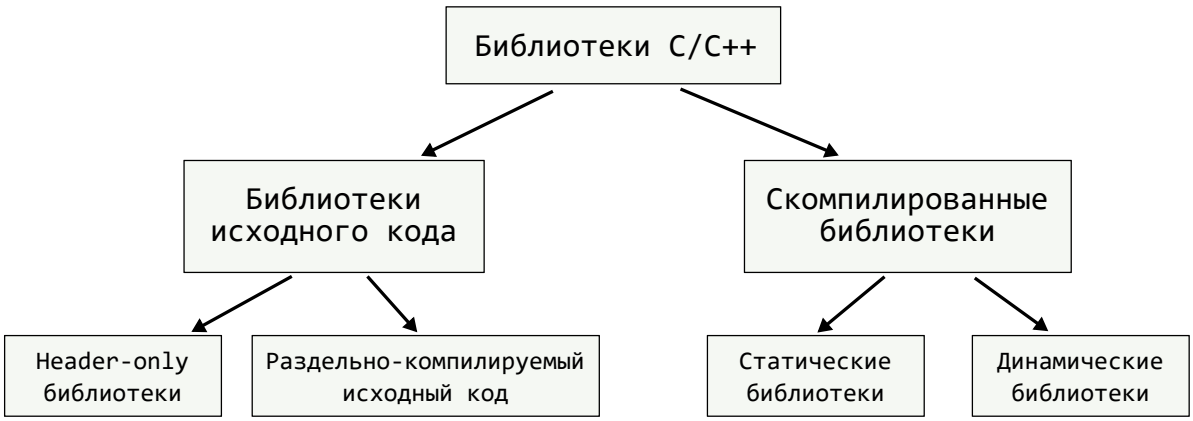
Семинар #1: Подключение библиотек. Библиотека raylib.

Часть 1: Библиотеки в языках C и C++

Библиотека в программировании – это набор готовых функций, классов или модулей(файлов), которые можно подключить к программе и использовать повторно для выполнения типовых задач.

В языках программирования C и C++ библиотеки можно разделить на следующие типы:



Разные типы библиотек создаются имеют свои особенности из создания и подключения. Для простоты будем предполагать, что мы используем язык C++ и компиляторы gcc и MinGW. Подключение библиотек на языке C и на других компиляторах во многом аналогично и отличается несущественно.

1. **Header-only библиотеки**
   Самый простой тип библиотек. Весь их код размещается в заголовочных файлах (`.h`/`.hpp`) и подключается с помощью директивы `#include`. Такие библиотеки легко использовать — как правило, достаточно добавить одну директиву `#include`. Их основной недостаток заключается в том, что код библиотеки компилируется отдельно для каждого `.cpp` файла, в который она подключена, что может увеличить время компиляции.

2. **Раздельно компилируемый исходный код**
   Библиотека представляет собой набор из заголовочных файлов(`.h`/`.hpp`) и файлов исходного кода(`.cpp`). Для подключения такой библиотеки вам нужно:

   - Добавить в код вашей программы необходимые заголовочные файлы, используя `#include`.
   - Указать компилятору пути до всех `.cpp` файлов библиотеки в дополнении ко всем файлам программы.

   Однако, такой способ подключения практически не используется, так как одна библиотека может содержать огромное количество `.cpp` файлов и подключение их всех может быть неудобным. Вместо этого, разработчики библиотеки предоставляют специальные сценарии (например, для программ `make` или `CMake`), используя которые вы можете скомпилировать библиотеку и превратить её в статическую или динамическую библиотеку.

