

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему РОЗРОБЛЕННЯ КРОСПЛАТФОРМНОЇ ПРОГРАМИ ДЛЯ
ПРОЄКТУВАННЯ ТА СИМУЛЯЦІЇ ГРАФОВИХ МОДЕЛЕЙ
DEVELOPMENT OF A CROSS-PLATFORM SYSTEM FOR DESIGNING AND
SIMULATING GRAPH-BASED MODELS

Виконав: студент 4 курсу, групи КНТ-113сп
Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ЩЕДРОВСЬКИЙ І.А.

(ПРИЗВИЩЕ та ініціали)

Керівник ФЕДОРОНЧАК Т.В.

(ПРИЗВИЩЕ та ініціали)

Рецензент БАБЕНКО Н.В.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
 (код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
 (назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ЩЕДРОВСЬКОГО Івана Андрійовича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Розроблення кросплатформної програми для проєктування та симуляції графових моделей. Development of a Cross-Platform System for Designing and Simulating Graph-Based Models

керівник проєкту (роботи) к.т.н., доцент, ФЕДОРОНЧАК Тетяна Василівна,
 (науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “07” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури системи. 3. Основні рішення щодо реалізації компонентів системи. 4. Експлуатація, тестування та експериментальне дослідження програми

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ФЕДОРОНЧАК Т.В., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)


(підпис)

Іван ЩЕДРОВСЬКИЙ
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис)

Тетяна ФЕДОРОНЧАК
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 91 с., 3 табл., 24 рис., 2 дод., 28 джерел.

ЛОГІЧНА СХЕМА, ГРАФОВА МОДЕЛЬ, ПРОЄКТУВАННЯ ТА СИМУЛЯЦІЯ СХЕМ, GO, EBITENGINE.

Об'єкт дослідження – це процес розроблення кросплатформного програмного забезпечення на основі графових моделей.

Предмет дослідження – це методи та програмні засоби представлення, обходу графових моделей і симуляції сигналів у вузлах графу.

Метою роботи є зменшення часу та трудомісткості проєктування цифрових логічних схем шляхом розроблення кросплатформного інструменту з інтерактивною симуляцією графових моделей.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, графові моделі, Go, ebitengine, цифрові логічні схеми, персональний комп'ютер з процесором Intel Core i5-9300HF під управлінням системи Arch Linux.

Результати. Створено програмний застосунок, який підтримує всі операційні системи та дозволяє спроектувати та симулювати логічні схеми.

Висновки. Створено кросплатформний застосунок з інтерактивною симуляцією графових моделей, який зменшує час та трудомісткість проєктування цифрових логічних схем.

Галузь використання – всі галузі, які працюють з комп'ютерами та потребують виконання логічних операцій.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 91 pages, 3 tables, 24 figures, 2 appendixes, 28 sources.

LOGIC CIRCUIT, GRAPH-BASED MODEL, CIRCUIT DESIGN AND SIMULATION, GO, EBITENGINE.

The object of research is the process of developing cross-platform software based on graph models.

The subject of research is the methods and software tools for representing, traversing graph models and simulating signals in graph nodes.

The goal of the work is to reduce the time and labor-intensiveness of designing digital logic circuits by developing a cross-platform tool with interactive simulation of graph models.

Materials, methods, and technical tools: structural and object-oriented programming, graph-based models, Go, ebitengine, digital logic circuits, a personal computer with an Intel Core i5-9300HF processor running Arch Linux.

Results. A software application has been created that supports all operating systems and allows for the design and simulation of logic circuits.

Conclusions. A cross-platform application with interactive simulation of graph models has been created, which reduces the time and labor-intensiveness of designing digital logic circuits.

Field of application – all industries that work with computers and require the performance of logical operations.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області	10
1.1 Основні поняття	10
1.2 Огляд існуючих програмних рішень	11
1.3 Постанова завдання	17
1.3.1 Функціональні вимоги до програми.....	17
1.3.2 Вимоги до надійності.....	21
1.3.3 Умови роботи програми	21
1.3.4 Умови розповсюдження програми	22
2 Розробка архітектури програми	23
2.1 Обґрунтування вибору середовища функціонування системи.....	23
2.2 Обґрунтування вибору програмних засобів	24
2.3 Обґрунтування вибору середовища розробки системи	27
2.4 Структурна схеми програми.....	30
2.5 Основні сценарії використання.....	31
3 Основні рішення щодо реалізації компонентів системи	33
3.1 Архітектура та структура програми	33
3.2 Використовувані моделі даних	34
3.3 Розробка модулів системи	43
4 Експлуатація, тестування та експериментальне дослідження програми.....	50
4.1 Призначення і умови використання програми	50
4.2 Характеристики програми	50
4.3 Інструкція по експлуатації програми	51
4.3.1 Звертання до програми	51
4.3.2 Вхідні і вихідні дані	51
4.4 Експлуатація програми	51

4.5 Тестування програми.....	58
Висновки	62
Перелік джерел посилання	63
Додаток А Фрагмент тексту програми.....	66
Додаток Б Слайди презентації	82

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- API – Application Programming Interface;
- GNU – General Public License;
- HTTP – Hypertext Transfer Protocol;
- IDE – Integrated Development Environment;
- LSP – Language Server Protocol;
- VHDL – VHSIC Hardware Description Language;
- VHSIC – Very High Speed Integrated Circuit;
- ОС – операційна система;
- ПЗ – програмне забезпечення.

ВСТУП

Сучасний етап розвитку цивілізації характеризується тотальною цифровізацією, де основою будь-якого обчислювального процесу є логічні операції. Цифрові схеми сьогодні стали невидимим фундаментом суспільства: від найпростіших побутових приладів до складних систем штучного інтелекту та космічних апаратів. Усе розмаїття сучасних технологій – мобільний зв'язок, глобальні мережі, автоматизоване виробництво – базується на принципах обробки дискретних сигналів за допомогою логічних вентилів.

Проте зі зростанням складності обчислювальних систем виникає гостра потреба у наочності та інструментах для проектування. Розуміння того, як саме одиниці та нулі перетворюються на складні алгоритми, є критично важливим як для навчання майбутніх інженерів, так і для розробки нових архітектур процесорів. Моделювання та симуляція логічних схем дозволяють перевірити працездатність цифрових систем ще до їхнього фізичного втілення, що значно економить ресурси та запобігає критичним помилкам у проектуванні апаратного забезпечення.

Мета цієї дипломної роботи полягає в зменшенні часу та трудомісткості проектування цифрових логічних схем шляхом розроблення кросплатформного інструменту з інтерактивною симуляцією графових моделей.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Основні поняття

Предмет науки (предметна область) – це ті сторони, зв'язки, відношення об'єкта, які вивчаються даною наукою [1].

Розробка програмного забезпечення (ПЗ) – це вид діяльності та процес, спрямований на створення та підтримку працездатності, якості та надійності ПЗ, використовуючи технології, методологію та практики з інформатики, керування проєктами, математики, інженерії та інших областей знання. Як і інші традиційні інженерні дисципліни, розробка ПЗ має справу з проблемами якості, вартості та надійності. Деякі програми містять мільйони рядків вихідного коду, які, як очікується, повинні правильно виконуватися в умовах, що змінюються. Складність ПЗ порівнянна зі складністю найбільш складних сучасних машин (таких, наприклад, як літаки) [2].

Складність програмного забезпечення це все, що відноситься до системи, що робить її більш складною та розуміння та внесення змін. Вона може проявлятися у різних формах. Наприклад, може бути важко зрозуміти, як працює певний фрагмент коду; може знадобитися багато зусиль для реалізації незначного покращення, або може бути незрозуміло, які частини системи необхідно змінити для його впровадження; може бути складно виправити один баг, не спричинивши появи іншого. Якщо програмну систему важко зрозуміти та змінити, то вона є складною; якщо її легко зрозуміти та змінити, то вона є простою [3].

Об'єкт дослідження – це процес розроблення кроссплатформного програмного забезпечення на основі графових моделей.

Предмет дослідження – це методи та програмні засоби представлення, обходу графових моделей і симуляції сигналів у вузлах графу.

Метою роботи є зменшення часу та трудомісткості проєктування цифрових логічних схем шляхом розроблення кроссплатформного інструменту з інтерактивною симуляцією графових моделей.

Графова модель представляє дані у вигляді пов'язаних сутностей. Якщо задача стосується зв'язків, шляхів, залежностей або мереж, така модель зазвичай підходить краще, ніж традиційна реляційна або документна структура [4].

В основі графової моделі даних лежить представлення даних у вигляді мережі. Кожна сутність стає вузлом, а кожен зв'язок між сутностями – ребром [5].

Логічна схема – це набір двох або більше логічних елементів, які з'єднані для реалізації більш складної логіки. Цифрова електронна схема в комп'ютерній системі зазвичай складається з мільйонів логічних елементів. Кожна схема призначена для виконання певної функції, наприклад додавання двійкових чисел [6].

Логічний вентиль є основним електронним компонентом цифрової схеми. Кожен вентиль виконує логічну операцію на одному або кількох входах для створення єдиного виходу. Кілька логічних вентилів можна комбінувати для створення складних схем для виконання обробки [7].

1.2 Огляд існуючих програмних рішень

Для визначення актуальності поставленої задачі було проаналізовано три існуючих програмних рішення, які створені для проєктування та симуляції логічних схем.

Перша програма це Logisim, яка була створена розробником та викладачем Карлом Берчем в 2000 році, остання версія програми вийшла в 2011 році. В 2014 році автор повідомив про офіційне припинення розробки. Приклад екрану цієї програми показаний на рис. 1.1 [8].

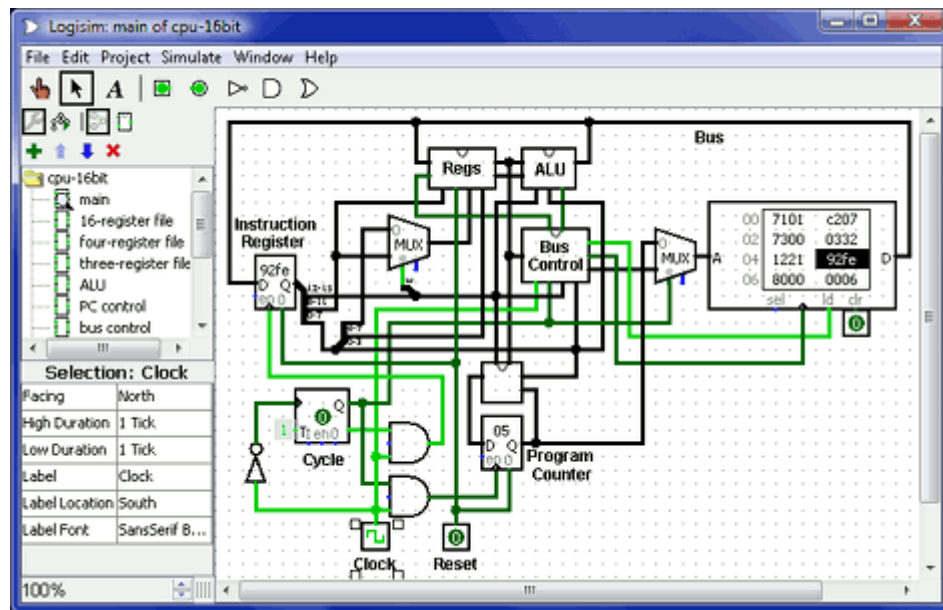


Рисунок 1.1 – Приклад виконання програми Logisim [8]

Із великого плюсу програми можна відмити, що застосунок має відкритий вихідний код під ліцензією GNU General Public License, згідно якої кінцеві користувачі можуть безпечно запускати, вивчати, поширювати та змінювати вихідний код застосунку. Вихідний код програми в тому числі знаходиться в публічному доступі [9].

Із плюсів програми можна відмітити її швидкодію та наявність великої кількості компонентів для проєктування схем. Також програма підтримує симуляцію схем.

Із мінусів, оскільки остання версія програми виходила в 2011 році, її інтерфейс є застарілим в порівнянні з сучасними програмами, і через такий інтерфейс в застосунку достатньо складно проєктувати схеми. Поле, на якому можливо розміщувати елементи є обмеженим та по ньому не можливо переміщуватись.

Інший недолік полягає в встановленні програми, оскільки для того, щоб використовувати застосунок потрібно також встановити Java від версії 5. В застосунку також використовується бібліотека Swing, і оскільки ця бібліотека не оновлювалась протягом довгого часу можна гарантувати, що швидкодія програми не є максимальною [9].

Одним з недоліків цієї програми є логіка створення зв'язків між елементами. В застосунку реалізована логіка, що зв'язки є незалежними від елементів та їх можна ставити на поле окремо, в тому числі від інших зв'язків. Це приводить до проблеми, коли користувач хоче перемістити зв'язок, але замість цього створюється новий.

Друга програма це Digital, яку створив Helmut Neemann в 2016 році [10]-[11]. Приклад цієї програми показаний на рис. 1.2.

