Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)

(рівень вищої освіти)

на тему: **Розроблення гри "Сокобан" засобами Python**

Виконав: студент 4 курсу, групи КН-41
спеціальності
122 – "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Ільків А.В.
(прізвище та ініціали)

Керівник Пірко І.Б.
(прізвище та ініціали)

Рецензент Овсяк О.В.
(прізвище та ініціали)

Львів – 2025 р.

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій

Кафедра комп'ютерних наук

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КН

Борецька І.Б.
"10" червня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Ільків Андрій Віталійович
(прізвище, ім'я, по батькові)

1. Тема роботи **Розроблення гри "Сокобан" засобами Python**

керівник роботи Пірко І. Б, канд. фіз.-мат. наук, доцент.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від 15.11. 2024 р. № С-882

2. Термін подання студентом роботи 10. 06. 2025 р.

3. Вихідні дані до роботи:

- вивчити предметну область, проаналізувати існуючі логічні ігри;
- розглянути і використати алгоритми, які лежать в основі математичної моделі представлення ігрового поля та ігрової динаміки;
- спроектувати логічну гру з допомогою мови програмування Python та бібліотеки Tkinter.
- представити результати роботи ігрового додатку.

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Розділ 1. Стан проблемної області

Розділ 2. Інформаційне та математичне забезпечення

Розділ 3. Програмне та технічне забезпечення

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Додаток А

6. Дата видачі завдання 18 листопада 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд літературних даних	18.11-30.12.2024	виконано
2	Розділ 1. Стан проблемної області	01.01-15.02.2025	виконано
3	Розділ 2. Інформаційне та математичне забезпечення	16.02-30.03.2025	виконано
4	Розділ 3. Програмне та технічне забезпечення	01.04-15.05.2025	виконано
5	Оформлення дипломної роботи	16.05-30.05.2025	виконано
6	Підготовка до захисту дипломної роботи, оформлення презентації	01.06-10.06. 2025	виконано

Студент _____ Ільків А.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Пірко І.Б.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 71 сторінку пояснювальної записки, 9 рисунків, 3 таблиці, 18 джерел, 2 додатки.

У дипломній роботі розглянуто процес розроблення логічної гри «Сокобан» засобами програмування Python із використанням бібліотеки Tkinter для створення графічного інтерфейсу. Запропоновано математичну модель ігрового простору, яка включає поле, гравця та об’єкти, що підлягають переміщенню. Реалізовано алгоритми взаємодії користувача з ігровим середовищем, зокрема переміщення гравця, перевірки умов досягнення мети та ведення журналу дій. Отримані результати можуть бути використані для створення схожих ігор, вивчення програмування ігрової логіки та навчання розробці інтерактивних додатків.

Ключові слова: *логічна гра, алгоритми переміщення, ігровий інтерфейс, Python, Tkinter*.

ABSTRACT

Diploma paper contains 71 pages of explanatory note, 9 figures, 3 tables, 18 sources, 2 appendix.

The thesis describes the process of developing the Sokoban logic game in Python programming tools using the Tkinter library to create a graphical interface. A mathematical model of the game space is proposed, which includes the field, the player, and the objects to be moved. Algorithms for user interaction with the game environment, including player movement, checking the conditions for achieving the goal, and keeping a log of actions, are implemented. The results obtained can be used to create similar games, study game logic programming, and learn how to develop interactive applications.

Keywords: *logic game, movement algorithms, game interface, Python, Tkinter*.

ТЕХНІЧНЕ ЗАВДАННЯ

В дипломній роботі потрібно розробити гру “Сокобан” засобами Python.

1. Дослідити концепцію гри, вивчити правила гри, її базову механіку, можливі модифікації та вимоги до реалізації.

2. Розробити математичну модель ігрового простору: представити ігрове поле, гравця, коробки та цільові точки.

3. Реалізувати алгоритми переміщення гравця відповідно до правил гри, врахування перешкод, коробок і граничних меж поля.

4. Створити з допомогою бібліотеки Tkinter графічний інтерфейс гри, використати для реалізації візуального відображення ігрового поля, елементів та взаємодії користувача з грою.

5. Додати функціонал для завантаження і збереження рівнів, забезпечити можливість завантаження різних рівнів гри з файлу та збереження прогресу гравця.

6. Реалізувати додаткові функції: додати можливість відкату ходу, автоматичного збереження стану гри та відображення кількості виконаних ходів.

7. Перевірити роботу гри на різних рівнях складності, протестувати правильність алгоритмів та відповідність вимогам технічного завдання.

.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

Canvas – інтерактивна платформа для створення, редагування проектів в реальному часі;

Kivy – відкритий фреймворк для розробки кросплатформених мобільних та десктопних додатків на Python, який підтримує багатокористувацький інтерфейс.

GUI (Graphical User Interface) – графічний інтерфейс користувача, який дозволяє взаємодіяти з комп'ютерними програмами за допомогою візуальних елементів.

Listbox – елемент графічного інтерфейсу користувача, який дозволяє відображати список варіантів або елементів, з яких користувач може вибрати один або кілька.

Pygame – бібліотека для Python, яка дозволяє створювати ігри та мультимедійні додатки.

Pyglet – це бібліотека для Python, яка призначена для розробки відеоігор та мультимедійних додатків, надаючи функціонал для роботи з графікою, звуком та введенням, а також підтримує кросплатформеність.

PyOpenGL – бібліотека для Python, яка надає обгортку для OpenGL, дозволяючи розробляти 3D-графіку та візуалізацію за допомогою OpenGL в Python-додатках.

Python – високорівнева мова програмування, що використовується для розробки веб-додатків, ігор, аналізу даних, штучного інтелекту та багатьох інших застосувань.

Tkinter – бібліотека для створення графічних інтерфейсів користувача в Python, яка надає інструменти для створення вікон, кнопок, міток, текстових полів та інших елементів інтерфейсу.

Unity – кросплатформений рушій для розробки відеоігор, який дозволяє створювати 2D та 3D ігри, інтерактивні додатки та симуляції.

ВСТУП

Актуальність дипломної роботи

Логічні ігри, такі як “Сокобан”, є популярними завдяки своїй здатності розвивати мислення, що робить тему розробки таких ігор актуальною в сучасному суспільстві. Використання мови програмування Python для створення гри демонструє її універсальність і популярність серед розробників, що сприяє актуальності цього вибору для навчання і професійного розвитку. Графічні інтерфейси на основі бібліотеки Tkinter забезпечують легкість розробки і простоту розгортання додатків, що відповідає сучасним тенденціям у створенні програмного забезпечення. Розробка ігор сприяє популяризації програмування серед молоді та стимулює інтерес до вивчення алгоритмів і структур даних.

Логічні ігри часто використовуються у сферах освіти та тренувань для розвитку навичок аналізу та планування, що підкреслює практичну важливість дослідження. Розробка гри на основі математичних моделей показує застосування теоретичних знань у реальних задачах, що є актуальним у сучасній освітній практиці. Ігрові додатки дозволяють досліджувати питання інтерфейсу користувача та оптимізації взаємодії, що є важливим у контексті створення програмного забезпечення.

Предмет дослідження – процес розроблення логічної гри “Сокобан” із використанням мови програмування Python та бібліотеки Tkinter.

Об’єкт дослідження – програмні засоби та алгоритми, що забезпечують створення логічної гри “Сокобан”.

Мета роботи – розроблення логічної гри “Сокобан” із використанням Python та бібліотеки Tkinter, що забезпечує візуалізацію ігрового процесу, коректну роботу алгоритмів переміщення гравця та зручний інтерфейс користувача.

Завдання:

1. Дослідити концепцію гри “Сокобан”, вивчити правила гри, її базову механіку, можливі модифікації та вимоги до реалізації.

2. Розробити математичну модель ігрового простору, представити ігрове поле, гравця, коробки та цільові точки.

3. Розробити адгоритми переміщення гравця відповідно до правил гри, врахування перешкод, коробок і граничних меж поля.

4. Створити з допомогою бібліотеки Tkinter графічний інтерфейс гри, використати для реалізації візуального відображення ігрового поля, елементів та взаємодії користувача з грою.

5. Додати функціонал для завантаження і збереження рівнів, забезпечити можливість завантаження різних рівнів гри з файлу та збереження прогресу гравця.

6. Реалізувати додаткові функції, додати можливість відкату ходу, автоматичного збереження стану гри та відображення кількості виконаних ходів.

7. Перевірити роботу гри на різних рівнях складності, протестувати правильність алгоритмів та відповідність вимогам технічного завдання.

Практичне значення одержаних результатів

Розроблена гра “Сокобан” може бути використана як навчальний інструмент для ознайомлення з основами алгоритмізації та програмування на Python. Реалізація графічного інтерфейсу за допомогою бібліотеки Tkinter демонструє можливості створення інтерактивних додатків без використання складних фреймворків. Алгоритми переміщення персонажа та управління ігровим простором можуть стати основою для створення інших логічних або стратегічних ігор. Отримані результати можуть бути корисними для студентів, викладачів та розробників програмного забезпечення як приклад застосування методів математичного моделювання в ігрових додатках.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Бібліотеки Python для розробки ігор

Розглянемо роль Python у розробці ігор, її сильні сторони, обмеження та застосування у різних ігрових жанрах. Стрімке зростання індустрії розробки ігор стимулювало інтерес до різноманітних мов та інструментів програмування [1], [2]. Python з його доступним синтаксисом та великими бібліотеками зайняв свою нішу, особливо для специфічних завдань розробки та ігрових жанрів. Однак щоб точно оцінити її придатність для різних проектів з розробки ігор, важливо розуміти її обмеження, а також сильні сторони.

Python, хоча традиційно не вважається основною мовою для високопродуктивних ігрових рушіїв, пропонує поєднання простоти використання, великих бібліотек та можливостей швидкого створення прототипів, що робить її цінним інструментом для різноманітних завдань з розробки ігор, особливо для освітніх цілей, невеликих проектів та специфічних ігрових механік. Його сильні сторони полягають не в тому, що він конкурує з відомими ігровими рушіями, побудованими на C++ або C#, а в тому, що він доповнює їх і забезпечує гнучку, доступну платформу для навчання, експериментів і задоволення специфічних потреб розробки.

Розглянемо основні бібліотеки Python, що використовуються для розробки ігор, порівняємо їхні можливості, сильні та слабкі сторони. Pygame – це широко розповсюджена бібліотека та розглянемо інші варіанти для конкретних потреб. Вибір бібліотеки суттєво впливає на процес розробки, впливаючи на такі фактори, як простота використання, продуктивність та специфічні функціональні можливості.

Pygame [1], [2], [3-5], [12] є найпоширенішою бібліотекою в розглянутій літературі. Її популярність зумовлена простотою використання та широкими функціональними можливостями для розробки 2D ігор. Багато дослідників та розробників використовували Pygame для створення широкого спектру 2D ігор, включаючи головоломки, [6], [7], космічні шутери [8] та навчальні ігри [13]. Його кросплатформенна сумісність є значною перевагою, що спрощує розгортання на

різних операційних системах. Однак існують і обмеження. Продуктивність Pygame може бути вузьким місцем у складних іграх, а її можливості для 3D-графіки обмежені, [4]. Це зумовлює необхідність використання додаткових бібліотек або рушіїв для більш вимогливих проектів. Відкритий характер Pygame [7] сприяє його доступності та широкому розповсюдженню серед спільноти. Обширна документація та легкодоступні навчальні посібники ще більше підвищують зручність використання, що робить його ідеальним вибором для початківців та освітніх цілей.

Окрім Pygame, декілька інших бібліотек пропонують спеціалізовані функції для розробки ігор на Python. Хоча ці бібліотеки рідше обговорюються, вони заслуговують на увагу через свої унікальні переваги. Pyglet [2] є ще одним популярним варіантом, що пропонує інший підхід до розробки 2D ігор. Його зосередженість на роботі з вікнами та інтеграції з OpenGL забезпечує більш прямий шлях до апаратного прискорення, потенційно покращуючи продуктивність порівняно з Pygame у певних сценаріях. Однак, документація та підтримка Pyglet може бути не такою широкою, як у Pygame. Kivy [14], хоча і не є бібліотекою для розробки ігор, пропонує потужні можливості для створення крос-платформних додатків зі складним користувацьким інтерфейсом. Її потенціал для розробки ігор полягає в тому, що вона придатна для створення унікального інтерактивного досвіду, особливо з використанням сенсорних інтерфейсів. Однак її придатність для високопродуктивної розробки ігор потребує подальшого вивчення.Такі бібліотеки як PyOpenGL [4] пропонують доступ до OpenGL для програмування 3D-графіки, хоча їх ефективна інтеграція в повноцінний ігровий рушій вимагає значних зусиль з розробки.

Вибір ігрового жанру суттєво впливає на вибір інструментів та технологій розробки. Доступність Python та потужні можливості Pygame роблять її особливо придатною для розробки 2D ігор [10-12]. Численні приклади демонструють цю універсальність. Головоломки, наприклад, добре підходять до можливостей Python [6], [7], дозволяючи створювати захопливий логічний ігровий процес. Космічні шутери [5], [8] виграють від можливостей Pygame по обробці спрайтів та виявленню зіткнень, що дозволяє легко створювати швидкі, візуально привабливі ігри. Освітні

ігри [13] також використовують простоту використання Python, дозволяючи розробникам створювати інтерактивні навчальні програми, не занурюючись у складну механіку рушія. Простота Python робить її гарним вибором для розробки освітніх ігор, допомагаючи студентам вивчати концепції програмування і водночас створювати цікаві ігри [15], [16].

Придатність Python для розробки 3D ігор менш широко задокументована в літературі. Хоча це не є неможливим, але створює більше проблем. Накладні витрати на продуктивність, пов’язані з інтерпретованою природою Python, стають більш значущими у 3D-іграх, які за своєю природою є більш вимогливими до обчислень. Хоча такі бібліотеки, як PyOpenGL [4], надають доступ до функцій OpenGL, створення надійного та продуктивного рушія для 3D ігор з нуля за допомогою цих інструментів вимагає значного досвіду та ймовірних компромісів у продуктивності. Використання ігрових рушіїв, таких як Unity, які можуть інтегрувати сценарії на Python для конкретних завдань, пропонує більш практичний підхід до розробки 3D-ігор. Однак цей підхід вимагає знання як Python, так і обраного ігрового рушія, що збільшує складність процесу розробки. Суттєвим обмеженням залишається відсутність доступних високопродуктивних 3D-бібліотек у мові Python, порівняно з C++ або C# [4]. Це вимагає значних зусиль з оптимізації або використання зовнішніх бібліотек та рушіїв для досягнення задовільної продуктивності у складних 3D-середовищах ігор. Як наслідок наразі Python рідше обирають для розробки масштабних 3D-ігор порівняно з іншими мовами, більш оптимізованими для обробки графіки.

1.2. Методології розробки ігор на Python

Розглянемо різні методології розробки ігор у поєднанні з Python, проаналізуємо їхню ефективність та придатність для різних масштабів проектів. Вибір методології суттєво впливає на успіх проекту, впливаючи на такі фактори, як ефективність, ремонтопридатність та масштабованість.

