

Семинар #1: Библиотека SFML.

Часть 1: Подключение библиотеки 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.

Подключение библиотеки на Windows с использованием пакетного менеджера MSYS2

Самый простой способ установки библиотеки на компьютер – это использованием пакетного менеджера. В данном руководстве будет рассматриваться установка библиотеки с помощью пакетного менеджера `pacman` среды MSYS2. Для того, чтобы установить SFML в среде MSYS2 сделайте следующее:

1. Найдите как называется пакет SFML в среде MSYS2. Для этого просто загляните `msys2 install sfml` и одной из первых ссылок должен быть страница библиотеки SFML сайта `packages.msys2.org`. Зайдите на эту страницу и найдите команду для установки SFML. Скопируйте эту команду. Это может быть команда:

```
pacman -S mingw-w64-x86_64-sfml
```

или просто

```
pacman -S sfml
```

2. Откройте терминал MSYS2 для установки пакетов. Если у вас установлен MSYS2, то это можно сделать, нажав Пуск и начав печатать "MSYS2". Вставьте команду для установки SFML в терминал и нажмите Enter. Возможно потребуется нажать клавишу Y и Enter, чтобы подтвердить установку. После этого библиотека установится на компьютер.
3. Убедитесь, что библиотека установилась. Для этого перейдите в папку, где установлен MSYS2, по умолчанию это `C:\msys64`. После этого найдите папку в которой установилась библиотека SFML. Если вы используете 64-х битную версию компилятора MinGW, то библиотека установится в папке `C:\msys64\mingw64`. В папке `C:\msys64\mingw64\bin` должны лежать `.dll` файлы библиотеки SFML. А в папке `C:\msys64\mingw64\include` – заголовочные файлы библиотеки.
4. Убедитесь, что путь до папки, в которой лежат `.dll` файлы библиотеки SFML прописан в переменной среды `PATH`. Если этого пути в переменной `PATH` нет, то добавьте его.
5. Если терминал был открыт, то закройте его, а потом откройте заново.

Всё, библиотека установлена. Теперь можно компилировать файл исходного кода, использующий библиотеку SFML следующим образом:

```
g++ main.cpp -lsfml-graphics -lsfml-window -lsfml-system
```

Подключение библиотеки на Linux с использованием пакетного менеджера

Нужно установить SFML с помощью стандартного пакетного менеджера. Предположим, что используется пакетный менеджер `apt`:

```
sudo apt install libsfml-dev
```

Всё, библиотека установлена. Теперь можно компилировать файл исходного кода, использующий библиотеку SFML следующим образом:

```
g++ main.cpp -lsfml-graphics -lsfml-window -lsfml-system
```

Тестирование библиотеки SFML

Чтобы протестировать, что библиотека установилась корректно, создайте в любой директории файл `main.cpp` и поместите туда простейшую программу, использующую SFML:

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        window.clear();
        window.draw(shape);
        window.display();
    }
}
```

Эту программу можно найти по адресу <https://www.sfm1-dev.org/tutorials/2.6/start-cb.php>.

После этого зайдите в терминал, перейдите в папку, содержащую этот файл и скомпилируйте его командой:

```
g++ main.cpp -lsfml-graphics -lsfml-window -lsfml-system
```

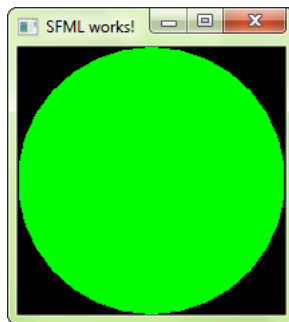
Если SFML был установлен корректно, то программа должна скомпилироваться и в папке должен создаться исполняемый файл `a.exe` (или `a.out` на Linux). Запустите этот файл командой:

```
.\a.exe
```

или, если вы работаете на Linux, то командой:

```
./a.out
```

Если SFML был установлен корректно, то программа должна завестись, создать окошко размером 200 на 200 пикселей, в котором будет нарисован зелёный круг. Если это произошло, то библиотека SFML подключилась корректно.



Подключение вручную на Windows

Если вы по какой-то причине не хотите использовать пакетные менеджеры (например, хотите установить другую версию библиотеки), то можно библиотеку подключить вручную. Для подключения библиотеки вам нужно сделать следующее:

1. Скачать нужную версию с сайта: sfml-dev.org. Зайдите на этот сайт, нажмите на Downloads, а затем на Latest stable version, и выберите версию библиотеки, соответствующую вашему компилятору. Убедитесь, что версия полностью соответствует вашему компилятору, иначе библиотека не будет работать. В нашем курсе предполагается, что вы используете компилятор GCC MinGW 64-bit, но, возможно, вы используете другой компилятор.
2. Распакуйте скачанный архив. Он будет содержать папку с названием SFML и номер версии, например SFML-2.6.1. Переместите эту папку в удобное вам место на диске. Очень важно, чтобы путь до этой папки не содержал пробелы, кириллицу и какие-либо странные символы.
3. Зайдите в папку SFML-<номер версии>. В ней должны содержаться папки bin, include, lib и другие. Зайдите в папку bin, там должны лежать .dll файлы библиотеки SFML. Запомните название этой папки.
4. Добавьте в переменную среды PATH путь до папки, содержащей .dll файлы библиотеки SFML.
5. Если терминал был открыт, то закройте его, а потом откройте заново.

Всё, библиотека установлена. Теперь можно компилировать файл исходного кода, использующий библиотеку SFML следующим образом:

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

Например, если я переместил папку SFML просто на диск C и путь до этой папки это C:\SFML-2.6.1, то нужная команда для компиляции будет:

