МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Институт математики и информационных систем
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Дата сдачи на проверку:

«__»_____ 2025 г.

Проверено:

«__»_____ 2025 г.

ИССЛЕДОВАНИЕ ФРАКТАЛОВ

Отчёт по лабораторной работе №7

по дисциплине

«Программирование»

Разработал студент гр. ИВТб-1301-05-00 _____/Черкасов А. А./

(подпись)

Заведующая кафедры ЭВМ _____/Долженкова М. Л./

(подпись)

Работа защищена «__»_____ 2025 г.

Киров

2025

Цель

Цель работы: Получение навыков реализации алгоритмов с рекурсивными вычислениями, знакомство с фракталами.

Задание

- Написать программу для визуализации фрактала "Кривая Гильберта".
- Предусмотреть возможность масштабирования, изменения глубины прорисовки и перемещения полученной фигуры.
- Построение множества ломанных, образующих фрактал, должно осуществляться в отдельном модуле.

Решение

Схемы алгоритмов решения задания представлены на рисунках 1.1, 1.2, 1.3. Исходный код решений представлен в Приложениях А1 - А9.

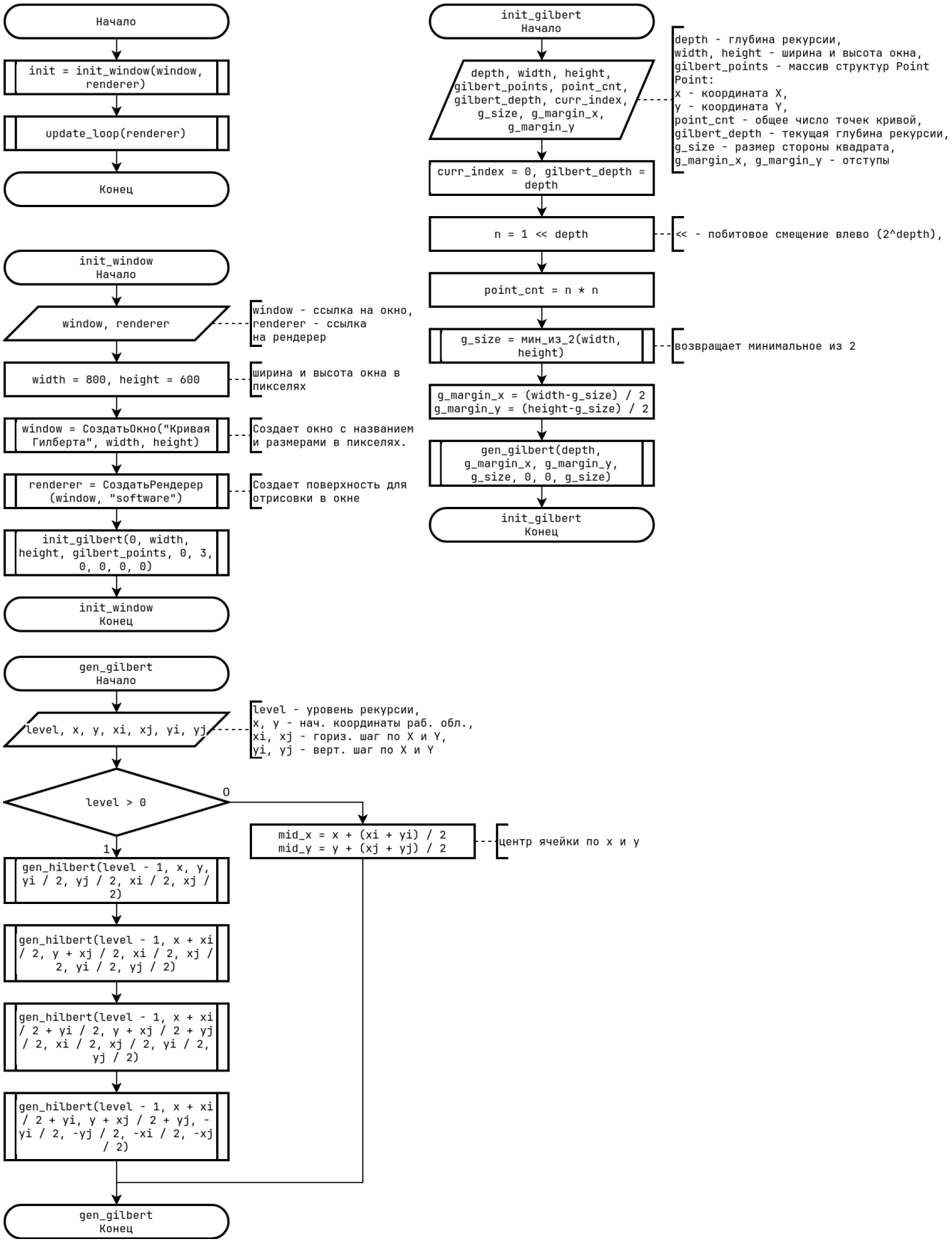

Рисунок 1.1 - Схемы алгоритмов основной программы, подпрограмм инициализации окна и кривой Гильберта, генерации кривой Гильберта.

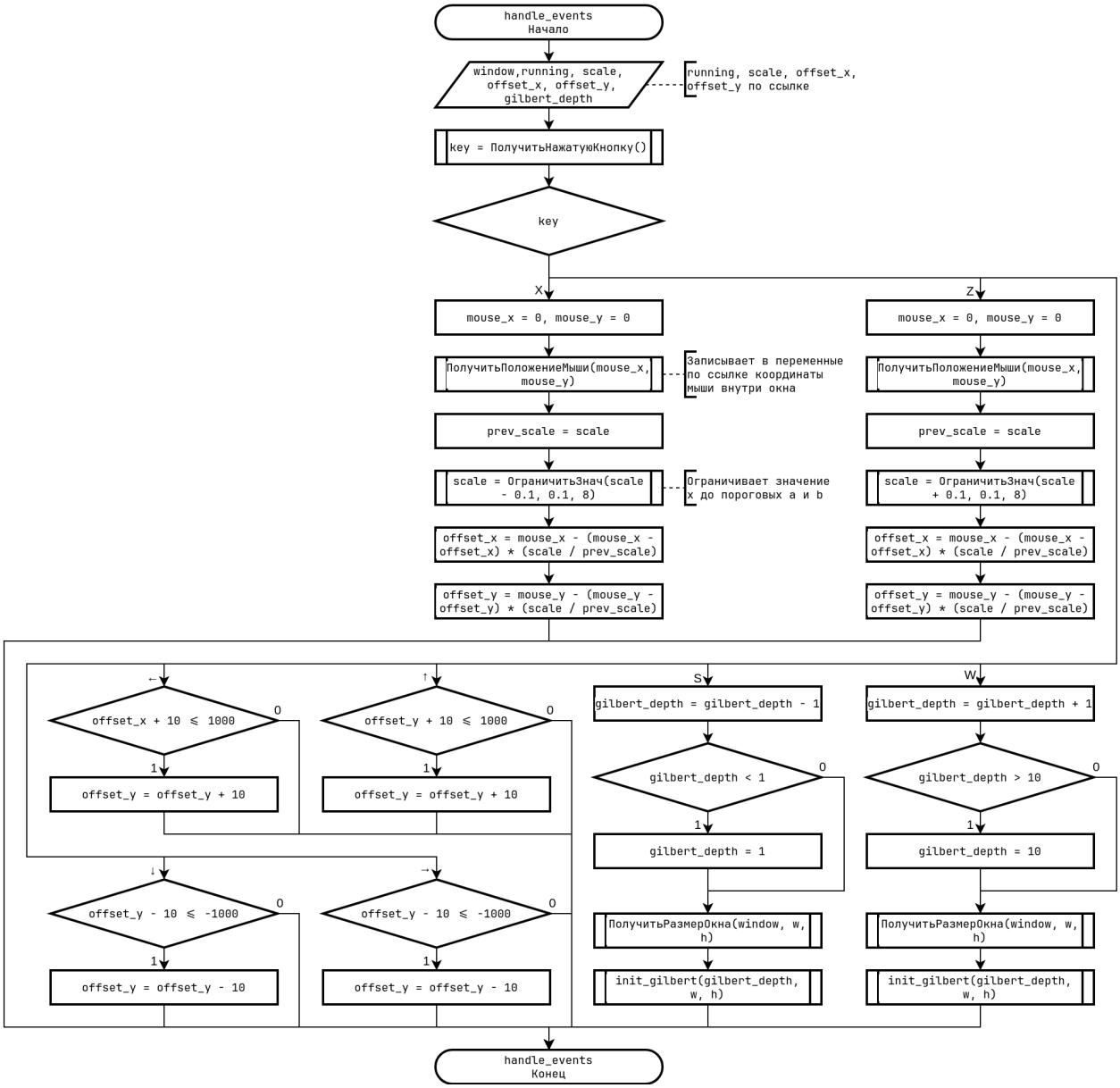

Рисунок 1.2 - Схема подпрограммы обработки ввода с клавиатуры.

