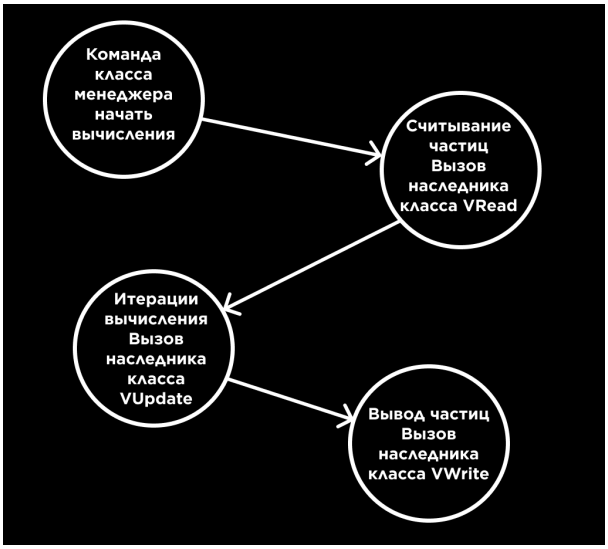
Руководство программиста

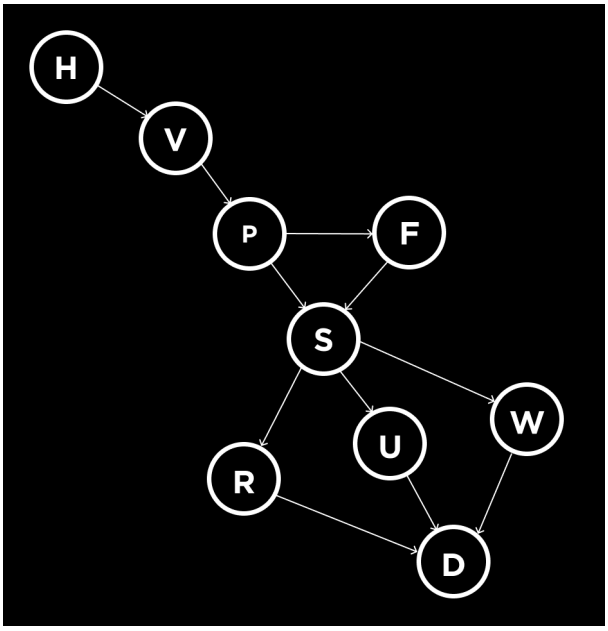
Общее описание

Алгоритм работы фреймворка хорошо показана на данной схеме.



Сначала вызывается функция simulate класса Dynamic, которая последовательно считывает, моделирует и выводит, используя наследников классов VRead, VUpdate и VWrite соответственно. Фреймворк состоит из 3 абстрактных классов, реализующий интерфейсы классов чтения, обновления и записи: VRead, VUpdate, VWrite, 3 класса контейнера: SystemState, Particle, Vector3, 2 заголовочных файлов: headers.h с описанием скорости света и подключением нужных библиотек и functions.h с описанием функций фабрик для генерации сил, а так же 1 классом менеджером - Dynamic, управляющий процессами в работе фреймворка.

Граф зависимости между классами выглядит так



> Здесь введены следующие обозначения:
>
> > headers.h - H
> > Vector3.h - V
> > Particle.h - P
> > forces.h - F
> > SystemState.h - S
> > VRead.h - R
> > VUpdate.h - U
> > VWrite.h - W
> > Dynamic.h - D

Интерфейсные классы

Всего в фреймворке описано 3 интерфейсных класса.

VRead - модуль для чтения

В классе *VRead* определены 3 функции:

- ввод вектора - без параметров, возвращает *Vector3*

- ввод частицы - без параметров, возвращает *Particle*
- ввод состояния системы - принимает кол-во ввозимых частиц, возвращает *SystemState*
- Функции, описанные в *VRead*:

```
virtual Vector3 readVector3() = 0;

virtual Particle readParticle() = 0;

virtual SystemState readSystem(int num_of_particles) = 0;
```

VUpdate - модуль для обновления

В классе *VUpdate* определена 1 функция для обновления частиц - *updateSystemState*, принимает на вход состояние системы, внешнюю силу, силу взаимодействия 2 частиц и промежуток времени. Очень важно, чтобы промежуток времени был достаточно мал. Функция возвращает новое состояние системы.