```
g++ main.cpp -I C:\SFML-2.6.1\include -L C:\SFML-2.6.1\lib -lsfml-graphics -lsfml-window -lsfml-system
```

Это команда очень длинная, но вы можете вызвать её один раз. После этого можно нажимать клавишу вверх на клавиатуре, чтобы повторить команду. Или же можно просто где-то сохранить команду (в .txt файле на диске), а потом просто копировать её и вставлять в терминал.

Подключение вручную на Linux

Этот способ совпадает со способом Windows, за исключением того, что вам не нужно устанавливать значение переменной PATH. Компилирование также совпадает:

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

Использование bat-скрипта на Windows

Так как постоянно прописывать в терминале команду для компиляции может быть затруднительно, то можно положить весь процесс сборки в специальный bat-скрипт. bat-скрипт - это просто файл кода языка терминала Windows. Для того, чтобы использовать такой в файл в нашем случае нужно сделать следующее:

1. Создать в той папке, где лежит файл main.cpp, новый текстовый файл.
2. Откройте этот новый текстовый файл и добавьте туда следующее:

```
g++ %1 -I<путь до include> -L<путь до lib> -lsfml-graphics -lsfml-window -lsfml-system  
.\a.exe
```

где вместо <путь до include> нужно подставить путь до include папки SFML, а вместо <путь до lib> – путь до lib папки SFML. Сохраните и закройте файл.

3. Переименуйте этот текстовый файл в файл с расширением .bat. Например, в run.bat. Убедитесь, что ваша операционная система показывает расширения всех файлов и что файл действительно называется run.bat, а не, например, run.bat.txt.

4. Откройте терминал и в терминале зайдите в папку, содержащую файлы `main.cpp` и `run.bat`
5. Выполните в терминале команду

```
run main.cpp
```

или, если эта команда не сработала, то:

```
.\run.bat main.cpp
```

После этого всё содержимое файла `run.bat` исполнится (за место `%1` подставится `main.cpp`), что означает, что ваша программа скомпилируется и запустится. То есть, теперь для компиляции и запуска программы достаточно написать в терминале одну команду `run`. Если понадобится скомпилировать другую программу, то файл `run.bat` можно будет скопировать к этой программе.

Часть 2: Основные 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.

```
sf::Vector2f a {1.0, 2.0};
sf::Vector2f b {3.0, -1.0};

sf::Vector2f c = 2.0f * (a + b);
std::cout << c.x << " " << c.y << std::endl; // Напечатает 8 2
```

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.

```
sf::Color a {100, 200, 50};
sf::Color b {100, 100, 0};

sf::Color c = a + b;
std::cout << c.r << " " << c.g << " " << c.b << std::endl; // Напечатает 200 255 50
```

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`. Функция `sf::sleep(sf::Time t)` - ожидает время `t`.

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 и возвращает объект `sf::Time` - время прошедшее с предыдущего запуска часов.

```
sf::Clock clock;

sf::Time t1 = clock.restart();
sf::sleep(sf::seconds(2));
sf::Time t2 = clock.restart();

std::cout << (t2 - t1).asMilliseconds() << std::endl; // Напечатает 2000
```

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::Titlebar | sf::Resize | sf::Close`

Этот параметр имеет значение по умолчанию (`sf::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

Простая программа, которая рисует движущийся круг

```
#include <SFML/Graphics.hpp>
int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Moving Circle", sf::Style::Default);
    window.setFramerateLimit(60);

    sf::CircleShape circle(30);
    circle.setPosition(sf::Vector2f{100, 100});

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        circle.move(sf::Vector2f{1, 1});

        window.clear(sf::Color::Black);
        window.draw(circle);

        window.display();
    }
}
```

Пояснения по программе:

- В строке:

```
sf::RenderWindow window(sf::VideoMode(800, 800), "Moving Circle", sf::Style::Default);
```

создаётся объект класса окна, устанавливается разрешение окна и названия окна, а также стиль окна.

- В строке:

