

Семинар #9: Подключение библиотек. Создание окна с помощью библиотеки SFML

Часть 1: Многофайловые программы. Библиотеки

Этапы сборки проекта на языке C++:

1. **Препроцессинг.** Обрабатываются директивы компилятора `#include`, `#define` и другие. Удаляются комментарии. Чтобы исполнить только этот шаг, нужно передать компилятору опцию `-E`:

```
g++ -E main.cpp > preprocessed.cpp
```

2. **Компиляция:** каждый файл исходного кода (файл расширения `.cpp`) транслируется в код на языке ассемблера. Чтобы исполнить только этапы препроцессинга и компиляции, нужно передать компилятору опцию `-S`:

```
g++ -S main.cpp
```

3. **Ассемблирование:** каждый файл на языке ассемблера транслируется в машинный код. В результате создаётся объектный файл с расширением `.o`. Чтобы исполнить процесс до этой стадии включительно нужно передать компилятору опцию `-c`:

```
g++ -c main.cpp
```

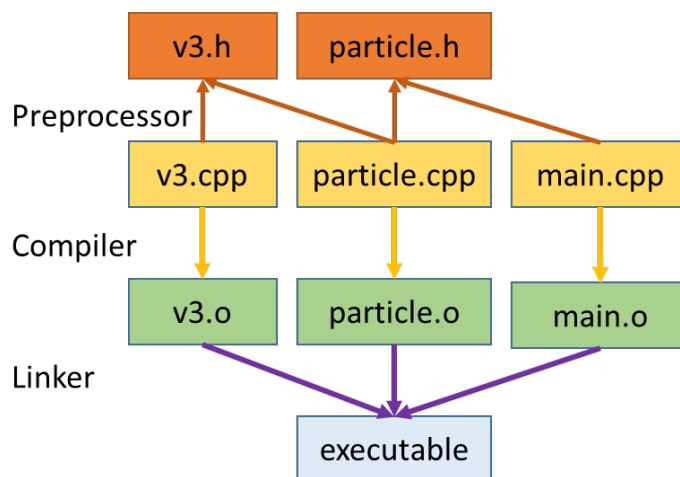
4. **Линковка:** Все объектные файлы сливаются друг с другом, а также с другими библиотеками. Даже если ваш проект состоит из одного файла, вы наверняка используете как минимум стандартную библиотеку и на этом этапе ваш код соединяется с другими библиотеками.

```
g++ main.o
```

Задание:

- В папке `0stages` лежит исходный код простой программы. Пройдите поэтапно все стадии сборки с этой программой.

Сборка многофайловой программы:



Можно собрать всё сразу:

```
g++ main.cpp particle.cpp v3.cpp
```

Либо можно собрать по частям:

```
g++ -c main.cpp
g++ -c particle.cpp
g++ -c v3.cpp
g++ main.o particle.o v3.o
```

Виды библиотек:

1. header-only библиотеки

Весь исходный код хранится в `.h` или `.hpp` файле и подключается с помощью директивы `#include` (очень просто подключить).

2. Исходный код

Библиотека поставляется в виде исходного кода (все `.h` и `.cpp` файлы). Для того чтобы использовать эту библиотеку, её нужно сначала скомпилировать, что может быть очень непросто для больших библиотек, так как процесс сборки может сильно отличаться на разных операционных системах и компиляторах.

3. Статическая библиотека

Библиотека поставляется в виде header-файлов (`.h`) и предварительно скомпилированных файлов библиотеки. Расширение статических библиотек на linux: `.a` (archive). Расширение на windows: `.lib` (library). Эти библиотеки подключаются на этапе линковки. После линковки содержимое этих библиотек содержится в исполняемом файле. Такие библиотеки проще подключить к проекту, чем исходный код. Однако, вам обязательно иметь версию библиотеки, скомпилированную на такой же ОС и на таком же компиляторе, иначе она не подключится. Обратите внимание, что статические библиотеки обязательно должны иметь префикс `lib`. Например, если мы хотим получить библиотеку под названием `image`, то файл должен называться `libimage.a`.

4. Динамическая библиотека

Библиотека поставляется в виде header-файлов (`.h`) и предварительно скомпилированных файлов библиотеки. Расширение динамических библиотек на linux: `.so` (от shared object). Расширение на windows: `.dll` (от dynamic link library). Эти библиотеки подключаются на этапе *выполнения программы*. Благодаря тому, что динамическая библиотека подключается на этапе выполнения, если несколько программ будут использовать одну и ту же библиотеку, то она будет загружаться в память лишь один раз.

