1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

1. **Генератор лабиринтов**
2. по дисциплине «Структуры данных»

1. Выполнили
2. студенты гр. 5151004/10001 Матылицкая А.А

Романова Е.Е.

<*подпись*>

1. Преподаватель
2. асс. преподавателя Панков И.Д.

<*подпись*>

1. «29» сентября 2023 г.
2. Санкт-Петербург
3. 2023

**СОДЕРЖАНИЕ**

[1 Теоретическая часть 3](#_Toc146890058)

[1.1 Алгоритм растущего дерева 3](#_Toc146890059)

[1.2 Алгоритм Sidewinder 4](#_Toc146890060)

[1.3 Алгоритм Эллера 4](#_Toc146890061)

[2 Практическая часть 4](#_Toc146890062)

[2.1 Техническая часть 4](#_Toc146890063)

[2.1.1 Структура game\_fichi 4](#_Toc146890064)

[2.1.2 Структура Node 4](#_Toc146890065)

[2.1.3 Структура button\_menu 4](#_Toc146890066)

[2.1.4 Реализация алгоритма растущего дерева 4](#_Toc146890067)

[2.1.5 Реализация алгоритма sidewinder 4](#_Toc146890068)

[2.1.6 Реализация алгоритма Эллера 4](#_Toc146890069)

[2.2 Графическая часть 4](#_Toc146890070)

[2.2.1 Работа с OpenGL 4](#_Toc146890071)

[2.2.2 Работа с Win Api 4](#_Toc146890072)

[2.2.3 Работа с диалоговыми окнами 4](#_Toc146890073)

[2.2.4 Работа с stb\_easy\_font 4](#_Toc146890074)

[3 Исследовательская часть 4](#_Toc146890075)

[3.1 Сравнение алгоритмов 4](#_Toc146890076)

[3.1.1 Сравнение алгоритмов по скорости работы 4](#_Toc146890077)

[3.1.2 Сравнение алгоритмов по объему используемой памяти 4](#_Toc146890078)

[4 исследование по времени генерации 4](#_Toc146890079)

[4.1 Графики 4](#_Toc146890080)

[ЗАКЛЮЧЕНИЕ 6](#_Toc146890081)

[список использованной литературы 7](#_Toc146890082)

[ПРИЛОЖЕНИЕ А 8](#_Toc146890083)

[Листинг программы «coursework.c» 8](#_Toc146890084)

Введение

В данной курсовой работе рассмотрена генерация лабиринтов с использованием трех алгоритмов: Growing Tree, Sidewinder и Эллера. Лабиринты широко применяются в различных областях, включая компьютерные игры, робототехнику и психологические эксперименты. Генерация лабиринтов является важным этапом в создании таких задач. В ходе работы будет рассмотрена последовательность шагов каждого алгоритма и их реализация на языке программирования Си. Также будет проведен анализ времени выполнения и сложности каждого алгоритма. В конце работы будут представлены полученные лабиринты для каждого алгоритма, которые можно использовать для дальнейших исследований или в различных приложениях.

Цель работы

Укрепить знания в использовании структур данных. Изучить работу с библиотекой OpenGL, а также реализовать и сравнить эти алгоритмы на языке программирования Си.

Поставленные задачи

Написать программу в ОС Windows, которая генерирует лабиринт различными алгоритмами. На вход программы подаются два аргумента: размер лабиринта и алгоритм, которым он будет сгенерирован. Также присутствует возможность сохранения сгенерированного лабиринта в файл, с последующей возможностью загрузить его в программу.

# Теоретическая часть

## Алгоритм растущего дерева

Алгоритм Growing Tree является одним из наиболее популярных алгоритмов генерации лабиринта. Он начинает с одной случайно выбранной клетки и последовательно обрабатывает соседние клетки. Затем выбирается следующая случайная клетка и продолжается процесс до тех пор, пока все клетки не будут посещены.

## Алгоритм Sidewinder

Алгоритм Sidewinder также является эффективным способом генерации лабиринта. Он работает поэтапно, начиная с создания сетки клеток и заканчивая прорывом стен для создания пути. В отличие от алгоритма Growing Tree, Sidewinder использует вероятности для определения, должны ли клетки быть соединены горизонтально или вертикально. Лабиринт генерируется по одной строке за раз: для каждой ячейки случайным образом выбирается, нужно ли вырезать проход, ведущий вправо. Если проход не вырезан, то мы считаем, что только что завершили горизонтальный проход, образованный текущей ячейкой и всеми ячейками слева, вырезавшими проходы, ведущими в неё. Случайным образом выбираем одну ячейку вдоль этого прохода и вырезаем проход, ведущий из неё вверх (это должна быть текущая ячейка, если соседняя ячейка не вырезала проход). В то время как лабиринт двоичного дерева всегда поднимается от самой левой ячейки горизонтального прохода, лабиринт sidewinder поднимается вверх от случайной ячейки. Единственный тип ячеек, который не может существовать в лабиринте sidewinder — это тупик с ведущим вниз проходом, потому что это будет противоречить правилу, что каждый проход, ведущий вверх, возвращает нас к началу. Лабиринты sidewinder склонны к появлению элитного решения, при котором правильный путь оказывается очень прямым, но рядом с ним есть множество длинных ошибочных путей, ведущих сверху вниз.

## Алгоритм Эллера

Алгоритм Эллера является еще одним эффективным методом генерации лабиринта. Он работает построчно, объединяя клетки в группы и случайным образом удаляя стены между соседними группами. Этот алгоритм обладает высокой производительностью и создает лабиринты с хорошей структурой. Это особый алгоритм, потому что он не только быстрый, но и не имеет очевидной смещённости или недостатков; кроме того, при его создании память используется наиболее эффективно. Для него даже не требуется, чтобы в памяти находился весь лабиринт, он использует объём, пропорциональный размеру строки. Он создаёт лабиринт построчно, после завершения генерации строки алгоритм больше её не учитывает. Каждая ячейка в строке содержится во множестве; две ячейки принадлежат одному множеству, если между ними есть путь по уже созданному лабиринту. Эта информация позволяет вырезать проходы в текущей строке без создания петель или изолированных областей. Создание строки состоит из двух частей: случайным образом соединяем соседние в пределах строки ячейки, т. е. вырезаем горизонтальные проходы, затем случайным образом соединяем ячейки между текущей и следующей строками, т. е. вырезаем вертикальные проходы. При вырезании горизонтальных проходов мы не соединяем ячейки, уже находящиеся в одном множестве (потому что иначе создастся петля), а при вырезании вертикальных проходов мы должны соединить ячейку, если она имеет единичный размер (потому что, если её оставить, она создаст изолированную область). При вырезании горизонтальных проходов мы соединяем ячейки, если они находятся в одинаковом множестве (потому что теперь между ними есть путь), а при вырезании вертикальных проходов, когда не соединяемся с ячейкой, помещаем её в отдельное множество (потому что теперь она отделена от остальной части лабиринта). Создание начинается с того, что перед соединением ячеек в первой строке каждая ячейка имеет собственное множество. Создание завершается после соединения ячеек в последней строке. Существует особое правило завершения: к моменту завершения каждая ячейка должна находиться в одинаковом множестве, чтобы избежать изолированных областей. (Последняя строка создаётся объединением каждой из пар соседних ячеек, ещё не находящихся в одном множестве.) Лучше всего реализовывать множество с помощью циклического двусвязного списка ячеек (который может быть просто массивом, привязывающим ячейки к парам ячеек с обеих сторон того же множества), позволяющего выполнять за постоянное время вставку, удаление и проверку нахождения соседних ячеек в одном множестве. Проблема этого алгоритма заключается в несбалансированности обработки разных краёв лабиринта; чтобы избежать пятен в текстурах нужно выполнять соединение и пропуск соединения ячеек в правильных пропорциях.