Простота у використанні та можливості швидкого створення прототипів часто називають серед основних переваг Python [1], [9-12]. Лаконічний синтаксис та

легкодоступні бібліотеки дозволяють розробникам швидко створювати функціональні прототипи та тестувати ігрові механіки без великих налаштувань чи шаблонного коду. Ця можливість швидкої ітерації є безцінною на початковому етапі проектування, дозволяючи швидко експериментувати та коригувати ігрові механіки на основі тестування та зворотного зв'язку. Цей підхід особливо корисний в освітньому середовищі [13], де студенти можуть швидко створювати і тестувати свої ідеї, сприяючи розвитку підходу навчання на практиці. Ітеративний характер цієї методології сприяє експериментуванню та постійному вдосконаленню, дозволяючи розробникам вирішувати проблеми та вдосконалювати ігровий дизайн протягом усього процесу розробки. Коротші цикли розробки, пов'язані зі швидким прототипуванням, добре підходять для невеликих проектів і навчальних середовищ, забезпечуючи швидший час виконання і швидший зворотний зв'язок.

Хоча не обговорюються конкретні методології, що використовуються при розробці ігор на Python, акцент на ітеративній розробці та швидкому прототипуванні чітко вказує на придатність гнучких практик [9-12]. Притаманна Python гнучкість і легкість модифікації роблять її добре пристосованою до ітеративної природи гнучкої розробки. Гнучкі методології, такі як Scrum або Kanban, можна ефективно застосовувати для управління процесом розробки, розбиваючи великі проекти на менші, керовані завдання. Такий підхід дозволяє частіше отримувати зворотній зв'язок та адаптуватися до мінливих вимог, що є важливим для складних ігрових проектів. Використання систем контролю версій таких як Git має вирішальне значення в гнучкому середовищі, дозволяючи ефективно співпрацювати та відстежувати зміни протягом усього життєвого циклу розробки. Однак детальний аналіз того, як саме різні гнучкі фреймворки застосовуються у розробці ігор на Python, потребує подальшого спеціального дослідження.

Інтерпретована природа Python, хоча і сприяє простоті використання, може призвести до обмеження продуктивності порівняно з компільованими мовами, такими як C++ або C#, особливо в іграх з інтенсивними обчисленнями [2], [4]. Ця різниця у продуктивності стає більш помітною у ресурсномістких іграх зі складними

фізичними симуляціями, системами штучного інтелекту або графікою високої роздільної здатності. Накладні витрати на інтерпретацію можуть призвести до сповільнення швидкості виконання та потенційного відставання, особливо у сценаріях у реальному часі. Стратегії зменшення вузьких місць у продуктивності включають використання оптимізованих алгоритмів, використання NumPy для чисельних обчислень або використання Cython для компіляції критичних до продуктивності ділянок коду, щоб підвищити швидкість. Інший підхід полягає у використанні Python переважно для ігрової логіки та написання сценаріїв, а завдання з інтенсивними обчисленнями перекласти на зовнішні бібліотеки або рушії, написані на більш продуктивних мовах. Вибір ігрового рушія суттєво впливає на продуктивність; деякі рушії краще оптимізовані для інтеграції Python, ніж інші.

Відносний дефіцит легкодоступних, високопродуктивних бібліотек 3D-графіки в Python порівняно з такими мовами, як C++ або C#, є значною перешкодою для 3D-ігрових проектів [4]. Хоча такі бібліотеки, як PyOpenGL, надають доступ до функцій OpenGL, створення повноцінного, продуктивного рушія для 3D-ігор вимагає значних зусиль та досвіду. Крім того, ефективне управління пам’яттю та ресурсами при розробці 3D ігор вимагає ретельної оптимізації, що може бути більш складним у Python порівняно з мовами з більш прямим контролем над управлінням пам’яттю. Інтеграція Python з існуючими рушіями 3D ігор (такими як Unity або Unreal Engine) пропонує більш практичне рішення. Однак такий підхід створює складнощі в управлінні взаємодією між скриптами Python та основними функціями ігрового рушія. Тому, хоча розробка 3D-ігор на Python не є неможливою, вона стикається з більшими труднощами, ніж розробка 2D-ігор, через обмеження продуктивності та відсутність широко розповсюджених високопродуктивних 3D-бібліотек, спеціально розроблених для Python.

Роль Python у розробці ігор полягає не в заміні усталених високопродуктивних механізмів, таких як ті, що створені на C++ або C#. Натомість його цінність полягає в його доступності, можливостях швидкого прототипування та його придатності для певної ігрової механіки та невеликих проектів [1]. Його простота у використанні та великі бібліотеки роблять його потужним інструментом для освітніх цілей, що

дозволяє студентам вивчати концепції програмування під час створення функціональних ігор. Його можливості швидкого створення прототипів також є корисними для незалежних розробників і невеликих команд, дозволяючи їм швидко повторювати ідеї та експериментувати з різними ігровими механізмами. Інтеграція Python з іншими технологіями, такими як ігрові движки чи спеціалізовані бібліотеки, розширює його потенціал, дозволяючи розробникам використовувати сильні сторони Python для конкретних завдань, одночасно покладаючись на інші інструменти для критичних для продуктивності аспектів розробки ігор. Майбутні дослідження можуть бути зосереджені на покращенні продуктивності Python для розробки 3D-ігор, вивченні нових методів оптимізації та розробці більш надійних 3D-бібліотек, спеціально оптимізованих для Python. Подальше дослідження застосування гнучких методологій у розробці ігор на Python, зокрема аналіз ефективності та результативності різних гнучких фреймворків у цьому контексті, також було б цінним. Постійний розвиток бібліотек і інструментів Python свідчить про те, що його важливість у розробці ігор, ймовірно, продовжуватиме зростати, особливо в конкретних нішах і для проектів, які віддають перевагу швидкому створенню прототипів, простоті розробки та доступності.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Розроблення графічного інтерфейсу програм з допомогою бібліотеки Tkinter

Tkinter - це стандартна бібліотека для створення графічних інтерфейсів користувача (GUI) в Python. Вона є обгорткою для бібліотеки Tcl/Tk, яка забезпечує інструменти для створення вікон, кнопок, текстових полів та інших елементів інтерфейсу. Tkinter дозволяє програмістам створювати десктопні додатки з графічними інтерфейсами без необхідності використовувати складніші фреймворки.

Серед різних фреймворків для створення графічних інтерфейсів Tkinter єдиний фреймворк, який входить до стандартної бібліотеки Python. Важливою перевагою Tkinter є його кросплатформеність, тому той самий код працює на Windows, macOS і Linux.

Як вже згадувалося, Tkinter - це стандартна бібліотека GUI для Python, яка є обгорткою для інструментарію Tk. Tkinter базується на інструментарії Tk, який був спочатку розроблений для мови команд Tcl. Оскільки Tk дуже популярний, його було портовано до багатьох інших мов програмування, включаючи Perl (Perl/Tk), Ruby (Ruby/Tk) та Python (Tkinter).

Портативність і гнучкість розробки GUI з Tk роблять його ідеальним інструментом для створення та впровадження різноманітних додатків. Python з Tkinter надає швидший і ефективніший спосіб створення додатків, які б зайняли багато часу, якби їх потрібно було програмувати безпосередньо на C/C++ за допомогою бібліотек операційної системи. В Tkinter використовують основні будівельні блоки, відомі як віджети, для створення різноманітних додатків.

Основні кроки налаштування GUI-додатка з використанням Tkinter у Python такі. Спочатку імпортують модуль Tkinter. Другий крок - створення об’єкта верхнього рівня вікна, який містить увесь GUI-додаток. На третьому етапі потрібно налаштувати всі компоненти GUI та їх функціональність. Далі потрібно підключити ці компоненти GUI до основного коду додатка.

У програмуванні графічного інтерфейсу всі основні віджети створюються на об’єкті вікна верхнього рівня. Об’єкт вікна верхнього рівня створюється за допомогою класу Tk в Tkinter. Створимо вікно верхнього рівня:

```
import tkinter as tk
# створення об’єкта вікна верхнього рівня
window = tk.Tk()
# встановлення заголовка вікна
window.title("Програма на Tkinter")
# запуск головного циклу подій
window.mainloop()
```

Цей код створює вікно з заголовком "Програма на Tkinter". Після цього вікно буде залишатися відкритим, поки ви не закриєте його.

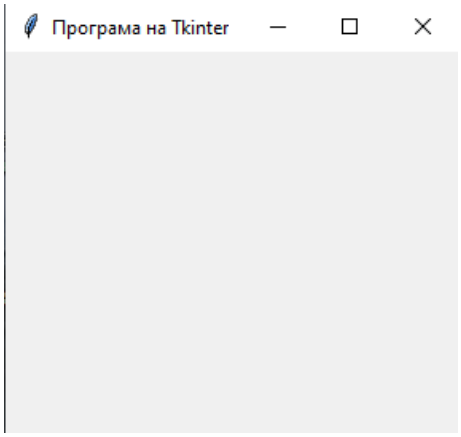

Рисунок 2.1 – Створення вікна

Метод tk.Tk() створює новий об’єкт вікна верхнього рівня (головне вікно програми). Це обов’язковий крок для створення будь-якого графічного інтерфейсу в Tkinter. Метод window.title("Програма на Tkinter") встановлює заголовок для вікна. Це дозволяє задати текст, який з’являється в верхній частині вікна програми. Метод window.mainloop() запускає основний цикл подій програми. Це необхідно для того, щоб вікно залишалося відкритим і взаємодіяло з користувачем, обробляючи події, такі як натискання кнопок чи переміщення вікна. Цей метод блокує подальше виконання програми, поки користувач не закриє вікно. Ці методи є основними для створення базового графічного інтерфейсу за допомогою Tkinter. У Tkinter є кілька основних концепцій, які використовують для створення графічних інтерфейсів користувача (GUI). Це вікна, віджети та фрейми.

Вікно - основний контейнер для всіх віджетів у Tkinter. Усі елементи інтерфейсу, такі як кнопки, мітки, текстові поля будуть розміщені всередині вікна. Верхнє вікно (top-level window) створюється за допомогою класу Tk. Вікно, яке створюють за допомогою цього класу, є основним вікном додатку.

Віджети - елементи графічного інтерфейсу, які взаємодіють із користувачем. Це можуть бути кнопки, мітки, текстові поля, поля вводу та інші елементи, які додають до вікна. Найпопулярніші віджети в Tkinter:

- Label: текстова мітка;
- Button: кнопка;
- Entry: поле для вводу тексту;
- Text: текстова область;
- Canvas: площина для малювання;
- Checkbutton: чекбокс;
- Radiobutton: радіокнопка.

Фрейм - це контейнер для організації інших віджетів, він дозволяє групувати віджети разом у логічному блоці. Фрейми можуть бути використані для організації елементів у декілька рядків або стовпців, надаючи більше контролю над їхнім розташуванням.

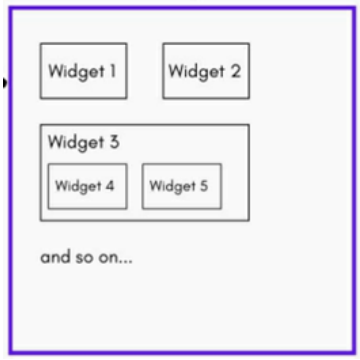

Рисунок 2.2 – Об’єкт вікна root (кореневе вікно)

Об’єкт вікна верхнього рівня в програмуванні GUI містить усі малі об’єкти вікон, які будуть частиною повного GUI. Ці малі об’єкти вікон можуть бути текстовими мітками, кнопками, списками, вони відомі як віджети.

Наявність об’єкта вікна верхнього рівня буде діяти як контейнер, в якому розміщують віджети: win = tkinter.Tk().

Об’єкт, який отримують при виклику tkinter.Tk(), називають кореневим вікном Root Window. Вікна верхнього рівня є самостійними частинами додатку, також можна мати більше одного вікна верхнього рівня, але тільки одне з них повинно бути кореневим вікном. Потрібно повністю спроектувати всі віджети, після цього додати функціональність. Віджети можуть бути самостійними або контейнерами. Якщо один віджет містить інші віджети, він вважається батьківським для цих віджетів.

Якщо віджет міститься в іншому віджеті, він називається дитиною батьківського віджета, а батьківський віджет є наступним найближчим контейнером. Віджети також мають деякі подібні поведінки, такі як натискання кнопки або ввід тексту в текстове поле, тому до цих дій прикріплені події. Поведінка віджетів генерує події, а відповідь GUI на події відома як зворотні виклики Callbacks, оскільки вони викликають функцію для обробки події.

Віджети в Tkinter

Існують різні елементи управління, такі як кнопки, мітки, смуги прокручування, радіокнопки та текстові поля, що використовуються в GUI-додатку. Ці компоненти або елементи управління графічного інтерфейсу відомі як віджети в Tkinter.

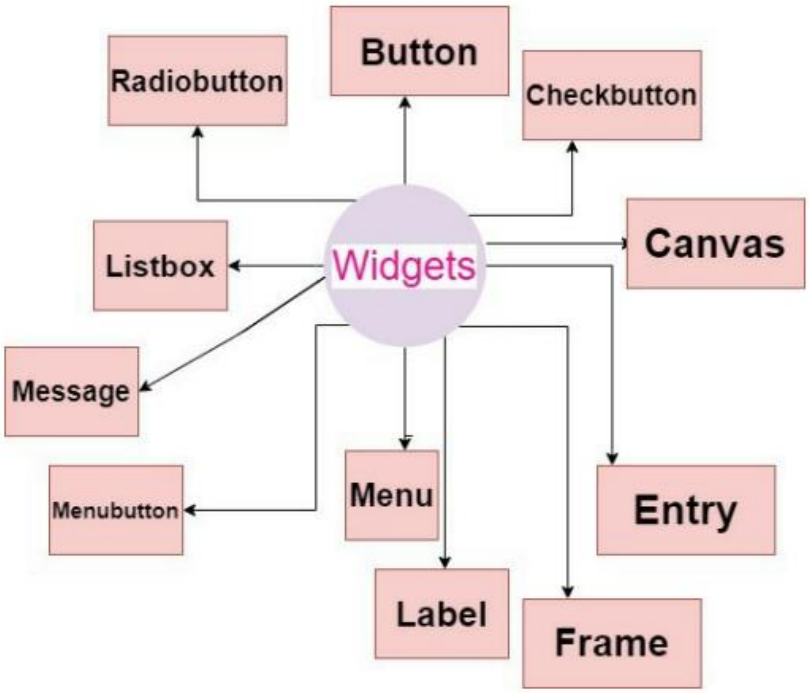

Рисунок 2.3 – Віджети в Tkinter

Таблиця 2.1 – Базові віджети в Tkinter

Назва віджета	Опис
Button	для додавання кнопки в додаток використовують віджет Button.
Canvas	для побудови складних макетів та зображень використовують віджет Canvas.
Entry	для відображення однорядкового текстового поля, яке приймає значення від користувача, використовують віджет Entry.
Listbox	щоб надати користувачу список варіантів, використовують віджет Listbox.
Radiobutton	якщо потрібно, щоб кількість варіантів відображалася у вигляді радіокнопок, використовують віджет Radiobutton. Можна вибрати лише одну опцію за раз.
Scrollbar	для прокручування вікна вгору та вниз використовують віджет Scrollbar.
Frame	для групування та організації інших віджетів використовують віджет Frame. Він діє як контейнер,в якому знаходяться інші віджети.