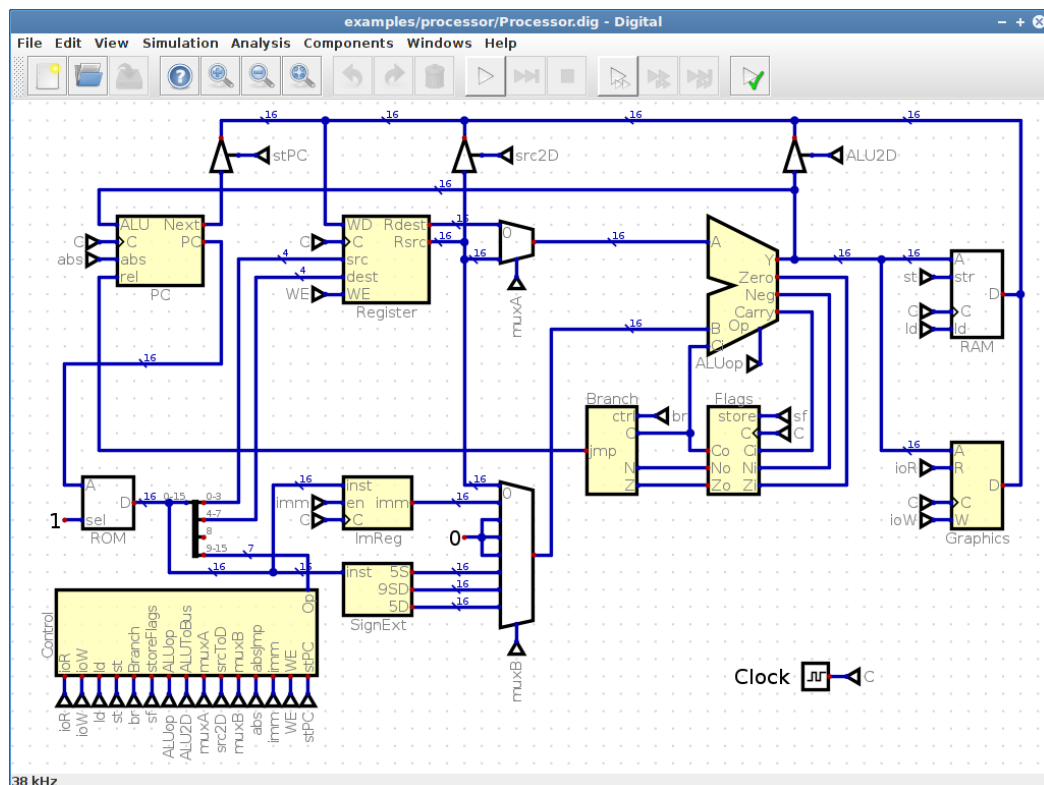


Рисунок 1.2 – Приклад виконання програми Digital [11]

Ця програма є новішою, а ніж Logisim та підтримується до сьогодні. Застосунок написаний також на Java, має ліцензію GNU General Public License, згідно якої кінцеві користувачі можуть безпечно запускати, вивчати, поширювати та змінювати вихідний код застосунку. Вихідний код програми в тому числі знаходиться в публічному доступі [11].

Із переваг цього застосунку можна відмітити його надзвичайно високу швидкодію під час симуляції, можливість експорту схем на мові опису апаратури інтегральних схем VHDL, а також можливість перевірки схеми

через таблиці істиності. Наступною перевагою є велика бібліотека елементів, з яких можна проєктувати схеми [11].

Також в застосунку реалізовано нескінченне поле, на якому можна розміщувати елементи, в тому числі наявна функція масштабування.

Із недоліків застосунку можна відмітити його складний інтерфейс. Для створення елементів потрібно відкривати меню, відкривати вкладку та шукати потрібний, що займає велику кількість часу. Оскільки інструмент орієнтований на низькорівневе проєктування процесорів, то він має високий поріг входу [10].

Іншим недоліком є логіка створення зв'язків між елементами. В застосунку для того, щоб створити зв'язок потрібно натиснути на одну точку екрану, провести мишу в іншу точку та натиснути ще раз, тобто зв'язки не прив'язані до елементів. Це також дозволяє створювати зв'язки, які залежать від інших зв'язків, але при цьому це створює проблеми під час переміщення елементів.

Недолік при переміщенні зв'язків або елементів полягає в тому, що зв'язки переміщуються по точкам, які знаходяться на краях. Обираючи зв'язок користувачу показується, що він обрав весь зв'язок, але після здвигу переміщується тільки одна з точок цього зв'язку. При переміщенні елементу, якщо користувач вибере його через натискання, то його зв'язки залишаться на позиціях до переміщення.

Третій розглянутий застосунок це Logicly. Його перша версія була випущена в 2010 році розробником Josh Tynjala. Він має активну підтримку і до сьогодення часу та створений, й використовується, для навчання студентів в різних університетах по всьому світу [12]-[13].

Цей застосунок не має відкритого вихідного коду, а також є повністю комерційним. Це унеможлиблює його модифікацію під специфічні потреби навчального процесу та унеможлиблює вивчення студентами його внутрішньої будови [13].

Приклад його використання показаний на рис. 1.3.

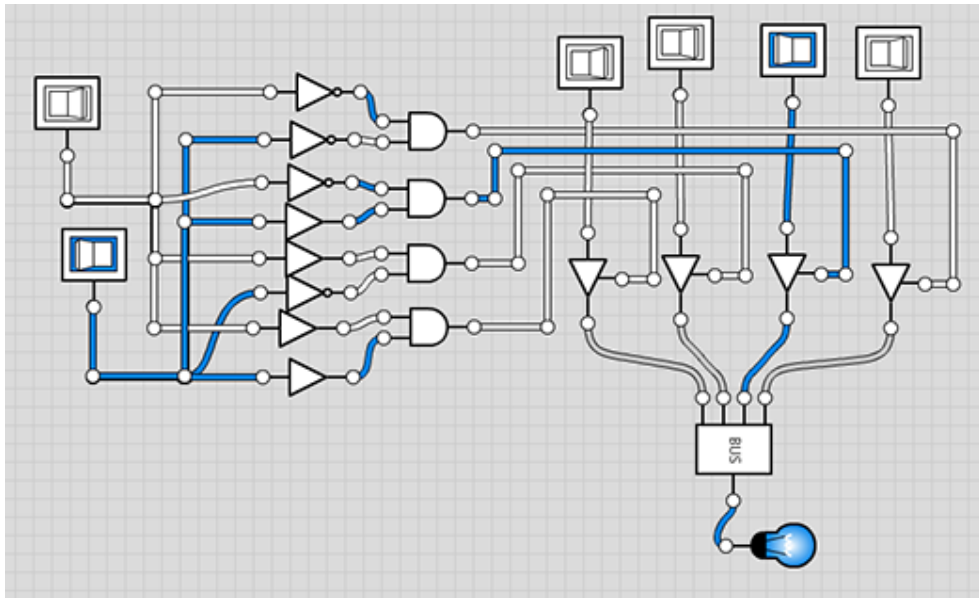


Рисунок 1.3 – Приклад використання Logicly [13]

Із переваг цього застосунку варто відмітити наявність безкоштовної демо-версії, а також простоту його використання. Інтерфейс застосунку, який показаний на рис. 1.4, є простим та зрозумілим, а тому з ним зручно працювати.

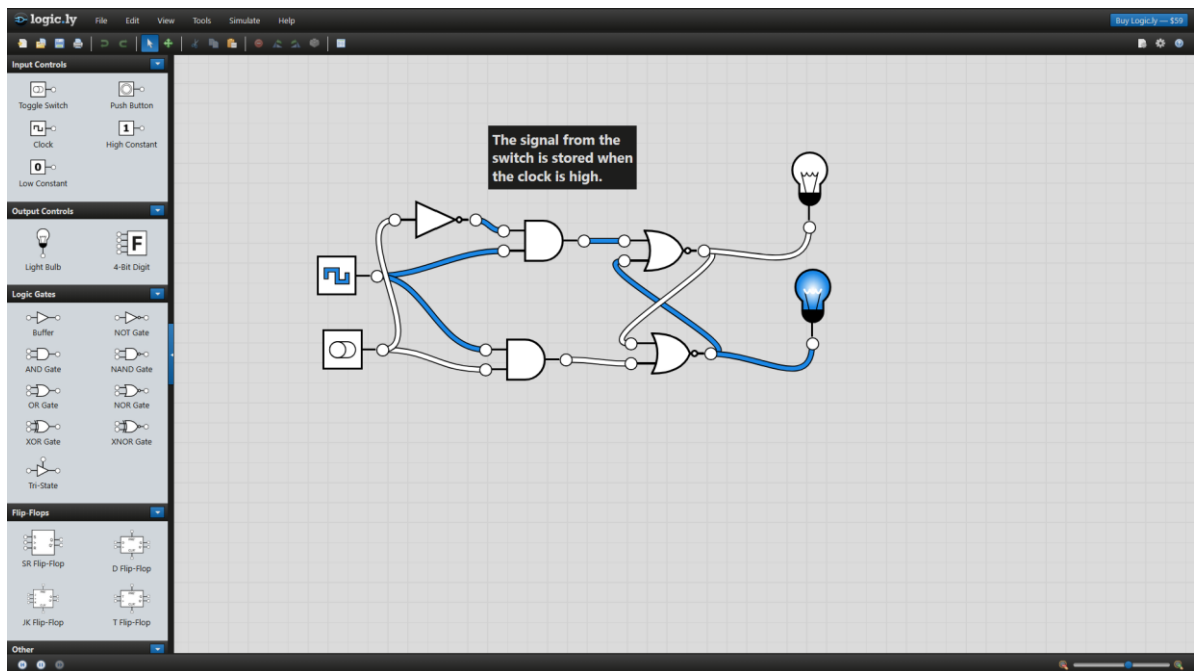


Рисунок 1.4 – Інтерфейс демо-версії застосунку Logicly [13]

Із плюсів, в програмі зручно реалізована логіка створення зв'язків, оскільки на кожному елементі є маркер, після кліку на який можна створити зв'язок. Зв'язок створюється тільки з одного елементу до іншого, тобто в програмі немає можливості створити зв'язок, який буде залежати від іншого зв'язку.

Із недоліків варто зауважити функціональну обмеженість застосунку в порівнянні з іншими, але враховуючи специфіку цього застосунку можливо це навпаки є його перевагою. Оскільки застосунок створений для навчальних цілей в ньому буде складно спроектувати великі схеми [13].

Наступним недоліком є відсутність функцій, які працюють через комбінації клавіш.

В табл. 1.1 наведено порівняння трьох розглянутих застосунків.

Таблиця 1.1 – Порівняння існуючих рішень

Критерій порівняння	Існуючі програми		
	Logisim	Digital	Logicly
Відкритий програмний код	Так	Так	Ні
Вартість	Безплатний	Безплатний	Комерційний
Ціль створення	Проектування великих схем	Проектування великих схем	Навчання студентів
Зручність інтерфейсу	Низька	Низька	Висока
Кількість елементів	Велика	Велика	Середня
Зручність створення зв'язків	Низька	Низька	Висока
Робоча область	Обмежена	Необмежена, з можливістю переміщення	Необмежена, з можливістю переміщення

Аналіз приведений аналогів показує, що деякі існуючі рішення є безкоштовними з відкритим кодом, але при цьому застарілі і мають незадовільний стан користувацького інтерфейсу та залежність від сторонніх середовищ для їх запуску. З іншого боку існують сучасні програми, але вони повністю комерційні та мають закритий вихідний код, що обмежує їх використання.

Таким чином, створення нової кросплатформної програми, яка буде мати зручний інтерфейс, високу швидкодію, і при цьому буде безкоштовною та відкритою є повністю актуальним.

1.3 Постанова завдання

Система, що розробляється, буде орієнтована як на студентів технічних спеціальностей, що вивчають основи цифрової логіки, так і на досвідчених розробників, яким потрібен швидкий та легкий інструмент для прототипування схем без необхідності встановлення громіздкого професійного програмного забезпечення.

Основний фокус системи, під час проєктування та розробки, буде саме на простоті її використання й швидкості її роботи.

1.3.1 Функціональні вимоги до програми

Застосунок повинен підтримувати набір функцій для проєктування та симуляції цифрових логічних схем на рівні логічних вентилів.

Для зручного проєктування схем в застосунку повинне бути безкінечне робоче поле, на якому користувач зможе розміщати, поєднувати елементи. Користувач повинен мати змогу переміщатись по полю, а також можливість наближати та віддаляти схему.

На полі у користувача повинна бути можливість створювати, видаляти та переміщати логічні вентиля, які далі називаються логічними елементами або просто елементами.

Між логічними елементами користувач повинен мати змогу створити зв'язок, або з'єднання. З одного елементу може виходити необмежена кількість зв'язків, а кількість зв'язків, що приходить в елемент визначається його типом. Наприклад, логічний ventиль «AND» повинен мати два вхідні зв'язки, а ось ventиль «NOT» повинен мати тільки один. Якщо кількість вхідних зв'язків не відповідає необхідному, то такий елемент повинен показати помилку користувачу і бути виключеним з симуляції схеми.

