

Компьютерная графика

I

Создание графического приложения с использованием RayLib

1 Создание первого приложения 3

- 1.1 Установка компиляторов и нужных библиотек 3
- 1.2 Структура проекта 5
- 1.3 Подготовка проекта 6
- 1.4 Добавление библиотек 7
- 1.5 Создание первого оконного приложения 9
- 1.6 Компиляция и запуск 12
- 1.7 Рисование графических элементов в оконном
приложении 13
- 1.8 Задание для самостоятельной работы 21
- 1.9 Контрольные вопросы 21
- 1.10 Загрузка проекта на портал course.sgu.ru 21

2 Простое рисование в окне 24

- 2.1 Предварительная подготовка 24
- 2.2 Определение рисунка 24
- 2.3 Изменение соотношения сторон рисунка 31
- 2.4 Задание для самостоятельной работы 33
- 2.5 Контрольные вопросы 33

3 Преобразования изображений 35

- 3.1 Предварительная подготовка 35
- 3.2 Предмет реализации 35
- 3.3 Геометрические преобразования 35
- 3.4 Внутренний формат представления объектов 44
- 3.5 Файловый ввод данных 47
- 3.6 Очистка проекта 55
- 3.7 Задание для самостоятельной работы 58
- 3.8 Контрольные вопросы 58

1 Создание первого приложения

1.1 Установка компиляторов и нужных библиотек

1.1.1 Linux (TODO)

Для разработки графических приложений нам понадобится достаточно много библиотек. Приведём команды для установки нужных пакетов для нескольких дистрибутивов.

ALT Linux:

```
1 apt-get install cmake make gcc-c++
2 apt-get install \
3     wayland-devel \
4     libwayland-client-devel \
5     libwayland-cursor-devel \
6     libwayland-egl-devel \
7     libxkbcommon-devel
8 apt-get install \
9     libX11-devel \
10    libXrandr-devel \
11    libXinerama-devel \
12    libXcursor-devel \
13    libXi-devel \
14    libgtk+3-devel
15 apt-get install libGL-devel
```

TODO: fedora, debian, arch



Нужно установить именно `*-devel` или `*-dev` пакеты, так как в них хранятся заголовочные файлы и символические ссылки на разделяемые библиотеки без указания версии в названии файла.

В Arch Linux нет такого деления на development пакеты. Если вы используете систему не перечисленную выше, то обратите внимание на то, какой подход к пакетированию используется в вашей системе.

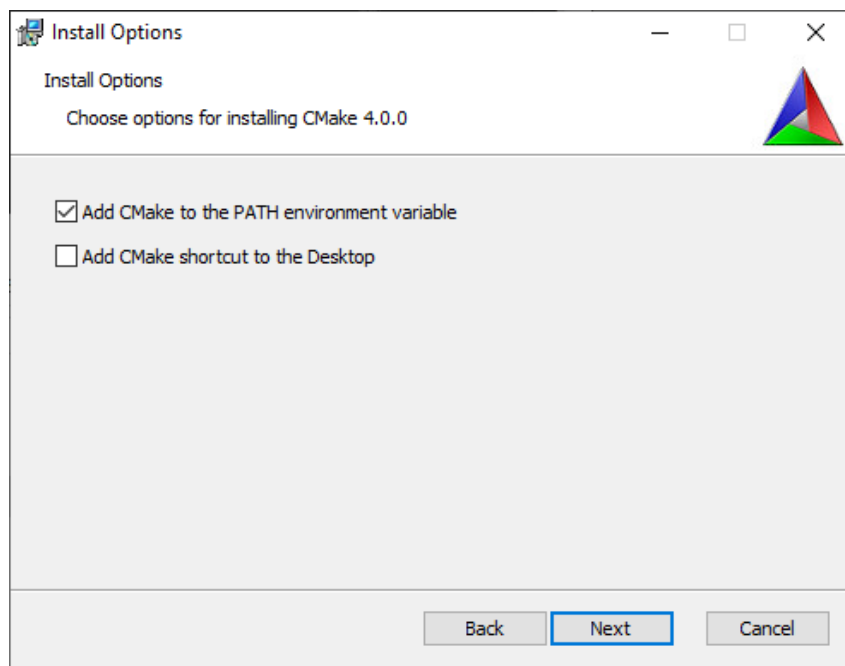
1.1.2 macOS (TODO)

1.1.3 Windows

На Windows, собирать этот проект также будем нативными средствами. В данном случае они отличаются от остальных UNIX систем.

CMake

CMake можно загрузить с сайта <https://cmake.org/download/>. Нас интересует Windows x64 Installer. При установке достаточно просто нажимать Next. Основной интересующий нас параметр при установке — это добавление CMake в переменную PATH. Этот параметр включён по умолчанию.



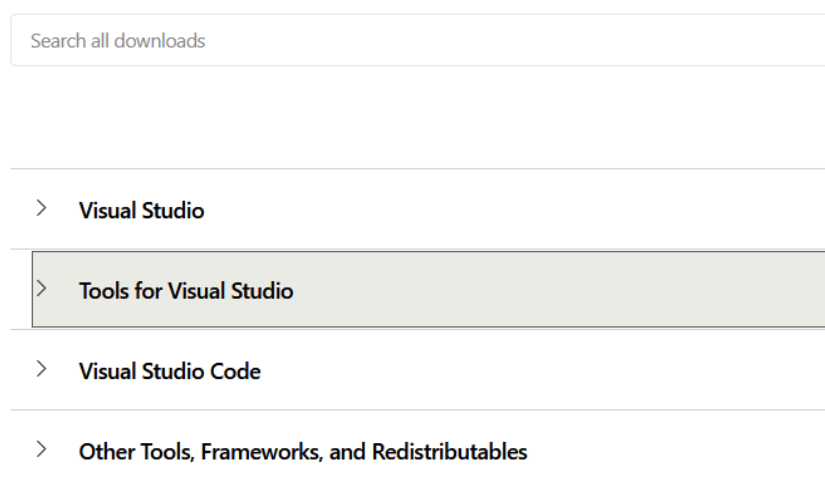
MSBuild

Для разработки проекта есть возможность использовать Visual Studio IDE, но в этом курсе такой подход не рекомендуется. Вместо IDE, можно установить standalone систему сборки Visual Studio проектов, а вести разработку в любом удобном редакторе.

Загрузить MSBuild можно со страницы <https://visualstudio.microsoft.com/downloads/>.

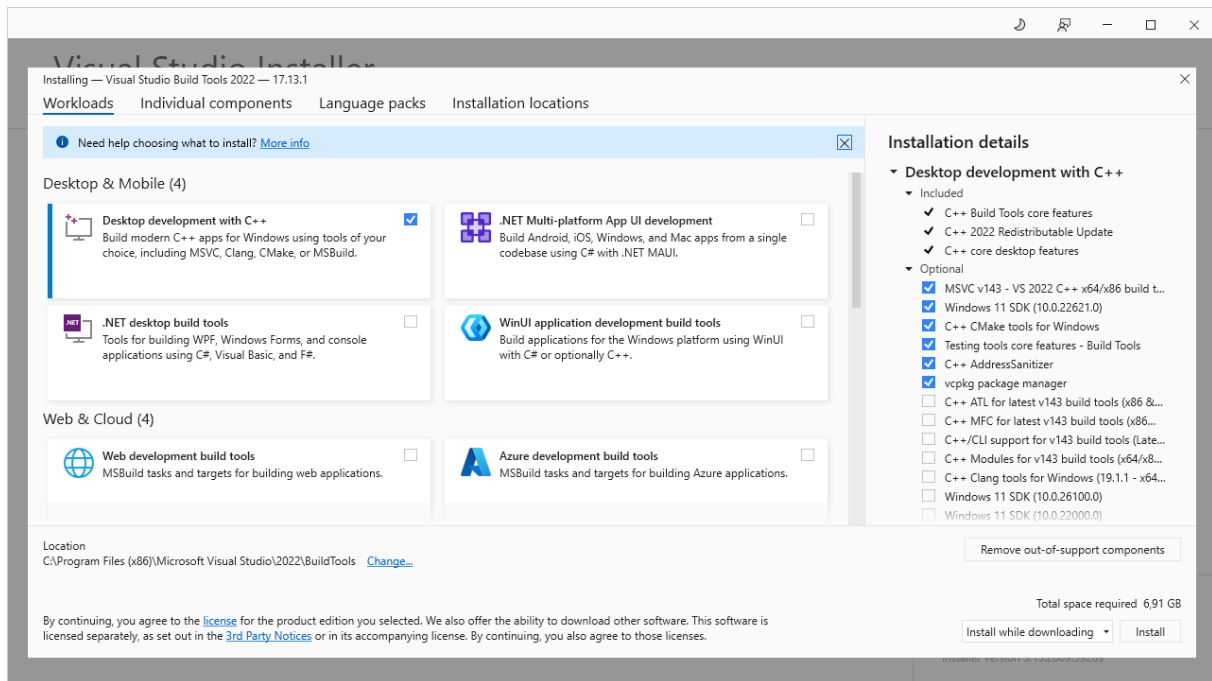
Нужно спуститься вниз страницы к разделу All Downloads и выбрать Tools for Visual Studio.

All Downloads



В появившемся меню нужно загрузить Build Tools for Visual Studio.

После завершения загрузки, нужно запустить установщик и выбрать в нём Desktop development with C++ и нажать Install.



1.2 Структура проекта

Первым делом необходимо подготовить проект, с помощью которого будут собираться все задания указанные в этом пособии. Для того, чтобы студенты и преподаватели могли собирать проекты на удобных им системам, будет использоваться CMake.

Помимо этого, нам понадобятся кросс-платформенные библиотеки для того, чтобы можно было писать одинаковый код для всех поддерживаемых систем.

Для выполнения заданий понадобятся следующие библиотеки:

- Native File Dialog (<https://github.com/mlabbe/nativefiledialog>) — для возможности открывать окно выбора файла нативно для всех систем;
- RayLib (<https://www.raylib.com/>) — для упрощения отрисовки примитивов, текста и т. д.;
- raygui (<https://github.com/raysan5/raygui>) — для отрисовки простых элементов графического интерфейса.

Структура проекта будет примерно следующей:

- 1 IvanovGraphics
- 2 └─ CMakeLists.txt

```

3 |— Libraries
4 |   |— nfd
5 |   |   |— CMakeLists.txt
6 |   |— raylib
7 |   |   |— CMakeLists.txt
8 |   |— raygui
9 |   |   |— CMakeLists.txt
10 |— Projects
11 |   |— lab1
12 |   |   |— CMakeLists.txt
13 |   |— lab2
14 |   |   |— CMakeLists.txt
15 |   ...
16 |   |— labN
17 |       |— CMakeLists.txt

```

1.3 Подготовка проекта

Основным файлом, устанавливающим структуру нашего проекта будет корневой файл `CMakeLists.txt`. Рассмотрим его по частям.

В первую очередь установим минимальную версию CMake, с помощью которой можно собрать этот проект. В данном случае нам подойдет CMake начиная с версии 3.5. Помимо этого, установим название проекта. Название проекта должно соответствовать формату `{фамилия транслитом}-graphics`.

```

1 cmake_minimum_required(VERSION 3.5)
2 project(ivanov-graphics)

```

Далее установим некоторые флаги сборки проекта. Зачем нужны эти флаги описано в комментариях к ним.

```

1 # Будем писать проект со свежей версией стандарта C++
2 set(CMAKE_CXX_STANDARD 20)
3 # Для лучшей переносимости между системами,
4 # статически вкомпилируем все зависимости
5 set(BUILD_SHARED_LIBS OFF)
6 # Для редакторов использующих clangd для автополнения
7 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
8 # Сохраним путь до корня проекта. Это важно для
9 # корректной генерации проекта для Visual Studio.
10 set(SOLUTION_ROOT ${CMAKE_CURRENT_LIST_DIR})
11 # Флаги, включающие дополнительный вывод предупреждений при сборке
12 if(CMAKE_COMPILER_IS_GNUCXX AND NOT WIN32)
13     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -pedantic")
14 endif()
15 endif()

```

Далее добавим все используемые библиотеки как поддиректории. Таким образом, при сборке проекта, CMake распознает их `CMakeLists.txt` файлы как подпроекты.

```

1 add_subdirectory(Libraries/nfd)
2 add_subdirectory(Libraries/raylib)
3 add_subdirectory(Libraries/raygui)

```

Так как количество библиотек не будет изменяться при выполнении лабораторных, мы можем их перечислить сразу и вручную. Количество же выполненных лабораторных будет со временем увеличиваться, поэтому добавим их не вручную, а с помощью цикла по каталогу проектов.

```

1 set/projects_dir ${CMAKE_CURRENT_LIST_DIR}/Projects)
2 file(GLOB children RELATIVE ${projects_dir} ${projects_dir}/*)
3 foreach(child ${children})
4     if(IS_DIRECTORY ${projects_dir}/${child})
5         add_subdirectory(${projects_dir}/${child})
6     endif()
7 endforeach()

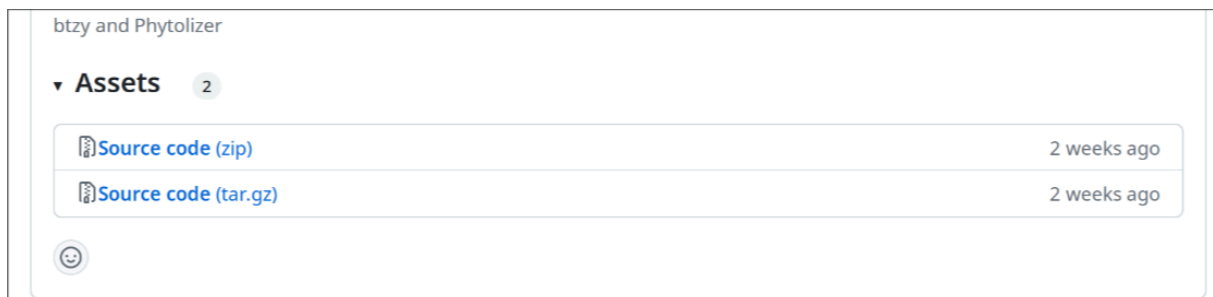
```

1.4 Добавление библиотек

Большинство библиотек добавляются в проект просто — просто добавлением каталога с их проектом в наш каталог Libraries и переименованием. Но для некоторых библиотек придётся создать проект CMake вручную.

1.4.1 Native File Dialog Extended

Со страницы с релизами <https://github.com/btzy/nativefiledialog-extended/releases> загрузим последний в виде архива с исходным кодом.













После распаковки получим каталог `nativefiledialog-extended-{версия}`. Переименуем его в `nfd` и положим в каталог `Libraries` внутри нашего проекта.

1.4.2 RayLib

Со страницы с релизами <https://github.com/raysan5/raylib/releases/> загрузим последний в виде архива с исходным кодом (архив `Source code.zip`).

▼ Assets 10

 raylib-5.5_linux_amd64.tar.gz	1.7 MB	Nov 18, 2024
 raylib-5.5_linux_i386.tar.gz	1020 KB	Nov 18, 2024
 raylib-5.5_macos.tar.gz	3.24 MB	Nov 18, 2024
 raylib-5.5_webassembly.zip	573 KB	Nov 18, 2024
 raylib-5.5_win32_mingw-w64.zip	796 KB	Nov 18, 2024
 raylib-5.5_win32_msvc16.zip	3.5 MB	Nov 18, 2024
 raylib-5.5_win64_mingw-w64.zip	1.61 MB	Nov 18, 2024
 raylib-5.5_win64_msvc16.zip	2.41 MB	Nov 18, 2024
 Source code (zip)		Nov 18, 2024
 Source code (tar.gz)		Nov 18, 2024

После распаковки получим каталог `raylib-x.y`. Переименуем его в `raylib` и положим в каталог `Libraries` внутри нашего проекта.

