Курс “C++ Basic”

Домашнее задание № 4 “Физический симулятор”

An object in motion will always be headed in the wrong direction.
An object at rest will always be in the wrong place
Gerrold's Laws of Infernal Dynamics

В этом задании мы на практике познакомимся с классами и методами[1], участвуя в разработке простого прототипа физического симулятора. Команда, которой он поручен, воспользовалась одной из сильных сторон ООП – возможностью проектирования программы в виде набора слабо связанных классов – и после согласования интерфейсов несколько разработчиков одновременно занялись реализацией разных частей. Одному достался графический движок, состоящий из нескольких классов, другому – физика.

Нам требуется реализовать одну из ключевых компонент – класс шара `Ball` с тем интерфейсом, на который рассчитан код коллег, и обеспечить чтение из файла модели для симуляции. В процессе работы потренируемся читать, использовать и дорабатывать сторонний код, написанный в объектном стиле.

Описание

Симулятор реализует упрощенную физическую модель, в которой с постоянной скоростью движутся шары разного размера, отталкиваясь при столкновении друг с другом или с границей области. Физический движок уже написан, нам не требуется разбираться в его устройстве. Достаточно знать, что ему требуется объект `Ball` с таким интерфейсом:

```
class Ball {
 public:
    void setVelocity(const Velocity& velocity);// задать скорость шара
    Velocity getVelocity() const;              // получить скорость шара
    void setCenter(const Point& center);       // задать координаты центра шара
    Point getCenter() const;                   // получить центр шара
    double getRadius() const;                  // получить радиус шара
    double getMass() const;                    // получить массу шара
};
```

Для отображения шара графическим движком нужно реализовать метод `draw`:

```
class Ball {
 public:
    // ...
    void draw(Painter& painter) const;        // нарисовать шар, используя painter
};
```

---

[1] Строго говоря, в стандарте C++ нет понятия метод – есть member function (функция класса). Для простоты здесь (и часто на вебинарах) будем называть их методами

Метод `draw` принимает на вход контекст для рисования – `painter`. Для отображения шара на плоскости (вид сверху) нам будет достаточно вызвать метод `painter`'а для рисования круга.

Последнее, что требуется сделать – завершить реализацию конструктора **`World::World`**, в котором читаем модель для симуляции из файла `worldFilePath`:

```
World::World(const std::string& worldFilePath) {
    // ...
    while (stream.good()) {
        stream >> x >> y >> vx >> vy;
        stream >> red >> green >> blue;
        stream >> radius;
        stream >> std::boolalpha >> isCollidable;
        // в этом месте необходимо сконструировать объект
        // нашего класса Ball ball(...);
        // и поместить его в вектор шаров
        // вызовом balls.push_back(ball);
    }
}
```

В конструкторе **`World`** уже реализовано чтение модели, включая всю необходимую информацию о шарах (`x, y, vx, vy`…) В исходном коде даны подробные комментарии о физическом смысле этих переменных. Нам необходимо немного доработать тело конструктора **`World`** для создания из каждой записи о шаре объектов **`Ball`** и помещения созданных шаров в вектор `balls`.

Задание

Доработать исходный код из примера, реализовав приведенные выше методы класса **`Ball`** (их объявления и реализации-заглушки уже есть в файлах `Ball.hpp` и `Ball.cpp` соответственно). Для этого понадобится добавить несколько полей. При необходимости можем добавить любые другие методы (например, для создания объектов нашего класса в конструкторе **`World`** будет удобно добавить конструктор класса **`Ball`**).

Ожидается, что после доработки исходный код программы будет компилироваться и запускаться командой `./physics <путь к файлу smile.txt>`, например, `./physics /home/user/physics/data/smile.txt` (тестовая модель находится в подкаталоге `data` примера), а в ходе работы программы увидим процесс симуляции столкновений шаров, завершающийся упорядоченной структурой.

Как собрать и запустить проект

В ОС Windows и MacOS сборка проекта должна заработать “из коробки”: используемая в проекте библиотека SFML и ее зависимости загружаются и собираются автоматически – требуется лишь доступ в интернет.

В ОС Linux возможны два пути.

1. Рекомендуемый - установить пакет разработчика SFML, используя пакетный менеджер вашей ОС.
   Например, в Debian и его производных (Ubuntu) для этого требуется выполнить команду `sudo apt-get install libsfml-dev`
   В Fedora и производных (Red Hat) `sudo dnf -y install SFML-devel`
2. Альтернативный путь – обеспечить сборку SFML из исходных кодов. Этот процесс запускается автоматически, если в системе не найден пакет SFML. В Linux для успешной сборки SFML требуется установить зависимости.