Функция, описанная в *VUpdate*:

```
virtual SystemState updateSystem(
                                 const SystemState& state,
                                 const
std::function<Vector3(Particle)>& external_f,
                                 const
std::function<Vector3(Particle, Particle)>& f_btw_two_par,
                                 double d_time) = 0;
```

Переменные, описанные в функции *updateSystem*:

- *state* - состояние системы
- *external_f* - внешняя сила

- _f_btw_two_par - сила взаимодействия двух частиц
- *d_time* - рассматриваемый промежуток времени

VWrite - модуль для вывода

В классе *VWrite* определены 3 функции:

- вывод вектора - принимает *Vector3*, ничего не возвращает
- вывод частицы - принимает *Particle*, ничего не возвращает
- вывод состояния системы - принимает *SystemState*, ничего не возвращает

Функции, описанные в VWrite:

```
virtual void printVector3(const Vector3& vec) = 0;

virtual void printParticle(const Particle& par) = 0;

virtual void printSystemState(const SystemState& state) =
0;
```

1. Стандартные реализации

Все реализации лежат в директории implem
В директории они лежат в 3 папках

Read

Для *Vread* есть 3 стандартные реализации

StdCinRead

Использует стандартный поток ввода stdin. Только конструктор по умолчанию. Все числа вводятся с разделителями: пробел или перенос строки.

- Вектор считывает, как 3 числа с плавающей.
- Частицу читает, как 2 вектора: скорость и координаты и 2 числа с плавающей: масса и спин.
- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

StreamRead

Использует стандартные потоки *std::istream*. Только конструктор от *std::istream*. В конструкторе происходит копирование потока. Все числа вводятся с разделителями: пробел или перенос строки.

- Вектор считывает, как 3 числа с плавающей точкой.
- Частицу читает, как 2 вектора: скорость и координаты и 2 числа с плавающей точкой: масса и спин.
- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

FileRead

Использует файловые потоки *std::ifstream*. Два конструктора

1. Конструктор от пути до файла: происходит открытие файла экземпляром класса *FileRead*.
2. Конструктор от *std::ifstream*: происходит копирование потока. Все числа вводятся с разделителями: пробел или перенос

строки.

- Вектор считывает, как 3 числа с плавающей точкой.
- Частицу читает, как 2 вектора: скорость и координаты и 2 числа с плавающей точкой: масса и спин.
- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

Update

Для класса *VUpdate* есть одна реализация - *UpdateNoCollisions*. Предоставляет стандартное обновление частиц, без учета их столкновений.
Обновление происходит по следующей схеме:

Расчет сил

В самом начале считаются силы, действующие на каждую частицу. Сила, действующая на частицу - это сумма сил, действующие между частицами, плюс внешняя сила. Силы считаются квадратным алгоритмом: во внешнем цикле перебираются частицы, для которых мы считаем силу, а во внутреннем, через которых считаем. Силы между двумя частицами записываются в вектор сил *std::vector<Vector3> particles_forces_array*. Если индекс в первом цикле равен индексу во втором, то в *particles_forces_array* остается значение по умолчанию (конструктор по умолчанию вектора 3).