Задания:

- **Класс изображения**

В папке `1image/0simple_image_class` лежит исходный код программы, которая использует класс `Image`. Это простой класс для работы с изображениями в формате `.ppm`. Скомпилируйте и запустите эту программу.

- **Случайные отрезки:**

Используйте этот класс, чтобы создать изображение, состоящее из 100 случайных отрезков случайного цвета. Для случайных чисел используйте функцию `rand()` из библиотеки `<cstdlib>`.

- **Шум:**

Добавьте в этот класс метод `void addNoise(float probability)`, который будет добавлять шум на картинку: каждый пиксель с вероятностью `probability` должен поменять цвет на случайный. Протестируйте этот метод на картинках из папки `~/data/ppm_images`.

- **Заголовочный файл**

Создайте файл `image.hpp` и перенесите класс `Image` в этот файл. Используйте директиву `#pragma once`, чтобы избежать в будущем возможного множественного включения. Таким образом получится мини-библиотека. Подключите этот заголовочный файл к вашей основной программе и скомпилируйте программу.

- **Раздельная компиляция:**

Создайте ещё один файл `image.cpp`. Перенесите из файла `image.hpp` в файл `image.cpp` все определения методов. Таким образом в файле `image.hpp` должны остаться определение класса и объявление всех его методов, а в файле `image.cpp` должны быть определения всех методов. Скомпилируйте эту программу.

- **Статическая библиотека**

Чтобы создать свою статическую библиотеку вам нужно:

1. Создать объектный файл необходимого исходного файла.
2. Превратить объектный файл (или файлы) в библиотеку, используя утилиту `ar`:

```
ar rvs libimage.a image.o
```
3. После этого файл `libimage.a` можно будет подключить к любому другому проекту примерно так:

```
g++ main.cpp -I<путь до header-файлов> -L<путь до libimage.a> -limage
```

Создайте статическую библиотеку из файла `image.cpp`. Создайте папку `image/` в которой будет храниться наша библиотека. В этой папке создайте ещё 2 папки: `include` и `lib/`. Поместите в папку `image/include` заголовочный файл `image.hpp`. Поместите в папку `image/lib` файл статической библиотеки `libimage.a`. Затем вам нужно удалить файл `image.cpp` и собрать программу используя только статическую библиотеку (не забывайте про опции `-I`, `-L` и `-l`).

- **Динамическая библиотека**

Чтобы создать динамическую библиотеку из файла исходного кода (`image.cpp`):

```
g++ -c -fPIC image.cpp -o image.o  
g++ -shared -o libimage.so image.o
```

Чтобы скомпилировать код с подключением динамической библиотеки:

```
g++ -o main.exe main.cpp libimage.so
```

или

```
g++ -o main.exe main.cpp -limage
```

Но для этого понадобится добавить в переменную среды `LD_LIBRARY_PATH` (на Windows нужно добавить в переменную среды `PATH`) путь до папки, содержащий библиотеку.

1. Создайте динамическую библиотеку и скомпилируйте саму программу с подключением динамической библиотеки
2. Проверьте чему равен размеры исполняемых файлов в случае подключения статической и динамической библиотеки.
3. Что будет происходить, если перенести файл динамической библиотеки в другую папку. Запустится ли исполняемый файл?

Часть 2: Библиотека SFML:

Библиотека SFML (Simple and Fast Multimedia Library) - простая и быстрая библиотека для работы с мультимедиа. Кроссплатформенная (т. е. одна программа будет работать на операционных системах Linux, Windows и MacOS). Позволяет создавать окно, рисовать в 2D и 3D, проигрывать музыку и передавать информацию по сети. Для подключения библиотеки вам нужно скачать нужную версию с сайта: sfml-dev.org.

Подключение вручную:

Для подключения библиотеки вручную через опции g++ нужно задать путь до папок include/ и lib/ и названия файлов библиотеки, используя опции -I, -L или -l.

```
g++ .\main.cpp -I<путь до include> -L<путь до lib> -lsfml-graphics -lsfml-window -lsfml-system
```

Например так:

```
g++ .\main.cpp -I./SFL-2.5.1/include -L./SFL-2.5.1/lib -lsfml-graphics -lsfml-window -lsfml-system
```

bash-скрипт:

Так как постоянно прописывать в терминале сборку проекта может быть затруднительно, то можно положить весь процесс сборки в специальный bash-скрипт. bash-скрипт - это просто файл кода языка терминала linux. (Для windows есть аналогичные bat-скрипты) Пример можно посмотреть в `2sfml/1bash_script`.