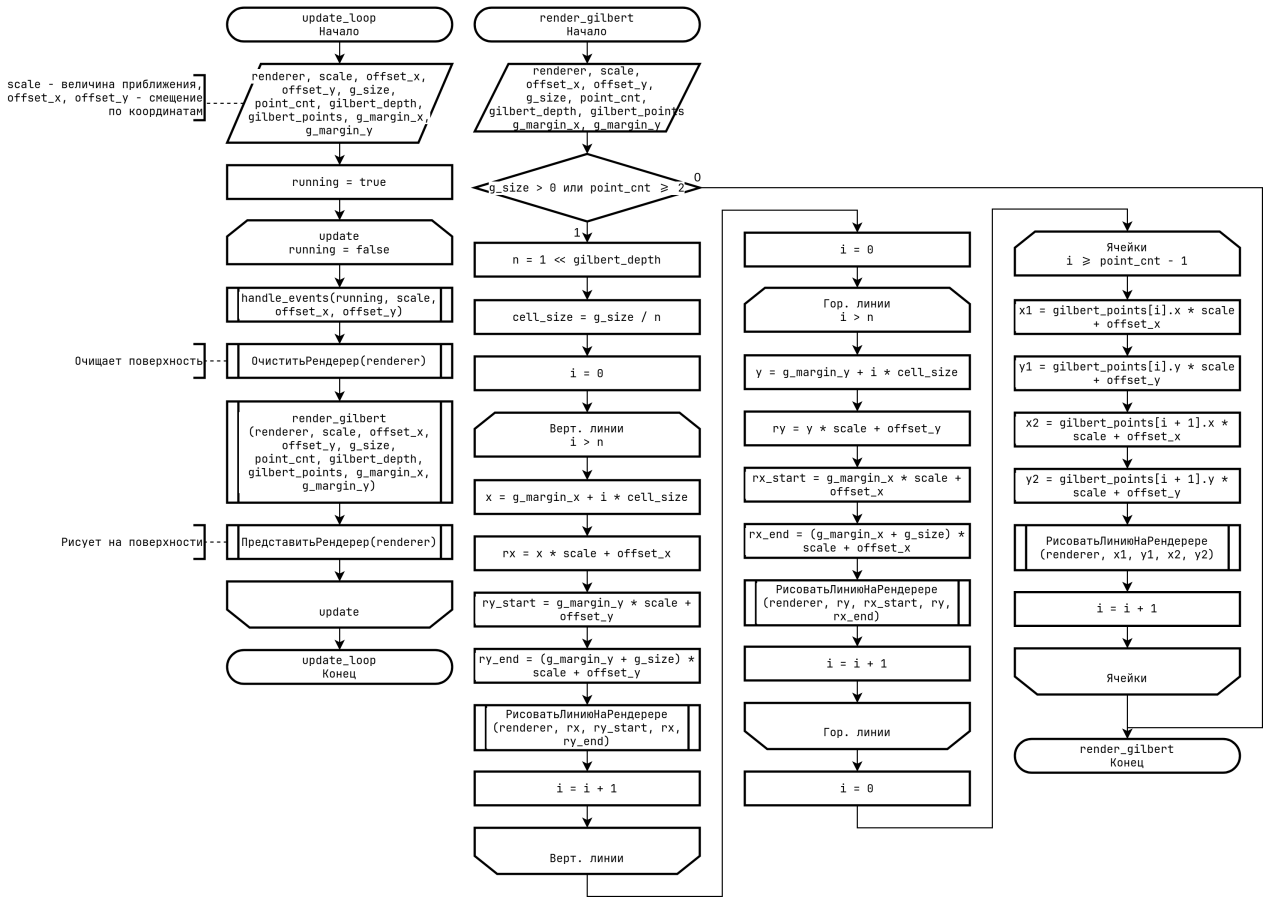

Рисунок 1.3 - Схемы алгоритмов подпрограмм цикла обновления и рендера кривой.

