;

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Институт искусственного интеллекта
Кафедра проблем управления

КУРСОВАЯ РАБОТА

по дисциплине **Основы программирования систем управления**

Тема курсовой работы: «Реализация программы вращения равностороннего треугольника».

Студент группы КВБО-07-23: Николин А. А. _____

Руководитель курсовой работы: ст. преп. Смирнов М. Ю. _____

Работа представлена к защите: «___» _____ 2024 г.

Допущен к защите: «___» _____ 2024 г.

Москва 2024

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА - Российский технологический университет»

РТУ МИРЭА

Институт искусственного интеллекта

Кафедра проблем управления

ЗАДАНИЕ

на выполнение курсовой работы по дисциплине «Основы программирования систем управления»

Утверждаю

Заведующий кафедрой ПУ

_____ Романов М. П.

Подпись

«___» _____ 2024 г.

Студент: Николин А. А. Группа: КВБО-07-23

Тема: «Реализация программы вращения равностороннего треугольника»

Исходные данные:

1. Вращение равностороннего треугольника
2. Изменение скорости вращения треугольника
3. Изменение размеров треугольника

Перечень вопросов, подлежащих обработке, и обязательного графического материала:

1. Использование компьютерной графики
2. Вращение геометрической фигуры
3. Блок-схема алгоритма работы программы

Срок представления к защите курсовой работы: до «___» ____ 2024 г.

Задание на курсовую работу выдал _____ Смирнов М. Ю.

«___» _____ 2024 г.

Задание на курсовую работу получил _____ Николин А. А.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На языке программирования *C* была разработана программа, которая выводит на экран вращающийся равносторонний треугольник. В программе реализована возможность изменения скорости вращения треугольника, а также его размеров. Управление параметрами скорости вращения и размеров треугольника реализовано с помощью пользовательского интерфейса в виде ползунков, которыми можно управлять мышкой.

Поставленная задача

Цель работы: целью работы является создание программы на языке программирования *C/C++*, способной отображать на экране вращающийся вокруг своего центра равносторонний треугольник, а также предоставляющей возможность изменения скорости вращения треугольника и его размеров во время работы программы.

Для упрощения процесса разработки были выделены основные подзадачи:

1. Подключить библиотеку с графикой.
2. Инициализировать программу, открыть графическое окно.
3. Отобразить треугольник на экране.
4. Реализовать вращение треугольника.
5. Создать интерфейс для изменения параметров.

Предмет и объект исследования

В данной работе выделен объект исследования - разработка компьютерных программ с графическим интерфейсом и пользовательским управлением на языках *C/C++*. Предмет исследования - язык программирования *C* и графическая библиотека *SDL2*. Объект

исследования включает в себя программное обеспечение, разрабатываемое в работе. Дополнительно, выделенный объект разработки также включает в себя алгоритмы для отрисовки равностороннего треугольника на экране, выбора точек для его построения, формулы для преобразования точек с течением времени, что реализует вращение. Объект исследования также включает в себя создание пользовательского интерфейса. В ходе работы были использованы математические формулы, как, например, перевод координат из *полярных* в *декартовые*, формула *линейной интерполяции*, формула *минимально-максимальной нормализации* и другие, что будет рассмотрено подробнее в ходе работы.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В ходе работы пришлось применить различные формулы, используя их в алгоритме для отображения треугольника на экране. Важно понимать, как работают все формулы, использованные в программе, для создания программы без ошибок.

Выбор точек для построения треугольника

В ходе работы алгоритм отображения треугольника на экране будет описан подробнее, но пока что, стоит учесть лишь то, что для построения треугольника нужны 3 точки в *декартовой системе координат* - вершины треугольника.Необходимо найти такие 3 точки, которые образуют равносторонний треугольник. Учитывая поставленную задачу, а именно вращение треугольника, было принято решение использовать выбор равноудаленных друг от друга точек на окружности для построения равностороннего треугольника. Необходимо найти 3 точки на окружности, такие что дуги между соседними точками равны 120°. Можно записать данное выражение как:

$$\forall \odot O \; \exists A, B, C \in \odot O \; : |\cup AB| = |\cup BC| = |\cup CA| = 120^\circ$$

Существует бесконечное множество наборов таких точек, однако, если считать, что одна из точек на окружности определена заранее, то существует ровно один набор таких точек, если не учитывать порядок точек. Докажем данное утверждение: пусть задана точка A, причем полярный угол точки $A = \phi(A)$. Тогда, по утверждению выше,

$$|\cup AB| = 120^\circ = \tfrac{2\pi}{3} \implies \phi(B) = \phi(A) + \tfrac{2\pi}{3}$$
$$|\cup BC| = 120^\circ = \tfrac{2\pi}{3} \implies \phi(C) = \phi(B) + \tfrac{2\pi}{3} = \phi(A) + \tfrac{4\pi}{3}$$

С изменением $\phi(A)$, полученные наборы точек образуют треугольник,

повернутый на $\phi(A)$.

Переход от полярной системы координат к декартовой

Отображение треугольника производится по 3 точкам, нахождение которых определено в разделе «Выбор точек для построения треугольника». Исходя из данного раздела, вершины треугольника заданы точками A, B, C в декартовых координатах. Однако, имеются лишь формулы для полярного представления этих точек: $\phi(A), \phi(B), \phi(C)$. В таком случае нам необходимо воспользоваться формулами для перехода из полярной системы координат в декартовую:

$$x = rcos(\phi) + x_0$$
$$y = rsin(\phi) + y_0$$

где x_0, y_0 - точка центра окружности вращения, r - радиус окружности вращения. Окружность вращения будет описанной окружностью для равностороннего треугольника. Используя эти формулы в работе вычисляются координаты вершин треугольника.

Линейная интерполяция

Для установки значения скорости вращения (далее - ω), а также размеров треугольника, а именно радиуса описанной вокруг него окружности (далее - r), необходимо воспользоваться формулой линейной интерполяции. Такая формула имеет 3 параметра: x_{min}, x_{max}, x_p, где x_{min} - минимальное значение функции, x_{max} - максимальное значение функции, x_t - степень интерполяции на отрезке $[0; 1]$. Задача функции заключается в том, чтобы найти значение x, которому соответствует значение функции при данной степени интерполяции x_t, так $x = x_{min}$ при $x_t = 0$ и $x = x_{max}$ при $x_t = 1$. Так выглядит формула линейной интерполяции для данной

программы:

$$Lerp(x_{min}, x_{max}, x_t) = x_t * (x_{max} - x_{min}) + x_{min}$$

Минимально-максимальная нормализация

Функция минимально-максимальной нормализации противоположна по смыслу функции линейной интерполяции. Данная функция принимает 3 параметра: x_{min}, x_{max}, x - минимальное значение функции, x_{max} - максимальное значение функции, x - текущее значение функции на отрезке $[x_{min}; x_{max}]$. Задача функции заключается в том, чтобы найти степень интерполяции x_t, которому соответствует текущее значение функции x, так $x_t = 0$ при $x = x_{min}$ и $x_t = 1$ при $x = x_{max}$. Так выглядит функция минимально-максимальной нормализации для данной программы:

$$MinMaxNorm(x_{min}, x_{max}, x) = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Данная формула будет использована в программе для определения значений ползунков на отрезке $[0; 1]$. Таким образом, ползунок в крайнем левом положении должен приводить к значению 0, а в крайнем правом 1.

Устранение зависимости от частоты кадров

В ходе работы задается вращение равностороннего треугольника, путем добавления в каждой итерации программы к углу поворота треугольника ϕ какого-то значения $\Delta\phi$, зависящего от скорости ω. Однако, если такое значение $\Delta\phi(\omega) = const$, то при увеличении частоты кадров в λ в α раз, скорость вращения треугольника пропорционально увеличится в α раз. Такое поведение программы является нежеланным, ведь разные пользователи будут наблюдать различную скорость вращения. Чтобы устранить данный недостаток, необходимо умножать скорость вращения ω на значение в секундах с момента прошлой итерации, Δt. Стоит заметить,

что, $\lambda \approx \frac{1}{\Delta t}$. В таком случае скорость вращения вычисляется как

$$\lim_{\lambda \to +\infty} \sum_{i=1}^{\lambda} \omega * \frac{1}{\lambda} = \lim_{\lambda \to +\infty} \sum_{i=1}^{\lambda} \omega * \Delta t =$$

$$\lim_{\lambda \to +\infty} \omega * \Delta t * \lambda = \lim_{\lambda \to +\infty} \omega * \frac{1}{\lambda} * \lambda = \lim_{\lambda \to +\infty} \omega = \omega$$

Использование такого метода вычисления скорости вращения ω гарантирует ее одинаковость вне зависимости от количества кадров в секунду λ.

Использованные библиотеки

В ходе работы были использованы как встроенные, так и сторонние библиотеки. Так, например, из встроенных библиотек использованы:

1. math.h - для работы с математическими функциями, как *cos*, *sin*.
2. stdbool.h - для использования булевой переменной (true/false)
3. stdint.h - для использования других целочисленных типов

Была использована сторонняя библиотека *SDL.h* для работы с графическим окном, отображения графики, обработки нажатий, и библиотека *SDL_ttf.h* для отображения текста. Данную библиотеку можно установить из официальных источников, в списке использованной литературы будет приложена ссылка на репозиторий данной библиотеки на *GitHub*, откуда ее можно загрузить.

ХОД РАБОТЫ

Создание графического окна

Для начала необходимо подключить все используемые в программе библиотеки.

```
#include "math.h"
#include "SDL2/SDL.h"
#include "stdbool.h"
#include "stdint.h"
#include "stdio.h"
```

Для удобства процесса разработки, а также простоты чтения кода другими людьми, было принято решение разбить программу на подпрограммы, каждая из которых находится в отдельном заголовочном файле и отвечает за определенные функции. Подключаются эти заголовочные файлы:

```
#include "texts.h"
#include "sliders.h"
#include "inputs.h"
```

Далее рассмотрен заголовочный файл `constants.h`, в котором находятся необходимые константы, как, например, ширина и высота окна, цвет заднего фона окна и другие.

```
// constants.h
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const SDL_Color BACKGROUND_COLOR = {0x3b, 0x3e, 0x4f, 0xff};
const SDL_Color TRIANGLE_COLORS[3] = {
  {0xff, 0, 0, 0xff},
  {0, 0, 0xff, 0xff},
  {0, 0xff, 0, 0xff}
};
```

Далее необходимо определить директиву `SDL_MAIN_HANDLED` для корректной работы программы. Это важно сделать *до* подключения библиотеки *SDL2*.

```
// main.c
#define SDL_MAIN_HANDLED
```

Необходимо инициализировать глобальные переменные, смысл которых будет оговорен далее в ходе работы.