Далее сумма всех сил записываются в *std::vector<Vector3> forces* посредством линейного прохода для дальнейшего использования.

```
for (int i = 0; i < state.size(); ++i) {
  for (int j = 0; j < state.size(); ++j) {
    if (i == j) continue;

    particles_forces_array[j] = f_btw_two_par(state[i],
state[j]);
  }

  forces[i] = external_f(state[i]);
  for (const auto& force : particles_forces_array) {
    forces[i] += force;
  }
}
```

Обновление системы

Сначала создается новое, изначально равное старому. Это сделано для того, чтобы все вектора сразу приняли нужный размер. `

```
SystemState new_state = state;
```

Далее линейным проходом пересчитываются координаты и скорость

```
for (int i = 0; i < state.size(); ++i) {
  ...
}
```

Координаты

Координаты считаются через старую, еще не обновленную скорость. Пересчет идет по формуле

$x = coordinates.x + velocity.x * dt$, где

- x - одна из 3 координат, пересчет ведется по всем
- $coordinates$ - координаты частицы
- $velocity$ - скорость частицы
- dt - промежуток времени

```
new_state[i].setCoordinates({state[i].getCoordinates().get
P1() + state[i].getVelocity().getP1()*d_time,

state[i].getCoordinates().getP2() +
state[i].getVelocity().getP2()*d_time,

state[i].getCoordinates().getP3() +
state[i].getVelocity().getP3()*d_time});
```

Скорость

Скорость пересчитывается через релятивистский закон Ньютона в динамике.

$$\frac{m_0 \mathbf{a}}{\sqrt{1 - v^2/c^2}} = \mathbf{F} - \frac{1}{c^2}\mathbf{v}(\mathbf{F}\mathbf{v}).$$

Здесь введены следующие обозначения:

- v - скорость частицы
- c - скорость света
- F - сила, действующая на частицу
- m_0 - масса частицы
- a - ускорение
  Ускорение задается формулой
  $(v_{k+1} - v_k)/dt$

где dt - рассматриваемый промежуток времени, k - индексы скоростей

В фреймворке скорость заменена на β по формуле
$\gamma = v/c$
Замена была сделана из соображений удобства

И замена на γ по формуле
$\gamma = \sqrt{1 - \beta_k^2}$
Замена была произведена именно по этой формуле, потому что это Лоренц-фактор

Из этой формулы я вывел следующие для вычисления β_{k+1}

$(c * dt * (F - \beta_k * (F; \beta_k)) * \gamma)/m_0 + \beta_k$

```
Vector3 numerator = d_time *
                    kLightSpeed *
                    (forces[i] - state[i].getVelocity() *
(forces[i]*state[i].getVelocity()));
double gamma = sqrt(1 -
state[i].getVelocity()*state[i].getVelocity());

new_state[i].setVelocity(numerator * gamma /
state[i].getMass() + state[i].getVelocity());
```

Как видно, вычисления разбиты на 3 части:

- numerator - числитель дроби без гамма фактора
- gamma- гамма фактор
- В конце просто остаточные вычисления

После обновления координат в новой системе обновляется время и она возвращается как результат функции.

```
new_state.setTime(new_state.getTime() + d_time);

return new_state;
```

Write

Для VWrite есть 3 стандартные реализации

StdCoutWrite

Использует стандартный поток вывода stdout. Только конструктор по умолчанию.

- Выводит вектор, как 3 числа числа с плавающей точкой, раздлленными пробелами
- Выводит частицу, как 2 вектора: скорость и координаты и два числа с плавающей точкой: масса и спин.
- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

StreamWrite

Использует стандартные потоки *std::ostream*. Только конструктор от *std::ostream*. В конструкторе происходит копирование потока.

- Выводит вектор, как 3 числа числа с плавающей точкой, раздлленными пробелами
- Выводит частицу, как 2 вектора: скорость и координаты и два числа с плавающей точкой: масса и спин.

- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

FileWrite

Использует файловые потоки *std::ofstream*. Два конструктора

1. Конструктор от пути до файла: происходит открытие файла экземпляром класса *FileWrite*.
2. Конструктор от *std::ofstream*: происходит копирование потока.

- Выводит вектор, как 3 числа числа с плавающей точкой, раздлленными пробелами
- Выводит частицу, как 2 вектора: скорость и координаты и два числа с плавающей точкой: масса и спин.
- Вводит состояние системы, как текущее время: одно число с плавающей точкой, а после этого *num_of_particles* частиц.