2.2. Математична модель гри “Сокобан”

Математична модель ігрового простору. Гра “Сокобан” відтворює простір, який представляється у вигляді двовимірного масиву розміром n × m, де n - кількість рядків, m - кількість стовпців. Кожна клітинка цього масиву може мати один із кількох можливих станів:

- 0 - порожня клітинка;
- 1 - стіна;
- 2 - коробка;
- 3 - ціль (місце для коробки);
- 4 - персонаж (гравець);

- 5 - коробка на цілі;
- 6 - персонаж на цілі.

Задача гри полягає в тому, щоб переміщати вантажника та коробки по ігровому полю таким чином, щоб усі коробки були розміщені на цілі (позначено клітинками 3). Переміщення вантажника обчислюється за допомогою математичних операцій, що базуються на координатах його поточної позиції (x_{man},y_{man}) та бажаного напрямку [18]:

- якщо вантажник рухається вгору, то його нові координати будуть ($x_{man}-1$,y_{man});
- якщо вантажник рухається вниз, то нові координати ($x_{man}+1$,y_{man});
- якщо вантажник рухається вліво, то нові координати (x_{man},$y_{man}-1$);
- якщо вантажник рухається вправо, то нові координати (x_{man},$y_{man}+1$).

Для кожного кроку перевіряється, чи не є нові координати стіною або іншою перешкодою. Якщо клітинка є вільною або містить коробку, вантажник може переміщатися.

Переміщення вантажника в грі Сокобан є центральним компонентом ігрового процесу, оскільки саме гравець, керуючи вантажником, взаємодіє з об’єктами на ігровому полі. Цей пункт описує математичну модель руху персонажа, умови його переміщення, а також алгоритм обробки команд.

Модель переміщення вантажника: вантажник гри позначається на ігровому полі координатами (x_{man},y_{man}), де x_{man} - індекс рядка, у якому знаходиться вантажник, y_{man} - індекс стовпця, у якому знаходиться вантажник. Його переміщення реалізується шляхом зміни його координат у відповідь на введення гравцем команди руху в одному з чотирьох напрямків:

- вгору (Up): $x_{new}=x_{man}-1$, $y_{new}=y_{man}$;
- вниз (Down): $x_{new}=x_{man}+1$, $y_{new}=y_{man}$;
- вліво (Left): $x_{new}=x_{man}$, $y_{new}=y_{man}-1$;
- вправо (Right): $x_{new}=x_{man}$, $y_{new}=y_{man}+1$.

Умови переміщення: перед кожним переміщенням вантажника перевіряються умови, які визначають, чи можливий рух у заданому напрямку. Це запобігає

порушенню правил гри (наприклад, вихід за межі поля або проходження через стіни).

Умова 1: Перевірка меж ігрового поля

Вантажник не може виходити за межі ігрового простору. Це перевіряється за допомогою умов:

$$0 \leq x_{\text{new}} < n \quad \text{та} \quad 0 \leq y_{\text{new}} < m \tag{2.1}$$

де n і m - кількість рядків та стовпців ігрового поля.

Умова 2: Перевірка типу клітинки: вантажник може переміщатися лише на клітинки, які є порожніми (0) або цільовими (3). Якщо клітинка $M[x_{new},y_{new}]$ має значення 1 (стіна), рух заборонений.

Умова 3. Переміщення коробки: якщо клітинка $M[x_{new},y_{new}]$ містить коробку (2 або 5), то можливість руху визначається наявністю вільного простору за коробкою в тому ж напрямку. Для перевірки використовується наступна формула:

$$M[x_{\text{new}} + \Delta x, y_{\text{new}} + \Delta y] \in \{0, 3\} \tag{2.2}$$

де Δx та Δy відповідають зміщенню, залежно від напряму руху.

Алгоритм переміщення вантажника можна розділити на кілька кроків: отримання команди: гравець вводить команду (натискає клавішу стрілки). Команда передається функції обробки подій. Обчислення нових координат: залежно від команди обчислюються нові координати (x_{new},y_{new}). Перевірка можливості переміщення: виконується перевірка умов. Оновлення ігрового стану: якщо переміщення можливе, змінюється стан клітинок на ігровому полі. Поточна клітинка вантажника змінюється на порожню (0) або цільову (3), нова клітинка вантажника змінюється на вантажника (4) або вантажника на цілі (6). Якщо вантажника рухає коробку, також оновлюються її координати та стан.

Оновлення координат вантажника: нові координати персонажа зберігаються як $x_{man}=x_{new}$, $y_{man}=y_{newx}$. Перевірка завершення гри: після кожного переміщення перевіряється, чи всі коробки знаходяться на цілях.

Алгоритм переміщення вантажника є базовим елементом гри Сокобан, що забезпечує взаємодію з ігровим простором. Використання перевірок меж поля, типу

клітинок та умов переміщення дозволяє дотримуватись правил гри. Реалізація переміщення вантажника безпосередньо пов’язана з логікою гри та формує основу для взаємодії між елементами на полі.

Алгоритм переміщення коробок: коли вантажник натрапляє на коробку, необхідно перевірити можливість її переміщення. Алгоритм переміщення коробки аналогічний алгоритму переміщення вантажника, але з урахуванням того, що коробка не може рухатися, якщо наступна клітинка або є стіною, або містить іншу коробку. Математично це можна виразити наступним чином:

Визначити координати клітинки, в яку буде переміщена коробка: (x_{box},y_{box}).

Перевірити, чи є ця клітинка вільною або містить ціль 3:

- якщо клітинка вільна або містить ціль, коробка може бути переміщена;
- якщо клітинка містить іншу коробку або стіну, переміщення неможливе.

Для вирішення рівня гри необхідно реалізувати алгоритм пошуку рішення. Оскільки гра має обмежену кількість можливих ходів і може мати різні варіанти розміщення коробок на полях, оптимальним є застосування методу пошуку в ширину або глибину. Ось базові математичні принципи для пошуку рішення:

- множина станів визначається як всі можливі конфігурації коробок і вантажника на полі;
- початковий стан - це розташування коробок і вантажника на стартовому рівні;
- кожен хід гри змінює стан гри, для кожного можливого ходу перевіряється, чи не призвів він до виграшного стану, де всі коробки знаходяться на цілі.

Оцінка складності пошуку рішення в гру “Сокобан” за допомогою алгоритмів пошуку в ширину або глибину може бути виконана через аналіз кількості можливих станів гри. Кількість можливих станів залежить від розміру ігрового поля та кількості коробок, тому складність вирішення завдання зростає експоненціально.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Розроблення графічного інтерфейсу гри “Сокобан” з допомогою бібліотеки Tkinter

Для створення графічного інтерфейсу гри “Сокобан” було обрано бібліотеку Tkinter. Це стандартна бібліотека для побудови GUI в Python, яка забезпечує гнучкий спосіб створення віконних додатків. Метою використання Tkinter є реалізація графічного відображення ігрового простору, включаючи сітку поля, вантажника та коробок.

Інтерфейс гри складається з таких компонентів:

- головне вікно: базове вікно, створене за допомогою класу Tk;
- ігрове поле: графічна сітка, яка реалізована на віджеті Canvas;
- панель управління грою: набір кнопок і текстове поле для взаємодії з грою;
- список логів: віджет Listbox для збереження та відображення дій гравця.

Графічне поле гри було створено за допомогою віджета Canvas, який дозволяє будувати прямокутники, що представляють ігрові клітинки. Поле будується як двовимірна сітка розміром n×m, де кожна клітинка має фіксований розмір. Код для створення сітки поля:

```
from tkinter import Tk, Canvas
# параметри поля
cell_size = 40  # розмір однієї клітинки
row_count, col_count = 10, 10  # кількість рядків і стовпців
# створення головного вікна
root = Tk()
root.title("Гра Сокобан")
# створення canvas для графічного поля
canvas = Canvas(root, width=col_count*cell_size, height=row_count*cell_size,
bg="white")
canvas.pack()
# побудова сітки
for row in range(row_count):
    for col in range(col_count):
        x1, y1 = col * cell_size, row * cell_size
        x2, y2 = x1 + cell_size, y1 + cell_size
        canvas.create_rectangle(x1, y1, x2, y2, fill="lightgray", outline="black")
```

```
root.mainloop()
```

Для переміщення вантажника використовуються події натискання клавіш, оброблені за допомогою методу bind. Кожна клавіша Up, Down, Left, Right відповідає певному напряму руху. Код для обробки подій:

```
def key_hndl(event):
    key = event.keysym
    if key == "Up":
        move_man(-1, 0)
    elif key == "Down":
        move_man(1, 0)
    elif key == "Left":
        move_man(0, -1)
    elif key == "Right":
        move_man(0, 1)
root.bind("<Key>", key_hndl)
```

Для управління грою було додано кнопки завантаження рівня, запуску та перезапуску гри, а також текстове поле для введення номера рівня:

```
from tkinter import Button, Entry, IntVar, ttk
# текстове поле для введення рівня
var_level = IntVar(value=1)
edt_level = ttk.Entry(root, width=5, textvariable=var_level)
edt_level.pack()
# кнопка для завантаження рівня
btn_load = Button(root, text="Load", command=fnc_load)
btn_load.pack()
```

Для запису та відображення дій гравця використано віджет Listbox. При кожному русі вантажника в лог записується команда:

```
from tkinter import Listbox
lbx_log = Listbox(root, width=20, height=10)
lbx_log.pack()
def log_action(action):
    lbx_log.insert("end", action)
    lbx_log.see("end")
```

Використання бібліотеки Tkinter дозволило створити зручний графічний інтерфейс для гри “Сокобан”. Відображення ігрового поля, обробка подій і керування грою реалізовані в компактному та функціональному вигляді. Бібліотека забезпечила можливість інтеграції графіки з основною логікою гри.

Сокобан - логічна гра, мета якої полягає в тому, що вантажник повинен розставити коробки по своїх місцях, штовхаючи їх перед собою. З допомогою клавішей стрілок переміщають вантажника. Якщо перед ним стоїть ящик і його можна зсунути, він зсувається. Кнопка дозволяє вибрати варіант рівня, номер якого введений в поле вводу.

Канва розбивається на 400 клітинок 20*20 і моделюється з допомогою двомірної таблиці cells – це двовимірний список комірок, кожна з яких містить значення та кольоровий квадрат з відповідним кольором.

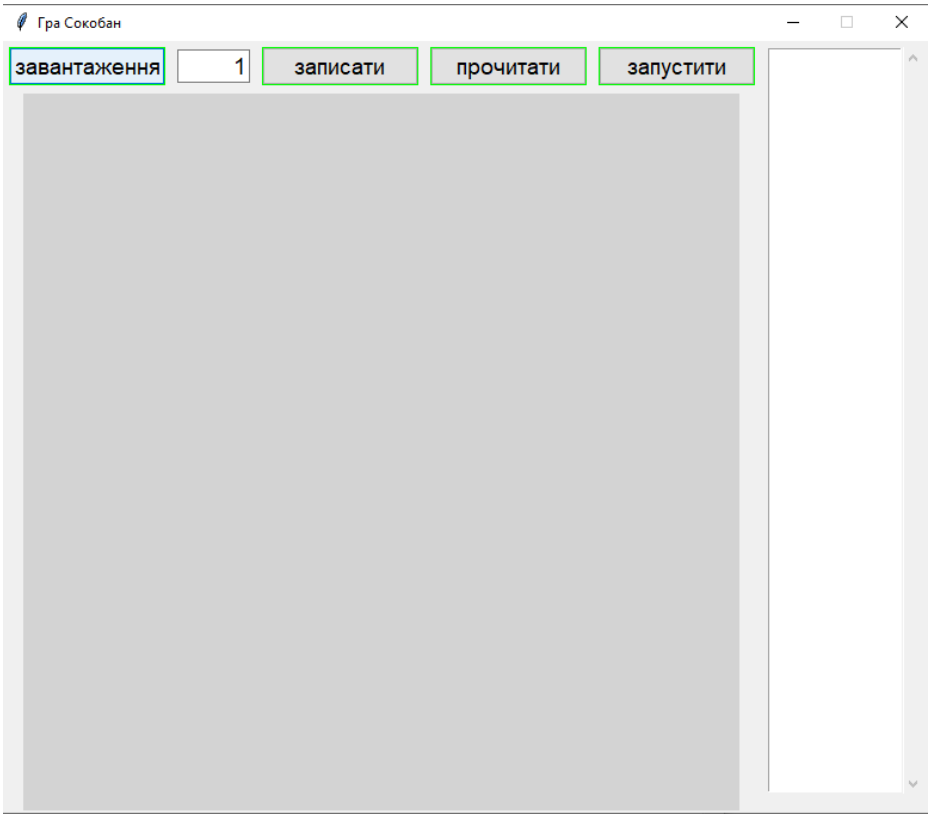

Рисунок 3.1 – Інтерфейс гри “Сокобан” при запуску

На ній знаходяться два елементи управління: командна кнопка load та текстове поле із номером рівня для завантаження в гру. Вибирають номер рівня 1 та натискають відповідну кнопку.

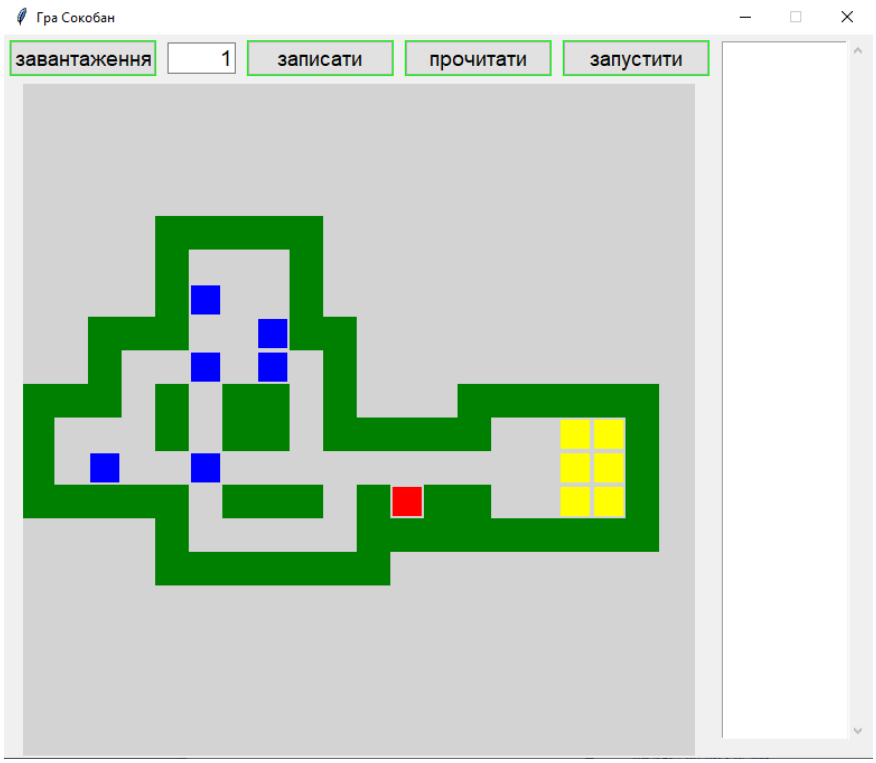

Рисунок 3.2 – Рівень 1 завантажено в ігровий додаток

Гра доволяє вибрати один з 50 рівнів - варіантів приміщень і початкового розташування коробок. Всі рівні нумеруються від 1, зберігаються в одному бінарному файлі Sokoban.bin. Зеленим кольором відображено стіни приміщення, червоним – вантажника, жовтий – місце, куди потрібно доставити груз, синій – коробки з вантажем, які потрібно з допомогою вантажника розмістити на потрібних місцях. Коли коробка на місці, її колір змінюється з синього на фіолетовий. Переміщаючи вантажника клавішами управління, потрібно перемістити всі ящики в жовту позицію.