Если на этом этапе возникла проблема – попробуйте самостоятельно изучить логи конфигурации и сборки и понять причину, это даст ценный опыт. В случае серьезных затруднений обратитесь к наставнику на портале в обсуждении ДЗ или в чате группы.

Советы по решению

С чего начать

Прежде всего стоит убедиться, что проект собирается в вашем окружении. Для этого откройте CMakeLists.txt в вашей IDE и выполните конфигурацию и сборку проекта. В случае проблем проверьте что установлен пакет SFML или все необходимые ему зависимости (см. Как собрать проект).

Выполняем базовую часть задания

В этой части мы будем вносить правки в файлы `Ball.hpp`, `Ball.cpp` и `World.cpp` (конструктор `World::World`). Остальные файлы можно бегло изучить, если это потребуется для лучшего понимания.
Чтобы не “сломать” чужой код, важно не менять согласованный интерфейс класса `Ball`. При необходимости можно добавить новые методы.
Методы `get/setVelocity` и `get/setCenter` – типичные геттеры/сеттеры для контролируемого доступа к состоянию объекта, которые должны обращаться к приватным полям класса. Для получения радиуса и массы сеттер не предусмотрен, эти параметры шара являются константными и не меняются в процессе существования объекта.
Обратите внимание на комментарии в `Ball.cpp` и `World.cpp` – они могут помочь при выполнении задания.

Как ориентироваться в коде проекта

В этом задании, как часто и в рабочих проектах, требуется доработать часть кодовой базы, с которой мы целиком не знакомы. Важно понять, какие модули предстоит изменить и сфокусироваться на их изучении. В нашем случае это классы `Ball` и

частично `World`. В интерфейсе `Ball` используются простые примитивы `Velocity` и `Point`, которые объявлены и реализованы в соответствующих файлах.

Существенная часть реализации проекта помещена в каталог `black_box` – изучение этих исходных файлов не требуется для выполнения задания. В частности, там находится реализация графического интерфейса. Для отрисовки шаров достаточно познакомится с публичным интерфейсом (`public` методами) класса `Painter` и воспользоваться методом рисования круга `Painter::draw(const Point& center, double radius, const Color& color)`.

Дополнительные задания

Дополнительные задания не обязательны для сдачи работы, но их выполнение – хороший способ узнать что-то новое.

1. Рефакторинг

Сложность 1/5

Реализация загрузки модели из файла (см. конструктор класса `World`, пометку *`//TODO: хорошее место для улучшения`*) выглядит довольно неказисто – сейчас, например, читаем в специально объявленные исключительно для этого переменные типа `double`: `x, y` и `red, green, blue` вместо того, чтобы сразу получать объекты `Point` и `Color` соответственно. Было бы здорово упростить этот код, выделив отдельные функции чтения объектов из `std::istream`. Для этого удобно воспользоваться перегрузкой оператора ввода из потока:

```
std::istream& operator>>(std::istream& stream, <ваш класс>& variable) {...}
```

2. Призрачные шары

Сложность 2/5

Неожиданно “прилетело” новое требование – в некоторых случаях требуется отключить обработку коллизий определенных шаров. При чтении модели из файла в теле конструктора `World` помимо прочих параметров каждого шара мы получаем флаг `isCollidable`. Нас попросили доработать программу так, чтобы шары, для которых `isCollidable==false`, не сталкивались с другими объектами, а проходили сквозь них. Небольшая сложность в том, что для этого понадобится дополнить реализацию не только класса `Ball`, но и код в `Physics.cpp`, т.к. обработка `isCollidable` еще не реализована на стороне физического движка, а его разработчик в отпуске на пляже.

Для самопроверки запустите `./physics <путь к файлу elephant.txt>`, например, `./physics /home/user/physics/data/elephant.txt` – ожидается, что по завершению симуляции вновь увидим некоторую упорядоченную структуру.

3. Наводим красоту

Сложность 5/5

Почему бы не сделать столкновения шаров более зрелищными? Графический движок примитивен, но даже с его ограниченными возможностями мы можем сымитировать разлетающиеся от места столкновения маленькие круглые частицы, исчезающие через небольшое время после удара. Это исключительно визуальный эффект, поэтому чтобы

не “ломать” физическую часть симулятора временные объекты лучше не представлять в виде объектов **`Ball`** в векторе `balls`. Хорошим решением будет создать новый класс, например, **`Dust`**, который будет хранить координаты, скорости и оставшееся время отображения частиц, образованных в результате столкновения. Координаты частиц можно обновлять каждый тик симуляции по аналогии с обновлением положения шаров, но для простоты игнорируя коллизии, т.е. просто прибавляя вектор скорости. Все **`Dust`**’ы, опять же по аналогии с шарами, можно хранить в новом `std::vector` в классе **`World`**. Желательно удалять объекты, отображение которых завершено.