1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт прикладной математики и механики
5. **Высшая школа кибербезопасности и защиты информации**

**КУРСОВАЯ РАБОТА**

**Игра «Жизнь»**

по дисциплине «Структуры данных»

1. Выполнили
2. студенты гр. 3651003/90001 Шкуренкова Д.В.
   * 1. Клочква Ж.Д.

1. Руководитель
2. ст. преподаватель Семьянов П.В.
4. Санкт-Петербург
5. 2020

Содержание

|  |  |  |  |
| --- | --- | --- | --- |
| Введение......................................................................................................... | |  | 6 |
| 1 | Теоретическая часть ......................................................................... |  | 10 |
| 1.1 | Название пункта (точки продолжать как выше сделано).............. |  | 10 |
| 2 | Практическая часть ................. |  | 16 |
| 2.1 | Название пункта.................................................... |  | 16 |
| Заключение.................................................................................................... | |  | 54 |
| Список использованных источников.......................................................... | |  | 56 |
| [Введение 3](#_Toc50197070)  [Цель работы 3](#_Toc50197071)  [Задачи 3](#_Toc50197072)  [1 Теоретические сведения 4](#_Toc50197073)  [1.1 Динамические массивы и их применение к реализации игры 4](#_Toc50197074)  [1.2 Правила игры «Жизнь» 7](#_Toc50197075)  [1.3 Особенности игры 8](#_Toc50197076)  [2 Реализация алоритма 10](#_Toc50197077)  [3 Реализация графического приложения 15](#_Toc50197078)  [4 Оценка эффективности работы 16](#_Toc50197079)  [5 Примеры работы программы 18](#_Toc50197080)  [заключение 22](#_Toc50197081)  [список использованных источников 23](#_Toc50197082) | |  |  |

Введение

Реализованные сегодня приложения способны обрабатывать колоссальные массивы данных в режиме реального времени, размер которых определяется уже в процессе, и не всегда возможно перед началом работы определить их размер. Для работы с такими данными используют динамические структуры. Основным преимуществом таких структур над статическими является то, что занимаемый ими объем памяти не является заранее фиксированным. Таким образом, размер занимаемой памяти может быть задан или же меняться в процессе работы программы и ограничивается разве что объемом оперативной памяти машины.

Кроме перечисленных достоинств динамических структур, существуют некоторые недостатки. К их числу можно привести более сложную реализацию. Поэтому, при работе со динамическими структурами у программиста гораздо выше вероятность совершить ошибку, так как необходимо следить за правильным использованием памяти, наличием висячих ссылок и пустых указателей.

Данная курсовая работа рассчитана на анализ структур, для поиска наиболее оптимального варианта реализации игры «Жизнь».

Цель работы

Реализовать игру «жизнь» используя оптимальную структуру данных.

Задачи

* Изучить алгоритм и правила игры «Жизнь»
* Исследовать существующие динамические структуры данных
* Выбрать оптимальную структуру, основываясь на алгоритме игры
* Разработать алгоритм работы будущего приложения
* Реализовать графическое приложение
* Оценить эффективность кода

# Теоретические сведения

## Динамические массивы и их применение к реализации игры

В качестве основной структуры данных в работе используется динамический массив. Массив — набор однотипных переменных, доступ к которым осуществляется по индексу. Динамический массив может изменять свой размер в зависимости от количества элементов в нём. Он поставляется со стандартными библиотеками на многих современных основных языках программирования. Динамические массивы преодолевают ограничение статических массивов, которые имеют фиксированную емкость, которую необходимо указать при распределении.

Простой динамический массив может быть создан путем выделения массива фиксированного размера, обычно большего, чем количество элементов, требуемых немедленно. Элементы динамического массива хранятся непрерывно в начале базового массива, а оставшиеся позиции ближе к концу базового массива зарезервированы или не используются. Элементы могут быть добавлены в конец динамического массива в постоянное время, используя зарезервированное пространство, до тех пор, пока это пространство не будет использовано полностью. Когда все пространство занято и нужно добавить дополнительный элемент, тогда необходимо увеличить размер базового массива фиксированного размера. Обычно изменение размера является дорогостоящим, поскольку оно включает выделение нового базового массива и копирование каждого элемента из исходного массива. Элементы могут быть удалены из конца динамического массива за постоянное время, так как изменение размера не требуется. Количество элементов, используемых содержимым динамического массива, является его логическим размером или размером , в то время как размер базового массива называется емкостью динамического массива или физическим размером , который является максимально возможным размером без перемещения данных.

Массив фиксированного размера будет достаточным в приложениях, где максимальный логический размер фиксирован (например, по спецификации) или может быть вычислен до выделения массива. Динамический массив может быть предпочтительнее, если: максимальный логический размер неизвестен или его трудно вычислить до выделения массива считается, что максимальный логический размер, указанный в спецификации, вероятно, изменится  
Амортизированная стоимость изменения размера динамического массива не оказывает существенного влияния на производительность или быстродействие

Чтобы избежать затрат на многократное изменение размера, динамические массивы изменяют размер на большую величину, например удваивают размер, и используют зарезервированное пространство для будущего расширения.

Динамический массив имеет производительность, аналогичную массиву, с добавлением новых операций для добавления и удаления элементов:  
Получение или установка значения по определенному индексу (постоянное время) Итерация элементов по порядку (линейное время, хорошая производительность кеша) Вставка или удаление элемента в середине массива (линейное время)Вставка или удаление элемента в конце массива (постоянное амортизированное время)

Динамические массивы используются для обработки наборов однородных данных, размер которых неизвестен точно на момент написания программы, но которые потенциально могут разместиться в доступной памяти. Без использования динамических массивов решение таких задач сводится к одной из стратегий: выделение статического массива большого размера, заведомо достаточного для полного размещения данных;  
использование статического массива в качестве буфера, в который данные загружаются частями; применение иных динамических структур данных (например, списков).