Makefile:

make – это специальная утилита, предназначенная для упрощения сборки проекта. В `2sfml/3makefile` содержится пример проекта с make-файлом. Содержимое make-файла представляет собой просто набор целей и соответствующих команд оболочки bash. Откройте make-файл и просмотрите его содержимое. Чтобы скомпилировать его просто:

```
make <имя цели>
```

либо просто

```
make
```

(в этом случае make запустит процесс создания первой цели)

Задания:

- **Сборка:** скомпилируйте и запустите проект. Используйте bash-скрипт или make-файл. Более подробно – в папке `2sfml`.
- **Движение по окружности:** Заставьте кружок двигаться по окружности.
- **Броуновское движение:** Создайте $n = 50$ кругов, которые будут двигаться случайным образом (направление и величина движения должны задаваться случайным образом на каждом кадре).
- **Задача n тел:** Создайте $n = 50$ кругов, так, чтобы они притягивались друг к другу гравитационной силой. Начальные положения и скорости задайте случайным образом. Сила гравитации в двух измерениях обратно пропорциональна расстоянию между объектами.

$$F \sim \frac{1}{R}$$

- **Задача N тел с массой**
Добавьте разную массу шарикам. При создании шарика масса должна задаваться случайным образом. Масса шарика должна быть пропорциональна площади (квадрату радиуса).
- **Электрические заряды**
Смоделируйте взаимодействие заряженных частиц. Для этого нужно добавить поле в структуру `Ball`, которое будет определять величину заряда. Эта величина может быть как положительной, так и отрицательной. В начале работы программы заряд должен задаваться случайно. Заряды должны взаимодействовать по закону Кулона. Гравитацией можно пренебречь. Цвета зарядов должны быть различными (красный для положительного заряда и синий для отрицательного, интенсивность цвета - пропорциональна величине заряда).

- **Нажатие мыши**

События нажатия мыши можно обработать с помощью следующего синтаксиса:

```
if (event.type == sf::Event::MouseButtonPressed)
{
    if (event.mouseButton.button == sf::Mouse::Right)
    {
        std::cout << "the right button was pressed" << std::endl;
        std::cout << "mouse x: " << event.mouseButton.x << std::endl;
        std::cout << "mouse y: " << event.mouseButton.y << std::endl;
    }
}
```

Внутри цикла `while (window.pollEvent(event))`.

Видоизмените вашу программу так, чтобы при нажатии левой кнопки мыши в том месте, где находится мышь, создавался бы шарик со средними массой и средним положительным зарядом зарядом. При нажатии правой кнопки мыши должен создаваться шарик с очень большой массой и очень большим положительным зарядом. При аналогичных нажатиях, но с зажатой клавишей Shift, должны создаваться отрицательные заряды.

Часть 1: Основные типы и классы библиотеки SFML

Типы целых чисел

Так как библиотека SFML кроссплатформенная, то в ней введены `typedef`-синонимы для целочисленных типов, например `Int8`, `Int64`, `Uint32` и другие. Эти типы гарантируют, что они будут соответствующего размера.

Классы математических векторов

Классы двумерных математических векторов `sf::Vector2<T>`. У них есть два публичных поля: `x` и `y`. Также, для них перегружены операции сложения с такими же векторами и умножения на числа. Также введены `typedef`-синонимы вроде `sf::Vector2f` для `sf::Vector2<float>` и другие.

Класс цвета

Класс цвета `sf::Color`. Имеет 4 публичных поля: `r`, `g`, `b`, `a` - компоненты цвета в цветовой модели RGB и прозрачность. Есть конструктор от 3-х или 4-х аргументов. Есть перегруженные операции для сравнения и сложения цветов. Есть уже определённые цвета вроде `sf::Color::Blue` и другие.

Класс строки

В SFML есть свой класс строки под названием `sf::String`. Поддерживает разные виды кодировок. Имеет конструкторы от стандартных строк C++ и строк в стиле C.

