# 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:

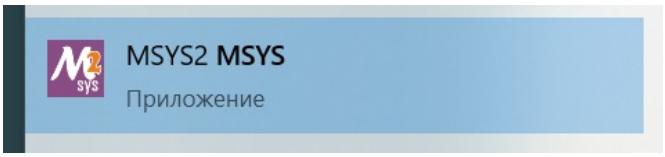
1. Найдите как называется пакет SFML в среде MSYS2. Для этого просто загуглите `msys2 install sfml` и одной из первых ссылок должен быть страница библиотеки SFML сайта `packages.msys2.org`. Зайдите на эту страницу и найдите команду для установки SFML. Скопируйте эту команду. Это может быть команда:

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

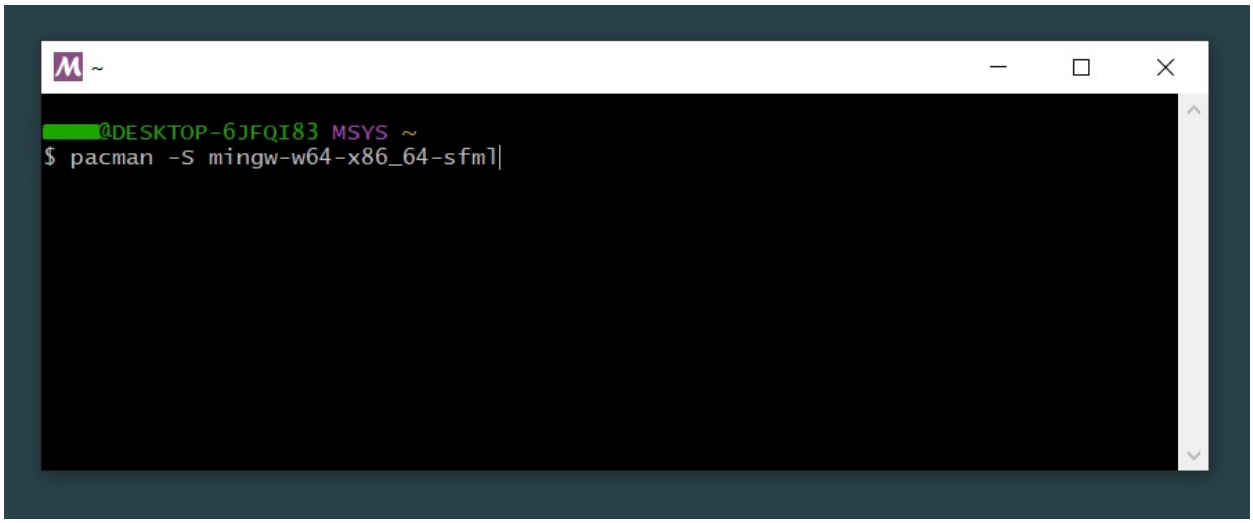
   или просто

   ```
   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
```

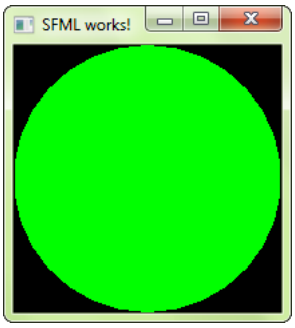
Если SFML был установлен корректно, то программа должна скомпилироваться и в папке должен создаться исполняемый файл `a.exe` (или `a.out` на Linux). Запустите этот файл командой:

```
.\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>`.

```
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::Black`, `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 строках:

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

  создаём объект круга радиуса `30` 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 вправо по оси X и на один 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` $\times$ `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);
  ```

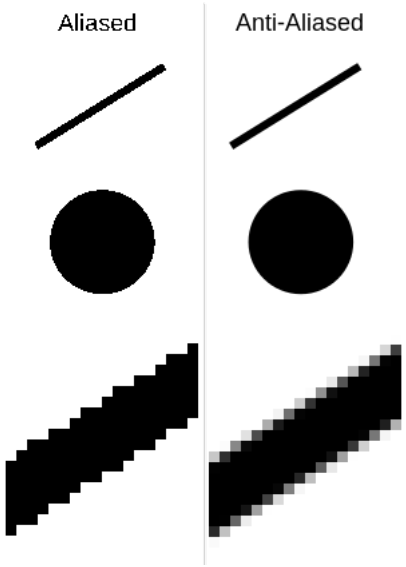
  кружок рисуется на скрытый холст. Опять, никаких видимых изменений на экране не будет, так как на экран отображается первичный холст.

- 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/0sfml_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.

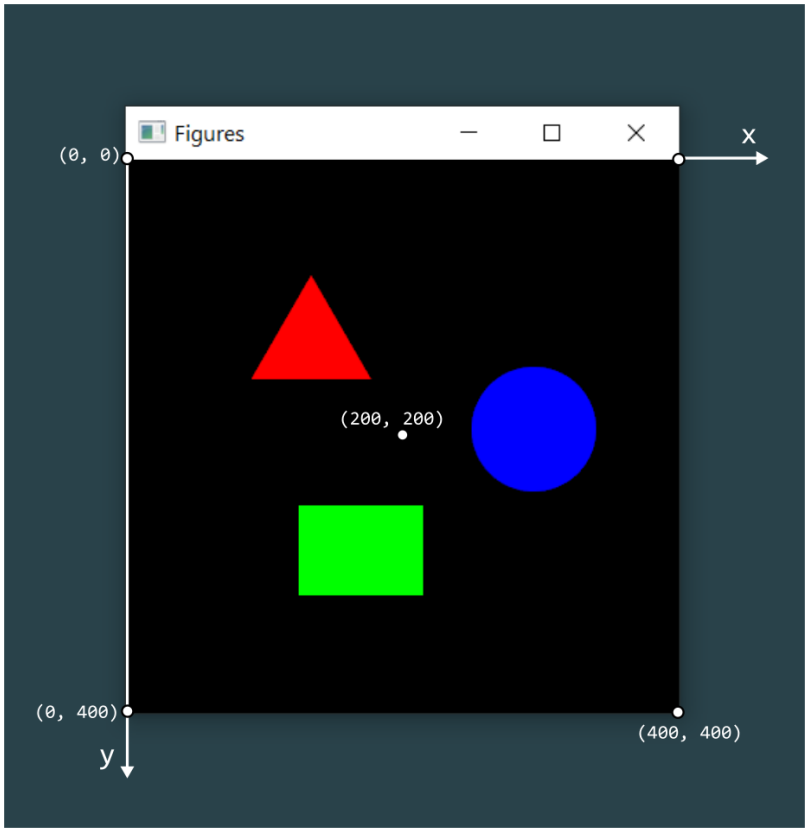
Рассмотрим следующие 2 системы координат:

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

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

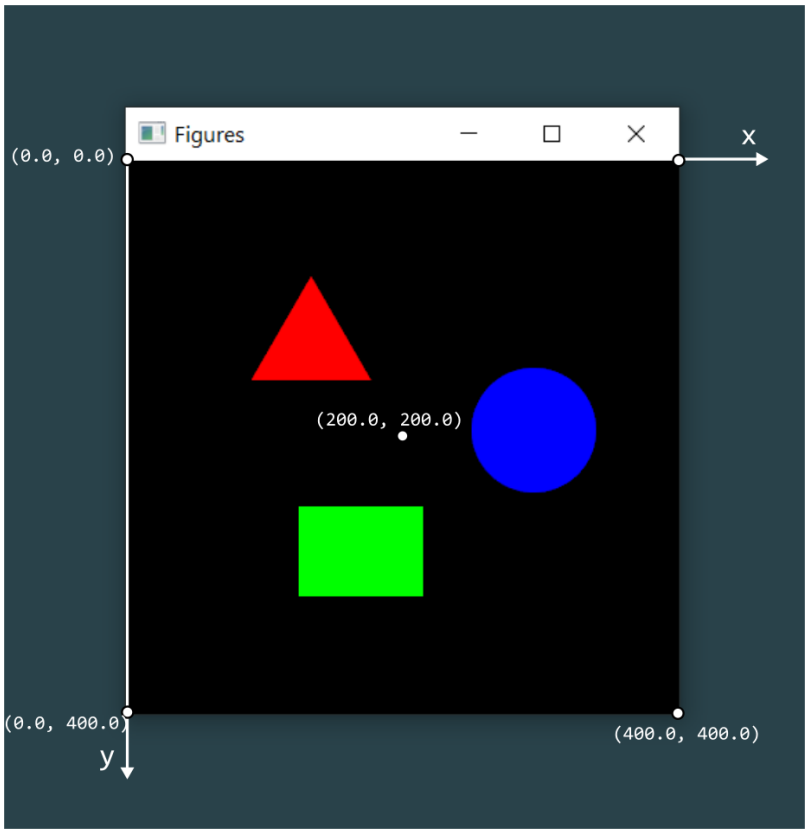
## Система координат сетки пикселей.

- 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 окне 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`.

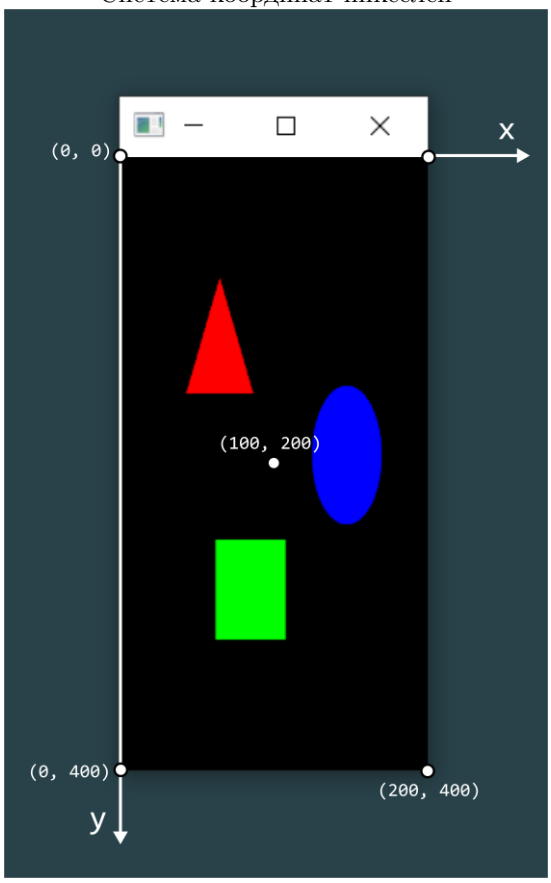


## Различие между системой координат пикселей и системой координат мира

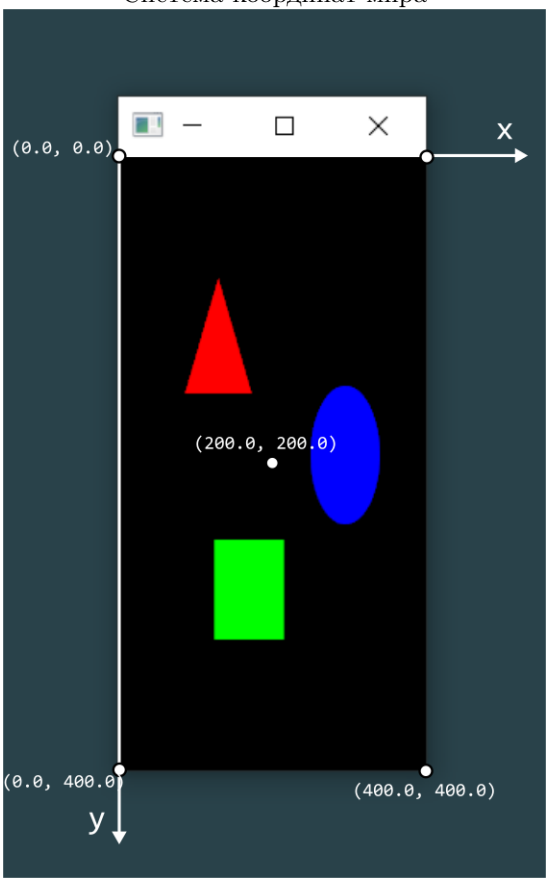
Эти две системы координат можно легко перепутать, так как при запуске программы все точки на экране имеют совпадающие значения в обоих системах координат. Разница же в системах координат проявится только если размер окна будет изменён.

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 курсора мыши в координатах пикселей окна.
- `setPosition(const sf::Vector2i&, const sf::Window&)` – устанавливает положение курсора 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` как это сделано в строках ниже? Будет ли программа работать корректно?

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

# Часть 4: Примеры программ с использованием библиотеки SFML

Рассмотрим примеры программ, рисующих движущиеся шарики в 2-х измерениях. Больше примеров можно найти в папках `code/1ball_movement` и `code/2many_balls`.

## Зацикленное движение шарика в 2d пространстве

Программа рисует один шарик, двигающийся с постоянной скоростью, но если шарик выходит за границы окна, он появляется с другой стороны.

Для того, чтобы добиться такого поведения шарика, мы будем просто проверять координаты шарика каждый кадр, и если, например, x-координата шарика будет больше ширины экрана, то мы просто будем вычитать из x-координата шарика значение ширины экрана. Более точно, нужно учесть ещё радиус шарика и перемещать его только тогда, когда он полностью выходит за границы окна.

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Moving Ball Cycle", sf::Style::Default);
    window.setFramerateLimit(60);

    // Создаём шарик радиуса 30 в точке с координатами (100, 100)
    sf::CircleShape ball(30);
    ball.setOrigin(sf::Vector2f{ball.getRadius(), ball.getRadius()});
    ball.setPosition(sf::Vector2f{100, 100});

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

        // Передвигаем шарик на вектор (10, 5)
        ball.move(sf::Vector2f{10, 5});

        // Если шарик выходит за границы окна, он появляется с другой стороны:
        sf::Vector2f max = window.getView().getSize(); // размеры окна
        if (ball.getPosition().x - ball.getRadius() > max.x)
            ball.move({-max.x - 2 * ball.getRadius(), 0});
        if (ball.getPosition().y - ball.getRadius() > max.y)
            ball.move({0, -max.y - 2 * ball.getRadius()});

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

        window.display();
    }
}
```