3. **Статическая библиотека**
   Библиотека представляет собой набор из заголовочных файлов(`.h`/`.hpp`) и файлов скомпилированного кода статической библиотеки, которые имеют расширения:

   - `.a` – на Linux, macOS, Windows(компиляторы MinGW и Clang); `.a` от слова archive
   - `.lib` – на Windows(компилятор MSVC)

   Также, как правило, названия таких файлов начинаются с `lib`. К примеру, если у нас есть простая статическая библиотека по имени `dog`, то она будет состоять минимум из двух файлов:

   - `dog.hpp`
   - `libdog.a`

Для подключения статической библиотеки к вашей программе вам нужно:

- Добавить в код вашей программы необходимые заголовочные файлы, используя `#include`.
- Указать компилятору путь до директории, содержащей заголовочные файлы, используя опцию `-I`.
- Указать компилятору путь до директории, содержащей файлы скомпилированного кода, используя опцию `-L`.
- Указать компилятору имя конкретного файла скомпилированного кода, используя опцию `-l`. При этом указывать приставку `lib` и расширение файла не нужно.

К примеру, если вы хотите подключить к вашей программе, состоящей из одного файла `main.cpp`, простую статическую библиотеку `dog`, упомянутую выше, то вам нужно сделать следующее:

- Добавить в начало вашего кода строку:

  ```
  #include "dog.hpp"
  ```

- Скомпилировать программу со следующими опциями:

  ```
  $ g++ main.cpp -I <include directory> -L <lib directory> -l dog
  ```

  где
  - `<include directory>` – путь до директории, содержащей файл `dog.hpp`.
  - `<lib directory>` – путь до директории, содержащей файл `libdog.a`.

  При этом пробел между опцией и аргументом обычно опускают, то есть пишут `-ldog`, а не `-l dog`.

4. **Динамическая библиотека**
   Библиотека представляет собой набор из заголовочных файлов(`.h`/`.hpp`) и файлов скомпилированного кода динамической библиотеки, которые имеют расширения:

   - `.so` – на Linux; `.so` от shared object
   - `.dll` – на Windows; `.dll` от dynamic link library
   - `.dylib` – на macOS; `.dylib` от dynamic library

   Названия таких файлов на Linux имеет приставку `lib`, а на Windows и macOS такой приставки нет. Также на Windows вместе с динамической библиотекой `.dll` обычно поставляется небольшая библиотека `.a` или `.lib`, которая используется для подключения этой динамической библиотеки.

   К примеру, если у нас есть простая динамическая библиотека на Lunux по имени `dog`, то она будет состоять минимум из двух файлов:

   - `dog.hpp`
   - `libdog.so`

   А на Windows такая библиотека будет обычно состоять из трёх файлов:

   - `dog.hpp`
   - `dog.dll`
   - `libdog.a` или `libdog.lib`

   Подключить динамическую библиотеку к вашей программе можно таким же образом, что и статическую.

Отличие динамических библиотек от статических

Основное отличие динамической библиотеки от статической заключается в том, что статическая библиотека встраивается в исполняемый файл на этапе компиляции, а динамическая библиотека загружается и подключается во время запуска программы. Такой подход имеет следующие преимущества:

- Статическая библиотека встраивается в итоговый исполняемый файл и увеличивает его размер, динамическая библиотека подключается только при запуске и не увеличивает размер исполняемого файла.
- Одна и та же динамическая библиотека может быть загружена в память только один раз и использована сразу несколькими программами, что экономит оперативную память.
- Динамическую библиотеку можно загружать и выгружать во время работы программы.

Недостаток динамических библиотек заключается в том, что для запуска программы, использующей их, необходимые библиотеки должны быть заранее установлены в системе. При их отсутствии программа не сможет запуститься.

Часть 2: Создание и подключение библиотек

Рассмотрим создание простой собственной библиотеки, используя все четыре её типа. Пусть имеется следующая программа, содержащая функцию `cube`:

main.cpp

```cpp
#include <iostream>

int cube(int x)
{
    return x * x * x;
}

int main()
{
    std::cout << cube(5);
}
```

Вынесем функцию `cube` в отдельный файл и создадим простейшую библиотеку с одной функцией. Разумеется, на практике библиотеки состоят из множества файлов исходного кода, каждый из которых содержит множество функций. Здесь же мы создаём минимальный пример в учебных целях.