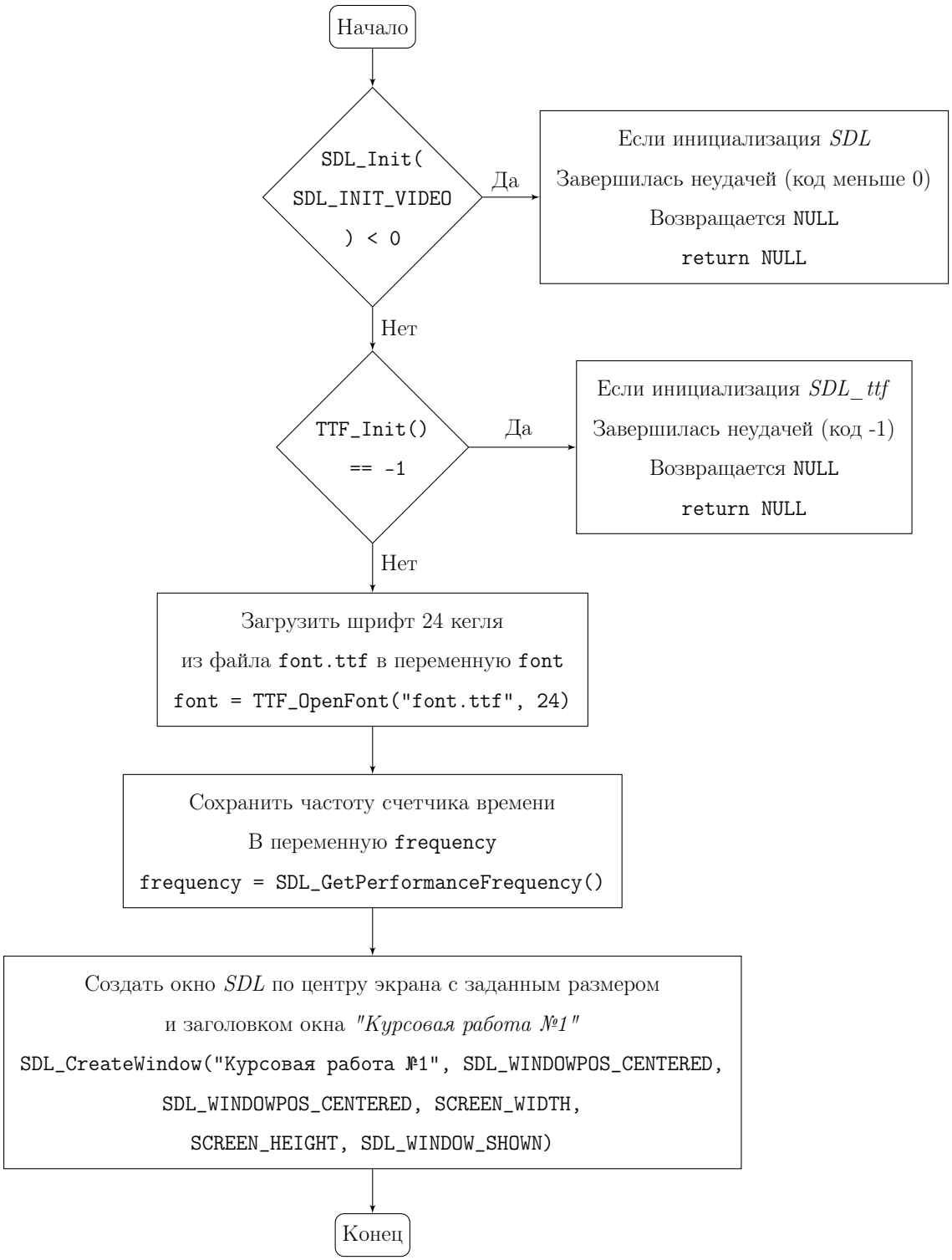
```
// main.c
bool run = true;
uint64_t frequency, lastTick;
```

Создается функция `InitSDL`, которая отвечает за инициализацию графической библиотеки *SDL2*, библиотеки для отображения текста *SDL_ttf*, открытие шрифта и установку переменной `frequency`, хранящей значение частоты счетчика, который будет использоваться для получения времени, прошедшего с прошлой итерации программы Δt в соответствии с главой «Устранение зависимости от частоты кадров».

```
// main.c
SDL_Window* InitSDL() {
  if (SDL_Init(SDL_INIT_VIDEO) < 0)
    return NULL;

  if (TTF_Init() == -1)
    return NULL;

  font = TTF_OpenFont("font.ttf", 24);
  frequency = SDL_GetPerformanceFrequency();

    return SDL_CreateWindow(
      "Курсовая работа №1",
      SDL_WINDOWPOS_CENTERED,
      SDL_WINDOWPOS_CENTERED,
      SCREEN_WIDTH,
      SCREEN_HEIGHT,
      SDL_WINDOW_SHOWN
    );
}
```

Функция `SDL_Init(SDL_INIT_VIDEO)` инициализирует библиотеку *SDL2* и настраивает ее на работу в видео-режиме, что не подключает модули для работы со звуком и другие ненужные функции. Функция `TTF_Init` инициализирует библиотеку *SDL_ttf* для работы с текстом и шрифтами формата *ttf*. Открывается заранее загруженный в директорию с программой шрифт с помощью `TTF_OpenFont("font.ttf", 24)`, где `24` - размер шрифта. Задается частота счетчика `frequency`. `SDL_CreateWindow`

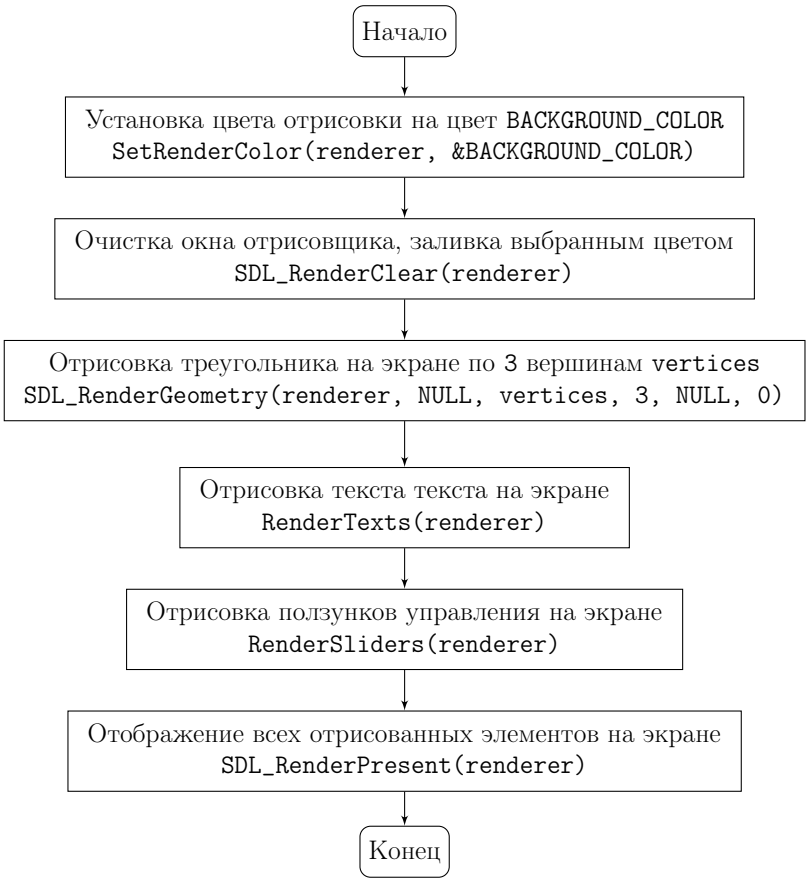
создает окно программы, где `Курсовая работа №1` - заголовок окна. Следующие 2 аргумента определяют изначальное положение окна на экране, в данном случае окно открывается по центру с помощью `SDL_WINDOWPOS_CENTERED`. Следующие 2 аргумента отвечают за ширину и высоту окна, это константы `SCREEN_WIDTH` и `SCREEN_HEIGHT` заданные заранее. Ниже представлена блок схема функции InitSDL.



Далее задается функция для отрисовки всех графических элементов и отображения их на экране. Некоторые функции объявлены в других файлах и будут описаны позже.

```
// main.c
void Render(SDL_Renderer* renderer) {
  SetRenderColor(renderer, &BACKGROUND_COLOR);
  SDL_RenderClear(renderer);
  SDL_RenderGeometry(renderer, NULL, vertices, 3, NULL, 0);
  RenderTexts(renderer);
  RenderSliders(renderer);
  SDL_RenderPresent(renderer);
}
```

Ниже представлена блок-схема для функции `Render`.



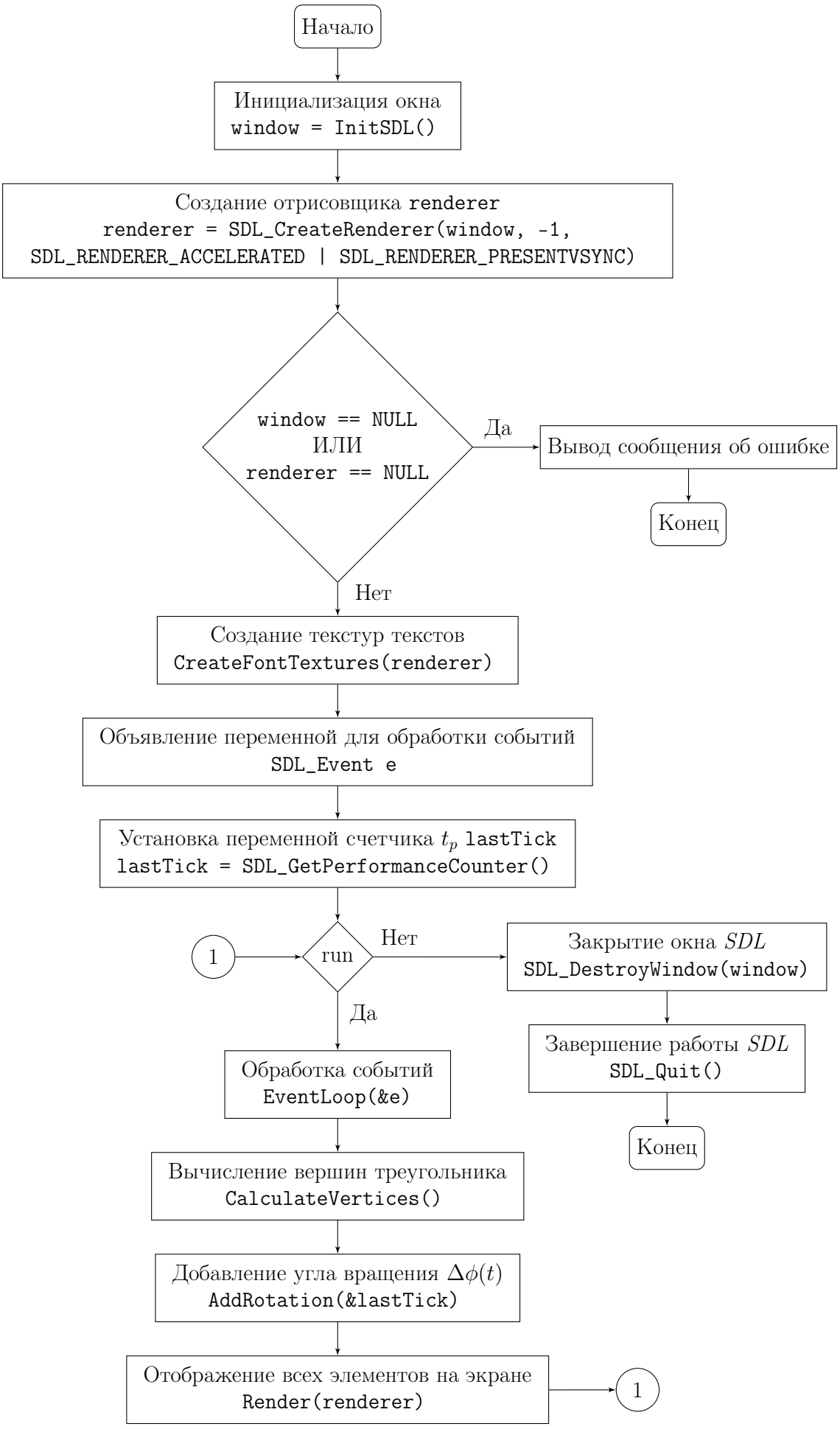
`SetRenderColor` - функция, которая устанавливает текущий цвет на заранее выбранный цвет заднего фона программы `BACKGROUND_COLOR`, эта функция рассмотрена в разделе «Вращение треугольника», а `SDL_RenderClear` очищает ранее отрисованные элементы и заливает окно текущим выбранным цветом. `SDL_RenderGeometry` - функция для

отрисовки любой геометрической фигуры по набору вершин. В качестве одного из аргументов передается массив типа `SDL_Vertex[]`, который содержит все вершины фигуры, а также число, количество вершин. В этом случае отрисовывается треугольник, поэтому в качестве аргумента передается `3` как число вершин. Также вызываются собственные функции для отрисовки текста и ползунков управления. После этого происходит отображение отрисованных элементов на экране с помощью функции `SDL_RenderPresent`.