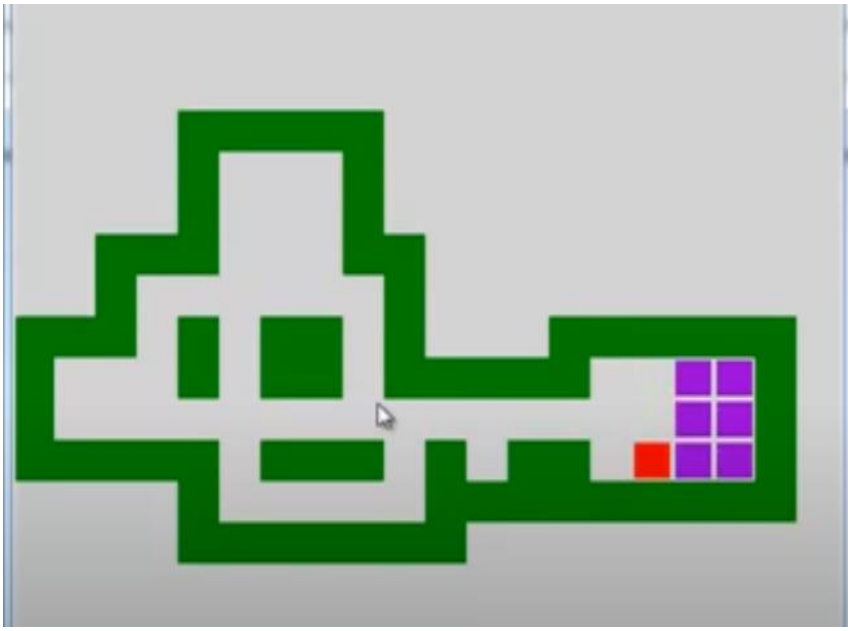

Рисунок 3.3 – Перший рівень гри пройдено

Рівні гри мають різну степінь складності. Їх можна завантажувати від 1 до 50.

```
from tkinter import *
from tkinter import ttk
from tkinter import font
```

Імпортують різні компоненти бібліотеки Tkinter, які використовуються для створення графічного інтерфейсу в Python, всі модулі та класи з цієї бібліотеки. Tkinter - бібліотека Python для створення графічних інтерфейсів, яка надає можливість створювати вікна, кнопки, текстові поля та інші елементи інтерфейсу.

ttk - це модуль, що належить до Tkinter і надає набір більш сучасних віджетів (елементів інтерфейсу), таких як кнопки, списки, прогрес-бари та інші компоненти. Вони мають кращий вигляд порівняно зі стандартними віджетами Tkinter. Ці імпорти дозволяють створювати графічний інтерфейс з допомогою Tkinter, а також налаштовувати зовнішній вигляд та стиль елементів інтерфейсу.

```
import inspect
import time
```

Модуль inspect надає функції для отримання інформації про об’єкти Python, такі як модулі, класи, функції, методи. Він може бути використаний, якщо потрібно проаналізувати виконання функцій або отримати додаткову інформацію під час розробки. Модуль time використовується для роботи з часом. Він дозволяє вимірювати час виконання програм, створювати затримки або форматувати час у

різних форматах. Його використовують для створення затримок між ходами, вимірювання часу для виконання певних дій.

```
row_beg = -1
col_beg = -1
row_count = 20
col_count = 20
man_row = 0
man_col = 0
```

Ініціалізують змінні, які використовуються для зберігання інформації про розмір і стан гри, а також позицію вантажника. Змінна row_beg використовується для зберігання індексу початкового ряду (координати) на ігровому полі. Змінна col_beg зберігає індекс початкового стовпця на ігровому полі. Змінна row_count вказує кількість рядів на ігровому полі. Поле має 20 рядків, що означає, що в грі буде сітка розміром 20x20. Змінна col_count вказує кількість стовпців на ігровому полі. Змінна man_row зберігає поточну позицію вантажника по вертикалі (ряди). Початково він знаходиться в першому ряду (індекс 0). Змінна man_col зберігає поточну позицію вантажника по горизонталі (стовпці). Початково персонаж знаходиться в першому стовпці (індекс 0). Ці змінні використовуються для відстеження розміру ігрового поля та місця розташування вантажника в грі. Вони є частиною механізму, що керує позицією гравця та іншими об’єктами на полі (коробками та цілями).

```
cell_size = 30
canv_width = cell_size * col_count
canv_height = cell_size * row_count
```

Визначають параметри, які пов’язані з розмірами клітинок та самого ігрового поля. cell_size = 30 - розмір однієї клітинки на ігровому полі в пікселях. Значення 30 означає, що кожна клітинка буде мати ширину та висоту 30 пікселів. Це дозволяє визначити розміри графічного відображення поля на екрані.

Змінна canv_width визначає ширину всього ігрового поля. Оскільки кожен стовпець має розмір cell_size пікселів, то ширина поля розраховується як добуток розміру клітинки на кількість стовпців (col_count). У даному випадку це буде: canv_width=30×20=600. Ігрове поле буде мати ширину 600 пікселів.

Змінна canv_height визначає висоту всього ігрового поля. Оскільки кожен ряд має розмір cell_size пікселів, то висота поля розраховується як добуток розміру клітинки на кількість рядків (row_count). У даному випадку це буде: canv_height=30×20=600. Ігрове поле буде мати висоту 600 пікселів.

Розміри ігрового поля будуть 600x600 пікселів, що означає, що кожен стовпець і ряд буде відображатися як квадрат з розмірами 30x30 пікселів. Ці параметри важливі для налаштування розмірів канвас в Tkinter, на якому буде відображатися поле гри.

Змінна cells = [] ініціалізує порожній список, який буде використовуватися для зберігання інформації про кожну клітинку на ігровому полі. Цей список може зберігати стан кожної клітинки поля: порожня клітинка, клітинка з коробкою, клітинка з вантажником, клітинка з місцем, куди потрібно перемістити коробку.

```
fon_color = "lightgray"
colors = [fon_color, "yellow", "blue", "darkviolet", "red", "green", "chocolate"]
```

Визначають кольори, які будуть використовуватися для різних елементів гри. Змінна fon_color задає колір фону ігрового поля. Це світло-сірий колір (lightgray). Він використовується для заповнення фону поля або інших елементів інтерфейсу, які повинні бути нейтральними та не відволікати увагу. Список кольорів colors використовується для різних елементів гри. Кожен колір у списку відповідає певному типу об’єкта або елемента на полі.

- fon_color – світло-сірий, використовується для фону;
- yellow – жовтий для позначення місця, куди потрібно доставити коробки;
- blue – синій використовується для позначення коробок, які потрібно перемістити;
- darkviolet – темно-фіолетовий використовується для коробок, які знаходяться на потрібних місцях;
- red – червоний для позначення вантажника;
- green – зелений для позначення стін приміщення;

Ці кольори дозволяють створити візуальне відображення гри, де кожен тип об’єкта (вантажник, коробка, стіна) має свій колір. Це покращує зручність для гравця та робить гру більш зрозумілою.

3.2. Розроблення функціональної частини ігрового додатку

Змінна step_time вказує на час затримки між кроками гравця в грі. Значення 0.2 означає, що між кожним кроком буде затримка в 0.2 секунди. Ця затримка є корисною для того, щоб кожен рух виглядав плавно і не відбувався занадто швидко, даючи гравцеві час для реакції або для того, щоб рухи виглядали більш природньо. Змінна flg_step є прапорцем, який використовується для відстеження стану кроку або дії гравця. Під час гри ця змінна може змінювати своє значення для вказівки на те, чи виконується зараз крок, чи очікується на наступний. Ці змінні використовуються для контролю темпу гри та відстеження прогресу кожного кроку, щоб забезпечити правильну взаємодію між гравцем та ігровим процесом.

```
def make_fragm(color_num, row_num, col_num):
    global cells
    cells[row_num][col_num][0] = color_num
    color = colors[ color_num]
    rect = cells[row_num][col_num][1]
```

Функція make_fragm(color_num, row_num, col_num) змінює стан клітинки на ігровому полі. Кожен елемент cells[row_num][col_num] є списком, в якому перший елемент (індекс 0) відповідає за зберігання колірного коду або значення, яке вказує на тип об’єкта в цій клітинці (вантажник, коробка). Функція make_fragm присвоює color_num (номер кольору) цьому елементу. Змінна color отримує значення кольору зі списку colors. Це дозволяє вибирати певний колір для клітинки на основі переданого номера кольору.

Функція make_fragm дозволяє змінювати колір клітинки на ігровому полі, а також працювати з графічними об’єктами, які представляють ці клітинки. Якщо гравець переміщає коробку або вантажника, ця функція може бути використана для оновлення кольору клітинки, де знаходиться новий об’єкт, або для зміни графічного відображення клітинки.

```
    if color_num in [1,2,3,4,6]:
```

```
        bg_color = fon_color
    else:
        bg_color = color
    canv.itemconfig(rect, fill = color, outline = bg_color)
```

Змінюють колір заповнення та обведення графічного об’єкта на ігровому полі. Ця умова перевіряє, чи значення color_num належить до списку [1, 2, 3, 4, 5]. У цьому випадку перевіряються специфічні кольори, які мають особливе значення для гри. Якщо color_num належить до одного з вищезгаданих значень, то змінна bg_color буде мати значення fon_color, яке є кольором фону. Якщо color_num не входить у список [1, 2, 3, 4, 5], то bg_color отримує значення color, що є кольором, вибраним для клітинки. Це означає, що фон буде того ж кольору, що й сам об’єкт.

Функція itemconfig() використовується для зміни властивостей графічного об’єкта на канвасі. rect - змінна, що посилається на прямокутник, який треба змінити. fill=color - параметр, який задає колір заповнення прямокутника. outline=bg_color - параметр, який задає колір контуру прямокутника. У результаті цей код змінює колір заповнення та обведення прямокутника на канвасі в залежності від значення color_num.

```
def read_cells():
    global man_row, man_col
    lst_loc = list( fh.read(400) )
    for num in range(400):
        row_num = num // 20
        col_num = num % 20
        color_num = lst_loc[num]
        make_fragm(color_num, row_num, col_num)
        if color_num == 4:
            man_row = row_num
            man_col = col_num
    man_colornum = cells[man_row][man_col][0]
```

Функція read_cells() відповідає за зчитування даних з файлу та оновлення ігрового поля відповідно до отриманої інформації. Зчитується колір (або інший параметр) для поточної клітинки, використовуючи значення в списку lst_loc. Це значення color_num буде використовуватись для визначення вигляду клітинки. Викликається функція make_fragm для зміни кольору або іншого вигляду клітинки

на полі на основі значення color_num. Ця функція також визначає, що розташування вантажника оновлюється на основі поточного значення color_num.

Якщо значення color_num дорівнює 4, то це означає, що клітинка містить вантажника. Після завершення циклу змінна man_colornum зберігає значення кольору або стану клітинки, де знаходиться вантажник. Функція read_cells() зчитує 400 значень з файлу, кожне з яких відповідає за стан певної клітинки на полі. Для кожної клітинки викликається функція make_fragm, яка оновлює вигляд клітинки відповідно до її значення. Якщо значення color_num для клітинки дорівнює 4, то оновлюються координати вантажника, щоб відслідковувати його місце розташування на полі.

```
def move_man(step_row, step_col):
    global cells, man_row, man_col
    man_colornum = cells[man_row][man_col][0]
    new_row = man_row + step_row
    new_col = man_col + step_col
    new_colornum = cells[new_row][new_col][0]
```

Функція move_man(step_row, step_col) відповідає за переміщення вантажника на ігровому полі на основі заданих кроків. man_row та man_col - координати поточного місцезнаходження вантажника на полі. Для обчислення нового рядка, на який потрібно перемістити вантажника, додається step_row до поточного рядка. step_row - крок у вертикальному напрямку. Якщо step_row = 1, вантажник переміщується вниз на 1 клітинку. Якщо step_row = -1, він переміщується вгору на 1 клітинку. Якщо step_row = 0, рядок не змінюється, вантажник не рухається вертикально. Для обчислення нового стовпця, на який потрібно перемістити вантажника, додається step_col до поточного стовпця man_col. step_col - крок у горизонтальному напрямку. Якщо step_col = 1, вантажник переміщується вправо на 1 клітинку. Якщо step_col = -1, він переміщується вліво на 1 клітинку. Якщо step_col = 0, стовпець не змінюється, вантажник не рухається горизонтально.

Після обчислення нових координат new_row і new_col функція зчитує стан нової клітинки, на яку вантажник намагається переміститися. Це значення допомагає визначити, чи можна його перемістити на цю клітинку (чи це порожня клітинка, чи вона зайнята коробкою чи стіною).

Функція move_man() має два параметри step_row і step_col, які визначають, як далеко і в якому напрямку вантажник має переміститися. Вона обчислює нові координати для нього на основі поточних координат і кроків, а також перевіряє стан нової клітинки. Таким чином вантажник може переміщатися по ігровому полю, перевіряючи умови для кожного кроку.

```
    if new_colornum == 5:
        return 0
```

Змінна new_colornum містить значення стану клітинки, на яку вантажник намагається переміститися. Це значення може відрізнятися в залежності від типу клітинки. Це може бути число, яке позначає різні об’єкти на полі, такі як стіни, коробки, цілі або порожні клітинки. Якщо new_colornum дорівнює 5, це означає, що клітинка містить об’єкт або стан, при якому не можна дозволити переміщення вантажник на цю клітинку. Якщо умова new_colornum == 5 виконується, функція повертає значення 0. Це може бути використано як сигнал для того, щоб повідомити про неможливість переміщення або про те, що потрібно припинити поточний процес (не дозволити рух на цю клітинку, якщо на ній є перешкода).

```
def move_man(step_row, step_col):
    global cells, man_row, man_col
    man_colornum = cells[man_row][man_col][0]
    new_row = man_row + step_row
    new_col = man_col + step_col
    new_colornum = cells[new_row][new_col][0]
    if new_colornum == 5:
        return 0
    man_row, man_col = new_row, new_col
    make_fragm(man_colornum, man_row, man_col)
    return 1
```

Якщо вантажник намагається рухатись на клітинку, де new_colornum == 5, функція поверне 0 і припинить рух. Якщо ж клітинка прохідна, позиція персонажа оновлюється, функція повертає 1, що означає успішне переміщення.

```
    if  new_colornum in [2,3]:
        next_row = new_row + step_row
        next_col = new_col + step_col
        next_colornum = cells[next_row][next_col][0]
        if next_colornum in[2,3,5]:
            return 0
```

```
if man_colornum == 4:
    color_num = 0
else:
    color_num = 1
make_fragm(color_num, man_row, man_col)
```

Додають ще одну умову для переміщення вантажника в грі, зокрема для взаємодії з коробками або іншими об’єктами на полі. Перевіряють, чи поточна клітинка (на яку вантажник намагається переміститися) містить коробку (2) або ціль (3). Це означає, що вантажник натрапив на об’єкт, з яким можна взаємодіяти (коробку). У цьому випадку гра буде перевіряти, чи можна перемістити цей об’єкт. Обчислюються нові координати, куди буде переміщено об’єкт (коробку), якщо вантажник спробує її рухати. Зчитується колір або стан нової клітинки, куди він намагається перемістити коробку, яку він штовхає. Перевіряють, чи можна перемістити об’єкт на нову клітинку. Якщо на новій клітинці знаходиться інша коробка (2), ціль (3) або стіна (5), то переміщення неможливе.