Первый вариант применим только когда размер набора данных меняется в небольшом, жёстко ограниченном диапазоне (например, в текстовой обработке ограничение в 1000 знаков на строку вполне приемлемо). В противном случае выбор маленького массива будет ограничивать функциональность программы, а большого (так, чтобы заведомо хватило для любых входных данных) приведёт к неэффективному расходованию памяти. Буферизация обработки усложняет алгоритм, а другие динамические структуры в задачах обработки больших последовательностей простых данных по эффективности не могут сравниться с массивом.

Использование динамических массивов позволяет выделить ровно столько памяти, сколько реально необходимо (сразу, если задача позволяет определить объём до загрузки данных, либо в процессе загрузки, расширяя массив по мере необходимости), загрузить все данные в один массив и единообразно их обработать. Недостатки, однако, имеются и у этой стратегии: снижение скорости работы из-за накладных расходов на изменение размера динамического массива;  
потенциальное снижение надёжности: при экстремально большом объёме входных данных попытка увеличить массив до соответствующих размеров может привести к внезапному существенному замедлению или даже отказу программы из-за недостатка свободной памяти.

В целом можно заметить, что поддержка динамических массивов повышает удобство разработки, но не освобождает программиста от необходимости проводить оценку потребностей программы в памяти; также она требует понимания особенностей конкретной реализации динамических массивов и согласования алгоритмов обработки данных с этими особенностями.

Мы выбрали именно динамический массив, поскольку само задание подразумевает наличие двумерной плоскости — поля игры, которое состоит из ячеек состояния клетки — живой или мертвой, что очень удобно реализуется с помощью массива нулей и единиц, в котором единица означает живую клетку, а ноль — мертвую. Динамический массив, в отличие от статического, позволяет игроку самостоятельно задать размер поля. Не в последнюю очередь выбор структуры был обусловлен желанием не усложнять код,  
а сделать упор на скорости его выполнения и графической составляющей — динамические массивы относительно просты в использовании и обеспечивают быструю навигацию по всему полю за счет наличия индексации. У нас стоял выбор между использованием в работе линейного списка или динамического массива, но было решено остановиться на последнем варианте, так как по сравнению со связанными списками динамические массивы имеют более быстрое индексирование (O(const) у массивов и О(n) у списков) и, как правило, более быструю итерацию из-за улучшенной локальности ссылок.

Стоит отметить, что линейные списки имеют некоторые преимущества перед динамическими массивами: во-первых, массивы требуют линейного времени для вставки или удаления в произвольном месте, поскольку все последующие элементы должны быть перемещены, а связанные списки могут делать это за постоянное время; во-вторых, в сильно фрагментированной области памяти может быть дорого или невозможно найти непрерывное пространство для большого динамического массива, в то время как связанные списки не требуют, чтобы вся структура данных хранилась непрерывно. Тем не менее, в нашей работе эти преимущества не имели особого смысла, поскольку поле после создания не меняло свой размер, постоянно изменялись только значения в ячейках.

## Правила игры «Жизнь»

Жизнь — это многоклеточное сообщество, населяющее пустыни Флатландии. Пустыня, оно же поле, представляет собой прямоугольную решетку, каждая ячейка которой вмещает одну клетку Жизни. Мерой течения времени служит смена поколений Жизни, приносящая в колонию клеток смерть и рождение.

История развития колонии является пошаговой эволюцией каждого предыдущего состояния жизни. Начало игры обуславливается начальной расстановкой клеток жизни на поле. Смена поколений будет происходить по следующим правилам.

1. Соседями клетки считаются все клетки, находящиеся в восьми ячейках, расположенных рядом с данной по горизонтали, вертикали или диагонали.
2. Если у некоторой клетки меньше двух соседей, она погибает от одиночества. Если клетка имеет больше трех соседей, она погибает от тесноты.
3. Если рядом с пустой ячейкой окажется ровно три соседние клетки Жизни, то в этой ячейке рождается новая клетка.
4. Гибель и рождение происходят в момент смены поколений. Таким образом, гибнущая клетка может способствовать рождению новой, но рождающаяся клетка не может воскресить гибнущую, и гибель одной клетки, уменьшив локальную плотность населения, не может предотвратить гибель другой.

Программа должна обрабатывать большие колонии без чрезмерной траты памяти или времени.

## Особенности игры

Колония может все время расти, непрерывно меняя свое расположение, форму или число клеток. Однако чаще колония становится в конце концов стационарной, начиная циклически повторять один и тот же конечный набор состояний. Длина цикла называется периодом колонии. По этому определению

Любая колония имеет преемника, но не у каждой есть предшественник. Такие изолированные колонии называются садами Эдема. Сад Эдема можно увидеть, только если поместить его на плоскость в качестве начальной конфигурации.

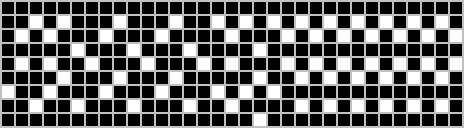


Рисунок 1 – кофигурация «Сад Эдема»

Существуют и другие различные известные конфигурации, самые популярные из которых представлены на рисунках 3-5, на рисунке 2 изображены примеры эволюции клеток. В последующей реализации графического приложения данные примеры можно использовать в качестве проверки.