Далее рассматривается функция `main`, с которой начинается работа программы.

```
int main(int argc, char* argv[]) {
    SDL_Window* window = InitSDL();
    SDL_Renderer* renderer = SDL_CreateRenderer(
      window,
      -1,
      SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC
    );

    if (window == NULL || renderer == NULL) {
      printf("SDL error: %s\n", SDL_GetError());
      return 1;
    }

    CreateFontTextures(renderer);

    SDL_Event e;
    lastTick = SDL_GetPerformanceCounter();

    while (run) {
      EventLoop(&e);
      CalculateVertices();
      AddRotation(&lastTick);
      Render(renderer);
    }

    SDL_DestroyWindow(window);
    SDL_Quit();

  return 0;
}
```

Далее представлена блок схема функции `main`.


Начало
Инициализация окна
window = InitSDL()
Создание отрисовщика renderer
renderer = SDL_CreateRenderer(window, -1,
SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC)
window == NULL
ИЛИ
renderer == NULL
Да
Вывод сообщения об ошибке
Конец
Нет
Создание текстур текстов
CreateFontTextures(renderer)
Объявление переменной для обработки событий
SDL_Event e
Установка переменной счетчика t_p lastTick
lastTick = SDL_GetPerformanceCounter()
1
run
Нет
Закрытие окна SDL
SDL_DestroyWindow(window)
Да
Завершение работы SDL
SDL_Quit()
Конец
Обработка событий
EventLoop(&e)
Вычисление вершин треугольника
CalculateVertices()
Добавление угла вращения $\Delta\phi(t)$
AddRotation(&lastTick)
Отображение всех элементов на экране
Render(renderer)
1

Используются заданные ранее функции для инициализации окна, также создается переменная-отрисовщик `renderer`, куда передаются флаги `SDL_RENDERER_ACCELERATED`, который сообщает программе, при возможности, использовать аппаратное ускорение, а также флаг `SDL_RENDERER_PRESENTVSYNC`, задающий режим *вертикальной синхронизации* для окна, что ограничит частоту кадров до текущей частоты обновления экрана компьютера. Таким образом, не будет потрачено лишних ресурсов компьютера на создание кадров, которые не увидит пользователь.

При ошибке в процессе инициализации программа выведет информацию об ошибке в консоль и завершит работу. Далее вызывается функция `CreateFontTextures`, которая создаст текстуры с текстом, данная функция описана в разделе «Отображение текста». Создается переменная типа `SDL_Event`, в которую будет записываться информация о событиях программы, таких как действия пользоваля, нажатия кнопок мыши. Задается переменная `lastTick`, в которой будет храниться значение счетчика в прошлой итерации программы, изначально заданная как текущее значение счетчика, которое можно получить с помощью функции `SDL_GetPerformanceCounter`.

Далее запускается основной цикл программы. Сначала запускается обработчик событий с помощью функции `EventLoop`, которая описана в разделе «Обработка событий». Происходит вычисление вершин треугольника на данный момент, с помощью функции `CalculateVertices`, которая описана в разделе «Вращение треугольника». Увеличивается текущий угол поворота треугольника относительно первой вершины, $\phi(A)$ с помощью метода `AddRotation`, который описан в разделе «Вращение треугольника». Последним вызовом в теле цикла является функция отрисовки на экране `Render`, описанная в данном разделе ранее. При завершении цикла необходимо закрыть окно с помощью `SDL_DestroyWindow`

и запускить функцию, производящую завершение работы подсистем *SDL2*, `SDL_Quit`.

Вращение треугольника

В заголовочном файле `triangle.h` находятся основные переменные, отвечающие за треугольник, в том числе скорость вращения треугольника, измеряющаяся в радианах в секунду, радиус описанной около треугольника окружности, текущее значение угла поворота первой вершины $\phi(A)$, а также массив типа `SDL_Vertex[]`, содержащий информацию о вершинах треугольника, а именно их положение на экране в *декартовой системе координат* и цвет вершины.

```
// triangle.h
double speed = M_PI / 2; // Радианы в секунду
double radius = 100;
double rotation;
SDL_Vertex vertices[3];
```

Также, для удобства была создана функция `SetRenderColor`, устанавливающая цвет отрисовщика на экране на необходимый цвет, передаваемый как `SDL_Color`.

```
// triangle.h
void SetRenderColor(SDL_Renderer* renderer, const SDL_Color*
   color) {
  SDL_SetRenderDrawColor(
    renderer,
    color->r,
    color->g,
    color->b,
    color->a
  );
}
```

Далее рассмотрен заголовочный файл `calculations.h`, в котором находятся функции для необходимых вычислений в программе. Для начала необходимо подключить заголовочный файл `triangle.h`, описанный в данном разделе выше. Задается переменная счетчика, которая используется

в соответствии с разделом «Создание графического окна».

```
// calculations.h
#include "triangle.h"

extern uint64_t frequency;
```

Создается функция `Lerp` для *линейной интерполяции*, которая описана в разделе «Линейная интерполяция».

```
// calculations.h
double Lerp(double x, double min, double max) {
  if (x <= 0)
    return min;

  if (x >= 1)
    return max;

  return x * (max - min) + min;
}
```

Единственным отличием данной функции от той, заданной формулой в разделе «Линейная интерполяция», это дополнительные условия проверки на x_t, не находящийся на отрезке $[0; 1]$, в случае чего возвращается соответственно либо минимальное, либо максимальное значение.

Далее рассмотрена функция `InverseLerp`, реализующая формулу из раздела «Минимально-максимальная нормализация».

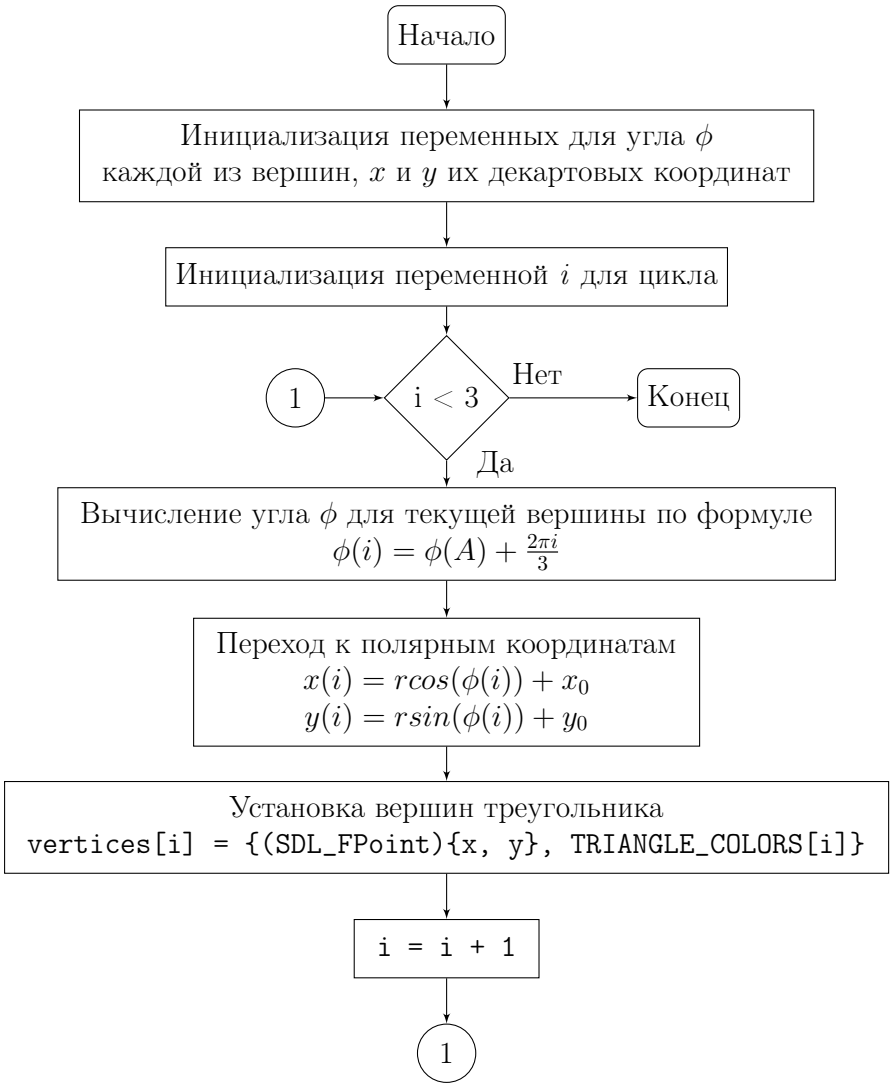
```
// calculations.h
double InverseLerp(double x, double min, double max) {
  if (x <= min)
    return 0;

  if (x >= max)
    return 1;

  return (x - min) / (max - min);
}
```

Как и в функции *линейной интерполяции*, отличием является дополнение формулы, описанной в разделе «минимально-максимальная нормализация», проверками на x. В случае, если он меньше x_{min}, возвращается 0, а если больше x_{max}, возвращается 1.

Далее рассмотрена функция `CalculateVerticies` для нахождения *полярного* угла ϕ для каждой точки A, B, C, а также преобразования их в *декартову систему координат.*

```
// calculations.h
void CalculateVertices() {
  double angle, x, y;

  for (int i = 0; i < 3; ++i) {
    angle = rotation + i * (2 * M_PI / 3);
    x = cos(angle) * radius + SCREEN_WIDTH / 2;
    y = sin(angle) * radius + SCREEN_HEIGHT / 2;
    vertices[i] = (SDL_Vertex){(SDL_FPoint){x, y},
   TRIANGLE_COLORS[i]};
  }
}
```

Ниже представлена блок-схема алгоритма функции `CalculateVertices`.



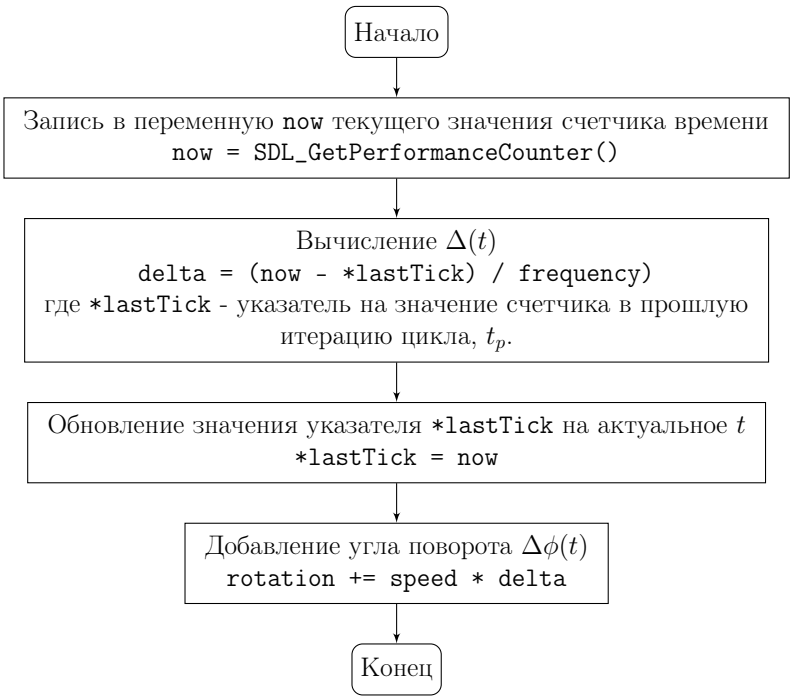
Сначала находится *полярный* угол каждой точки, пусть каждая

вершина A, B, C имеет свой индекс k, начиная с 0. Так, *полярный* угол вершин можно записать как $\phi(k) = \phi(A) + 2\pi rk$, где r - радиус описанной около равностороннего треугольника окружности. Имея *полярный* угол вершин треугольника, его можно перевести в *декартову систему координат* по формулам, описанным в разделе «Переход от полярной системы координат к декартовой». Задается каждая из вершин массива с вершинами `vertices`, используя полученные координаты, а также цвет вершины из заголовочного файла с константами `constants.h`.