1.4.3 raygui

Библиотека `raygui` не поставляется с `CMakeFiles.txt` проектом, поэтому создадим его самостоятельно.

Для этого точно так же загрузим последний релиз из <https://github.com/raysan5/raygui/releases> и из всех файлов, которые находятся в этом архиве возьмём только `raygui.h`.

В директории `Libraries` создадим каталог `raygui`, в который добавим файл `CMakeLists.txt` со следующим содержанием:

```

1 cmake_minimum_required(VERSION 3.5)
2 project (raygui C)
3
4 # Disable all warnings
5 if (MSVC)
6     add_compile_options(/w)
7 else()
8     add_compile_options(-w)
9 endif()
10
11 add_library (raygui STATIC src/raygui.c)
12 target_include_directories (raygui PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)
13 target_link_libraries (raygui PUBLIC raylib)

```

В нём можно заметить упоминания каталога `include/` и файла `src/raygui.c`. Создадим директорию `include` и положим в неё скачанный файл `raygui.h`. После этого создадим директорию `src` и добавим в неё файл `raygui.c`:

```

1 #define RAYGUI_IMPLEMENTATION
2 #include <raygui.h>

```




raygui является так называемой single-header библиотекой. То есть для её использования в любом проекте можно просто добавить файл `raygui.h` и включать его в любые необходимые файлы `.c/.cpp`.

Но без включённого макроса `RAYGUI_IMPLEMENTATION` в эти файлы будут включаться только определения функций, без их реализаций. При компоновке приложения возникнут ошибки отсутствующих символов.

Использование таких библиотек в многофайловых проектах не совсем тривиально, поэтому скомпилируем её как отдельную статическую библиотеку. Таким образом мы точно не получим ни ошибок об отсутствующих символах, ни о появляющихся дважды.

1.5 Создание первого оконного приложения

После того, как мы добавили все библиотеки и создали основной файл проекта `CMakeLists.txt`, нужно создать наш собственный подпроект. Для этого в директории `Projects` создадим директорию нашего первого проекта — `lab1`. В ней нам понадобится исполняемый файл `main.cpp` и файл проекта `CMakeLists.txt`.

```
1 IvanovGraphics
2 |— CMakeLists.txt
3 |— Libraries/
4 |— Projects
5   |— lab1
6     |— main.cpp
7     |— CMakeLists.txt
```

1.5.1 Файл проекта

Так как нам нужно собрать отдельный исполняемый файл для этого подпроекта, нам нужно описать этот подпроект также и для CMake в файле `CMakeLists.txt`.

Сначала напомним автоматизированное вычисление названия проекта. Это полезно для того, чтобы можно было просто скопировать без изменений каталог с предыдущей лабораторной для создания новой. Таким образом, при регенерации проекта всё равно создастся новый подпроект с корректным названием.

```
1 get_filename_component(ProjectId ${CMAKE_CURRENT_LIST_DIR} NAME)
2 string(REPLACE " " "_" ProjectId ${ProjectId})
3 project(${ProjectId} C CXX)
```

Далее добавим автоматический выбор всех `.hpp` и `.cpp` файлов в каталоге лабораторной и добавим их к текущему проекту.

```
1 file(GLOB_RECURSE HEADERS "[^.]*.hpp")
2 file(GLOB_RECURSE SOURCES "[^.]*.cpp")
3 add_executable(${PROJECT_NAME} ${SOURCES} ${HEADERS})
```



Регулярное выражение для выбора файлов начинается с `[^.]`. Это означает, что будут выбраны все файлы `.hpp/.cpp`, не начинающиеся с точки. Некоторые редакторы во время изменения файлов создают такие скрытые файлы и это может привести к ошибкам.



В промышленной разработке, обычно, не используют глобы для выбора всех файлов проектов, а вместо этого перечисляют все файлы по отдельности. Это также позволяет избавиться от ошибок.

В нашем учебном проекте нет такой необходимости, поэтому настроим максимальную автоматизацию генерации проекта.

Прикомпонуем библиотеки, которые будут использоваться повсеместно в лабораторных.

```
1 target_link_libraries(${PROJECT_NAME} LINK_PRIVATE nfd)
2 target_link_libraries(${PROJECT_NAME} LINK_PRIVATE raygui)
3 target_link_libraries(${PROJECT_NAME} LINK_PRIVATE raylib)
```

В случае, если для запуска проектов будет использоваться Visual Studio, то для корректной подгрузки файлов из каталога `Assets` необходимо установить в эту переменную корень проекта.

```
1 if(WIN32)
2     set_target_properties(${PROJECT_NAME}
3         PROPERTIES VS_DEBUGGER_WORKING_DIRECTORY ${SOLUTION_ROOT})
4 endif()
```

В качестве названия получаемого исполняемого файла установим название текущего проекта:

```
1 set_target_properties(${PROJECT_NAME} PROPERTIES OUTPUT_NAME ${PROJECT_NAME})
```

1.5.2 Код для создания окна

Чтобы созданный вами проект начал функционировать, пора написать код самой программы.

Создадим файл `main.cpp`, в котором будет находиться код нашего проекта. В библиотеке RayLib используется обычная функция `main()`, как и в консольных приложениях.



В системах Windows для графических приложений зачастую используется функция `WinMain()`. В неё передаются низкоуровневые примитивы WinAPI для создания окон. Реализуя данную функцию вместо `main()` также не запускается консоль при запуске графического приложения.

Данный подход в этом курсе не рассматривается.

В своём текстовом редакторе или IDE откройте файл `main.cpp` и добавьте в него следующий код.



Здесь и далее постарайтесь не копировать код из методички, а писать его вручную: среда программирования поможет вам с помощью системы авто-дополнения. Если вы освоитесь с ней, то это позволит избежать ошибок в дальнейшем.

```
1 #include <raylib.h>
2
3 int main() {
4     SetConfigFlags(FLAG_WINDOW_RESIZABLE);
5     InitWindow(600, 480, "Lab Uno");
6     SetTargetFPS(60);
7
8     while (!WindowShouldClose()) {
9         BeginDrawing();
10        EndDrawing();
11    }
12    CloseWindow();
13
14    return 0;
15 }
```

Так как RayLib — библиотека на языке C, в ней используются только обычные функции, без классов. По этой причине для изменения параметров окна, нужно изменять глобальное состояние программы.

Для изменения параметров создаваемого окна используется процедура `SetConfigFlags`. Чтобы мы имели возможность изменять размер окна, передадим флаг `FLAG_WINDOW_RESIZABLE`. Другие флаги, которые можно передать в `SetConfigFlags` можно посмотреть в заголовочном файле `raylib.h`.



RayLib — достаточно простая и хорошо написанная, с точки зрения качества кода, библиотека. Любые интересующие вас функции и параметры можно посмотреть в заголовочном файле `raylib.h`. Рядом со всеми функциями есть комментарий с кратким описанием того, что она делает.

Помимо самого файла, так же есть «шпаргалка» с той же самой информацией на странице <https://www.raylib.com/cheatsheet/cheatsheet.html>

Инициализировать окно можно с помощью процедуры `InitWindow`, в которую передаются ширина, высота и строка с названием окна. Для удобства работы, будем ограничивать количество кадров в секунду во всех библиотеках числом 60.

Далее следует цикл работы приложения, который будет работать до тех пор, пока не закроется окно. Факт закрытия окна можно узнать с помощью функции `WindowShouldClose()`. Внутри цикла производится отрисовка графики, обработка

событий (нажатие на кнопки и т. д.), а также вычисления, которые должны производиться каждый кадр работы окна.



Вычисления, которые достаточно производить только один раз нужно выполнить до начала цикла для экономии ресурсов и избежания ошибок. Например, не стоит загружать изображения для отрисовки внутри цикла, так как это может привести к утечкам памяти.

Но если изображение загружается по какому-то событию (например, открытию диалогового окна) и добавляется в массив с загруженными изображениями, который находится за пределами цикла, то никаких утечек не произойдёт.

В связи с тем, как устроен рендеринг графики в низкоуровневых библиотеках, для более эффективной работы в RayLib вся отрисовка должна происходить между вызовами процедур `BeginDrawing()` и `EndDrawing()`.

Для того, чтобы освободить все ресурсы, которые выделил RayLib при создании нашего окна необходимо вызвать процедуру `CloseWindow()` после того, как цикл завершит свою работу.

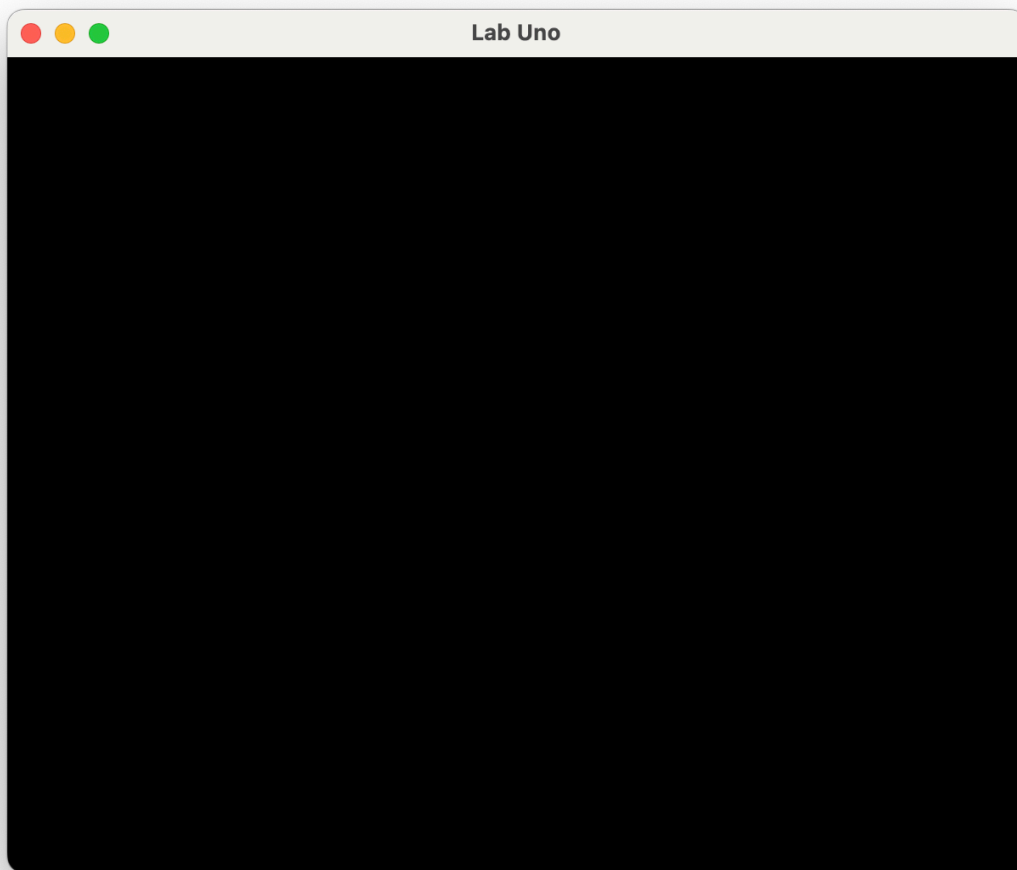
1.6 Компиляция и запуск

При условии того, что все необходимые пакеты были установлены, достаточно выполнить две команды.

```
1 cmake -B Build
2 cmake --build Build --parallel
```

В результате, CMake сгенерирует файлы для сборки с помощью утилиты Make и после этого запустит сборку проекта используя все доступные ресурсы системы.

Теперь запустить полученный исполняемый файл `Build/Projects/lab1/lab1`. Вы должны увидеть вот такое окно, заполненное чёрным цветом.



TODO: путь до бинарника на Windows

При запуске этих программ проект собрался в режиме Debug. Это полезно для отладки, но при этом в программе очень много лишнего и она работает несколько медленнее из-за отключенных оптимизаций. Собрать в режиме Release можно с помощью следующих команд.

```
1 cmake -B Build -D CMAKE_BUILD_TYPE=Release
2 cmake --build Build --parallel
```

1.7 Рисование графических элементов в оконном приложении

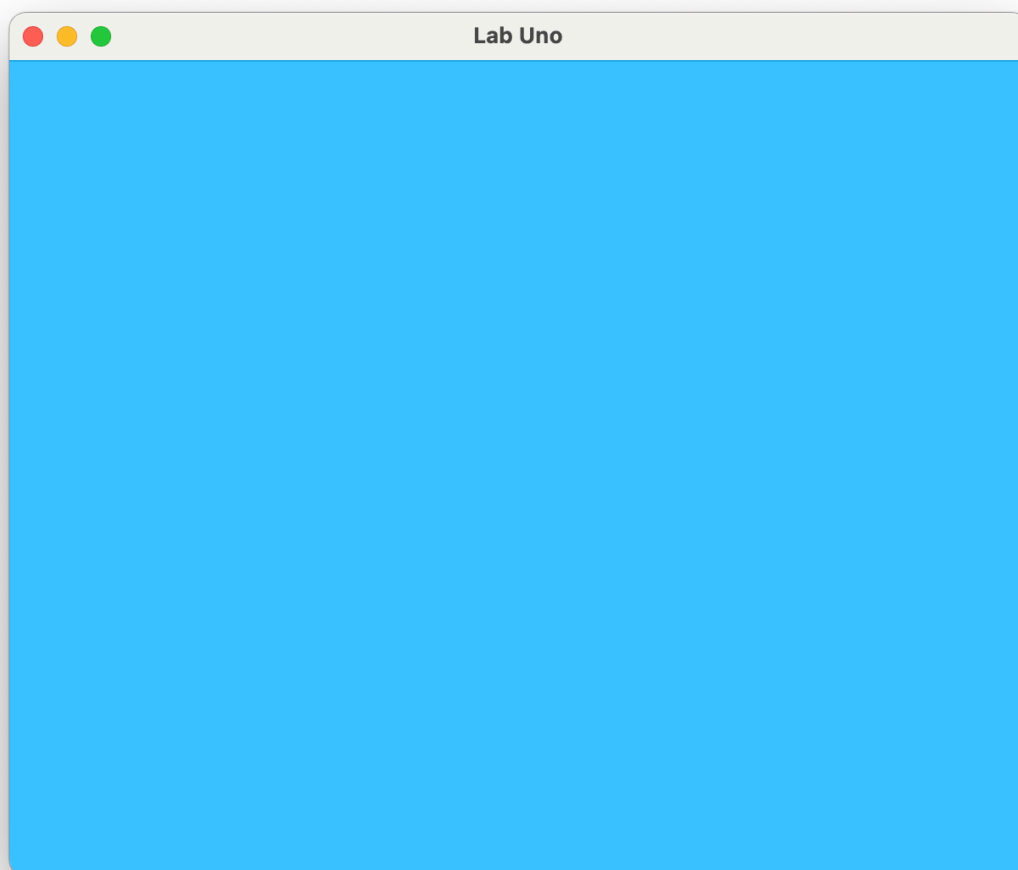
1.7.1 Закраска области рисования

Для закрашки всего окна одним цветом, можно вызвать процедуру `ClearBackground()`, в которую передаётся структура `Color { r, g, b, a }`. В самой библиотеке есть некоторые предопределённые цвета, которые можно использовать по названию, вместо создания структуры `Color` с указанными компонентами RGBA. Здесь укажем цвет `SKYBLUE`.