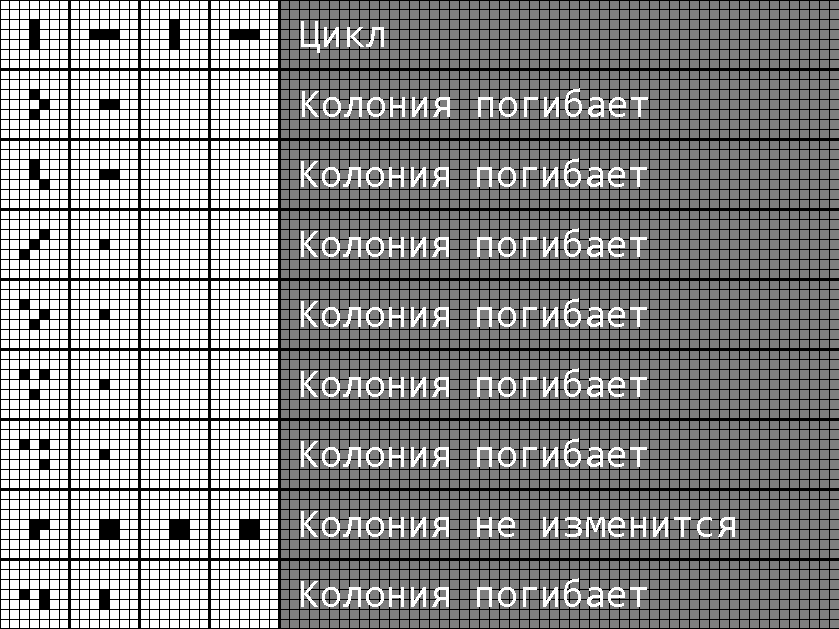


Рисунок 2 – примеры эволюции клеток жизни



Рисунок 3 –Натюрморты - группа статичных фигур



Рисунок 4 – Осцилляторы с периодом 2

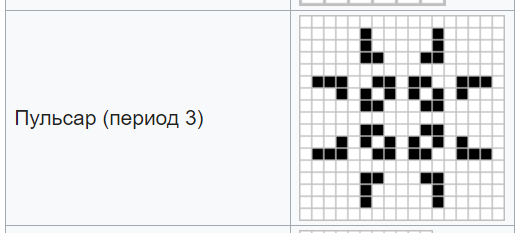


Рисунок 5 – Пульсар с периодом 3

# Реализация алоритма

Основной алгоритм, в котором реализуется логика игры, выглядит следующим образом: после запуска программы пользователь вводит длину и ширину игрового поля (количество ячеек), затем задает его начальное состояние — выбирает клетки, которые будут живыми или мертвыми. Само игровое поле представляет собой одномерный динамический массив структур клеток, в которых находятся два состояния клетки — текущее и будущее, как показано ниже:

Листинг 1 – Структура клетки

// Структура одной клетки с текущим и будущим состояниями: 0 - мертва, 1 - жива

struct {

unsigned short state\_1;

unsigned short state\_2;

} typedef cell;

Такая структура была сделана для того, чтобы при просмотре ячеек записывать их новые состояния в отдельные переменные, не теряя при этом старые значения, которые еще понадобятся для вычисления соседей клеток, находящихся по краям поля, в связи с его тороидальностью, что будет пояснено позже.

Будущее состояние ячейки вычисляется с помощью основной функции Conway путем подсчета количества ее живых соседей. За вывод полученного нового состояния поля отвечает процедура PrintDesk. Игра может продолжаться бесконечно, поэтому для ее завершения необходимо просто закрыть программу.

Основная функция, реализующая логику игры — функция Conway. Прежде всего, она проверяет, в какой из двух переменных состояний находятся все ячейки текущего поколения — state\_1 или state\_2. Это нужно для того, чтобы новые значения каждой ячейки записывать в их state\_2 или state\_1 соответственно. Далее функция просматривает каждую клетку поля слева направо сверху вниз и считает количество ее живых соседей.

В данной работе поле представляет собой тороидальную поверхность, изображенную на рисунке 6 — левая и правая, верхняя и нижняя части поля "склеены" друг с другом, угловые ячейки соседствуют с угловыми на противоположной стороне диагонали.

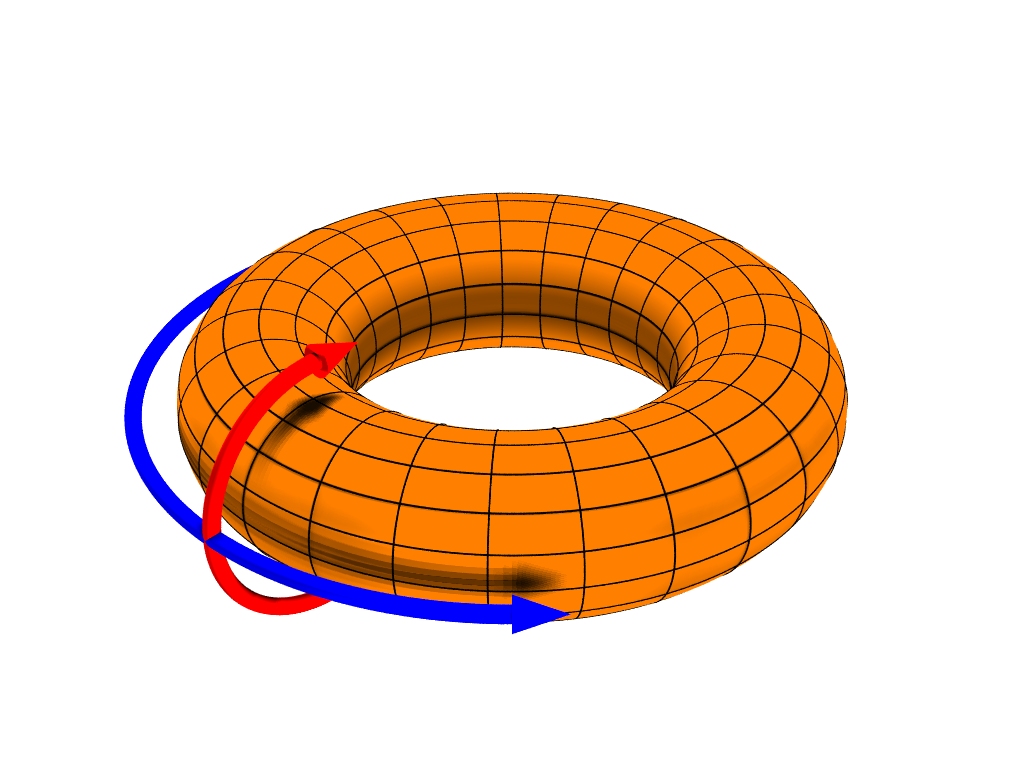


Рисунок 6 – Тор (поверхность)

Алгоритм просмотра соседей разбит на 8 блоков — по количеству смежных клеток у каждой ячейки. Проверка начинается с северо-западной клетки и заканчивается на юго-восточной (см. рисунок 7).



