**ДЕПАРТАМЕНТ ОБРАЗОВАНИЯ И НАУКИ ГОРОДА МОСКВЫ**

****

**ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ПРОФЕССИОНАЛЬНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ГОРОДА МОСКВЫ**

**«ПОЛИТЕХНИЧЕСКИЙ КОЛЛЕДЖ ИМЕНИ П.А. ОВЧИННИКОВА»**

**Специальность:**

09.02.07 Информационные системы и программирование

**КУРСОВАЯ РАБОТА**

**МДК.01.01. Разработка программных модулей**

тема: «Автоматическое построение лабиринтов»

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент: | Иванов Иван Петрович | | | | |  | | |
|  | |  | **Ф.И.О** | | | **подпись** | | |
| Группа №: | | ИС-31 |  | |  |  | |  |
|  | |  | |  | |  | |  |
| Руководитель: | | Олейник Владимир Иванович | | | |  | |  |
|  | |  | **Ф.И.О** | | | **подпись** | | |
|  | |  | | | |  |  |  |
| Оценка: |  | | Дата: | | «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_2023 г. | | | |
|  | |  | | | |  | |  |

**Москва**

**2023**

# СОДЕРЖАНИЕ

|  |  |  |
| --- | --- | --- |
|  |  | Стр. |
|  | Введение……………………………………………………………….….. | 3 |
| 1. | Алгоритм создания лабиринта…………………………………………... | 4 |
| 1.1. | Определение исходных точек…………………………………………… | 4 |
| 1.2. | Генерация проходов……………………………………………………… | 4 |
| 1.3. | Определение конечной точки лабиринта……………………………….. | 7 |
| 2. | Разработка приложения………………………………………………….. | 9 |
| 2.1. | Создание функций для генерации лабиринта…………………………... | 9 |
| 2.2. | Создание графического интерфейса…………………………………….. | 15 |
| 2.3. | Создание игрового процесса…………………………………………….. | 18 |
|  | Заключение………………………………………………………………… | 21 |
|  | Список использованной литературы и интернет – источников……….. | 22 |
|  | Приложение А…………………………………………………………….. | 23 |

**Введение**

Компьютеры достаточно давно и прочно вошли в нашу жизнь. В настоящее время компьютер вполне может рассматриваться как универсальное средство для работы, творчества, развлечения человека. Электронные вычислительные машины стали заменой для многих вещей, и внедрение их в различные сферы жизни общества помогают им встать на место тех вещей, которые они могут заменить. Компьютер позволяет хранить большие объемы информации занимая минимальное место Современные технологии позволяют максимально быстро обрабатывать информацию с использованием средств её защиты. Так же появление компьютера поспособствовало развитию науки. Однако, без человеческого контроля компьютерные технологии не смогут выполнять поставленные задачи и правильно функционировать. Задачей данной курсовой работы стала разработка программы для игры «Питон». Разработанная программа, предназначенная для изображения движущегося “питона”, состоящего из трех отрезков одинаковой длины, может использоваться людьми, не имеющими квалификации программиста. Программа разработана с использованием современных концепций программирования – объектно-ориентированного программирования и языка Object Pascal. Пояснительная записка содержит техническое задание, описание программы, программу и методику испытаний, описание применения, текст программы.

**1. Алгоритм создания лабиринта**

**1.1. Определение исходных точек**

Установим следующее правило для определения начальной точки генерации лабиринта. Пусть ширина лабиринта равна m, высота n, координаты оси OX и OY. При этом нумерация клетки-лабиринта начинается с нуля, а клетка с координатами (0, 0) находится в нижнем левом углу. Начальная точка нашего лабиринта должна находиться с края, т.е. е. одна из координат должна равняться либо получится, либо координата x=n-1, либо y=m-1 соответственно (см. Рис. 1.)

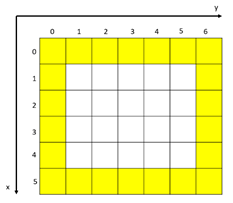


Рис. 1. Координаты матрицы лабиринта и место формирования исходных точек.