Создание и подключение header-only библиотеки

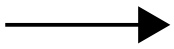
Создать такую библиотеку очень просто — достаточно вывести функцию в отдельный заголовочный файл.

main.cpp

```cpp
#include <iostream>

int cube(int x)
{
    return x * x * x;
}

int main()
{
    std::cout << cube(5);
}
```



main.cpp

```cpp
#include <iostream>
#include "cube.hpp"

int main()
{
    std::cout << cube(5);
}
```

cube.hpp

```cpp
#pragma once

static int cube(int x)
{
    return x * x * x;
}
```

После этого файл `cube.hpp` можно считать header-only библиотекой. Для подключения такой библиотеки достаточно просто добавить одну директиву `#include` в файл исходного кода, где эта библиотека требуется. Чтобы скомпилировать программу, нужно просто написать:

```
$ g++ main.cpp
```

Указывать файл `cube.hpp` для компилятора не нужно, так как он сам вставит этот файл внутрь файла `main.cpp` благодаря директиве `#include`.

Ключевое слово `static` используется для предотвращения ошибок компоновки при подключении заголовка в нескольких файлах. Если не указать `static` перед функцией, то в программе из нескольких файлов при подключении заголовка как минимум к двум `.cpp` файлам возникнет ошибка множественного определения функции. При использовании `static` функция будет видна только в пределах одного .cpp файла, поэтому каждый `.cpp` файл получит собственную реализацию `cube`, и ошибка не возникнет. Альтернативой является использование ключевого слова `inline`.

Создание и подключение библиотек раздельно-компилируемого кода

Используем раздельную компиляцию:

main.cpp

```cpp
#include <iostream>

int cube(int x)
{
    return x * x * x;
}

int main()
{
    std::cout << cube(5);
}
```

main.cpp

```cpp
#include <iostream>
#include "cube.hpp"

int main()
{
    std::cout << cube(5);
}
```

cube.hpp

```cpp
#pragma once
int cube(int);
```

cube.cpp

```cpp
#include "cube.hpp"
int cube(int x)
{
    return x * x * x;
}
```

Совокупность файлов `cube.hpp` и `cube.cpp` можно рассматривать как библиотеку. Для подключения такой библиотеки к файлу `main.cpp` нужно сделать следующее:

1. Подключить заголовочный файл `cube.hpp` с помощью `#include`.
2. Указать файл исходного кода `cube.cpp` при компиляции:

```
$ g++ main.cpp cube.cpp
```

Недостатки такого подхода проявляются, если библиотека содержит не один, а множество файлов исходного кода – иногда сотни или даже тысячи. В этом случае:

- Может показаться, что потребуется подключать много заголовочных файлов, однако этого можно избежать, создав один общий заголовок, который включает все необходимые файлы.
- При компиляции нужно будет указывать все `.cpp` файлы библиотеки.
- Библиотека будет компилироваться вместе с проектом, что может сильно замедлить время компиляции.

Создание статических библиотек

Для создания статической библиотеки, нужно сначала скомпилировать все `.cpp` в объектные файлы `.o`. В данном случае нужно скомпилировать только один файл `cube.cpp` в файл `cube.o` с помощью опции компилятора `-c`:

```
$ g++ -c cube.cpp
```

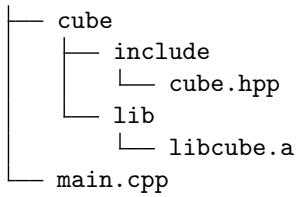
После этого нужно будет объединить все объектные файлы в один файл статической библиотеки с помощью специальной утилиты `ar`:

```
$ ar rcs libcube.a cube.o
```

Для объединения нескольких объектных файлов, нужно их всех указать в конце этой команды. В итоге получится файл статической библиотеки `libcube.a`.