Рисунок 7 – Обозначение соседних ячеек рассматриваемой клетки

Программа вычисляет соседей исходя из индексов i и j положения обрабатываемой на данный момент ячейки, учитывая особенности тороидального поля. Одна из «проблемных» ситуаций изображена на рисунке 8: если рассматриваемая ячейка находится в столбце с индексом j = 0, т.е. ее координатами будет набор (i, 0) то ее соседкой с западной стороны будет ячейка на той же строке, но в последнем столбце (i, ширина\_поля - 1). Больше примеров представлено на рисунках 9 и 10, где красным цветом выделена рассматриваемая ячейка, а синим — все ее соседи, статусы которых будут учитываться при определении нового статуса красной ячейки.

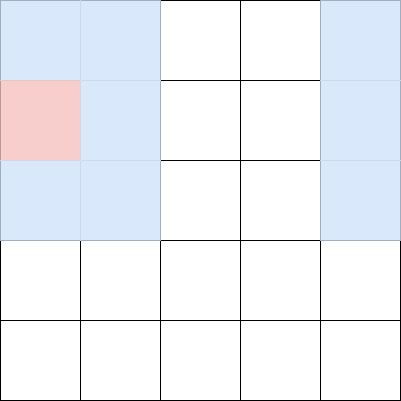


Рисунок 8 – Определение соседних клеток (синим) рассматриваемой ячейки (красным)

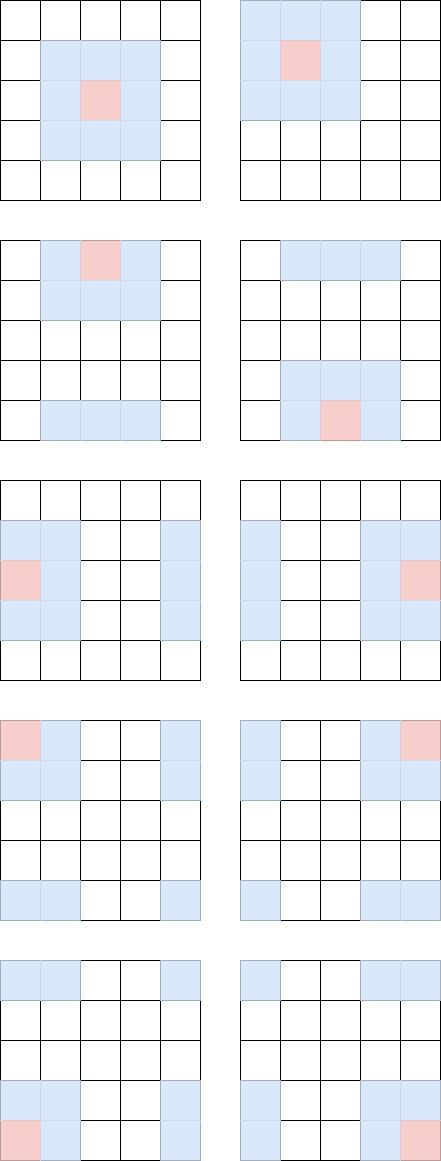


Рисунок 9 – Примеры выбора соседей (синим) для рассматриваемых ячеек (красным)

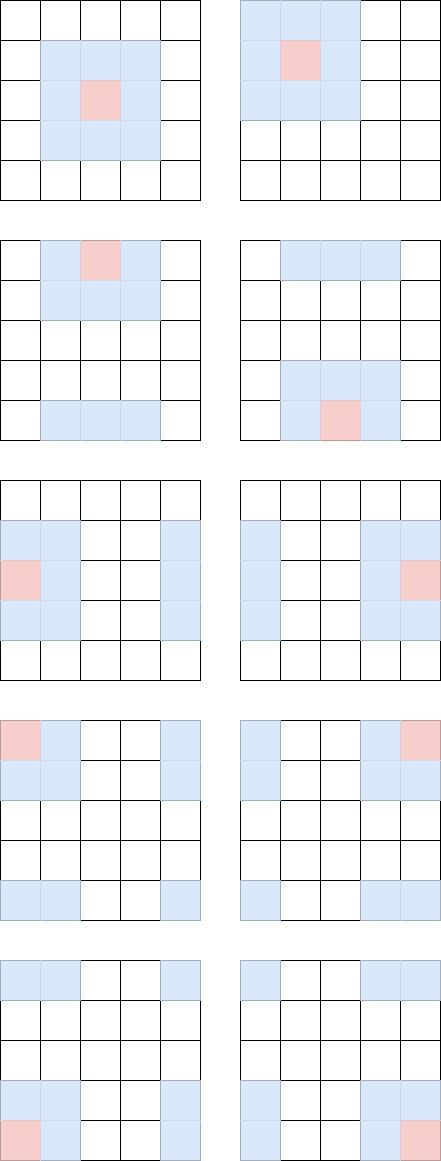


Рисунок 10 – Примеры выбора соседей (синим) для рассматриваемых ячеек (красным)

После того, как были обработаны все соседи клетки, начинается расчет ее нового состояния. Если она была жива, и вокруг нее находится две или три живых соседки, то она продолжает жить и в следующем поколении. Если нет, то умирает. Если клетка была мертва, но вокруг нее оказалось ровно три живых, то теперь она становится живой, рождается. В противном случае она так и остается неживой.

Функция Conway завершает свою работу после того, как новые состояния были определены для всех ячеек поля.

PrintDesk проходит по всему массиву ячеек поля слева направо сверху вниз, проверяя статус каждой ячейки — жива или мертва — и выводя его на экран. Процедура использует переданный ей параметр state, который указывает, в какой из переменных структуры клетки сейчас находится ее только что вычисленное новое состояние.

# Реализация графического приложения

Так как конечным продуктом курсовой работы должна являться игра, необходимо реализовать графическую составляющую. В качестве основной библиотеки была выбрана SDL. После подключения её в проект, были изучены основные функции, которые позволили сделать игру оконным приложением.