Важливо зауважити, що з'єднання повинні створювати виключно з одного елементу в інший елемент. Застосунок не буде підтримувати з'єднання від елементу в цей самий елемент, а також не буде підтримування створення зв'язку з іншого зв'язку в елемент. Це також означає, що зв'язки не можуть існувати без прив'язки до елементів. Ситуація, коли існує з'єднання без одного з елементів є критичною помилкою та не повинна виникати.

Програма буде підтримувати всі основні логічні ventелі, а саме «AND», «OR», «XOR» та «Buffer». Також програма повинна підтримувати інвертовані варіанти ventилів, а саме «NAND», «NOR», «XNOR» та «NOT» [7].

Для роботи створення повноцінних схем і роботи з ними застосунок повинен підтримувати спеціальні елементи, які будуть подавати вхідний сигнал та користувач повинен мати змогу його змінювати.

В застосунку повинні бути елементи, які будуть зчитувати сигнал, і пропускати його через себе. Загально такі елементи будуть використовуватись для перегляду результату виконання схеми, але також можуть використовуватись всередині схеми, оскільки вони пропускають сигнал.

Також повинен бути присутній спеціальний елемент, який буде працювати для з'єднання декількох незалежних схем. Він потрібен, оскільки таким чином користувач зможе створити схему з багатьма вхідними з'єднаннями, а після скопіювати її декілька разів і правильно підключити.

Користувач повинен мати змогу обрати декілька елементів одночасно, в тому числі через затискання лівої кнопки миші та візуальне виділення по області, й видалити, перемістити або скопіювати їх разом. Під час всіх операцій пов'язанні посилання повинні також переміщатись, видалятись або копіюватись, але копіюватись повинні тільки посилання у яких обидва елементи обрані.

Для надання користувачу максимальної зручності використання застосунку, користувач повинен мати можливість виконати потрібну дію через натискання спеціальної комбінації клавіш. Це потрібно, та важливо, оскільки це дозволить пришвидшити роботу з застосунком для досвідчених користувачів.

В застосунку повинна бути реалізована система коли користувач може зберегти поточну схему в файл з розширенням застосунку, а після відкрити його в майбутньому. В програмі це буде називатись проєктом. Користувач може відкрити новий проєкт з файлу, закрити його, при закритті буде створюватись новий пустий проєкт, а також зберегти його. Якщо проєкт був відкритий з файлу, то при стандартному збереженні він повинен бути оновлений в цьому файлі. Якщо проєкт ще не був збережений, або користувач хоче зберегти його в новий файл, він повинен мати змогу це зробити. Програма не повинна підтримувати відкриття декількох проєктів, але повинна підтримувати візуальну демонстрацію назви проєкту, що відкритий, та чи були останні зміни в ньому збережені.

В програмі повинен бути візуальний інтерфейс, через який користувач зможе:

- відкрити новий проєкт, зберегти, зберегти в новий файл та закрити поточний;
- побачити назву та чи збережений відкритий проєкт;
- побачити список всіх комбінацій клавіш та їх дій;
- створити новий елемент на робочій області.

Важливо зауважити, що цей список є необхідним, але може доповнюватись новими функціями під час створення програми.

У користувача повинна також бути візуальна можливість створювати зв'язки з елементів, через наведення на роз'єм елемента та після натискання на інший елемент.

Застосунок повинен підтримувати симуляцію схеми, тобто розрахунок вихідних параметрів схеми на основі вхідних, а також візуальну демонстрацію сигналу на кожному кроці схеми.

Симуляція повинна завжди починатись з вхідних елементів та рухатись поступово до інших. Тобто, наприклад, вихідний сигнал вентилю «NOT» буде розрахований тільки тоді, коли логіка симуляції дійде до цього елемента з правильно розрахованим вхідним сигналом. Дана логіка є дуже важливою, оскільки при розрахунку всіх станів одночасно схема може прийти в невизначений стан.

При зміні вхідних сигналів схема повинна перераховувати свій стан, але при цьому враховувати попередній, а не розраховувати все з нуля. Це важливо для правильної роботи, наприклад, комірок пам'яті, стан яких напряму залежить від попереднього.

Симуляція схеми повинна підтримувати наявність циклів, коли, наприклад, перший елемент залежить від другого, а другий залежить від першого. Наявність циклів є необхідною умовою для правильної роботи комірок пам'яті та створення достатньо складних схем.

При виникненні нескінченного циклу в схемі, що зазвичай відповідає постійній змінній вихідного сигналу з 0 на 1 та з 1 на 0, програма повинна показати користувачу помилку.

1.3.2 Вимоги до надійності

В програмі повинна бути реалізована система обробки помилок користувача. Такі помилки не повинні призводити до зупинки застосунку, а натомість застосунок повинен показувати користувачу що саме пішло не так.

Наприклад, якщо користувач під'єднав три вхідні зв'язки в елемент, що підтримує тільки два, програма повинна це зрозуміти та візуально показати користувачу, що цей елемент не буде працювати правильно. Або ж, інший приклад, при завантаженні проєкту з файлу, який був пошкоджений або має не правильну структуру, застосунок повинен також показати користувачу помилку в вигляді сповіщення.

В застосунку повинні бути реалізовані внутрішні перевірки, які будуть автоматично закривати його при критичних помилках. Під цю категорію також підпадають інваріанти, тобто умови, які завжди повинні бути правдою. Наприклад, якщо система починає симуляцію та розуміє, що в загальному сховищі ще не була створена потрібна для симуляції інформація, то програма повинна автоматично закритись, оскільки подібні помилки можуть привести до непередбачуваного стану програми, який в свою чергу приводить до великої кількості проблем з безпекою та стабільністю роботи в цілому [14].

Обробка ж внутрішніх помилок застосунку лягає на мову програмування. Наприклад, при спробі доступу хеш таблиці значень, яка ще не була створена, або ж спроби отримання значення за вказівником коли він вказує ще не створену пам'ять, застосунок повинен автоматично закриватись з помилкою, і подібні помилки повинна відстежувати мова програмування.

1.3.3 Умови роботи програми

Програма повинна встановлюватись безпосередньо на комп'ютер користувача.

Мінімальними вимогами до програмного забезпечення є:

- сучасний 64-бітний процесор;
- мінімум 4 ГБ оперативної пам'яті;
- відеокарта з підтримкою OpenGL 2.1 або вище;
- дисплей з мінімальною роздільною здатністю в 1280 на 720 пікселів.

Програма повинна підтримувати стабільну роботу на операційних системах Microsoft Windows 10, Windows 11, Mac OS 13, а також системи Linux. Для роботи на Linux у користувача повинна бути встановлена графічна оболонка, наприклад Wayland або X11, повинна бути підтримка OpenGL, а також стандартних бібліотек C. Застосунок не повинен мати доступу до мережі інтернет, для забезпечення максимальної стабільності його роботи, а також для безпеки користувача.

Для коректної роботи програми у користувача повинні бути необхідні пристрої введення інформації, а саме миша з коліщатком та клавіатура з підтримкою багатоклавішних комбінацій. Програма не повинна потребувати прав адміністратора для її запуску та подальшої роботи.

1.3.4 Умови розповсюдження програми

Даний застосунок повинен бути безкоштовний і доступний для завантажування користувачами через мережу Інтернет. Програмний продукт повинен мати відкритий вихідний код.

Програма повинна постачатись у вигляді одного виконавчого файлу і не потребувати додаткових операцій встановлення на комп'ютер користувача.

Всі графічні елементи та шрифти повинні бути вбудовані в один файл, разом з кодом застосунку.

Ліцензія застосунку повинна дозволяти всім користувачам запускати, переглядати, змінювати та досліджувати вихідний код програми, але при цьому залишаючи авторські права за розробником. Для кожної цільової операційної системи повинен бути окремий скомпільований файл.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Обґрунтування вибору середовища функціонування системи

Для вибору найкращого середовища функціонування системи визначимо основні вимоги від середовища для роботи застосунку.

Середовище повинно мати підтримку показу інформації на екрані в гарному розширенні. Застосунок орієнтований на проєктування та симуляцію схем, і згідно завдання повинен мати гарну, зручну візуальну частину через яку користувач зможе просто взаємодіяти з програмою.

Користувач повинен мати можливість використовувати застосунок без підключення до мережі інтернет. Ця вимога призвана забезпечити конфіденційність та безпеку користувача. Без підключення до мережі застосунок не зможе автоматично встановлювати оновлення та не зможе передавати дані користувача, а тому, в випадку компрометації застосунку і випуску нової несанкціонованої версії, користувачі будуть захищені. Також таке обмеження накладає додатковий тиск на застосунок з точки зору можливих помилок в його роботі та оптимізації, оскільки через неможливість виконувати обчислення на окремому пристрої, застосунок повинен працювати швидко завжди, а через неможливість швидко оновити застосунок у всіх користувачів, в ньому повинно бути мінімум можливих проблем.

Середовище повинно мати підтримку вводу інформації через мишку та клавіатуру, або їх аналоги. Для забезпечення швидкості використання програми у користувача, згідно завдання, повинна бути можливість виконувати деякі дії натискаючи потрібні комбінації клавіш. Також для забезпечення максимальної зручності у користувача повинна бути можливість змінювати/переміщування та видаляти елементи використовуючи мишку, або її аналоги.

Згідно цих вимог, в якості середовища виконання програми підходять мобільні та застосунки, що встановлюються безпосередньо на комп'ютер користувача.

Також під цей опис можуть підходити веб застосунки, але для забезпечення роботи без інтернету потрібно робити додаткові налаштування, і при цьому перше завантаження застосунку все одно буде відбуватись через мережу. Також, із можливих проблем, це зав'язка на оточення браузера, і в тому числі на всі обмеження цього оточення. Наприклад, використовуючи браузер програма не зможе напряду взаємодіяти з файловою системою, щоб зберігати поточний проєкт.

Мобільні застосунки не є доцільним середовищем функціонування через обмежену підтримку пристроїв введення інформації. Хоча наразі є можливість підключити мишку та клавіатуру до мобільних пристроїв, це є скоріше виключенням з правил.

Тому, згідно всіх вище описаних вимог було визначення, що застосунок буде встановлюватись безпосередньо на комп'ютер користувача. Застосунок повинен працювати на системах Windows 10 та Windows 11, MacOS та Linux. При встановленні застосунку на Linux у користувача повинен бути встановлена система для відображення інформації, потрібні драйвера й інші інструменти для роботи програми.

2.2 Обґрунтування вибору програмних засобів

Для реалізації застосунку було обрано мову програмування Go.

Ця мова програмування була створена всередині компанії Google в 2007 році трьома спів-авторами, а саме Робертом Гризмером, Робом Пайком й Кеном Томсоном [15]-[16].

Роберт Гризмер до роботи над Go працював в Google над найпопулярнішим двигуном для запуску JavaScript V8 [15]-[17].

Роб Пайк і Кен Томсон разом працювали над операційною системою Plan 9 в Bell Labs, а також вони є спів-авторами стандарту кодування символів UTF-8 [15]-[17].

Кен Томсон відомий тим, що коли працював в Bell Labs, він проектував та створював операційну систему Unix. Також він є автором мови програмування B, працював над створенням регулярних виразів [15].

Кен Томсон вважається одним з найкращих, тобто входить в десятку кращих, розробників за всі часи [18].

При створенні мови програмування Go перед авторами постала задача створити мову з гарним балансом. По-перше, щоб вона була елегантна та проста, тобто щоб код на цій мові можна було просто писати та читати. А по-друге, щоб вона була такою самою швидкою як C++. І найбільш важлива характеристика це можливість працювати з паралельністю, тобто можливістю одночасно виконувати декілька задач, в простому та швидкому вигляді [17].

Однією з основних переваг цієї мови – наявність великої кількості вбудованих та стандартних інструментів. Наприклад, в цій мові є вбудована підтримка для роботи з картинками, розгортання HTTP серверів, а також читання та запису файлів. Через таке різноманіття вбудованих інструментів в Go можливо створити повноцінні серверні застосунки без встановлення додаткових бібліотек, що якісно відрізняє цю мову від інших [15].

Зазвичай Go порівнюють з мовами Rust та Zig, оскільки вони дуже схожі за сферою застосування та загальним положенням. На табл. 2.1 показано порівняння мов Go, Rust та Zig.

