения. А если точка соединена только с пружинами или не соединена ни с чем, то её положение однозначно определяется координатами x и y. Это означает, что каждая формула во всех описываемых функциях должна иметь два варианта — для материальной точки с актуальной обобщенной координатой φ_i и с актуальной обобщенной координатой x_i, y_i.

Рисунок 4 наглядно демонстрирует два этих случая. В случае левой системы все обобщенные координаты будут выражаться x_i и y_i. А в случае правой системы координаты верхней материальной точки будут выражаться через φ_i. Это означает, что в своей формуле нижняя точка также должна использовать координату, посчитанную через φ_i.

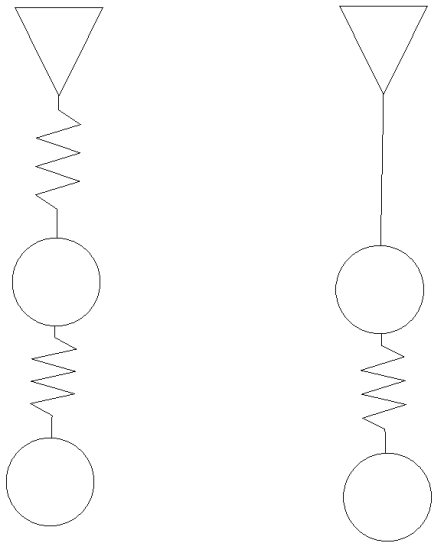

Рисунок 4 — пример систем с разными обобщенными координатами

Листинг 2 показывает, как реализовано вычисление силы, с которой действует пружина на две материальные точки, что соответствует одному из случаев функции createSpringAcceleration. Здесь k — жесткость пружины, m — масса материальной точки, L0 — длина пружины в состоянии спокойствия, x1Index и x2Index — индексы тех двух материальных точек, которые соединяет пружина, для их нахождения используется метод findIndexOfDrawableByHash. capturingAccelerationX и capturingAccelerationY — это уже посчитанные замыкания. Важно отметить, что изначально замыкание для x_i и φ_i инициализируются на возвращение нуля, а замыкание для y_i возвращает $-g$, где g — сила тяжести.

Массив args — это массив координат и скоростей, который собирается упомянутым выше методом getConnectablesPosition.

```
[capturingAccelerationX, k, m, L0, x2Index, x1Index](
     std::valarray<double> args)
{
     double x2 = args[6*x2Index];
     double y2 = args[6*x2Index + 2];
     double x1 = args[6*x1Index];
     double y1 = args[6*x1Index + 2];
     double square = std::hypot(x2 - x1, y2 - y1);
     return capturingAccelerationX(args) +
          (-1.0) / m * k * (square - L0) * (x2 - x1)  / square;
}
[capturingAccelerationY, k, m, L0, x2Index, x1Index](
     std::valarray<double> args)
{
     double x2 = args[6*x2Index];
     double y2 = args[6*x2Index + 2];
     double x1 = args[6*x1Index];
     double y1 = args[6*x1Index + 2];
     double square = std::hypot(x2 - x1, y2 - y1);
     return capturingAccelerationY(args) +
          (-1.0) / m * k * (square - L0) * (y2 - y1) / square;
}
```

Листинг 2 - пример замыкания для двух материальных точек, связанных пружиной

Заканчивая описание замыканий, стоит отметить, что в массиве args хранятся только материальные точки. Это означает, что если с другого конца пружины вместо материальной точки будет находиться стационарная точка, то найти ее индекс не выйдет. В таком случае происходит захват координат этой точки в замыкание как констант.

Метод getConnectablesPosition является максимально простым в реализации. Его задача заключается в том, чтобы пройтись по массиву matPoints и на его основе создать std::valarray.

Несколько более интересным является метод applySpeedsAndCoordinatesToModel. В математической модели координаты точек хранятся исключительно в виде двух координат: x, y. Это означает, что в случае математического маятника необходимо получить искомое из угла.