Вывод

Выполненная лабораторная работа позволила на практике исследовать принципы построения и визуализации фрактальной кривой Гильберта с использованием рекурсивного алгоритма. Для взаимодействия с пользователем реализован интерфейс на базе SDL3, обеспечивающий плавное изменение глубины прорисовки (клавиши W/S), масштабирования (Z/X) и перемещения фрактала (стрелки). Работа способствовала углублению навыков рекурсивного программирования, освоению приёмов модульного проектирования на языке C и практической работе с графической библиотекой SDL3.

Приложение А1. Исходный код

```c
#include <SDL3/SDL.h>

#include <stdio.h>
#include <string.h>

#include "args_parser.h"
#include "utils.h"
#include "window.h"

int main(int argc, char *argv[]) {
  WindowConfig config = {.w_title = "Кривая Гилберта // W/S - увел./уменш. "
                                    "глубину // Z/X - увел./уменьш. масштаб",
                         .w_width = 800,
                         .w_height = 600};

  parse_arguments(argc, argv, &config);

  if (init_window(&config) != 0) {
    fprintf(stderr, "Failed to initialize window\n");
    return 1;
  }

  update_loop();

  return 0;
}
```

Приложение А2. Исходный код

```c
#ifndef ARGS_PARSER_H
#define ARGS_PARSER_H
#include "window.h"
```

```c
#include <SDL3/SDL.h>
// Парсинг аргументов командной строки
int parse_arguments(int argc, char *argv[], WindowConfig *config);

#endif // ARGS_PARSER_H
```

Приложение А3. Исходный код

```c
#include "args_parser.h"
#include <stdio.h>
#include <string.h>

int parse_arguments(int argc, char *argv[], WindowConfig *config) {
  for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-n") == 0 || strcmp(argv[i], "--name") == 0) {
      if (i + 1 < argc && argv[i + 1][0] != '-') {
        config->w_title = argv[++i];
      } else {
        fprintf(stderr, "Error: Missing value for %s\n", argv[i]);
        return 1;
      }
    } else if (strcmp(argv[i], "-s") == 0 || strcmp(argv[i], "--size") == 0) {
      if (i + 1 < argc && argv[i + 1][0] != '-') {
        if (sscanf(argv[++i], "%dx%d", &config->w_width, &config->w_height) !=
                2 ||
            config->w_width < 160 || config->w_height < 120) {
          fprintf(stderr, "Invalid size. Using 800x600.\n");
          config->w_width = 800;
          config->w_height = 600;
        }
      } else {
        fprintf(stderr, "Error: Missing value for %s\n", argv[i]);
        return 1;
      }
```

```
    } else if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
      printf("Usage: %s [options]\n", argv[0]);
      printf("Options:\n");
      printf("  -n, --name <title>   Set the window title\n");
      printf("  -s, --size <WIDTHxHEIGHT>   Set the window size (minimum "
             "160x120)\n");
      printf("  -h, --help          Show this help message\n");
      return 0;

    } else {
      fprintf(stderr, "Unknown option: %s\n", argv[i]);
      return 1;
    }
  }
  return 0;
}
```

Приложение А4. Исходный код

```
#ifndef WINDOW_H
#define WINDOW_H

#include <SDL3/SDL.h>
#include <stdio.h>

extern SDL_Window* window;
extern SDL_Renderer* renderer;

extern float scale;
extern int offset_x;
extern int offset_y;

typedef struct WindowConfig {
  const char* w_title;
```

```
  int w_width;
  int w_height;
} WindowConfig;

int init_window(WindowConfig *config);

void update_loop();

void handle_events(bool *running);

void destroy_window();

#endif
```