**1.2. Генерация проходов**

После того, как мы определили точку начала нашего лабиринта, можно приступить к генерации самого лабиринта. При формировании мы должны соблюдать некоторые правила:

1. Лабиринт не должен иметь циклов, т.е. е. в каждой камере лабиринта существует только один путь.
2. В лабиринте должны отсутствовать замкнутые области, к каждой камере мы должны иметь доступ.

Из исходных точек программа должна найти все доступные пути перехода. Затем мы «переходим» в клетки, которые выберем из имеющихся. При переходе мы отметим, в каких клетках мы уже побывали. И тогда эти клетки уже не будут доступны, как варианты смены. Так мы сохраняем циклы. Выбор будет происходить случайным образом, это как раз и сделает запутанным и непредсказуемым лабиринт. В ходе выполнения алгоритма может начаться момент, когда у нас не будет доступных вариантов для перехода (см. рис. 2). Тогда мы вернёмся обратно туда, откуда пришли эти клетки, и проверим, есть ли в этой клетке варианты для изменений. Если да, то переходим по этому варианту (если вариантов несколько, выбираем случайным образом один из вариантов). Так проводятся разветвления лабиринта. Если же мы и в этой клетке, не нашли вариантов для перехода, то тогда мы возвращаемся в клетки, из которых мы пришли в текущую клетку, и так до тех пор, пока мы не найдём вариант для перехода.

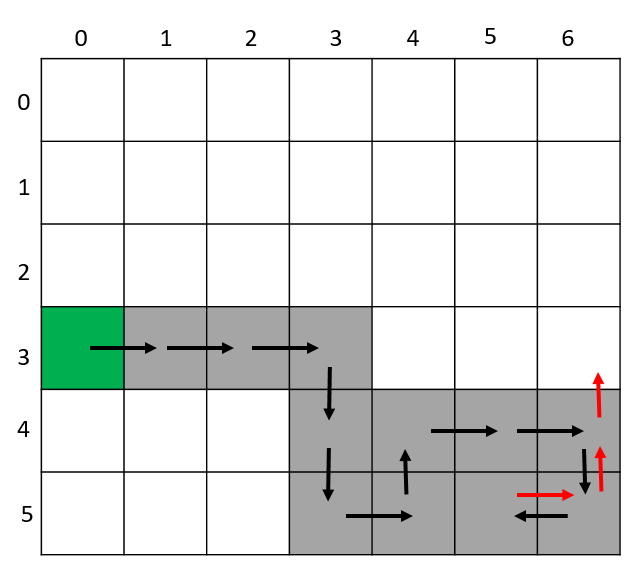


Рис. 2. Механизм образования разветвлений лабиринта.

И только когда все летки заняты, мы останавливаем работу алгоритма и считаем, что лабиринт полностью сгенерирован. Так мы повторяем замкнутые области (см. рис. 3.).

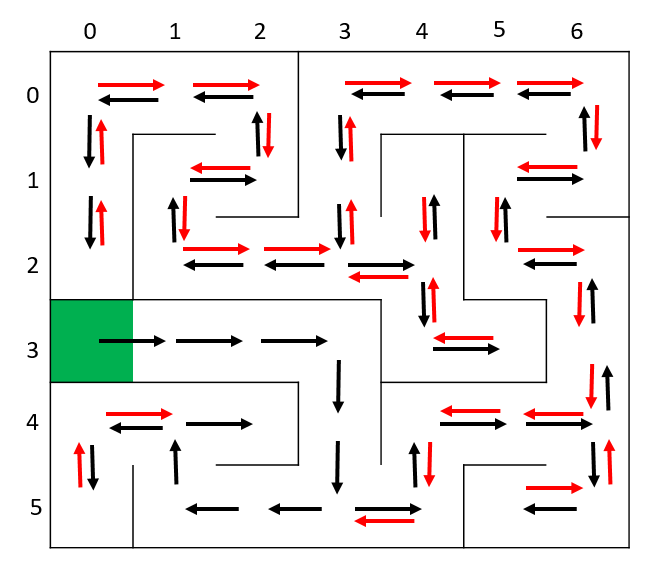


Рис. 3. Обход всех клеточных матриц.