```
window.setFramerateLimit(60);
```

Ограничивает максимальное количество кадров в секунду (англ. *frames per second* или *fps*) числом 60. Если не прописать эту строку, то на мощных компьютерах все движения в программе будут происходить быстрее, так как за секунду будет выполняться намного больше, чем 60 итерации главного цикла. Метод `setFramerateLimit` заставляет программу ожидать после каждого цикла, чтобы общее время одной итерации главного цикла была равна как минимум 1/60 секунды.

Но нужно понимать, что этот метод ограничивает только максимальное количество fps. Если за один кадр выполняется много вычислений, то fps может просесть ниже 60. Из-за этого все движения объектов в программе будут происходить медленнее. Чтобы скорость движения объектов не зависела от мощности компьютера, нужно высчитывать время, занятое на каждом кадре, и передвигать объект в соответствии с этим временем.

- В строке:

```
sf::CircleShape circle(30);
circle.setPosition(sf::Vector2f{100, 100});
```

создаём объект круга и устанавливаем его положение в точку с координатами (100, 100). Учтите, что в SFML ось Y направлена сверху вниз. То есть значение $y = 0$ будет находиться в самом верху экрана, а значение $y = 800$ будет находиться в самом низу нашего экрана высотой 800 пикселей.

- Далее, со строки:

```
while (window.isOpen())
```

начинается *главный цикл* программы. Каждая итерация этого цикла – это один кадр прогаммы. Цикл заканчивается когда у объекта окна вызовется метод `close`.

- В строках:

```
sf::Event event;
while (window.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        window.close();
}
```

написан *цикл обработки событий*. Событиями могут быть, например, нажатие клавиш или кнопок мыши, движение мыши, изменение размера экрана, закрытие окна. За время одного кадра может произойти несколько событий. Все эти события помещаются в специальную очередь. В начале каждой итерации главного цикла нужно взять из этой очереди все события и обработать их.

В данном простом цикле обработки событий, обрабатывается только событие закрытия окна (нажатие на красный крестик). При нажатии на красный крестик, программа будет закрываться.

- В строке:

```
circle.move(sf::Vector2f{1, 1});
```

мы передвигаем кружок на один пиксель вправо по оси X и на один пиксель вниз по оси Y.

- В строке:

```
window.clear(sf::Color::Black);
```

мы закрашиваем *скрытый холст* черным цветом. Это нужно делать, чтобы закрасить то, что было нарисовано на предыдущем кадре. Скрытый холст – это просто двумерный массив из цветов пикселей размера *ширина окна* x *высота окна*, находящийся в памяти компьютера. После закрашки скрытого холста никаких изменений на экране не произойдёт, так как скрытый холст на экран не отображается. В это время на экран отображается содержимое *первичного холста*.

- В строке:

```
window.draw(circle);
```

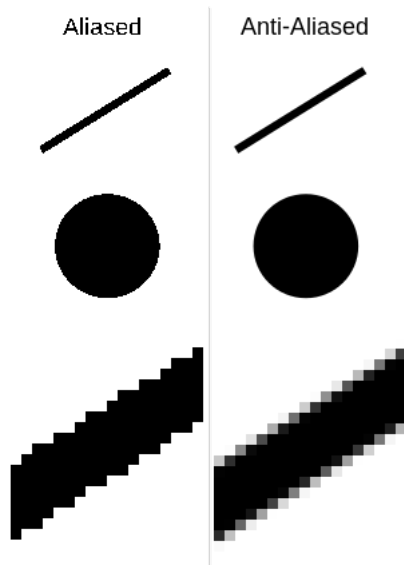
кружок рисуется на скрытый холст. Опять, никаких видимых изменений на экране не будет, так как на экран отображается первичный холст.

- В строке:

```
window.display();
```

скрытый и первичный холст меняются местами. Скрытый холст становится первичным, а первичный скрытым. Теперь всё, что мы нарисовали на скрытый холст станет видимым. Такой способ чередования холстов называется *двойная буферизация*. Он обеспечивает плавность анимации и отсутствие мерцаний. Если бы двойной буферизации не было, то в какие-то моменты времени мы видели бы на экране частично отрисованное изображение кадра, а в какие-то моменты весь кадр. Это выглядело бы как мерцание всех рисуемых объектов.

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 `code/sfml_basics`.

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 шрифта

Класс шрифта `sf::Font` нужен для загрузки данных шрифта с диска и использования этого шрифта при отрисовке текста. Файл шрифта можно найти в интернете, он имеет расширение `.ttf`. Загрузить шрифт в программу можно с помощью метода `loadFromFile` класса `sf::Font`:

```
sf::Font font;  
if (!font.loadFromFile("consola.ttf"))  
{  
    std::cout << "Error. Can't load font!" << std::endl;  
    std::exit(1);  
}
```

Метод `loadFromFile` будет искать файл относительно исполняемого файла. То есть в примере выше нужно убедиться, что файл `consola.ttf` лежит в той же папке, что и исполняемый файл.

Класс текста

`sf::Text` – класс объекта для отображения текста на экране. У объектов этого класса можно задать шрифт, содержимое строки, размер шрифта, цвет, стиль с помощью соответствующих методов. Положение текста на экране задаётся с помощью методов, аналогичных методам фигур: `setPosition`, `move`, `rotate` и других.

Пример программы, которая рисует вращающийся текст

```
#include <SFML/Graphics.hpp>
#include <iostream>
int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Rotating Text", sf::Style::Default);
    window.setFramerateLimit(60);

    sf::Font font;
    if (!font.loadFromFile("consola.ttf"))
    {
        std::cout << "Error! Can't load font!" << std::endl;
        std::exit(1);
    }

    sf::Text text;
    text.setFont(font);
    text.setString(L"Привет");
    text.setCharacterSize(50);
    text.setFillColor(sf::Color(70, 160, 100));
    text.setStyle(sf::Text::Bold | sf::Text::Underlined);
    text.setPosition({300, 200});

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        text.rotate(0.1f);

        window.clear(sf::Color::Black);
        window.draw(text);

        window.display();
    }
}
```

```
#include <SFML/Graphics.hpp>

sf::Text getText(std::string fontFile)
{
    sf::Font font;
    if (!font.loadFromFile(fontFile))
    {
        std::cout << "Error! Can't load font!" << std::endl;
        std::exit(1);
    }

    sf::Text text;
    text.setFont(font);
    text.setString(L"Привет");
    text.setCharacterSize(50);
    text.setFillColor(sf::Color(70, 160, 100));
    text.setStyle(sf::Text::Bold | sf::Text::Underlined);
    text.setPosition({300, 200});
    return text;
}

int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Rotating Text", sf::Style::Default);
    window.setFramerateLimit(60);

    sf::Text text = getText("consola.ttf");

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        text.rotate(0.1f);

        window.clear(sf::Color::Black);
        window.draw(text);

        window.display();
    }
}
```

Часть 3: 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` предстала 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`.

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.

Часть 4: 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 отпускании мыши должен остаться на месте.

Часть 5: Задачи

- **Вращающийся квадрат** Создайте программу, которая будет рисовать на экране вращающийся квадрат.
- **Движение по окружности** Создайте программу, которая будет рисовать на экране круг, движущийся по окружности.
- **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 зажать 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`.
- **Перетаскивание:** В папке `1draggable/` содержится заготовка исходного кода для этого задания. Эта программа просто рисует прямоугольник на экране. Сделайте его перетаскиваемым мышью. При нажатии на него и последующим движении мыши он должен начать двигаться вместе с курсором. При отпуске мыши должен остаться на месте.
- **Класс Draggable:** Создайте класс `Draggable`, который будет описывать прямоугольник, который можно перетаскивать мышью.
- **Кнопка:** Создайте кнопку. Логика работы должна этой кнопки аналогичной логике работы обычной кнопки в ОС Windows:
 - Кнопка представляет собой прямоугольник некоторого цвета и с текстом внутри.
 - Изначально кнопка имеет некоторый заданный цвет.
 - При наведении курсора мыши на кнопку, её цвет меняется.
 - При нажатии и зажатии левой кнопки мыши (ЛКМ) над кнопкой, её цвет меняется.
 - При отпуске ЛКМ, если курсор всё ещё находится на прямоугольнике, происходит некоторое действие (например, печать в консоль).
 - В иных случаях действие не происходит (например, если мы зажали ЛКМ вне кнопки и отпустили над кнопкой, или если мы зажали ЛКМ над кнопкой и отпустили вне кнопки).
- **Класс кнопки:** Напишите класс `Button`, который будет описывать кнопку.
 - Создайте 1 круг. Сделайте так, чтобы при нажатии на кнопку цвет круга менялся бы на случайный.
 - Создайте 4 кнопки. Сделайте так, чтобы при нажатии на эти кнопки положение круга смещалось на 10 пикселей в одном из 4-х направлений (влево, вправо, вверх, вниз).
- **Флажки:** Напишите класс `Checkbox`, который будет описывать флажок. Флажок должен включать в себя квадратик, на который можно нажимать и менять состояние флажка (вкл/выкл), а также текст рядом с этим квадратиком. Создайте несколько флажков и кнопку. При нажатии на кнопку в консоль должно печататься тексты всех включенных флажков.
- **Контекстное меню:** Напишите класс `ContextMenu`, описывающий контекстное меню. Создайте круг. И добавьте опции в контекстное меню так, чтобы можно было менять цвет и размер круга.