Приложение А5. Исходный код

```
#include "window.h"
#include "render_gilbert.h"
#include "utils.h"

SDL_Window *window = NULL;
SDL_Renderer *renderer = NULL;
float scale = 1.0f;
int offset_x = 0;
int offset_y = 0;

int init_window(WindowConfig *config) {
  int init_res = SDL_Init(SDL_INIT_VIDEO);
  const char *err_msg = SDL_GetError();
  if (init_res != 0 && strlen(err_msg) > 0) {
    fprintf(stderr, "SDL_Init Error: %s\n", err_msg);
    return 1;
  }
```

```
  window = SDL_CreateWindow(config->w_title, config->w_width, config->w_height,
                            SDL_WINDOW_RESIZABLE);

  if (window == NULL) {
    fprintf(stderr, "SDL_CreateWindow Error: %s\n", SDL_GetError());
    SDL_Quit();
    return 1;
  }

  renderer = SDL_CreateRenderer(
      window, "software"); // direct3d11 direct3d12 direct3d opengl opengles2
                           // vulkan gpu software
  if (renderer == NULL) {
    fprintf(stderr, "SDL_CreateRenderer Error: %s\n", SDL_GetError());
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 1;
  }

  init_gilbert(gilbert_depth, config->w_width, config->w_height);

  return 0;
}

void handle_events(bool *running) {
  SDL_Event event;
  while (SDL_PollEvent(&event)) {
    switch (event.type) {

      // #region MARK: KEY_DOWN
      // >---------------------KEY_DOWN EVENTS----------------------<

    case SDL_EVENT_KEY_DOWN: {
      SDL_Keycode key = event.key.key;
```

```
switch (key) {
case SDLK_ESCAPE: {
  *running = false;
} break;

case SDLK_X: { // Зум-ин (увеличение)
  float mouse_x, mouse_y;
  SDL_GetMouseState(&mouse_x, &mouse_y);
  float prev_scale = scale;
  scale = SDL_clamp(scale - 0.1f, 0.1f, 8.0f);

  // Корректируем смещение относительно мыши
  offset_x = mouse_x - (mouse_x - offset_x) * (scale / prev_scale);
  offset_y = mouse_y - (mouse_y - offset_y) * (scale / prev_scale);
} break;

case SDLK_Z: { // Зум-аут (уменьшение)
  float mouse_x, mouse_y;
  SDL_GetMouseState(&mouse_x, &mouse_y);
  float prev_scale = scale;
  scale = SDL_clamp(scale + 0.1f, 0.1f, 10.0f);

  // Смещение относительно мыши
  offset_x = mouse_x - (mouse_x - offset_x) * (scale / prev_scale);
  offset_y = mouse_y - (mouse_y - offset_y) * (scale / prev_scale);
} break;

// Перемещение
case SDLK_UP: {
  if (offset_y + 10 <= 1000) {
    offset_y += 10;
    fprintf(stdout, "Move up | Y%d\n", offset_y);
  }
} break;
```

```
case SDLK_DOWN: {
  if (offset_y - 10 >= -1000) {
    offset_y -= 10;
    fprintf(stdout, "Move down | Y%d\n", offset_y);
  }
} break;

case SDLK_LEFT: {
  if (offset_x + 10 <= 1000) {
    offset_x += 10;
    fprintf(stdout, "Move left | X%d\n", offset_x);
  }
} break;

case SDLK_RIGHT: {
  if (offset_x - 10 >= -1000) {
    offset_x -= 10;
    fprintf(stdout, "Move right | X%d\n", offset_x);
  }
} break;

// Изменение глубины
case SDLK_W: {
  gilbert_depth++;
  if (gilbert_depth > 10)
    gilbert_depth = 10;
  int w, h;
  SDL_GetWindowSize(window, &w, &h);
  fprintf(stdout, "Increase depth | %d\n", gilbert_depth);
  init_gilbert(gilbert_depth, w, h);
} break;

case SDLK_S: {
```

```
        gilbert_depth--;
        if (gilbert_depth < 1)
          gilbert_depth = 1;
        int w, h;
        SDL_GetWindowSize(window, &w, &h);
        fprintf(stdout, "Decrease depth | %d\n", gilbert_depth);
        init_gilbert(gilbert_depth, w, h);
      } break;

      default:
        break;
      }
      break;
    }
      // #endregion MARK: KEY_DOWN

      // #region MARK: OTHER_EVENTS
      // >--------------------------OTHER_EVENTS--------------------------<
    case SDL_EVENT_WINDOW_RESIZED: {
      int w, h;
      SDL_GetWindowSize(window, &w, &h);
      init_gilbert(gilbert_depth, w, h);
      break;
    }

    case SDL_EVENT_QUIT: {
      *running = false;
    } break;

    default:
      break;
    }
    // #endregion MARK: OTHER_EVENTS
}
```

```
}

void update_loop() {
  bool running = true;
  while (running) {
    handle_events(&running);

    // Buffer clear
    SDL_Color bg_color = hexa_to_rgba(CP_MOCHA_BASE, 1.0);
    SDL_SetRenderDrawColor(renderer, bg_color.r, bg_color.g, bg_color.b,
                           bg_color.a);
    SDL_RenderClear(renderer);

    render_gilbert(scale, offset_x, offset_y);

    SDL_RenderPresent(renderer);
  }
}

void destroy_window() {
  if (renderer) {
    SDL_DestroyRenderer(renderer);
    renderer = NULL;
  }
  if (window) {
    SDL_DestroyWindow(window);
    window = NULL;
  }
  fprintf(stdout, "Bye!\n");
  SDL_Quit();
}
```