Для хранения информации о переходах и о клетках, которые мы уже посещали, создадим две таблицы – таблицу переходов и таблицу достижимости.

**1.3. Определение конечной точки лабиринта**

Конечная точка лабиринта в программе Определяется простым методом по формуле (n-1-x\_(нач.); m-1- y\_(нач.)), n и m – ширина и высота лабиринта, a x\_(нач.) и y\_( нач.) – координаты точки начала лабиринта. Благодаря такому созданию конечной точки лабиринта она будет располагаться на противоположной стороне от начальных точек, что увеличит путь до точки лабиринта, а значит сделает его интереснее.

Однако в таком расположении исходных точек и конечной точки также есть свой определенный недостаток. В случае, когда высота или ширина лабиринта будет являться нечётными числами, начальная и конечная точки могут быть сгенерированы на одной линии в центре (см. рис. 4). В таком случае мы заново генерируем лабиринт.

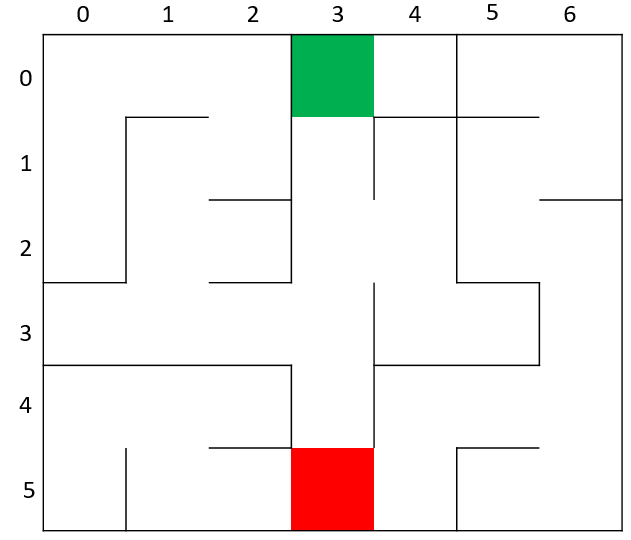


Рис. 4. Вариант генерации простого лабиринта.

**2. Разработка приложения**

**2.1. Создание функций для генерации лабиринта**

Созданы вспомогательные функции формирования точек начала и конца лабиринта *start\_point\_generate* и *Finish\_point\_generate* , которые будут переходить в фазу лабиринта и возвращать координаты начальных и конечных точек согласно алгоритму, который был описан выше (см. листинг 1).

Листинг 1. Функции формирования исходных точек и конечной точки лабиринта.

def start\_point\_generate(n, m):

"""Функция выбора точки начала лабиринта"""

if random.choice([True, False]):

if random.choice([True, False]):

start = (0, random.randint(0, m - 1))

else:

start = (n - 1, random.randint(0, m - 1))

else:

if random.choice([True, False]):

start = (random.randint(0, n - 1), 0)

else:

start = (random.randint(0, n - 1), m - 1)

return start

def finish\_point\_generate(start, n, m):

"""Выбор точки конца лабиринта"""

return n - 1 - start[0], m - 1 - start[1]

С помощью функций модуляции *случайным* образом определяется с какой стороны будет точка начала, а затем также случайным образом определяется вторая координата.

Далее нам нужно описать процесс самой генерации лабиринта. Нам понадобится вспомогательная функция transition\_choice *(* см. листинг 2.), которая будет определять клетки, которые мы перейдём далее. Создаётся список, в котором мы обеспечиваем доступность текущего соседа клетки, а затем по порядку, который оказался ли мы в данный момент из остановившейся клетки. Если нет, то координаты этой клетки мы вносим в список. Когда все соседние клетки подтвердятся, мы проверяем список доступных для изменения соседей наличия в нем хотя бы одного элемента. Если список непустой, случайным образом формируется один из пунктов списка. Это и будут координаты для точек перехода. Перечитаем эти координаты для матриц перехода (подробнее о матрицах переходов я расскажу далее). Если список окажется пустым, функция вернёт несуществующие координаты, чтобы далее мы смогли обработать красный цвет в ячейке, из которой мы пришли.