Подключение статических библиотек

Предположим, что наш проект имеют следующую структуру:



Видно, что вся библиотека `cube` вынесена в отдельную папку с одноимённым названием. Внутри этой папки есть папка `include` для всех заголовочных файлов библиотеки и `lib` для всех файлов скомпилированного кода библиотеки. Чтобы скомпилировать файл `main.cpp`, подключив к нему библиотеку нужно выполнить команду:

```
$  g++ main.cpp -I./cube/include -L./cube/lib -lcube
```

Создание динамических библиотек на Linux

Для создания динамической библиотеки из файлов исходного кода на Linux нужно выполнить команду:

```
$ g++ -fPIC -shared -o libcube.so cube.cpp
```

Для объединения нескольких объектных файлов, нужно их всех указать в конце этой команды. В итоге получится файл динамической библиотеки `libcube.so`.

Подключение динамических библиотек на Linux

Подключение динамической библиотеки происходит в два этапа. Чтобы подключить такую библиотеку нужно:

1. Указать положение динамической библиотеки на этапе компиляции, используя опции `-I`, `-L` и `-l`, также как и для статической. Но, в отличии от статической библиотеки, динамическая на этом этапе не добавляется полностью в исполняемый файл. На данном этапе компилятор просто проверяет, что библиотека существует и является корректной (например, то что существуют все необходимые функции).

2. Указать положение динамической библиотеки при запуске программы. В Linux компилятор для поиска динамической библиотеки проверяет последовательно следующее:

   (a) Атрибут исполняемого файла `RPATH`

   (b) Переменную среды `LD_LIBRARY_PATH`

   (c) Атрибут исполняемого файла `RUNPATH`

   (d) `/etc/ld.so.cache` – это бинарный кэш-файл, содержащий предварительно вычисленный список динамических библиотек и путей к ним, созданный утилитой `ldconfig`.

   (e) Системные пути: `/lib`, `/usr/lib`, `/usr/local/lib`.

   Если при запуске программы динамическую библиотеку найти не удалось, то будет выведена ошибка вида:

   ```
   error while loading shared libraries: libcube.so: cannot open shared object file
   ```

   Решить подобную проблему можно одним из следующих способов:

   - Жёстко зашить путь до динамической библиотеки в исполняемом файле. Для этого нужно скомпилировать программу, используя следующую опцию:

     ```
     $ g++ main.cpp -L. -lmath -Wl,-rpath,'/path/to/dynlib'
     ```

   - Добавив путь до библиотеки в переменную `LD_LIBRARY_PATH`:

     ```
     $ LD_LIBRARY_PATH=/path/to/dynlib/:$LD_LIBRARY_PATH ./a.out
     ```

   - Переместив динамическую библиотеку в один из системных путей.

Помимо возможности подключения динамической библиотеки при запуске программы, есть возможность подключить библиотеку в произвольный момент выполнения программы, используя библиотеку `dlfcn.h`.

Создание динамических библиотек на Windows MinGW

В Windows при создании динамической библиотеки `cube.dll` нужно создать специальную импортную библиотеку `libcube.dll.a`, которая будет необходима при подключении динамической библиотеки. Сделать это можно с помощью следующей команды:

```
g++ -shared -o cube.dll -Wl,--out-implib,libcube.dll.a cube.cpp
```

Подключение динамических библиотек на Windows MinGW

1. Указать положение динамической библиотеки на этапе компиляции, используя опции `-I`, `-L` и `-l`, также как и для статической. Но, в отличии от статической библиотеки, динамическая на этом этапе не добавляется полностью в исполняемый файл. На данном этапе компилятор просто проверяет, что библиотека существует и является корректной (например, то что существуют все необходимые функции).

2. Указать положение динамической библиотеки при запуске программы. В Windows компилятор для поиска динамической библиотеки проверяет последовательно следующее:

   (a) Директория, в которой находится исполняемый файл

   (b) Системная директория `System32` (например, `C:\Windows\System32`)

   (c) Системная директория `Windows` (например, `C:\Windows`)

   (d) Текущая рабочая директория

   (e) Директории из переменной среды `PATH`

   (f) Директории, добавленные через `AddDllDirectory`, `SetDllDirectory` и `SetDefaultDllDirectories`.