Структура `Color` содержит цвет именно в формате RGBA (то есть с альфа-каналом). Если создать новый цвет выражением `Color { 255, 0, 0 }`, то у нас получится красный цвет с нулевым альфа-каналом, то есть полностью прозрачный. Если забыть об этом, то может возникнуть ситуация, что команды отрисовки сработали, но в окне ничего не появилось.

Можно перекомпилировать и запустить проект. В результате окно должно заполниться ярко-голубым цветом неба.



1.7.2 Вычерчивание линий

Для рисования линий в RayLib используется группа процедур `DrawLine*()`. Входные параметры всех этих процедур можете изучить сами и подобрать ту, которая подходит Вам больше всего в конкретном случае.

В методичке в дальнейших заданиях мы будем использовать только процедуру `DrawLineEx(start, end, thickness, color)` как наиболее полную из доступных.

Сначала начертим толстую красную линию, соединяющую верхний левый и правый нижний угол.

Для того, чтобы начертить отрезок, нужно знать координаты начала и конца отрезка. Система координат в окне — левосторонняя. Начало координат находится в верхнем левом углу окна, так что с точкой начала отрезка все ясно.

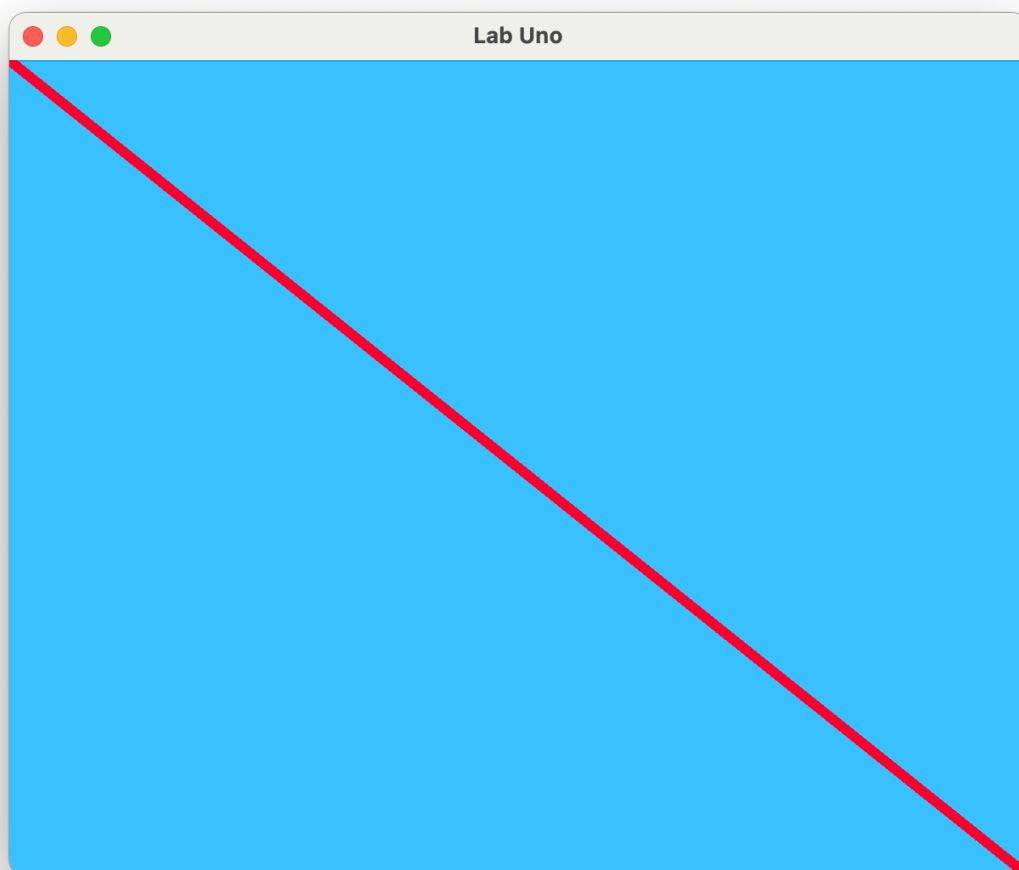
Чтобы узнать координаты правого нижнего угла, следует выяснить текущие размеры нашего окна. Делается это вызовом функций `GetScreenWidth()` и `GetScreenHeight()`. Эти значения целочисленные, но так как далее они будут передаваться в конструкторы структур точек, их необходимо привести к типу `float`.

```
1 const float Wx = static_cast<float>(GetScreenWidth());
2 const float Wy = static_cast<float>(GetScreenHeight());
```

Теперь можем нарисовать красную линию толщиной 6 пикселей из одного угла окна в другое.

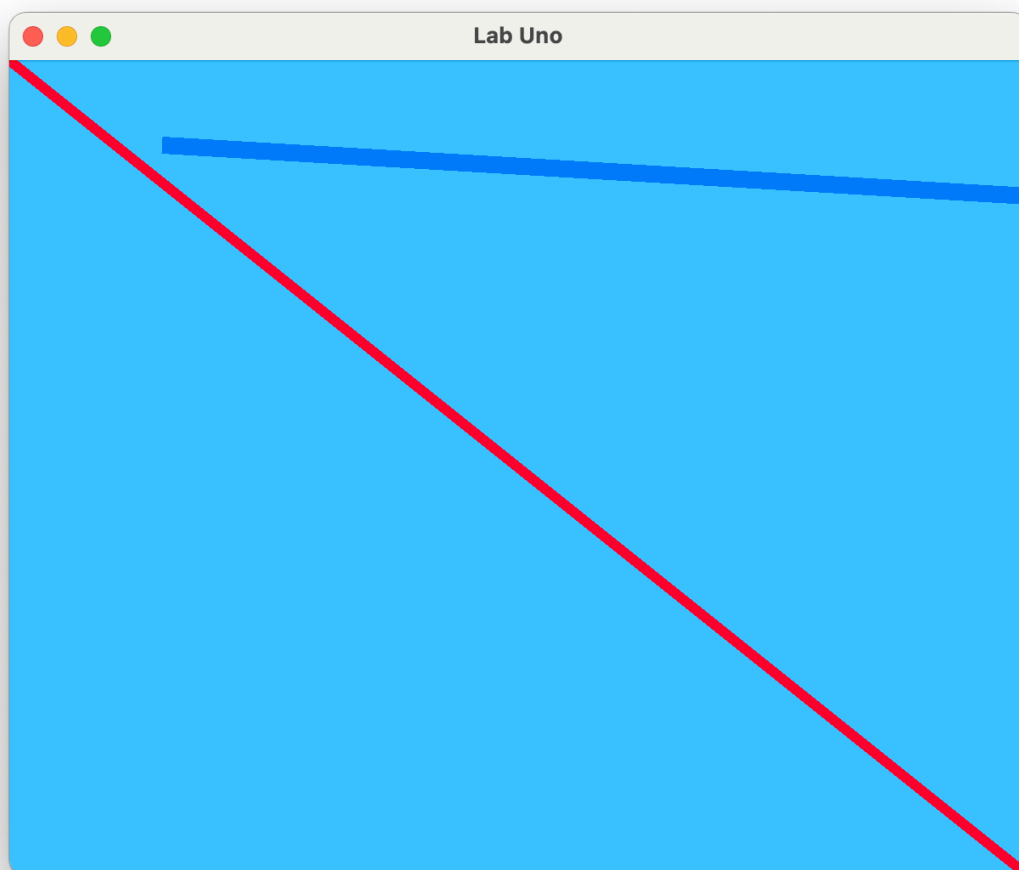
```
1 DrawLineEx({0, 0}, {Wx, Wy}, 6, RED);
```

После запуска появится окно с нарисованной линией. Так как мы каждый кадр вычисляем размеры окна и относительно них рисуем линию, при изменении размера, линия всё ещё будет корректно рисоваться из одного угла в другой.



Нарисуем ещё одну линию синего цвета ещё большей толщины из точки (90, 50) до точки, касающейся правой стороны окна, на высоте 80.

```
1 DrawLineEx({90, 50}, {Wx, 80}, 10, BLUE);
```

1.7.3 Вывод текста

Текст можно вывести с помощью процедуры `DrawText`. Изучим её сигнатуру

```
1 void DrawText(  
2     const char *text,  
3     int posX, int posY,  
4     int fontSize,  
5     Color color  
6 )
```

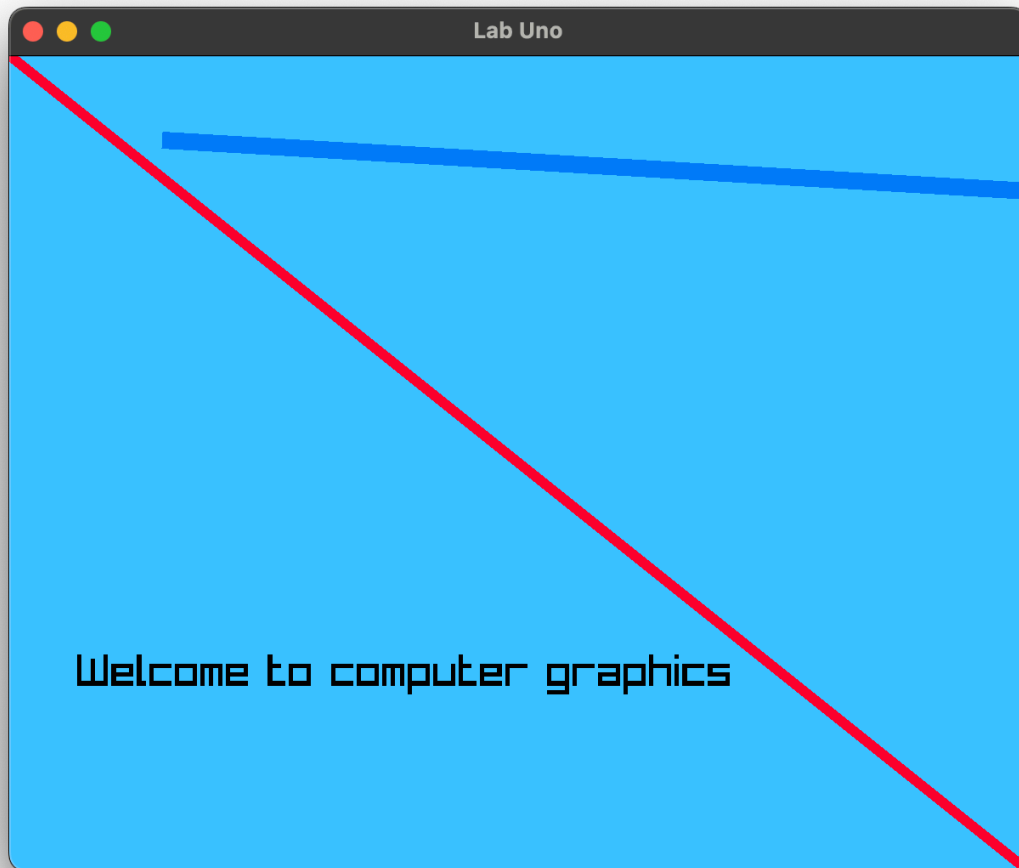
Первым параметром передаётся текст, который будет нарисован в окне. Следует заметить, что будет использован встроенный bitmap шрифт, поддерживающий только ASCII символы.

Далее указываем координаты верхнего левого угла прямоугольника с текстом, размер текста в пикселях и цвет текста.

Выведем надпись «Welcome to computer graphics» чёрного цвета и размера 26 пикселей в координате (40, 350):

```
1 DrawText("Welcome to computer graphics", 40, 350, 26, BLACK);
```

Увидим следующую картину:



Ограничение только ASCII символами в нашем случае очень не удобно, поэтому используем более продвинутые возможности работы с текстом в RayLib.

Для этого воспользуемся следующей функцией загрузки шрифта:

```
1 LoadFontEx(  
2     const char *fileName,  
3     int fontSize,  
4     int *codepoints,  
5     int codepointCount  
6 )
```

В ней мы должны указать путь до файла со шрифтом, размер шрифта при загрузке и некоторые `codepoints` и их количество.



Так как с помощью одного байта можно зашифровать только 256 символов и самой распространённой однобайтовой кодировкой является ASCII, другим языкам приходится кодировать свои символы несколькими байтами. На смену ASCII пришёл стандарт UTF-8, имеющий наиболее широкую поддержку.

Символы кириллицы не находятся в ASCII, они кодируются двумя байтами. Так как один байт кириллического символа не имеет логического смысла, стоит рассматривать символы только как группы байт. Такая группа и называется `codepoint`.

По умолчанию, RayLib загрузит из шрифта только `codepoints`, соответствующие ASCII символам. По этой причине нам не подходит функция `LoadFont`, которая принимает только путь до шрифта — она не загрузит все нужные нам символы.

Создадим константу с перечислением всех необходимых нам символов в глобальном пространстве нашей программы.

```
1 const char *const LETTERS =  
2     "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ" \br/>3     "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~" \br/>4     "абвгдеёжзийклмнопрстуфхцчщъыьэюяАБВГДЕЁЖИЙКЛМНОПРСТУФХЦЧЩЪЫЬЭЮЯ";
```



Заметим, что для создания глобальной строковой константы стоит указывать не только константное содержимое, но и константное значение самого адреса указателя.

Загрузим перед началом цикла отрисовки новый шрифт. Сначала разделим список наших символов на UTF-8 `codepoints` с помощью функции `LoadCodepoints`. При разделении на `codepoints`, RayLib сам посчитает их итоговое количество. Оно понадобится нам далее.

```
1 int cnt = 0;  
2 int *codepoints = LoadCodepoints(LETTERS, &cnt);
```

Теперь можно загрузить шрифт Roboto из директории `Assets/Fonts`. Укажем начальный размер шрифта в 100 пикселей. При отрисовке самих строк, размер шрифта строки будет уменьшен до нужного. Так как мы не будем рисовать текст большего размера, нам достаточно 100 пикселей.