Листинг 2. Функция выбора дальнейшего пути в генераторном лабиринте.

def transition\_choice(x, y, rm):

"""Функция выбора дальнейшего пути в генерации лабиринта"""

choice\_list = []

if x > 0:

if not rm[x - 1][y]:

choice\_list.append((x - 1, y))

if x < len(rm) - 1:

if not rm[x + 1][y]:

choice\_list.append((x + 1, y))

if y > 0:

if not rm[x][y - 1]:

choice\_list.append((x, y - 1))

if y < len(rm[0]) - 1:

if not rm[x][y + 1]:

choice\_list.append((x, y + 1))

if choice\_list:

nx, ny = random.choice(choice\_list)

if x == nx:

if ny > y:

tx, ty = x \* 2, ny \* 2 - 1

else:

tx, ty = x \* 2, ny \* 2 + 1

else:

if nx > x:

tx, ty = nx \* 2 - 1, y \* 2

else:

tx, ty = nx \* 2 + 1, y \* 2

return nx, ny, tx, ty

else:

return -1, -1, -1, -1

В основной функции (см. листинг 3.), которая и будет ограничивать лабиринт, мы будем использовать матрицу достижимости, чтобы сохранить информацию о том, в каких точках мы уже были, а в каких нет; матрицу переходов, хранящую информацию о расположении стенок в лабиринте; про маршрутизированный путь в виде списка, чтобы мы могли в случае необходимости вернуться в клетку, из которой пришли. На рис. 5. показано, как будут сопровождаться матрицы переходов и достижения в процессе работы программы (0 и 1 – значения True и False).

