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, 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 проделайте следующее:

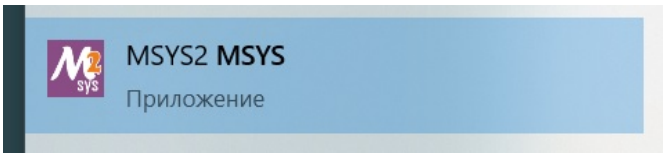
1. Найдите как называется пакет SFML в среде MSYS2. Для этого просто загуглите `msys2 install sfml` и одной из первых ссылок должен быть страница библиотеки SFML сайта `packages.msys2.org`. Зайдите на эту страницу и найдите команду для установки SFML. Скопируйте эту команду. Это может быть команда:

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

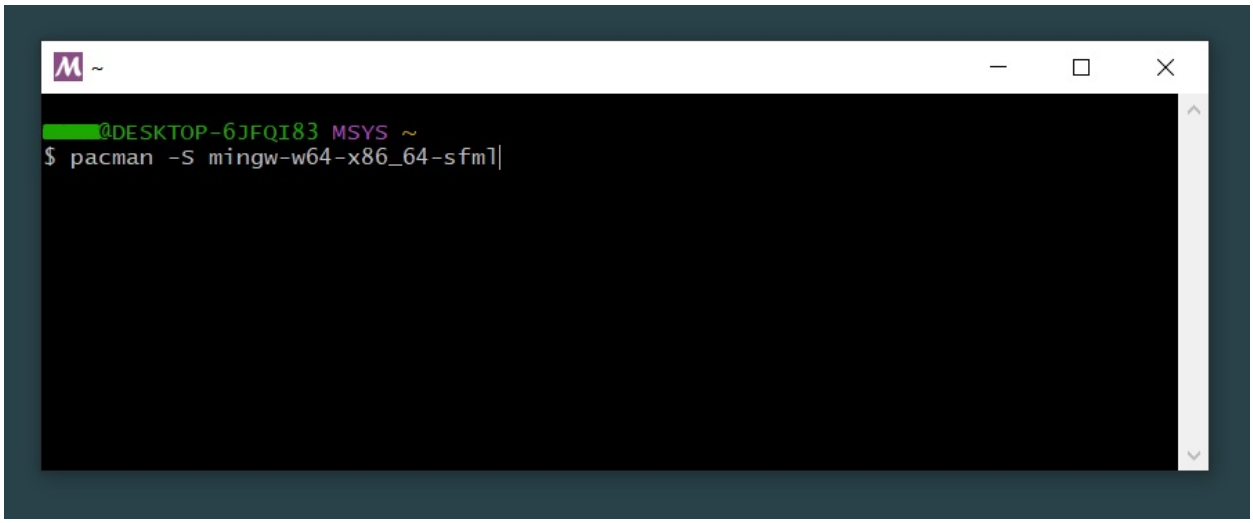
   или просто

   ```
   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".



3. 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.

4. 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.

5. 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.

6. 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:

```cpp
#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.sfml-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
```

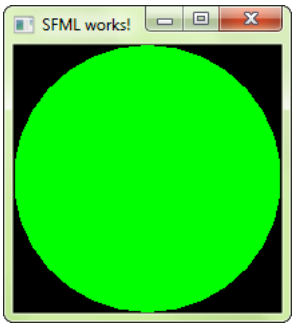
Если SFML был установлен корректно, то программа должна скомпилироваться и в папке должен создаться исполняемый файл `a.exe` (или `a.out` на Linux). Запустите этот файл командой:

```
.\a.exe
```

или, если вы работаете на Linux, то командой:

```
./a.out
```

Если SFML был установлен корректно, то программа должна запуститься, 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 вида `SFML-<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 понадобиться скомпилировать другую программу, то файл `run.bat` можно будет скопировать к этой программе.

Часть 2: Основные классы библиотеки SFML

Классы математических векторов

Классы двумерных математических векторов `sf::Vector2<T>`. У них есть два публичных поля: `x` и `y`. Также, для них перегруженны операции сложения с такими же векторами и умножения на числа. Также введены `typedef`-синонимы вроде `sf::Vector2f` для `sf::Vector2<float>`, `sf::Vector2i` для `sf::Vector2<int>` 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
```