Листинг 3 демонстрирует реализацию данного метода.

```
void ModelingModel::applySpeedsAndCoordinatesToModel(
     std::valarray<double> arr)
{
    for (int i = 0; i < matPoints.size(); i++)
    {
        if (arr[i * 6 + 4] == 0.0 && arr[i * 6 + 5] == 0.0)
        {
            matPoints[i]->setX(arr[i * 6]);
            matPoints[i]->setSpeedX(arr[i * 6 + 1]);
            matPoints[i]->setY(arr[i * 6 + 2]);
            matPoints[i]->setSpeedY(arr[i * 6 + 3]);
        }else
        {
            Rod *rod = (Rod*) matPoints[i]->getRod();
            MaterialPoint *matPoint;
            StationaryPoint *statPoint;
            fillConnectablesByRod(rod, matPoint, statPoint);
            double length = rod->getDefaultLength();
            double x = statPoint->getCenter().x -
               std::sin(arr[i * 6 + 4]) * (length);
            double y = statPoint->getCenter().y +
               std::sin((arr[i * 6 + 4] - PI / 2.0)) * (length);
            double angularSpeed = arr[i * 6 + 5];
            matPoint->setX(x);
            matPoint->setY(y);
            matPoint->setAngularSpeed(angularSpeed);
        }
    }
}
```

Листинг 3 - реализация метода applySpeedsAndCoordinatesToModel

Тестирование

Тестирование — это необходимый этап разработки каждого проекта. Без тестирования нельзя быть уверенным в правильности работы приложения, в его стабильности. Так что этот этап требует не меньшего внимания, чем непосредственно написание самого приложения.

Можно сказать, что тестирование - это процесс выполнения программ с целью обнаружения ошибок. Чем лучше и полнее будут написаны тесты для приложения, тем больше ошибок будет выявлено. Поэтому в первую очередь необходимо определить те критерии, по которым будет определяться правильность работы программы. В данном случае было решено тестировать два параметра — визуальная правильность и выполнение закона сохранения энергии. Пользоваться вторым пунктом позволяет тот факт, что в моделируемой системе отсутствуют внешние и диссипативные силы. Этот факт сильно облегчает проверку закона сохранения энергии. Проверка обеспечивается созданием систем, для которых и проверяется правильность работы. В данном случае были рассмотрены базовые случаи всех взаимодействий, а также системы, которые включают в себя несколько разных взаимодействий и все возможные взаимодействия.

Для тестирования в приложение было добавлено несколько новых функций: вычисление суммарной кинетической энергии, вычисление суммарной потенциальной энергии и логгирование данных в формате CSV. В лог выводятся все координаты и скорости на каждом этапе вычислений, а также кинетическая и потенциальная энергии. Данный формат позволяет просто строить графики по выведенным данным.

Вычисление энергий производится по формулам

$$W = \sum_{i=0}^{n} \frac{m_i \sqrt{\dot{x}_i^{\,2} + \dot{y}_i^{\,2}}}{2} + \sum_{i=0}^{h} \frac{m_i \dot{\varphi}_i^{\,2}}{2}$$

$$E = \sum_{i=0}^{r} \frac{k_i \Delta x_i^2}{2} + \sum_{i=0}^{l} m_i g y_i$$

где n — количество материальных точек, не соединенных пружиной, h — количество материальных точек, соединенных пружиной, r — количество пружин, l — количество материальных точек.

Также для тестирования приложения был написан скрипт на языке Python для открытия файлов логов и вывода графиков. Скрипт позволяет строить графики энергий — кинетическую, потенциальную и их сумму, а также строить графики по каждому отдельному столбцу. Результаты работы данного скрипта будут демонстрироваться ниже в показе графиков для каждой системы.

Для тестирования приложения был выбран ряд тестов, который является наиболее показательным и который позволяет проще всего оценить правильность работы программы.