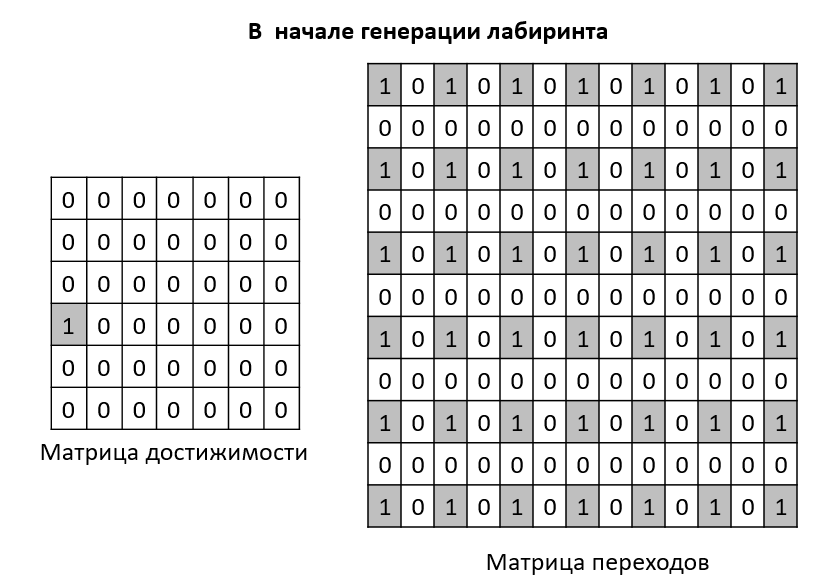
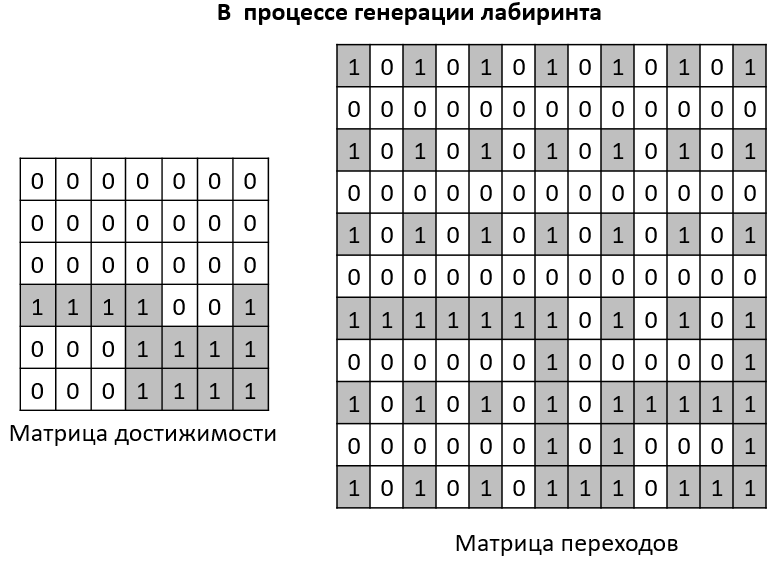
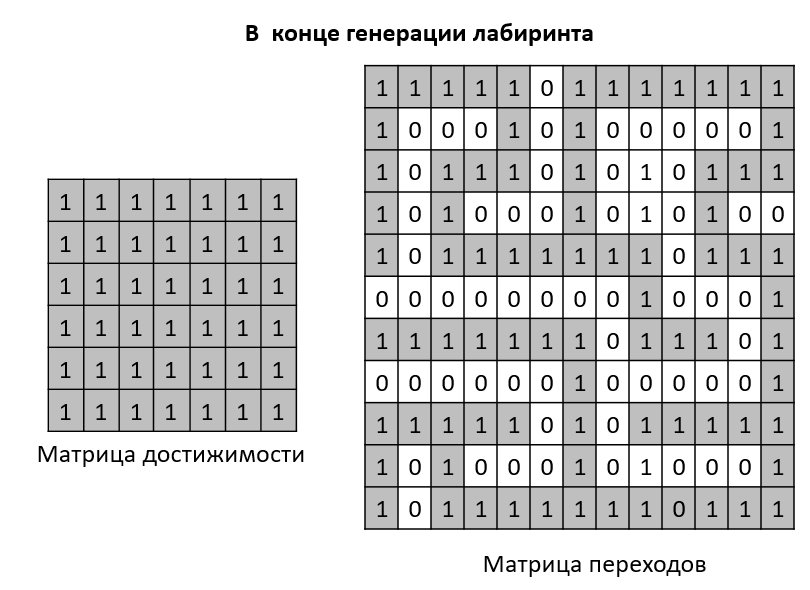
  

Рис. 5. Изменения матриц достижимости и переходов в выполнение генерации лабиринта

Для начала мы сгенерируем начальную точку, которая и будет первым элементом в маршрутном списке. Далее с помощью функции *перехода\_выбора* определяется ячейка, которая произойдёт переход. Если такой ячейки нет, удаляем текущее положение из маршрутного списка и переходим назад к тех пор, пока не найдём, куда перейти (здесь хорошо подошёл цикл *while* ). Функция генерации получает координаты точек начала и конца лабиринта и матрицу переходов, c помощью которых мы далее сможем изобразить лабиринт не полотне.

Листинг 3. Создание генерации лабиринта.

def create\_labyrinth(n=5, m=5):

"""Генерация лабиринта"""

reach\_matrix = []

for i in range(n): # создаём матрицу достижимости ячеек

reach\_matrix.append([])

for j in range(m):

reach\_matrix[i].append(False)

transition\_matrix = []

for i in range(n \* 2 - 1): # заполнение матрицы переходов

transition\_matrix.append([])

for j in range(m \* 2 - 1):

if i % 2 == 0 and j % 2 == 0:

transition\_matrix[i].append(True)

else:

transition\_matrix[i].append(False)

start = start\_point\_generate(n, m)

finish = finish\_point\_generate(start, n, m)

list\_transition = [start]

x, y = start

reach\_matrix[x][y] = True

x, y, tx, ty = transition\_choice(x, y, reach\_matrix)

for i in range(1, m \* n):

while not (x >= 0 and y >= 0):

x, y = list\_transition[-1]

list\_transition.pop()

x, y, tx, ty = transition\_choice(x, y, reach\_matrix)

reach\_matrix[x][y] = True

list\_transition.append((x, y))

transition\_matrix[tx][ty] = True

x, y, tx, ty = transition\_choice(x, y, reach\_matrix)

return transition\_matrix, start, finish # возвращаем матрицу проходов, начальную и конечную точку