Приложение А6. Исходный код

```
#ifndef RENDER_GILBERT_H
#define RENDER_GILBERT_H

#include "utils.h"
#include <SDL3/SDL.h>

/**
 * @brief Структура точки с целочисленными координатами.
 */
typedef struct {
  int x; ///< координата X
  int y; ///< координата Y
} Point;

/**
 * Глобальный массив точек, представляющих кривую Гилберта.
 */
extern Point *gilbert_points;

/**
 * Общее число точек кривой (n*n, где n = 2^depth).
 */
extern int point_count;

/**
 * Текущий уровень рекурсии (детализации) кривой Гилберта.
 */
extern int gilbert_depth;

/**
 * Рабочая область (центрированный квадрат), в которой генерируется кривая.
 */
```

```
extern float g_margin_x; ///< Отступ от левого края до квадрата
extern float g_margin_y; ///< Отступ от верхнего края до квадрата
extern float g_size;     ///< Размер стороны квадрата

/**
 * @brief Рекурсивно генерирует точки кривой Гилберта.
 *
 * @param level Уровень рекурсии.
 * @param x Начальная координата X рабочей области.
 * @param y Начальная координата Y рабочей области.
 * @param xi Горизонтальный шаг по X.
 * @param xj Горизонтальный шаг по Y.
 * @param yi Вертикальный шаг по X.
 * @param yj Вертикальный шаг по Y.
 */
void gen_hilbert(int level, float x, float y, float xi, float xj, float yi,
                 float yj);

/**
 * @brief Инициализирует кривую Гилберта.
 *
 * Вычисляет центрированный квадрат (рабочую область) для генерации кривой,
 * выделяет память для точек и генерирует кривую с помощью рекурсии.
 *
 * @param depth Уровень рекурсии (детализации).
 * @param width Ширина окна.
 * @param height Высота окна.
 */
void init_gilbert(int depth, int width, int height);

/**
 * @brief Отрисовывает квадратную сетку и кривую Гилберта.
 *
 * Точки кривой находятся в центрах ячеек сетки, которая
```

```
 * рассчитывается исходя из рабочей области. Применяются масштаб (scale)
 * и смещение (offset_x, offset_y).
 *
 * @param scale Коэффициент масштабирования.
 * @param offset_x Горизонтальное смещение.
 * @param offset_y Вертикальное смещение.
 */
void render_gilbert(float scale, int offset_x, int offset_y);

#endif // RENDER_GILBERT_H
```