# Практическая часть

## Техническая часть

Перечисление wind используется для обозначения местоположения пользователя в программе.

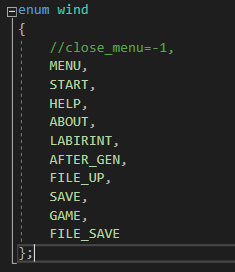


Рисунок 1 — Перечисление wind

В программе присутствуют функции для выделения и очищения динамической памяти: prof\_calloc и free\_calloc. Функции работы с линейным списком: push (добавление элемента в список), free\_spisok (перемещение указателя на следующий элемент списка). Функция init\_matrix для построения матрицы, в которую будет записываться лабиринт. Функция razmer\_kletki для оптимального вывода лабиринта, учитывая его размер. Функции для сохранения сгенерированного лабиринта в файл. Функция show\_procents для вывода прогресса выполнения программы.

### Структура game\_fichi

Структура game\_fichi отвечает за данные, которые в целом описывают работу программы. Заголовок файла имеет поля, представленные в таблице 1

Таблица 1 — Поля структуры game\_fichi

|  |  |
| --- | --- |
| Поле структуры | Поле структуры |
| size\_labirint; | Получение размера лабиринта от пользователя |
| Поле структуры | Поле структуры |
| size\_for\_matrix; | Преобразование размера лабиринта от пользователя к размеру, необходимому для работы алгоритма |
| size\_kl; | Нахождение размера клеток для отображения лабиринта |
| choice\_labirint; | Получение от пользователя какой лабиринт генерировать |
| game\_procces: | Процессы программы |
| status; | Отображение какого окна сейчас происходит |
| flag; | Флаг для возвращения к нужному окну после кнопки “help” |
| procent; | Отображение статуса обработки |

### Структура Node

Структура Node представляет собой односвязный линейный список, который используется для реализации алгоритма Эллера. В последующей оптимизации программы планируется использовать список в других двух алгоритмах. Каждый элемент списка содержит переменную типа int для хранения данных и указатель на следующий элемент списка.

### Структура button\_menu

Структура button\_menu содержит поля для инициализации кнопок в меню: массив типа char для названия каждой кнопки, массив типа float для координат кнопки в окне и переменную типа bool для маркера нажатия кнопки.

### Реализация алгоритма растущего дерева

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 2 — Реализация алгоритма растущего дерева

Алгоритм работает через два вложенных цикла. Сначала создаётся матрица и инициализируется нулём или единицей, соответственно, полем или стеной. Обрабатывается каждая ячейка матрицы по условиям алгоритма, если текущая ячейка не подходит под все условия, то рандомно выбирается удалить стену сверху или снизу.

### Реализация алгоритма sidewinder

Алгоритм стандартного смещения:

1. Выбрать начальный ряд;
2. Выбрать начальную клетку ряда и сделать её текущей;
3. Инициализировать пустое множество;
4. Добавить текущую клетку в множество;
5. Решить, прокладывать ли путь направо;
6. Если проложили, то перейти к новой клетке и сделать её текущей. Повторить шаги 3-6;
7. Если не проложили, выбираем случайную клетку множества и прокладываем оттуда путь наверх. Переходим на следующий ряд и повторяем 2-7;
8. Продолжать, пока не будет обработан каждый ряд;

Для проверки на множество использовался простой счётчик k, который считал текущее количество подряд идущих полей лабиринта. Когда алгоритм доходит до выбора стен сверху, проверяется условие на количество полей в множестве: если в множестве находится только одна клетка, то удаляется стенка сверху, иначе рандомно выбирается стена, которую нужно удалить.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 3 — Реализация алгоритма sidewinder

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 4 — Реализация алгоритма sidewinder

### Реализация алгоритма Эллера

Алгоритм:

1. Инициализируйте ячейки первого ряда так, чтобы каждая из них находилась в своем собственном множестве.
2. Затем случайным образом объедините смежные ячейки, но только если они не находятся в одном множестве. При объединении смежных ячеек объедините множества обеих ячеек в одно множество, что означает, что все ячейки в обоих множествах теперь связаны (существует путь, который соединяет любые две ячейки в множестве).
3. Для каждого множества случайным образом создайте вертикальные соединения вниз к следующему ряду. Каждое оставшееся множество должно иметь как минимум одно вертикальное соединение. Ячейки в следующем ряду, таким образом, связаны должны иметь то же самое множество, что и ячейка над ними.
4. Заполните следующий ряд, помещая все оставшиеся ячейки в их собственные множества.
5. Повторяйте до достижения последнего ряда.
6. Для последнего ряда объедините все смежные ячейки, которые не имеют общего множества, и опустите вертикальные соединения.

Основной цикл по количеству строк происходит в процедуре void eller(int size\_lab, int\*\* matrix), в ней создаётся линейный список на строку. А также, создается одномерный массив на хранение множеств и их количества. Процедура void black\_string(Node\* head, int y, int size, int\*\* matrix) позволяет быстро создавать верхнюю и нижнюю границу лабиринта, потому в них нет необходимости выбирать удалять стену или нет. Функция int prof\_right\_bord\_el(Node\* p, int y, int size, int count\_kl, int number\_lot, int\* lot, int\*\* matrix) проверяет алгоритм на правую границу и возвращает единицу, если текущая ячейка граница и ноль, если нет. Процедура void right\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix) создаёт стены справа от текущей ячейки. Процедура void bottom\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix) создаёт стены снизу от предыдущей строки.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 5 — Реализация алгоритма Эллера

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 6 — Реализация алгоритма Эллера

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Рисунок 7 — Реализация алгоритма Эллера

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 8 — Реализация алгоритма Эллера

## Графическая часть

### Работа с OpenGL

Для реализации графической составляющей был использован кроссплатформенный интерфейс прикладного программирования OpenGL (Open Graphics Library), а именно библиотека GLUT, которая в основном отвечает за системный уровень операций ввода-вывода при работе с операционной системой.