**2.2. Создание графического интерфейса**

Для создания графического интерфейса я использовал Pygame. Pygame — это библиотека модулей для языка Python, созданная для разработки 2D-игр. Pygame также может называться «фреймворком». В программировании понятий «библиотека» и «фреймворк» несколько различаются. Но когда речь идет о классификации конкретного инструмента, не все так однозначно.

В любом случае, фреймворк является более мощным по-сравнением с библиотекой, он накладывает свою специфику на особенности программирования и сферу использования продукта. С точки зрения специфики Pygame – это каркас. Однако его сложно назвать «мощным образом». По своему объему и функционалу это скорее библиотека.

Также существует понятие «игрового движка» как программной среды для разработки игр. По своему назначению Pygame можно считать игровым движком. В то же время, с точки зрения выбора программного обеспечения, Pygame — это API для Питона с API-библиотекой SDL.

API – это интерфейс (в основном набор функций и классов) для прикладного (часто более высокоуровневого) программирования, который предоставляет, например, та или иную библиотеку. SDL – это библиотека, которая работает с отключенными устройствами компьютера.

Что именно использовать Pygame? Оно в легком понимании в отрасли и прототипировании. Pygame – небольшая библиотека. Сам Python позволяет писать короткий и длинный код. Так что это хорошее начало, чтобы познакомиться с особенностями разработки игр. Более опытными программистами Pygame может использоваться для быстрого создания прототипов игр, чтобы посмотреть, как все будет работать. После этого программа перезаписывается на другой язык. Другими словами, за исключением Pygame в легком обучении и быстрой разработке.

Для того, чтобы изобразить на полотне лабиринта, я создал функцию *draw\_labyrinth* (см. листинг 4). Функция использует следующие значения:

* Матрицу переходов
* Координаты начала
* Координаты конца
* Толщина проходов
* Толщина стен
* Цвет проходов (картеж из 3-х элементов)
* Цвет стен (картеж из 3-х элементов)
* Толщина границы вокруг лабиринта
* Цвет точки начала
* Цвет точки конца

Листинг 4. Функция рисования лабиринта

def draw\_labyrinth(matrix, start, finish, width\_line=20, width\_walls=5, color\_way=(255, 255, 255),

color\_wall=(0, 0, 0),

border=5, color\_start=(0, 255, 0), color\_finish=(255, 0, 0)):

"""Рисование лабиринта"""

width = (len(matrix) // 2 + 1) \* width\_line + (len(matrix) // 2) \* width\_walls + border \* 2

height = (len(matrix[0]) // 2 + 1) \* width\_line + (len(matrix[0]) // 2) \* width\_walls + border \* 2

for i in range(width):

for j in range(height):

if i < border or width - i <= border or j < border or height - j <= border: # отображение границ лабиринта

pygame.draw.line(window, color\_wall, [i, j], [i, j], 1)

else:

if (i - border) % (width\_line + width\_walls) <= width\_line:

x = (i - border) // (width\_line + width\_walls) \* 2

else:

x = (i - border) // (width\_line + width\_walls) \* 2 + 1

if (j - border) % (width\_line + width\_walls) <= width\_line:

y = (j - border) // (width\_line + width\_walls) \* 2

else:

y = (j - border) // (width\_line + width\_walls) \* 2 + 1

if matrix[x][y]:

pygame.draw.line(window, color\_way, [i, j], [i, j], 1)

else:

pygame.draw.line(window, color\_wall, [i, j], [i, j], 1)

pygame.draw.rect(window, color\_start, (

border + start[0] \* (width\_line + width\_walls), border + start[1] \* (width\_line + width\_walls), width\_line,

width\_line))

pygame.draw.rect(window, color\_finish, (

border + finish[0] \* (width\_line + width\_walls), border + finish[1] \* (width\_line + width\_walls), width\_line,

width\_line))