Викликається функція make_fragm(), яка оновлює вигляд клітинки після того, як вантажник перемістився або взаємодіє з об’єктами. Вона оновлює клітинку на новій позиції, де стоїть вантажник.

Якщо на поточній клітинці є коробка (2) або ціль (3), то перевіряється, чи можна перемістити цей об’єкт. Для цього обчислюються координати нової клітинки, куди має переміститися об’єкт. Якщо на цій клітинці вже є інша коробка, ціль або стіна, то переміщення неможливе, функція повертає 0. Якщо переміщення можливе, то поточна клітинка вантажника очищається, коробка переміщується на нову клітинку. Цей код дозволяє реалізувати основний механізм переміщення коробок у грі.

```
if new_colornum == 2:
    man_colornum = 4
else:
    man_colornum = 6
man_row = new_row
man_col = new_col
make_fragm(man_colornum, man_row, man_col)
```

Перевіряється, чи знаходиться вантажник на клітинці, яка містить коробку (2). Якщо так, то man_colornum = 4 - змінюється колір або стан вантажника (код 4

означає, що персонаж знаходиться на цілі або на початковій клітинці після переміщення коробки). Якщо на клітинці немає коробки (значення new_colornum не дорівнює 2), то man_colornum = 6 - змінюється колір або стан вантажника (код 6 позначає клітинку, на якій стоїть вантажник без коробки).

Змінні man_row та man_col оновлюються новими значеннями new_row та new_col, щоб змінити позицію вантажника на полі. Це оновлює його координати після того, як він перемістився. Функція make_fragm() викликається для оновлення вигляду клітинки, на якій тепер знаходиться вантажник. Вона використовує нові координати вантажника (нові man_row та man_col) та відповідний колір або стан (man_colornum), щоб оновити відображення на екрані.

Якщо вантажник переміщається на клітинку, де знаходиться коробка (код 2), то вантажник змінює свій стан на 4 (це означає, що він переміщує коробку або стоїть на цілі). Якщо ж він переміщається на іншу клітинку, то його стан змінюється на 6 (вільна клітинка або місце без коробки). Після зміни стану вантажника оновлюються його координати та відображення гри через виклик функції make_fragm(), яка оновлює вигляд клітинки на полі.

Цей фрагмент є частиною механізму переміщення вантажника, який взаємодіє з різними об’єктами на полі, такими як коробки або цілі. Код також обробляє зміни візуального відображення гри залежно від того, чи переміщає вантажник коробку, чи просто змінює своє положення на полі.

```
if next_colornum == 0:
    color_num = 2
else:
    color_num = 3
make_fragm(color_num, next_row, next_col)
return 2
```

Цей фрагмент коду відповідає за взаємодію вантажника з об’єктами на полі, зокрема коробками та цілями, оновлює відображення гри на основі того, чи вдалося перемістити коробку на нову клітинку, чи ні. Перевіряють, чи клітинка, на яку планується перемістити коробку, є порожньою (0). Це означає, що на новій клітинці немає жодного об’єкта (стін, коробок або цілей). Якщо клітинка порожня, то

встановлюється color_num = 2, що означає, що на цю клітинку буде поміщена коробка (2).

Якщо на новій клітинці вже є інший об’єкт (не порожня клітинка), то встановлюється color_num = 3, що вказує на те, що клітинка містить ціль (місце, куди потрібно перемістити коробку). Викликається функція make_fragm(), яка оновлює візуальне відображення гри, змінюючи колір клітинки. Якщо клітинка була порожньою, тепер на неї поміщається коробка (2). Якщо на клітинці вже була ціль, то на неї може бути переміщена коробка (3).

Якщо клітинка порожня (0), то на цю клітинку буде переміщена коробка (колір 2). Тобто, якщо вантажник може перемістити коробку на порожнє місце, коробка буде туди поставлена. Якщо клітинка не порожня (має інший колір), то на новій клітинці є ціль (3), і коробка буде переміщена на цільову клітинку. Це може бути механізм завершення рівня або просто переміщення коробки на місце, яке вказує на ціль. make_fragm() викликається для оновлення візуального відображення цієї клітинки, щоб змінився її вигляд в залежності від того, чи є там коробка або ціль.

Цей код забезпечує логіку для переміщення коробок на нові клітинки, де вони можуть або бути розміщені на порожньому місці, або націлені на клітинки, що позначають цілі.

```
if man_colornum == 4:
    color_num = 0
else:
    color_num = 1
make_fragm(color_num, man_row, man_col)
```

Перевіряють, чи man_colornum дорівнює 4. Код 4 позначає спеціальну клітинку, на якій вантажник вже був, наприклад, клітинка, що позначає стартову точку або клітинку, на якій він взаємодіє з об’єктами (рухає коробку чи досягнув цілі). Якщо man_colornum != 4, це означає, що вантажник знаходиться на клітинці, де код не дорівнює 4 (він рухається по іншому полю або вже переміщує коробку).

Встановлюється color_num = 1, що вказує на те, що вантажник зараз знаходиться на клітинці, клітинка не є порожньою. Це може означати, що клітинка вже зайнята вантажником (він знаходиться на відкритій клітинці або після того, як

він перемістив коробку). Викликається функція make_fragm(), яка змінює вигляд клітинки на полі. Вона оновлює колір або стан клітинки, а також змінює координати вантажника, щоб візуально відобразити його нову позицію.

Якщо вантажник покидає клітинку, яка була позначена як 4, то клітинка повинна стати порожньою (0).. Викликається функція make_fragm(), яка змінює відображення цієї клітинки в грі, щоб показати, що вантажник тепер знаходиться на іншій клітинці. Цей код дозволяє коректно відображати зміни позиції вантажника на полі, оновлюючи клітинки, з яких він покидає, та на яких він з'являється.

```
if new_colornum == 0:
    man_colornum = 4
elif new_colornum == 1:
    man_colornum = 6
man_row = new_row
man_col = new_col
make_fragm(man_colornum, man_row, man_col)
return 1
```

Перевіряється, чи new_colornum (колір клітинки, на яку вантажник переміщується) дорівнює 0. Якщо клітинка порожня (код 0), то man_colornum = 4 - встановлюється новий код для вантажника, який може позначати, що він переміщується на порожню клітинку (або на клітинку, яку він залишає після переміщення коробки).

Якщо вантажник переміщається на порожню клітинку (0), то він змінює свій колір або стан на 4 (це означає, що він покидає порожню клітинку). Якщо вантажник переміщається на клітинку з кодом 1, він змінює свій колір на 6 (це може бути клітинка з коробкою, або просто ще один тип клітинки, на якій він перебуває). Після переміщення оновлюється відображення клітинки, на яку переміщується вантажник. Функція повертає 1, що може означати успішне завершення переміщення.

Якщо вантажник переміщається на порожню клітинку (0), то його стан змінюється на 4, він візуально відображається на новій клітинці. Якщо він переміщається на клітинку з кодом 1, наприклад, на клітинку з коробкою, то його стан змінюється на 6, він переміщається на нову клітинку.

```
def move_back(step_row, step_col, param):
```

```
    global cells, man_row, man_col
    if param not in ["1","2"]:  return 0
    step_row = - step_row;  step_col = - step_col
    man_colornum = cells[man_row][man_col][0]
    new_row = man_row + step_row
    new_col = man_col + step_col
    new_colornum = cells[new_row][new_col][0]
```

Функція move_back дозволяє вантажнику переміщуватися назад на одну клітинку в зворотному напрямку. Спочатку перевіряється параметр param для визначення, чи дозволено виконувати рух. Далі координати кроку інвертуються, обчислюються нові координати для руху вантажника назад. Цей фрагмент є частиною механізму для реалізації зворотних рухів вантажника в грі, де необхідно контролювати, коли і куди він може рухатися назад.

```
        if new_colornum == 0:
            man_colornum = 4
        elif new_colornum == 1:
            man_colornum = 6
        man_row = new_row
        man_col = new_col
        make_fragm(man_colornum, man_row, man_col)
        return 1
```

Оновлюють позиції вантажника на новій клітинці після його переміщення. Він враховує тип нової клітинки та забезпечує коректне відображення змін на ігровому полі.

```
root = Tk()
stl = ttk.Style()
clr_root = "lime"
dFont = font.Font(family="helvetica", size=14)
stl.configure('.', font=dFont, background=clr_root, foreground= "black")
btn_width = 8
pnl_top = Frame(root)
pnl_top.grid(row = 2, column = 0, columnspan = 5)
```

Створюється головне вікно програми, яке є базовим контейнером для всіх графічних елементів. Змінна root стає об’єктом класу Tk, який представляє це вікно. Ініціалізується стиль для елементів інтерфейсу за допомогою модуля ttk. stl – це об’єкт класу Style, який дозволяє налаштовувати вигляд віджетів. Задається колір

фону для основного вікна програми. Створюється об’єкт шрифту для використання в інтерфейсі. Колір фону встановлюється lime, колір тексту встановлюється чорний.

Встановлюється стандартна ширина кнопок, яка буде використовуватися в інтерфейсі. Це дозволяє забезпечити однаковий розмір кнопок у програмі. Створюється панель pnl_top, яка буде розміщуватися у головному вікні root. Frame використовується для групування віджетів у логічні секції. Панель розміщується в сітці:

```
canv = Canvas(pnl_top, width = canv_width, height = canv_height, background =
fon_color )
canv.pack()
```

Canvas - віджет у Tkinter, який дозволяє створювати графічні елементи, такі як лінії, прямокутники, овали, текст, зображення. У цьому випадку Canvas додається до панелі pnl_top, яка була створена раніше. Визначають розміри полотна: canv_width - ширина полотна, canv_height - висота полотна. Встановлюють колір фону полотна. Значення fon_color було визначено як lightgray. Використовується метод pack, щоб розмістити полотно на панелі. В інтерфейсі з’явиться полотно, на якому будуть відображатися графічні елементи гри (клітинки та рух вантажника).

```
def fnc_load():
    global flg_step
    flg_step = 1
    try:
        level_num = var_level.get()
    except:
        level_num = -1
```

Функція fnc_load відповідає за завантаження рівня гри. Виконується спроба отримати номер рівня з об’єкта var_level, це об’єкт Tkinter, який містить номер вибраного рівня з текстового поля. Якщо отримати номер рівня не вдалося, встановлюється значення -1, що вказує на помилку або відсутність вибраного рівня.

Функція змінює стан змінної flg_step, вказуючи на початок процесу завантаження рівня. Далі вона намагається отримати номер рівня з var_level, якщо це не вдається, задає значення -1 для подальшої обробки.

```
    if level_num < 1:
        level_num = level_count + 1 + level_num
        if level_num < 1:
```

```
        level_num = 1
```

Обробляють номер рівня гри. Перевіряється, чи вибраний номер рівня є меншим за 1. Це може відбуватися, якщо користувач ввів некоректне значення (від'ємне число або 0), або якщо рівень не був вибраний. Якщо номер рівня менший за 1, обчислюється нове значення для level_num. level_count - загальна кількість рівнів у грі.

```
    if level_num >=  level_count:
        level_num =  level_count
```

Перевіряється, чи значення level_num номер рівня перевищує або дорівнює загальній кількості рівнів level_count. level_count - змінна, яка зберігає загальну кількість доступних рівнів у грі. Якщо номер рівня перевищує максимальну кількість рівнів або дорівнює їй, номер рівня встановлюється на максимальний доступний рівень level_count. Це гарантує, що номер рівня не виходитиме за межі допустимого діапазону.

```
btn_load = ttk.Button( root, text = "загрузити", width = btn_width,
command = fnc_load, takefocus = 0)
btn_load.grid(row = 1, column = 0, sticky=E+N, pady = 5 , padx = 5)
```

Створюється кнопка за допомогою віджета Button з модуля ttk. Параметри: root: батьківський елемент, у якому розміщується кнопка (головне вікно програми). text=load: текст, який буде відображений на кнопці. width=btn_width: ширина кнопки. btn_width задається раніше в програмі. command=fnc_load: функція, що викликається при натисканні кнопки. У цьому випадку це fnc_load, яка відповідає за загрузку рівня. Розташовують кнопку на головному вікні за допомогою менеджера компоновки grid.

Після створення кнопки вона автоматично відображається в головному вікні root. При натисканні кнопки викликається функція fnc_load, яка відповідає за завантаження вибраного рівня, оновлення ігрового інтерфейсу.

```
edt_level = ttk.Entry(root, width = 5, textvariable= var_level, justify = RIGHT, font
= dFont )
edt_level.grid(row = 1, column = 1, padx= 5, pady = 5)
```

Створюється текстове поле для введення даних з такими параметрами:

- root: батьківське вікно, в якому буде розміщене текстове поле;
- width=5: ширина текстового поля, вимірюється в кількості символів, які можуть вміститися в полі вводу;
- textvariable=var_level: прив’язує текстове поле до змінної var_level. Це дозволяє зчитувати значення, введене в текстовому полі, і використовувати його в програмі.
- justify=RIGHT: вирівнювання тексту у полі вводу по правому краю.

Розміщують текстове поле на головному вікні за допомогою менеджера компоновки grid. Текстове поле дозволяє користувачу вводити числове значення, яке буде відображене в змінній var_level. Оскільки текстове поле прив’язане до var_level, будь-яка зміна значення в текстовому полі автоматично оновить var_level, і навпаки - зміна значення var_level також оновить вміст текстового поля.

```
def fnc_write():
    global flg_step
    flg_step = 1
    file_name = str(var_level.get())+".log"
    fhn = open(file_name, "w")
    lst = list(lbx_log.get( 0, END))
    lst = [line for line in lst if line[-1] != "0"]
    lst.append("")
    fhn.write('\n'.join(lst))
    fhn.close()
```

Функція fnc_write виконує запис в лог-файл. Отримується значення змінної var_level (яка зберігає поточний рівень) і перетворюється на рядок. Це значення використовується для створення імені файлу, в який буде записано логи, з додаванням розширення .log. Наприклад, якщо var_level містить значення 3, то ім’я файлу буде 3.log. Відкривається файл для запису. Отримуються всі елементи з віджету lbx_log (список). Метод get() отримує всі елементи від початку (індекс 0) до кінця (індекс END) елементів у списку lbx_log. Результат зберігається в змінній lst як список рядків.