Приложение А7. Исходный код

```
#include "render_gilbert.h"
#include "utils.h"
#include "window.h"
#include <math.h>
#include <stdlib.h>

/* Глобальный массив точек кривой и параметры */
Point *gilbert_points = NULL;
int point_count = 0;
int gilbert_depth = 3;
static int current_index = 0;

/* Параметры рабочей области (центрированного квадрата, где строится кривая) */
float g_margin_x = 0.0f; // отступ по горизонтали
float g_margin_y = 0.0f; // отступ по вертикали
float g_size = 0.0f;     // размер стороны квадрата

/**
 * @brief Добавляет точку в массив точек.
 *
 * Точка записывается как центр ячейки, вычисленный рекурсивным алгоритмом.
```

```
 *
 * @param x Координата X точки.
 * @param y Координата Y точки.
 */
static void add_point(float x, float y) {
  if (current_index < point_count) {
    gilbert_points[current_index].x = (int)x;
    gilbert_points[current_index].y = (int)y;
    current_index++;
  }
}

/**
 * @brief Рекурсивная генерация точки кривой Гилберта.
 *
 * @param level Уровень рекурсии.
 * @param x Начальная координата X рабочей области.
 * @param y Начальная координата Y рабочей области.
 * @param xi Горизонтальный шаг по X.
 * @param xj Горизонтальный шаг по Y.
 * @param yi Вертикальный шаг по X.
 * @param yj Вертикальный шаг по Y.
 */
void gen_hilbert(int level, float x, float y, float xi, float xj, float yi,
                 float yj) {
  if (level <= 0) {
    // Вычисляем центр ячейки
    float mid_x = x + (xi + yi) / 2.0f;
    float mid_y = y + (xj + yj) / 2.0f;
    add_point(mid_x, mid_y);
  } else {
    gen_hilbert(level - 1, x, y, yi / 2.0f, yj / 2.0f, xi / 2.0f, xj / 2.0f);
    gen_hilbert(level - 1, x + xi / 2.0f, y + xj / 2.0f, xi / 2.0f, xj / 2.0f,
                yi / 2.0f, yj / 2.0f);
```

```
    gen_hilbert(level - 1, x + xi / 2.0f + yi / 2.0f, y + xj / 2.0f + yj / 2.0f,
                xi / 2.0f, xj / 2.0f, yi / 2.0f, yj / 2.0f);
    gen_hilbert(level - 1, x + xi / 2.0f + yi, y + xj / 2.0f + yj, -yi / 2.0f,
                -yj / 2.0f, -xi / 2.0f, -xj / 2.0f);
  }
}

/**
 * @brief Инициализация кривой Гилберта.
 *
 * Вычисляет рабочую область (центрированный квадрат) для генерации кривой,
 * выделяет память для точек и генерирует кривую.
 *
 * @param depth Уровень рекурсии (детализации).
 * @param width Ширина окна.
 * @param height Высота окна.
 */
void init_gilbert(int depth, int width, int height) {
  if (gilbert_points) {
    free(gilbert_points);
    gilbert_points = NULL;
  }
  current_index = 0;
  gilbert_depth = depth;

  // Число ячеек: n = 2^depth, общее число точек = n*n
  int n = 1 << depth;
  point_count = n * n;
  gilbert_points = malloc(point_count * sizeof(Point));

  // Определяем рабочую область - квадрат, вписывающийся в окно (центрированный)
  g_size = fminf((float)width, (float)height);
  g_margin_x = ((float)width - g_size) / 2.0f;
  g_margin_y = ((float)height - g_size) / 2.0f;
```

```
  // Генерируем кривую в рабочей области
  gen_hilbert(depth, g_margin_x, g_margin_y, g_size, 0, 0, g_size);
}

/**
 * @brief Отрисовывает квадратную сетку и кривую Гилберта.
 *
 * Сетка строится в пределах рабочей области, где каждая ячейка имеет размер
 * g_size/n. Точки кривой, генерируемые функцией gen_hilbert, находятся в
 * центрах ячеек. Применяется масштабирование и смещение.
 *
 * @param scale Коэффициент масштабирования.
 * @param offset_x Горизонтальное смещение.
 * @param offset_y Вертикальное смещение.
 */
void render_gilbert(float scale, int offset_x, int offset_y) {
  if (g_size <= 0 || point_count < 2)
    return;

  int n = 1 << gilbert_depth;   // Количество ячеек по стороне
  float cell_size = g_size / n; // Размер ячейки

  // Рисуем квадратную сетку
  SDL_Color grid_color = hexa_to_rgba(CP_MOCHA_SURFACE_1, 1.0);
  SDL_SetRenderDrawColor(renderer, grid_color.r, grid_color.g, grid_color.b,
                         grid_color.a);

  // Вертикальные линии
  for (int i = 0; i <= n; i++) {
    float x = g_margin_x + i * cell_size;
    int rx = (int)(x * scale + offset_x);
    int ry_start = (int)(g_margin_y * scale + offset_y);
    int ry_end = (int)((g_margin_y + g_size) * scale + offset_y);
```

```
    SDL_RenderLine(renderer, rx, ry_start, rx, ry_end);
  }

  // Горизонтальные линии
  for (int i = 0; i <= n; i++) {
    float y = g_margin_y + i * cell_size;
    int ry = (int)(y * scale + offset_y);
    int rx_start = (int)(g_margin_x * scale + offset_x);
    int rx_end = (int)((g_margin_x + g_size) * scale + offset_x);
    SDL_RenderLine(renderer, rx_start, ry, rx_end, ry);
  }

  // Рисуем кривую Гилберта (соединяем центры ячеек)
  SDL_Color curve_color = hexa_to_rgba(CP_MOCHA_RED, 1.0);
  SDL_SetRenderDrawColor(renderer, curve_color.r, curve_color.g, curve_color.b,
                         curve_color.a);

  for (int i = 0; i < point_count - 1; i++) {
    int x1 = (int)(gilbert_points[i].x * scale + offset_x);
    int y1 = (int)(gilbert_points[i].y * scale + offset_y);
    int x2 = (int)(gilbert_points[i + 1].x * scale + offset_x);
    int y2 = (int)(gilbert_points[i + 1].y * scale + offset_y);
    SDL_RenderLine(renderer, x1, y1, x2, y2);
  }
}
```