Для того, чтобы библиотека работала, в коде необходимо проинициализировать последующие действия с помощью функции init(), которая проверяет готовность к работе видео, возможность рендера. На каждом шаге в случае неудачи, происходит выход из функции, с возможностью получить подробную информацию о прерывании через SDL\_GetError(). Кроме того, init() запускает окно gWindow и рендерер gRenderer, которой в данном случае является поверхностью поля. После инициализации, запускается процесс расстановки начального положения клеток жизни в функции startDraw(), пользователь сам выбирает начальное расположение. В данной функции идет обработка событий e с помощью SDL\_PollEvent(). Если действие выход, соответственно игра завершается, если SDL\_GetMouseState зафиксирован клик правой кнопкой мыши сопутствующие координаты – состояние соответвующей клетки изменяется(если state\_1 было 1, то но становится 0, и наоборот). После этого отрисовывается текущее состояние доски в printDesk(), алгоритм которой будет указан ниже. Клик левой кнопки мыши означает, что начальное положение клеток жизни было задано и действие переходит в игровой цикл gameLoop(). Аналогично с процедурой startDraw(), в игровом цикле происходит обработка событий e. Работа функции завершается только в том случае, если пользователем было вызвано закрытие окна приложения. В остальных случаях действие переходит к основному процессу игры – функции Conway(), которая определяет последующее поколение. И далее, на каждом шаге на рендерер отображается текущее состояние доски процедурой printDesk(). Прежде всего, отображаются полосы решётки, показывающие границы клетки. Их количество зависит от введённых размеров поля пользователем до старта игры в консоли. Аналогично алгоритму Conway() происходит разделение в зависимости от текущего состояние клетки (state\_1 или state\_2). В этих ветвях идёт прохождение по массиву с последующей рисовкой прямоугольника, отвечающего за соответствующую клетке с состоянием 1, то есть живой клетки. За это отвечает SDL\_RenderFillRect(). Важно отметить, что при каждом вызове функции сначала очищается поверхность поля для избегания наложения состояний, а в конце функции необходимо обновить gRenderer, чтобы новое состояние отразилось.

# Оценка эффективности работы програмы

## Расчёт сложности работы программы

Сложность будет считаться для одного шага эволюции Жизни, то есть для одно цикла gameLoop(). Для облегчения задачи за n будет считаться сторона поля, то есть во внимание берется квадратная решетка. Основными действиями в цикле является функция Conway() и графическая процедура printDesk(). Рассмотрим основные операции этих элементов.

Для отображения линий используются два цикла, суммарная сложность которых 2n. Закрашивание живых клеток происходит в двойном цикле, представленном на листинге 2, сложность которого n2. В итоге значение сложности получилось O(n2).

Листинг 1 – Циклы отображения линий внутри printDesk()

for (int i = 1; i <= height + 1 ; i++) {

SDL\_RenderDrawLine(gRenderer, i \* (SCREEN\_WIDTH / width),

0, i \* (SCREEN\_WIDTH / width), SCREEN\_HEIGHT);

}

for (int i = 1; i <= width + 1; i++) {

SDL\_RenderDrawLine(gRenderer, 0, i\*(SCREEN\_HEIGHT / height),

SCREEN\_WIDTH, i \* (SCREEN\_HEIGHT / height));

}

Листинг 2 – Двойной цикл

for (int i = 0; i < height; i++) {

for (int j = 0; j < width; j++) {

if (desk[i \* width + j].state\_1 == 1) {

SDL\_Rect fillRect = { j \* (SCREEN\_WIDTH / width),

i \* (SCREEN\_HEIGHT / height), SCREEN\_WIDTH / width,

SCREEN\_HEIGHT / height };

SDL\_SetRenderDrawColor(gRenderer, 0x80, 0x80, 0x80, 0xFF);

SDL\_RenderFillRect(gRenderer, &fillRect);

}

}

}

Каждый шаг эволюции в Conway() предполагает проход по каждой клетке, следовательно – сложность равна n2. Количество внутренних сравнений можно не учитывать, так как оно является константой.

Листинг 3 – Начало двойного цикла в функции Conway()

for (i = 0; i < height; ++i) {

for (j = 0; j < width; ++j) {

В конечном счете сложность алгоритма составляет O(n2)

## Расчёт времени работы программы

Для того, чтобы отследить время работы одного шага жизни колонии, будет использована функция perf\_time(), внутри которой работает QueryPerformanceCounter() из библиотеки windows.h. Необходимо отследить, как зависит время работы приложения от начальных данных: размера поля и количества живых начальных клеток. Данная информация представлена ниже в виде таблицы и графика.

Таблица 1 – Зависимость времени работы от начальных данных

|  |  |  |
| --- | --- | --- |
| **Размер поля** | **Количество живых клеток** | **Время работы одного шага** |
| 5х5 | 5 | 6814 |
| 10х10 | 5 | 9218 |
| 15х15 | 5 | 11628 |
| 20х20 | 5 | 14192 |
| 25х25 | 5 | 16885 |
| 25х25 | 10 | 18780 |
| 25х25 | 15 | 16447 |
| 25х25 | 20 | 17381 |

Как видно по данным таблицы время работы шага увеличивается с увеличением размера поля, однако при неизменном размере поля, время остаётся примерно одинаковым. Это явление объясняется тем, что алгоритм предусматривает проход по каждой ячейке поля, соответственно время работы сильно зависит от размеров в большей мере чем от количества живых клеток.

Из наблюдений за колонией Жизни было выявлено то, что время работы так же зависит от сложности текущей конфигурации. Чем сложнее фигура, там больше времени уходит на её обработку. То есть если рассматривать один и тот же раунд игры, можно заметить, что на разном шаге время обработки будет разное. Данное явление продемонстрировано на рисунке 11.