В работе программы использовались следующие функции: glClearColor – установление цвета окна, glMatrixMode(GL\_PROJECTION) – создание матрицы, gluOrtho2D(0, 640, 480, 0) – установление орфографической проекции, glScalef – изменение размера для отображения текста, glFinish() и glLoadIdentity() – для завершения работы с отрисовкой и glRectf – для отрисовки квадратов, на которых строится вся отрисовка лабиринта.

### Работа с Win Api

Win Api – это основной набор интерфейсов прикладного программирования (API), обеспечивает доступ к функциям операционной системы Windows, таких как создание окон, управление ресурсами, работа с файлами, реагирование на события и еще многое другое.

Главный цикл происходит в WINAPI WinMain, в котором создаётся главное окно для работы программы.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 9 – Цикл для создания окна

WindowProc отвечает за обработку сообщений в главном окне через получение сообщений от мыши(lParam) и клавиатуры(wParam) .

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 10 – ­Обработка сообщений при нажатии мышки

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 11 – Обработка сообщений при нажатии клавиатуры

### Работа с диалоговыми окнами

Диалоговое окно – это временное окно, которое создает прикладная программа, чтобы получить данные, вводимые пользователем. Было создано три диалоговых окна: первое отвечает за получение размера лабиринта, второе за выбор алгоритма и третье за получение названия файла для сохранения и загрузки. Такое окно создаётся в проекте visual studio через ресурсные файлы.

Изображение выглядит как текст, дисплей, снимок экрана, программное обеспечение

Автоматически созданное описание

Рисунок 12 – Пример диалогового окна

Получение сообщений происходит в процедуре callback. При нажатии на кнопку OK устанавливается дескриптор окна редактирования, c помощью функции GetDlgItem. Через функцию GetWindowText – в массив char\* text\_dailog записываются данные с окна.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок 13 — Обработка сообщений с окна

### Работа с stb\_easy\_font

stb\_easy\_font — это библиотека для создания простых текстовых графиков в программировании. Она предоставляет простой и удобный способ отображения текста на экране без необходимости использования сложных графических библиотек. Она позволяет выводить текст различного размера и цвета, а также позволяет создавать простые графические элементы, такие как рамки и прямоугольники.

# Исследовательская часть

## Сравнение алгоритмов

### Сравнение алгоритмов по скорости работы

Growing Tree. Скорость работы этого алгоритма зависит от выбранной стратегии. Если выбирается случайная ячейка, то алгоритм может быть медленным, так как может зациклиться в тупиках. Если выбирается самая старая ячейка, то алгоритм будет работать быстрее, так как будет строить путь прямо к началу лабиринта.

Sidewinder. Этот алгоритм работает эффективно и быстро для генерации прямых путей в лабиринте. Поскольку проход осуществляется по каждой строке и соединение происходит только с соседней ячейкой, алгоритм имеет линейную сложность и работает быстро.

Алгоритм Эллера. Этот алгоритм работает построчно и соединяет ячейки в каждой строке с ячейками в следующей строке. В каждой строке алгоритм выполняет объединение наборов ячеек, что может быть затратным по времени.

Итак, sidewinder является самым быстрым из данных алгоритмов, особенно для прямых путей в лабиринте, growing tree может быть средней или выше среднего скорости, в зависимости от выбранной стратегии, алгоритм Эллера может быть средней или ниже среднего скорости, особенно при работе с большими лабиринтами.

Результаты тестирования программы на лабиринте, размером 30 клеток представлены в таблице 2.

Таблица 2 — Сравнение скорости выполнения алгоритма

|  |  |
| --- | --- |
| Алгоритм | Время генерации лабиринта |
| Growing tree | 1 минута и 12 секунд |
| Sidewinder | 1 минута и 1 секунда |
| Алгоритм Эллера | 1 минута и 6 секунд |

### Сравнение алгоритмов по объему используемой памяти

Рассмотрим объём дополнительной памяти или стека, необходимый для реализации алгоритма. Эффективные алгоритмы требуют только битовой карты самого лабиринта, в то время как другие требуют объёма памяти, пропорционального одной строке (N), или пропорционального количеству ячеек (N2) (например, growing tree). Некоторым алгоритмам даже не нужно иметь в памяти весь лабиринт, и они могут добавлять части лабиринта бесконечно, например, алгоритму Эллера нужен объём памяти для хранения строки, но большего ему не требуется, потому что достаточно хранить только одну строку лабиринта. Алгоритму Sidewinder тоже нужно хранить только одну строку лабиринта. Исходя из этого, sidewinder является самым не затратным по памяти, а growing tree - самым затратным.

# исследование по времени генерации

## Графики

В результате измерений был построен график, представленный на рисунке 14.

Рисунок 14 — График зависимости времени от размера лабиринта

Исходя из полученного графика, можно сделать вывод, что время генерации лабиринта прямо пропорционально размеру генерируемого лабиринта.

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы были закреплены навыки работы с различными структурами данных: перечисление, структура, список. Также были изучены и составлены алгоритмы, основанные на генерации лабиринтов.

Благодаря данным, полученным в результате тестов, была составлена таблица и графики зависимости общего времени от размера лабиринта. Как видно из графиков в большинстве случаев при увеличении размера лабиринта время работы алгоритма возрастает.

Таким образов, при разработке алгоритмов были получены хорошие результаты. Навыки, полученные при выполнении данной работы, пригодятся в будущем не только для разработки программ, но и для улучшения процессов, происходящих в практической жизни.