## Движение шарика в гравитационном поле с упругими столкновениями от границ

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

    // Создаём шарик радиуса 30 в точке (100, 100)
    sf::CircleShape ball(30);
    ball.setOrigin({ball.getRadius(), ball.getRadius()});
    ball.setPosition(sf::Vector2f{100, 100});

    // Задаём начальную скорость и ускорение шарика
    sf::Vector2f ballVelocity {300, 0};
    sf::Vector2f ballAcceleration {0, 1000};
    float dt = 1.0f / 60;

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

        // Уравнения движения
        ballVelocity += ballAcceleration * dt;
        ball.move(ballVelocity * dt);

        sf::Vector2f max = window.getView().getSize(); // размеры окна

        // Обрабатываем столкновения со стенками:
        if (ball.getPosition().x + ball.getRadius() > max.x)
            ballVelocity.x *= -1;

        if (ball.getPosition().x - ball.getRadius() < 0)
            ballVelocity.x *= -1;

        if (ball.getPosition().y + ball.getRadius() > max.y)
            ballVelocity.y *= -1;

        if (ball.getPosition().y - ball.getRadius() < 0)
            ballVelocity.y *= -1;

        window.clear(sf::Color::Black);
        window.draw(ball);
        window.display();
    }
}
```

### Работа с несколькими шариками

Для создания программы, которая рисует множество шариков, удобно написать класс, отвечающий за управление движением каждого отдельного шарика.

```
class Ball
{
private:
    sf::RenderWindow& mRenderWindow;
    sf::Vector2f mPosition {};
    sf::Vector2f mVelocity {};
    float mRadius          {};
public:
    Ball(sf::RenderWindow& window) : mRenderWindow{window} {}

    void update(float dt)
    {
        mPosition += mVelocity * dt;
        handleCollisions();
    }

    void handleCollisions()
    {
        sf::Vector2f max = mRenderWindow.getView().getSize();
        if (float dx = mPosition.x + mRadius - max.x; dx > 0)
            mVelocity.x *= -1;

        if (float dx = -mPosition.x + mRadius; dx > 0)
            mVelocity.x *= -1;

        if (float dy = mPosition.y + mRadius - max.y; dy > 0)
            mVelocity.y *= -1;

        if (float dy = -mPosition.y + mRadius; dy > 0)
            mVelocity.y *= -1;
    }

    void draw() const
    {
        static sf::CircleShape circle;
        circle.setFillColor(sf::Color::White);
        circle.setRadius(mRadius);
        circle.setOrigin({mRadius, mRadius});
        circle.setPosition(mPosition);
        mRenderWindow.draw(circle);
    }
    void setPosition(sf::Vector2f position) {mPosition = position;}
    void setVelocity(sf::Vector2f velocity) {mVelocity = velocity;}
    void setRadius(float radius)            {mRadius = radius;}
    sf::Vector2f getPosition() const {return mPosition;}
    sf::Vector2f getVelocity() const {return mVelocity;}
    float        getRadius()   const {return mRadius;}
};
```

После этого создадим контейнер (например, `std::vector`) и будем хранить в нём необходимое число шариков:

```
sf::RenderWindow window(sf::VideoMode(800, 800), "Many Balls + Collisions");
window.setFramerateLimit(60);
float dt = 1.0f / 60;

std::vector<Ball> balls;
for (int i = 0; i < 10; ++i)
{
    Ball ball{window};
    ball.setRadius(10);
    ball.setPosition({200.0f + 50.0f * i, 600.0f - 50.0f * i});
    ball.setVelocity({500, 500});
    balls.push_back(ball);
}
```

Для передвижения шариков, обработки столкновений со стенками и отрисовки, просто пробегаем по контейнеру и вызываем соответствующий метод шарика:

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

    for (auto& ball : balls)
        ball.update(dt);

    window.clear(sf::Color::Black);
    for (auto& ball : balls)
        ball.draw();
    window.display();
}
```

# Часть 5: События

События (англ. *events*) – это механизм обработки пользовательского ввода и других действий, происходящих в окне приложения. События могут включать в себя нажатия клавиш, движение мыши, закрытие окна и т. д. Класс `sf::Event` используется для хранения информации о произошедшем событии. Объекты этого класса содержат тип события и дополнительные данные, связанные с ним (например, какая клавиша была нажата или координаты курсора мыши).

Рассмотрим пример программы, использующей события. Следующая программа рисует круг и передвигает его при каждом нажатии на клавишу пробел (`sf::Keyboard::Space`):

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

    sf::CircleShape circle(50);
    circle.setOrigin({circle.getRadius(), circle.getRadius()});
    circle.setPosition({100, 400});

    while (window.isOpen())  // Главный цикл
    {
        sf::Event event;
        while (window.pollEvent(event))  // Цикл обработки событий
        {
            if (event.type == sf::Event::Closed)
                window.close();

            if (event.type == sf::Event::KeyPressed)
            {
                if (event.key.code == sf::Keyboard::Space)
                    circle.move({50, 0});
            }
        }

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

Пояснения по программе:

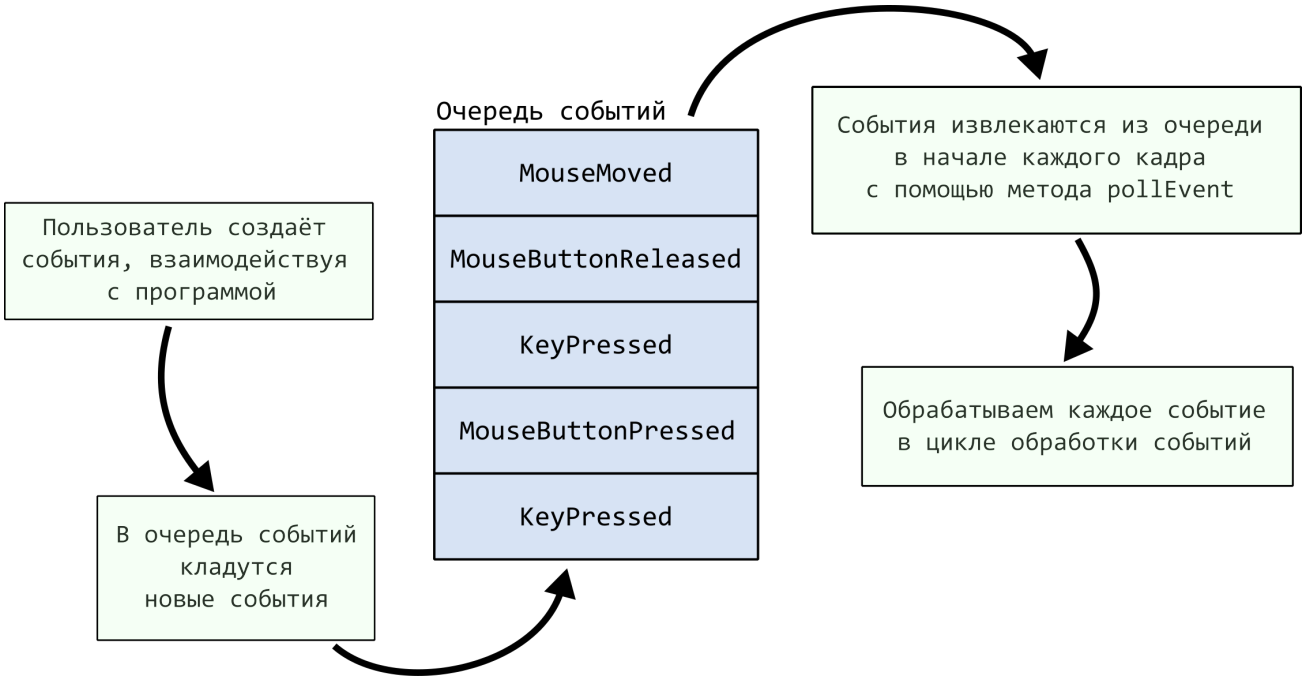
- Сначала мы создаём объект типа `sf::Event`, который может хранить внутри себя информацию о событии.
- Метод `pollEvent` принимает объект `event` по ссылке, затем извлекает одно событие из внутренней *очереди событий* и присваивает объекту `event` это событие. Метод `pollEvent` возвращает `true`, если в очереди событий было хотя бы одно событие, и `false`, если очередь событий была пуста. Таким образом цикл обработки событий работает до тех пор пока в очереди событий не закончатся события.
- Очередь событий – это специальная очередь внутри SFML, которая хранит в себе все события, произошедшие за время текущего кадра. В начале каждого кадра мы извлекаем поочерёдно все события из этой очереди и обрабатываем их.
- После того как событие из очереди получено, его нужно обработать. Будем обрабатывать по-разному в зависимости от типа события. Тип события хранится в поле `event.type`.