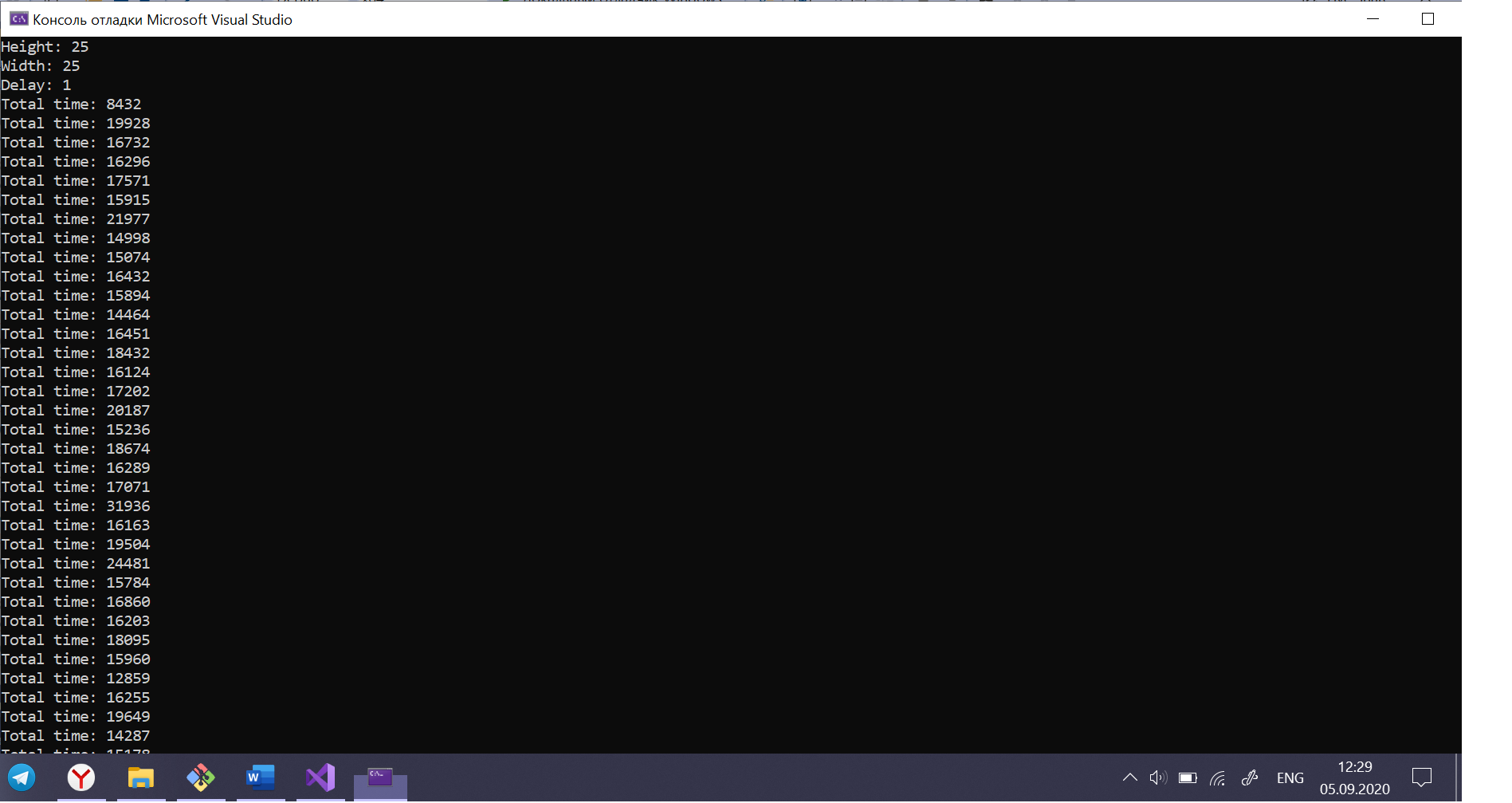


Рисунок 11 – Время работы программы на каждом шаге эволюции

# Примеры работы программы

Как было отмечено в пункте 1.3 показателем правильности алгоритма служит предполагаемое поведение определенных начальных фигур клеток Жизни. На рисунках 12-16 ниже представлены примеры работы приложения для некоторых конфигураций.