После этого передаём полученные данные о `codepoints` и получаем на выходе нужный нам шрифт, который может нарисовать все нужные нам символы.

```
1 Font f = LoadFontEx("Assets/Fonts/Roboto-Regular.ttf", 100, codepoints, cnt);
```

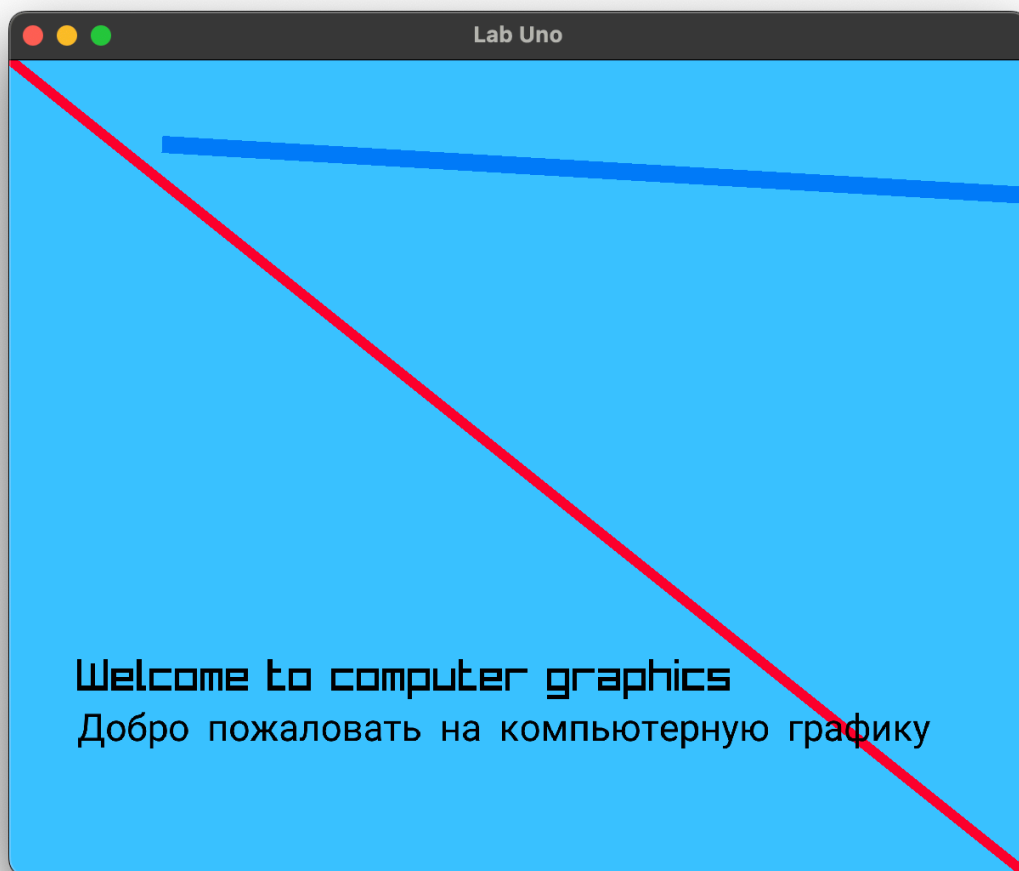
В цикле отрисовки добавим чуть ниже первой строки, другую строку на русском языке. Для этого понадобится использовать процедуру `DrawTextEx`, которая помимо

стандартных параметров принимает также сам шрифт и расстояние между символами в строке.

Укажем ранее загруженный шрифт и расстояние между символами равное 0. Помимо этого, параметр координаты должен передаваться в виде структуры координаты, а не отдельными параметрами.

```
1 DrawTextEx(f, "Добро пожаловать на компьютерную графику", {40, 380}, 26, 0, BLACK);
```

Получим следующую картинку.



Далее для отрисовки текста будем использовать только процедуру `DrawTextEx`.

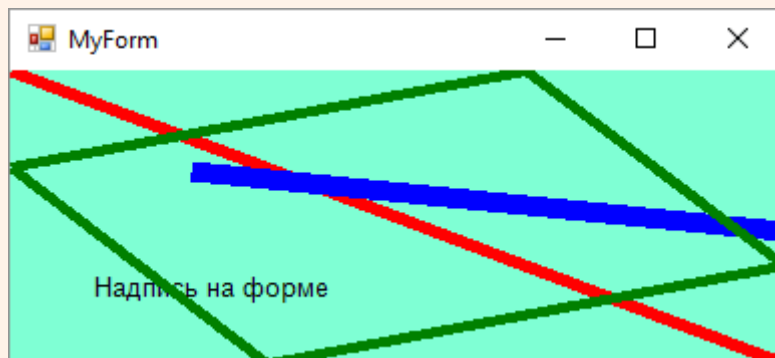


Указанный шрифт можно загрузить на сайте Google Fonts: <https://fonts.google.com/specimen/Roboto> Этот шрифт является открытым и свободным, поэтому его можно использовать без ограничений.

1.8 Задание для самостоятельной работы

Задание 1

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Название проекта должно соответствовать формату {фамилия транслитом}-graphics.
2. Добавьте в получившийся проект следующее дополнение. Соедините точки на сторонах окна, делящие каждую сторону в соотношении 2:1, линиями зелёного цвета толщины 5 так, как показано на рисунке.



TODO

3. Архив получившегося проекта загрузите на портал как ответ на задание 1 (см. раздел Раздел 1.10 о составе архива).



1.9 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Как определить размер окна по горизонтали и вертикали?
2. Как соединить две точки отрезком?
3. Как установить цвет и толщину линии отрезка?
4. Как закрасить фон в окне?
5. Как вывести строку в определенных координатах?
6. Как настроить атрибуты шрифта выводимой строки?
7. Как привязать конец отрезка к точке, отстоящей от правого нижнего угла окна на заданное расстояние по горизонтали и вертикали?

1.10 Загрузка проекта на портал course.sgu.ru

Так как размер загружаемого файла может быть не более 1 Мб, необходимо загружать архив, содержащий только нужные компоненты.



В качестве ответа на задание принимаются только архивы формата zip. Ответы в любых других форматах не проверяются и не принимаются.

1.10.1 Git

Для автоматизированного формирования архива можно использовать git, если проект ведётся с помощью этой системы контроля версий.

Необходимо корректно настроить файл `.gitignore` для того, чтобы в историю не попали лишние файлы, например, файлы сборки проекта и лишние скрытые файлы.

```
1 .*
2 *~
3 !.gitignore
4 !.gitattributes
5 !.clangd
6 !.clang-format
7 !.github
8 *.zip
9 Build
```

Некоторые скрытые файлы всё же полезно иметь в репозитории, поэтому их помечаем отрицанием в этом файле.

Здесь можно заметить файл `.gitattributes`. Так как мы хотим хранить в репозитории исходники наших библиотек, их не получится поместить в `.gitignore`. Но исходники имеют очень большой размер на диске и получаемый архив не получится загрузить на course.sgu.ru.

Добавим в этот файл следующее содержимое:

```
1 Libraries/** export-ignore
```

Теперь можно создавать правильные архивы для загрузки на course с помощью команды

```
1 git archive --format zip -o task1_ivanov.zip HEAD
```

Архив должен называться `task1_ivanov.zip`, где вместо `ivanov` указана ваша фамилия, записанная латиницей.

Для последующих заданий в имени архива должен изменяться номер задания.

1.10.2 Создание архива вручную

При работе над проектом, в нём появится директория Build, которая совсем не нужна при загрузке решения, так как проверяются не только собранные бинарные файлы, а сам код и его собираемость. К тому же проверка может осуществляться на другой ОС.

Кроме этого, директория Libraries имеет большой объём и при этом является общей для всех. Её нужно исключить из финального архива.

Таким образом, в архиве должны находиться файлы и каталоги

- Assets/
- Projects/

- CMakeLists.txt

Указанные файлы поместите в архив с названием `task1_ivanov.zip`, где вместо `ivanov` указана ваша фамилия, записанная латиницей.

Для последующих заданий в имени архива должен изменяться номер задания.

2 Простое рисование в окне

2.1 Предварительная подготовка

За основу проекта для второго задания возьмем уже готовый проект, получившийся в результате выполнения первого задания. Достаточно сделать копию папки с первой лабораторной.

Перед вами ставится задача изобразить рисунок, соответствующий вашему варианту (следует предварительно выбрать номер свободного варианта на портале course.sgu.ru: ссылка «Выбор варианта для задания 2»).

2.2 Определение рисунка

В качестве задания вашего варианта выступает рисунок, исполненный в некоторой координатной сетке. Здесь в качестве примера возьмём следующее изображение.

TODO: добавить картинку

Клетки на рисунке даны лишь для привязки концов отрезка к некоторым координатам и для определения пропорций рисунка. Ваша задача — вывести все отрезки, составляющие данный рисунок.

Попробуем нарисовать линии, из которых состоит данный рисунок. Сначала свяжем с рисунком систему координат. Пусть начало координат находится в левом нижнем углу, а две клетки соответствуют одной единице. Тогда можно перечислить все отрезки из которых состоит рисунок.

Создадим класс для хранения разных типов рисунков. Для этого добавим в проект новый файл `figure.hpp` со следующим содержанием:

```
1 #pragma once
2
3 #include <vector>
4
5 namespace ssu {
6 struct Figure {
7     std::vector<float> vertices;
8 };
9 } // namespace ssu
```

В отдельном файле `hare.hpp` добавим переменную с вершинами этого рисунка:

```
1 #pragma once
2
3 #include "figure.hpp"
4
5 namespace ssu::figure {
6 static const Figure HARE = {
7     // clang-format off
8     .vertices = {
9         // голова
10         0.5f, 3.f, 1.f, 4.5f, // от левой щеки вверх до уха
```


11	1.f, 4.5f, 0.5f, 6.f,	// левое ухо слева снизу вверх
12	0.5f, 6.f, 0.5f, 7.5f,	// левое ухо слева
13	0.5f, 7.5f, 1.f, 8.f,	// левое ухо верх слева
14	1.f, 8.f, 1.5f, 8.f,	// левое ухо верх середина
15	1.5f, 8.f, 2.f, 7.5f,	// левое ухо верх справа
16	2.f, 7.5f, 1.5f, 6.f,	// левое ухо справа сверху вниз
17	1.5f, 6.f, 1.5f, 4.5f,	// левое ухо справа до макушки
18	1.5f, 4.5f, 3.f, 4.5f,	// макушка
19	3.f, 4.5f, 3.f, 6.f,	// правое ухо слева снизу вверх
20	3.f, 6.f, 2.5f, 7.5f,	// правое ухо слева
21	2.5f, 7.5f, 3.f, 8.f,	// правое ухо верх слева
22	3.f, 8.f, 3.5f, 8.f,	// правое ухо верх середина
23	3.5f, 8.f, 4.f, 7.5f,	// правое ухо верх справа
24	4.f, 7.5f, 4.f, 6.f,	// правое ухо сверху вниз
25	4.f, 6.f, 3.5f, 4.5f,	// правое ухо справа
26	3.5f, 4.5f, 4.f, 3.f,	// от правого уха вниз до щеки
27	4.f, 3.f, 3.5f, 1.5f,	// правая скула
28	3.5f, 1.5f, 2.5f, 1.f,	// подбородок справа
29	2.5f, 1.f, 2.f, 1.f,	// подбородок снизу
30	2.f, 1.f, 1.f, 1.5f,	// подбородок слева
31	1.f, 1.5f, 0.5f, 3.f,	// левая скула
32		
33	// туловище	
34	4.f, 3.f, 5.5f, 3.5f,	// спина от головы вправо
35	5.5f, 3.5f, 7.f, 3.5f,	// спина верх
36	7.f, 3.5f, 7.5f, 2.5f,	// спина сверху до хвоста
37	7.5f, 2.5f, 8.f, 2.5f,	// хвост сверху
38	8.f, 2.5f, 8.f, 2.f,	// хвост справа
39	8.f, 2.f, 7.5f, 2.f,	// хвост низ справа налево
40	7.5f, 2.f, 7.5f, 0.5f,	// задняя нога справа сверху вниз
41	7.5f, 0.5f, 6.5f, 0.5f,	// задняя нога низ
42	6.5f, 0.5f, 6.5f, 1.f,	// задняя нога слева
43	6.5f, 1.f, 6.f, 1.f,	// между задних ног
44	6.f, 1.f, 6.f, 0.5f,	// левая задняя нога справа
45	6.f, 0.5f, 5.f, 0.5f,	// левая задняя нога низ
46	5.f, 0.5f, 5.f, 1.f,	// левая задняя нога слева
47	5.f, 1.f, 4.f, 1.f,	// между задними и передними ногами
48	4.f, 1.f, 4.f, 0.5f,	// правая передняя нога справа
49	4.f, 0.5f, 3.f, 0.5f,	// правая передняя нога низ
50	3.f, 0.5f, 3.f, 1.f,	// правая передняя нога слева
51	3.f, 1.f, 2.5f, 1.f,	// между передних ног
52	2.5f, 1.f, 2.5f, 0.5f,	// передняя нога справа
53	2.5f, 0.5f, 1.5f, 0.5f,	// передняя нога низ
54	1.5f, 0.5f, 1.5f, 1.25f,	// передняя нога слева
55		
56	// левый глаз	
57	1.5f, 3.5f, 1.5f, 3.f,	// левый глаз слева сверху вниз
58	1.5f, 3.f, 2.f, 3.f,	// левый глаз низ
59	2.f, 3.f, 2.f, 3.5f,	// левый глаз справа
60	2.f, 3.5f, 1.5f, 3.5f,	// левый глаз верх
61		
62	// правый глаз	
63	2.5f, 3.5f, 2.5f, 3.f,	// правый глаз слева
64	2.5f, 3.f, 3.f, 3.f,	// правый глаз снизу

```

65     3.f, 3.f, 3.f, 3.5f,    // правый глаз справа
66     3.f, 3.5f, 2.5f, 3.5f, // правый глаз сверху
67
68     // ушные раковины
69     1.f, 5.5f, 1.f, 7.f,    // левая ушная раковина
70     3.5f, 5.5f, 3.5f, 7.f, // правая ушная раковина
71
72     // нос
73     2.f, 2.5f, 2.5f, 2.5f,  // нос сверху
74     2.5f, 2.5f, 2.25f, 2.f,  // нос справа
75     2.25f, 2.f, 2.f, 2.5f    // нос слева
76 },
77 // clang-format on
78 };
79 } // namespace ssu::figure

```

В функции `main` добавим переменную `figure`, которая будет ссылаться на переменную с рисунком:

```

1 ssu::Figure &figure = ssu::figure::HARE;

```

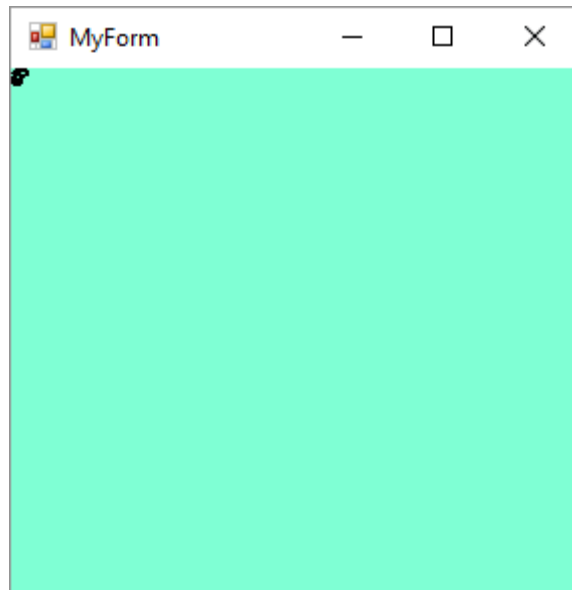
Теперь нужно вывести каждый отрезок на форму. Сделать это нужно внутри основного цикла отрисовки:

```

1 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
2     DrawLineEx(
3         {figure.vertices[i], figure.vertices[i + 1]},
4         {figure.vertices[i + 2], figure.vertices[i + 3]},
5         2,
6         BLACK
7     );
8 }

```

Но после запуска мы увидим следующую картину:



TODO

Черные вкрапления в верхнем левом углу как раз и есть наше изображение. Так как мы выводили отрезки без масштабирования координат, максимальная координата точки не превышает 8.5, а следовательно размер нашей картинки на экране не превышает девяти пикселей как в ширину, так и в высоту.

Чтобы наше изображение приняло приемлемый размер, определим перед циклом коэффициент масштабирования, на который домножим каждую из координат.

```
1 const float S = 30.0f;
2 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
3     DrawLineEx(
4         {S * figure.vertices[i], S * figure.vertices[i + 1]},
5         {S * figure.vertices[i + 2], S * figure.vertices[i + 3]},
6         2,
7         BLACK
8     );
9 }
```

Теперь изображение примет следующий вид.