### Типы событий в библиотеке SFML

- `sf::Event::Closed` – событие, которое происходит когда пользователь пытается закрыть окно, например, когда нажимает на красный крестик в углу окна или когда использует комбинацию клавиш `Alt + F4`.
- `sf::Event::KeyPressed` – происходит когда пользователь нажал на какую либо клавишу. Код клавиши хранится в поле `event.key.code`.
- `sf::Event::KeyReleased` – происходит когда пользователь отпустил какую либо клавишу. Код клавиши также хранится в поле `event.key.code`.
- `sf::Event::MouseButtonPressed` – происходит когда пользователь нажал на какую либо кнопку мыши. Код кнопки мыши хранится в поле `event.mouseButton.button`. Координаты мыши в момент нажатия в системе координат пикселей хранятся в полях `event.mouseButton.x` и `event.mouseButton.y`.
- `sf::Event::MouseButtonReleased` – происходит когда пользователь отпустил какую либо кнопку мыши. Код кнопки мыши хранится в поле `event.mouseButton.button`. Координаты мыши в момент отпускания в системе координат пикселей хранятся в полях `event.mouseButton.x` и `event.mouseButton.y`.
- `sf::Event::MouseMoved` – происходит когда пользователь передвинул курсор мыши. Координаты мыши в момент передвижения курсора в системе координат пикселей хранятся в полях `event.mouseMove.x` и `event.mouseMove.y`. Обратите внимание, что название поля в котором хранятся координаты мыши отличается для этого события.
- `sf::Event::MouseWheelScrolled` – происходит когда пользователь прокручивает колёсико мыши.

### Очередь событий

Очередь событий (англ. *event queue*) – это очередь, хранящая объекты событий (`std::queue<sf::Event>`) и находящаяся внутри библиотеки SFML. Напрямую с этой очередь мы взаимодействовать не можем. События добавляются в эту очередь автоматически при нажатии на клавиши, кнопок и т. д.. События извлекаются из очереди с помощью метода `pollEvent`.




В начале каждого кадра из очереди извлекаются и обрабатываются все накопленные события. Это означает, что в любой момент времени в очереди находятся только те события, которые произошли в течение текущего кадра

### Пример обработки событий. События нажатия и отпускания кнопки мыши

Данная программа, обрабатывает нажатия и отпускания левой клавиши мыши (ЛКМ). Программа рисует белый круг. При нажатии на ЛКМ круг меняет свой цвет на красный. При отпускании ЛКМ круг опять становится белым.

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 800), "Mouse Button Pressed or Relesed");
    window.setFramerateLimit(60);

    sf::CircleShape circle(200);
    circle.setPosition({200, 200});

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

            if (event.type == sf::Event::MouseButtonPressed)
            {
                if (event.mouseButton.button == sf::Mouse::Left)
                    circle.setFillColor(sf::Color::Red);
            }

            if (event.type == sf::Event::MouseButtonReleased)
            {
                if (event.mouseButton.button == sf::Mouse::Left)
                    circle.setFillColor(sf::Color::White);
            }
        }

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

### Пример обработки событий. Получение координат мыши при нажатии.

Координаты мыши для событий `MouseButtonPressed` и `MouseButtonReleased` хранятся в полях `event.mouseButton.x` и `event.mouseButton.y`. Однако это координаты в системе координат пикселей. Для того, чтобы перевести эти координаты в систему координат мира нужно использовать метод `mapPixelToCoords`. Если этого не сделать, то программа будет работать некорректно, при изменении размеров окна.

Следующая программа рисует кружок. При нажатии на левую кнопку мыши кружок меняет положение на то, в котором находится курсор. Также, при нажатии, программа печатает в консоль координаты мыши в системе координат пикселей и в системе координат мира.

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(800, 600), "Mouse Button Pressed Coordinates");
    window.setFramerateLimit(60);

    sf::CircleShape circle(30);
    circle.setOrigin({circle.getRadius(), circle.getRadius()});
    circle.setPosition({200, 200});

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

            if (event.type == sf::Event::MouseButtonPressed)
            {
                if (event.mouseButton.button == sf::Mouse::Left)
                {
                    sf::Vector2i mousePixel {event.mouseButton.x, event.mouseButton.y};
                    sf::Vector2f mousePosition = window.mapPixelToCoords(mousePixel);

                    std::cout << "Left Mouse Button is pressed\n";
                    std::cout << "Pixel: " << mousePixel.x << " " << mousePixel.y;
                    std::cout << std::endl;
                    std::cout << "Position: " << mousePosition.x << " " << mousePosition.y;
                    std::cout << std::endl;

                    circle.setPosition(mousePosition);
                }
            }
        }

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

### Пример обработки событий. Получение координат мыши при перемещении курсора.

Следующая программа рисует кружок, который перемещается вслед за курсором. Для этого обрабатывается событие `sf::Event::MouseMoved`. Координаты мыши в этом событии хранятся в полях `event.mouseMove.x` и `event.mouseMove.y`.

**Обратите внимание, что названия полей в котором хранятся координаты мыши для этого события отличается от событий нажатия и отпускания кнопки мыши!**. Если использовать поля `event.mouseButton.x` и `event.mouseButton.y` вместо полей `event.mouseMove.x` и `event.mouseMove.y` при обработке этого события, это приведёт к некорректной работе программы.