Функція fnc_write отримує поточний рівень гри з var_level. Визначається ім’я файлу для збереження логу (наприклад, 3.log). Лог записується в файл, причому

фільтруються рядки, що закінчуються на 0. Вміст логу зберігається у файл, після чого файл закривається.

```
btn_write = ttk.Button( root, text = "записати", width = btn_width,
command = fnc_write, takefocus = 0)
btn_write.grid(row = 1, column = 2, sticky=E+N, pady = 5 , padx = 5)
```

Створюють кнопку за допомогою віджету ttk.Button з бібліотеки ttk:

- root: батьківське вікно для цієї кнопки;
- text="записати": текст, що буде відображатися на кнопці. В даному випадку це записати (для збереження даних або запису в лог).
- width=btn_width: ширина кнопки.
- command=fnc_write: при натисканні на кнопку буде викликана функція fnc_write, яка виконує запис у лог або інші операції.

Кнопка btn_write з'являється на інтерфейсі користувача. При натисканні на неї викликається функція fnc_write, яка відповідає за запис логів у файл. Кнопка має текст записати, що інформує користувача про її функцію. Користувач натискає на неї, всі необхідні дані зберігаються в файлі для подальшого використання або аналізу.

```
def fnc_read():
    global flg_step
    flg_step = 0
    lbx_log.delete( 0, END)
    file_name = str(var_level.get())+".log"
    try:
        fhn = open(file_name )
        lst = fhn.readlines()
        fhn.close()
        for str_loc in lst:
            lbx_log.insert(END, str_loc.strip())
        lbx_log.selection_set( END )
        lbx_log.see(END)
    except:
        pass
```

Функція fnc_read() призначена для зчитування даних із файлу, ім'я якого залежить від поточного рівня або іншої змінної. Функція намагається відкрити файл, зчитати його вміст та вставити його рядки в lbx_log (список логів), щоб показати

користувачеві. У разі помилки (якщо файл не існує або сталася інша проблема) помилка ігнорується завдяки блоку except.

```
btn_read = ttk.Button( root, text = "read", width = btn_width, command = fnc_read,
takefocus = 0)
btn_read.grid(row = 1, column = 3, sticky=E+N, pady = 5 , padx = 5)
```

При натисканні на кнопку прочитати буде викликана функція fnc_read, яка зчитує логи або дані з файлу та відображає їх у відповідному віджеті Listbox. Кнопка виглядає як стандартна кнопка з написом прочитати.

```
def fnc_run():
    global flg_step
    flg_step = 1
    lst = list(lbx_log.get( 0, END))
    for index, key in enumerate(lst):
        if not flg_step:
            lst = []
            break
        lbx_log.select_clear(0, END)
        lbx_log.selection_set( index )
        lbx_log.see(index)
        if key[-1] in ["0","1","2"]:
            key = key[ : -1]
        if key in dct_func.keys():
            func = dct_func[key]
            func( )
            canv.update()
        time.sleep( step_time )
```

Функція fnc_run() обробляє команди з журналу lbx_log і виконує відповідні дії, які записані в словник з функціями dct_func (). Команди з журналу обробляються по черзі, для кожної команди викликається відповідна функція. Якщо команда передбачає певну дію на Canvas, то він оновлюється після виконання команди.

```
btn_run = ttk.Button( root, text = "run", width = btn_width, command = fnc_run,
takefocus = 0)
btn_run.grid(row = 1, column = 4, sticky=E+N, pady = 5 , padx = 5)
pnl_log = Frame(root)
pnl_log.grid(row = 0, column = 6, rowspan = 3, sticky=N, pady = 5 , padx = 5)
```

Створюється кнопка run в контейнері root (головне вікно). При натисканні кнопки викликається функція fnc_run, яка запускає виконання логічних кроків (рух

вантажника чи обробка дій). Створюється контейнер Frame в основному вікні root для відображення елементів, які будуть використовуватися для журналу або логів.

Користувач натискає на кнопку виконати, виконується функція fnc_run(), яка обробляє кроки гри чи системи, відображаючи зміни на екрані. Панель pnl_log створюється для розміщення віджетів, для журналу, логів чи інших елементів відображення інформації (список дій чи повідомлень).

```
lbx_log = Listbox(pnl_log, width = 10, height = 27, font = dFont,
selectmode = SINGLE)
lbx_log.pack(side="left", fill="y")
```

Створюють віджет Listbox в панелі pnl_log. Це елемент інтерфейсу для відображення списку елементів. У цьому випадку pnl_log - контейнер, на якому буде розміщено цей список.

- width=10: ширина списку визначена як 10 символів.
- height=27: висота списку 27 рядків, що означає, що одночасно будуть відображатися 27 елементів списку.
- selectmode=SINGLE: цей параметр визначає, що можна вибрати тільки один елемент зі списку. Вибір більше ніж одного елемента одночасно не дозволяється.

Віджет Listbox надає користувачеві можливість переглядати список елементів і вибирати один із них. У цьому випадку список буде розміщений в панелі pnl_log. Список займає всю висоту панелі, дозволяючи прокручувати елементи списку, якщо їх більше, ніж вміщує панель. Для додавання можливості прокручування списку можна використовувати scrollbar, щоб користувач міг переглядати список, коли його висота перевищує видимий розмір.

```
sbr_log = Scrollbar(pnl_log, orient="vertical")
sbr_log.pack(side="right", fill="y")
sbr_log.config(command = lbx_log.yview)
lbx_log.config(yscrollcommand = sbr_log.set)
```

Створюється віджет Scrollbar в панелі pnl_log, що дозволяє користувачеві прокручувати вміст списку. Скроллбар буде контролювати прокручування вмісту lbx_log. orient="vertical": вказує, що скроллбар буде вертикальним. Це означає, що скроллбар дозволить прокручувати список по вертикалі, якщо його вміст не

вміщується в межах висоти панелі. Скроллбар буде розміщений з правого боку панелі pnl_log. Скроллбар заповнюватиме висоту панелі pnl_log, дозволяючи користувачу прокручувати весь список вертикально. Коли елементи списку не вміщуються в області відображення, скроллбар забезпечує користувачу можливість переглядати інші елементи.

```
for row_num in range(row_count):
    row = []
    for col_num in range(col_count):
        x_beg = col_num * cell_size;  x_end = x_beg + cell_size
        y_beg = row_num * cell_size;  y_end = y_beg + cell_size
        rect = canv.create_rectangle(x_beg+1, y_beg+1, x_end-1, y_end-1, fill =
fon_color, outline = fon_color, width = 2 )
        row.append([0,rect])
    cells.append(row)
```

Створюється ігрове поле у вигляді матриці клітинок, де кожна клітинка представлена прямокутником. Кожен прямокутник можна використовувати для відображення різних елементів гри (вантажника, стінки чи ящика)..

```
file_name = "sokoban.bin"
fh = open( file_name, "rb+")
level_count = fh.seek(0,2) // 400
```

Файл sokoban.bin відкривається для читання та запису. Розмір файлу визначається за допомогою seek(), а потім цей розмір ділиться на 400, що дає кількість рівнів, збережених у файлі sokoban.bin. Якщо кожен рівень займає 400 байтів, то весь файл буде містити кілька рівнів гри, це значення можна використовувати для визначення кількості доступних рівнів у грі.

```
def shift_down( ):
    res = move_man( 1, 0)
    return res
```

Функція shift_down перемішає персонажа вниз по ігровому полі. Всередині функції викликається функція move_man(1, 0). Параметри 1, 0 вказують, що персонаж повинен переміститися:

1 - на 1 клітинку вниз по вертикалі (рядок збільшується).

0 - по горизонталі без змін (стовпець не змінюється).

Функція move_man повертає результат, який зберігається в змінній res. Це може бути 0 (якщо рух неможливий або є перешкода), або 1 (якщо рух успішний). Функція shift_down є абстракцією для руху персонажа вниз. Вона викликає основну логіку руху через функцію move_man, передаючи їй параметри для переміщення вниз. Це дозволяє зручно керувати рухами в грі без необхідності щоразу викликати функцію move_man з тими ж параметрами.

```
def shift_up( ):
    res =  move_man( -1, 0)
    return res
```

Функція shift_up викликає основну функцію для переміщення персонажа move_man, передаючи їй аргументи для переміщення вгору. Це спрощує код, даючи змогу використовувати одну і ту саму логіку для переміщення в різних напрямках, лише змінюючи параметри напрямку в кожній функції.

```
def shift_right( ):
    res =  move_man( 0,1)
    return res
```

Функція shift_right призначена для руху вантажника вправо. Вона використовує основну логіку руху через функцію move_man, передаючи їй параметри для переміщення вправо.

```
def shift_left():
    res =  move_man( 0,-1)
    return res
```

Функція shift_left викликає основну функцію для переміщення персонажа move_man, передаючи їй аргументи для переміщення вліво.

```
def step_back():
    global cells, man_row, man_col
    param = 0
    cmd = " "
    step_row = 0; step_col = 0
    if lbx_log.size() == 0:  return 0
```

Функція step_back готується до виконання кроку назад у грі, але наразі вона лише перевіряє, чи є команди для виконання. Наступні кроки в функції будуть обробляти історію дій та виконувати зворотній рух або відкат дії, використовуючи дані з lbx_log.

```
    while  lbx_log.size() >0:
```

```
        last_key = lbx_log.get( END )
        lbx_log.delete( END )
        param = last_key[-1]
        cmd = last_key[:-1]
        if param in ["1","2"]:
            break
```

Цей фрагмент коду дозволяє відкотити гру назад, переглядаючи і обробляючи останні дії з lbx_log. Функція по черзі витягує останні команди з логів і використовує їх для визначення, чи можна відкатити поточний стан гри. Якщо виявлено, що параметр дії є 1 або 2, цикл припиняється, подальші дії вже не виконуються.

```
    if cmd == "Up":
        step_row = -1
    elif cmd == "Down":
        step_row = +1
    elif cmd == "Left":
        step_col = -1
    elif cmd == "Right":
        step_col = +1
    res = move_back(step_row, step_col, param)
    return res
```

Цей фрагмент коду дозволяє відкотити рух персонажа в одному з чотирьох напрямків (вгору, вниз, вліво, вправо), використовуючи інформацію з логів. Завдяки цьому гравець може повернутись назад на один крок, і при цьому враховувати правильний напрямок руху за допомогою змінних step_row та step_col.

```
dct_func = {"Left":shift_left, "Right":shift_right,
                "Up":shift_up, "Down":shift_down, "z":step_back }
```

Створюють словник під назвою dct_func, в якому зберігаються пари ключ-значення. У цьому випадку ключі - це рядки, що представляють команди або кнопки, а значення - функції, які будуть викликані при натисканні цих команд. Коли користувач натискає кнопку Up, у словнику dct_func шукається ключ Up, відповідна функція shift_up викликається для виконання певного руху.

```
def key_hndl(event):
    global key
    key = event.keysym
    if not key in dct_func.keys():
        return
```

```
    func = dct_func[key]
    res = func( )
    if key != "z":
        lbx_log.insert( END, key + str(res) )
        lbx_log.select_clear(0, END)
        lbx_log.selection_set( END )
        lbx_log.see(END)
    return
```

Функція key_hndl реалізує обробку натискання клавіші, перевіряючи, чи є вона в словнику dct_func. Якщо є, викликається відповідна функція, результат виконання записується в лог. Це дозволяє відслідковувати дії користувача і забезпечує взаємодію з грою.

```
root.bind("<Key>", key_hndl)
root.resizable(width=False, height=False)
root.mainloop()
fh.close()
```

Це прив’язка обробника подій клавіатури до основного вікна root. Функція key_hndl буде викликатися кожного разу, коли користувач натискає будь-яку клавішу на клавіатурі. Вона отримує подію як параметр і обробляє натискання клавіші відповідно до визначених функцій у словнику dct_func. Метод root.resizable забороняє змінювати розміри вікна користувачем. Параметри width=False та height=False означають, що вікно не можна змінювати по ширині та висоті. Метод root.mainloop() запускає головний цикл обробки подій Tkinter. Цей цикл триватиме до того часу, поки користувач не закриє вікно.

3.3. Створення нових рівнів у грі “Сокобан”

У додатку Б представлено програмний код редактора рівнів для гри Сокобан, написаний на Python з використанням бібліотеки Tkinter для створення графічного інтерфейсу.

Параметри та змінні:

- row_count та col_count визначають розмір ігрового поля 20x20 клітинок.
- cell_size визначає розмір кожної клітинки на полі.

- cells - двовимірний список, який моделює ігрове поле. Кожен елемент списку містить номер кольору та прямокутник, який відображається на ігровому полі.

Функції:

- make_fragm(color_num, row_num, col_num) створює фрагмент на полі з заданим кольором у вказаній клітинці;
- write_cells() записує поточний стан поля у файл у вигляді байтів;
- read_cells() зчитує стан поля з файлу та відображає його на ігровому полі.

Після цього проектують графічний інтерфейс редактора рівнів. Для цього створюється головне вікно root за допомогою методу Tk(). На вікні розміщується канва, на якій відображається ігрове поле. Кнопки та радіокнопки дозволяють вибирати колір, завантажувати, зберігати та додавати рівні.

Обробка подій:

- mouse_click(event) обробляє клік миші, змінюючи колір клітинки на полі.
- mouse_move(event) обробляє рух миші з затиснутою лівою кнопкою, дозволяючи малювати на полі.

Файл sokoban.bin використовується для зберігання рівнів. Кожен рівень займає 400 байт (20x20 клітинок). Кожен рівень зберігається у цьому файлі у вигляді послідовності байтів. Кожен байт відповідає одній клітинці на полі.

Щоб створити новий рівень, потрібно запустити редактор рівнів. Відкриється графічний інтерфейс цього редактора з ігровим полем 20x20 клітинок.

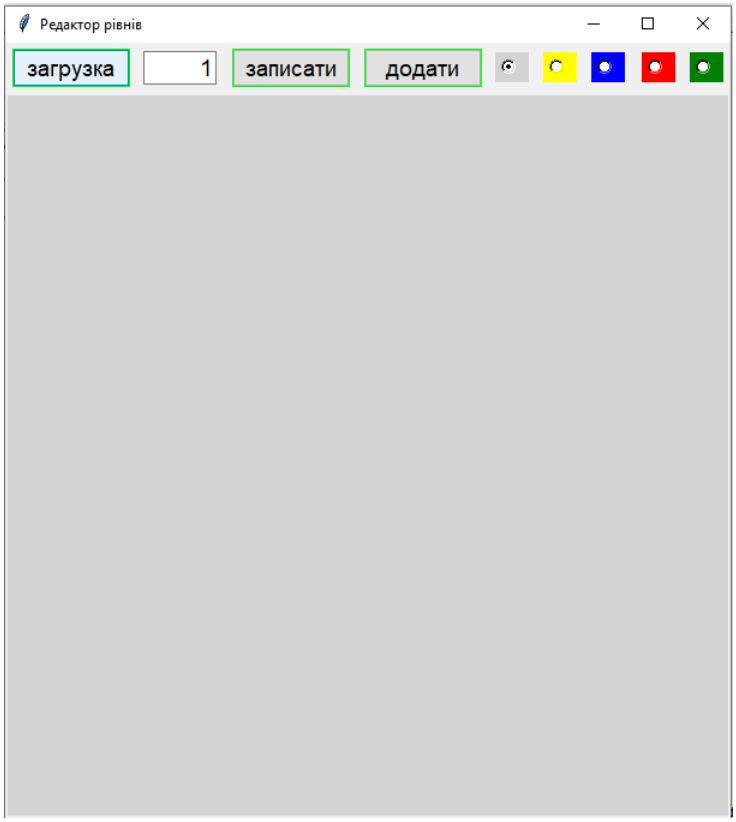

Рисунок 3.4 – Графічний редактор редактора рівнів

Ігрові клітинки можуть мати різні кольори, кожна з яких відповідає певному типу об’єкта в грі. Для цього є радіокнопки, що дозволяють вибрати колір для клітинки. Кольори відповідатимуть наступним об’єктам:

- 0 світло-сірий – фон;
- 1 зелений - стіни або перешкоди;
- 2 синій – коробки на ігровому полі;
- 3 червоний - вантажник (гравець);
- 4 жовтий - цілі для коробок (місце, куди потрібно пересунути всі сині коробки).