Далее рассмотрена функция `AddRotation` для добавления необходимого угла $\Delta\phi$ к углу $\phi(A)$ за одну итерацию программы.

```
void AddRotation(uint64_t* lastTick) {
  uint64_t now = SDL_GetPerformanceCounter();
  double delta = (double)(now - *lastTick) / frequency;
  *lastTick = now;
  rotation += speed * delta;
}
```

Ниже представлена блок схема функции `AddRotation`.



Сначала получается значение счетчика и записывается в переменную `now`, пусть это будет t. Если t_p - значение счетчика в предыдущую итерацию

программу, а γ - частота счетчика, хранимая в переменной `frequency`, то Δt, время между итерациями программы, вычисляется по формуле:

$$\Delta t = \frac{t - t_p}{\gamma}$$

Вычислив Δt, увеличивается текущий полярный угол поворота точки A $\phi(A)$ на $v * \Delta t$, где v - заданная в заголовочном файле `constants.h` константа - скорость вращения, измеряемая в радианах в секунду. Вычисления в этом разделе ссылаются на формулы и расчеты из раздела «Устранение зависимости от частоты кадров». Ниже представлена блок схема функции `AddRotation`.

Отображение ползунков

Для начала необходимо подключить заголовочный файл с расчетами `sliders.h`.

```
// sliders.h
#include "calculations.h"
```

Далее определены константы отрисовки ползунков, такие как их ширина, цвет и отступы.

```
// sliders.h
const int SLIDER_PADDING = 150;
const int SLIDER_OFFSET = 50;
const int SLIDER_WIDTH = 5;
const int HANDLE_WIDTH = 16;
const SDL_Color SLIDER_COLOR = {0x16, 0x16, 0x16, 0xff};
const SDL_Color HANDLE_COLOR = {0xff, 0xff, 0xff, 0xff};
const int slidersAmount = 2;
```

Для удобства была создана структура `Slider`, в которой находятся все специфичные для ползунка данные.

```
// sliders.h
typedef struct {
  const int index;
  const double min;
  const double max;
```

```
  double* value;
  SDL_Rect rect;
  bool dragged;
} Slider;
```

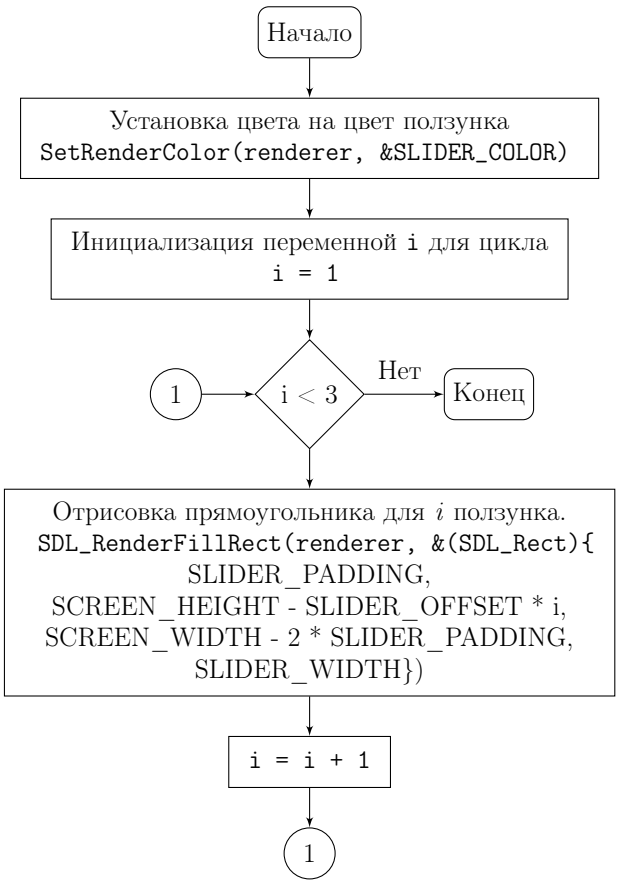
Далее созданы переменные типа структуры `Slider`, а также создан массив `sliders[]`, в который занесены оба ползунка.

```
// sliders.h
const Slider radiusSlider = {1, 10, 200, &radius};
const Slider speedSlider = {2, -2 * M_PI, 2 * M_PI, &speed};
Slider sliders[] = {radiusSlider, speedSlider};
```

Для начала определена функция `RenderSliderBars`, которая отрисовывает основную часть ползунков, их рабочий диапазон.

```
// sliders.h
void RenderSliderBars(SDL_Renderer* renderer) {
  SetRenderColor(renderer, &SLIDER_COLOR);

  for (int i = 1; i < 3; ++i) {
    SDL_RenderFillRect(renderer, &(SDL_Rect){
      SLIDER_PADDING,
      SCREEN_HEIGHT - SLIDER_OFFSET * i,
      SCREEN_WIDTH - 2 * SLIDER_PADDING,
      SLIDER_WIDTH
    });
  }
}
```

Ниже представлена блок схема функции `RenderSliderBars`.

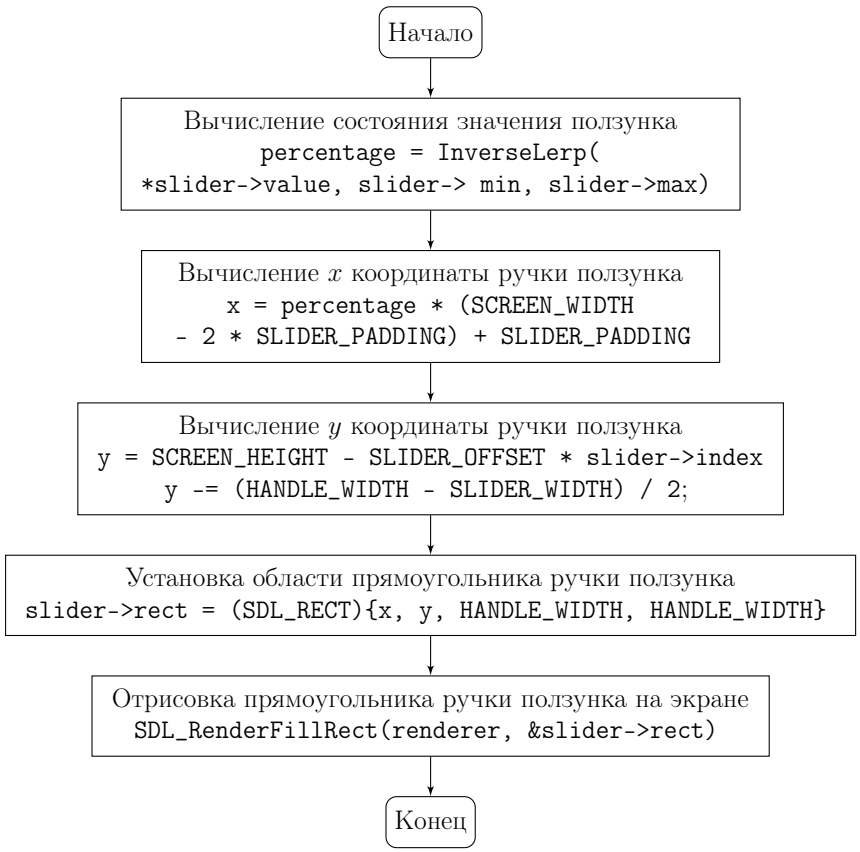


Данная функция выполняет отрисовку прямоугольников для всего количества ползунков. Имеется 2 ползунка, поэтому происходит отрисовка двух ползунков. Используются ранее определенные константы, каждый ползунок располагается немного выше другого на экране.

Далее рассмотрена функция `RenderHandle`, отрисовывающая ручку управления ползунком на экране в зависимости от значения, которым управляет ползунок.

```
// sliders.h
void RenderHandle(SDL_Renderer* renderer, Slider* slider) {
  double percentage = InverseLerp(*slider->value, slider->min,
   slider->max);
  int x = percentage * (SCREEN_WIDTH - 2 * SLIDER_PADDING) +
   SLIDER_PADDING;
  int y = SCREEN_HEIGHT - SLIDER_OFFSET * slider->index;
  y -= (HANDLE_WIDTH - SLIDER_WIDTH) / 2;
  slider->rect = (SDL_Rect){x, y, HANDLE_WIDTH, HANDLE_WIDTH};
  SDL_RenderFillRect(renderer, &slider->rect);
}
```

Ниже представлена блок схема функции `RenderHandle`.

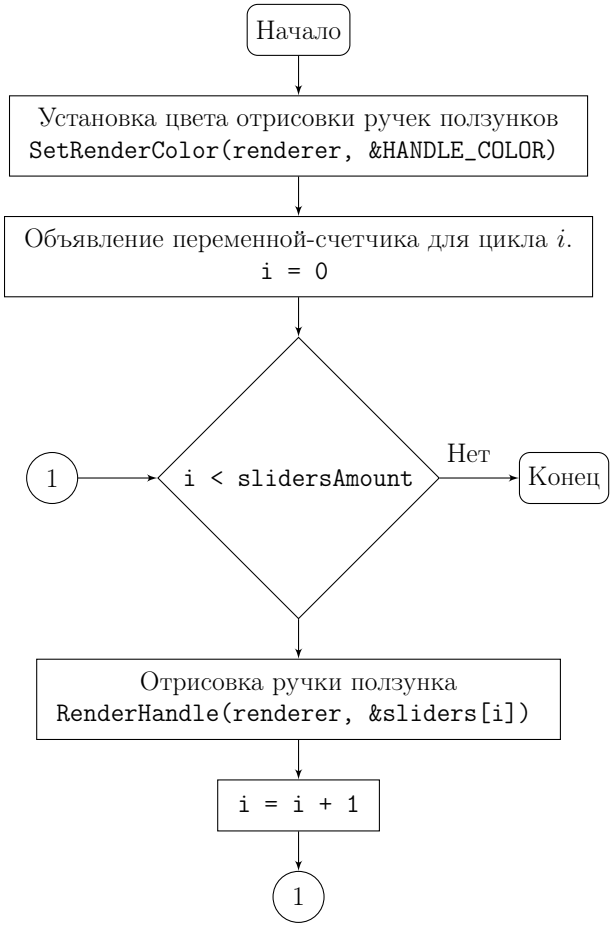


Сначала вычисляется нормализованное значение ползунка с помощью фукцнии, описанной в разделе «Минимально-максимальная нормализация». Далее вычисляется положение ручки на экране по оси x, путем умножения ширины ползунка на его нормализованное значение и добавления отступа слева. Затем вычисляется значение ручки по оси y на основе индекса ползунка, как оно вычислялось при отрисовке ползунков. Затем из положения ручки по оси y вычитается $\frac{w_h - w_s}{2}$, где w_h - ширина ручки, заданная константой `HANDLE_WIDTH`, а w_s - ширина ползунка, заданная константой `SLIDER_WIDTH`. Это необходимо для того, чтобы ручка была визуально центрирована относительно ползунка по высоте. Затем создается прямоугольник `SDL_Rect` и устанавливается на поле `rect` данного ползунка. После этого вызывается функция отрисовки прямоугольника `SDL_RenderFillRect`, куда передаются данные о созданном прямоугольнике.