Но нужно помнить, что операции умножения и деления на число перегружены только с числами соответствующих типов. Например, для `sf::Vector2f` есть перегрузки только с числами типа `float`, но нет с числами типа `double` и `int`.

```
sf::Vector2f a {1.0, 2.0};
sf::Vector2f b = 2.0f * a;  // OK
sf::Vector2f c = 2.0 * a;   // Ошибка, нет перегрузки с числом типа double
sf::Vector2f d = 2 * a;     // Ошибка, нет перегрузки с числом типа int
```

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 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)` Конструктрор, с двумя обязательными и двумя необязательными аргументами. Его аргументы:
  - Видеорежим `sf::VideoMode m` - определяет размер окна.
  - Заголовок окна `title`
  - Стиль окна `style`, 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 кнопочкой 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 `sf::ContextSettings`, задаёт некоторые настройки окна, такие как антиалиасинг.

- `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
- `hasFocus` – проверяет, активно ли окно.

Простая программа, которая создаёт окно зелёного цвета

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

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear(sf::Color(0, 255, 0));
        window.display();
    }
}
```

Классы фигур

В SFML есть несколько классов для работы с простыми фигурами: `sf::CircleShape` (круг), `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 заданой 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 вполняться 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 будет происходить медленнее. Чтобы скорость движения объектов не зависила от мощности компютера 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 вних. То есть значение `y = 0` будет находится в самом верху экрана, а значение `y = 800` будет находиться в самом низу нашего экрана высотой 800 пикселей.

- Дальше, 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 (например, 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 закраски 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);
  ```

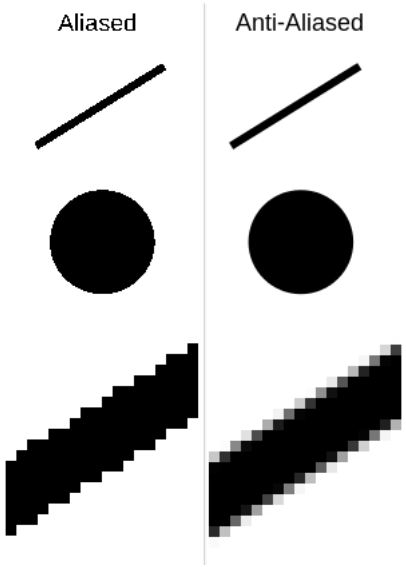
  кружок рисуется на скрытый холст. Опять, никаких видимых изменений на экране не будет, так как на экран отображается первичный холст.

- 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` четвёртым параметром. Пример в папке `code/sfml_basics`.

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`.

```
sf::Time t = sf::seconds(5);
sf::sleep(t);  // Программа будет ожидать 5 секунд
```

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 прямоугольника

Класс прямоугольника `sf::Rect<T>` описывает прямоугольник со сторонами параллельными осям координат. Тип `T` задаёт тип в которых будут хранится числа, описывающие прямоугольник. Есть `typedef`-ы: `sf::FloatRect` для `sf::Rect<float>` и `sf::IntRect` для `sf::Rect<int>`. У этого класса есть следующие публичные поля:

- `left` – x-координата левой стороны прямоугольника
- `top` – y-координата верхней стороны прямоугольника
- `width` – ширина прямоугольника
- `height` – высота прямоугольника

Также у класса есть следующие методы:

- `bool contains(T x, T y)` – проверяет, находится ли точка с координатами `(x, y)` внутри прямоугольника или нет.
- `bool contains(sf::Vector2<T> p)` – проверяет, находится ли точка `p` внутри прямоугольника или нет.
- `bool intersects(const sf::Rect<T>& rect)` – проверяет пересекается ли прямоугольник с прямоугольником `rect`.

Не следует путать класс `sf::Rect<T>` и класс фигуры `sf::RectangleShape`. Класс `sf::Rect<T>` описывает просто прямоугольник со сторонами параллельными осям координат, с ним нельзя почти ничего делать. Класс `sf::RectangleShape` описывает прямоугольную фигуру, объекты такого класса можно перемещать, вращать, масштабировать, рисовать на экран, изменять цвет и так далее.