Первый тест — пружинный маятник. Его ожидаемое поведение — колебание энергий по синусоидам. Результаты можно увидеть на рисунке 5.

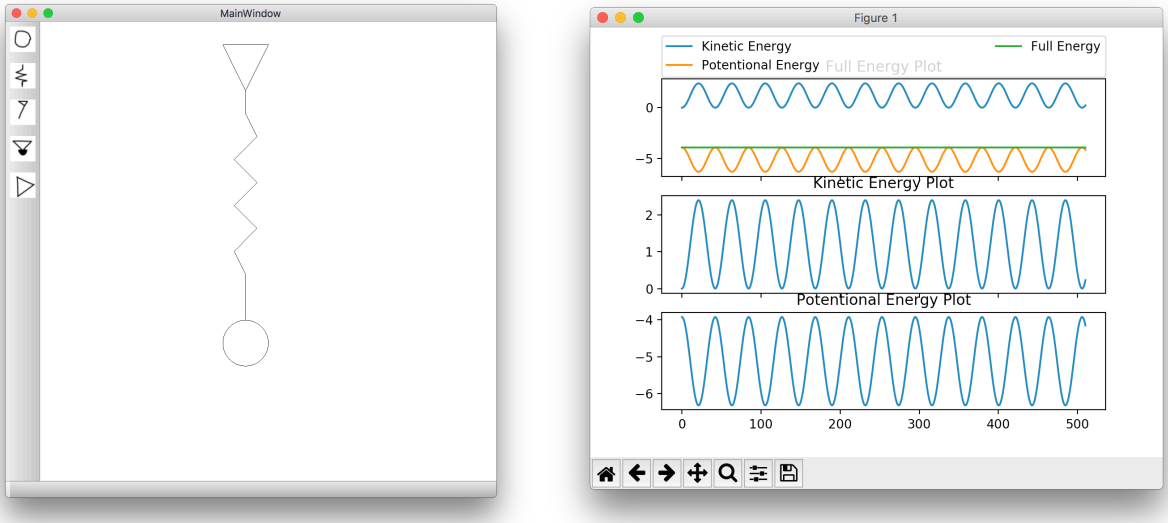

Рисунок 5 - Пружинный маятник

Второй тест — математический маятник при малых колебаниях. Ожидаемое поведение — также колебание энергии по синусоидам. Его можно видеть на рисунке 6.

Третий тест — математический маятник при полных колебаниях. Ожидаемое поведение — совпадение начальной скорости, необходимой для

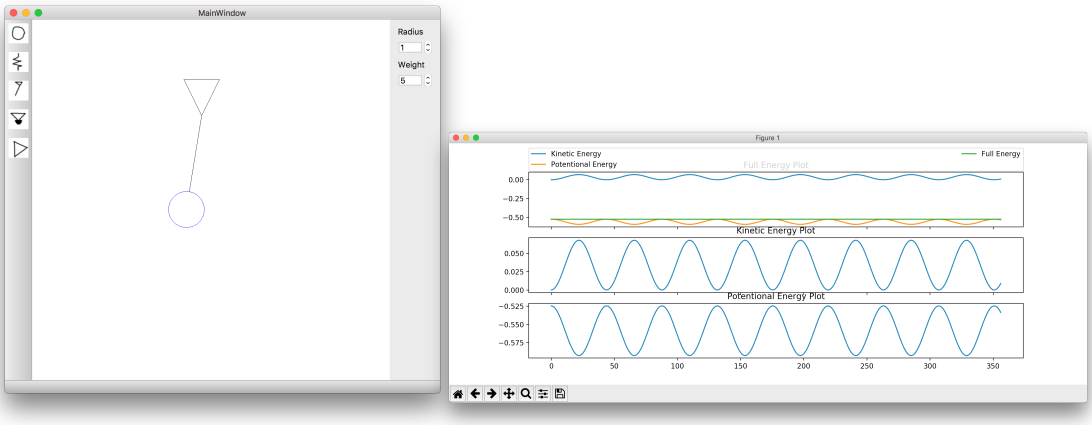

Рисунок 6 - математический маятник, малые колебания

совершения полного оборота с расчетной. Вывести эту скорость можно, пользуясь законом сохранения энергии:

$$\frac{mv^2}{2} = 2mgl$$

$$v^2 = 4gl$$

$$v = 2\sqrt{gl}$$

$$v = wR$$

$$wl = 2\sqrt{gl}$$

$$w = 2\sqrt{\frac{g}{l}} = 8.85$$

Действительно, скорость совпала. Так как выше приведено округленное значение, то становится естественным поведение маятника, при котором при начальной скорости 8.85 он еще не совершает полный оборот, а при скорости 8.86 уже совершает его. Данному случаю соответствует рисунок 7.

Следующий тест — «колыбель Ньютона». Это механическая система, призванная демонстрировать преобразование кинетической энергии в потенциальную и обратно. В данной модели отсутствуют столкновения между объектами, так что вместо столкновений придется использовать пружины. Это накладывает определенные изменения по сравнению с оригинальной системой, но, тем не менее, «колыбель Ньютона» позволяет эффективно проверить

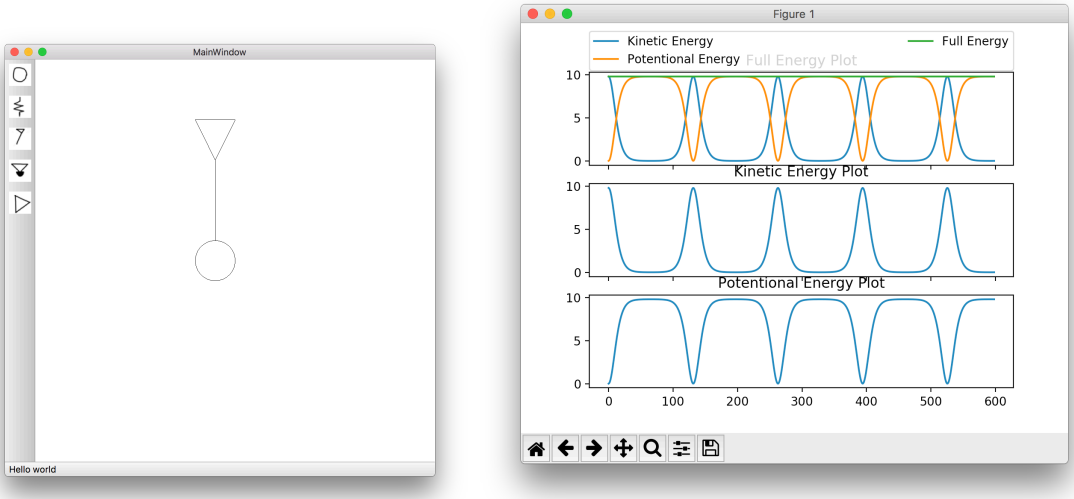

Рисунок 7 - Математический маятник, полный оборот

правильность работы системы. В этой и последующих системах становится проблематично наложить какие-либо требования на график, как это было сделано в пружинном или математическом маятниках. Поэтому ожидаемое поведение системы — сохранение энергии. Пример «колыбели Ньютона» можно видеть на рисунке 8.