Далее рассмотрена функция `RenderSliderHandles`, созданная для удобства отрисовки ручек управления ползунками.

```
// sliders.h
void RenderSliderHandles(SDL_Renderer* renderer) {
  SetRenderColor(renderer, &HANDLE_COLOR);

  for (int i = 0; i < slidersAmount; ++i) {
    RenderHandle(renderer, &sliders[i]);
  }
}
```

Ниже представлена блок схема функции `RenderSliderHandles`.

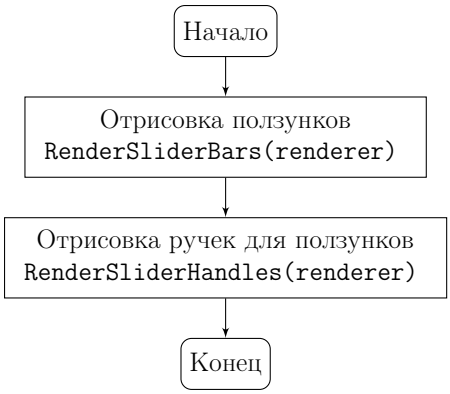


Изначально устанавливается цвет отрисовки ручек по заданной константе `HANDLE_COLOR` с помощью описанной ранее в разделе «Вращение треугольника» функции `SetRenderColor`. Затем, для каждого ползунка происходит вызов описанной ранее функции `RenderHandle` для отрисовки его ручки.

Также, создана функция `RenderSliders`, которая вызывает все необходимые функции отрисовки, то есть отрисовку ползунков и отрисовку ручек.

```
// sliders.h
void RenderSliders(SDL_Renderer* renderer) {
  RenderSliderBars(renderer);
  RenderSliderHandles(renderer);
}
```

Ниже представлена блок схема функции `RenderSliders`.



Обработка событий

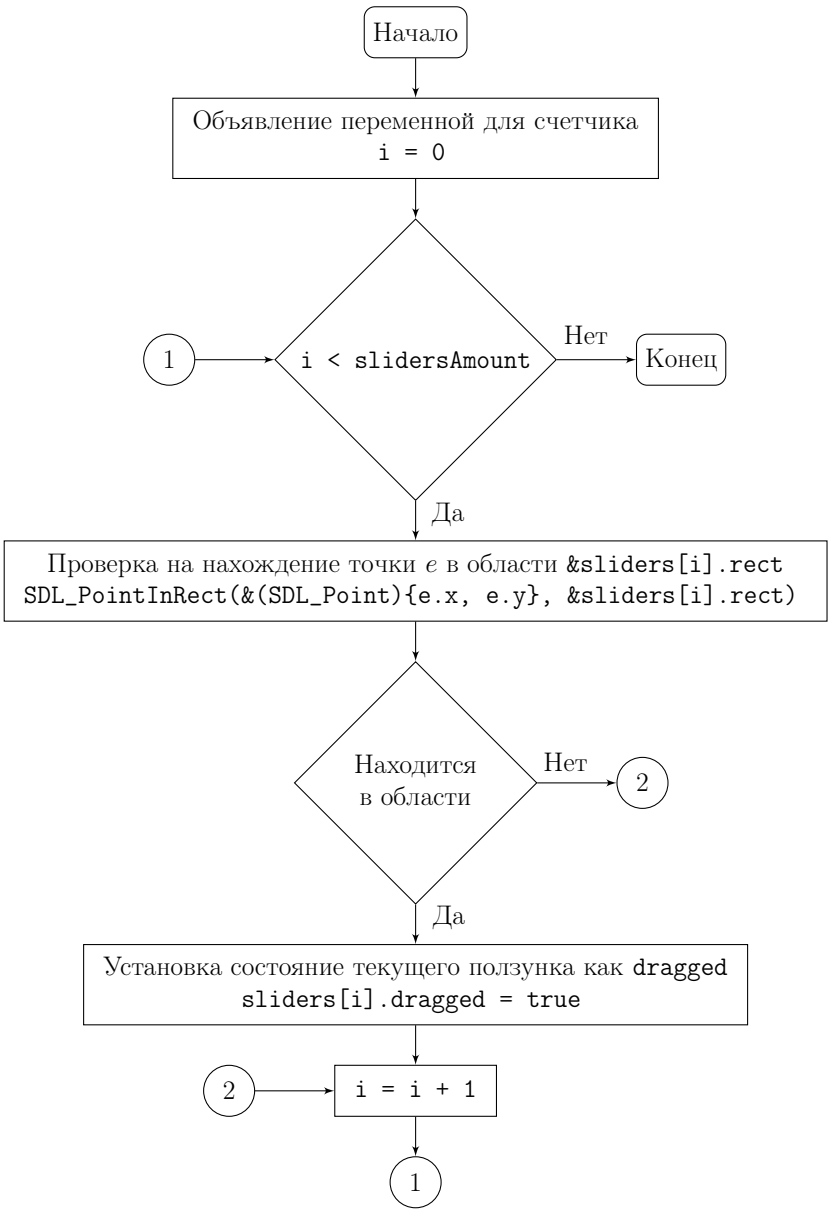
Для обработки события завершения работы программы необходимо хранить состояние работы программы. Эта переменная обновляется при закрытии пользователем программы и уверяет благополучное завершение программы.

```
// inputs.h
extern bool run;
```

Была создана функция `MouseDown`, отвечающая за события при нажатии на кнопку мыши.

```
// inputs.h
void MouseDown(SDL_MouseButtonEvent e) {
  for (int i = 0; i < slidersAmount; ++i) {
    if (SDL_PointInRect(&(SDL_Point){e.x, e.y}, &sliders[i].rect)
   )
      sliders[i].dragged = true;
  }
}
```

Ниже представлена блок схема функции `MouseDown`.



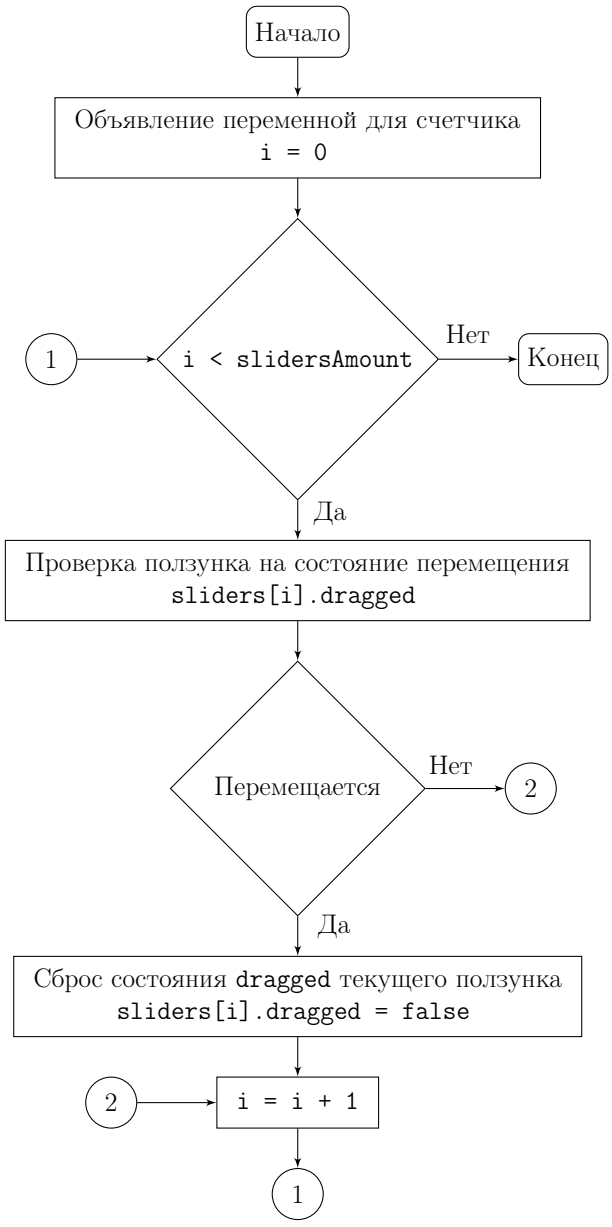
Функция проверяет каждый ползунок, и, если точка нажатия мыши с координатами `e.x, e.y` находится внутри прямоугольника, заданного как `rect` для каждого ползунка в разделе «Отображение ползунков», то данная функция задает этому ползунку состояние `dragged` как `true`, то есть ползунок сейчас двигается.

Далее, определена функция `MouseUp`, которая отвечает за события при поднятии кнопки мыши.

```
// inputs.h
void MouseUp(SDL_MouseButtonEvent e) {
  for (int i = 0; i < slidersAmount; ++i) {
    if (sliders[i].dragged)
      sliders[i].dragged = false;
```

```
    }
}
```

Ниже представлена блок схема функции `MouseUp`.

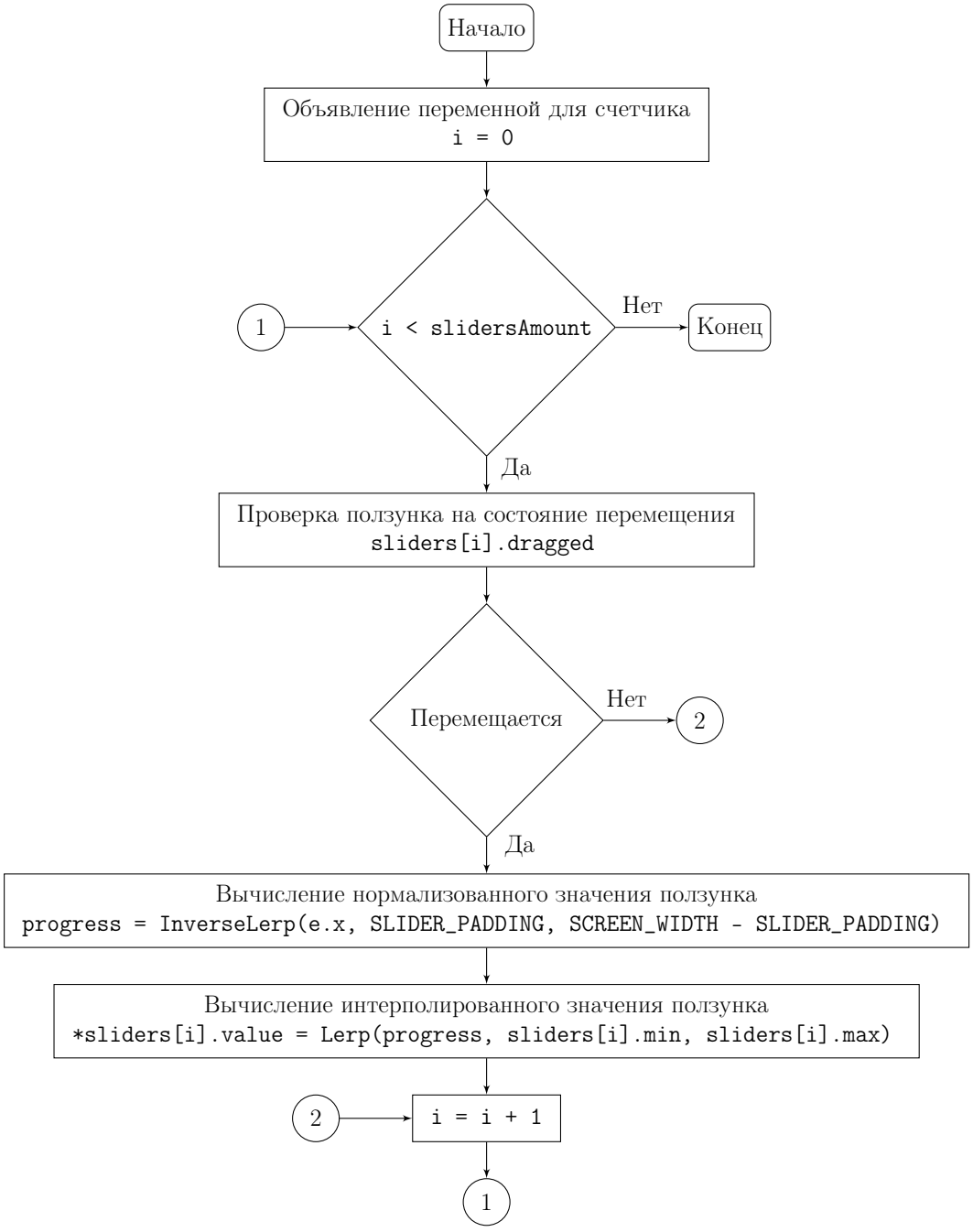


В данном методе проверяется, двигается ли каждый ползунок в момент поднятия кнопки мыши с помощью состояния `dragged`, которое задается в функции `MouseDown`. Если же какой-то из ползунков двигается пользователем, то при поднятии кнопки мыши его состояние `dragged` меняется на `false`, то есть движение пользователем заканчивается.