Таблиця 2.1 – Порівняння мов програмування

Параметр порівняння	Мова програмування		
	Go	Rust	Zig
1	2	3	4
Складність вивчення	Середня	Дуже висока	Висока
Рік випуску	2009	2010	2016
Швидкодія	Висока	Висока	Висока
Швидкість розробки	Висока	Середня	Середня

Продовження таблиці 2.1

1	2	3	4
Управління пам'яттю	Автоматичне, через збирач сміття	Автоматичне, через Borrow Checker	Ручне
Документація	Повноцінна	Повноцінна	Майже відсутня
Розробник	Компанія Google	Некомерційна організація Rust Foundation	Некомерційна організація Zig Software Foundation

Мова програмування Rust, на відмінну від Go, є дуже великою та має репутацію такої, що складно вивчити. Ця мова може гарантувати безпеку, а особливо безпеку використання пам'яті, тобто гарантію, що ми не можемо отримати інформації з не правильного вказівника, або ж що очищення відбудеться декілька разів. Безпечна програма не має невідомої поведінки. Якщо в нашій програмі щось пішло не так, то одна з найкращих речей, яка може відбутись це повна її зупинка, оскільки альтернативно, якщо помилку не було зловлено, наша програма переходить в невизначений стан, де її наступні дії будуть залежати від великої кількості факторів, наприклад чи спрацював збиральник сміття, чи отримали ми дані тощо, і така наявна поведінка веде до появи багів, які з'являються тільки в деяких ситуаціях та в деякий час та проблем з безпекою [15].

Мова програмування Zig є найновішою з трьох. Стандартна бібліотека цієї мови програмування наразі майже не має документації. Мова Zig має переваги, які є в Go та Rust одночасно, і при цьому надає більше контролю над роботою програми. Невизначена поведінка також є важливою, і тут називається протиправною, мова намагається визначити її під час виконання програми та закриває застосунок, якщо така поведінка з'являється [15].

Для реалізації візуальної частини програми було використано ebitengine та ebitenui. Бібліотека ebitengine це дуже простий двигун для створення ігор на Go. Вона дозволяє показувати зображення, текст та

малювати різні фігури на екрані користувача. Великою перевагою цієї бібліотеки є її простота та близькість до стандартної бібліотеки Go [19].

Для створення візуальної частини було взято саме готову бібліотеку, а не OpenGL або інші API, оскільки ця бібліотека одночасно підтримує багато середовищ та різні API. Наприклад, ebitengine підтримує відображення елементів на всіх операційних системах, в браузерях через WebAssembly та навіть на мобільних пристроях [19].

Також було використано бібліотеку zenity, яка дозволяє запитувати у користувача файл в файловій системі для збереження або відкриття проєкту. Ця бібліотека потрібна, оскільки різні операційні системи мають свій API для запиту файлів, показу нотифікацій, підтверджень інформації тощо [20].

Графічна частина застосунку буде реалізована з використанням піксельної графіки, тобто зображень, де роздільна здатність настільки мала, що ми можемо побачити окремі пікселі. Для створення піксельної графіки для застосунку було використано програму Aseprite. Ця програма підтримує як створення окремих піксельних зображень, так і анімацій, або набору з декількох зображень, які поєднуються в одне велике, де кожне окреме зображення знаходиться в спеціальній сітці. Це потрібно, та буде використовуватись в розробці цього застосунку, оскільки один раз імпортувати зображення та розрізати його на менші програмно буде ефективніше, а ніж імпортувати сотні маленьких [21].

2.3 Обґрунтування вибору середовища розробки системи

Для розробки програмного забезпечення було використано інтегроване середовище розробки «GoLand» від компанії JetBrains.

Згідно з опитуванням розробників на Go за 2025 рік, GoLand є другою за популярністю IDE. Графік опитування щодо інтегрованих середовищ розробки показаний на рис. 2.1 [22].

What is your favorite code editor for Go?

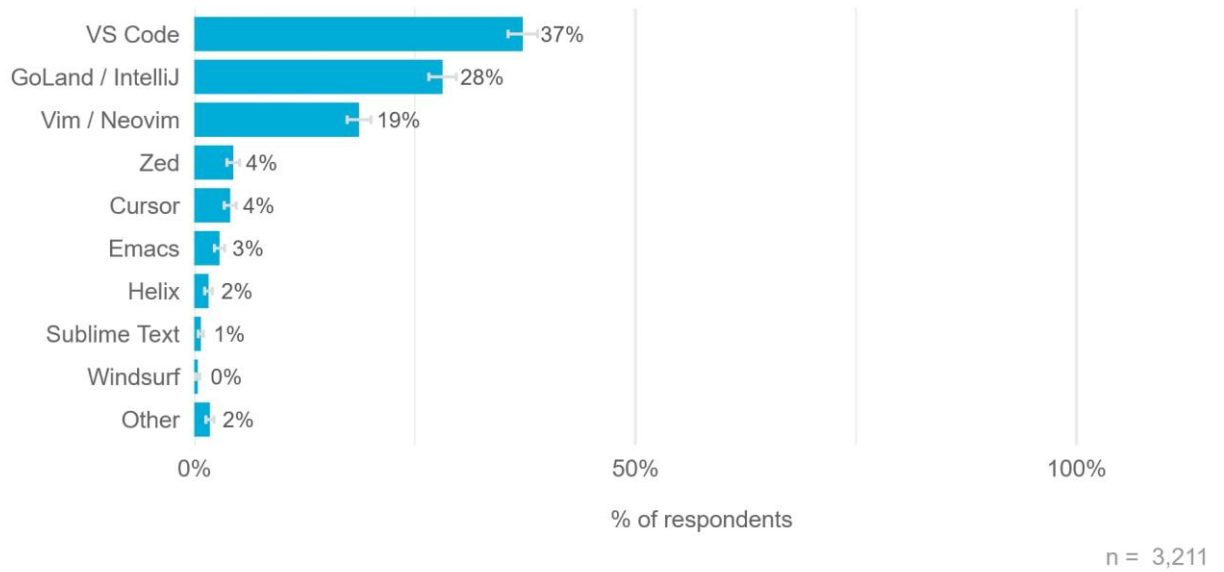


Рисунок 2.1 – Опитування про інтегроване середовище розробки на Go [22]

Згідно результатів опитування від розробників Go яке показано на рис. 2.1, найбільш популярним середовищем розробки на Go є Visual Studio Code. Ця IDE підтримує майже всі існуючі мови програмування та має велику кількість доповнень, які можуть створювати звичайні користувачі [23].

На відміну від Visual Studio Code, яка підтримує майже всі мови програмування, GoLand був спеціально створений саме для роботи з Go та має всі інструменти створенні з задумом, що вони будуть працювати саме з Go. Це є ключовою відмінністю інтегрованого середовища розробки яке орієнтовано на всі мови та всі можливі ситуації від середовища, яке чітко орієнтоване саме на конкретну мову програмування з її застосуваннями [23]-[24].

На табл. 2.2 показано порівняння середовищ розробки GoLand, Visual Studio Code та Neovim.

Таблиця 2.2 – Порівняння середовищ розробки

Параметр порівняння	Середовище розробки		
	GoLand	Visual Studio Code	Neovim
Мета створення середовища	Розробка програм виключно на мові програмування Go	Розробка програм на всіх мовах програмування	Розробка програм на всіх мовах програмування
Швидкодія	Висока	Висока	Найкраща
Налаштування для роботи	Майже не є потрібні	Потрібно встановити необхідні розширення для мови Go	Потрібно встановити велику кількість розширень для повноцінної роботи
Рівень кастомізації	Низький	Середній	Дуже високий
Кількість доповнень	Середня	Висока	Дуже висока
Автоматична синхронізація між іншими середовищами розробки	Вбудована, через персональний аккаунт	Не вбудована, можлива через додаткові розширення	Не вбудована, можлива через додаткові розширення
Власник	Компанія JetBrains	Компанія Microsoft	Немає, безкоштовний відкритий проєкт

Другою перевагою GoLand над Visual Studio Code та в тому числі Neovim є наявність всіх необхідних інструментів для розробки відразу після встановлення середовища розробки. Наприклад, в Neovim, щоб отримати гарну підтримку Go потрібно встановити менеджер плагінів, встановити LSP та після додаткові плагіни для оптимізації або зручності роботи. В Visual Studio Code ситуація схожа, оскільки для підтримки Go потрібно також встановити необхідні доповнення [24]-[25].

Додатковою проблемою є кількість доповнень. Якщо середовище підтримує велику кількість мов програмування та розробник працює з

великою кількістю мов, то кількість доповнень може доходити до сотень, що буде сповільнювати роботу середовища в цілому [26].

Додатковою перевагою GoLand є автоматична синхронізація налаштувань та доповнень з іншими програмними продуктами компанії JetBrains. Це дозволяє з мінімальною кількістю зусиль перейти з одного з одного середовища в інше, оскільки розробнику не потрібно знову змінювати налаштування та встановлювати доповнення [24].

2.4 Структурна схеми програми

Внутрішня схема застосунку буде складатись з трьох основних шарів:

- шар презентації, візуальна частина;
- шар бізнес-логіки, внутрішньої логіки застосунку;
- шар взаємодії з інформацією.

Схема застосунку показана на рис. 2.2.

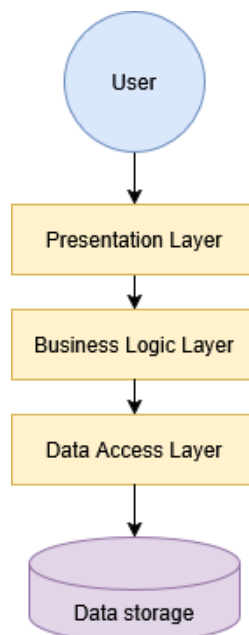


Рисунок 2.2 – Схема застосунку

Користувач буде взаємодіяти тільки з шаром, який відповідає за взаємодію. Цей шар буде залежати від внутрішньої логіки застосунку, а внутрішня логіка буде залежати від сховища інформації.

Важливо зауважити, що шар взаємодії та відображення буде залежати напряду та повністю від внутрішньої логіки, а ось внутрішня логіка не буде залежати від сховища, а натомість буде працювати з ним через спеціальний адаптер.

Такий підхід дозволить в майбутньому підключити інші сховища без змін в бізнес логіці застосунку. І також така схема дозволяє в майбутньому створити інше відображення, яке буде залежати від внутрішньої логіки.

Загально спроектована система є простою для розуміння та змін в майбутньому, що означає її низьку складність [3].

2.5 Основні сценарії використання

Під час проєктування системи було створено діаграму прецедентів – діаграму, яка зображує відношення між акторами, тобто сутностями, що взаємодіють з системою, та прецедентами, тобто описом поведінки системи.

У користувача є три основні варіанти використання системи:

- керування проєктами;
- перегляд та взаємодія зі схемою;
- вихід;

Перший це керування проєктами. Тут користувач зможе відкривати проєкти через імпортування з файлу на диску, зберігати в відкритий файл, зберігати в новий файл і закривати поточний проєкт. При цьому сценарії

Другий це перегляд схеми. Тут містяться всі варіанти взаємодії користувача з робочим полем, а також зі схемою:

- створення елементів;
- створення зв'язків;
- видалення елементів;

- видалення зв'язків;
- обирання елементів по області;
- переміщення елементів;
- загальне переміщення по робочому полю;
- зміну видів елементів;
- зміну типу на інвертований;
- симуляцію схеми.

І остання взаємодія користувача це вихід з програми. При виході застосунок буде перевіряти чи відкритий проєкт та запитувати у користувача чи потрібно його зберегти, як при звичайному закритті проєкту.

Діаграма прецедентів показана на рис. 2.3.



Рисунок 2.3 – Діаграма прецедентів

3 ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Архітектура та структура програми

На рис. 3.1 зображена діаграма залежностей модулів від інших модулів.

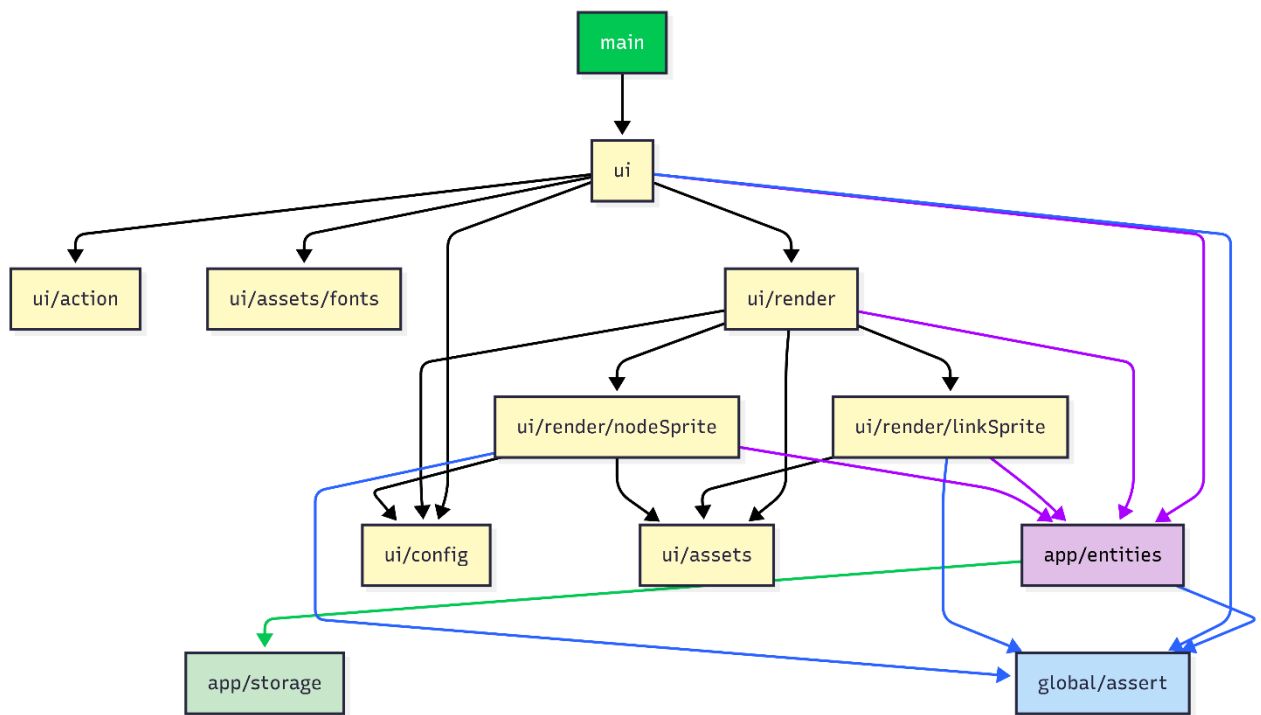


Рисунок 3.1 – Діаграма залежності модулів

Як можна побачити з діаграми на рис. 3.1, більшість модулів застосунку відповідають саме за взаємодію з користувачем. Внутрішня логіка знаходиться в одному модулі `app/entities`, а логіка збереження інформації знаходиться в модулі `app/storage`.