список использованной литературы

1. Лабиринты: классификация, генерирование, поиск решений [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/445378/ (дата обращения 10.08.2023)
2. Классические алгоритмы генерации лабиринтов. Часть 1: вступление [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/320140/ (дата обращения 10.08.2023)
3. Maze Generation: Eller’s Algorithm [Электронный ресурс] – Режим доступа: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm (дата обращения 14.08.2023)
4. Просто алгоритм генерации лабиринта на C/C++ [Электронный ресурс] – Режим доступа: https://ru.stackoverflow.com/questions/482663/Простой-алгоритм-генерации-лабиринта-на-c- (дата обращения 30.05.2023)
5. Генерация лабиринтов [Электронный ресурс] – Режим доступа: https://[algolist.manual.ru/games/maze.php](http://algolist.manual.ru/games/maze.php) (дата обращения 6.06.2023)
6. Генерация и решение лабиринта с помощью метода поиска в глубину по графу [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/262345/ (дата обращения 4.08.2023)
7. Неприлично простая реализация неприлично простого алгоритма генерации лабиринта [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/319532/ (дата обращения 9.05.2023)
8. Maze Generator in C [Электронный ресурс] – Режим доступа: https:// codereview.stackexchange.com/questions/277817/ (дата обращения 19.07.2023)
9. OpenGL Particle System Maze Generation and Gradient Coloring with Sidewinder Algorithm Visual [Электронный ресурс] – Режим доступа: https://www.youtube.com/watch?v=x17KtEEYqk8 (дата обращения 3.09.2023)
10. Генерация лабиринтов: Алгоритм Эллера [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/667576/ (дата обращения 23.08.2023)
11. Алгоритм Эллера для генерации лабиринтов [Электронный ресурс] – Режим доступа: https://habr.com/ru/articles/176671/ (дата обращения 23.08.2023)

# ПРИЛОЖЕНИЕ А

Листинг программы «coursework.c»

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <GL/glut.h>

#include <C:\Users\1\source\repos\курсач\_на\_допсу\glut.h>

#include <C:\Users\1\source\repos\курсач\_на\_допсу/stb\_easy\_front.h>

#include <C:\Users\1\source\repos\курсач\_на\_допсу\resource.h>

#include <math.h>

#include <time.h>

#include <Windows.h> //Для sleep

#pragma comment(lib, "opengl32.lib")

//#pragma comment(linker,"/SUBSYSTEM:WINDOWS /ENTRY:mainCRTStartup")

struct ficha {

int size\_labirint;

int size\_for\_matrix;

int size\_kl;

int choice\_labirint;

int game\_procces;// процессы самой программы лабиринта

int status;//отображение какого окна происходит сейчас

int flag;//флаг возвращение обратно к нужному окну после helpa

int procent;

}game\_fichi;

int\*\* matrix = NULL;

typedef struct {

int data;

struct Node\* next;

} Node;

int width, height;

HWND edit\_1;

HWND edit\_2;

HWND edit\_3;

TCHAR text\_dialog[20];

char c\_text[20];

enum wind

{

//close\_menu=-1,

MENU,

START,

HELP,

ABOUT,

LABIRINT,

AFTER\_GEN,

FILE\_UP,

SAVE,

GAME,

FILE\_SAVE

};

typedef struct BUTTON //для кнопок

{

char name[20];

float variable\_size[4];

BOOL click;

} button\_menu;

button\_menu button[] =

{

{"start",{200.0, 50.0, 450.0, 125.0}, FALSE},

{"help",{200.0, 200.0, 450.0, 275.0}, FALSE},

{"about",{200.0, 350.0, 450.0, 425.0}, FALSE},

{"enter size",{200.0, 100.0, 450.0,175.0},FALSE},

{"upload file",{200.0, 275.0, 450.0,350.0},FALSE},

};

int btn\_size = sizeof(button) / sizeof(button[0]);

#define file\_not\_opend 0xff

#define stop 2

#define TRUE 1

#define one\_cord 1

int\*\* prof\_calloc(int\*\* A);

void free\_calloc(int\*\* A);

void razmer\_kletki(void);

//void glut\_draw\_labirint(void);

void black\_rect(int x1, int y1, int x2, int y2);

void white\_rect(int x1, int y1, int x2, int y2);

void init\_matrix(int\*\* matrix);

void growing\_tree(int\*\* matrix);

void sidewinder(int\*\* matrix);

void push(Node\*\* head, Node\*\* tail, int size);//односвязный список как очередь

void free\_spisok(Node\* head);

void black\_rect(int x1, int y1, int x2, int y2);

void white\_rect(int x1, int y1, int x2, int y2);//чисто квадратик рисует

void eller(int size\_lab, int\*\* matrix);

void right\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix);

void bottom\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix);

void black\_string(Node\* head, int y, int size, int\*\* matrix);

int prof\_right\_bord\_el(Node\* p, int y, int size, int count\_kl, int number\_lot, int\* lot, int\*\* matrix);

void print\_string(float x, float y, char\* text, float r, float g, float b)//Функция для вывода текста на экран. Была в "stb\_easy\_font.h". Передаём координаты, текст, и цвет

{

static char buffer[99999]; // ~500 chars

int num\_quads;

num\_quads = stb\_easy\_font\_print(x, y, text, NULL, buffer, sizeof(buffer));

glColor3f(r, g, b);

glEnableClientState(GL\_VERTEX\_ARRAY);

glVertexPointer(2, GL\_FLOAT, 16, buffer);

glDrawArrays(GL\_QUADS, 0, num\_quads \* 4);

glDisableClientState(GL\_VERTEX\_ARRAY);

}

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);//основное окно/меню

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM); //диалоговое окно на получение размера лабиринта

BOOL CALLBACK DlgProc\_2(HWND, UINT, WPARAM, LPARAM); //диалоговое окно на получение алгоритма лабиринта

BOOL CALLBACK DlgProc\_3(HWND, UINT, WPARAM, LPARAM); //диалоговое окно на сохранение названия лабиринта в файл

void EnableOpenGL(HWND hwnd, HDC\*, HGLRC\*);

void DisableOpenGL(HWND, HDC, HGLRC);

void button\_show(button\_menu button)

{

if (button.click)glColor3f(0.3, 0.3, 0.3);//mint

else glColor3f(0.6, 0.9, 0.8);

glRectf(button.variable\_size[0], button.variable\_size[1], button.variable\_size[2], button.variable\_size[3]);

}

BOOL button\_in\_click(int x, int y, button\_menu btn)

{

return(x > btn.variable\_size[0]) && (x < btn.variable\_size[2])

&& (y > btn.variable\_size[1]) && (y < btn.variable\_size[3]);

}

BOOL tap\_button(int x, int y, float upx, float upy, float dwx, float dwy) {//Для считывания нажатия мышкой. В неё передаём координаты краёв кнопок. Т.е. если в пределах этих краёв тыкнули курсором, то нажатие произошло

return(x > upx) && (x < dwx) && (y > upy) && (y < dwy);

}

void init\_(void)

{

glClearColor(0.3, 0.3, 0.4, 0.0);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, 640, 480, 0);//орфографической проекции

}

void show\_menu(void)

{

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

for (int i = 0; i < 3; i++)

{

button\_show(button[i]);

}

glScalef(4, 4, 1);

print\_string(61, 18, " Start ", 0.04, 0.2, 0.2);

print\_string(60, 55, " Help ", 0.04, 0.2, 0.2);

print\_string(61, 94, " About ", 0.04, 0.2, 0.2);