Далее рассмотрена функция `UpdateSliderValues` из заголовочного файла `sliders.h`.

```
// sliders.h
void UpdateSliderValues(SDL_MouseMotionEvent e) {
  for (int i = 0; i < slidersAmount; ++i) {
    if (sliders[i].dragged) {
      double progress = InverseLerp(e.x, SLIDER_PADDING,
   SCREEN_WIDTH - SLIDER_PADDING);
      *sliders[i].value = Lerp(progress, sliders[i].min, sliders[
   i].max);
    }
  }
}
```

Ниже представлена блок схема функции `UpdateSliderValues`.



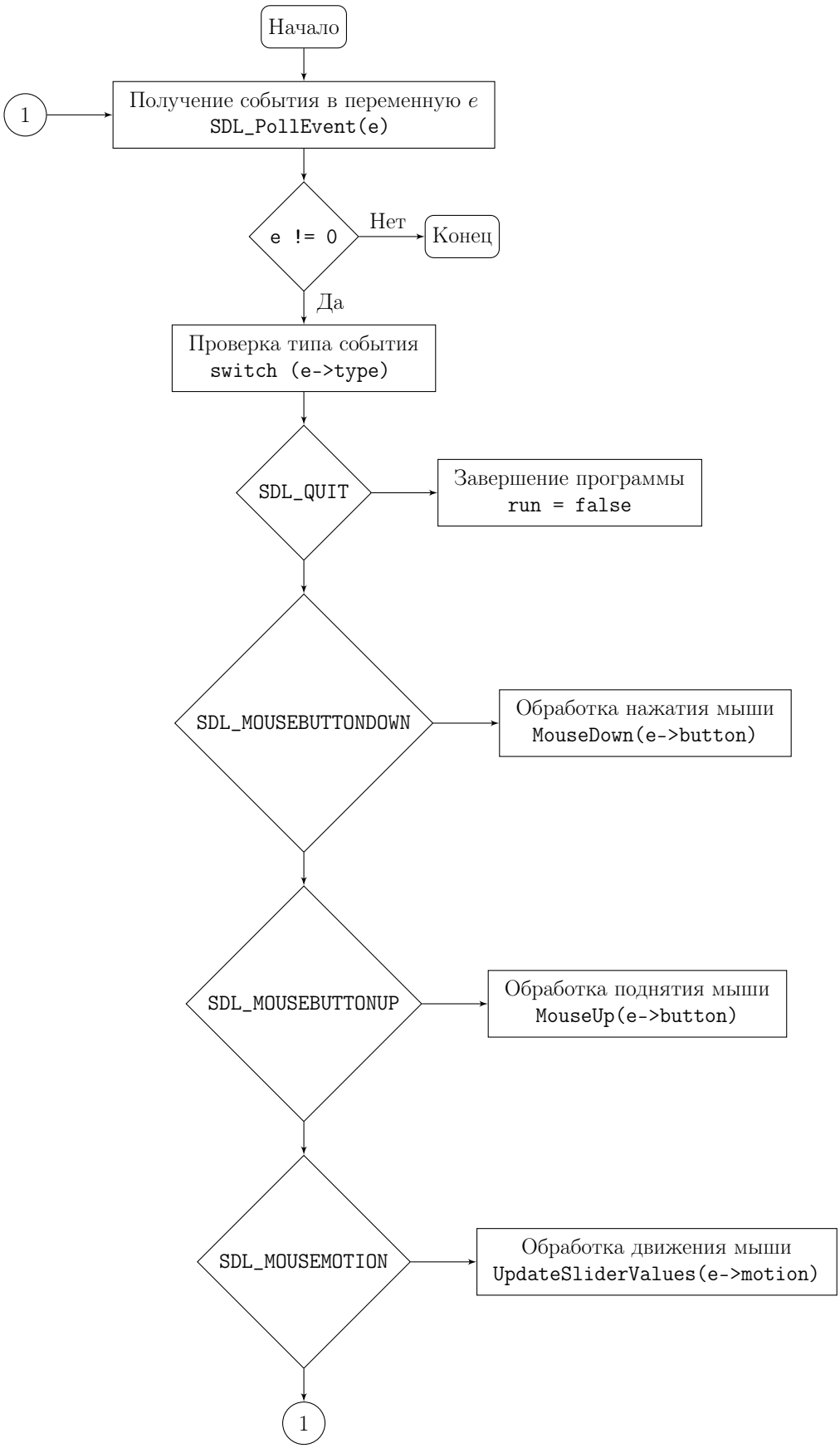
В данной функции происходит обновление значения ползунка, которое

вычисляется по формуле из раздела «Минимально-максимальная нормализация», и задается как поле `value` для каждого ползунка из массива `sliders[]`. Причем, обновление значения ползунка происходит только, если ползунок имеет активное состояние `dragged`.

Далее рассмотрена функция `EventLoop`, в которой происходит обработка всех событий типа `SDL_Event`.

```
// inputs.h
void EventLoop(SDL_Event* e) {
    while(SDL_PollEvent(e) != 0)
    {
      switch (e->type) {
      case SDL_QUIT:
        run = false;
        break;

      case SDL_MOUSEBUTTONDOWN:
        MouseDown(e->button);
        break;

      case SDL_MOUSEBUTTONUP:
        MouseUp(e->button);
        break;

      case SDL_MOUSEMOTION:
        UpdateSliderValues(e->motion);
        break;
      }
  }
}
```

Ниже представлена блок схема функции `EventLoop`.



Данная функция проверяет все события с помощью цикла `while`, получая их из функции `SDL_PollEvent`. Далее, в зависимости от типа

события `e->type`, принимается какое-либо действие. В случае выхода из программы `SDL_QUIT`, переменная `run` устанавливается на `false`, таким образом, программа закрывается. При случае `SDL_MOUSEBUTTONDOWN`, который вызывается при нажатии на кнопки мыши, вызывается описанная ранее функция `MouseDown`, причем в качестве аргумента передается информация о событии мыши `e->button`, хранящий такую информацию, как `x` и `y` координаты нажатия. В случае события `SDL_MOUSEBUTTONUP`, которое вызывается при поднятии кнопки мыши, вызывается функция `MouseUp` описанная ранее, в которую также передается информация о мыши в качестве аргумента. В случае события движения мыши, вызывается метод `UpdateSliderValues`, но туда передается информация о движении мыши `e->motion`.

Отображение текста

Для отрисовки текста необходимо подключить библиотеку *SDL_ttf.h*, а также проинициализировать ее и загрузить шрифт, что было реализовано в функции `InitSDL` файла `main.c` и описано в разделе «Создание графического окна». В заголовочном файле, отвечающем за текст, подключены необходимые библиотеки и заголовочный файл с константами `constants.h`.

```
#include "SDL2/SDL.h"
#include "SDL2/SDL_ttf.h"
#include "constants.h"
```

Далее, для удобства, была создана структура `Text`, содержащая все параметры, необходимые для отрисовки текста.

```
typedef struct {
  const char* text;
  const int x;
  const int y;
  SDL_Color color;
  SDL_Texture* texture;
```

```
  SDL_Rect rect;
} Text;
```

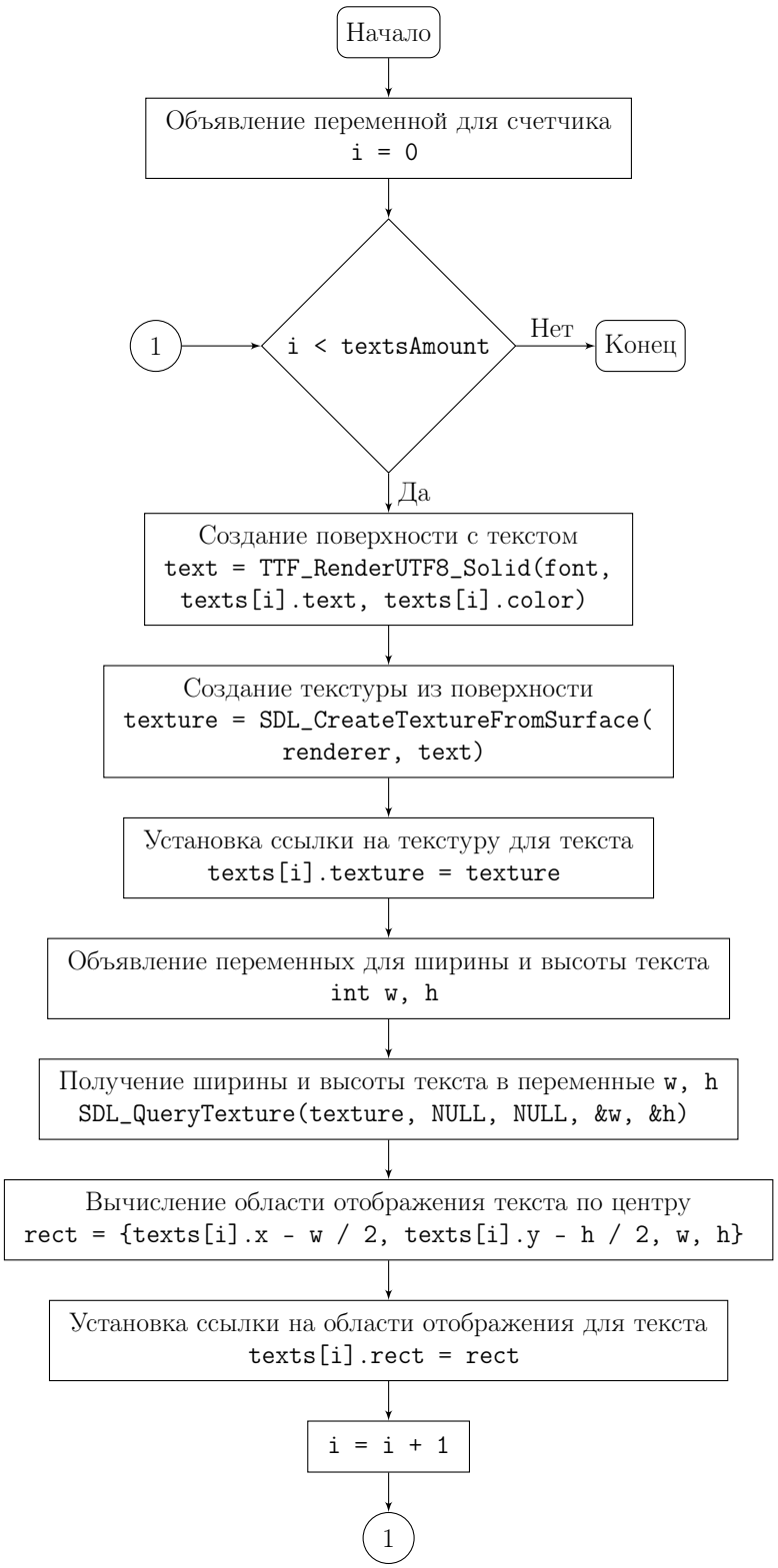
Далее создается переменная типа `TTF_Font`, куда из функции `InitSDL`, описанной ранее, загружается подключенный шрифт. Также, создается массив типа `Text[]`, в котором будут храниться все тексты для отображения, и создана переменная-счетчик количества отображаемого текста `textsAmount`.

```
TTF_Font* font;
Text texts[] = {
  {"Курсовая работа №1, вар. 3.23", SCREEN_WIDTH / 2, 50, (SDL_Color
   ){0xff, 0xff, 0xff, 0xff}},
  {"Радиус", SCREEN_WIDTH / 2, SCREEN_HEIGHT - 25, (SDL_Color){0
   xff, 0xff, 0xff, 0xff}},
  {"Скорость вращения", SCREEN_WIDTH / 2, SCREEN_HEIGHT - 75, (
   SDL_Color){0xff, 0xff, 0xff, 0xff}}
};
int textsAmount = 3;
```

Далее определяется функция `CreateFontTextures`, которая создает и задает текстуры для каждого из отображаемых текстов.

```
void CreateFontTextures(SDL_Renderer* renderer) {
  for (int i = 0; i < textsAmount; ++i) {
    SDL_Surface* text = TTF_RenderUTF8_Solid(font, texts[i].text,
    texts[i].color);
    SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer,
    text);
    texts[i].texture = texture;
    int w, h;
    SDL_QueryTexture(texture, NULL, NULL, &w, &h);
    SDL_Rect rect = {texts[i].x - w / 2, texts[i].y - h / 2, w, h
   };
    texts[i].rect = rect;
  }
}
```

Ниже представлена блок схема функции `CreateFontTextures`.