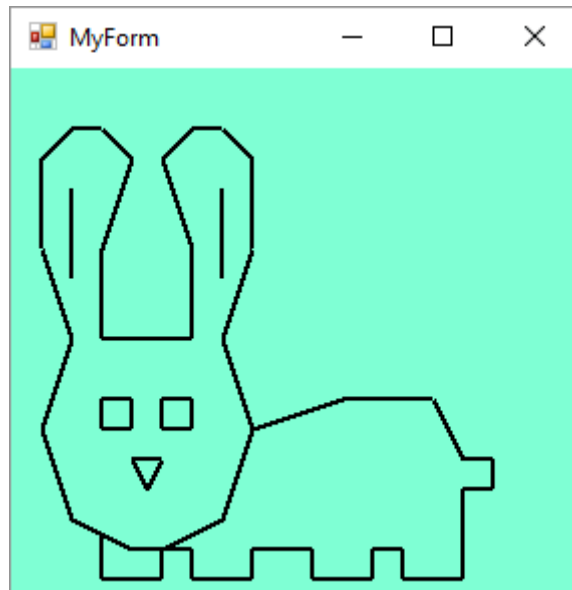
TODO

Размер изображения вполне приемлем. Но так как ось Oy в окне направлена сверху вниз, то картинка остается перевернутой относительно горизонтальной оси.

Для того, чтобы рисунок принял нужное положение, умножим (при выводе отрезка) каждую координату y на -1 . В этом случае все изображение перевернется, но пропадет из поля зрения, так как оно полностью переместится в отрицательную часть оси Oy (так как начало координат в верхнем левом углу). Чтобы сместить его в видимую часть окна, добавим к каждой получившейся координате y одну и ту же положительную величину Ty . Например:

```
1  const float S = 30.0f;
2  const float Ty = 270.0f;
3  for (size_t i = 0; i < figure.vertices.size(); i += 4) {
4      DrawLineEx(
5          {S * figure.vertices[i], Ty - S * figure.vertices[i + 1]},
6          {S * figure.vertices[i + 2], Ty - S * figure.vertices[i + 3]},
7          2,
8          BLACK
9      );
10 }
```

Изображение пример следующий вид:



TODO

Но величины `S` и `Tu` сейчас ничем не обусловлены и им присвоены «магические» числа, которые непонятно откуда взялись. Чтобы придать этим величинам смысл, давайте условимся, что

1. какой бы ни был размер окна, заданное изображение должно быть максимально возможно увеличено так, чтобы полностью помещаться в окне;
2. точка левого верхнего угла левой верхней клетки заданного изображения должна совпадать с левой верхней угловой точкой.

Для этого нам понадобятся величины `Vx` и `Vy` — размеры заданного рисунка в единицах нашей выбранной системы координат, а так же понадобятся размеры видимой части области рисования `Wx` и `Wy`.

Значения `Vx` и `Vy` присущи рисунку и не зависят от формы, поэтому добавим их в класс `Figure`. Теперь этот класс будет выглядеть вот так:

```
1 struct Figure {
2     std::vector<float> vertices;
3     float Vx;
4     float Vy;
5 };
```

К переменной `HARE` тоже добавим инициализацию этих полей:

```
1 static const Figure HARE = {
2     ...
3     .Vx = 8.5f,
4     .Vy = 8.5f,
5 };
```

Значения `Wx` и `Wy` вычислим в основном цикле отрисовки. Их необходимо вычислять именно здесь так как размер окна может меняться во время использования программы.

```
1 const float Wx = static_cast<float>(GetScreenWidth());
2 const float Wy = static_cast<float>(GetScreenHeight());
```

Кроме того, вычислим значения `figureAspect` и `windowAspect` — соотношения сторон нашего рисунка и окна.

```
1 float figureAspect = figure.Vx / figure.Vy; // соотношение сторон рисунка
```

Переменная `windowAspect` должна вычисляться в цикле отрисовки.

```
1 const float windowAspect = Wx / Wy;
```

Зная эти величины, мы можем определить, каков должен быть коэффициент увеличения `S`, чтобы рисунок полностью поместился в окне: если значение соотношения сторон рисунка меньше чем соотношение сторон окна, то следует увеличивать в `Wy / Vy` раз, а в противном случае в `Wx / Vx` раз.

```
1 float S = figureAspect < windowAspect ? Wy / figure.Vy : Wx / figure.Vx;
```

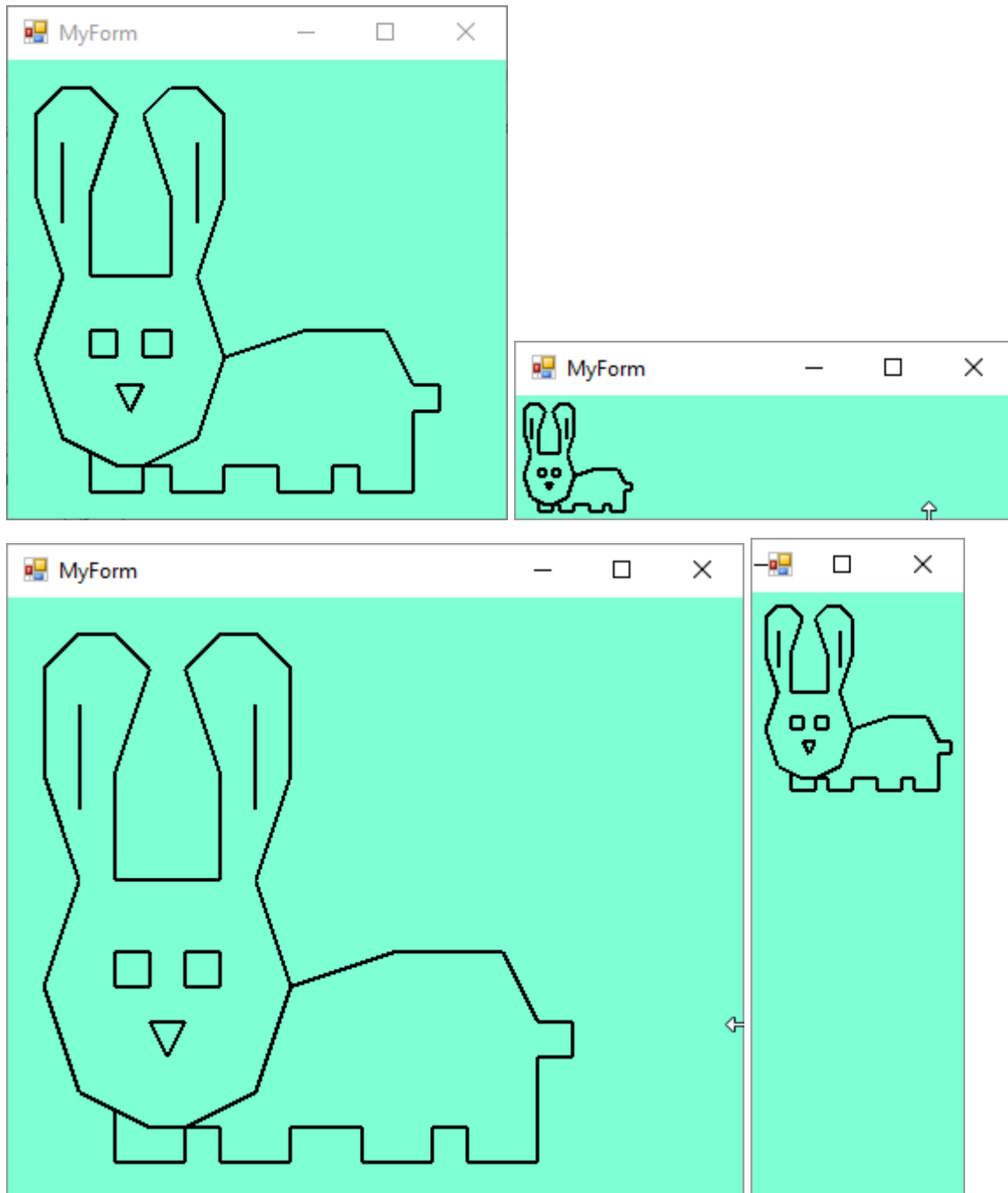
Для того, чтобы выполнилось и второе условие, нужно определить правильную величину смещения рисунка по оси Oy .

Если рисунок не смещать, то точка начала координат рисунка совпадает с точкой начала координат окна. Нам же нужно совместить с точкой начала координат окна верхний левый угол увеличенного рисунка. Таким образом величина смещения должна равняться высоте увеличенного рисунка, т. е. $S \cdot V_y$.

```
1 float Ty = S * figure.Vy; // смещение в положительную сторону по оси Oy после
    смены знака
```

Теперь, все наши выдвинутые условия выполняются: рисунок увеличивается и уменьшается вместе с увеличением или уменьшением окна, оставаясь полностью видимым, а его левый верхний угол привязан к левому верхнему углу окна.

TODO



2.3 Изменение соотношения сторон рисунка

Наряду с тем режимом вывода рисунка, что мы уже реализовали, добавим возможность выводить рисунок с изменением его соотношения сторон до соотношения сторон окна. Будем менять режим вывода рисунка по нажатию клавиши **M**. Для этого добавим переменную `keepAspect` с логическим значением `true`. Когда её значение будет равно `true` будем выводить рисунок в уже реализованном режиме, а в противном случае — с изменённым соотношением сторон.

Выяснить была ли нажата кнопка можно с помощью функции `IsKeyPressed`. В случае, если эта функция вернёт `true`, мы инвертируем значение переменной `keepAspect`. Делать это необходимо в основном цикле отрисовки.

```

1 if (IsKeyPressed(KEY_M)) {
2     keepAspect = !keepAspect;
3 }

```



В RayLib также есть функция `IsKeyDown`. Её отличие от `IsKeyPressed` состоит в том, что она возвращает `true` всё время, пока клавиша остаётся нажатой. `IsKeyPressed` же возвращает `true` только один раз после того, как на клавишу нажали и будет возвращать `false` до тех пор, пока её не отпустят и не нажмут снова.

В этом конкретном случае `IsKeyDown` не подходит, так как клавиша остаётся нажатой какой-то небольшой промежуток времени, а не мгновенно. И всё это время картинка будет переворачиваться каждый кадр.

Чтобы соотношение сторон рисунка совпадало с соотношением сторон окна нужно, чтобы коэффициенты увеличения по осям Ox и Oy были различными. Поэтому вместо коэффициента `S` создадим по одному коэффициенту для каждой из осей, которые обозначим `Sx` и `Sy`. Чтобы уже реализованная версия работала без изменений, присвоим этим коэффициентам одно и то же значение. Сохраним этот код, для случая истинного значения `keepAspect`. В случае, когда `keepAspect` равно `false` вычислим эти коэффициенты по отдельности. Не забываем, что в остальном коде нужно заменить `S` на `Sx` и `Sy` при вычислении соответствующих координат.

```

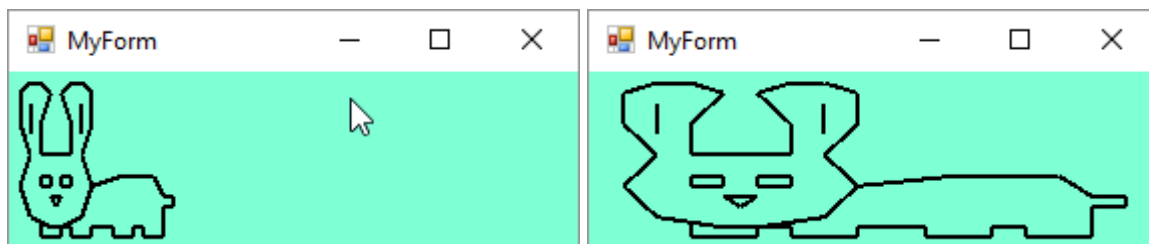
1 float Sx, Sy;
2 if (keepAspect) {
3     float figureAspect = figure.Vx / figure.Vy;
4     Sx = Sy
5     = figureAspect < windowAspect ? Wy / figure.Vy : Wx / figure.Vx;
6 } else {
7     Sx = Wx / figure.Vx;
8     Sy = Wy / figure.Vy;
9 }
10
11 const float Ty = Sy * figure.Vy; // смещение в положительную сторону по оси Oy
    после смены знака
12 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
13     DrawLineEx(
14         {Sx * figure.vertices[i], Ty - Sy * figure.vertices[i + 1]},
15         {Sx * figure.vertices[i + 2], Ty - Sy * figure.vertices[i + 3]},
16         2,
17         BLACK
18     );
19 }

```

Сразу после запуска приложения изменения не будут заметны. Но нажатие клавиши **M** приведет к изменению соотношения сторон выведенного рисунка.

TODO:

картинки



2.4 Задание для самостоятельной работы

Задание 2

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Название проекта должно соответствовать формату {фамилия транслитом}-graphics.
2. Выберите для себя свободный вариант задания 2 на портале course.sgu.ru (Ссылка «Выбор варианта изображения изображения для задания 2»).
3. Добавьте в получившийся проект следующее дополнение. Назначьте реакцию на нажатие клавиши **N**, приводящую к смене изображения построенного примера на рисунок, соответствующий вашему варианту, и наоборот.



Элементы изображения, соответствующего вашему варианту, понадобятся в последующих заданиях. Поэтому настоятельно рекомендуем делать комментарии к координатам подобно комментариям в примере.



В некоторых изображениях присутствуют закрашенные многоугольники. В этом задании перед вами не стоит задача закраски изображения. Вашей задачей является только отрисовка всех ребер изображения, включая контуры закрашенных многоугольников.

4. Архив получившегося проекта загрузите на портал как ответ на задание 2 (см. раздел Раздел 1.10 о составе архива).

2.5 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Как задана система координат в окне? Где находится начало координат? Куда направлены координатные оси?
2. Система координат в окне правая или левая?
3. Система координат, связанную с рисунком, которую мы ввели и используем в разделе Раздел 2.2, правая или левая?
4. Что обозначают значения V_x и V_y ?

5. Объясните смысл значений `Sx`, `Sy` и `Ty`.
6. Почему при отрисовке отрезка, координаты y , извлекаемые из массива `vertices`, домножаются на отрицательные числа?
7. В чем смысл сравнения значений переменных `windowAspect` и `figureAspect`?
8. Что обозначает переменная `keepAspect`?
9. Как назначить реакцию на *нажатие* клавиши для приложения raylib?

3 Преобразования изображений

3.1 Предварительная подготовка

За основу проекта для второго задания возьмем уже готовый проект, получившийся в результате выполнения первого задания. Достаточно сделать копию папки со второй лабораторной.

В рамках этого задания вы продолжите работать с рисунком, соответствующим вашему варианту в задании 2. В ходе построения решения мы будем изменять те или иные готовые фрагменты, поэтому предварительно производить «очистку» проекта не следует.

3.2 Предмет реализации

Мы перестроим проект следующим образом.

1. Определим реакции на нажатия клавиш, которые приводят к изменению вывода изображения в окне: реализуем основные преобразования изображения, такие как поворот, масштабирование, сдвиг и др.
2. Изменим формат представления в программе объектов для изображения.
3. Реализуем чтение исходных данных изображения из файла.