}

void show\_start(void)

{

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

for (int i = 3; i < 5; i++)

{

button\_show(button[i]);

}

glScalef(4, 4, 1);

print\_string(49, 30, " Enter size ", 0.04, 0.2, 0.2);

print\_string(49, 75, " Upload file ", 0.04, 0.2, 0.2);

}

void show\_about(void)

{

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

glScalef(3, 3, 1);

print\_string(30, 30, "Created by : \nRomanova Ekaterina Evgenievna\nMatylitskaya Alena Andreevna\nRussia,\nSaint - Petersburg\nPolytech University, IKiZI\n4851004 / 20001\n\nclick anywhere to continue", 0.87, 0.87, 0.87);

}

void show\_help(void)

{

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

glScalef(3, 3, 1);

print\_string(10, 10, "'F1' - help\n'esc' - exit\n'R' - restart labirint\n'1' - create algoritm 'Growing Tree'\n'2' - create algoritm 'Sidewinder'\n'3' - create algoritm 'Eller'\nclick anywhere to continue", 0.87, 0.87, 0.87);

}

void show\_procents(int cur)

{

/\*ShowWindow(GetConsoleWindow(), SW\_SHOW);

Sleep(1000);\*/

game\_fichi.procent = (int)(( cur/ (game\_fichi.size\_for\_matrix) )\* 100);

system("cls");

printf("Processing: %d%%", (int)game\_fichi.procent);

Sleep(15);

}

void show\_labirint(void)

{

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

if (game\_fichi.choice\_labirint == 1)

{

matrix = prof\_calloc(matrix);

init\_matrix(matrix);

growing\_tree(matrix);

}

else if (game\_fichi.choice\_labirint == 2)

{

matrix = prof\_calloc(matrix);

init\_matrix(matrix);

sidewinder(matrix);

}

else if (game\_fichi.choice\_labirint == 3)

{

matrix = prof\_calloc(matrix);

eller(game\_fichi.size\_for\_matrix ,matrix);

}

glFinish();

glLoadIdentity();

}

void rewrite\_file\_matrix(int\*\* matrix, char\* name)

{

FILE\* save\_l = fopen(name, "w");//файл на сохранение лабиринта

if (save\_l == NULL)

exit(1);

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++)

{

for (int j = 0; j < game\_fichi.size\_for\_matrix; j++)

{

if (matrix[i][j] == 0)// поле иначе там стены

fputc('0', save\_l);

else

fputc('1', save\_l);

}

fputc('\n', save\_l);

}

free\_calloc(matrix);

fclose(save\_l);

}

void draw\_upload\_file(char\* name)

{

FILE\* upload\_file = fopen(name, "r");

if (upload\_file == NULL)

exit(1);

else

{

char str[4096] = { 0 };

fgets(str, 4096, upload\_file);

game\_fichi.size\_labirint = strlen(str)-1;

razmer\_kletki();

fseek(upload\_file, 0, SEEK\_SET);

int count\_for\_procents = 0;

//show\_procents(count\_for\_procents);

count\_for\_procents++;

init\_();

glClear(GL\_COLOR\_BUFFER\_BIT);

int i = 0, j = 0;

while (!feof(upload\_file))

{

char c = fgetc(upload\_file);

if (c == '1')

{

black\_rect(j, i, j + 1, i + 1);

j++;

}

else if(c == '0')

{

white\_rect(j, i, j + 1, i + 1);

j++;

}

else

{

count\_for\_procents += i;

//show\_procents(count\_for\_procents);

i++; j = 0;

}

}

glFinish();

}

fclose(upload\_file);

}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

WNDCLASSEX wcex;

HWND hwnd;

HDC hDC;

HGLRC hRC;

MSG msg;

BOOL bQuit = FALSE;

/\* register window class \*/

wcex.cbSize = sizeof(WNDCLASSEX);

wcex.style = CS\_OWNDC;

wcex.lpfnWndProc = WindowProc;

wcex.cbClsExtra = 0;

wcex.cbWndExtra = 0;

wcex.hInstance = hInstance;

wcex.hIcon = LoadIcon(NULL, IDI\_APPLICATION);

wcex.hCursor = LoadCursor(NULL, IDC\_ARROW);

wcex.hbrBackground = (HBRUSH)GetStockObject(BLACK\_BRUSH);

wcex.lpszMenuName = NULL;

wcex.lpszClassName = L"GLSample";

wcex.hIconSm = LoadIcon(NULL, IDI\_APPLICATION);

if (!RegisterClassEx(&wcex))

return 0;

///\* create main window \*/

hwnd = CreateWindowEx(0, L"GLSample",L"OpenGL Menu",WS\_OVERLAPPEDWINDOW,500,200,800,600,NULL,NULL,hInstance,NULL);

game\_fichi.choice\_labirint = 0;

game\_fichi.size\_labirint = 0;

game\_fichi.game\_procces = 0;

game\_fichi.flag =-6;

game\_fichi.status = MENU;

ShowWindow(hwnd, nCmdShow);

/\* enable OpenGL for the window \*/

EnableOpenGL(hwnd, &hDC, &hRC);

/\* program main loop \*/

while (!bQuit)

{

/\* check for messages \*/

if (PeekMessage(&msg, NULL, 0, 0, PM\_REMOVE))

{

/\* handle or dispatch messages \*/

if (msg.message == WM\_QUIT)

{

bQuit = TRUE;

}

else

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

else

{

/\* OpenGL animation code goes here \*/

init\_();

glClearColor(0.3, 0.3, 0.4, 0.0);

if (game\_fichi.status == MENU)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

show\_menu();

if (game\_fichi.game\_procces == FILE\_SAVE)

{

rewrite\_file\_matrix(matrix, c\_text);

int mess = MessageBox(hwnd, L"Файл сохранён",L"Message", MB\_OK);

if (mess == IDOK)

game\_fichi.game\_procces = 0;

}

}

else if (game\_fichi.status == HELP)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

show\_help();

}

else if (game\_fichi.status == START)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

show\_start();

}

else if (game\_fichi.status == ABOUT)

{

glClear(GL\_COLOR\_BUFFER\_BIT);

show\_about();

glLoadIdentity();

}

else if (game\_fichi.status == LABIRINT)

{

if (game\_fichi.game\_procces == 0)

{

DialogBoxParam(hInstance, MAKEINTRESOURCE(IDD\_DIALOG1), 0, (DlgProc), 0);

DialogBoxParam(hInstance, MAKEINTRESOURCE(IDD\_DIALOG2), 0, (DlgProc\_2), 0);

}

show\_labirint();