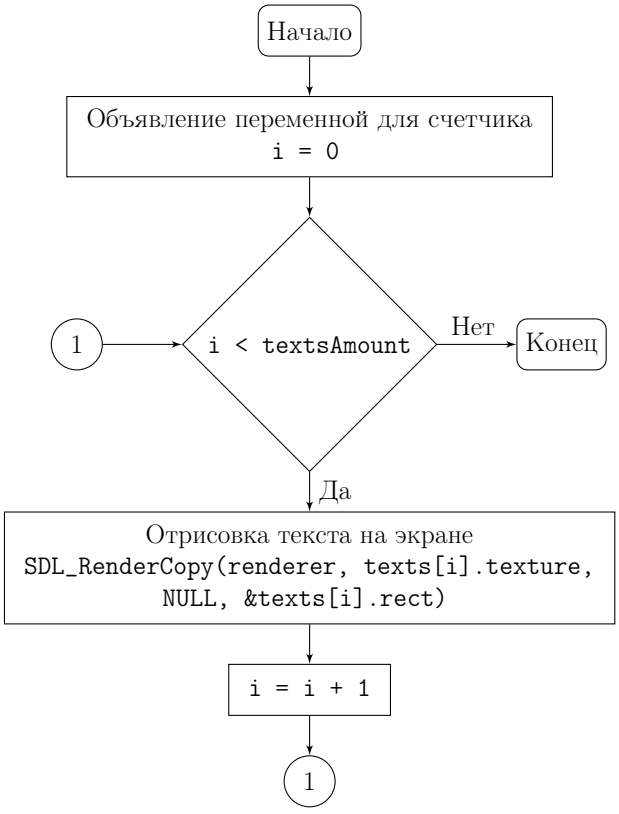
Сначала создается поверхность `SDL_Surface` с текстом `text` с помощью функции `SDL_RenderUTF8_Solid`, используя поля из структуры `Text` для данного текста. Далее эта поверхность конвертируется в текстуру с помощью функции `SDL_CreateTextureFromSurface` и задается в соответственное поле структуры `Text.texture`. Далее определяется ширина

и высота созданного текста с помощью функции `SDL_QueryTexture`, куда передается указатель на переменные ширины и высоты `&w`, `&h`. Создается прямоугольная область `SDL_Rect`, откуда берется положение текста на экране. Туда передается `x`, `y` координаты текста, а также его ширина и высота. Для визуального центрирования текста относительно заданных в структуре `Text` координат `x` и `y`, из их положения вычитается половина ширины и высоты отрисованного текста соответственно. Ширина и высота передается без изменений, а созданная область задается в поле `Text.rect`.

Далее рассмотрена функция `RenderTexts`, которая отрисовывает все созданные тексты на экране.

```
1 void RenderTexts(SDL_Renderer* renderer) {
2   for (int i = 0; i < textsAmount; ++i) {
3     SDL_RenderCopy(renderer, texts[i].texture, NULL, &texts[i].
    rect);
4   }
5 }
```

Ниже представлена блок схема функции `RenderTexts`.



Так как все тексты отрисованы в текстуры, их остается лишь

отобразить на экране в необходимый момент. Отображение текстуры на экран производится с помощью функции `SDL_RenderCopy`, куда необходимо передать отрисовщик типа `SDL_Renderer`, текстуру для отрисовки, которая в нашем случае хранится в поле `Text.texture`, а также прямоугольную область текстуры, которая находится в поле `Text.rect`.

ВЫВОД

В ходе данной курсовой работы был изучен принцип создания программ с графическим интерфейсом на языке *C* с использованием библиотеки *SDL2*. Была разработана программа для отображения на экране компьютера вращающегося равностороннего треугольника. Предусмотрена возможность изменения скорости вращения треугольника, а также его размеров, что реализовано с помощью управления ползунками, что является простым, удобным и интуитивно понятным для пользоваля интерфейсом.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Документация SDL2 / [Электронный ресурс] // SDL2 Wiki : [сайт]. — URL: `https://wiki.libsdl.org/SDL2/FrontPage` (дата обращения: 20.05.2024).

2. Обучающие пособия SDL2 / [Электронный ресурс] // Lazy Foo' Productions : [сайт]. — URL: `https://www.lazyfoo.net/tutorials/SDL/` (дата обращения: 20.05.2024).

3. Исходный код SDL2 / [Электронный ресурс] // GitHub : [сайт]. — URL: `https://github.com/libsdl-org/SDL/tree/SDL2` (дата обращения: 20.05.2024).

4. Исходный код SDL_ttf / [Электронный ресурс] // GitHub : [сайт]. — URL: `https://github.com/libsdl-org/SDL_ttf` (дата обращения: 20.05.2024).

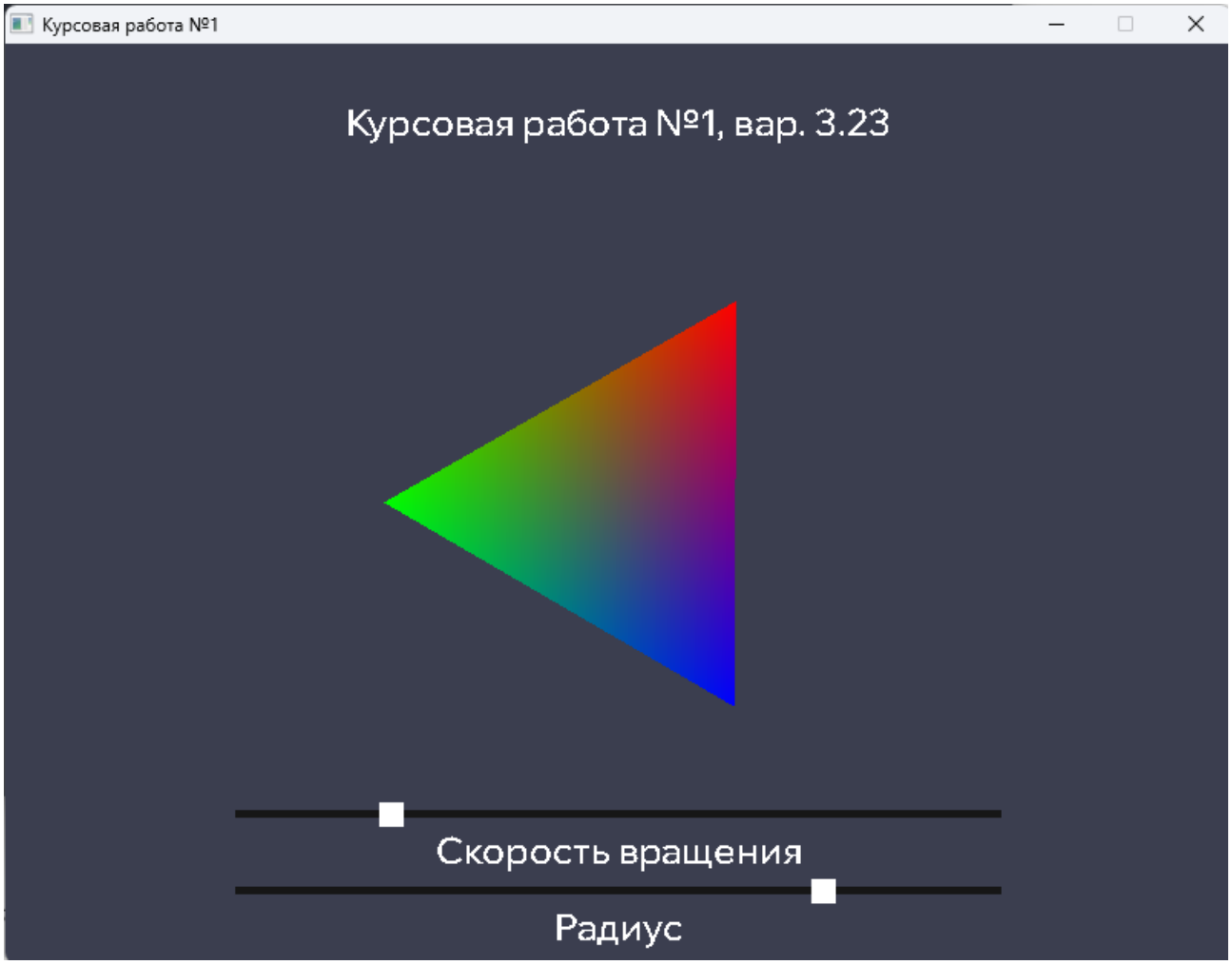
ПРИЛОЖЕНИЕ А

Репозиторий с исходным кодом программы на GitHub.