3.3 Геометрические преобразования

Реализуем элементарные геометрические преобразования нашего изображения. Назначим реакции на нажатия клавиш, приводящие к тем или иным преобразованиям. Каждое преобразование будет применяться ко всем точкам изображения (т. е. ко всему изображению сразу).

Преобразования будем представлять в матричной форме. Для этого, каждую точку с координатами (x, y) переведем в однородные координаты: $(x, y, 1.0)$. Определим матрицу начального преобразования `initT`; включающую в себя масштабирование и сдвиг, которые мы определили в решении предыдущего задания. Тогда, если нужно изобразить отрезок от точки **A** до точки **B** (где каждая точка представлена вектором-столбцом однородных координат), будем изображать отрезок от **A** до **B**.

Каждое последующее преобразование (на нажатия клавиш) будем представлять своей матрицей `Ti`. Получится, что первоначальное изображение после нажатия клавиш будет состоять из отрезков от **A** до **B**.

Если обозначим матрицу `initT` через **T**, то в этих обозначений наши отрезки должны быть от **A** до **B**.

Тогда получаем следующий алгоритм:

- для получения начального изображения присвоим `T = initT` и отрисуем все отрезки от **A** до **B**.
- после каждого нажатия клавиши, которой соответствует преобразование, выраженное матрицей `Ti`; присвоим `T = Ti * T` и отрисуем все отрезки от **A** до **B**.

Таким образом, в процедуре отрисовки мы всегда будем изображать отрезки от **A** до **B**. Начальное значение матрицы **T** мы присвоим при инициализации изображения,

а в обработчике `IsKeyDown` мы будем изменять матрицу `T`. Но, прежде чем перейти к реализации этого, нам понадобится реализовать матричные операции.

3.3.1 Реализация матричных операций

Для представления двумерных точек нам понадобятся векторы размерности 2, для двумерных точек в однородных координатах понадобятся векторы размерности 3. Кроме того, нам нужно каким то образом представить матрицы и операции умножения матрицы на матрицу и на вектор (в последующих заданиях нам могут понадобиться другие операции с матрицами и векторами, которые мы реализуем по мере необходимости).

Создадим новый файл `matrix.hpp` и добавим в него директиву `#pragma once`. Все изменения мы будем вносить после этой строки.

Сначала создадим класс `Vec2` для векторов размерности 2. Составляющими экземпляра данного класса будут две вещественные координаты `y`.

```
1 struct Vec2 {
2     float x = 0;
3     float y = 0;
4
5     Vec2() {}
6 }
```

После стандартного конструктора без аргументов добавим еще один, с двумя вещественными параметрами.

```
1 Vec2(float a, float b) : x(a), y(b) {}
```

По аналогии определим класс `Vec3` для векторов размерности 3 с дополнительной координатой `z`.

Тип `Vec3` — основной рабочий тип элементов (точек) в наших вычислениях. Поэтому в классе `Vec3` определим еще несколько операций.

Во первых, для упрощения перехода от евклидовых координат к однородным, добавим еще один конструктор, в котором первый аргумент — двумерный вектор, а второй — дополнительная третья координата.

```
1 Vec3(Vec2 v, float c) : Vec3(v.x, v.y, c) {}
```

Определим операцию `*` для элементов `Vec3` как результат покомпонентного произведения исходных векторов

$$(x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = (x_1 \cdot x_2, y_1 \cdot y_2, z_1 \cdot z_2)$$

Такая операция не имеет ничего общего с векторным или скалярным произведением векторов, но она пригодится нам в дальнейшем.

Сначала в классе `Vec3` перегрузим оператор `*=`, добавив после описания конструкторов следующий код

```
1 Vec3 &operator*=(const Vec3 &v) {
2     this->x *= v.x;
3     this->y *= v.y;
4     this->z *= v.z;
5     return *this;
6 }
```

Теперь, пользуясь оператором `*=`, перегрузим операцию `*`

```
1 const Vec3 operator*(const Vec3 &v) const {
2     return Vec3(*this) *= v; // делаем временную копию текущего объекта,
3                               // которую домножаем на данный вектор,
4                               // и возвращаем ее как результат
5 }
```

Для того, чтобы к элементам вектора можно было обратиться по индексу, переопределим оператор `[]`:

```
1 float &operator[](size_t i) {
2     // ссылку на текущий объект рассматриваем как ссылку
3     // на нулевой элемент массива значений типа float,
4     // после чего обращаемся к его i-му элементу
5     return (reinterpret_cast<float *>(this))[i];
6 }
```

Также стоит добавить `const []` для константных объектов:

```
1 const float &operator[](size_t i) const {
2     return (reinterpret_cast<const float *>(this))[i];
3 }
```

После описания класса `Vec3` добавим описание операции скалярного произведения трехмерных векторов `v1` и `v2`. Это можно реализовать как вычисление суммы координат результата операции `v1 * v2`;

```
1 inline float dot(const Vec3 &a, const Vec3 &b) {
2     Vec3 tmp = a * b; // вычисляем произведения соответствующих
3     // координат
4     return tmp.x + tmp.y + tmp.z; // и возвращаем их сумму
5 }
```



Следует отметить, что помещение реализации функций в заголовочные файлы может привести к проблемам. Для того, чтобы излишне не увеличивать размер проекта, мы не переносим реализации в отдельный `.cpp` файл.

Но для этого необходимо дать компилятору *подсказку*, что вызов функции `dot` может быть заменён на её реализацию.

Все методы в классах/структурах неявно помечаются как `inline`. При этом операторы не являются методами, а «дружественными» (`friend`) функциями.

Теперь мы готовы описать класс `Mat3` для представления матриц 3×3 . Представим матрицу как набор из трех строк — трех векторов размерности 3.

```
1 struct Mat3 {
2     Vec3 row1{};
3     Vec3 row2{};
4     Vec3 row3{};
5
6     Mat3() {}
```

Добавим несколько дополнительных конструкторов.

Пусть первый получает в качестве аргументов три вектора (по аналогии с конструкторами для векторов).

```
1 Mat3(Vec3 r1, Vec3 r2, Vec3 r3) : row1(r1), row2(r2), row3(r3) {}
```

Еще один — конструктор для создания диагональной матрицы с одинаковым числом на главной диагонали.

```
1 Mat3(float a) {
2     row1 = Vec3(a, 0.f, 0.f);
3     row2 = Vec3(0.f, a, 0.f);
4     row3 = Vec3(0.f, 0.f, a);
5 }
```

Перегрузим оператор `[]`, опять же, по аналогии с перегрузкой для `Vec3`.

```
1 Vec3 &operator[](size_t i) {
2     return (reinterpret_cast<Vec3 *>(this))[i];
3 }
```

При перегрузке оператора `[]` для векторов и матриц, можно обращаться к элементам матрицы как к элементам двумерного массива. Пользуясь этим определим метод `transpose` для транспонирования матрицы (применение `A.transpose()` изменяет значение `A` на транспонированную матрицу, после чего возвращает это значение).

```
1 Mat3 &transpose() {
2     Mat3 tmp(*this); // делаем временную копию матрицы
3     for (int i = 0; i < 3; ++i) {
4         for (int j = 0; j < 3; ++j) {
```

```

5         (*this)[i][j] = tmp[j][i]; // заменяем элементы текущего объекта
6                                     // из временной копии
7     }
8 }
9 return *this;
10 }

```

Теперь реализуем операцию умножения матрицы на вектор. Для этого произведем перегрузку оператора `*`. Для определения операции вспомним, что результат произведения матрицы на вектор-столбец — это вектор, каждый элемент которого вычисляется как скалярное произведение соответствующей строки матрицы с заданным вектором.

$$\begin{bmatrix} \text{row}_1 \\ \text{row}_2 \\ \text{row}_3 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} \text{row}_1 \cdot v_1 \\ \text{row}_2 \cdot v_2 \\ \text{row}_3 \cdot v_3 \end{bmatrix}$$

Это можно записать следующим образом.

```

1 const Vec3 operator*(const Vec3 &v) const {
2     Vec3 res{}; // создаем новый вектор (для результата)
3     for (int i = 0; i < 3; ++i) {
4         res[i] = dot((*this)[i], v); // i-й элемент вектора - скалярное
        произведение
5     }
6     return res;
7 }

```



Может показаться, что эта функция возвращает значение прямо со стека из-за чего данный код не будет работать корректно.

В C++ существуют такие понятия, как Copy elision, Return Value Optimization и Named Return Value Optimization. В этом случае используется NRVO. О том, что означают эти понятия мы предлагаем вам разобраться самостоятельно.

Умножение матриц можно представить следующим образом: каждый столбец матрицы-результата получается в результате умножения первой матрицы на соответствующий столбец второй.

$$M_1 \cdot [\text{col}_1 \text{ col}_2 \text{ col}_3] = [M_1 \cdot \text{col}_1 \quad M_1 \cdot \text{col}_2 \quad M_1 \cdot \text{col}_3]$$

Таким образом, нам нужно перебрать столбцы второй матрицы. Но так как наши матрицы хранятся по строкам, будем работать с транспонированной матрицей: каждая строка транспонированной матрицы есть соответствующий столбец исходной. Поэтому наш алгоритм для произведения матриц A и B будет следующим:

- транспонируем матрицу B , получим B^T ;
- перемножим матрицу A с каждой строкой B^T , результаты объединим в матрицу C ;
- результат — матрица C^T , полученная транспонированием матрицы C .

Перегрузим операторы `*=` и `*`, реализовав этот алгоритм

```
1 Mat3 &operator*=(const Mat3 &m) {
2     Mat3 A(*this); // создаем копии исходных матриц
3     Mat3 B(m);
4     B.transpose(); // транспонируем вторую матрицу
5     for (int i = 0; i < 3; i++) {
6         (*this)[i] = A * B[i]; // в i-ю строку текущего объекта записываем
7                                 // результат перемножения первой матрицы с i-й
        строкой
8                                 // транспонированной матрицы,
9     }
10    return this->transpose(); // транспонируем текущий объект, получаем
    результат
11 }
12
13 const Mat3 operator*(const Mat3 &m) const {
14     return Mat3(*this) *= m;
15 }
16
```

И в завершение, опишем функцию `normalize`, получающую для вектора типа `Vec3` вектор `Vec2`, организовав переход из однородных координат в евклидовы (разделив первые две координаты на третью).

```
1 inline Vec2 normalize(const Vec3 &v) {
2     return Vec2(v.x / v.z, v.y / v.z);
3 }
```

На текущем этапе разработка `matrix.hpp` завершена. Подключим этот файл к проекту. В начале файла `main.cpp` добавьте строку

```
1 #include "matrix.hpp"
```

Если в файле `matrix.hpp` ошибок не допущено, то проект должен скомпилироваться и работать так же, как и раньше.

3.3.2 Элементарные преобразования

По примеру, описанному в предыдущем разделе, добавим в проект еще один заголовочный файл с названием `transform.hpp`. В этом файле определим функции для элементарных преобразований, из которых будем строить все наши преобразования в проекте.

Так как мы будем использовать операции, реализованные в `matrix.hpp`, его необходимо включить в `transform.hpp`. Нам понадобятся функции вычисления синуса и косинуса. Поэтому следует подключить еще и `math.h`.

Для каждого преобразования напомним функцию, возвращающую матрицу этого преобразования.

Начнем с преобразования переноса. Как вы должны знать, матрица переноса в двумерной графике определяется двумя параметрами `Ty`. Параметры являются величинами сдвига вдоль соответствующих осей координат, по отношению к исходному изображению. Матрицу переноса можно получить из единичной матрицы, заполнив её последний столбец этими параметрами.

Это можно реализовать следующей процедурой.

```
1 inline Mat3 translate(float Tx, float Ty) {
2     Mat3 res = Mat3(1.f);
3     res[0][2] = Tx;
4     res[1][2] = Ty;
5     return res;
6 }
```

Преобразование масштабирования также зависит от двух коэффициентов `Sy`. Аналогично, создадим матрицу преобразования масштабирования из единичной матрицы. Только в этом случае параметры масштабирования замещают собой элементы матрицы на главной диагонали.

```
1 inline Mat3 scale(float Sx, float Sy) {
2     Mat3 res = Mat3(1.f);
3     res[0][0] = Sx;
4     res[1][1] = Sy;
5     return res;
6 }
```

Реализуем еще одну функцию, на случай когда $Sx = Sy = S$. Воспользуемся для этого уже полученной функцией `scale`.

```
1 inline Mat3 scale(float S) {
2     return scale(S, S);
3 }
```

Наконец, разработаем функцию, выдающую матрицу поворота относительно начала координат против часовой стрелки на заданный угол. Матрица зависит от одного параметра — величины угла поворота `theta`. Первые два элемента на главной диагонали матрицы заполняются косинусами заданного угла, а соседние с ними элементы — синусами. Так как в окне формы система координат левая, то во второй строке матрицы знак при синусе — минус, а в первой — плюс.

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Получим следующую процедуру.

```

1 inline Mat3 rotate(float theta) {
2     Mat3 res = Mat3(1.f);
3     res[0][0] = static_cast<float>(cos(theta));
4     res[1][1] = res[0][0];
5     res[0][1] = static_cast<float>(-sin(theta));
6     res[1][0] = -res[0][1];
7     return res;
8 }

```

Пока, в разработке файла `transform.hpp` остановимся на достигнутом. Подключим наш файл к проекту. В начале файла `main.cpp` добавьте строку

```

1 #include "transform.hpp"

```

Если в файле `transform.hpp` ошибок не допущено, то проект должен скомпилироваться и работать так же, как и раньше.

3.3.3 Начальная отрисовка в матричной форме

Как мы уже говорили в начале раздела Раздел 3.3, нам понадобятся две вспомогательные матрицы `T` и `initT`, с помощью которых мы будем преобразовывать изображение на форме. Опишем их перед основным циклом отрисовки.

```

1 Mat3 T = Mat3(1.f); // матрица, в которой накапливаются все преобразования
2                      // первоначально - единичная матрица
3 Mat3 initT;         // матрица начального преобразования

```

Будем здесь предполагать, что в результате выполнения второго задания в цикле присутствует фрагмент, в котором производится отрисовка всех имеющихся отрезков¹:

```

1 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
2     DrawLineEx(
3         {Sx * figure.vertices[i], Ty - Sy * figure.vertices[i + 1]},
4         {Sx * figure.vertices[i + 2], Ty - Sy * figure.vertices[i + 3]},
5         2,
6         BLACK
7     );
8 }

```

Если внимательно посмотреть на вызов `DrawLineEx`, то можно получить представление о том преобразовании, что применяется к каждой точке. Это преобразование и запишем в матрицу `initT`. Рассмотрим этот момент поподробней:

- каждая координата x домножается на `Sx`, а каждая координата y — на `-Sy`, т. е. происходит масштабирование с этими параметрами;
- после масштабирования происходит перенос по оси Oy на `Ty`, а по оси Ox сдвиг отсутствует или, по другому, по оси Ox происходит сдвиг на 0.

¹Если Ваша версия цикла отрисовки отличается значительно, вы можете вернуться к версии этой процедуры, полученной в конце главы Раздел 2

Перепишем этот фрагмент несколько иначе, подключив матричную форму.

```
1 // Преобразования применяются справа налево. Сначала масштабирование, а потом
2 // перенос. В initT совмещаем эти два преобразования.
3 initT = translate(0, Ty) * scale(Sx, -Sy);
4 Mat3 M = T * initT; // совмещение начального преобразования и
5                     // накопленных преобразований
6 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
7     // начало отрезка в однородных координатах
8     Vec3 A = Vec3(figure.vertices[i], figure.vertices[i + 1], 1);
9     // конец отрезка в однородных координатах
10    Vec3 B = Vec3(figure.vertices[i + 2], figure.vertices[i + 3], 1);
11    Vec2 a = normalize(M * A); // начало отрезка после преобразования
12    Vec2 b = normalize(M * B); // конец отрезка после преобразования
13    DrawLineEx({a.x, a.y}, {b.x, b.y}, 2, BLACK);
14 }
```

Проведенные исправления не должны привести к каким-то изменениям в функционировании программы.