//glut\_draw(hDC);

game\_fichi.game\_procces = GAME;

game\_fichi.status = AFTER\_GEN;

}

else if (game\_fichi.status == AFTER\_GEN)

{

Sleep(1300);

int mess = MessageBox(hwnd, L"Хотите сохранить лабиринт?", L"Message", MB\_YESNO);

switch (mess)

{

case IDYES:

{

DialogBoxParam(hInstance, MAKEINTRESOURCE(IDD\_DIALOG3), 0, (DlgProc\_3), 0);

game\_fichi.status = MENU;// return menu

game\_fichi.game\_procces = 0;

game\_fichi.game\_procces = FILE\_SAVE;

break;

}

case IDNO:

{

game\_fichi.status = MENU;// return menu

game\_fichi.game\_procces = 0;

if (matrix != NULL)

{

free\_calloc(matrix);

}

break;

}

}

}

else if (game\_fichi.status == FILE\_UP)

{

if (game\_fichi.game\_procces == 0)

{

DialogBoxParam(hInstance, MAKEINTRESOURCE(IDD\_DIALOG3), 0, (DlgProc\_3), 0);

}

draw\_upload\_file(c\_text);

game\_fichi.game\_procces = GAME;

}

SwapBuffers(hDC);

//glLoadIdentity();

}

}

/\* shutdown OpenGL \*/

DisableOpenGL(hwnd, hDC, hRC);

/\* destroy the window explicitly \*/

DestroyWindow(hwnd);

return msg.wParam;

}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

switch (uMsg)

{

case WM\_CLOSE:

PostQuitMessage(0);

break;

case WM\_LBUTTONDOWN:

if (game\_fichi.status == MENU)

{

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 200.0, 200.0, 450.0, 275.0))

{

game\_fichi.status = HELP;

}

else if (tap\_button(LOWORD(lParam), HIWORD(lParam), 200.0, 50.0, 450.0, 125.0))

{

game\_fichi.status = START;

}

else if (tap\_button(LOWORD(lParam), HIWORD(lParam), 200.0, 350.0, 450.0, 425.0))

{

game\_fichi.status = ABOUT;

}

}

else if ((game\_fichi.status == HELP && game\_fichi.game\_procces == 0) || (game\_fichi.status == ABOUT && game\_fichi.game\_procces == 0))

{

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 0, 0, 800, 800)) //То, куда бы мы не тыкнули, Нас возвращает в меню

{

game\_fichi.status = MENU;

}

}

else if (game\_fichi.status == HELP && game\_fichi.flag !=-6)

{

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 0, 0, 800, 800)) //То, куда бы мы не тыкнули, Нас возвращает в win flags

{

game\_fichi.status = game\_fichi.flag;

game\_fichi.flag = -6;

}

}

else if (game\_fichi.status == START)

{

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 200.0, 100.0, 450.0, 175.0))

{

game\_fichi.status = LABIRINT;

}

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 200.0, 275.0, 450.0, 350.0))

{

game\_fichi.status = FILE\_UP;

}

}

else if (game\_fichi.status == FILE\_UP && game\_fichi.game\_procces == GAME)

{

if (tap\_button(LOWORD(lParam), HIWORD(lParam), 0, 0, 800, 800)) //То, куда бы мы не тыкнули, вернуться в меню

{

game\_fichi.status = MENU;

}

}

break;

case WM\_MOUSEMOVE:

for (int i = 0; i < 5; i++)

button[i].click = button\_in\_click(LOWORD(lParam), HIWORD(lParam), button[i]);

break;

case WM\_SIZE:

width = LOWORD(lParam);

height = HIWORD(lParam);

glViewport(0, 0, width, height);

glLoadIdentity();

float k = width / (float)height;

glOrtho(-k, k, -1, 1, -1, 1);

break;

case WM\_DESTROY:

return 0;

case WM\_KEYDOWN:

{

switch (wParam)

{

case VK\_ESCAPE:

PostQuitMessage(0);

break;

case VK\_F1:

game\_fichi.flag = game\_fichi.status;

game\_fichi.status = HELP;

break;

case 0x52: //R

game\_fichi.status = LABIRINT;

break;

}

}

break;

default:

return DefWindowProc(hwnd, uMsg, wParam, lParam);

}

return 0;

}

BOOL CALLBACK DlgProc(HWND hwmd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

switch (uMsg) //обрабатываем сообщения

{

case WM\_INITDIALOG: //инициализация с добавлениями компонентов

edit\_1 = GetDlgItem(hwmd, IDC\_EDIT1);

SetFocus(edit\_1);

break;

case WM\_COMMAND: //здесь обработчик команд полей ввода

switch (LOWORD(wParam))

{

case IDOK:

//game\_fichi.size\_kl = 33;

GetWindowText(edit\_1, text\_dialog, 5);

wcstombs(c\_text, text\_dialog, 5);

game\_fichi.size\_labirint = atoi(c\_text);

razmer\_kletki();

EndDialog(hwmd, 0);

break;

case IDCANCEL:

EndDialog(hwmd, 0);

game\_fichi.status = START;

return FALSE;

}

break;

case WM\_CLOSE: //обработчик закрытия

EndDialog(hwmd, 0);

return FALSE;

}

return FALSE;

}

BOOL CALLBACK DlgProc\_2(HWND hwmd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

switch (uMsg) //обрабатываем сообщения

{

case WM\_INITDIALOG: //инициализация с добавлениями компонентов

edit\_2 = GetDlgItem(hwmd, IDC\_EDIT2);

SetFocus(edit\_2);

break;

case WM\_COMMAND: //здесь обработчик команд полей ввода

switch (LOWORD(wParam))

{

case IDOK:

GetWindowText(edit\_2, text\_dialog, 2);

wcstombs(c\_text, text\_dialog, 5);

game\_fichi.choice\_labirint = atoi(c\_text);

memset(c\_text, '0', 20);

EndDialog(hwmd, 0);

break;

case IDCANCEL:

EndDialog(hwmd, 0);

game\_fichi.status = START;

return FALSE;

}

break;

case WM\_CLOSE: //обработчик закрытия

EndDialog(hwmd, 0);

return FALSE;

}

return FALSE;

}

BOOL CALLBACK DlgProc\_3(HWND hwmd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

switch (uMsg) //обрабатываем сообщения

{

case WM\_INITDIALOG: //инициализация с добавлениями компонентов

edit\_3 = GetDlgItem(hwmd, IDC\_EDIT3);

SetFocus(edit\_3);

break;

case WM\_COMMAND: //здесь обработчик команд полей ввода

switch (LOWORD(wParam))

{

case IDOK:

GetWindowText(edit\_3, text\_dialog, 20);

wcstombs(c\_text, text\_dialog, 20);