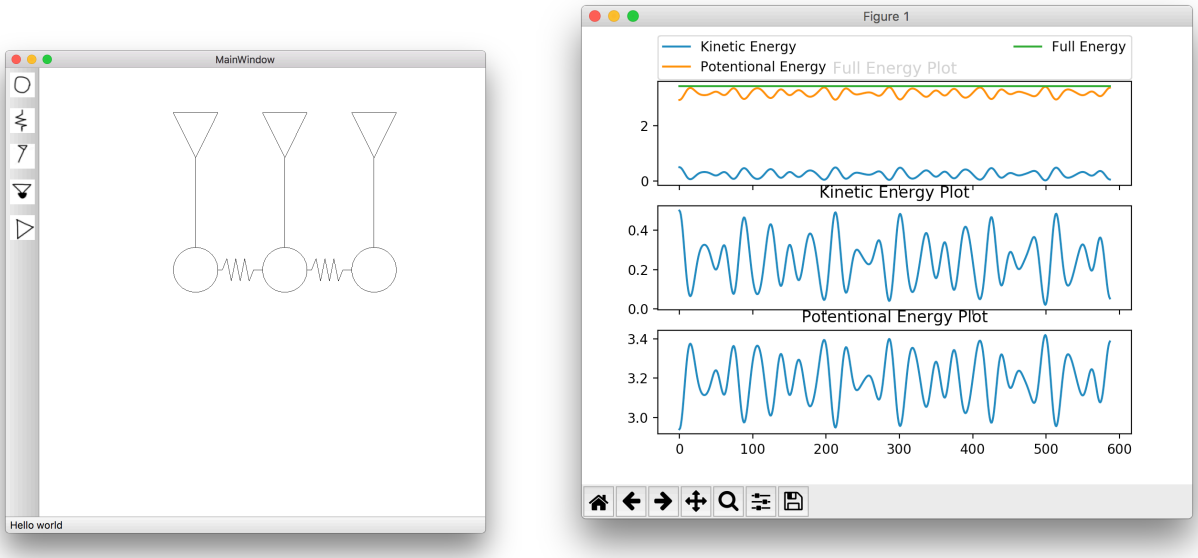

Рисунок 8 - «Колыбель Ньютона»

Пятый тест — система с двумя стержнями для проверки правильности выбора актуальных обобщенных координат. Ожидаемый результат — сохранение энергии. Этот тест можно увидеть на рисунке 9.

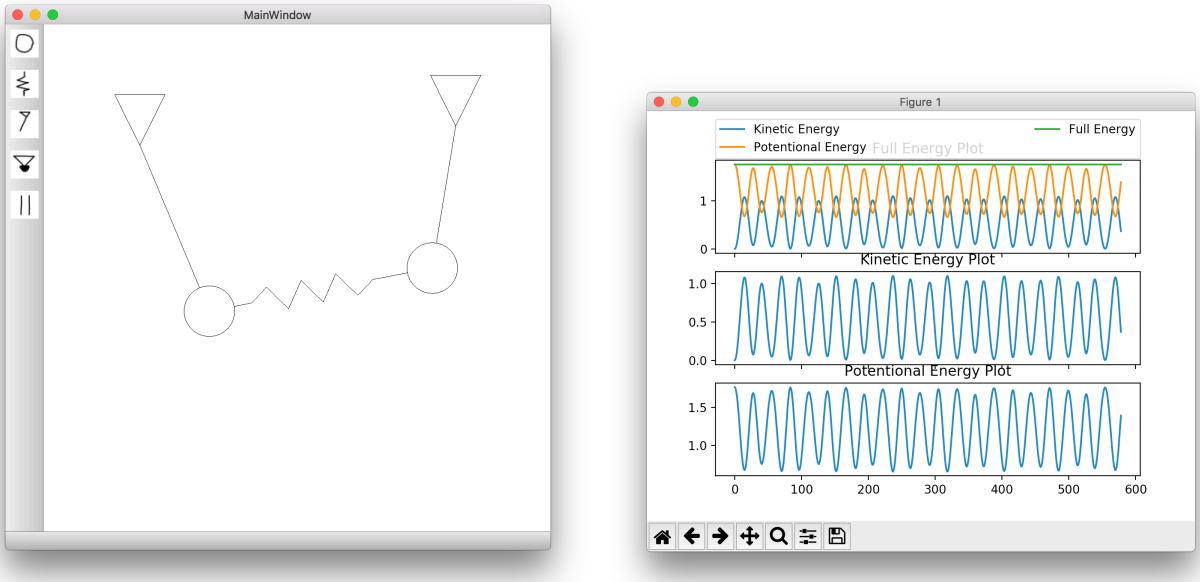

Рисунок 9 - Произвольная система с двумя стержнями

Следующий тест призван проверить все взаимодействия, которые могут быть смоделированы выбранной математической моделью. Ожидаемый результат — сохранение энергии. Рассмотренная произвольная система изображена на рисунке 10.