3.3.4 Назначение горячих клавиш

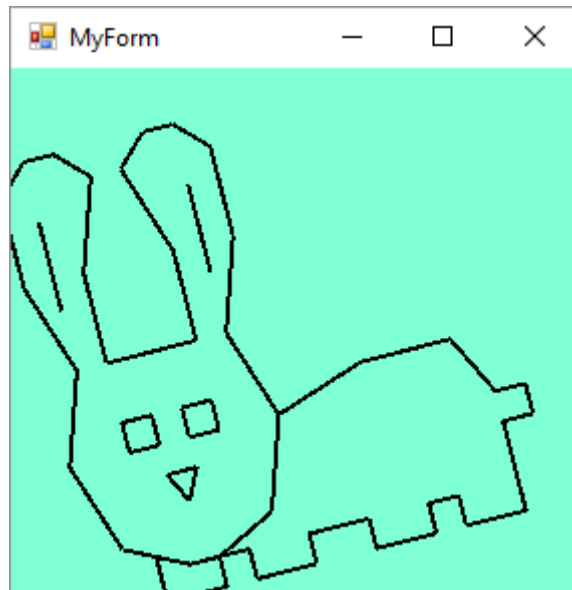
Добавим несколько дополнительных реакций на нажатия клавиш. Но прежде, добавим пару переменных с координатами центра текущего окна.

```
1 const float Wcx = Wx / 2.0f;
2 const float Wcy = Wy / 2.0f;
```

Добавим обработчик нажатия на клавишу **Q** — поворот изображения вокруг центра окна на -0.01 радиана. Преобразование будет заключаться в переносе начала координат в середину окна, повороте и обратном переносе. Это преобразование совместим с тем, что уже накопилось в **T**, т. е. домножим матрицу **T** слева последовательно на матрицы этих преобразований.

```
1 if (IsKeyDown(KEY_Q)) {
2     T = translate(-Wcx, -Wcy) * T; // перенос начала координат в (Wcx, Wcy)
3     T = rotate(-0.01f) * T; // поворот на -0.01 радиан относительно
4                             // нового центра
5     T = translate(Wcx, Wcy) * T; // перенос начала координат обратно
6 }
```

Если все проведено корректно, то после запуска проекта нажатие клавиши **Q** будет приводить к повороту рисунка по часовой стрелке.



TODO

Добавим ещё одно преобразование на нажатие клавиши **W** — сдвиг изображения вверх на 1 пиксель

```
1 if (IsKeyDown(KEY_W)) {  
2     T = translate(0, -1) * T;  
3 }
```

И на клавишу **C** — сброс всех сделанных преобразований.

```
1 if (IsKeyPressed(KEY_C)) {  
2     T = Mat3(1);  
3 }
```

Запустите проект и опробуйте новые назначенные клавиши.

3.4 Внутренний формат представления объектов

Объект, который мы рисуем, состоит из набора отрезков. При создании этого набора нам пришлось столкнуться с тем, что большинство точек, упоминаемых в нем, повторяются дважды (а возможно и большее количество раз). Например, как конец одного отрезка и начало другого, инцидентного с ним, отрезка. Удобнее было бы представить изображение как набор ломаных линий, где каждая линия задана последовательностью точек. Более того, хотелось бы задать характеристики каждой такой ломаной, такие как цвет и толщина линии.

Изменим структуру `Figure` соответствующим образом.

Координаты каждой точки ломаной будем представлять в виде двумерного вектора, поэтому понадобится подключение файла заголовка с описанием типа `vec2`. Кроме того, саму ломаную будем представлять в виде набора таких точек объединенных в контейнер `std::vector`.

Опишем структуру, которая представляет из себя одну ломанную линию определённого цвета и определённой толщины.

```
1 struct Path {
2     std::vector<Vec2> vertices;
3     Color color;
4     float thickness;
5 }
```

Добавим простой конструктор от трех аргументов, который будет объединять все заданные аргументы в объект.

```
1 Path(std::vector<Vec2> vertices, Color color, float thickness)
2     : vertices(vertices)
3     , color(color)
4     , thickness(thickness) {}
```

Теперь в структуре `Figure` можно хранить не отдельные координаты, а массив ломаных линий.

```
1 struct Figure {
2     std::vector<Path> paths;
3     float Vx, Vy;
4 }
```

Добавим пустой конструктор для того, чтобы можно было создать переменную и заполнить её значениями динамически после инициализации.

```
1 Figure() = default;
```

Добавим в него аналогичный конструктор для указанных полей.

```
1 Figure(std::vector<Path> paths, float Vx, float Vy)
2     : paths(paths)
3     , Vx(Vx)
4     , Vy(Vy) {}
```

Дальнейшие изменения коснутся содержимого файла `main.cpp`.

Так как файлы со старыми фигурами теперь не соответствуют новому интерфейсу структуры `Figure`, их нужно исправить. Заголовочный файл с фигурой из своего варианта пока что исключите из `main.cpp`.

Начнём исправлять фигуру из файла `hare.hpp`.

В первую очередь, с существующим конструктором у нас больше не получится проинициализировать объект напрямую — необходимо вызвать конструктор. Заполним переменную `HARE` новыми значениями:

```

1 static const Figure HARE = Figure(
2     {
3         ssu::Path(
4             {{0.5f, 3.f}, {1.f, 4.5f}, {0.5f, 6.f}, {0.5f, 7.5f}, {1.f, 8.f},
5              {1.5f, 8.f}, {2.f, 7.5f}, {1.5f, 6.f}, {1.5f, 4.5f}, {3.f, 4.5f},
6              {3.f, 6.f}, {2.5f, 7.5f}, {3.f, 8.f}, {3.5f, 8.f}, {4.f, 7.5f},
7              {4.f, 6.f}, {3.5f, 4.5f}, {4.f, 3.f}, {3.5f, 1.5f}, {2.5f, 1.f},
8              {2.f, 1.f}, {1.f, 1.5f}, {0.5f, 3.f}},
9             RED,
10            2
11        ),
12        ssu::Path(
13            {{1.5f, 3.5f}, {1.5f, 3.f}, {2.f, 3.f}, {2.f, 3.5f}, {1.5f, 3.5f}},
14            GREEN,
15            4
16        ),
17        ssu::Path(
18            {{2.5f, 3.5f}, {2.5f, 3.f}, {3.f, 3.f}, {3.f, 3.5f}, {2.5f, 3.5f}},
19            GREEN,
20            4
21        ),
22    },
23    8.5f,
24    8.5f
25 );

```

Теперь необходимо исправить цикл, отрисовывающий линии. Теперь перейдем к изменению обработчика события *Paint*.

Опять обратимся к циклу, ответственному за вывод линий. Сейчас (после последних изменений) он выглядит так:

```

1 for (size_t i = 0; i < figure.vertices.size(); i += 4) {
2     Vec3 A = Vec3(figure.vertices[i], figure.vertices[i + 1], 1);
3     Vec3 B = Vec3(figure.vertices[i + 2], figure.vertices[i + 3], 1);
4     Vec2 a = normalize(M * A);
5     Vec2 b = normalize(M * B);
6     DrawLineEx({a.x, a.y}, {b.x, b.y}, 2, BLACK);
7 }

```

Вместо этого цикла вставим отрисовку линий из списка `figure.paths`.

```

1 for (const auto &lines : figure.paths) {
2
3 }

```

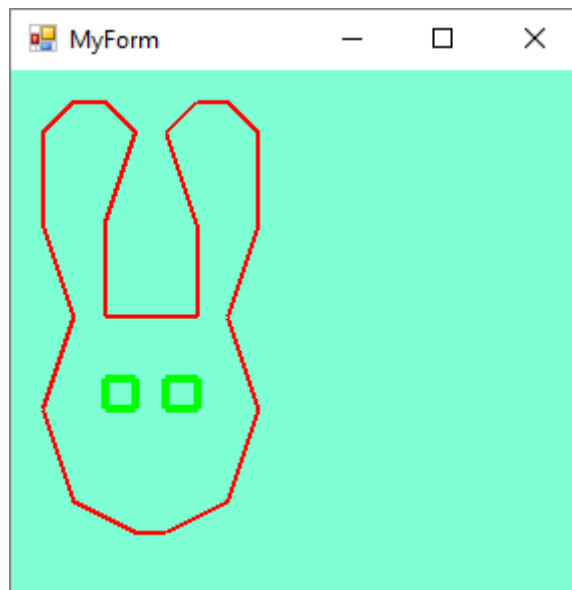
Теперь нужно организовать цикл по всем отрезкам в `lines` или, скажем лучше, цикл пробегающий по всем точкам — концам отрезков. Конец первого отрезка — вторая точка в списке отрезков (точка с индексом 1). У каждого отрезка, за исключением первого, начальная точка — это конечная точка предыдущего отрезка.

Перед отрисовкой каждую точку переводим в однородные координаты, домножаем на матрицу преобразований и возвращаем в обычные евклидовы координаты.

Таким образом, получаем следующий цикл.

```
1 Vec2 start = normalize(M * Vec3(lines.vertices[0], 1));
2 for (const auto &line : lines.vertices) {
3     const Vec2 end = normalize(M * Vec3(line, 1));
4     DrawLineEx(
5         {start.x, start.y},
6         {end.x, end.y},
7         lines.thickness,
8         lines.color
9     );
10
11     start = end;
12 }
```

Если все введено без ошибок, то запуск приложения приведет к получению следующего изображения.



TODO

3.5 Файловый ввод данных

Безусловно, утомительно и некрасиво наполнять список набором команд подобным тому, что приведен в качестве примера в предыдущем разделе. Вместо этого организуем ввод исходных данных для изображения из текстового файла.

Сначала обговорим формат входного файла.

3.5.1 Формат входного файла

Для простоты исполнения, будем использовать следующий формат входного файла.

В файле могут встречаться пустые строки, а также строки комментариев. Строка комментариев должна начинаться с символа `#`.

Кроме пустых строк и строк с комментариями в файле могут встречаться строки команд следующего вида:

frame Vx Vy

команда установки высоты и ширины изображения. Если команда встречается в файле более одного раза, то каждый последующий экземпляр команды отменяет действие предыдущей;

color R G B

команда установки цвета для последующих объектов;

thickness p

команда установки толщины линий последующих объектов;

path n

задается ломаная линия состоящая из **n** точек. Команда объявляет, что в следующих строках (отличных от пустых и строк с комментариями) представлены координаты **n** точек, по паре координат в отдельной строке.

Например, изображение, составленное нами в обработчике *Load*, можно задать файлом со следующим содержанием.

```
1 frame 8.5 8.5
2
3 # голова
4 color 255 0 0
5 thickness 2
6 path 23
7 # от левой щеки вверх
8 0.5 3.
9 1. 4.5
10 0.5 6.
11 # левое ухо верх слева
12 0.5 7.5
13 1. 8.
14 1.5 8.
15 2. 7.5
16 1.5 6.
17 # макушка
18 1.5 4.5
19 3. 4.5
20 3. 6.
21 # правое ухо верх слева
22 2.5 7.5
23 3. 8.
24 3.5 8.
25 4. 7.5
26 4. 6.
27 3.5 4.5
28 # правая скула
29 4. 3.
```



```

30 3.5 1.5
31 2.5 1.
32 2. 1.
33 # левая скула
34 1. 1.5
35 0.5 3.
36
37 # глаза
38 color 0 255 0
39 thickness 4
40
41 # левый глаз
42 path 5
43 1.5 3.5
44 1.5 3.
45 2. 3.
46 2. 3.5
47 1.5 3.5
48
49 # правый глаз
50 path 5
51 2.5 3.5
52 2.5 3.
53 3. 3.
54 3. 3.5
55 2.5 3.5

```

Сохраните содержимое этого примера в текстовом файле с именем `hare.txt`.

В дальнейшем, чтобы не предусматривать в коде дополнительный анализ, будем считать, что файл задан всегда корректно.

3.5.2 Чтение входного файла

Добавим в нашу программу загрузку данных из файла.

Сначала в файле перед строкой с добавим

```

1 #include <fstream>
2 #include <sstream>

```

Будем определять имя файла для открытия с помощью диалогового окна открытия файлов. Само диалоговое окно будет открываться по нажатию кнопки в интерфейса. Для этого используем библиотеки NFD и raygui.

Подключим библиотеки к нашему проекту:

```

1 #include <nfd.h>

```

Далее добавим кнопку в окно. Добавить её необходимо между вызовами `BeginDrawing` и `EndDrawing`. Расположим её верхнюю левую точку на постоянном расстоянии в 140 пикселей от правого края окна и в 20 пикселей от верхнего края. Ширина и высота

кнопки — 120 и 30 пикселей соответственно. В качестве надписи установим строку «OPEN FILE»:

```
1 if (GuiButton({Wx - 140, 20, 120, 30}, "OPEN FILE")) {  
2  
3 }
```



raugui — это библиотека с подходом Immediate Mode GUI. При таком подходе, элементы интерфейса рисуются сразу же при вызове функции соответствующего примитива. Они не хранят никакого состояния и детали отрисовки вычисляются на каждом новом кадре (например, если курсор наведён на кнопку).

В случае кнопки, эта функция также возвращает была ли нажата кнопка в этот момент. Поэтому с помощью простого `if` можно обработать нажатие. Такой подход удобен для отладки графических приложений и зачастую используется для создания отладочного интерфейса на стадии разработки большинства компьютерных игр.

Их противоположность — Retained Mode GUI. Например: GTK, QT, Windows Forms, UWP, Cocoa и большинство HUD интерфейсов компьютерных игр.

В обработчике нажатия на кнопку опишем настройки диалогового окна и процесс чтения файла.

Сначала необходимо создать переменную, в которую NFD запишет путь до выбранного файла. Для этого библиотека предоставляет тип `nfdchar_t` (который фактически является обычным `char`):

```
1 nfdchar_t *outPath
```

Так как наши фигуры хранятся в файлах с расширением `.txt`, полезно добавить фильтр для диалогового окна, чтобы можно было выбирать только из подходящих нам файлов. На случай, если по каким-то причинам понадобится открыть модель с другим расширением, добавим фильтр с «Любыми файлами»:

```
1 nfdfilteritem_t filterItem[2]  
2 = {"Text files", "txt"}, {"All files", "*"};
```

Теперь можно открыть диалоговое окно с нашими параметрами:

```
1 nfdresult_t result  
2 = NFD_OpenDialog(&outPath, filterItem, 2, nullptr);
```

NFD при выборе файла записывает путь в определённую ранее переменную и возвращает статус открытия. Обработаем статус того, что всё прошло успешно.