EndDialog(hwmd, 0);

break;

case IDCANCEL:

EndDialog(hwmd, 0);

game\_fichi.status = START;

return FALSE;

}

break;

case WM\_CLOSE: //обработчик закрытия

EndDialog(hwmd, 0);

return FALSE;

}

return FALSE;

}

void EnableOpenGL(HWND hwnd, HDC\* hDC, HGLRC\* hRC)

{

PIXELFORMATDESCRIPTOR pfd;

int iFormat;

/\* get the device context (DC) \*/

\*hDC = GetDC(hwnd);

/\* set the pixel format for the DC \*/

ZeroMemory(&pfd, sizeof(pfd));

pfd.nSize = sizeof(pfd);

pfd.nVersion = 1;

pfd.dwFlags = PFD\_DRAW\_TO\_WINDOW |

PFD\_SUPPORT\_OPENGL | PFD\_DOUBLEBUFFER;

pfd.iPixelType = PFD\_TYPE\_RGBA;

pfd.cColorBits = 24;

pfd.cDepthBits = 16;

pfd.iLayerType = PFD\_MAIN\_PLANE;

iFormat = ChoosePixelFormat(\*hDC, &pfd);

SetPixelFormat(\*hDC, iFormat, &pfd);

/\* create and enable the render context (RC) \*/

\*hRC = wglCreateContext(\*hDC);

wglMakeCurrent(\*hDC, \*hRC);

}

void DisableOpenGL(HWND hwnd, HDC hDC, HGLRC hRC)

{

wglMakeCurrent(NULL, NULL);

wglDeleteContext(hRC);

ReleaseDC(hwnd, hDC);

}

void razmer\_kletki(void)

{

if ((game\_fichi.size\_labirint > 0) && (game\_fichi.size\_labirint <= 7))

{

game\_fichi.size\_kl = 33;

}

else if ((game\_fichi.size\_labirint > 7) && (game\_fichi.size\_labirint < 15))

{

game\_fichi.size\_kl = 16;

}

else if ((game\_fichi.size\_labirint >= 15) && (game\_fichi.size\_labirint < 25))

{

game\_fichi.size\_kl = 9;

}

else if ((game\_fichi.size\_labirint >= 25) && (game\_fichi.size\_labirint < 40))

{

game\_fichi.size\_kl = 5;

}

else

{

game\_fichi.size\_kl = 5;

}

}

void black\_rect(int x1, int y1, int x2, int y2)

{

glColor3f(0.0, 0.0, 0.0);

glRectf(x1 \* game\_fichi.size\_kl, y1 \* game\_fichi.size\_kl, x2 \* game\_fichi.size\_kl, y2 \* game\_fichi.size\_kl);

glFinish();

Sleep(1);

}

void white\_rect(int x1, int y1, int x2, int y2)

{

glColor3f(1.0, 1.0, 1.0);

glRectf(x1 \* game\_fichi.size\_kl, y1 \* game\_fichi.size\_kl, x2 \* game\_fichi.size\_kl, y2 \* game\_fichi.size\_kl);

glFinish();

Sleep(1);

}

void init\_matrix(int\*\* matrix)

{

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++)

{

int j = 0;

for (; j < game\_fichi.size\_for\_matrix; j++)

{

if (i % 2 == 0 || j % 2 == 0)

{

matrix[i][j] = 1;//стена

}

else matrix[i][j] = 0;//поле

}

matrix[i][j - 1] = stop;//правая граница лабиринта

}

}

void growing\_tree(int\*\* matrix)

{

int count\_for\_procents = 0;

//show\_procents(count\_for\_procents);

count\_for\_procents++;

srand(time(NULL));

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++)

{

for (int j = 0; j < game\_fichi.size\_for\_matrix; j++)

{

if (matrix[i][j] == 1 || matrix[i][j] == 2)//стены лабиринта

{

black\_rect(j, i, j + 1, i + 1);

if ((i == 0) || (i == game\_fichi.size\_for\_matrix - 1) || (j == 0) || (j == game\_fichi.size\_for\_matrix - 1))

matrix[i][j] = stop;

}

else

{

white\_rect(j, i, j + 1, i + 1);

if ((matrix[i - 1][j] == stop) && (matrix[i][j + 1] == stop))//это обход верхней правой клетки

{

black\_rect(j + 1, i, j + 2, i + 1);

j++;

}

else if (matrix[i - 1][j] == stop)// проход верхнего ряда

{

white\_rect(j + 1, i, j + 2, i + 1);

j++;

matrix[i][j] = 0;

}

else if (matrix[i][j + 1] == stop)// правый край стена

{

white\_rect(j, i - 1, j + 1, i);

black\_rect(j + 1, i, j + 2, i + 1);

matrix[i - 1][j] = 0;

j++;

}

else if (rand() % 2 == 1)//удалить стену с права

{

white\_rect(j + 1, i, j + 2, i + 1);

black\_rect(j, i - 1, j + 1, i);

matrix[i][j + 1] = 0;

j++;

}

else // по рандому получатся удалить стену сверху

{

white\_rect(j, i - 1, j + 1, i);

black\_rect(j + 1, i, j + 2, i + 1);

matrix[i - 1][j] = 0;

j++;

}

}

count\_for\_procents += j;

//show\_procents(count\_for\_procents);

}

}

}

void sidewinder(int\*\* matrix)

{

int count\_for\_procents = 0;

//show\_procents(count\_for\_procents);

count\_for\_procents++;

srand(time(NULL));

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++)

{

int k = 0;//количество полей, взятых в множество

for (int j = 0; j < game\_fichi.size\_for\_matrix; j++)

{

if (matrix[i][j] == 1 || matrix[i][j] == 2)//стены лабиринта

{

black\_rect(j, i, j + 1, i + 1);

if ((i == 0) || (i == game\_fichi.size\_for\_matrix - 1) || (j == 0) || (j == game\_fichi.size\_for\_matrix - 1))

matrix[i][j] = stop;

}

else

{

k++;

white\_rect(j, i, j + 1, i + 1);

if ((matrix[i - 1][j] == stop) && (matrix[i][j + 1] == stop))//это обход верхней правой клетки

{

black\_rect(j + 1, i, j + 2, i + 1);

j++;

}

else if (matrix[i - 1][j] == stop)// проход верхнего ряда

{

white\_rect(j + 1, i, j + 2, i + 1);

j++;

matrix[i][j] = 0;

}

else if (matrix[i][j + 1] == stop)//правый край

{

black\_rect(j + 1, i, j + 2, i + 1);

if (k != 1)