Приложение А8. Исходный код

```
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <stdlib.h>
```

```
#include <SDL3/SDL.h>

// Damn hex color palete

#define CP_MOCHA_CRUST 0x11111b     // Catppuccin Mocha Crust
#define CP_MOCHA_MANTLE 0x181825    // Catppuccin Mocha Mantle
#define CP_MOCHA_BASE 0x1e1e2e      // Catppuccin Mocha Base
#define CP_MOCHA_SURFACE_0 0x313244 // Catppuccin Mocha Surface 0
#define CP_MOCHA_SURFACE_1 0x45475a // Catppuccin Mocha Surface 1
#define CP_MOCHA_SURFACE_2 0x585b70 // Catppuccin Mocha Surface 2
#define CP_MOCHA_OVERLAY_0 0x6c7086 // Catppuccin Mocha Overlay 0
#define CP_MOCHA_OVERLAY_1 0x7f849c // Catppuccin Mocha Overlay 1
#define CP_MOCHA_OVERLAY_2 0x9399b2 // Catppuccin Mocha Overlay 2
#define CP_MOCHA_SUBTEXT_0 0xa6adc8 // Catppuccin Mocha Subtext 0
#define CP_MOCHA_SUBTEXT_1 0xbac2de // Catppuccin Mocha Subtext 1
#define CP_MOCHA_TEXT 0xcdd6f4      // Catppuccin Mocha Text
#define CP_MOCHA_ROSEWATER 0xf5e0dc // Catppuccin Mocha Lavander
#define CP_MOCHA_FLAMINGO 0xf2cdcd  // Catppuccin Mocha Flamingo
#define CP_MOCHA_PINK 0xf5c2e7      // Catppuccin Mocha Pink
#define CP_MOCHA_MAUVE 0xcba6f7     // Catppuccin Mocha Mauve
#define CP_MOCHA_RED 0xf38ba8       // Catppuccin Mocha Red
#define CP_MOCHA_MAROON 0xeba0ac    // Catppuccin Mocha Maroon
#define CP_MOCHA_PEACH 0xfab387     // Catppuccin Mocha Peach
#define CP_MOCHA_YELLOW 0xf9e2af    // Catppuccin Mocha Yellow
#define CP_MOCHA_GREEN 0xa6e3a1     // Catppuccin Mocha Green
#define CP_MOCHA_TEAL 0x94e2d5      // Catppuccin Mocha Teal
#define CP_MOCHA_SKY 0x89dceb       // Catppuccin Mocha Sky
#define CP_MOCHA_SAPPHIRE 0x74c7ec  // Catppuccin Mocha Sapphire
#define CP_MOCHA_BLUE 0x89b4fa      // Catppuccin Mocha Blue
#define CP_MOCHA_LAVANDER 0xb4befe  // Catppuccin Mocha Lavander

SDL_Color hexa_to_rgba(unsigned int hex, double opacity);

#endif // UTILS_H
```

Приложение А9. Исходный код

```
#include "utils.h"

#include <SDL3/SDL.h>

SDL_Color hexa_to_rgba(unsigned int hex, double opacity) {
  SDL_Color color;
  color.r = (hex >> 16) & 0xFF; // Extract the red component
  color.g = (hex >> 8) & 0xFF;  // Extract the green component
  color.b = hex & 0xFF;         // Extract the blue component
  color.a = (unsigned char)(opacity *
                            255); // Set the alpha component based on opacity
  return color;
}
```