https://github.com/Broyler/Term1Paper/

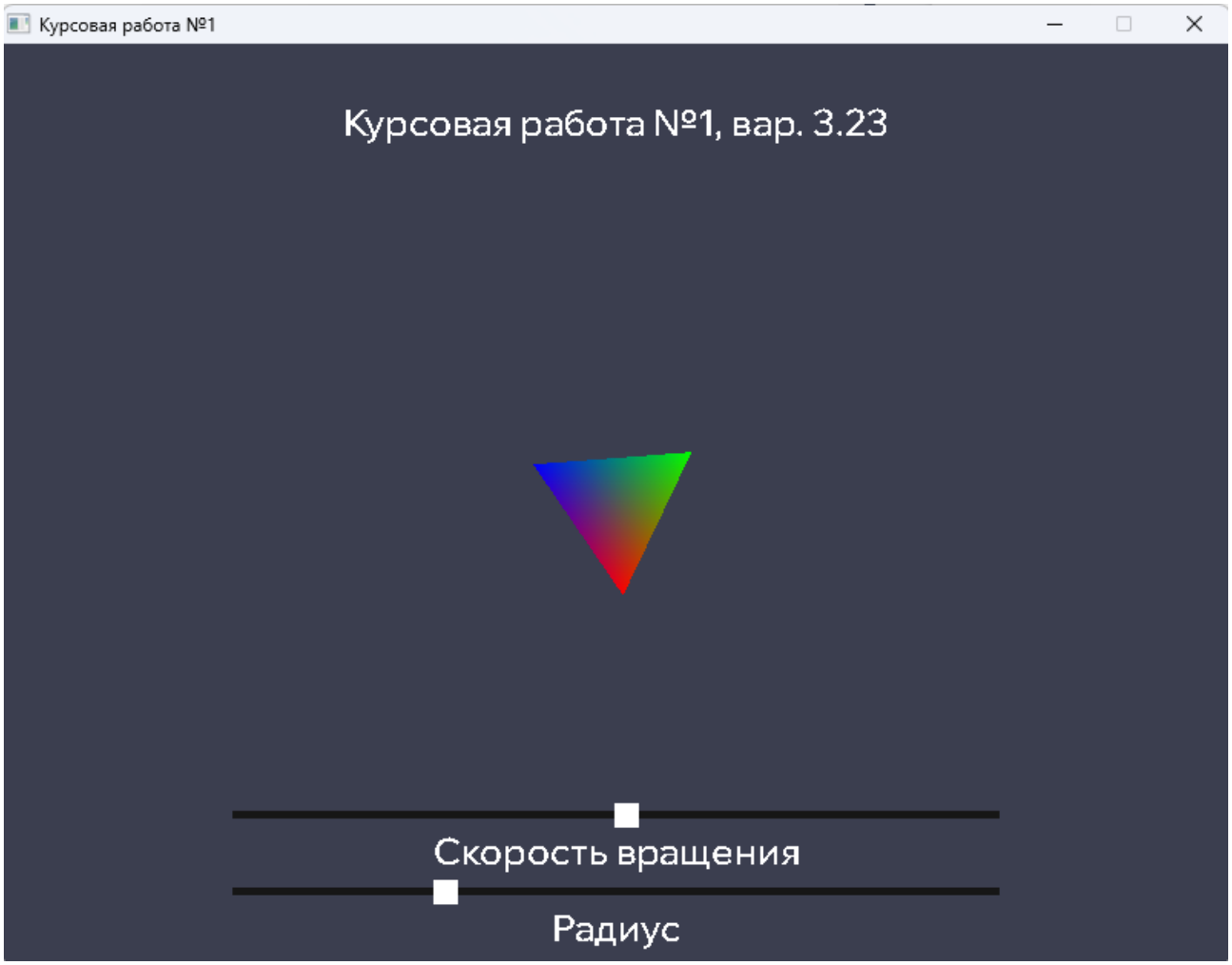
ПРИЛОЖЕНИЕ Б

Снимок экрана работы программы



ПРИЛОЖЕНИЕ В

Снимок экрана работы программы



ПРИЛОЖЕНИЕ Г

Исходный код файла main.c

```
#define SDL_MAIN_HANDLED

#include "math.h"
#include "SDL2/SDL.h"
#include "stdbool.h"
#include "stdint.h"
#include "stdio.h"
#include "texts.h"
#include "sliders.h"
#include "inputs.h"

bool run = true;
uint64_t frequency, lastTick;

SDL_Window* InitSDL() {
  if (SDL_Init(SDL_INIT_VIDEO) < 0)
    return NULL;

  if (TTF_Init() == -1)
    return NULL;

  font = TTF_OpenFont("font.ttf", 24);
  frequency = SDL_GetPerformanceFrequency();

    return SDL_CreateWindow(
      "Курсовая работа №1",
      SDL_WINDOWPOS_CENTERED,
      SDL_WINDOWPOS_CENTERED,
      SCREEN_WIDTH,
      SCREEN_HEIGHT,
      SDL_WINDOW_SHOWN
    );
}

void Render(SDL_Renderer* renderer) {
  SetRenderColor(renderer, &BACKGROUND_COLOR);
  SDL_RenderClear(renderer);
  SDL_RenderGeometry(renderer, NULL, vertices, 3, NULL, 0);
  RenderTexts(renderer);
  RenderSliders(renderer);
  SDL_RenderPresent(renderer);
}

int main(int argc, char* argv[]) {
    SDL_Window* window = InitSDL();
    SDL_Renderer* renderer = SDL_CreateRenderer(
      window,
```

```
48        -1,
49        SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC
50      );
51
52      if (window == NULL || renderer == NULL) {
53        printf("SDL error: %s\n", SDL_GetError());
54        return 1;
55      }
56
57      CreateFontTextures(renderer);
58
59      SDL_Event e;
60      lastTick = SDL_GetPerformanceCounter();
61
62      while (run) {
63        EventLoop(&e);
64        CalculateVertices();
65        AddRotation(&lastTick);
66        Render(renderer);
67      }
68
69      SDL_DestroyWindow(window);
70      SDL_Quit();
71
72    return 0;
73  }
```

ПРИЛОЖЕНИЕ Д

Исходный код заголовочного файла triangle.h

```
double speed = M_PI / 2;  // Радианы в секунду
double radius = 100;
double rotation;
SDL_Vertex vertices[3];

void SetRenderColor(SDL_Renderer* renderer, const SDL_Color*
   color) {
  SDL_SetRenderDrawColor(
    renderer,
    color->r,
    color->g,
    color->b,
    color->a
  );
}
```

ПРИЛОЖЕНИЕ Е

Исходный код заголовочного файла constants.h

```
const int SCREEN_WIDTH = 800;
const int SCREEN_HEIGHT = 600;
const SDL_Color BACKGROUND_COLOR = {0x3b, 0x3e, 0x4f, 0xff};
const SDL_Color TRIANGLE_COLORS[3] = {
  {0xff, 0, 0, 0xff},
  {0, 0, 0xff, 0xff},
  {0, 0xff, 0, 0xff}
};
```

ПРИЛОЖЕНИЕ Ж

Исходный код заголовочного файла calculations.h

```
#include "triangle.h"

extern uint64_t frequency;

double Lerp(double x, double min, double max) {
  if (x <= 0)
    return min;

  if (x >= 1)
    return max;

  return x * (max - min) + min;
}

double InverseLerp(double x, double min, double max) {
  if (x <= min)
    return 0;

  if (x >= max)
    return 1;

  return (x - min) / (max - min);
}

void CalculateVertices() {
  double angle, x, y;

  for (int i = 0; i < 3; ++i) {
    angle = rotation + i * (2 * M_PI / 3);
    x = cos(angle) * radius + SCREEN_WIDTH / 2;
    y = sin(angle) * radius + SCREEN_HEIGHT / 2;
    vertices[i] = (SDL_Vertex){(SDL_FPoint){x, y},
   TRIANGLE_COLORS[i]};
  }
}

void AddRotation(uint64_t* lastTick) {
  uint64_t now = SDL_GetPerformanceCounter();
  double delta = (double)(now - *lastTick) / frequency;
  *lastTick = now;
    rotation += speed * delta;
}
```

ПРИЛОЖЕНИЕ И

Исходный код заголовочного файла inputs.h

```
extern bool run;

void MouseDown(SDL_MouseButtonEvent e) {
  for (int i = 0; i < slidersAmount; ++i) {
    if (SDL_PointInRect(&(SDL_Point){e.x, e.y}, &sliders[i].rect)
   )
      sliders[i].dragged = true;
  }
}

void MouseUp(SDL_MouseButtonEvent e) {
  for (int i = 0; i < slidersAmount; ++i) {
    if (sliders[i].dragged)
      sliders[i].dragged = false;
  }
}

void EventLoop(SDL_Event* e) {
    while(SDL_PollEvent(e) != 0)
    {
      switch (e->type) {
      case SDL_QUIT:
        run = false;
        break;

      case SDL_MOUSEBUTTONDOWN:
        MouseDown(e->button);
        break;

      case SDL_MOUSEBUTTONUP:
        MouseUp(e->button);
        break;

      case SDL_MOUSEMOTION:
        UpdateSliderValues(e->motion);
        break;
      }
  }
}
```

ПРИЛОЖЕНИЕ К

Исходный код заголовочного файла sliders.h

```
#include "calculations.h"

typedef struct {
  const int index;
  const double min;
  const double max;
  double* value;
  SDL_Rect rect;
  bool dragged;
} Slider;

const int SLIDER_PADDING = 150;
const int SLIDER_OFFSET = 50;
const int SLIDER_WIDTH = 5;
const int HANDLE_WIDTH = 16;
const SDL_Color SLIDER_COLOR = {0x16, 0x16, 0x16, 0xff};
const SDL_Color HANDLE_COLOR = {0xff, 0xff, 0xff, 0xff};
const Slider radiusSlider = {1, 10, 200, &radius};
const Slider speedSlider = {2, -2 * M_PI, 2 * M_PI, &speed};
Slider sliders[] = {radiusSlider, speedSlider};
const int slidersAmount = 2;

void RenderSliderBars(SDL_Renderer* renderer) {
  SetRenderColor(renderer, &SLIDER_COLOR);

  for (int i = 1; i < 3; ++i) {
    SDL_RenderFillRect(renderer, &(SDL_Rect){
      SLIDER_PADDING,
      SCREEN_HEIGHT - SLIDER_OFFSET * i,
      SCREEN_WIDTH - 2 * SLIDER_PADDING,
      SLIDER_WIDTH
    });
  }
}

void RenderHandle(SDL_Renderer* renderer, Slider* slider) {
  double percentage = InverseLerp(*slider->value, slider->min,
   slider->max);
  int x = percentage * (SCREEN_WIDTH - 2 * SLIDER_PADDING) +
   SLIDER_PADDING;
  int y = SCREEN_HEIGHT - SLIDER_OFFSET * slider->index;
  y -= (HANDLE_WIDTH - SLIDER_WIDTH) / 2;
  slider->rect = (SDL_Rect){x, y, HANDLE_WIDTH, HANDLE_WIDTH};
  SDL_RenderFillRect(renderer, &slider->rect);
}

void RenderSliderHandles(SDL_Renderer* renderer) {
```

```
46    SetRenderColor(renderer, &HANDLE_COLOR);
47
48    for (int i = 0; i < slidersAmount; ++i) {
49      RenderHandle(renderer, &sliders[i]);
50    }
51  }
52
53  void RenderSliders(SDL_Renderer* renderer) {
54    RenderSliderBars(renderer);
55    RenderSliderHandles(renderer);
56  }
57
58  void UpdateSliderValues(SDL_MouseMotionEvent e) {
59    for (int i = 0; i < slidersAmount; ++i) {
60      if (sliders[i].dragged) {
61        double progress = InverseLerp(e.x, SLIDER_PADDING,
     SCREEN_WIDTH - SLIDER_PADDING);
62        *sliders[i].value = Lerp(progress, sliders[i].min, sliders[
     i].max);
63      }
64    }
65  }
```

ПРИЛОЖЕНИЕ Л

Исходный код заголовочного файла texts.h

```
#include "SDL2/SDL.h"
#include "SDL2/SDL_ttf.h"
#include "constants.h"

typedef struct {
  const char* text;
  const int x;
  const int y;
  SDL_Color color;
  SDL_Texture* texture;
  SDL_Rect rect;
} Text;

TTF_Font* font;
Text texts[] = {
  {"Курсовая работа №1, вар. 3.23", SCREEN_WIDTH / 2, 50, (SDL_Color
   ){0xff, 0xff, 0xff, 0xff}},
  {"Радиус", SCREEN_WIDTH / 2, SCREEN_HEIGHT - 25, (SDL_Color){0
   xff, 0xff, 0xff, 0xff}},
  {"Скорость вращения", SCREEN_WIDTH / 2, SCREEN_HEIGHT - 75, (
   SDL_Color){0xff, 0xff, 0xff, 0xff}}
};
int textsAmount = 3;

void CreateFontTextures(SDL_Renderer* renderer) {
  for (int i = 0; i < textsAmount; ++i) {
    SDL_Surface* text = TTF_RenderUTF8_Solid(font, texts[i].text,
    texts[i].color);
    SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer,
    text);
    texts[i].texture = texture;
    int w, h;
    SDL_QueryTexture(texture, NULL, NULL, &w, &h);
    SDL_Rect rect = {texts[i].x - w / 2, texts[i].y - h / 2, w, h
   };
    texts[i].rect = rect;
  }
}

void RenderTexts(SDL_Renderer* renderer) {
  for (int i = 0; i < textsAmount; ++i) {
    SDL_RenderCopy(renderer, texts[i].texture, NULL, &texts[i].
   rect);
  }
}
```