{

int f = rand() % k + 1; //выбор какую над какой клеткой удалить стену (первой, второй и тд);

f = j - k + f;

white\_rect(f, i - 1, f + 1, i);

matrix[i - 1][f] = 0;

k = 0;

}

else//у нас в множестве только одна клетка поэтому удалеем верхнюю стену

{

white\_rect(j, i - 1, j + 1, i);

matrix[i - 1][j] = 0;

}

j++;

}

else if (rand() % 2 == 1) //удаляем правую стену и считаем сколько клеток у нас

{

k++;

white\_rect(j + 1, i, j + 2, i + 1);

j++;

matrix[i][j] = 0;

}

else

{

black\_rect(j + 1, i, j + 2, i + 1);

if (k != 1)

{

int f = rand() % k + 1; //выбор какую над какой клеткой удалить стену (первой, второй и тд);

f = j - k + f;

white\_rect(f, i - 1, f + 1, i);

matrix[i - 1][f] = 0;

k = 0;

}

else//у нас в множестве только одна клетка поэтому удалеем верхнюю стену

{

white\_rect(j, i - 1, j + 1, i);

matrix[i - 1][j] = 0;

}

j++;

}

}

count\_for\_procents += j;

//show\_procents(count\_for\_procents);

}

}

}

int\*\* prof\_calloc(int\*\* A)

{

game\_fichi.size\_for\_matrix = (2 \* game\_fichi.size\_labirint) + 1;

A = (int\*\*)calloc(game\_fichi.size\_for\_matrix, sizeof(int));

if (A != NULL)

{

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++)

{

A[i] = (int\*)calloc(game\_fichi.size\_for\_matrix, sizeof(int));

}

}

else

exit(1);

return A;

}

void free\_calloc(int\*\* A)

{

for (int i = 0; i < game\_fichi.size\_for\_matrix; i++) {

free(A[i]);

}

free(A);

}

void push(Node\*\* head, Node\*\* tail, int size)

{

int val = 0;

while (val != size)

{

Node\* new\_ = (Node\*)malloc(sizeof(Node));

if (new\_ != NULL)

{

memset(new\_, 0x00, sizeof(Node));

new\_->data = val++;

new\_->next = NULL;

if ((\*head) == NULL)

{

(\*head) = new\_;

(\*tail) = new\_;

}

else

{

(\*tail)->next = new\_;

\*tail = (Node\*)(\*tail)->next;

}

}

}

}

void free\_spisok(Node\* head)

{

Node\* p = head;

while (p)

{

Node\* f = (Node\*)p->next;

free(p);

p = f;

}

//head = NULL;

}

void black\_string(Node\* head, int y, int size, int\*\* matrix)// для первой и последней строки

{

Node\* p = head;

while (p)

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 1;

p = (Node\*)p->next;

}

}

int prof\_right\_bord\_el(Node\* p, int y, int size, int count\_kl, int number\_lot, int\* lot, int\*\* matrix)

{

if (p->data == size - 1)

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);//самая правая граница всегdа

matrix[y][p->data] = 1;

lot[p->data] = -2;

p = (Node\*)p->next;

if (count\_kl != 0)

{

lot[number\_lot] = count\_kl;

}

return TRUE;

}

return 0;

}

void right\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix)

{

Node\* p = head;

int count\_kl = 0, number\_lot = p->data;

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);//левая граница всегdа

matrix[y][p->data] = 1;

lot[number\_lot++] = -2;//первый лот всегда пустой, потому что это левая стена

p = (Node\*)p->next;

while (p)

{

if (prof\_right\_bord\_el(p, y, size, count\_kl, number\_lot, lot, matrix) == TRUE) return;

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

//matrix[y][p->data] = 0;

count\_kl++;

p = (Node\*)p->next;

if (prof\_right\_bord\_el(p, y, size, count\_kl, number\_lot, lot, matrix) == TRUE) return;

if (rand() % 2 == TRUE)//ставим стену справа

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 1;

lot[p->data] = -1;//текущее множество относится к исключению под стеной, поэтому будет выбор ставить стену либо нет в нижнем ряду

lot[number\_lot] = count\_kl;

count\_kl = 0;

p = (Node\*)p->next;

number\_lot = p->data;

}

else//не ставим стену справа

{

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 0;

count\_kl++;

p = (Node\*)p->next;

}

}

}

void bottom\_bordes\_eller(Node\* head, int y, int\* lot, int size, int\*\* matrix)

{

Node\* p = head;

int count\_m = 0, flag = 0;

while (p)

{

if (lot[p->data] == -2)

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 1;

p = (Node\*)p->next;

}

else if (lot[p->data] == 1)//только один элемент в множестве, удаляем стену снизу

{

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 0;

p = (Node\*)p->next;

}

else if (lot[p->data] == -1)//сверху есть стена, есть выбор ставить стену или нет в текущем поле

{

//rand() % 2 == TRUE ? black\_rect(p->data, y, p->data + one\_cord, y + one\_cord) : white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

if (rand() % 2 == TRUE)

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 1;

}

else

{

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 0;

}

p = (Node\*)p->next;

}

else

{

if (flag == 0) flag = lot[p->data];

if (rand() % 2 == TRUE)//выбор поставить стену в множестве

{

count\_m++;//подсчет пройдённых клеток множества

if (count\_m == flag)//пункт, что в каждом множестве хотя бы одна клетка была без нижней стены

{

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 0;

p = (Node\*)p->next;

}

else

{

black\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 1;

p = (Node\*)p->next;

}

}

else

{

white\_rect(p->data, y, p->data + one\_cord, y + one\_cord);

matrix[y][p->data] = 0;

p = (Node\*)p->next;

}

}

}

}

void eller(int size\_lab, int\*\* matrix)

{

int count\_for\_procents = 0;

//show\_procents(count\_for\_procents);

count\_for\_procents++;

srand(time(NULL));

int\* lot = (int\*)calloc(size\_lab, sizeof(int));//для хранения количество множеств

if (lot != NULL)

{

for (int i = 0; i < size\_lab; i++)// цикл идёт по киличеству строк, т.е. размер лабиринта

{

Node\* head = NULL; // каждая строчка выделяется заново, поэтому изначально был создан массив на множества

Node\* tail = NULL;

push(&head, &tail, size\_lab);

if (i == 0 || i == size\_lab - 1)//верхняя и нижняя граница

{

black\_string(head, i, size\_lab, matrix);

}

else if (i % 2 == 0)//это нижняя граница строк

{

bottom\_bordes\_eller(head, i, lot, size\_lab, matrix);

}

else//выбор границ справа

{

right\_bordes\_eller(head, i, lot, size\_lab, matrix);

}

free\_spisok(head);

count\_for\_procents += i;

//show\_procents(count\_for\_procents);

}

}

free(lot);

}