2. Свои реализации

Для любого из интересных классов есть возможность написать свои реализации. Для этого достаточно унаследовать от них и переопределить все функции.

Классы контейнеры

Vector3 - трехмерный вектор

Vector3 представляет собой трехмерный вектор.

В классе описаны 3 поля:

- Параметр 1 (*p1_*). Тип: *double*

- Параметр 2 (*p2_*). Тип: *double*
- Параметр 3 (*p3_*). Тип: *double*

В классе определены следующие конструкторы:

- Конструктор по умолчанию: все будет заполнено нулями
- Конструктор от 3 переменных типа *double*: копирование этих переменных в параметры вектора
- Копирующий конструктор: поэлементное копирование параметров

Для этого класса перегружены следующие операторы:
$+$, $+=$ -- сложение векторов
$-$, $-=$ -- вычитание векторов
$*$ -- скалярное произведение векторов, умножение вектора на double
$/$ -- деление вектора на число
$==$ - сравнение двух векторов на равенство
$!=$ - сравнение двух векторов на неравенство

Так же определены методы get и set для всех параметров вектора

Particle - частица

Класс *Particle* описывает состояние одной частицы

В классе описаны 4 поля:

- Скорость частицы (*velocity_*). Тип *Vector3*
- Координаты частицы (*coordinates_*). Тип *Vector3*
- Масса частицы (*mass_*). Тип *double*
- Изоспин частицы (*I_*), задает тип частицы. Тип *float*
  Определены следующие конструкторы:

- Конструктор по умолчанию: *mass_* и *I_* нули, *velocity_* и *coordinates_* - конструкторы по умолчанию
- Конструктор от 4 переменных: *Vector3 velocity*, *Vector3 coordinates*, *double mass*, *float I*. Копирование *velocity* -> *velocity_*, *coordinates* -> *coordinates_*, *mass* -> *mass_*, *I* -> *I_*
- Конструктор копирования: поэлементное копирование

В классе описаны методы get и set для *velocity_* и *coordinates_*, а так же методы get для *mass_* и *I_*.

Так же в классе Particle описана булевая функция *isNormalVelocity()*, она не принимает параметров и возвращает true, если все измерения скорости строго меньше скорости света.

```
bool isNormalVelocity() {
  return velocity_.getP1() < kLightSpeed &&
         velocity_.getP2() < kLightSpeed &&
         velocity_.getP3() < kLightSpeed;
}
```

SystemState - состояние системы

Класс *SystemState* описывает текущее состояние системы

Класс хранит 2 параметра:

- Текущее время (*time_*). Тип *double*
- Вектор частиц (*particles_*). Тип *std::vector<Particle>*

Определены следующие конструкторы:

- Конструктор по умолчанию: *time_* равно нулю, *particles_* размера 0;

- Конструктор от 2 переменных: *double time*, *std::vector<Particle> particles*. Копирование *time -> time_*, *particles -> particles_*
- Конструктор копирования: поэлементное копирование

Для класса перегружены константный и неконстантный операторы квадратные скобки для индексирования по вектору *particles_*. При выходе за границы массива программа упадет с сообщением "Array of particles in SystemState.h out of range"

```
const Particle& operator[](int idx) const {
  if (idx >= particles_.size() || idx < 0) {
    throw std::runtime_error("Array of particles in
SystemState.h out of range");
  }
  return particles_[idx];
}
```

Для всех полей определены геттеры и сеттеры.
Для класса определена функция *size()*, которая возвращает размер вектора *particles_*

Dynamic - класс менеджер

Dynamic - это класс менеджер, контролирующий роботу всего фреймворка.
Он хранит в себе:

- Состояние системы (*state_*). Тип *SystemState*
- Внешнюю силу, действующую на частицы (*external_f_*). Тип *std::function<Vector3(Particle)>*
- Силу, действующую между двумя частицами (*f_btw_two_par_*). Тип *std::function<Vector3(Particle, Particle)>*