   Если при запуске программы динамическую библиотеку найти не удалось, то будет выведена ошибка вида:

   ```
   error while loading shared libraries: cube.dll: cannot open shared object file
   ```

   Решить подобную проблему можно одним из следующих способов:

   - Добавить в переменную среды `PATH` путь до библиотеки.
   - Скопировать динамическую библиотеку в ту же директорию, где находится исполняемый файл или в текущую рабочую директорию.
   - Скопировать динамическую библиотеку в системные директории `Windows` или `System32`.

Часть 3: Основная информация о raylib

Часть 4: Подключение библиотеки raylib

Позволяет создавать окно, рисовать в 2D, проигрывать музыку и передавать информацию по сети.

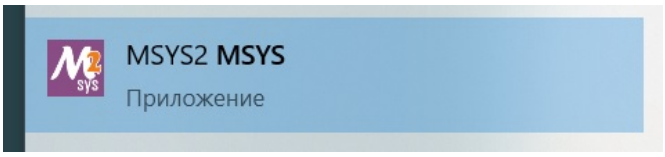
Подключение библиотеки на Windows с использованием пакетного менеджера MSYS2

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 raylib в среде MSYS2 проделайте следующее:

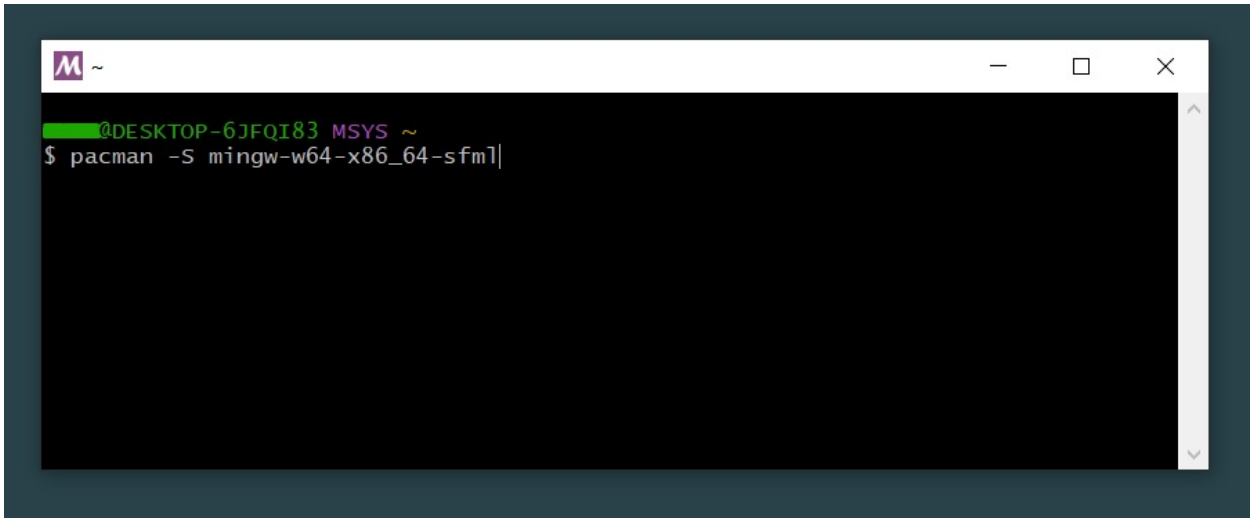
1. Найдите как называется пакет raylib в среде MSYS2. Для этого просто загуглите `msys2 install ucrt64 raylib` и одной из первых ссылок должен быть страница библиотеки SFML сайта `packages.msys2.org`. Зайдите на эту страницу и найдите команду для установки SFML. Скопируйте эту команду. Это может быть команда:

   ```
   pacman -S mingw-w64-ucrt-x86_64-raylib
   ```

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.