У фигур есть метод `getGlobalBounds`, который возвращает прямоугольник со сторонами параллельными осям координат, описанный вокруг фигуры.

```
sf::CircleShape c(50);
sf::FloatRect r = c.getGlobalBounds();  // Прямоугольник r описанный вокруг фигуры круга c

// Проверим, находится ли точка p внутри прямоугольника r
sf::Vector2f p {30, 30};
if (r.contains(p))
    std::cout << "Point inside Rect!" << std::endl;
```

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`. Для работы с кириллицей эту строку нужно инициализировать широким строковым литералом. В отличии от обычного он предваряется буквой `L`.

```
sf::String a = "Hello";
sf::String b = L"Привет";
```

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();
    }
}
```

Опасность при использовании `sf::Font` и `sf::Text`

При работе с классами `sf::Font` и `sf::Text` нужно понимать, что `sf::Text` хранит внутри себя не весь шрифт, а только указатель на шрифт. То есть, при задании шрифта у текста:

```
text.setFont(font);
```

в объект `sf::Text` НЕ копируется весь объект `sf::Font`, а копируется только адресс объекта `sf::Font`. Следовательно, если соответствующий объект шрифта уничтожится раньше объекта текста, то в объекте текста будет храниться висячая ссылка на шрифт и работа с таким текстом приведёт к неопределённому поведению.

```cpp
#include <iostream>
#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);

    // Создаём текст, но объект sf::Font создастся и уничтожится внутри функции getText
    sf::Text text = getText("consola.ttf");
    // Далее объект text хранит в себе указатель на удалённый объект sf::Font
    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);  // UB
        window.display();
    }
}
```

Рисование прямых линий

Под линией в программировании понимается не линия в математическом смысле этого слова, а прямой отрезок, ограниченный двумя точками.

Рисование прямых линий в коде не похоже на рисование фигур. В библиотеке нет специального класса, описывающего отдельную линию, но есть класс `sf::Vertex` описывающий вершину, имеющую некоторые координаты и цвет. Для того, чтобы нарисовать линии нужно сначала создать массив, хранящий вершины этих линий. Для рисования одной линии нужен массив из двух вершин, но можно отрисовать сразу много линий, если создать массив из большого количества вершин. После того, как массив создан, нужно вызвать перегрузку метода `draw` класса `sf::RenderWindow`, которая предназначена для рисования линий. У этой перегрузки есть 3 параметра:

- Массив вершин
- Количество вершин для отрисовки
- Тип примитива отрисовки. Может принимать следующие значения:
  - `sf::Lines` рисует не соединённые линии. Если в массиве n вершин, то нарисует $n/2$ линий.
  - `sf::LineStrip` рисует соединённые линии. Если в массиве n вершин, то нарисует $n-1$ линий.
  - Есть ещё несколько примитивов, но мы их использовать не будем.

Пример программы, которая рисует 2 белых линии: линию с координатами `{(0, 0), (50, 100)}` и линию с координатами `{(200, 200), (400, 400)}`.

```cpp
#include <SFML/Graphics.hpp>
int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Drawing Lines", sf::Style::Default);

    sf::Vertex vertices[] =
    {
        sf::Vertex(sf::Vector2f(0, 0),     sf::Color::White),
        sf::Vertex(sf::Vector2f(50, 100),  sf::Color::White),
        sf::Vertex(sf::Vector2f(200, 200), sf::Color::White),
        sf::Vertex(sf::Vector2f(400, 400), sf::Color::White)
    };

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

        window.draw(vertices, 4, sf::Lines);
        // window.draw(vertices, 4, sf::LineStrip);

        window.display();
    }
}
```

Если в качестве типа примитива выбрать `sf::LineStrip`, то дополнительно нарисуется линия с координатами `{(50, 100), (200, 200)}`.

Рисование кривых

SFML не поддерживает рисование кривых линий напрямую. Для того, чтобы нарисовать кривую, нужно нарисовать её, используя большое количество маленьких прямых линий. Например, код ниже рисует синусоиду:

```
#include <cmath>
#include <SFML/Graphics.hpp>
int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Drawing Sin Func", sf::Style::Default);

    std::vector<sf::Vertex> vertices;
    for (int i = 0; i < 200; ++i)
    {
        float x = 4 * i;
        float y = 400 + 100 * std::sin(x / 30);
        vertices.push_back(sf::Vertex(sf::Vector2f(x, y), sf::Color::White));
    }

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear(sf::Color::Black);
        window.draw(vertices.data(), vertices.size(), sf::LineStrip);
        window.display();
    }
}
```

Рисование линий с толщиной

Предыдущий способ рисования линий рисует линии с толщиной в 1 пиксель. SFML не поддерживает рисование линий, толщиной в несколько пикселей напрямую. Есть два способа отрисовки линии с толщиной:

- Использовать тип примитива `sf::TriangleStrip` при отрисовке линий. Но при этом нужно добавить дополнительные вершины в массив.
- Вместо линий можно рисовать очень тонкие прямоугольники, например как это сделано в этой функции:

```
void drawThickLine(sf::RenderWindow& window, sf::Vector2f start, sf::Vector2f finish, float
    thickness, sf::Color c)
{
    static sf::RectangleShape rect;
    sf::Vector2f r = finish - start;
    rect.setSize({std::sqrt(r.x * r.x + r.y * r.y), thickness});
    rect.setFillColor(c);
    rect.setOrigin({0, thickness / 2});
    rect.setPosition(start);
    rect.setRotation(std::atan2(r.y, r.x) * 180 / std::numbers::pi);
    window.draw(rect);
}
```

Класс клавиатуры

Класс клавиатуры `sf::Keyboard` используется для проверки нажата ли та или иная клавиша в данный момент. Для проверки используется статический метод этого класса `isKeyPressed`. Этому методу нужно передать на вход код клавиши. Код клавиши можно получить, также используя класс `sf::Keyboard`.

Внутри класса `sf::Keyboard` находится `enum`, который описывает все эти коды. Примеры кодов клавиш:

`Space, Enter, Escape, LShift, RShift, LControl, Left, Right, Up, Down, W, A, S, D ...`

То есть, например, для клавиши пробел нужно использовать `sf::Keyboard::Space`.

Программа, рисующая круг, который движется только тогда, когда зажата клавиша Пробел (Space)

```
#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();
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))
            circle.move(sf::Vector2f{1, 1});

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

        window.display();
    }
}
```

Класс мыши. Проверка на нажатие кнопок мыши.

Класс мыши `sf::Mouse` похож на класс клавиатуры. Проверка на нажатие кнопок мыши происходит схожим образом:

```
// Проверка на то, что нажата левая кнопка мыши
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
    ...