Класс окна

Прежде чем начать рисовать, нужно создать окно, которое будет отображать то, что мы нарисовали. Для этого в SFML есть класс `sf::RenderWindow`. Вот его основные методы:

- `RenderWindow(sf::VideoMode m, const sf::String& title, sf::Uint32 style, sf::ContextSettings& s)`
Конструктор, с двумя обязательными и двумя необязательными аргументами. Его аргументы:

- Видеорежим - определяет размер окна.
- Заголовок окна
- Стилль окна, необязательный аргумент, может принимать следующие значения:

- `sf::Style::None`
- `sf::Style::Titlebar` – окно с заголовком
- `sf::Style::Resize` – окно у которого можно менять размер
- `sf::Style::Close` – окно с кнопкой закрывания
- `sf::Style::Fullscreen` – полноэкранный режим
- `sf::Style::Default = sf::Style::Titlebar | sf::Style::Resize | sf::Style::Close`

Этот параметр имеет значение по умолчанию (`sf::Style::Default`).

- Дополнительные настройки контекста OpenGL, необязательный аргумент.

- `getPosition` и `setPosition` - получить или установить положение окна.
- `getSize` и `setSize` - получить или установить размер окна в пикселях.
- `setFramerateLimit` – установить лимит для количества кадров в секунду.
- `clear` - принимает цвет и очищает скрытый холст этим цветом
- `draw` - рисует объект на скрытый холст
- `display` - отображает на экран всё что было нарисовано на скрытом холсте

Классы фигур

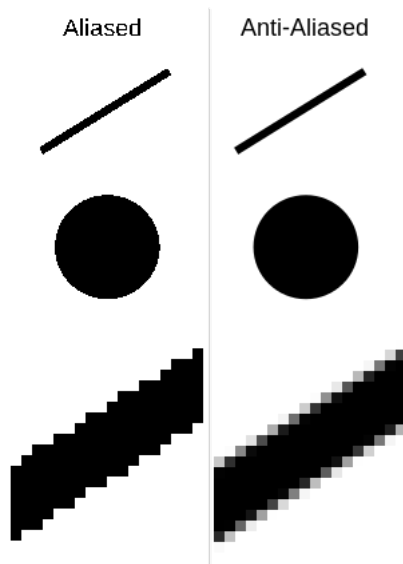
В SFML есть несколько классов для работы с простыми фигурами: `sf::CircleShape` (круг или эллипс), `sf::RectangleShape` (прямоугольник), `sf::ConvexShape` (фигура сложной формы, задаваемая точками). У этих классов есть общие методы:

- `setOrigin` - установить локальное начало координат фигуры. Положение этой точки задаётся относительно верхнего левого угла прямоугольника, ограничивающего фигуру. По умолчанию эта точка равна $(0, 0)$, то есть локальным началом координат фигуры считается её верхний левый угол. Эта точка важна, так как относительно неё происходят все операции поворота и масштабирования.
- `setPosition`, `getPosition` - задать и получить координаты фигуры. Фигура перемещается таким образом, чтобы её `origin` оказался в заданной точке.
- `move` - принимает 2D вектор и передвигает фигуру на этот вектор.
- `setRotation`, `getRotation` - задать и получить угол (в градусах) вращения фигуры вокруг точки `origin`
- `rotate` - принимает вещественное число и вращает фигуру на этот угол (в градусах)
- `setScale`, `getScale` - задать и получить величину масштабирования (2D вектор)
- `scale` - принимает 2D вектор и растягивает или сжимает фигуру по x и по y соответственно
- `setFillColor`, `getFillColor` – устанавливает/возвращает цвет заливки фигуры

Задачи:

- В папке `00circle` приведён пример программы, которая рисует круг. Измените эту программу, чтобы круг рисовался в левом верхнем углу экрана.
- Измените эту программу, чтобы круг рисовался в правом нижнем углу экрана.
- Нарисуйте 100 кругов в случайных местах экрана, случайного размера и случайного цвета.
- В папке `01shapes` приведён пример программы, которая рисует несколько фигур. Используйте методы выше и сделайте так, чтобы круг двигался по прямой, прямоугольник вращался, а фигура сложной формы сжималась по y и растягивалась по x. Подберите скорости этих операций, чтобы они были не слишком быстрыми.
- Перейдите в полноэкранный режим

Anti-Aliasing



Вы могли заметить, что фигуры выглядят не очень красиво - имеют зазубрены. Это связано с тем, что рисования происходит на прямоугольной сетке пикселей и при проведении линий под углом образуются ступеньки. Для борьбы с этим эффектом был придуман специальный метод сглаживания, который называется антиалиасинг. Он уже автоматически реализован во всех библиотеках компьютерной графики. Чтобы установить его в SFML, нужно прописать опцию:

```
sf::ContextSettings settings;  
settings.antiAliasingLevel = 8;
```

И передать `settings` на вход для конструктора `RenderWindow`. Пример в папке `02antialiasing`.