Сначала мы определяем размер изображения, которое должно получиться в результате рисования, а затем проходим по каждому пикселю и закрашиваем его в нужный нам цвет в зависимости от координат точки. Мы проверяем, является ли эта точка проходом стеной или рамкой лабиринта. А в самом конце закрашиваем точки начала и конца в выборе цвета. В результате работы этой функции мы получаем изображение случайно сгенерированного лабиринта (рис. 6.), на котором в перспективе можно создать игрока с перемещением его по полю.

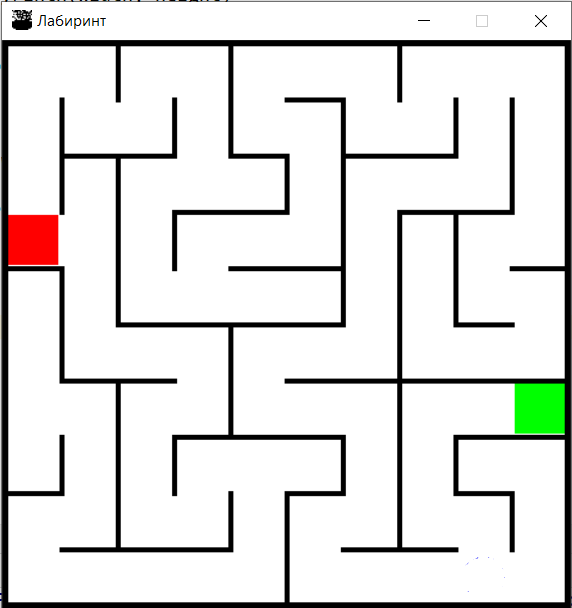


Рис. 6. – Графический интерфейс программы

**2.3. Создание игрового процесса**

Создан игроком, который будет перемещаться по лабиринту, когда пользователь будет нажимать определенные клавиши на клавиатуре. Каждый основной цикл программного обеспечения мы будем проверять, будут нажаты клавиши и в зависимости от этого будут выполняться или не выполняться определенные действия.

Для того, чтобы изобразить игрока на поле лабиринта, используйте функцию *draw\_player* (см. листинг 5).

После перемещения игрока необходимо убрать его старое положение. Для этого я использую функцию *delete\_player* (см. листинг 5.). Затем снова нарисуем игрока, но уже на другом месте. В программе я предусмотрел оставление одинаковых меток в лабиринте, чтобы игрок, где он уже проходил. Эту возможность можно подключить к определенному лицу на клавиатуре.

Также программа может подсчитывать количество пройденных лабиринтов, время, затраченное на прохождение лабиринта, а также может сохранять и записывать время прохождения лабиринта. Вся эта информация выводится на информационной таблице внизу экрана. Чтобы работать со временем, мне понадобился модуль *времени* .

Листинг 5. Функции изображения игрока на полотне и его удаление.

def draw\_player():

"""Отрисовка игрока на экране"""

pygame.draw.circle(window, color\_player, (border + player[0] \* (width\_line + width\_walls) + width\_line // 2,

border + player[1] \* (width\_line + width\_walls) + width\_line // 2), width\_line // 2 - 3)

def delete\_player():

"""Функция удаления игрока при движении и оставления следов"""

if (player[0], player[1]) == start:

pygame.draw.circle(window, color\_start, (border + player[0] \* (width\_line + width\_walls) + width\_line // 2,

border + player[1] \* (width\_line + width\_walls) + width\_line // 2), width\_line // 2 - 3)

else:

pygame.draw.circle(window, color\_way, (border + player[0] \* (width\_line + width\_walls) + width\_line // 2,

border + player[1] \* (width\_line + width\_walls) + width\_line // 2), width\_line // 2 - 3)

if trace:

pygame.draw.circle(window, color\_trace, (border + player[0] \* (width\_line + width\_walls) + width\_line // 2, border + player[1] \* (width\_line + width\_walls) + width\_line // 2), width\_line // 3 - 3)