Також в застосунку присутній додатковий модуль `assert`, який дозволяє забезпечити інваріантивність.

Шар взаємодії та відображення поділений на основні модулі:

- модуль `action` відповідає за прослуховування подій від користувача та перетворення комбінацій натиснутих клавіш в зрозумілі програмі події;

- модуль `assets` містить в собі всі графічні елементи, які використовуються в програмі, а також допоміжні функції для роботи з ними. Модуль `assets/fonts` містить шрифти, які використовуються в застосунку;
- модуль `config` містить всі основні змінні, які використовуються для відображення;
- модуль `render` займається відображенням елементів на екран. Його під-модулі `linkSprite` та `nodeSprite` займаються відображенням зв'язків та елементів відповідно;
- головний модуль `ui` поєднує в собі всі модулі та містить головну логіку взаємодії, а саме створення об'єктів `ebitengine`, роботу з оновленнями та відображенням інформації на екран.

Логіка шару внутрішньої логіки знаходиться в одному модулі, оскільки вона є тісно пов'язаною та її складно відокремити без збільшення складності застосунку.

3.2 Використовувані моделі даних

Під час створення застосунку було створено 75 різних сутностей.

Всі сутності можна розділити на три основні групи, за шарами. Перші це сутності для шару візуалізації, наприклад, управління камерою або переміщенням елементів. Наступні це сутності для шару бізнес-логіки, наприклад, головні структури для елементів та зв'язків, а також логіка їх переміщення й симуляції. Також існують сутності для шару доступу до інформації.

Також сутності поділяються на приватні, тобто ті, до яких є доступ тільки зсередини модулю, і публічні, які можуть використовувати інші модулі. В Go приватні пишуться з маленької, а публічні з великої.

На рис. 3.2 показана діаграма всіх сутностей шару представлення, а на рис. 3.3 показана діаграма всіх сутностей шару внутрішньої логіки. На рис. 3.3 також показана головна сутність шару доступу до інформації.

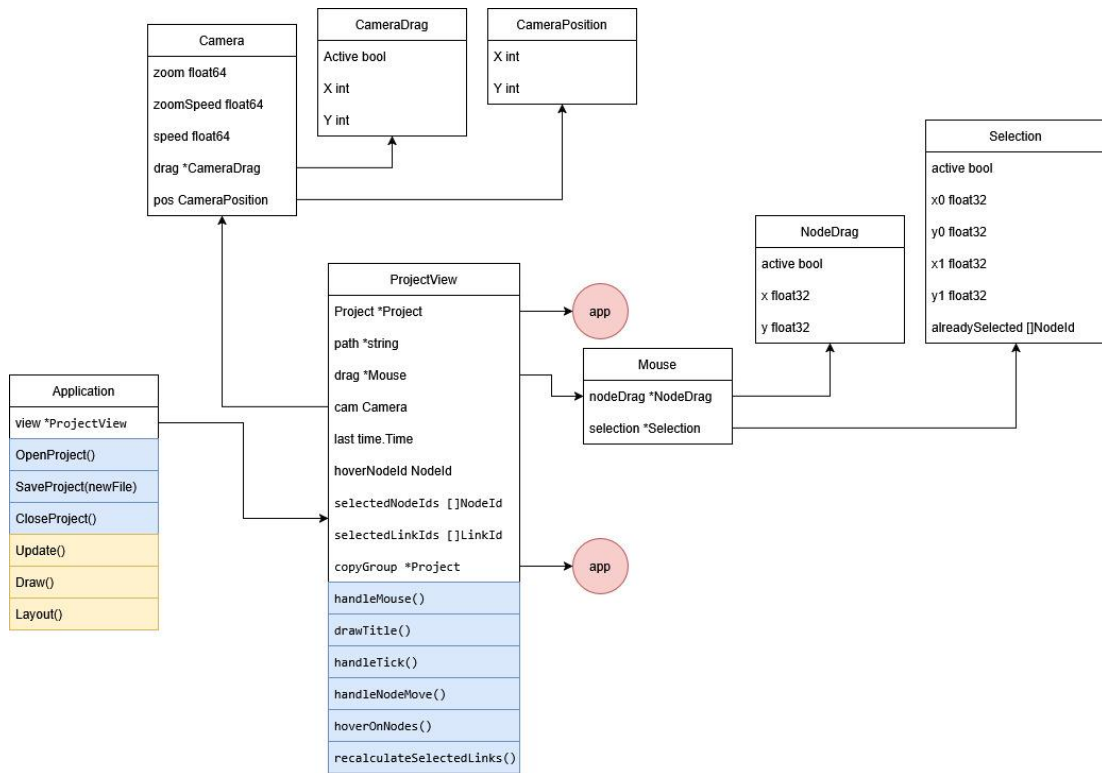


Рисунок 3.2 – Діаграма всіх сутностей шару взаємодії з користувачем

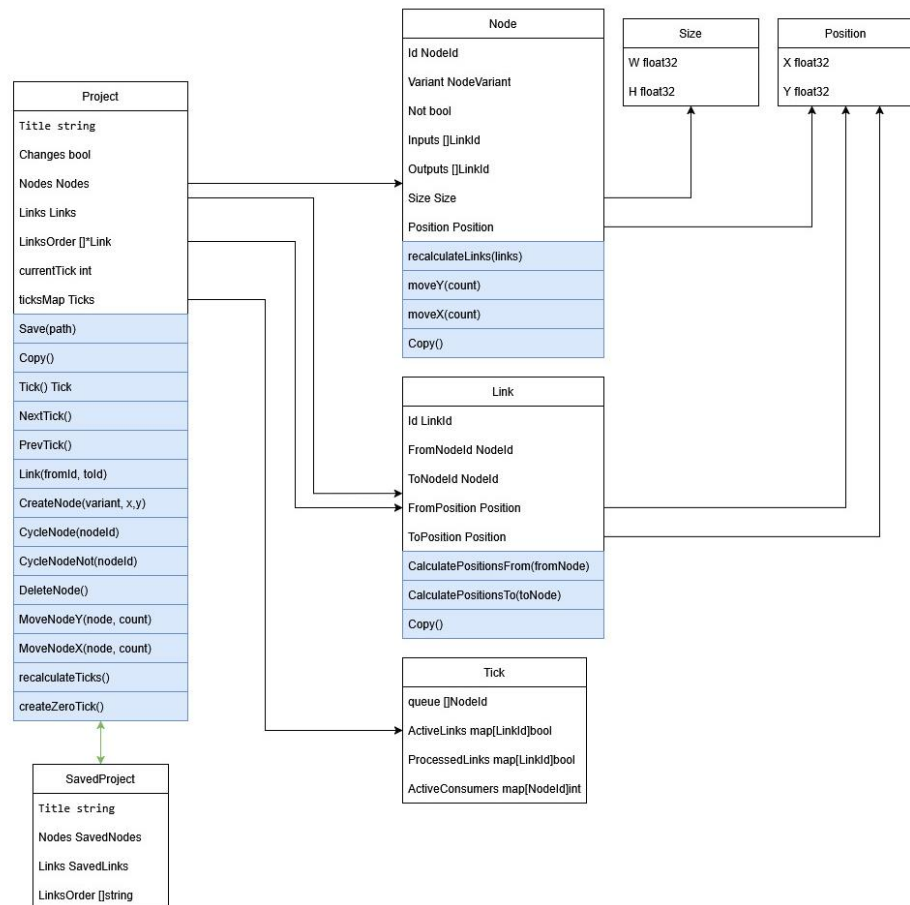


Рисунок 3.3 – Діаграма всіх сутностей шару внутрішньої логіки

Логічні елементи в програмі представлені під назвою Node. Вони мають свій унікальний ідентифікатор, який генерується на основі випадкових чисел. Варіант елемента означає його вид, в нього входять логічні вентиля AND, OR, XOR та BUFFER, а також вхідні елементи, які в програмній реалізації називаються Source, вони можуть бути активними та ні, а також вихідні елементи, які в програмі називаються Consumer, які можуть не мати стану, бути не активними та активними. Логічні елементи також мають інформацію про свій розмір, який визначається окремо для довжини та ширини й розраховується в кількості комірок, які будуть використані під час відображення елемента. І елементи знають про зв'язки які в них приходять, в полі Inputs, та які з них виходять, в полі Outputs. Поле Not означає чи є цей елемент інвертованою версією звичайного, і якщо так, то його розмір буде збільшуватись на один по довжині.

Логічні елементи мають декілька основних методів:

- метод recalculateLinks є приватним та дозволяє елементу перерахувати позиції зв'язків відносно елементу. Це потрібно, щоб правильно визначити позиції всіх зв'язків при створенні нових або переміщенні елементу;
- метод moveY переміщує елемент по осі Y;
- метод moveX переміщує елемент по осі X;
- метод Copy дозволяє створити повноцінну копію логічного елемента.

В застосунку також є стани логічних елементів, а саме стан за замовчуванням, стан коли на елемент було наведено, коли елемент є активним, тобто користувач його обрав, та коли на елементі є помилка. Додатково, в застосунку є стани для роз'ємів елементів, а саме коли роз'єм пустий, тобто в елементі не вистачає ще одного з'єднання, і стан помилки, коли з'єднань забагато.

Наступною головною сутністю є зв'язок, який в програмній реалізації називається Link. Зв'язок має свій унікальний ідентифікатор, знає з якого

елементу та в який елемент він йде, а також має інформацію про його позицію, з якої він повинен виходити та позицію, в яку він повинен приходити.

Зв'язки мають три методи:

- метод `CalculatePositionsFrom` дозволяє розрахувати позиції зв'язку відносно елементу, з якого цей зв'язок виходить;
- метод `CalculatePositionsTo` дозволяє розрахувати позиції зв'язку відносно елементу, з якого цей зв'язок входить;
- метод `Copy` створює повноцінну копію зв'язку.

Зв'язки також мають стани, а саме стан за замовчуванням, активний, тобто коли вихід елементу дорівнює 1, та обраний, коли користувач обрав з'єднання або обидва логічні елементи, між якими є це з'єднання.

Наступна сутність це `Tick`. Ця сутність представляє собою стан схеми в момент часу. Саме в ній зберігається інформація про те, які зв'язки наразі є активними, які зв'язки програма вже перевірила та які вихідні елементи мають який стан. Додатково в цій сутності зберігається черга елементів, які будуть перераховані при створенні наступного тіку.

Головною сутністю шару внутрішньої логіки є `Project`, який зберігає всю про внутрішні елементи в поточному проєкті, які потрібні для роботи застосунку. Ця сутність має 8 полів.

Поле `Title`, яке відповідає назві проєкту. Це поле відповідає назві файлу, в який збережений проєкт.

Поле `Changes`, яке означає, що в проєкті є незбереженні зміни. Якщо проєкт було тільки створено та він ще не має елементів, або ж користувач тільки відкрив проєкт, то це поле буде мати значення `false`. Якщо ж користувач зробив будь-які зміни з елементами, наприклад перемістив щось або створив/видалив, то це поле буде мати значення `true` до поки користувач не збереже проєкт знову.

Поле `Nodes` представляє собою хеш-таблицю всіх логічних елементів, що є в проєкті. Ключем виступає унікальний ідентифікатор елементу, а

значенням посилання на сам елемент. Використовується для швидкого отримання доступу до елементів по ідентифікатору.

Поле `Links` представляє собою хеш-таблицю всіх зв'язків. Ключем виступає унікальний ідентифікатор зв'язку, а значенням посилання на зв'язок. Використовується для швидкого доступу до зв'язку по його ідентифікатору.

Поле `LinksOrder` є масивом, який складається з посилань на зв'язки. Цей масив потрібен та використовується відображення зв'язків в одному порядку. Мова програмування Go не гарантує, що при проходженні по хеш-таблиці ми будемо отримувати значення в одному порядку, а тому, для коректного їх відображення, нам потрібно їх зберігати локально.