Главный цикл

Как правило, у любой программы, работающей на основе событийно-ориентированной модели, есть главный цикл. На каждой итерации данного цикла программа должна проделать все необходимые операции по подготовке и отрисовке следующего кадра. Число итераций этого цикла называется числом кадров в секунду (англ. frames per seconds - fps).

В папке `4mainloop` представлена простейшая программа с главным циклом. Сейчас основной цикл программы работает без перерывов и, так как наша программа очень проста, то количество кадров в секунду может достигать огромных значений - больше 1000 fps. Мониторы не обновляют экран с такой скоростью и человеческий глаз тоже не способен воспринять такую частоту кадров. Поэтому не имеет смысла задавать fps очень высоким, его желательно ограничить. Это можно сделать с помощью метода `setFramerateLimit`. Пример в папке `06framerate_limit`.

Этот метод ограничивает лишь максимальное количество кадров. Если за один кадр выполняется много вычислений, то fps может просесть ниже 60. Из-за этого программы, которые завязаны на времени, могут работать некорректно. Например, в нашем примере скорость движения шарика зависит от числа кадров в секунду. Чтобы шарик двигался одинаково независимо от fps нужно высчитывать время, занятое на каждом кадре. Пример, как это делать в папке `07clock_time`.

Класс времени

Класс `sf::Time` для работы со временем. Есть методы `asSeconds`, `asMilliseconds` и `asMicroseconds`, которые возвращают время в виде числа в соответствующих единицах. Перегружены операторы сложения, умножения и другие. Есть дружественные функции `sf::seconds`, `sf::milliseconds` и `sf::microseconds`, которые принимают число, и возвращают соответствующие объект класса `sf::Time`.

Класс часов

`sf::Clock` – это маленький класс для измерения времени. У него есть:

- Конструктор по умолчанию, часы запускаются автоматически после создания.
- Метод `getElapsedTime()` – возвращает объект `sf::Time` – время прошедшее с последнего запуска часов.
- Метод `restart()` – заново запускает часы.

Проверка на нажатие клавиш и кнопок

Класс клавиатуры

Класс клавиатуры `sf::Keyboard`. Внутри этого класса, в публичной части, объявлен перечисляемый тип `Key`, в котором перечислены все клавиши. Например, чтобы проверить нажатие на пробел понадобится `sf::Keyboard::Space`. Название всех клавиш можно найти по следующей ссылке: [Тут](#).

У этого класса есть метод

- `isKeyPressed` – принимает клавишу и проверяет нажата ли она.

Пример – в папке `08is_key_pressed`.

Класс мыши

Класс мыши `sf::Mouse`. Внутри этого класса, в публичной части, объявлен перечисляемый тип `Button` в котором перечислены все кнопки мыши. У этого класса есть метод:

- `isButtonPressed` принимает на вход `sf::Mouse::Button` и проверяет нажата ли соответствующая кнопка.
- `getPosition()` – возвращает положение мыши на в координатах всего экрана.
- `setPosition(const sf::Vector2i&)` – устанавливает положение мыши на в координатах всего экрана
- `getPosition(const sf::Window&)` – возвращает положение мыши на в координатах данного окна.
- `setPosition(const sf::Vector2i&, const sf::Window&)` – устанавливает положение мыши на в координатах данного окна.

Пример – в папке `09is_button_pressed`.

Задачи:

- Создайте 2 объекта: круг и квадрат. Круг должен двигаться при нажатии на стрелки. Квадрат должен двигаться при нажатии на WASD.
- Сделайте так, чтобы при нажатии на левую кнопку мыши координаты круга становились бы равными координатам мыши.
- Сделайте так, чтобы при нажатии на **Enter** цвет квадрата менялся случайным образом каждый кадр.
- Сделайте так, чтобы квадрат передвигался вправо на 50 пикселей каждые 2 секунды. При этом, все остальное должно работать как прежде, то есть функцию `sf::sleep` использовать не получится.
- Сделайте так, чтобы цвет круга плавно зависел от положения курсора на экране.
- Создайте новый круг белого цвета и сделайте так, чтобы при наведении на него курсора, он становился красным.

Работа с текстом

Для работы с текстом есть два класса. Класс шрифта `sf::Font` и класс текста `sf::Text`. Пример работы с текстом в папке `03text`.