Вызовем пока не определённую функцию `readFromFile` с аргументом, содержащим прочитанный путь. После того, как эта функция загрузит фигуру, её можно присвоить переменной `figure`. После этого необходимо очистить память, выделенную под хранение пути до файла с помощью функции `NFD_FreePath`.

```
1 if (result == NFD_OKAY) {
2     figure = readFromFile(outPath);
3     NFD_FreePath(outPath);
4 }
```

В диалоговом окне помимо кнопки открытия файла есть также кнопка отмены. Обрабатываем этот случай, а также любые другие ошибки:

```
1 if (result == NFD_OKAY) {
2     figure = readFromFile(outPath);
3     NFD_FreePath(outPath);
4 } else if (result == NFD_CANCEL) {
5     std::cerr << "INFO: NFD: user pressed cancel" << std::endl;
6 } else {
7     std::cerr << "ERROR: " << NFD_GetError() << std::endl;
8 }
```

Теперь пора написать функцию `readFromFile`. Она принимает путь до файла и возвращает прочитанную фигуру. Для упрощения кода проекта, внутри функции будем предполагать, что файл может быть прочитан и в нём не допущено ошибок.

```
1 ssu::Figure readFromFile(const char *fileName) {
2     std::ifstream in(fileName);
3
4 }
```

Опишем переменную фигуры, а также временные переменные толщины и компонент цвета, которые будут обновляться по мере чтения файла:

```
1 ssu::Figure figure;
2 int r, g, b;
3 float thickness;
```

После этого читаем и обрабатываем каждую строку файла. В результате необходимо вернуть полученную фигуру.

```
1 std::string line; // временная переменная, в которую считываются строки
2 while (in) {
3     // считываем очередную строку
4     getline(in, line);
5
6     // обрабатываем строку
7 }
8 return figure;
```

Для каждой прочитанной строки сначала проводим проверку — не является ли она пустой строкой или строкой с комментариями. Выделим её в отдельную функцию.

```
1 bool isIgnorableLine(const std::string &line) {
2     return line.find_first_not_of(" \t\r\n") == std::string::npos
3         || line.front() == '#';
4 }
```

Нужно добавить эту проверку внутри цикла сразу после считывания строки:

```
1 if (isIgnorableLine(line)) {
2     continue;
3 }
```

Если строка таковой не является, то из нее будем производить чтение информации. Для этого сначала представляем ее в виде стандартного строкового потока.

```
1 stringstream s(line);
```

Каждая значащая строка файла начинается с имени команды (за исключением строк, следующих за командой). Поэтому, прочитаем из потока имя команды:

```
1 string cmd; // переменная для имени команды
2 s >> cmd; // считываем имя команды
```

Последующие действия должны зависеть от того, какая команда была прочитана:

```
1 if (cmd == "frame") {
2     // размеры изображения
3 } else if (cmd == "color") {
4     // цвет линии
5 } else if (cmd == "thickness") {
6     // толщина линии
7 } else if (cmd == "path") {
8     // набор точек
9 }
```

В случае, если прочитана команда `frame`, необходимо прочитать из потока 2 числа, которые можно сразу сохранить в поля переменной `figure`. После этого обновим значение соотношения сторон

```
1 s >> figure.Vx >> figure.Vy;
```

Если прочитана команда `color`, то считываем из потока три составляющие цвета в заготовленные для этого переменные.

```
1 s >> r >> g >> b; // считываем три компоненты цвета
```

Если прочитана команда `thickness`, то считываем значение толщины.

```
1 s >> thickness; // считываем значение толщины
```

Обработка команды `path` является более сложной. Сначала опишем временный список для прочитанных точек.

```
1 vector<Vec2> vertices; // список точек ломаной
```

Опишем и прочитаем из строки количество точек.

```
1 int n; // количество точек
2 s >> n;
```

Чтобы прочитать нужное количество точек, заведем еще одну строковую переменную, и организуем цикл, который считывает построчно файл. Считаем, что после этого цикла все точки прочитаны и находятся в списке `vertices`. Поэтому после цикла из этого списка и значений переменных `r`, `g`, `b` и `thickness` сформируем объект и запишем его в список `figure.paths`.

```
1 string str1; // дополнительная строка для чтения из файла
2 while (n > 0) { // пока не все точки считаны
3     getline(in, str1); // считываем в str1 из входного файла очередную строку
4     // так как файл корректный, то на конец файла проверять не нужно
5
6     --n;
7 }
8 // все точки считаны, генерируем ломаную (path) и кладем ее в список
9 figure.paths.push_back(ssu::Path(
10     vertices,
11     Color{
12         static_cast<uint8_t>(r),
13         static_cast<uint8_t>(g),
14         static_cast<uint8_t>(b),
15         255
16     },
17     thickness
18 ));
```

Внутри цикла, так же, как и раньше, проверим, что прочитанная строка не пуста и не является комментарием.

```
1 if (isIgnorableLine(str1)) {
2     continue;
3 }
4 // прочитанная строка не пуста и не комментарий
5 // значит в ней пара координат
```

Если в строке пара координат, то прочитаем её, сформируем объект и добавим его в список.

```
1 float x, y; // переменные для считывания
2 stringstream s1(str1); // еще один строковый поток из строки str1
3 s1 >> x >> y;
4 vertices.push_back(Vec2(x, y)); // добавляем точку в список
```

В результате процедура должна принять следующий вид.

```
1 ssu::Figure readFromFile(const char *fileName) {
2     std::ifstream in(fileName);
3
4     ssu::Figure figure;
5     int r, g, b;
6     float thickness;
7
8     std::string line; // временная переменная, в которую считываются строки
9     while (in) {
10         // считываем очередную строку
11         getline(in, line);
12         if (isIgnorableLine(line)) {
13             continue;
14         }
15
16         std::stringstream s(line);
17
18         string cmd; // переменная для имени команды
19         s >> cmd; // считываем имя команды
20         if (cmd == "frame") { // размеры изображения
21             s >> figure.Vx >> figure.Vy;
22             std::cout << figure.Vx << ' ' << figure.Vy << std::endl;
23         } else if (cmd == "color") { // цвет линии
24             s >> r >> g >> b; // считываем три компоненты цвета
25         } else if (cmd == "thickness") { // толщина линии
26             s >> thickness; // считываем значение толщины
27         } else if (cmd == "path") { // набор точек
28             vector<Vec2> vertices; // список точек ломаной
29             int n; // количество точек
30             s >> n;
31             std::string str1; // дополнительная строка для чтения из файла
32             while (n > 0) { // пока не все точки считаны
33                 getline(in, str1);
34                 if (isIgnorableLine(str1)) {
35                     continue;
36                 }
37                 float x, y;
38                 stringstream s1(str1);
39                 s1 >> x >> y;
40                 vertices.push_back(Vec2(x, y)); // добавляем точку в список
41                 --n;
42             }
43             // все точки считаны, генерируем ломаную (path) и кладем ее в список
```

```

44         figure.paths.push_back(ssu::Path(
45             vertices,
46             Color{
47                 static_cast<uint8_t>(r),
48                 static_cast<uint8_t>(g),
49                 static_cast<uint8_t>(b),
50                 255
51             },
52             thickness
53         ));
54     }
55 }
56 return figure;
57 }

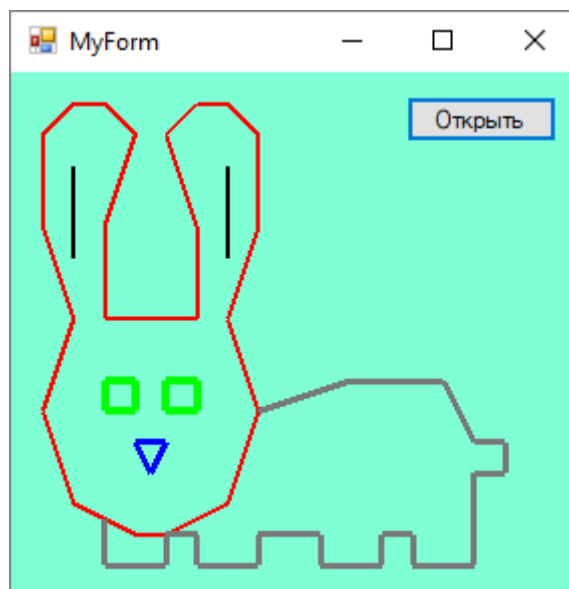
```

Если сейчас запустить проект, нажать на кнопку открытия файла и выбрать файл, то данные из файла будут считаны и изображение построено. Но сейчас изображение ничем не отличается от того, что было построено без загрузки файла.



Стоит обратить внимание на то, что принцип чтения из файла позволяет добавлять в наш файл незначущую информацию после параметров команд или координат точек, что может послужить возможностью добавлять комментарии к каждой строке входного файла.

На портале course.sgu.ru к заданию 3 приложен файл `Hare_full.txt`. Если этот файл открыть в приложении, должна получиться следующая картинка.



TODO

3.6 Очистка проекта

Внесем в проект еще несколько изменений. Сейчас у нас можно изменять значение начального преобразования с помощью нажатия клавиши **M** или изменения размера окна. Отменим такую возможность.

Прежде всего, удалим реакции на клавиши **M** и **N** (реакция на **N** стала бесполезной, так как теперь у нас выводится лишь то одно изображение, элементы которого сохранены в списке).

Далее условимся, что принцип, введенный в разделе Раздел 2.2:

1. какой бы ни был размер окна, заданное изображение должно быть максимально возможно увеличено так, чтобы полностью помещаться в окне;
2. точка левого верхнего угла левой верхней клетки заданного изображения должна совпадать с левой верхней угловой точкой.

будет действовать только при первой отрисовке изображения, но не будет действовать при изменении размеров окна при выведенном на него изображении.

Это можно сделать, если вычисление матрицы производить в момент загрузки изображения.

В основном цикле за вычисление `initT` отвечает блок

```
1 float Sx, Sy;
2 if (keepAspect) {
3     const float figureAspect = figure.Vx / figure.Vy;
4     Sx = Sy
5     = figureAspect < windowAspect ? Wy / figure.Vy : Wx / figure.Vx;
6 } else {
7     Sx = Wx / figure.Vx;
8     Sy = Wy / figure.Vy;
9 }
10
11 const float Ty = Sy * figure.Vy;
12 initT = translate(0, Ty) * scale(Sx, -Sy);
```

Так как мы отменили изменение переменной `keepAspect`, то блок можно выкинуть. Останется фрагмент:

```
1 float Sx, Sy;
2 const float figureAspect = figure.Vx / figure.Vy;
3 Sx = Sy = figureAspect < windowAspect ? Wy / figure.Vy : Wx / figure.Vx;
4
5 const float Ty = Sy * figure.Vy;
6 initT = translate(0, Ty) * scale(Sx, -Sy);
```

Теперь отпала нужда в двух параметрах масштабирования `Sx` и `Sy`. Заменяем их одним параметром `S`. Получим

```
1 const float figureAspect = figure.Vx / figure.Vy;
2 const float S
3     = figureAspect < windowAspect ? Wy / figure.Vy : Wx / figure.Vx;
4 const float Ty = S * figure.Vy;
5 initT = translate(0, Ty) * scale(S, -S);
```


И теперь этот блок перенесем в блок обработки нажатия на кнопку после строки, в которой загружается новая фигура из файла.

```
1 if (result == NFD_OKAY) {
2     figure = readFromFile(outPath);
3     const float figureAspect = figure.Vx / figure.Vy;
4     const float S = figureAspect < windowAspect ? Wy / figure.Vy
5                                                         : Wx / figure.Vx;
6     const float Ty = S * figure.Vy;
7     initT = translate(0, Ty) * scale(S, -S);
8     NFD_FreePath(outPath);
9 }
```

Наконец, так как матрица `initT` после загрузки изображения из файла перевычисляться не будет, то отпадает необходимость в матрице `M`: можно просто матрице `T` всегда, в качестве начального значения, присваивать матрицу `initT`. Первый раз сделаем это сразу в после вычисления `initT` в блоке загрузки изображения.

```
1 initT = translate(0, Ty) * scale(S, -S);
2 T = initT;
```

То же самое нужно будет сделать в обработчике `IsKeyPressed` при нажатии клавиши `C` (вместо того, чтобы присваивать единичную матрицу).

```
1 if (IsKeyPressed(KEY_C)) {
2     T = initT;
3 }
```

Таким образом, матрицу `M` в основном цикле заменим на `T`.

Удалим из проекта те инструкции, которые нам больше не нужны.

Параметр `keepAspect` стал не нужен. Удалим его объявление. То же самое касается возможных параметров, которые Вы добавляли в приложении при выполнении задания 2.

Так как теперь будут отображаться только фигуры из файлов, переменной `figure` при старте ничего не нужно присваивать. Стандартный конструктор создаст такой объект, что программа продолжит корректно работать, но не станет выводить никакую фигуру при запуске.

Таким образом, перед началом основного цикла останется три переменных:

```
1 ssu::Figure figure;
2 Mat3 T = Mat3(1.f);
3 Mat3 initT;
```

3.7 Задание для самостоятельной работы

Задание 3

1. Создайте приложение, включающее в себя все, что было описано выше в качестве примера. Название проекта должно соответствовать формату {фамилия транслитом}-graphics.
2. В заголовочном файле `transform.hpp` определите процедуры `mirrorY` без параметров, выдающие матрицы зеркального отражения относительно осей Ox и Oy соответственно.
3. Назначьте реакции на нажатие клавиш
 - `E` — поворот изображения по часовой стрелке на 0.01 радиан относительно текущего центра окна;
 - `S`, `A`, `D` — сдвиг изображения соответственно вниз, влево и вправо на 1 пиксель;
 - `R`, `Y` — поворот изображения соответственно по и против часовой стрелки на 0.05 радиан относительно текущего центра окна;
 - `T`, `G`, `F`, `H` — сдвиг изображения соответственно вверх, вниз, влево и вправо на 10 пикселей;
 - `Z`, `X` — соответственно увеличение и уменьшение изображения относительно текущего центра окна в 1.1 раза;
 - `U`, `J` — зеркальное отражение относительно горизонтальной и вертикальной оси, проходящей через центр окна, соответственно;
 - `I`, `K` — соответственно растяжение и сжатие изображения по горизонтали относительно текущего центра окна в 1.1 раза;
 - `O`, `L` — соответственно растяжение и сжатие изображения по вертикали относительно текущего центра окна в 1.1 раза;
4. Создайте текстовый файл, описывающий изображение, соответствующее Вашему варианту в задании 2, который корректно загружается и отображается в проекте. Разбейте Ваше изображение на осмысленные фрагменты (посоветуйтесь с преподавателем о том, из каких фрагментов должна быть выполнена Ваша картинка), для которых задайте свои цвета и толщину линий.
5. Архив получившегося проекта загрузите на портал как ответ на задание 3. Текстовый файл с описанием Вашего изображения должен быть включен в архив.

3.8 Контрольные вопросы

Подготовьте ответы на следующие вопросы.

1. Что такое «однородные координаты» точки?
2. Откуда берется содержимое матриц элементарных преобразований, описанных и реализованных в разделе Раздел 3.3.2?
3. Что произойдет, если в преобразованиях умножать вектор на матрицу не слева, а справа?
4. Матрицей какого преобразования является матрица `initT`?
5. Матрицей какого преобразования является матрица `T`?
6. Как организуется поворот изображения относительно начала координат? Из каких элементарных преобразований строится это преобразование?
7. Как организовать зеркальное отражение изображения относительно вертикальной оси, проходящей через центр окна?
8. Что произойдет с изображением, если вместо масштабирования относительно центра окна использовать элементарные преобразования масштабирования?
9. Что такое «компоненты RGB»?
10. Как можно установить цвет рисуемого объекта из составляющих RGB?
11. Откуда в результирующем проекте берутся цвета линий изображения?