Поле `currentTick` є приватним та вказує на індекс поточного тіку. Застосунок підтримує наявність декількох тіків та можливості переміщуватись по ним, для, наприклад, створення анімації симуляції схеми.

Поле `ticksMap` є приватним та містить хеш-таблицю всіх тіків з їхніми індексами. Ключем виступає число, а значенням тік.

Також, оскільки `Project` є центральною сутністю в шарі застосунку, він має достатньо велику кількість методів та функцій, які з ним пов'язані.

Функція `OpenProject` дозволяє створити новий проєкт на основі шляху до файлу. Всередині ця функція викликає сховище, яке читає інформацію про файл і переводить його в `SavedProject`, який потім через спеціальний адаптер переводиться в `Project`.

Метод `Save` дозволяє зберегти поточний проєкт по переданому шляху. Всередині цей метод перетворює `Project` на `SavedProject`, а потім викликає сховище на запис файлу.

Метод `Copy` дозволяє створити повноцінну копію проєкта. Копія є глибокою, тобто далі ми можемо вільно змінювати інформацію, переміщувати елементи тощо без хвилювання, що ми змінимо оригінальний проєкт.

Метод `Tick` повертає поточний тік, використовується для відображення поточного стану. Ця функція існує щоб сховати внутрішню та не показувати хеш-таблицю та індекс поточно тіку.

Метод `NextTick` та метод `PrevTick` дозволяють перейти на наступний та попередній тіки відповідно.

Метод `Link` дозволяє створити зв'язок між двома елементами, ідентифікатори яких передаються в функцію. Якщо зв'язок все існує, то він буде навпаки видалений.

Метод `CreateNode` дозволяє створити новий елемент згідно його варіанту.

Метод `CycleNode` дозволяє змінити варіант елемента на наступний, при цьому враховуючи групу, в якій він знаходиться. Наприклад, якщо це вхідний елемент, то він може змінитись на іншу версію вхідного елемента, тобто з не активного на активну або навпаки. Якщо це елемент з групи логічних, то він може змінитись тільки на інший логічний елемент, але не на інвертовану. Вихідні елементи не змінюються.

Метод `CycleNodeNot` дозволяє інвертувати вихід елемента, наприклад, вентиль `AND` перетворить на `NAND` і навпаки. Працює тільки на логічних вентилях.

Метод `DeleteNode` дозволяє видалити елемент разом зі всіма зв'язками які входять та виходять з нього. Для видалення декількох елементів цей метод повинен викликатись для кожного елемента окремо.

Метод `MoveNodeX` та `MoveNodeY` дозволяють переміщувати елементи вздовж осей `X` та `Y` відповідно. Зміна позицій елементів повинна відбуватись тільки через ці методи, оскільки вони також змінюють поле `Changes`, яке відповідає за незбереженні зміни в проєкті.

Метод `recalculateTicks` дозволяє запустити логіку симуляції з врахуванням попереднього стану.

Метод `createZeroTick` дозволяє створити перший пустий тік, з яким далі буде запущена симуляція для отримання першого стану програми.

Сутність `SavedProject` представляє собою копію `Project` з переробленими типами полів та використовується для збереження проєкту.

Для перетворення Project в SavedProject та навпаки використовуються адаптери.

Було розглянути всі сутності шару застосунку, або ж шару внутрішньої логіки. Перейдемо до сутностей шару взаємодії з користувачем та відображення.

Головною сутністю на цьому шарі є Application, в ній зберігається поточний відкритий проєкт в полі view, а точніше обгортка для візуального представлення, та головні методи:

- метод OpenProject дозволяє відкрити новий проєкт за шляхом, який вибере користувач;
- метод SaveProject дозволяє зберегти поточний проєкт в файл, з якого він був відкритий або зберегти в новий;
- метод CloseProject дозволяє закрити поточний проєкт та створити пустий.

Наступною сутністю є обгортка над Project з шару застосунку, яка додає до нього всі необхідні поля й методи для взаємодії з користувачем – ProjectView. Ця сутність має досить велику кількість важливих полів.

Перше поле це Project, який є посиланням на Project з application шару.

Друге поле це шлях, по якому збережений поточний проєкт. Це поле є вказівником на рядок для того, щоб застосунок міг чітко відрізнити коли шлях є пустим, а коли в ньому немає значення, оскільки Go автоматично ставить стандартні значення для всіх типів.

Наступне поле це drag і це поле, в свою чергу, є вказівником на сутність Mouse, яка буде розглянута пізніше.

Поле cam містить в собі інформацію про камеру та буде також розглянуто пізніше.

Наступне поле це last, яке містить час останнього відображення інформації. Це поле використовується для розрахунку переміщення, оскільки швидкість переміщення елементів на робочому полі повинно завжди

залишатись однаковим не в залежності від швидкості комп'ютера користувача.

Далі йде поле `hoverNodeId`, яке містить унікальний ідентифікатор елемента, на який зараз навів мишку користувач. Це поле використовується, наприклад, для того, щоб змінювати варіант елемента.

В полі `selectedNodeIds` зберігається масив унікальних ідентифікаторів елементів, які наразі обрав користувач.

В полі `selectedLinkIds` зберігаються масив унікальних ідентифікаторів зв'язків. До списку обраних посилань також автоматично додаються посилання, якщо користувач обрав обидва елементи, між якими воно знаходиться.

В полі `copyGroup` зберігається інформація про елементи та зв'язки, які копіював користувач. Воно представляє собою проєкт, оскільки таким чином ми можемо пере використовувати методи та вже існуючі структури, що значно зменшить складність та повторюваність коду за рахунок спрощення системи, оскільки, як відомо, складність це будь-що, що робить систему складнішою для розуміння та змін [3].

Також ця сутність має методи.

Першим методом є `handleMouse`, який на основі позиції миші користувача, а також дій з нею, які розраховуються окремо, обирає елементи, обирає групи елементів, або ж переміщує ці елементи. Цей метод напряду пов'язаний з сутністю `Mouse`.

Наступний метод це `drawTitle`, який показує на екрані користувача назву поточного проєкту та чи збережений він.

Метод `handleTick` є обгорткою, яка на основі дій користувача викликає потрібні методи проєкту.

Метод `handleNodeMove` дозволяє користувачу переміщувати обрані елементи за допомогою клавіатури. Переміщення за допомогою миші реалізовано в `handleMouse`.

Метод `hoverOnNodes` перевіряє, чи не навівся користувач на елемент, і якщо так, то оновлює поле `hoverNodeId`.

Метод `recalculateSelectedLinks` знаходить зв'язки у яких обрано обидва елементи та додає їх до списку обраних.

Для роботи з камерою було створено сутність `Camera`. Ця сутність має:

- поле `zoom` яке містить значення поточного масштабування всіх зображень, які будуть відображатись на робочій області. Також шейдер для створення сітки потребує цього параметру;
- поле `zoomSpeed` означає швидкість з якою буде змінюватись значення `zoom` коли користувач буде прокручувати колесо миші;
- поле `speed` означає швидкість з якою буде переміщатись позиція камери, коли користувач буде змінювати її положення з клавіатури;
- поле `drag` містить в собі сутність `CameraDrag`;
- поле `pos` містить в собі позицію камери по X та Y осям, та використовується як здви́г при відображенні елементів.

Сутність `CameraDrag` містить в собі координати початку переміщення камери через мишку та затискання пробілу, а також поле, яке визначає чи це переміщення є активним. Для переміщення камери застосунок спочатку запам'ятовує її стартову позицію після віднімання позиції миші, а після, при переміщенні, змінює позицію камери на стартову з додаванням позиції миші. Таким чином ми можемо здвигати камеру залежно від того, наскільки ми здвинули мишку відносно початкових координат.

Сутність `Mouse` містить в собі два поля:

- поле `nodeDrag` містить в собі інформацію про елемент, який наразі переміщує користувач. Це поле має тип `NodeDrag`, тобто сутність яка містить поля початку та кінця переміщення та поле чи це переміщення є активним. При переміщенні декількох елементів розраховується позиція найвищого зліва;
- поле `selection` містить в собі інформацію про область, яку користувач хоче обрати мишкою. Тобто, користувач затискає ліву кнопку

миші в одній позиції та веде її іншу обираючи все, що знаходиться всередині неї. Це поле має тип Selection.

Сутність Selection має поле активності, яке означає, чи обирання елементів наразі активне, має поля для точки старту та закінчення області, а також поле alreadySelected, яке містить елементи, які були обрані до початку створення нової області. Це поле потрібно, щоб користувач міг затиснути shift та при виборі нової області додати нові елементи до вже обраних, а не тільки обрати нові.

3.3 Розробка модулів системи

Для реалізації симуляції в застосунку було реалізовано спеціальну систему тіків, де тік – це один стан системи. Кожен тік містить в собі інформацію про те, які зв'язки є активними, які вихідні елементи є активними, а також які зв'язки було перевірено.

Для розрахунку кожного тіку застосунок додає в чергу вхідні елементи. Алгоритм буде працювати до поки черга не буде вільна, або ж до поки не настане ліміт обчислень. На кожній ітерації алгоритм бере перший елемент з черги, обчислює його значення та після проходиться по вихідним зв'язкам і додає елементи з них в чергу.

Простіше кажучи, алгоритм працює як пошук в ширину, починаючи з вхідних вузлів і до кінця графу. Важливо зауважити, що алгоритм буде обчислювати стан елемента тільки якщо всіх вхідні зв'язки вже були перевірені та мають правильне значення.

При внесенні змін в схему застосунок замінює поточний тік на новий, який створюється на основі стану попереднього. Якщо реалізувати створення нового тіку з повним перерахуванням всіх станів зв'язків та вихідних елементів, то ми отримаємо неприємну ситуацію, коли ми не можемо реалізувати комірки пам'яті.

Наприклад, якщо ми хочемо реалізувати тригер, або flip-flop, елемент, який має два стабільні стани та може перемикатись між ними, нам потрібно, щоб алгоритм розумів попередній стан системи.

Застосунок одночасно зберігає не один тік, а їх набір. В поточній реалізації це потрібно для майбутньої реалізації елементів, що можуть змінювати свій стан з часом. Наприклад, для реалізації синхронізованого сигналу, або ж тактового сигналу, який постійно перемикається між 1 та 0 з заданим проміжком часу. Також використання набору тіків замість одного відкриває можливість майбутньої анімації того, яким чином було розраховано стан застосунку. Для створення такої анімації один тік повинен проходити по черзі елементів з попереднього тіку.

Основна графічна частина застосунку реалізована з використанням цифрової піксельної графіки. Всі графічні елементи зберігаються в одному png файлі, який після, при виконанні програми, розбивається на окремі картинки.

Логіка відображення схеми на робочому полі включає в себе три основні етапи, а саме відображення сітки, відображення зв'язків та відображення логічних елементів.

Для реалізації відображення сітки було створено шейдер на мові Kage, яку підтримує ebitengine. Ця мова шейдерів не є окремою мовою, а скоріше є надбудовою над Go для роботи з шейдерами. Вона потрібна оскільки вихідний шейдер для відеокарти буде залежати від середовища, на якому буде запускатись застосунок.

Іншим варіантом показати сітку на екрані є відображати окремі квадрати на деяких координатах, але такий варіант є дуже не оптимізованим, оскільки зображення при достатньому віддаленні може складатись з десятків тисяч таких маленьких квадратів [27].

Для відображення зв'язків спочатку потрібно мати початкове та кінцеве положення зв'язку. Ці положення розраховуються автоматично під час переміщення елементів або створення нових зв'язків. Далі, всі зв'язки, які виходять з одного елементу, додаються в спеціальну групу. Такі з'єднання

будуть відображатись разом для правильного відображення поворотів та пересічень. Група містить в собі координати початку, координати кінця, а також статус зв'язків цієї групи. Зв'язки можуть бути обраними користувачем, можуть бути активними, згідно схеми, та можуть мати стан за замовчуванням.

Група зв'язків спеціальним алгоритмом, який буде описано пізніше, переводиться набір векторів, які будуть відображатись на екрані користувача. Після цього, для кожної групи застосунок створює своє проміжне представлення в вигляді сітки, де кожен елемент сітки представляє собою майбутнє зображення. Ці зображення можуть бути, наприклад, прямою лінією, поворотом, прямою лінією з поворотом та пересіченням.

Після того, як було створено окреме представлення для групи, це представлення накладається на головне представлення, в якому будуть знаходитись всі елементи, та яке буде відображатись користувачу. Саме на цьому етапі розраховуються, наприклад, пересічення активних та не активних зв'язків, показуються помилки якщо два зв'язки з незалежних елементів накладаються один на одного, тощо. Після того, як загальне представлення було сформовано, виконується відображення всіх його елементів.