Для створення нового рівня потрібно спочатку вибрати колір для розміщення даних об’єктів на ігровому полі. Далі треба натискати лівою кнопкою миші на клітинки поля, щоб змінити їхній колір згідно з вибором. Можна змінювати колір кожної клітинки по черзі. Залежно від вибраного кольору клітинки будуть представляти різні об’єкти на полі (стіни, вантажника або коробки). Якщо потрібно змінити колір кількох клітинок, треба натиснути ліву кнопку миші і переміщати її по полю. Таким чином можна заповнити кілька клітинок одним кольором.

Коли рівень буде готовий, потрібно зберегти його в файл. Для цього треба ввести номер рівня в текстове поле. Це буде номер рівня, який потрібно створити чи замінити. Треба натиснути кнопку записати, щоб добавити поточний рівень у файл. Якщо рівень з номером, який ввели в текстове поле, вже існує, він буде перезаписаний. Якщо потрібно додати новий рівень до файлу без заміни існуючих, треба натиснути кнопку додати. Це додасть новий рівень в кінець файлу.

Збереження рівнів у файл виконується через бінарний формат (файл sokoban.bin), тому збережені рівні будуть містити інформацію про кольори клітинок у вигляді байтів.

Рівень не можна створити, не використовуючи правильні кольори, адже кожен колір має своє значення у грі (стіни, вантажник, коробки. Можна створювати різні рівні, змінюючи їх структуру, наприклад, змінюючи розташування стін чи коробок.

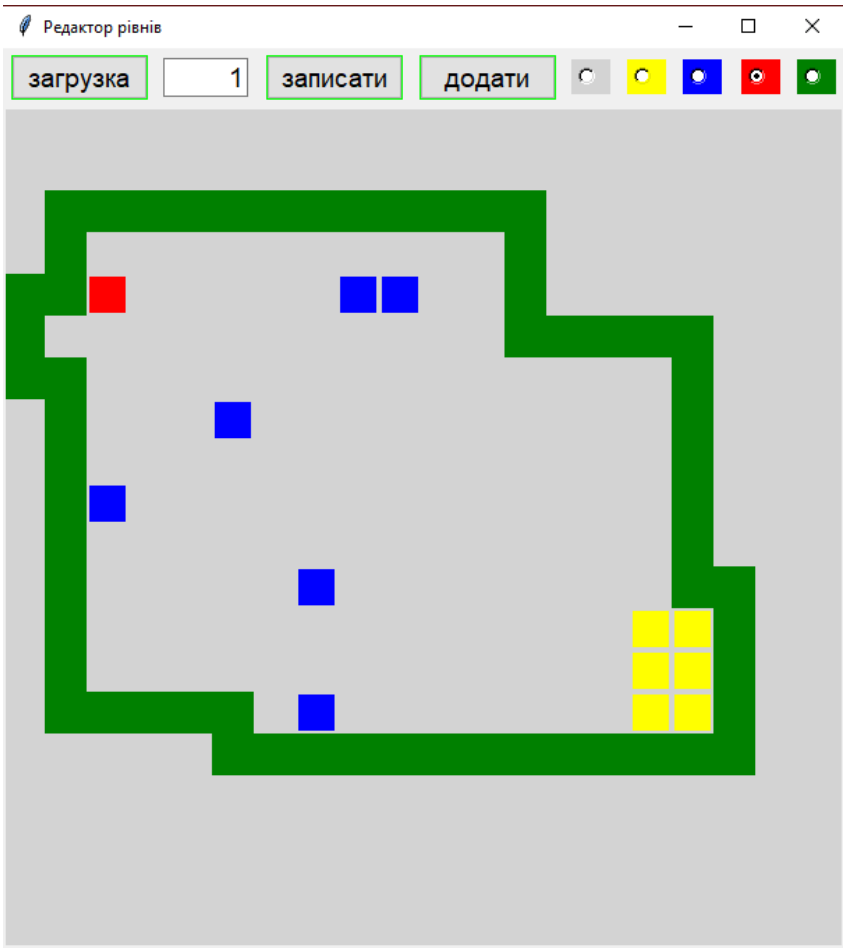

Рисунок 3.5 – Новий спроектований ігровий рівень

Якщо треба переглянути або редагувати вже існуючий рівень, вводять його номер в текстове поле. Після цього слід натиснути кнопку загрузка, щоб завантажити рівень з файлу і відобразити його на ігровому полі.

3.4. Запис ходів в грі

Програма реалізує гру “Сокобан” з функцією запису і відтворення ходів, що дозволяє зберігати та повторно виконувати кроки гравця. Модуль time призначений для обробки часу. Для завантаження поточного рівня гри з файлу та відображення його на ігровому полі служить функція read_cells. Функція move_man відповідає за переміщення вантажника, перевіряючи можливість переміщення на сусідні клітинки та проштовхування коробок. Важливою частиною є функція для запису ходів fnc_write, яка зберігає команди в файл логів, щоб гравець міг відтворити свій прогрес. Лог ходів зберігається у вигляді тексту, де кожен хід додається до списку в віджеті Listbox в інтерфейсі гри.

Функція fnc_run використовується для відтворення всіх записаних ходів. Вона читає лог і по черзі виконує кожен запис, що відповідає певному кроку. Усі ходи записуються в форматі тексту, де кожен хід відображається як клавіша, яка була натиснена гравцем (це напрямки Left, Right, Up, Down).

Кожен хід записується у список за допомогою методу lbx_log.insert(), після цього його можна зберегти у файл через кнопку **write**. Програма має функцію **read**, яка завантажує вже збережений лог з файлу і відображає його в списку, дозволяючи гравцю переглянути свої попередні ходи.

При натисканні на кнопки для управління вантажником, таких як load, write, read гравець може управляти логами, зберігати нові чи відтворювати старі записи. Кожен хід зберігається в окремому файлі, який називається за номером рівня, наприклад: 1.log, 2.log.

Кожен запис в логу містить текстову команду, яка відповідає конкретному кроку в грі, наприклад, Left для руху вліво або Right для руху вправо. Після натискання клавіші, що відповідає певному руху, програма додає цей хід до логу за допомогою команди lbx_log.insert(). Всі функції управління клавіатурою (вперед,

назад, вліво, вправо) визначені у словнику dct_func, де кожній клавіші відповідає певна функція.

Відтворення логів здійснюється за допомогою функції fnc_run, яка читає лог і за допомогою затримки, використовуючи метод time.sleep()), по черзі виконує команди. Якщо гравець хоче переглянути вже збережений лог, він може натиснути кнопку **read**, щоб побачити усі виконані ходи. Завдяки цьому механізму гравець може не тільки записати свій прогрес, а й повернутися до нього пізніше, відтворивши всі попередні кроки. Весь процес записи та відтворення ходів реалізовано через запис текстових команд, що дозволяє гравцю відтворювати свою гру або просто зберігати її для подальшого перегляду.

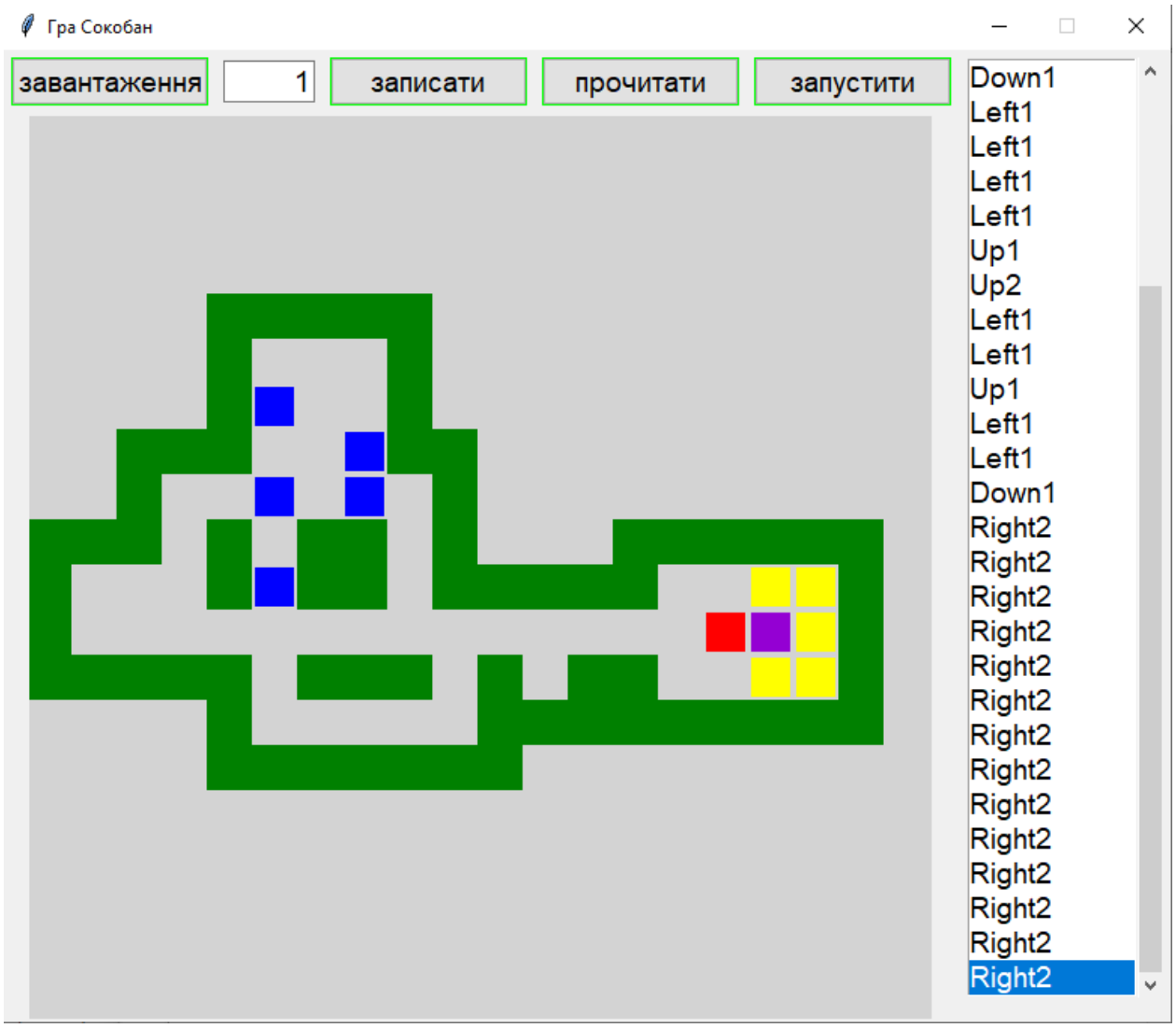

Рисунок 3.6 – Запис ходів у грі

3.5. Характеристики апаратного та програмного забезпечення

Для проектування інформаційної системи використовувався комп’ютер з наступними характеристиками.

Таблиця 3.1 – Характеристики персонального комп’ютера

комплектуюча	**фірма**	**характеристики**
процесор	AMD	3.4 GHz, 4 ядра
материнська плата	Gigabyte	Z490, DDR4
операційна пам’ять	Kingston	DDR4 8 ГБ
відеокарта	NVIDIA	RTX 3070, 8 GB
жорсткий диск	Seagate	1 TB SSD
монітор	Samsung	27"

В дипломній роботі використано такі програмні продукти.

Таблиця 3.2 – Програмне забезпечення

Операційна система	Windows 11
Середовище розробки	Python, Tkinter, PyCharm
Растровий графічний редактор	Adobe Photoshop 2024
Векторний графічний редактор	CorelDraw 2022
Текстовий редактор	Microsoft Word 2016

ВИСНОВКИ

У дипломній роботі розглянуто основні етапи розробки гри “Сокобан” із використанням мови програмування Python та бібліотеки Tkinter. Проаналізовано основи створення графічного інтерфейсу користувача, що дозволило реалізувати зручний інтерфейс гри. Розроблено математичну модель ігрового простору, яка включає представлення ігрового поля, об’єктів і персонажа у вигляді матриці.

Описано алгоритми переміщення персонажа, що забезпечують коректність гри відповідно до правил. Здійснено інтеграцію механізмів перевірки кінцевого стану гри, що дозволяють визначати перемогу або неможливість подальших дій. Використання бібліотеки Tkinter забезпечило високу швидкість розробки графічного інтерфейсу та взаємодії з ігровими елементами. Гра підтримує функції завантаження та збереження рівнів, що дозволяє зберігати прогрес користувача. Реалізовано можливість відкату останнього ходу, що значно підвищує зручність для гравця.

Використання структури даних типу список для логування дій забезпечило ефективність обробки дій користувача. У процесі роботи вдалося поєднати простоту реалізації та функціональність ігрового процесу.

Тестування програми показало її стабільність та відповідність заявленим функціональним вимогам. Гра адаптована для різних рівнів складності, що робить її привабливою для широкої аудиторії користувачів. Використання Python забезпечило кросплатформеність гри, що дозволяє запускати її на різних операційних системах. Розробка гри демонструє практичне застосування об’єктно-орієнтованого підходу у програмуванні. Створення гри дозволило закріпити теоретичні знання у сфері алгоритмів та програмної інженерії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Junghanns A., Schaeffer J. Sokoban: Enhancing general single-agent search methods using domain knowledge. Artif. Intell., 129(1-2), 2001. P. 219-251.

2. Idris I. Instant pygame for python game development how-to. 2013. 76 p.

3. Mayer C. Brain games Python. Leanpub, 2021. 195 p.

4. Rodas de Paz A., Howse J. Python game programming by example. Packt Publishing, 2015. 230 p.

5. French M. Python programming workbook for game development: an essential beginners guide to learn how to code game with pygame. Independently published, 2024. 354 p.

6. McGugan W. Beginning game development with python and pygame from novice to professional. Apress, 2007. 316 p.

7. Jackson M. Python game development: creating interactive games with python and pygame library. Independently published, 2024. 281 p.

8. Turing A. Python Game Logic. Publifye, 2025. 286 p.

9. Kafle Sachin. Learning Python by Building Games: A beginner’s guide to Python programming and game development. Packt Publishing, 2019. 496 p.

10. Sweigart Al. Making games with Python & Pygame. CreateSpace independent publishing platform, 2012. 366 p.

11. Kalb I. Object-oriented Python: master OOP by building games and GUI. No Starch Press, 2022. 416 p.

12. McManus S. Mission Python: code a space adventure game. No starch press, 2018. 280 p.

13. Harrison K., McGugan W. Beginning Python games development: with PyGame. Apress, 2nd Edition, 2015. 308p.

14. Hussain F., Hussain K. Object-oriented Python: master OOP through game development and gui applications. Sonar Publishing, 2024. 571 p.