// Проверка на то, что нажато колёсико мыши
if (sf::Mouse::isButtonPressed(sf::Mouse::Middle))
    ...
```

Для того, чтобы лучше понимать как получать координаты курсора мыши, разберёмся сначала какие системы координат существуют в SFML.

Часть 3: Системы координат в SFML

В библиотеки SFML существуют несколько систем координат. Одни из частых ошибок при работе с библиотекой – это ошибки связанные с системами координат. Если передать в функцию, которая принимает точку в одной системе, точку из другой системы координат, то поведение программы будет не таким, каким вы ожидали.

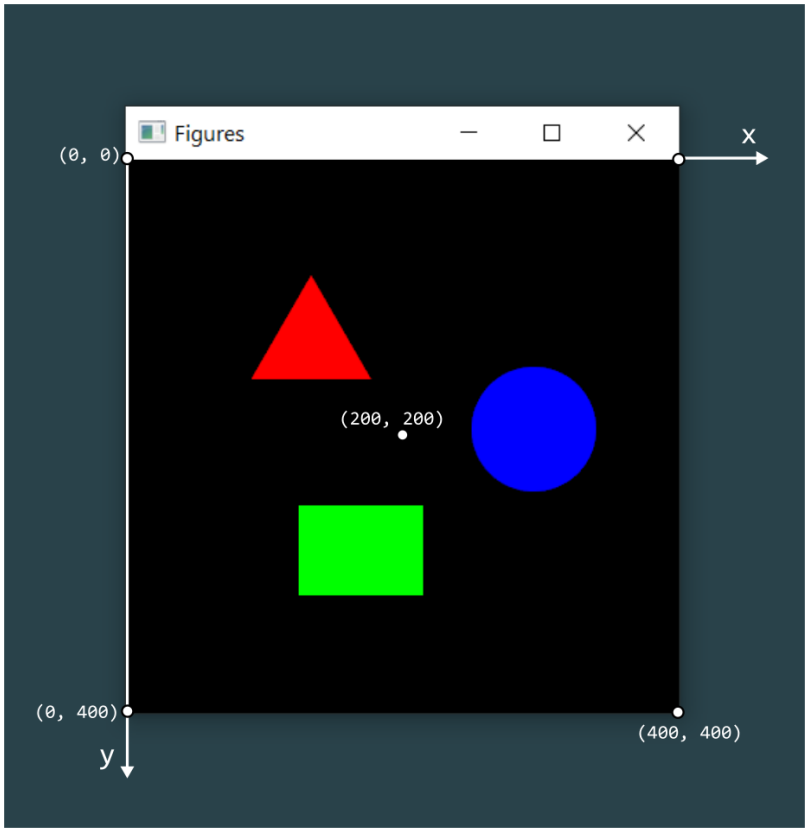
Рассмотрим следующие 2 системы координат:

1. Система координат сетки пикселей
2. Система координат мира объектов

Для определённости будем предполагать, что окно имеет размеры 400 на 400 пикселей.

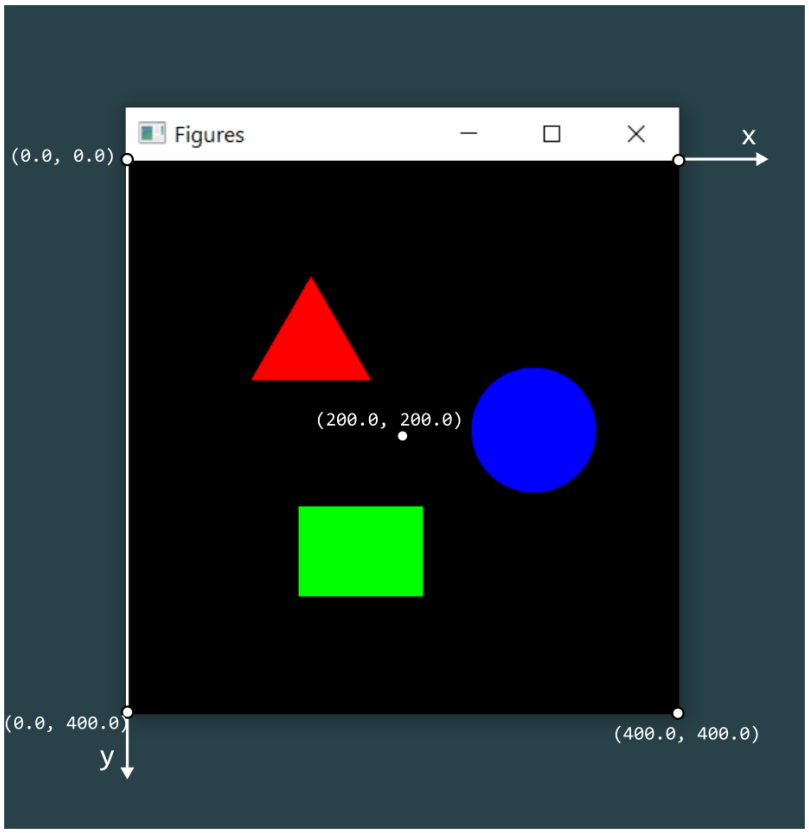
Система координат сетки пикселей.

- В этой системе описывается положение каждого пикселя в окне
- Центр координат – в верхнем левом углу окна.
- Ось Ox идёт слева направо, ось Oy – сверху вниз.
- Эта система целочисленная. Координаты хранятся в числах типа `int`. Для хранения точки обычно используется `sf::Vector2i`.
- Нижний правый угол имеет координаты, соответствующие размерам окна (в данном случае `(400, 400)`).
- При изменении размеров окна, размер также сетки пикселей меняется. Соответственно, при изменении размеров окна, координаты нижнего правого угла изменятся.
- В этой системе координат обычно возращаются координаты курсора мыши функциями библиотеки.



Система координат мира объектов

- В этой системе описывается положение объектов (кругов, прямоугольников, линий и т. д.).
- Центр координат – по умолчанию в верхнем левом углу окна.
- Ось Ox идёт слева направо, ось Oy – сверху вниз.
- Эта вещественная система координат. Координаты хранятся в числах типа `float`. Для хранения точки обычно используется `sf::Vector2f`.