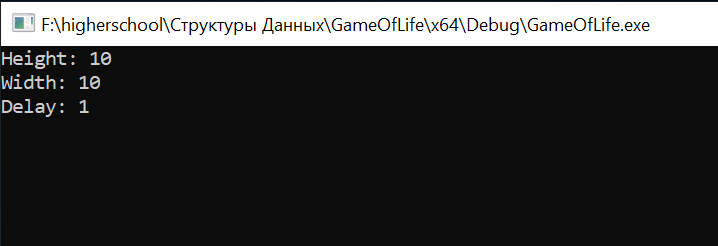


Рисунок 12 – Ввод начальных данных для игры

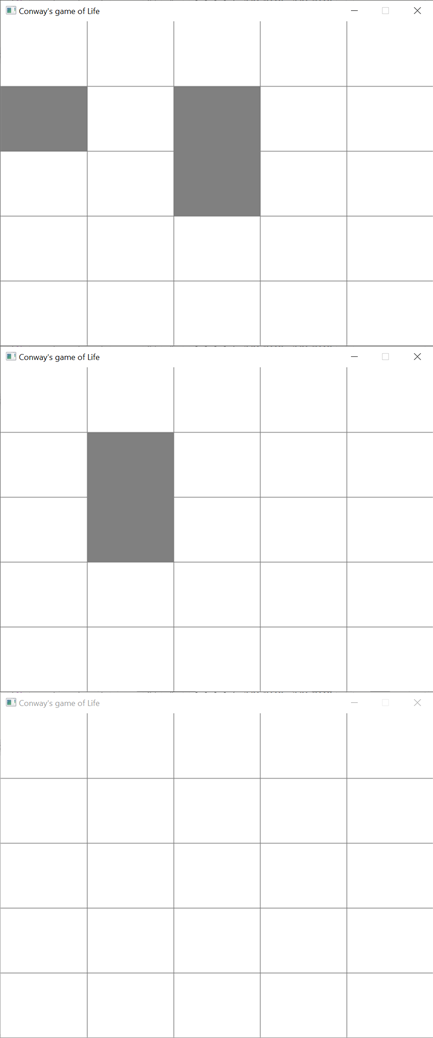


Рисунок 13 – Проигрышное начальное расположение

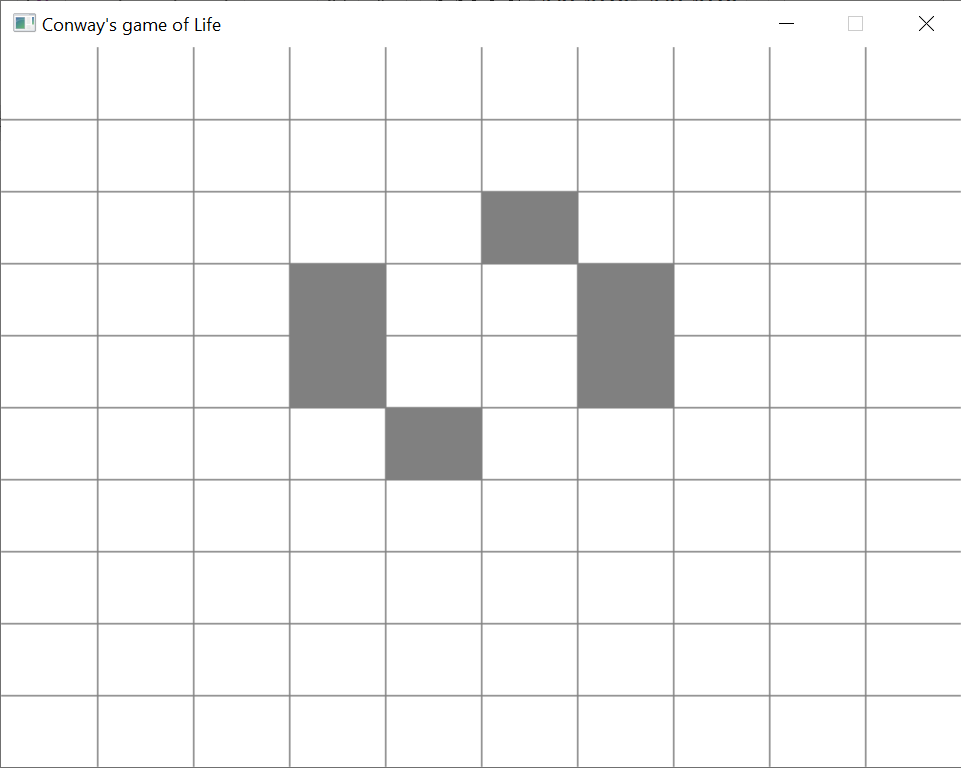
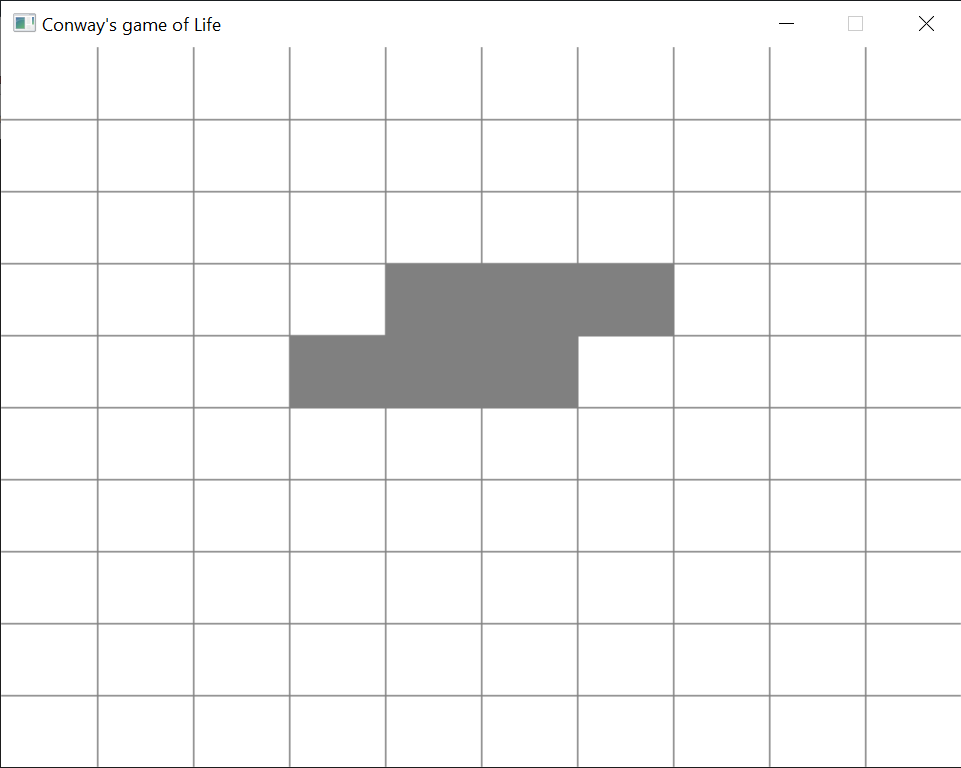