15. Tracey S.M. Magpi essentials make games with Python. New York: Raspberry Pi Ltd., 2015. 154 p.

16. Moore A. Python GUI programming with Tkinter: develop responsive and powerful GUI applications with Tkinter. Packt Publishing. 2021. 665 p.

17. Chandrakar S., Bahadure N. B. Building modern GUIs with tkinter and Python: Building user-friendly GUI applications with ease building modern GUIs with tkinter and Python. BPB Online. 2023. 365 p.

18. Bhaskar C. Tkinter GUI application development blueprints. Packt Publishing. 452 p.

ДОДАТКИ

ДОДАТОК А

game.py

```
from tkinter import *
from tkinter import ttk
from tkinter import font

import inspect
import time

row_beg = -1
col_beg = -1
row_count = 20
col_count = 20
man_row = 0
man_col = 0

cell_size = 30
canv_width = cell_size * col_count
canv_height = cell_size * row_count

cells = []

fon_color = "lightgray"
colors = [fon_color, "yellow", "blue", "darkviolet", "red", "green", "chocolate"]

step_time = 0.2
flg_step = 0

def print_funcname():
    print(inspect.stack()[1][3])

def make_fragm(color_num, row_num, col_num):
    global cells

    cells[row_num][col_num][0] = color_num
    color = colors[ color_num]
    rect = cells[row_num][col_num][1]

    if color_num in [1,2,3,4,6]:
```

```
        bg_color = fon_color
    else:
        bg_color = color
    canv.itemconfig(rect, fill = color, outline = bg_color)


def read_cells():
    global man_row, man_col
    lst_loc = list(fh.read(400))
    for num in range(400):
        row_num = num // 20
        col_num = num % 20
        color_num = lst_loc[num]
        make_fragm(color_num, row_num, col_num)
        if color_num == 4:
            man_row = row_num
            man_col = col_num
    man_colornum = cells[man_row][man_col][0]

def move_man(step_row, step_col):
    global cells, man_row, man_col

    man_colornum = cells[man_row][man_col][0]
    new_row = man_row + step_row
    new_col = man_col + step_col
    new_colornum = cells[new_row][new_col][0]

    if new_colornum == 5:
        return 0

    if  new_colornum in [2,3]:
        next_row = new_row + step_row
        next_col = new_col + step_col
        next_colornum = cells[next_row][next_col][0]
        if next_colornum in[2,3,5]:
            return 0

        if man_colornum == 4:
            color_num = 0
        else:
            color_num = 1
        make_fragm(color_num, man_row, man_col)
```

```
            if new_colornum == 2:
                man_colornum = 4
            else:
                man_colornum = 6
            man_row = new_row
            man_col = new_col
            make_fragm(man_colornum, man_row, man_col)

            if next_colornum == 0:
                color_num = 2
            else:
                color_num = 3
            make_fragm(color_num, next_row, next_col)

            return 2

        if man_colornum == 4:
            color_num = 0
        else:
            color_num = 1
        make_fragm(color_num, man_row, man_col)

        if new_colornum == 0:
            man_colornum = 4
        elif new_colornum == 1:
            man_colornum = 6
        man_row = new_row
        man_col = new_col
        make_fragm(man_colornum, man_row, man_col)
        return 1

    def move_back(step_row, step_col, param):
        global cells, man_row, man_col

        if param not in ["1","2"]:  return 0

        step_row = - step_row;  step_col = - step_col
        man_colornum = cells[man_row][man_col][0]

        new_row = man_row + step_row
        new_col = man_col + step_col
        new_colornum = cells[new_row][new_col][0]
```

```
if param == "1":

    if man_colornum == 4:
        color_num = 0
    else:
        color_num = 1
    make_fragm(color_num, man_row, man_col)

    if new_colornum == 0:
        man_colornum = 4
    elif new_colornum == 1:
        man_colornum = 6
    man_row = new_row
    man_col = new_col
    make_fragm(man_colornum, man_row, man_col)
    return 1

if param == "2":
    box_row = man_row - step_row
    box_col = man_col - step_col
    box_colornum = cells[box_row][box_col][0]

    if box_colornum == 2:
        color_num = 0
    else:
        color_num = 1
    make_fragm(color_num, box_row, box_col)

    if man_colornum == 4:
        color_num = 2
    else:
        color_num = 3
    make_fragm(color_num, man_row, man_col)

    if new_colornum == 0:
        man_colornum = 4
    else:
        man_colornum = 6
    man_row = new_row
    man_col = new_col
    make_fragm(man_colornum, man_row, man_col)

    return 2
```

```
root = Tk()
root.title("Гра Сокобан")
stl = ttk.Style()
clr_root = "lime"
dFont = font.Font(family="helvetica", size=14)
stl.configure('.', font=dFont, background=clr_root, foreground= "black")

btn_width = 11
pnl_top = Frame(root)
pnl_top.grid(row = 2, column = 0, columnspan = 5)

canv = Canvas(pnl_top, width = canv_width, height = canv_height,
background = fon_color )
canv.pack()

def fnc_load():
    global flg_step
    flg_step = 1
    try:
        level_num = var_level.get()
    except:
        level_num = -1

    if level_num < 1:
        level_num = level_count + 1 + level_num
        if level_num < 1:
            level_num = 1

    if level_num >=  level_count:
        level_num =  level_count

    var_level.set(level_num)

    fh.seek( 400*(level_num-1) )
    read_cells()
    lbx_log.delete( 0, END )

btn_load = ttk.Button( root, text = "завантаження", width = btn_width,
command = fnc_load, takefocus = 0)
btn_load.grid(row = 1, column = 0, sticky=E+N, pady = 5 , padx = 5)

var_level = IntVar()
var_level.set(1)
```

```
edt_level = ttk.Entry(root, width = 5, textvariable= var_level, justify = RIGHT,
font = dFont)
edt_level.grid(row = 1, column = 1, padx= 5, pady = 5)

def fnc_write():
    global flg_step
    flg_step = 1
    file_name = str(var_level.get())+".log"
    fhn = open(file_name, "w")
    lst = list(lbx_log.get( 0, END))
    lst = [line for line in lst if line[-1] != "0"]
    lst.append("")
    fhn.write('\n'.join(lst))
    fhn.close()

btn_write = ttk.Button( root, text = "записати", width = btn_width,
command = fnc_write, takefocus = 0)
btn_write.grid(row = 1, column = 2, sticky=E+N, pady = 5 , padx = 5)

def fnc_read():
    global flg_step
    flg_step = 0
    lbx_log.delete( 0, END)

    file_name = str(var_level.get())+".log"
    try:
        fhn = open(file_name)
        lst = fhn.readlines()
        fhn.close()
        for str_loc in lst:
            lbx_log.insert(END, str_loc.strip())
        lbx_log.selection_set(END)
        lbx_log.see(END)
    except:
        pass

btn_read = ttk.Button( root, text = "прочитати", width = btn_width,
command = fnc_read, takefocus = 0)
btn_read.grid(row = 1, column = 3, sticky=E+N, pady = 5 , padx = 5)

def fnc_run():
    global flg_step
    flg_step = 1
```

```
    lst = list(lbx_log.get( 0, END))

    for index, key in enumerate(lst):
        if not flg_step:
            lst = []
            break

        lbx_log.select_clear(0, END)
        lbx_log.selection_set( index )
        lbx_log.see(index)

        if key[-1] in ["0","1","2"]:
            key = key[ : -1]

        if key in dct_func.keys():
            func = dct_func[key]
            func( )
            canv.update()

        time.sleep( step_time )

btn_run = ttk.Button( root, text = "запустити", width = btn_width, command = fnc_run,
takefocus = 0)
btn_run.grid(row = 1, column = 4, sticky=E+N, pady = 5 , padx = 5)

pnl_log = Frame(root)
pnl_log.grid(row = 0, column = 6, rowspan = 3, sticky=N, pady = 5 , padx = 5)

lbx_log = Listbox(pnl_log, width = 10, height = 27, font = dFont,
selectmode = SINGLE)
lbx_log.pack(side="left", fill="y")

sbr_log = Scrollbar(pnl_log, orient="vertical")
sbr_log.pack(side="right", fill="y")

sbr_log.config(command = lbx_log.yview)
lbx_log.config(yscrollcommand = sbr_log.set)

for row_num in range(row_count):
    row = []
    for col_num in range(col_count):
        x_beg = col_num * cell_size;  x_end = x_beg + cell_size
        y_beg = row_num * cell_size;  y_end = y_beg + cell_size
```

```
        rect = canv.create_rectangle(x_beg+1, y_beg+1, x_end-1, y_end-1,
        fill = fon_color, outline = fon_color, width = 2 )
        row.append([0,rect])
    cells.append(row)

file_name = "sokoban.bin"
fh = open( file_name, "rb+")
level_count = fh.seek(0,2) // 400

def shift_down( ):
    res = move_man(1, 0)
    return res

def shift_up( ):
    res =  move_man(-1, 0)
    return res

def shift_right( ):
    res =  move_man(0,1)
    return res

def shift_left():
    res =  move_man(0,-1)
    return res

def step_back():
    global cells, man_row, man_col
    param = 0
    cmd = " "
    step_row = 0; step_col = 0
    if lbx_log.size() == 0:  return 0

    while  lbx_log.size() >0:
        last_key = lbx_log.get(END)
        lbx_log.delete(END)
        param = last_key[-1]
        cmd = last_key[:-1]
        if param in ["1","2"]:
            break

    if cmd == "Up":
        step_row = -1
    elif cmd == "Down":
```

```
        step_row = +1
    elif cmd == "Left":
        step_col = -1
    elif cmd == "Right":
        step_col = +1

    res = move_back(step_row, step_col, param)
    return res

dct_func = {"Left":shift_left, "Right":shift_right,
                "Up":shift_up, "Down":shift_down, "z":step_back}

def key_hndl(event):
    global key
    key = event.keysym

    if not key in dct_func.keys():
        return

    func = dct_func[key]
    res = func( )

    if key != "z":
        lbx_log.insert(END, key + str(res))
        lbx_log.select_clear(0, END)
        lbx_log.selection_set(END)
        lbx_log.see(END)
    return

root.bind("<Key>", key_hndl)

root.resizable(width=False, height=False)
root.mainloop()
fh.close()
```

ДОДАТОК Б

level.py

```
from tkinter import *
from tkinter import ttk
from tkinter import font

import inspect

row_beg = -1
col_beg = -1
row_count = 20
col_count = 20

cell_size = 30
canv_width = cell_size * col_count
canv_height = cell_size * row_count

cells = []

fon_color = "lightgray"
colors = [fon_color, "yellow", "blue", "lime", "red", "green"]

def print_funcname():
    print(inspect.stack()[1][3])

def make_fragm(color_num, row_num, col_num):
    global cells
    cells[row_num][col_num][0] = color_num
    color = colors[ color_num]
    rect = cells[row_num][col_num][1]
    if color_num in [1,2,3,4]:
        bg_color = fon_color
    else:
        bg_color = color
    canv.itemconfig(rect, fill = color, outline = bg_color)

def write_cells():
    lst_loc = [0 for num in range(400)]
    for num in range(400):
        row_num = num // 20
        col_num = num % 20
```

```
            lst_loc[num] = cells[row_num] [col_num][0]
        fh.write( bytes(lst_loc) )

    def read_cells():
        lst_loc = list( fh.read(400) )
        for num in range(400):
            row_num = num // 20
            col_num = num % 20
            color_num = lst_loc[num]
            make_fragm(color_num, row_num, col_num)

    root = Tk()
    root.title("Редактор рівнів")
    stl = ttk.Style()
    clr_root = "lime"
    dFont = font.Font(family="helvetica", size=14)
    stl.configure('.', font=dFont, background=clr_root, foreground= "black")

    btn_width = 11
    pnl_top = Frame(root)
    pnl_top.grid(row = 2, column = 0, columnspan = 10)

    canv = Canvas(pnl_top, width = canv_width, height = canv_height,
    background = fon_color)
    canv.pack()

    def mouse_click(event):
        row_num = event.y //cell_size
        col_num = event.x //cell_size
        color_num = var_color.get()
        make_fragm(color_num, row_num, col_num)

    canv.bind("<Button-1>", mouse_click)

    def mouse_move(event):
        global row_beg, col_beg
        row_num = event.y //cell_size
        col_num = event.x //cell_size
        color_num = var_color.get()
        if row_num == row_beg and col_num == col_beg and cells[ row_num][col_num][0]
== color_num:
            return
        make_fragm(color_num, row_num, col_num)
```

```
        row_beg = row_num
        col_beg = col_num

    canv.bind('<B1-Motion>',mouse_move)

    def fnc_load():
        level_num = var_level.get()
        if level_num < 1:
            level_num = level_count + 1 + level_num
            if level_num < 1:
                level_num = 1

        if level_num >=  level_count:
            level_num =  level_count

        var_level.set(level_num)

        fh.seek( 400*(level_num-1) )
        read_cells()

    btn_load = ttk.Button(root, text = "завантаження", width = btn_width,
    command = fnc_load)
    btn_load.grid(row = 1, column = 0, sticky=E+N, pady = 5 , padx = 5)

    var_level = IntVar()
    var_level.set(1)

    edt_level = ttk.Entry(root, width = 5, textvariable= var_level, justify = RIGHT,
font = dFont)
    edt_level.grid(row = 1, column = 1, padx= 5, pady = 5)

    def fnc_write():
        level_num = var_level.get()
        if level_num < 1:
            level_num = 1

        if level_num >=  level_count:
            level_num =  level_count

        var_level.set(level_num)

        beg_num = 400*(level_num-1)
```

```
    fh.seek( 400*(level_num-1) )
    write_cells()

btn_write = ttk.Button( root, text = "записати", width = btn_width,
command = fnc_write)
btn_write.grid(row = 1, column = 2, sticky=E+N, pady = 5 , padx = 5)

def fnc_add():
    global level_count
    fh.seek( 0, 2 )
    write_cells()
    level_count = fh.seek( 0, 2 ) // 400
    var_level.set(level_count)

btn_add = ttk.Button( root, text = "додати", width = btn_width,
command = fnc_add)
btn_add.grid(row = 1, column = 3, sticky=E+N, pady = 5 , padx = 5)

var_color = IntVar()
for val in [0,1,2,4,5]:
    color = colors[val]
    Radiobutton(root, text="", variable=var_color, value=val, bg = color)\
                      .grid( row = 1, column = 4 + val, pady = 5 , padx = 5)

cells = []
for row_num in range(row_count):
    row = []
    for col_num in range(col_count):
        x_beg = col_num * cell_size;  x_end = x_beg + cell_size
        y_beg = row_num * cell_size;  y_end = y_beg + cell_size
        rect = canv.create_rectangle(x_beg+1, y_beg+1, x_end-1, y_end-1,
        fill = fon_color, outline = fon_color, width = 2)
        row.append([0,rect])
    cells.append(row)

file_name = "sokoban.bin"
fh = open( file_name, "rb+")
level_count = fh.seek(0,2) // 400

root.mainloop()
fh.close()
```