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

```
g++ main.cpp -lraylib
```

Подключение библиотеки на Linux с использованием пакетного менеджера

Тестирование библиотеки

Подключение вручную на Windows

Подключение вручную на Linux

Компиляция библиотеки

Часть 5: Примеры простых программ raylib

Структура программы

Простая программа, которая создаёт окно

```
#include <raylib.h>
```

Простая программа, которая рисует круг

Система координат

Простая программа, которая рисует движущийся круг

```
#include <raylib.h>
```

Ограничение количества кадров в секунду сверху

Вычисление длительности кадра

Часть 6: Основные структуры библиотеки raylib

Структуры математических векторов

Структура цвета

Структура прямоугольника

Часть 7: Рисование фигур

Рисование кругов

Рисование прямоугольников

Рисование линий

Рисование треугольников

Рисование кривых

Рисование произвольных многоугольников

Рисование более сложных фигур

Пример программы, которая рисует вращающийся текст

Часть 8: Рисование текста

Текст со стандартным шрифтом

Выбор шрифта для текста

Часть 9: Рисование изображений

Обработка нажатий клавиш

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:

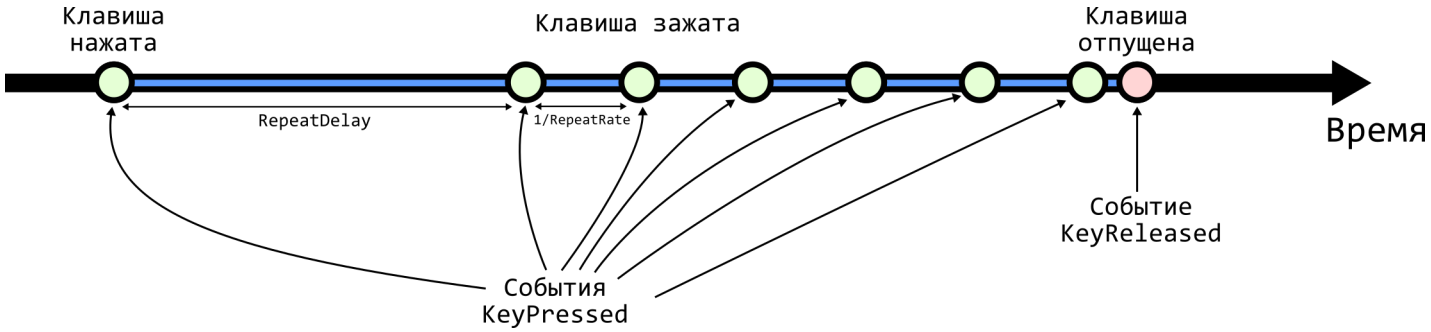
- Первый символ `A` добавиться сразу при нажатии.
- Второй символ `A` добавиться через некоторое относительно долгое время.
- Все последующие символы `A` будут добавляться с очень короткой задержкой между ними.

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:



Обработка взаимодействий с мышью

Проверка нажатия

Получение координат курсора мыши

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

Различие между двумя видами обработки нажатий клавиш и кнопок

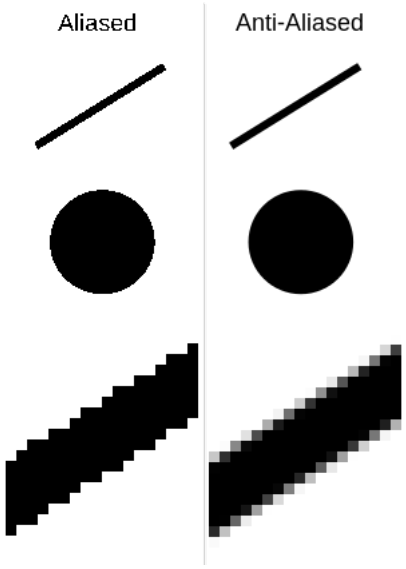
Конфигурация окна

Растягиваемое окно

Полноэкранный режим

Окно, растянутое на полный экран

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`.

Часть 10: Примеры программ с использованием библиотеки SFML

Рассмотрим примеры программ, рисующих движущиеся шарики в 2-х измерениях. Больше примеров можно найти в

Зацикленное движение шарика в 2d пространстве

Движение шарика в гравитационном поле с упругими столкновениями от границ

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 11: 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.