Обрані зв'язки відображаються таким самим чином, але шляхом накладання зображень обраних зв'язків на вже існуючі зображення інших станів. Це було реалізовано таким чином, оскільки така логіка є простішою в реалізації, оскільки нам не потрібно розраховувати всі ситуації, коли обраний елемент накладається на елементи інших станів. Такий варіант буде працювати завжди правильно, оскільки ми всього на всього відображаємо нові елементи по вже існуючим векторам.

Відображення логічних елементів реалізовано через проходження по всім елементам в пам'яті. На цьому етапі також перевіряється поточний стан елементів – якщо елемент обраний, то він буде відображений як обраний, якщо користувач навів на нього мишкою, то він буде показувати як наведений. Якщо ж елементі не вистачає вхідних зв'язків, то такий елемент буде показаний як помилковий разом з помилковим роз'ємом.

Відображення зображень зв'язків, а також зображень логічних елементів включає в себе переміщення та масштабування їх на екрані з використанням координат камера та поточного масштабу.

Для створення векторів було використано алгоритм ортогонального маршрутизування з'єднань, або ж Orthogonal Connector Routing. За допомогою цього алгоритму ми можемо зробити з'єднання з одного елементу до іншого, яке буде йти з кутами під 90 градусів, при цьому виходити з правильної позиції елементів та оминати їх.

Для реалізації цього алгоритму потрібно мати два логічних елементи. Спочатку з кожного з елементів створюються по три лінії горизонтально, якщо з'єднання буде горизонтальне, одна зверху, одна знизу та одна по місцю, де буде відбуватись підключення, та дві вертикальні, справа та зліва від елемента. Приклад таких ліній показаний на рис. 3.4 [28].

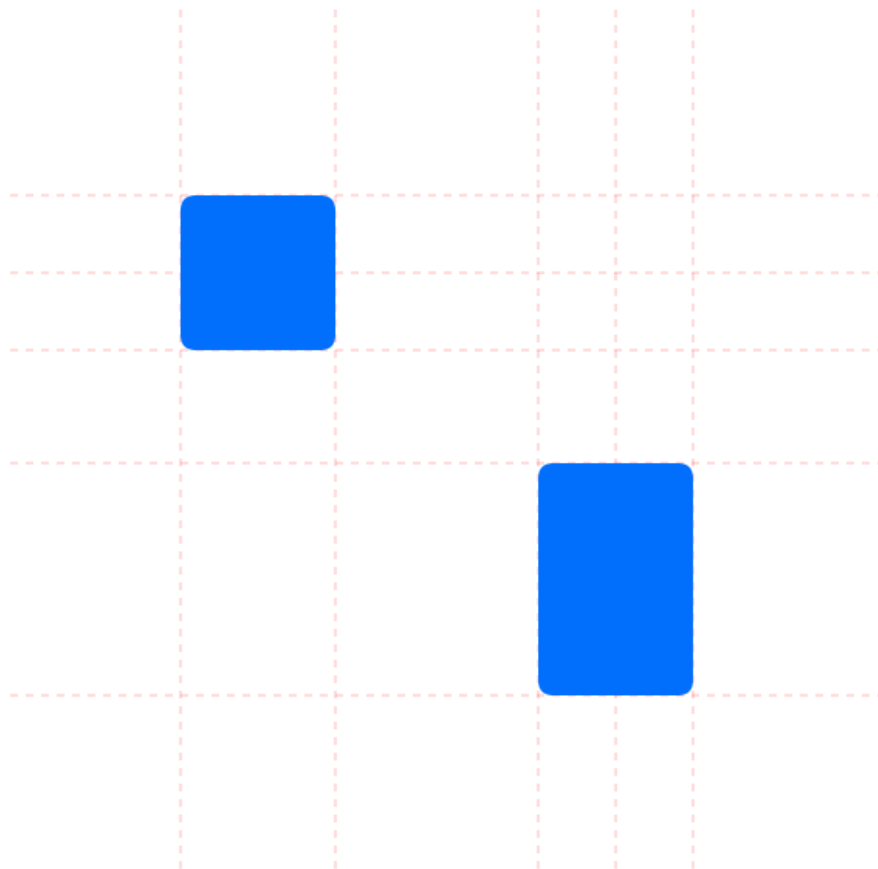


Рисунок 3.4 – Приклад лінійок для побудови матриці блоків [28]

Як можна побачити з рис. 3.4, при пересіченні лінії утворюють блоки. Наступним кроком є створити ці блоки на основі цих лінійок, а також визначити їх категорію на основі їх положення. Блоки, які знаходяться всередині будуть внутрішніми, блоки по кутах будуть кутовими, а блоки на сторонах будуть краями.

Після розбиття елементів на блоки, на кожному блоці створюються точки. На всіх блоках створюються точки по кутах. На внутрішніх створюються додаткові точки всередині блоків. На краях додаються також точки по центру зовнішньої сторони блоку.

Після створення точок ми повинні прибрати всі точки, які є дотичними до наших елементів. І також після ми додаємо точки для входу та виходу зв'язків. Приклад сітки блоків та точок показаний на рис. 3.5 [28].

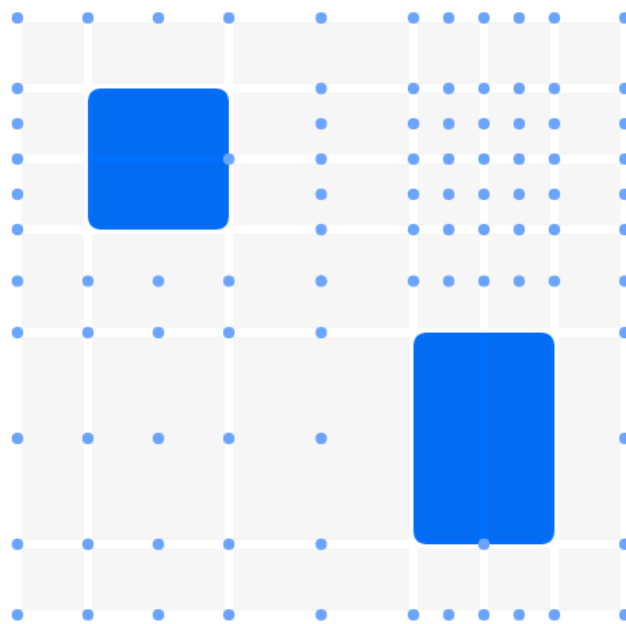


Рисунок 3.5 – Приклад сітки блоків та точок [28]

Далі на основі точок створюється граф. Між точками створюється з'єднання якщо вони можуть дістати одна до одної без пропуску інших точок. З'єднання між точками, які проходять через логічні елементи, не створюється, окрім точок, що виходять з логічних елементів.

Після створення графу виконується пошук найкоротшого шляху в графі, в цій роботі було використано стандартний алгоритм Дейкстри. Алгоритм було модифіковано таким чином, щоб уникнути великої кількості поворотів, а саме додаванням додаткової ваги при кожній зміні напрямку руху в графі. Приклад готового результату алгоритму показано на рис. 3.6 [28].

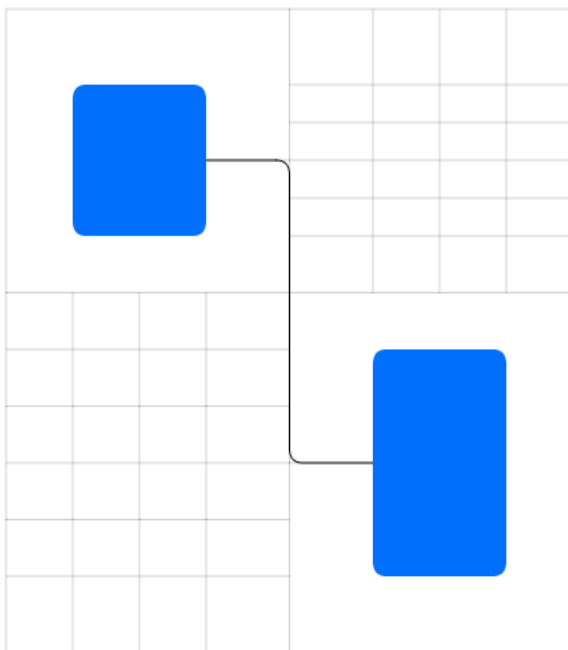


Рисунок 3.6 – Приклад результату виконання алгоритму [28]

Реалізація логіки роботи з проєктами в застосунку включає в себе три основні шари. Загальна схема показана на рис. 3.7.

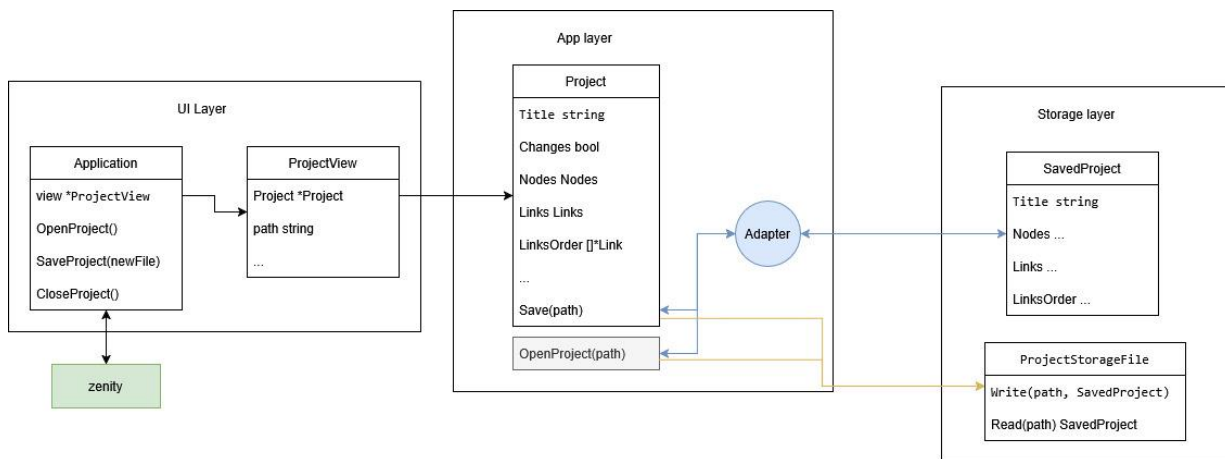


Рисунок 3.7 – Загальна схема шарів для роботи з проєктами

Першою та головною структурою в схемі є `Application`. Наразі в ньому, в полі `view`, зберігається поточний відкритий проєкт з яким взаємодіє користувач. В майбутньому в цю структуру можна буде додати підтримку для декількох відкритих проєктів одночасно, але при цьому користувач завжди буде в один момент часу зможе взаємодіяти тільки з одним проєктом.

Коли користувач намагається відкрити, зберегти або закрити проєкт викликаються саме методи на `Application`. Ці методи використовують поточний відкритий проєкт для отримання доступу до внутрішньої репрезентації проєкту, яка називається `Project` та зберігається на шарі внутрішньої логіки застосунку.

Далі, при збереженні проєкту його внутрішнє представлення переводиться в спеціальну структуру, яка буде зберігатись, `SavedProject`. Ця трансформація потрібна щоб відокремити внутрішню логіку проєкту від того, яким чином від буде зберігатись. При відкритті проєкту виконується зворотна операція – `SavedProject` через спеціальний адаптер переводиться в звичайний `Project`.

Перший шар, який відповідає за взаємодію з користувачем, потрібен, оскільки при відкритті проєктів потрібно розглядати декілька станів застосунку:

- коли користувач намагається закрити проєкт, в якому є зміни, застосунок покаже користувачу запитання, що при закритті проєкту поточні зміни не будуть збереженні;
- коли користувач намагається зберегти проєкт, але проєкт ще не зберігався, застосунок запитує шлях у користувача куди саме та під яким ім'ям зберігати проєкт;
- коли користувач намагається відкрити проєкт, то спочатку виконується закриття поточного, в тому числі з перевіркою на наявність не збережених змін.

Шар сховища був реалізований таким чином, щоб в майбутньому підтримувати декілька різних варіантів зберігати проєкти.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Призначення і умови використання програми

Призначенням цього застосунку є спрощення проєктування цифрових логічних схем для користувачів, а також надання можливості виконати симуляцію схеми для перевірки правильності її роботи.

Основною умовою використання програми є наявність комп'ютеру з однією з підтримуваних операційних систем, а саме Microsoft Windows 10, Windows 11, Mac OS 13 або Linux. Для стабільної та правильної роботи застосунку всі драйвера повинні бути оновлені до останньої версії.

4.2 Характеристики програми

Розмір вихідних файлів застосунку – 1 мегабайт.

Розмір скомпільованого файлу застосунку для системи Windows – 15 мегабайтів.

Розмір скомпільованого файлу застосунку для системи Linux - 3 мегабайти.

Потрібно враховувати, що в розмір скомпільованої програми включаються всі потрібні налаштування для роботи на визначеній операційній системі, а також включаються графічні елементи, шрифти.

Програма включає в собі набір основних функцій.

Можливість проєктування цифрових логічних схем через візуальний інтерфейс. Основна взаємодія користувача з програмою відбувається на робочому полі де знаходяться логічні елементи та зв'язки між ними.