Задачи:

- Создайте вращающийся текст.
- Сделайте так, чтобы при нажатии клавиши пробел у текста задавалась случайная позиция, случайный поворот, случайный цвет и случайное масштабирование(в разумных пределах).
- Создайте 2 поля текста. В первом нужно печатать положение мыши в системе отсчёта всего экрана. Во втором поле текста нужно печатать координаты мыши в системе отсчёта окна. Для перевода чисел в строку используйте функцию `std::to_string`.

Часть 2: События

- **KeyPressed:** В папке `1key_events` лежит пример программы, которая обрабатывает нажатия клавиш. Измените программу так, чтобы при нажатии на клавишу Enter кружок менял цвет на случайный.
- **KeyReleased:** Измените программу так, чтобы при *отпускании* клавиши пробел прямоугольник менял цвет на случайный (событие `sf::Event::KeyReleased`).
- **MouseButtonPressed:** В папке `2mouse_events` лежит пример программы, которая обрабатывает нажатия и движение мыши. Измените программу так, чтобы при нажатии на правую кнопку мыши, прямоугольник перемещался к положению мыши. Событие должно срабатывать только в момент нажатия, прямоугольник не должен двигаться при зажатии кнопки.

```
if (event.type == sf::Event::MouseButtonPressed)
{
    if (event.mouseButton.button == sf::Mouse::Right)
    {
        std::cout << "the right button was pressed" << std::endl;
        std::cout << "mouse x: " << event.mouseButton.x << std::endl;
        std::cout << "mouse y: " << event.mouseButton.y << std::endl;
    }
}
```

- **MouseMoved:** Событие, которое срабатывает тогда, когда двигается мышь.

```
if (event.type == sf::Event::MouseMoved)
{
    std::cout << "new mouse x: " << event.mouseMove.x << std::endl;
    std::cout << "new mouse y: " << event.mouseMove.y << std::endl;
}
```

Измените программу так, чтобы прямоугольник окрашивался в красный цвет тогда и только тогда, когда курсор мыши находится на прямоугольнике. Во всё остальное время прямоугольник должен быть зелёным.

- **Перетаскивание:** Создайте новый прямоугольник и сделайте его перетаскиваемым. При нажатии на него и последующим движении мыши он должен начать двигаться вместе с курсором. При отпуске мыши должен остаться на месте.

Часть 3: Задачи

- **Кнопка:** Создайте "кнопку". Кнопка представляет собой прямоугольник и некий текст поверх этой кнопки. Логика работы должна быть аналогичной логике работы обычной кнопки в ОС Windows. При наведении на прямоугольник он должен немного менять цвет. При зажатии кнопки мыши на прямоугольнике он должен менять цвет на третий. Кнопка должна быть зажата пока зажата кнопка мыши, даже если курсор уже вышел за пределы кнопки. При отпускании мыши, если курсор всё ещё находится на прямоугольнике, должно срабатывать некоторое действие. В качестве действия – пусть круг будет менять цвет на случайный.
 - Создайте свой класс `Button`, который будет описывать данное поведение.
 - Используйте этот класс и создайте 4 кнопки с надписями `Left`, `Right`, `Down`, `Up`. При нажатии на эти кнопки круг должен перемещаться на 20 пикселей в соответствующем направлении.
- **Шарики:** В папке `collision_circles` содержится заготовка кода.
 - Используйте этот код, чтобы найти пересечение двух шаров. Если в процессе движения шары начнут накладываться друг на друга, то они должны окрашиваться в красный цвет. После прекращения наложения, шары должны опять стать белыми. Для этого добавьте поле типа `sf::Color` в класс `Ball` и метод `bool is_colliding(const Ball& b) const`, который будет проверять 2 кружка на столкновение.
 - Измените программу так, чтобы кружки упруго отскакивали друг от друга. Для этого нужно, при столкновении шариков, обратить составляющую скорости параллельную прямой, соединяющую центры шариков.
 - Добавьте возможность добавления нового шарика по нажатию правой кнопки мыши.
 - Добавьте возможность стенки по нажатию левой кнопки мыши. Нужно нажать ЛКМ в одной точки и отпустить в другой, чтобы получить стенку. Стенка – это просто отрезок. Но шарики должны от него должны отскакивать. Про обнаружение столкновений можно посмотреть в папке `collision_examples`.
- **Pong:** Создайте игру Pong на 2 игрока. Первый игрок должен управлять ракеткой используя клавиши `W` и `S`. Второй игрок – стрелочки вниз и вверх.