- Нижний правый угол имеет координаты, соответствующие размерам окна при запуске программы (в данному случае `(400.0, 400.0)`).
- При изменении размеров окна, в окно продолжается отображаться та же самая часть мира объектов. Соответственно, при изменении размеров окна координаты нижнего правого угла НЕ изменятся.
- Эту систему координат можно перенастроить, используя класс `sf::View`.

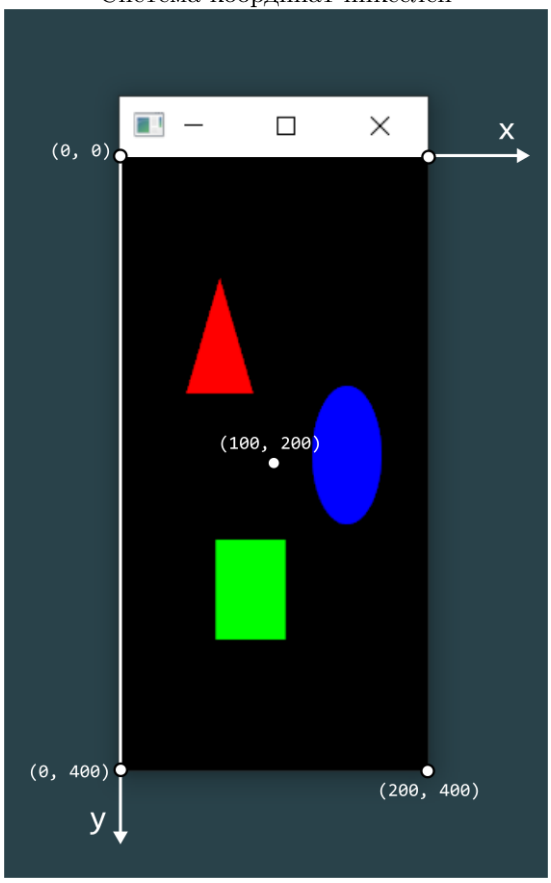


Различие между системой координат пикселей и системой координат мира

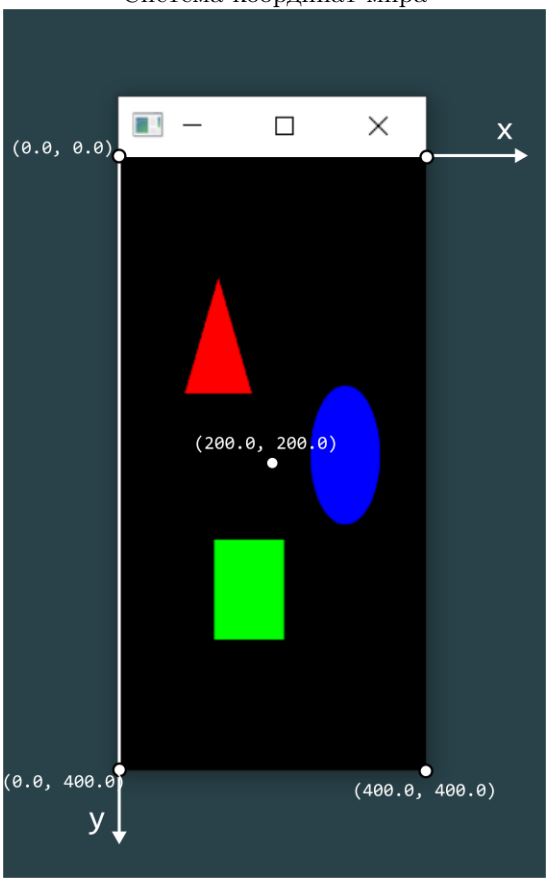
Эти две системы координат можно легко перепутать, так как при запуске программы все точки на экране имеют совпадающие значения в обоих системах координат. Разница же в системах координат проявится только если размер окна будет изменён.

Например, предположим, что у нас было окно размером 400 на 400 пикселей, и затем мы его сжали в 2 раза по горизонтали, так что размер окна стал 200 на 400 пикселей. После такого преобразования сетка пикселей также стала меньше в 2 раза Отрисовываемая область мира же при этом не изменилась. В окне по-прежнему будут отображаться те же самые объект, что и до сжатия, только теперь они будут также сжаты в 2 раза. Теперь точки в окне имеют разные значения в разных системах координат. Если вы будете использовать координаты из одной системы в той функции, где требуются координаты из другой системы, то программа будет вести себя некорректно.

Система координат пикселей



Система координат мира



Конвертация координат из системы координат пикселей в систему координат мира

Для конвертации точки из одной системы координат в другую используются методы класса `sf::RenderWindow`:

- `Vector2f mapPixelToCoords(Vector2i point)` – конвертирует точку из системы координат пикселей в систему координат мира.
- `Vector2i mapCoordsToPixel(Vector2f point)` – конвертирует точку из системы координат мира в систему координат пикселей.