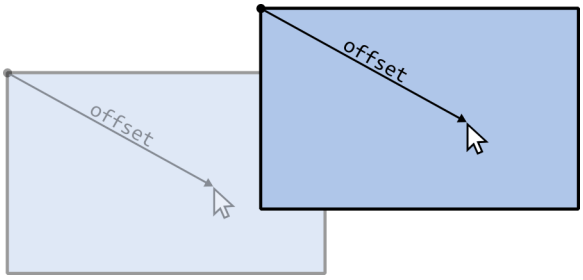
Реализация перетаскиваемой таблички без использования класса

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:

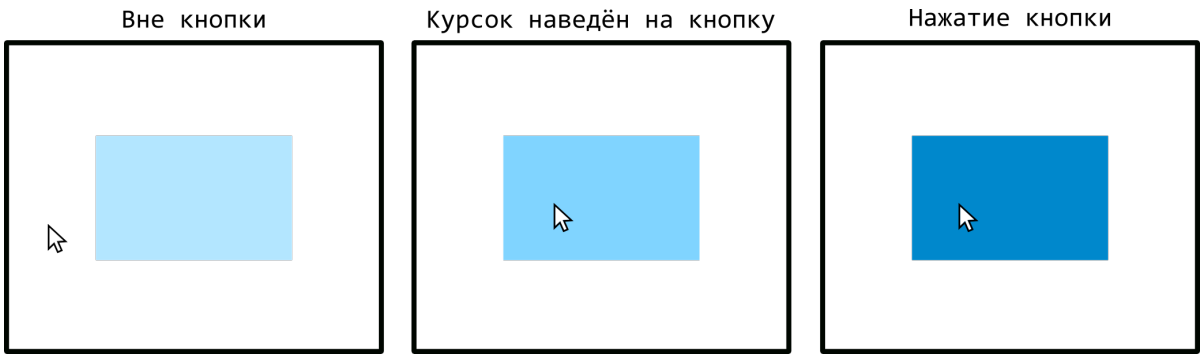
1. Кнопка представляет собой прямоугольник.
2. Изначально кнопка имеет некоторый заданный цвет.
3. При наведении курсора мыши на кнопку, её цвет меняется.
4. При нажатии и зажатии левой кнопки мыши (ЛКМ) над кнопкой, её цвет меняется на цвет, отличный от первых двух.
5. При отпускании ЛКМ, если курсор 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.

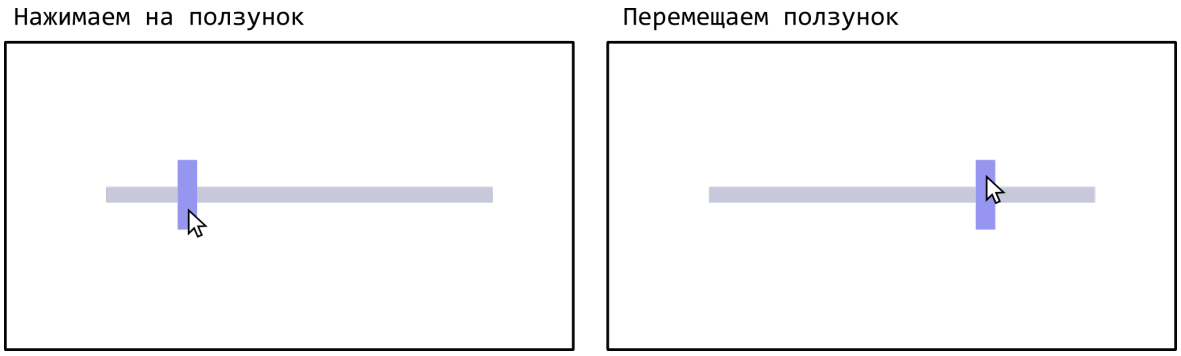


Для того, чтобы реализовать кнопку, делаем следующее. При событии нажатия на ЛКМ, если курсор находится внутри кнопки, то устанавливаем значение `mIsPressed = true`. При событии отпускания ЛКМ, если `mIsPressed == true` и курсор также находится внутри кнопки, то клик на кнопку произошёл, и мы устанавливаем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