- Указатель на модуль чтения (*reader_*). Тип *std::unique_ptr<VRead>*
- Указатель на модуль обновления (*updater_*). Тип *std::unique_ptr<VUpdate>*
- Указатель на модуль записи (*writer_*). Тип *std::unique_ptr<VWrite>*

У него есть только один конструктор: на вход подаются:

- *std::unique_ptr<VRead> reader*
- *std::unique_ptr<VUpdate> updater*
- *std::unique_ptr<VWrite> writer*
- *std::function<Vector3(Particle)> external_f*
- *std::function<Vector3(Particle, Particle)> f_btw_two_par*
  В конструкторе проходит копирование сил и передача указателей

simulate - метод симуляции

В *Dynamic* определен метод *simulate*, предназначенный для симуляции движения частиц. Метод получает время симуляции, разбивает его на заданное количество промежутков (промежутки по времени должны быть маленькими) и проводит обновление частиц за все эти промежутки.

На вход метод принимает 4 параметра:

- *num_of_iterations* - количество итераций обновления. Тип *int*
- *time* - общее время движения частиц. Тип *double*
- *num_of_particles* - количество частиц. Тип *size_t*
- *save_states* - флаг, нужно ли сохранять промежуточные значения системы. Тип *bool*

В начале считывается состояние системы, с помощью наследника класса *VRead reader_* и записывается в *state_*.

```
state_ = reader_->readSystem(num_of_particles);
```

Далее создается вектор состояний системы размера 0. Если *save_states* равно true, то под вектор резервируется размер, равный *num_of_iterations*.

После этого проходит проверка, что в *state_* все частицы корректные, с помощью метода *isNormalVelocity()* класса Particle. Если метод *isNormalVelocity()* вернул false, то программа завершает свою работу с ошибкой "The speed of a particle is greater than the speed of light".

```
for (auto part : state_.getParticles()) {
  if (!part.isNormalVelocity()) {
        throw std::runtime_error("The speed of a particle
is greater than the speed of light");
  }
}
```

Если функция продолжила свою работу, то дальше идут итерации обновления частиц. Проходит *num_of_iterations* итераций, на каждой из которых выполняется обновление с помощью наследника *VUpdate updater_*.

```
state_ = updater_->updateSystem(state_, external_f_,
f_btw_two_par_, time / num_of_iterations);
```

После этого идет проверка на корректность скоростей частиц. После этого, если *save_states* равен true, идет запись в вектор

состояний системы.

```
if (save_states) {
  states.emplace_back(state_);
}
```

В конце, когда все итерации пройдены, в зависимости от переменной *save_states* выводятся либо все состояния системы, либо только конечное.

```
if (save_states) {
  for (int i = 0; i < num_of_particles; ++i) {
    writer_->printSystemState(states[i]);
  }
} else {
  writer_->printSystemState(state_);
}
```

Интерфейс для представления сил

Для представления сил было решено использовать *std::function*, возвращающий *Vector3* - силу. Есть два вида сил: взаимодействие двух частиц и постоянная сила. Первая описывается с помощью *std::function<Vector3(Particle, Particle)>* (на вход идут 2 частицы, на выходе *Vector3*). Вторая описывается с помощь *std::function<Vector3(Particle)>* (на вход идет одна частица, на выходе *Vector3*).

1. Стандартные реализации для сил

Стандартные реализации представляют собой функции-фабрики. На вход им подаются различные константы, а на выходе вы

получаете *std::function<Vector3(Particle, Particle)>* либо *std::function<Vector3(Particle)>*.

В данный момент реализовано 2 функции для силы между частицами (возвращают *std::function<Vector3(Particle, Particle)>*):

- сила Кулона
- сила Юкавы
  И две внешние силы (возвращают *std::function<Vector3(Particle)>*):
- постоянное электрическое поле
- постоянное магнитное поле

2. Добавление собственной силы

Для добавления своей силы достаточное создать *std::function* с нужными параметрами и передать их классу *Dynamic*

- *std::function<Vector3(Particle, Particle)>* для взаимодействия двух частиц
- *std::function<Vector3(Particle)>* для внешней силы