Рисунок 14 – Конфигурация Жаба с периодом 2

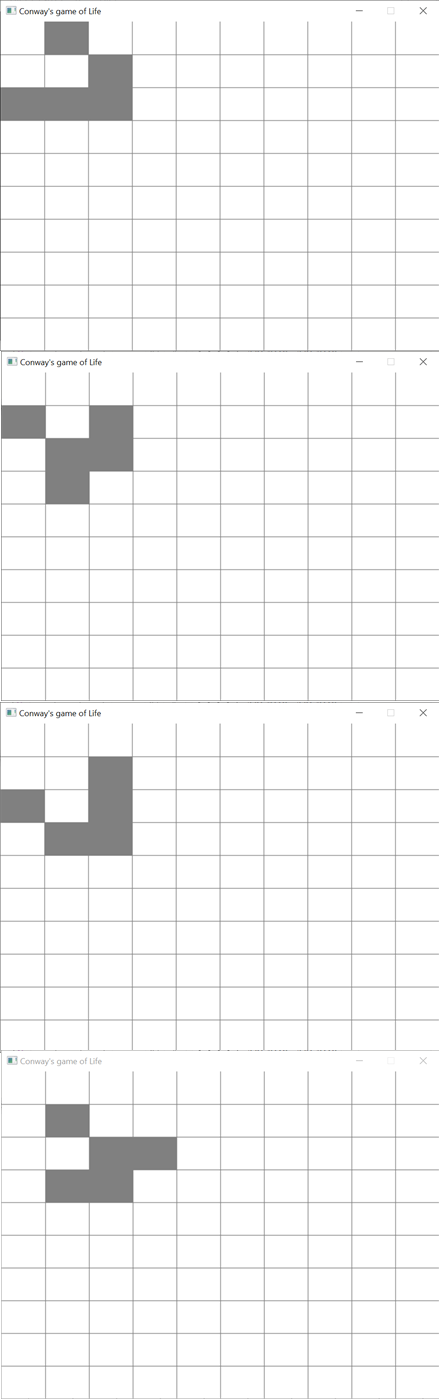


Рисунок 15 – Конфигурация планер с периодом 4

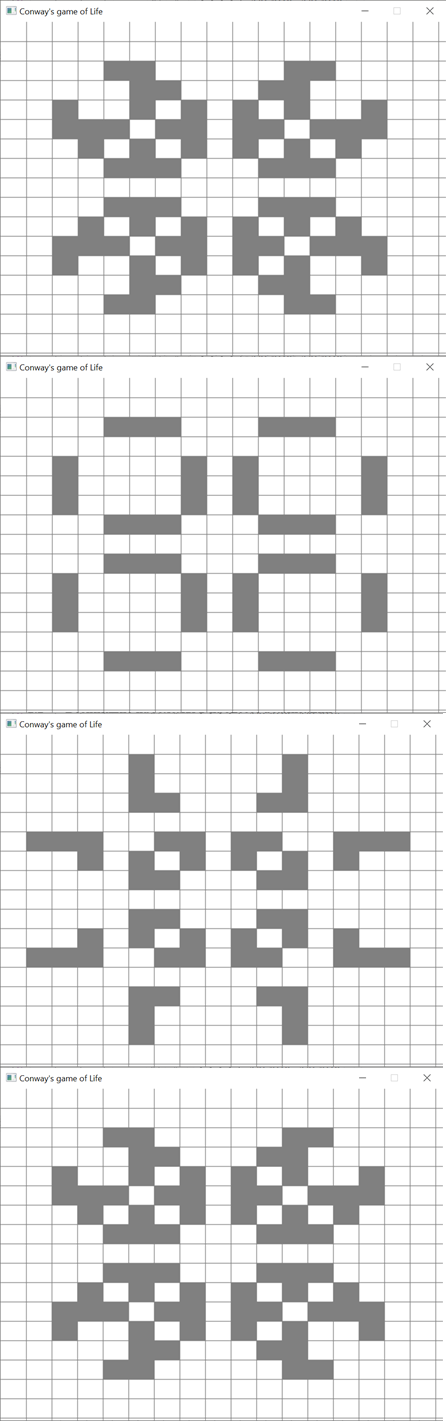


Рисунок 16 – Пульсар

Рисунок 17 иллюстрирует верность алгоритма в тех случаях, когда клетки выходят за границы поля, потому что задача получения бесконечной плоскости была решена способом, в котором решетка поля натягивается на тор.

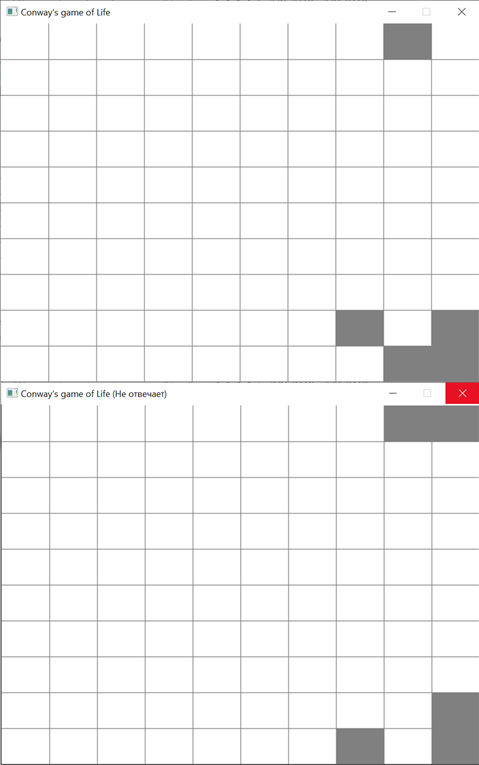


Рисунок 17 – Планер пересекает границы поля

заключение

В данной работе были исследованы динамические структуры данных на примере реализации игры «Жизнь». В качестве основной структуры приложения был выбран динамический массив, который оказался наиболее подходящим для данной работы благодаря простоте использования и наличием быстрого индексирования. Был разработан алгоритм программы, а также создана ее графическая оболочка. Полученное приложение при тестировании показало, что оно корректно отвечает правилам игры и затрачивает на свою работу приемлемое количество времени и памяти.

список использованных источников

1. Уэзерелл Ч. Этюды для программистов [Электронный ресурс]. URL: https://coollib.com/b/214357/read. – (дата обращения: 10.04.2020).
2. LifeWiki [Электронный ресурс]. URL: https://www.conwaylife.com/wiki/Conway's\_Game\_of\_Life. – (дата обращения: 10.04.2020).
3. Гарднер М. Математические досуги: Пер. с англ. – М.: Рипол Классик, 1972. – 496 с.
4. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. – М.: Невский Диалект, 2001. – 352 с.
5. Кнут Д. Э. Искусство программирования, том 1. Основные алгоритмы: В 4 т.: Пер. с англ. – М.: Издательский дом «Вильямс», 2018. – Т.1. – 722 с.
6. SDL Wiki [Электронный ресурс]. URL: https://wiki.libsdl.org/. – (дата обращения: 01.05.2020)