Ещё предусмотрена возможность минимизации повторений лабиринта. Все лабиринты включаются в список, и при каждой новой генерации происходит проверка наличия вновь сгенерированного лабиринта на наличие в списке лабиринтов, которые уже были сгенерированы ранее. Если сгенерированный лабиринт найден в списке, то генерация происходит повторно, если подряд прошло 20 повторных генераций, программа начинает выдавать лабиринты с маршрутным проходом, т.е. е. лабиринты в точке, начало которой находится на одной линии с точкой конца лабиринта, а затем начинают повторяться.

Можно пройти новый лабиринт, не дойдя до конца текущего лабиринта, а также вступить в начальную точку по нажатию специальных клавиш на клавиатуре.

Все вышеперечисленные действия позволяют осуществлять (см. код в файле Labyrinth\_main.py), которые вызываются при нажатии клавиш.

**Заключение**

В результате проведенной работы мной создана система аватоматического построения лабиринтов. Для создания системы я использовал язык программирования «Python», был разработан алгоритм решения задачи, составлена и отлажена программа, оформлена документация на программу. Проведенные испытания показали, что программа работает в полном объеме, все требования технического задания выполнены.

[**Список источников**](https://github.com/Yan-Minotskiy/labyrinth_generating#%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA-%D0%B8%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%BE%D0%B2)

1. Python 3 для начинающих и чайников – уроки программирования [Электронный ресурс] URL: [https://pythonworld.ru](https://pythonworld.ru/) (дата обращения: 14.12.2019)
2. Pygame и разработка игр. [Электронный ресурс] URL: <https://younglinux.info/pygame/pygame> (дата обращения: 24.12.19)
3. Язык программирования Python: что такое и где используется – Логотип и история [Электронный ресурс] URL: <https://all-python.ru/osnovy/yazyk-programmirovaniya.html> (дата обращения: 24.12.2019)

**Приложение А**

**Листинг программы**

unit Unit1;

interface

uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, ImgList, ExtCtrls, StdCtrls, Menus, Grids;

type list=record name:string[10];

ch:integer; end;

spisok=list; fspisok=file of list;

TForm1 = class(TForm)

Image1: TImage;

Button1: TButton;

Button2: TButton;

Timer1: TTimer;

MainMenu1: TMainMenu;

N3: TMenuItem;

StringGrid1: TStringGrid;

N1: TMenuItem;

N2: TMenuItem;

N4: TMenuItem;

procedure FormCreate(Sender: TObject);

procedure Button1Click(Sender: TObject);

procedure Button2Click(Sender: TObject);

procedure Timer1Timer(Sender: TObject);

procedure N3Click(Sender: TObject);

procedure N4Click(Sender: TObject);

procedure N2Click(Sender: TObject);

private { Private declarations } public { Public declarations } end;

const h=40;

var Form1: TForm1;

x,y,xn,yn,n,q1,w1,e1,i:integer;

way,way2:integer; fcount:fspisok;

40 count:spisok;

implementation {$R \*.dfm} procedure TForm1.FormCreate(Sender: TObject);

begin i:=0;

assignfile(fcount,'count.dat');

if fileExists('count.dat') then begin reset(fcount);

read(fcount,count); closefile(fcount);

rewrite(fcount);

count.ch:=count.ch+1;

write(fcount,count);

closefile(fcount);

end;

x:=380;

y:=260;

xn:=380;

yn:=260;

n:=1;

Image1.Canvas.Moveto(300,300);

Image1.Canvas.LineTo(340,300);

Image1.Canvas.LineTo(340,260);

Image1.Canvas.LineTo(380,260);

end;

procedure TForm1.Button1Click(Sender: TObject);

begin Timer1.Enabled:=true;

end;

procedure TForm1.Button2Click(Sender: TObject);

begin Timer1.Enabled:=false;

end;

procedure zakrasform(n1: integer; var Im1: TImage);

end;

if n1=2 then begin Im1.Canvas.MoveTo(340,300);