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

    sf::CircleShape circle(30);
    circle.setOrigin({circle.getRadius(), circle.getRadius()});
    circle.setPosition({400, 400});

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

            if (event.type == sf::Event::MouseMoved)
            {
                sf::Vector2i mousePixel {event.mouseMove.x, event.mouseMove.y};
                sf::Vector2f mousePosition = window.mapPixelToCoords(mousePixel);
                circle.setPosition(mousePosition);
            }
        }
        window.clear(sf::Color::Black);
        window.draw(circle);
        window.display();
    }
}
```

Эти и другие примеры работы с событиями в SFML можно найти в папке `code/4events`.

## Различие между двумя видами обработки нажатий клавиш и кнопок

Как вы могли заметить, в библиотеке SFML есть два способа обработки нажатий на клавиши и кнопки:

1. Проверка, нажата ли та или иная клавиша в данный момент, с помощью статических методов классов `sf::Keyboard` и `sf::Mouse`.
2. Использование механизма событий.

Два этих способа различаются друг от друга. Разберём отличия на примере нажатия левой кнопки мыши:

### Проверка нажатия на ЛКМ с использованием метода класса `sf::Mouse`

```
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))
        std::cout << "LMB is pressed" << std::endl;

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

- Проверяет, зажата ли ЛКМ в данный момент.
- Действие, соответствующее зажатию клавиши будет происходить каждый кадр во время которого была зажата ЛКМ.
- Если вы нажали ЛКМ, а затем отпустили через 10 кадров, то действие произойдёт 10 раз.
- Если вы нажали и отпустили ЛКМ вне проверки (это может произойти если программа "подвисла"), то действие не произойдёт ни разу.
- Лучше использовать, если нужно проверить, что кнопка `зажата`.

### Проверка нажатия на ЛКМ с использованием событий

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

        if (event.type == sf::Event::MouseButtonPressed)
        {
            if (event.mouseButton.button == sf::Mouse::Left)
                std::cout << "LMB is pressed" << std::endl;
        }
    }

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

- Проверяет, была ли нажата ЛКМ с момента предыдущего цикла обработки событий.
- Действие, соответствующее нажатию клавиши произойдёт ровно 1 раз за каждое нажатие.
- Если вы нажали ЛКМ, а затем отпустили через 10 кадров, то действие произойдёт 1 раз.
- Если вы нажали и отпустили ЛКМ вне проверки (это может произойти если программа "подвисла"), то действие тоже произойдёт 1 раз.
- Лучше использовать, если нужно отловить нажатие кнопки. Под нажатием понимается момент перехода из состояния `кнопка не зажата` в состояние `кнопка зажата`.

Предположим, что программа работает с частотой 100 кадров в секунду. Далее, предположим, что вы нажали на кнопку мыши и держали её зажатой 5 секунд, а потом отпустили. Тогда процесс нажатия, зажатия и отпускания кнопки мыши можно проиллюстрировать на следующей схеме:


## Автоповтор при нажатиях на клавиши клавиатуры

События, возникающие при нажатии клавиш клавиатуры, несколько отличаются от событий, связанных с нажатием кнопок мыши. Чтобы наглядно продемонстрировать это различие, рассмотрим конкретный пример. Представьте, что вы открыли текстовый редактор и зажали клавишу `A` на некоторое время. Что произойдёт? Будет ли добавлен только один символ `A`, или новые символы будут появляться непрерывно? На самом деле, процесс будет следующим:

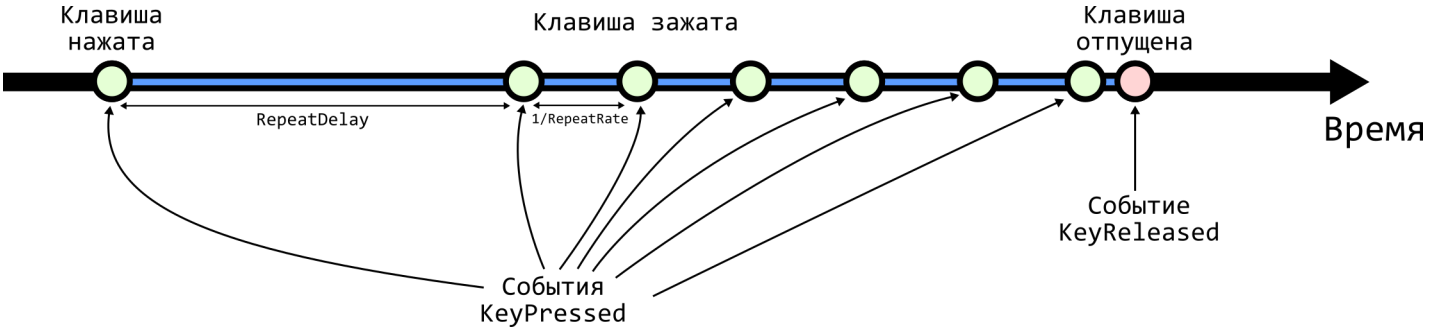
- Первый символ `A` добавиться сразу при нажатии.
- Второй символ `A` добавиться через некоторое относительно долгое время.
- Все последующие символы `A` будут добавляться с очень короткой задержкой между ними.

Такое поведение при зажатиях клавиш наблюдается не только в текстовых редекторах, но и в большинстве других приложений. Это связано с тем, что данная функциональность реализована на уровне операционной системы. Она была добавлена для того, чтобы сделать взаимодействие с клавиатурой более удобным и предсказуемым. Таким образом, с точки зрения операционной системы, событие, соответствующее нажатию клавиши, происходит не только в момент её физического нажатия, но и в определённые моменты при её удержании. Это позволяет системе генерировать повторяющиеся события, которые обрабатываются приложениями для реализации автоповтора.

Вводятся следующие определения:

- Задержка повтора (англ. *Repeat Delay*) – время между моментом нажатия клавиши и началом автоповтора событий, связанных с этой клавишей, при её удержании. Значение этого параметра равно около *500 мс*.
- Частота повторений (англ. *Repeat Rate*) – частота, с которым события повторяются после начала автоповтора при удержании клавиши. Обычно, значение этого параметра равно примерно *1 / (40 мс)*. Обратите внимание, что даже при автоповторе события обычно повторяются реже чем кадры.