```
sf::Vector2i p {200, 200};                  // Координаты пикселя (200, 200)
sf::Vector2f q = window.mapPixelToCoords(p); // Координаты точки в мире, соответствующие
                                             // этому пикселю
```

Класс мыши. Получение координат курсора мыши.

Помимо того, что класс мыши `sf::Mouse` позволяет проверить была ли нажата та или иная кнопка мыши, он также может может дать координаты курсора мыши. Для этой цели в классе `sf::Mouse` есть следующие статические методы:

- `getPosition(const sf::Window&)` – возвращает положение мыши на в координатах пикселей окна.
- `setPosition(const sf::Vector2i&, const sf::Window&)` – устанавливает положение мыши в координатах пикселей окна.

Обратите внимание, что эти функции работают в системе координат пикселей. Если вы хотите использовать координаты мыши, полученные из этих функций, в функции, которая работает в системе координат мира, то нужно будет сделать преобразование координат с помощью метода `mapPixelToCoords`.

Пример программы, которая перемещает круг к курсору, при нажатии на ЛКМ

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

    sf::CircleShape c(50);
    c.setOrigin({50, 50});
    c.setFillColor(sf::Color::Green);

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

        if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
        {
            sf::Vector2i mousePixel = sf::Mouse::getPosition(window);
            sf::Vector2f mousePosition = window.mapPixelToCoords(mousePixel);
            c.setPosition(mousePosition);
        }
        window.clear(sf::Color::Black);
        window.draw(c);
        window.display();
    }
}
```

Попробуйте самостоятельно ответить на следующие вопросы:

- Что делает метод `setOrigin` класса `sf::CircleShape`? Почему в этот метод передаются именно такие значения? Что будет если убрать строку с вызовом этого метода или передать другие значения?
- Что если не делать преобразование координат `mapPixelToCoords`, а просто сразу передать координаты пикселя в метод `setPosition`? Будет ли программа работать корректно?

```
sf::Vector2i mousePixel = sf::Mouse::getPosition(window);
c.setPosition({(float)mousePixel.x, (float)mousePixel.y});
```

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 прямоуголиник окрашивался в красный цвет тогда и только тогда, когда курсор мыши находится на прямоугольнике. Во всё остальное время прямоуголиник 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

- 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 6: 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 накладываются 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 отпускании мыши должен остаться на месте.
- **Класс `Draggable`:** Создайте класс `Draggable`, который будет описывать прямоугольник, который можно перетаскивать мышкой.
- **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 отпускании ЛКМ, если курсор всё ещё находится на прямоугольнике, происходит некоторое действие (например, печать в консоль).
  - В иных случая 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.