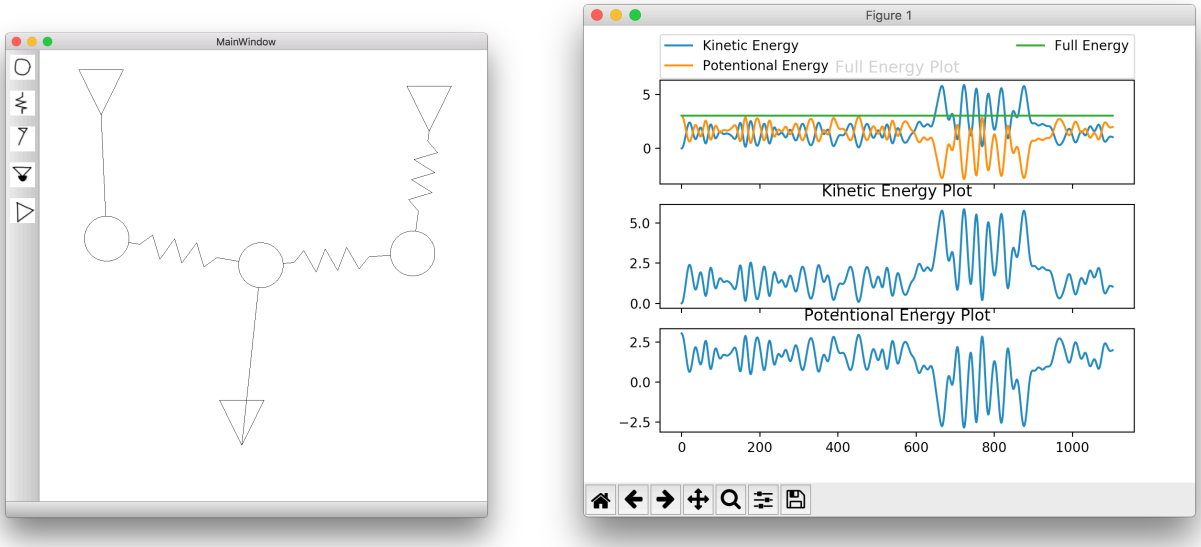

Рисунок 10 - Произвольная система

Заключение

Математическое моделирование систем реального мира является крайне важным в наше время. В ходе данной курсовой работы было произведено моделирование двумерной школьной механики и реализована программа на языке C++ с использованием библиотеки Qt.

Для решения поставленной задачи было использовано уравнение Лагранжа второго рода для моделирования физических систем, состоящих из математических точек, пружин, закрепленных точек и стержней. Это решение требует расписать полную энергию выбранной системы, после чего из уравнения Лагранжа выразить ускорение и проинтегрировать полученное выражение, получив скорости и координаты системы. Для интегрирования был выбран метод численного интегрирования Рунге-Кутты четвертого порядка.

Для проверки созданной программы был реализован подсчет полной энергии системы. Если для введенной системы выполняется закон сохранения энергии, значит программа работает верно.

Таким образом, в курсовой работе был разработан, реализован и протестирован достаточно крупный проект, выполняющий интерактивное моделирование двумерной школьной механики.

Список литературы

1. Ландау Л.Д., Лифшиц Е.М. Механика. — Издание 4-е, исправленное. — М.: Наука, 1988. — 215с.

2. Документация по Qt[Электрон. ресурс] // Режим доступа: http:// doc.qt.io/qt-5/, *свободный. — Загл. с экрана.(Дата обращения 05.12.2017)*