Процесс нажатия, зажатия и отпускания клавиши клавиатуры можно проиллюстрировать на схеме:




В библиотеке SFML событие `sf::Event::KeyPressed` происходит не только при непосредственном нажатии на клавишу, но и при автоповторе.

# Часть 6: Реализация элементов графического интерфейса

Графический интерфейс пользователя (англ. *Graphical User Interface* или *GUI*) — это способ взаимодействия пользователя с программой или устройством, основанный на визуальных элементах, таких как кнопки, меню, окна, иконки и другие графические компоненты. В данной части попробуем реализовать некоторые простые элементы графического интерфейса, используя библиотеку SFML. Больще примеров реализаций можно найти в папке `code/5gui`.

## Перетаскиваемая табличка

Перетаскиваемая табличка (англ. *draggable*) – простейший элемент графического интерфейса. Представляет собой прямоугольник, который можно перетаскивать, используя мышь.

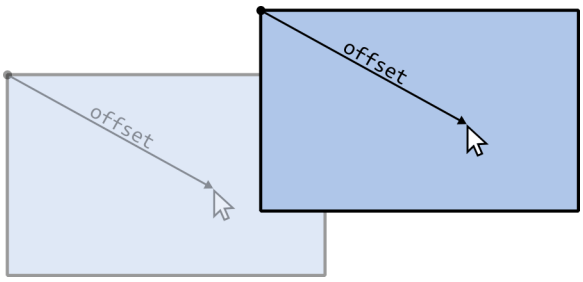
### Реализация перетаскиваемой таблички без использования класса

Для реализации такого элемента интерфейса заведём следующие переменные:

- `draggableShape` – прямоугольник, то есть объект типа `sf::RectangleShape`, используемый для хранения положения и цвета прямоугольника, а также для его отрисовки.
- `isDragged` – булевое значение, которое равно `true`, в те моменты, когда табличка перетаскивается.
- `offset` – двумерный вектор, равный разницей между положением курсора мыши и положением верхнего левого угла таблички в момент последнего нажатия на табличку.

Будем делать следующее:

- При событии нажатия на левую кнопку мыши:
  - Проверять, находился ли курсор мыши внутри прямоугольника в момент нажатия.
    Если да, то делаем следующие два шага.
  - Устанавливаем значение переменной `isDragged = true`.
  - Устанавливаем значение переменной `offset = mousePosition - draggableShape.getPosition()`
- При событии движения курсора мыши:
  - Если `isDragged` находится в состоянии `true`, то будем устанавливаем значение положения прямоугольника как `draggableShape.setPosition(mousePosition - offset)`.
- При событии отпускания левой кнопки мыши:
  - Устанавливаем значение переменной `isDragged = false`.




Полный код этой реализации можно найти в папке `code/5gui/0draggable/0no_class`.

### Реализация перетаскиваемой таблички с созданием класса `Draggable`

Гораздо удобнее создать класс (назовём его `Draggable`), который будет полностью описывать перетаскиваемые таблички. В этот класс мы поместим все необходимые переменные и функции, описывающие этот элемент интерфейса. Полный код этой реализации можно найти в папке `code/5gui/0draggable/1using_class`.

После того, как такой класс создан, можно легко написать программу, которая использует сразу несколько таких перетаскиваемых табличек. Пример можно посмотреть в `code/5gui/0draggable/2cards`.

## Кнопка

Попробуем создать кнопку. Логика работы кнопки должна быть аналогичной логике работы обычной кнопки в ОС Windows. Требования к простейшей кнопке такие:

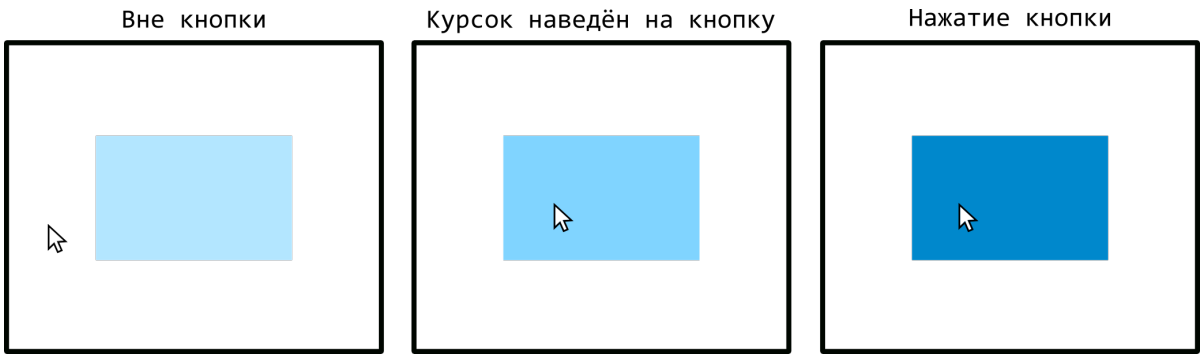
1. Кнопка представляет собой прямоугольник.
2. Изначально кнопка имеет некоторый заданный цвет.
3. При наведении курсора мыши на кнопку, её цвет меняется.
4. При нажатии и зажатии левой кнопки мыши (ЛКМ) над кнопкой, её цвет меняется на цвет, отличный от первых двух.
5. При отпускании ЛКМ, если курсор всё ещё находится на прямоугольнике, происходит некоторое действие (например, печать в консоль).
6. 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).

Реализацию простейшей кнопки можно найти тут: `code/5gui/1button`. Для того, чтобы реализовать кнопку, создадим класс `Button`, который будет состоять из следующих полей:

- `sf::RectangleShape mShape` – объект класса прямоугольника SFML. Внутри `mShape` хранятся координаты, размеры и текущий цвет прямоугольника, предствляющего нашу кнопку.
- `sf::Color mDefaultColor` – цвет кнопки, когда она не зажата и на неё не наведён курсор.
- `sf::Color mHoverColor` – цвет кнопки, когда она не зажата, но на неё наведён курсор.
- `sf::Color mPressedColor` – цвет кнопки, когда она находится в зажатом состоянии.
- `bool mIsPressed` – переменная, равная `true`, когда кнопка находится в зажатом состоянии.
- `sf::RenderWindow& mRenderWindow` – также храним ссылку на окно SFML, на которое будем отрисовывать кнопку. Эту ссылку можно было бы не хранить, а просто передавать во все функции, где окно понадобится, но тогда код был бы более громоздким.

Также у класса будут следующие публичные методы:

- `void draw()` – рисует кнопку на окне. Ссылка на окно хранится внутри объекта `Button`. Поэтому тут её передавать не нужно.
- `bool handleEvent(const sf::Event& event)` – принимает событие, и обрабатывает всё, что касается кнопки. Возвращает `true`, если кнопка была нажата за время предыдущего кадра.

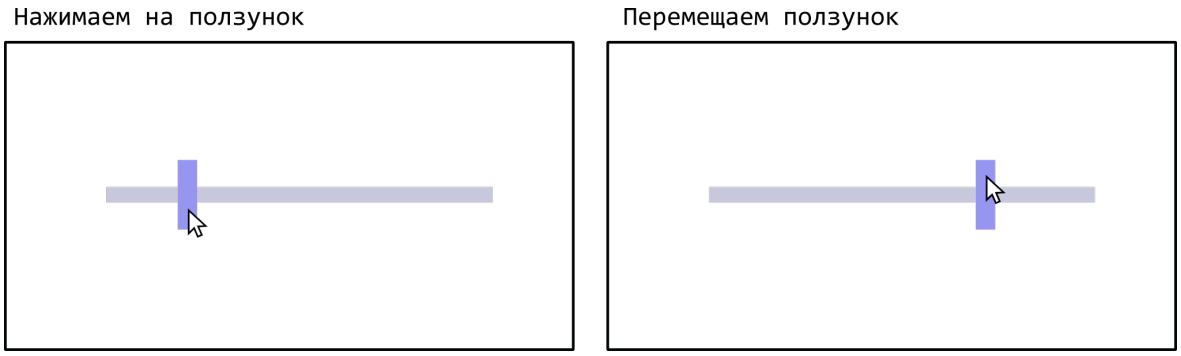



Для того, чтобы реализовать кнопку, делаем следующее. При событии нажатия на ЛКМ, если курсор находится внутри кнопки, то устанавливаем значение `mIsPressed = true`. При событии отпускания ЛКМ, если `mIsPressed == true` и курсор также находится внутри кнопки, то клик на кнопку произошёл, и мы устанавливаем `mIsPressed = false` и возвращаем `true` из метода `handleEvent`.

## Слайдер

Слайдер – это элемент графического интерфейса, который позволяет пользователю выбирать значение из определённого диапазона, перемещая ползунок вдоль горизонтальной или вертикальной шкалы. Компоненты слайдера:

- Ползунок (англ. *thumb*) – подвижный элемент, который пользователь перетаскивает для выбора значения.
- Шкала (англ. *track*) – линия или дорожка, вдоль которой перемещается ползунок.




Реализацию простейшего слайдера можно найти тут: `code/5gui/2slider`. Для того, чтобы реализовать слайдер, создадим класс `Slider`, состоящий из следующих полей:

- `sf::RectangleShape mTrackShape` – объект класса прямоугольника, который представляет шкалу прокрутки. Внутри хранятся положение, размеры и цвет шкалы прокрутки.
- `sf::RectangleShape mThumbShape` – объект класса прямоугольника, который представляет ползунок. Внутри хранятся положение, размеры и цвет ползунка.
- `bool mIsPressed` – Эта переменная равна `true`, когда ползунок находится в нажатом состоянии.
- `sf::RenderWindow& mRenderWindow` – также храним ссылку на окно SFML, на которое будем отрисовывать кнопку. Эту ссылку можно было бы не хранить, а просто передавать во все функции, где окно понадобится, но тогда код был бы более громоздким.

Также у класса будут следующие публичные методы:

- `void draw()` – рисует слайдер на окне. Ссылка на окно хранится внутри объекта `Slider`. Поэтому тут её передавать не нужно.
- `bool handleEvent(const sf::Event& event)` – обрабатывает всё, что касается слайдера.
- `float getValue()` – возращает значение положения слайдера от 0 до 100.

Для начала создадим вспомогательный метод, который бы устанавливал положение ползунка с учётом ограничений на его движение. Изменять положение ползунка будем только с использованием этого метода.

```
void setRestrictedThumbPosition(sf::Vector2f position)
{
    float min = mTrackShape.getPosition().x - mTrackShape.getSize().x / 2.0f;
    float max = mTrackShape.getPosition().x + mTrackShape.getSize().x / 2.0f;
    mThumbShape.setPosition({std::clamp(position.x, min, max), mThumbShape.getPosition().y});
}
```

Чтобы реализовать класс слайдера делаем следующее.

- При событии нажатия на левую кнопку мыши: если курсор мыши находится внути прямоугольника ползунка или внутри прямоугольника шкалы, то изменяем положение ползунка на положение курсора (используя `setRestrictedThumbPosition`) и устанавливаем `mIsPressed = true`.
- При событии движения курсора мыши: если `mIsPressed == true`, то изменяем положение ползунка на положение курсора (используя `setRestrictedThumbPosition`).
- При событии отпускания левой кнопки мыши: устанавливаем `mIsPressed = false`.