Більшість взаємодій з програмою відбуваються через натискання комбінацій клавіш на клавіатурі, оскільки таким чином взаємодія з програмою є швидшою.

Застосунок надає можливість виконати симуляцію схеми, під час якої буде розраховано стан всіх зв'язків та вихідних елементів, які в ній знаходяться.

4.3 Інструкція по експлуатації програми

4.3.1 Звертання до програми

Для експлуатації програми користувачу потрібно скачати та локально запустити скомпільовану версію застосунку, яка підходить під використовувану операційну систему.

Програмісту, для внесення змін в застосунок, потрібно клонувати віддалений репозиторій з вихідним кодом. Для локальної розробки у розробника повинна бути встановлена відповідна версія мови програмування Go, її можна знайти в файлі `go.mod`, який описує один модуль Go. Також потрібно встановити необхідні пакети. Запуск, тестування, компіляція та форматування відбувається через компілятор Go.

4.3.2 Вхідні і вихідні дані

Вхідними даними застосунку є:

- інформація про схему від користувача;
- файл, в якому збережена інформація про проєкт.

Вихідними даними застосунку є:

- графічне відображення цифрової логічної схеми;
- графічне відображення схеми після симуляції.

4.4 Експлуатація програми

Програма складається з чотирьох основних візуальних елементів. На рис. 4.1 показаний приклад виконання програми.

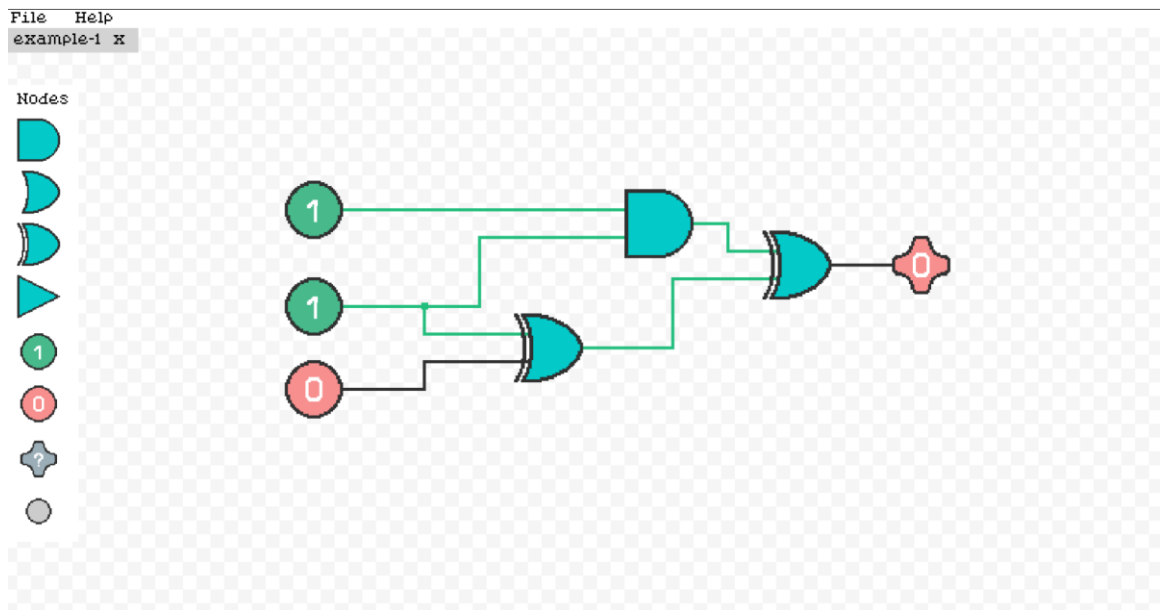


Рисунок 4.1 – Приклад виконання програми

Перший візуальний елемент це меню застосунку.

Меню має два основні пункти, а саме file та help.

В під-меню пункту file користувач може:

- відкрити новий проєкт;
- зберегти або зберегти проєкт в файл;
- закрити поточний проєкт;
- закрити застосунок, з попереднім закриттям проєкту.

При натисканні на пункт help користувачу відкриється вікно з повним описом всіх комбінацій клавіш та взаємодій з програмою.

Другим візуальним елементом є рядок, де наразі показується поточний відкритий проєкт.

Тут написана назва проєкту, яка відповідає назві файлу. Якщо ж проєкт ще не був збережений у файл, то він називається «Unknown».

Біля назви проєкту може відображатись зірочка, якщо в ньому є не збереженні зміни.

Біля назви, після зірочки, є кнопка, яка дозволяє закрити поточний проєкт. Якщо в ньому є незбереженні зміни, то застосунок запитає у користувача чи дійсно він хоче закрити проєкт, оскільки не збереженні в файл

зміни будуть втрачені. Якщо користувач відповідає так, то поточний проєкт закривається, а на його місці створюється пустий.

Вікно з комбінаціями клавіш показано на рис. 4.2.



Рисунок 4.2 – Вікно з комбінаціями клавіш

На рис. 4.3 показано приклад зірочки при не збережених змінах в проєкті та запитання до користувача щодо його закриття.

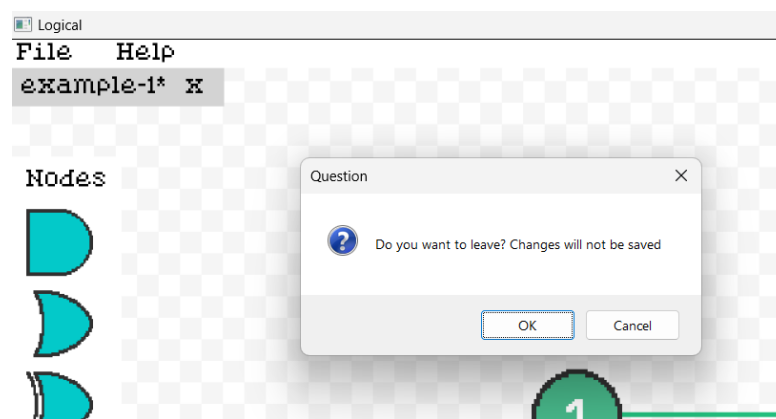


Рисунок 4.3 – Незбереженні зміни в проєкті

Наступним візуальним елементом є ліва панель, в якій знаходяться логічні елементи, елементи для інформації на вході, елементи для інформації

на виході та з'єднувач. Після натискання на один з елементів, він автоматично створиться по центру робочої області.

Останнім, та головним, візуальним елементом є робоча область. Саме тут користувач буде проєктувати та симулювати схеми. Головна взаємодія з цією областю відбувається за допомогою миші та через комбінації клавіш.

Для створення елементів користувач може скористатись або візуальною панеллю, або ж натиснути на спеціальні клавіші:

- клавіша «I» створює новий елемент для вхідної інформації;
- клавіша «O» створює новий елемент для вихідної інформації;
- клавіша «P» створює новий з'єднувач;
- клавіша «N» створює новий логічний елемент.

Для того щоб перетворити логічний елемент на його інвертовану версію користувачу потрібно навести мишу на потрібний елемент та натиснути на клавішу «S».

Якщо користувач вирішить змінити тип елементу в межах його групи, наприклад, з елементу «AND» на «OR», то для цього йому потрібно навести мишу на потрібний елемент та натиснути на клавішу «C». Після цього варіант елементу буде змінений на наступний в його групі. При використанні на логічних елементах вони змінюються в порядку «AND», «OR», «XOR» та «BUF», а при використанні на вхідних елементах їх значення змінюється на протилежне. Ця функція не працює з виходами та з'єднувачами.

Для створення з'єднань між двома елементами користувач потрібно обрати елемент, з якого буде починатись з'єднання, навести мишу на елемент, в який буде йти з'єднання та натиснути клавішу «L». Якщо таке з'єднання вже існує, то воно буде видалено.

Для зручної роботи з робочим полем воно є необмеженим в розмірі і в застосунку була реалізована можливість переміщуватись та масштабувати елементи на ньому. Для переміщення користувачу потрібно затиснути пробіл та затиснути ліву кнопку й перетягувати мишу. Для вертикального переміщення можна прокрутити коліщатко миші, а для горизонтального

потрібно додатково затиснути кнопку «Shift». Також можливе переміщення через натискання стрілочок на клавіатурі. Для зміни масштабу потрібно затиснути клавішу «Ctrl» та крутити коліщатко миші.

В застосунку реалізована функція обирання окремих елементів, а також обирання групи елементів за допомогою миші по області. Для цього потрібно затиснути ліву кнопку миші й виділити область. Всі елементи, що повністю знаходяться всередині цієї області будуть обрані разом з з'єднаннями, що йдуть від одного обраного елементу до іншого обраного.

Приклад обирання по області показано на рис. 4.4.

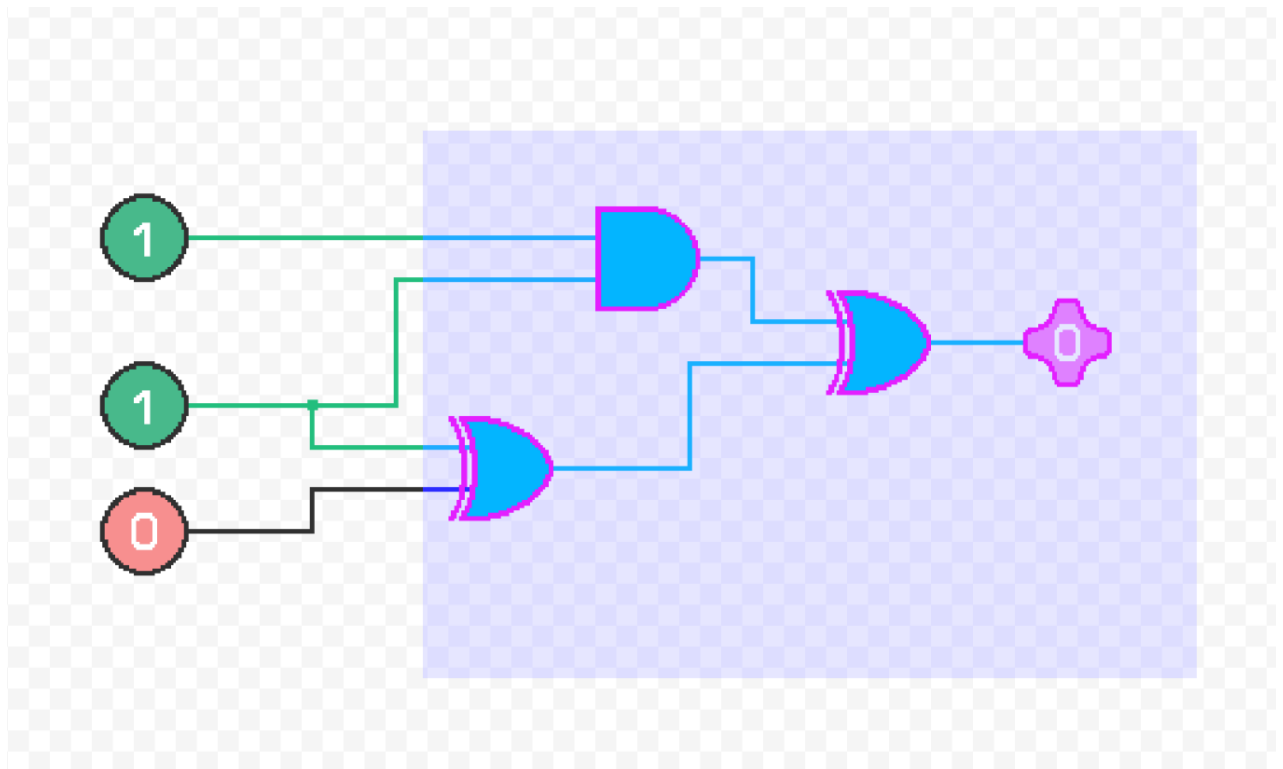


Рисунок 4.4 – Обирання по області

Також в застосунку можна додати елементи до вже обраних елементів. Для цього потрібно затиснути клавішу «Shift» та клікнути, або обрати по області потрібні елементи.

Над обраними елементами можна провести три основні дії, а саме перемістити, видалити, або скопіювати.

Для переміщення потрібно взяти один елемент з обраних та потягнути його з затиснутою лівою кнопкою миші. Якщо елемент до цього не був обраний, то він стане обраним та інші обрані елементи перестануть бути такими, якщо не натиснуто «Shift».

При видаленні елементів, яке відбувається через натискання на клавішу Delete, будуть видалені всі зв'язки, які в тому числі йдуть в ці елементи і йдуть з цих елементів, навіть якщо вони не обрані візуально. Це пов'язно з тим, що зв'язки не є незалежними та можуть існувати тільки між двома елементами.

При створенні копії застосунок зберігає в локальній пам'яті інформацію про всі обрані елементи, а також інформацію про всі зв'язки між ними. Для створення копії потрібно обрати елементи та натиснути «Ctrl + C». Після копіювання користувач може додати скопійовані елементи до проєкту не обмежену кількість разів. Для вставки використовується комбінація «Ctrl + V». На рис. 4.5 показаний приклад вставки елементів.

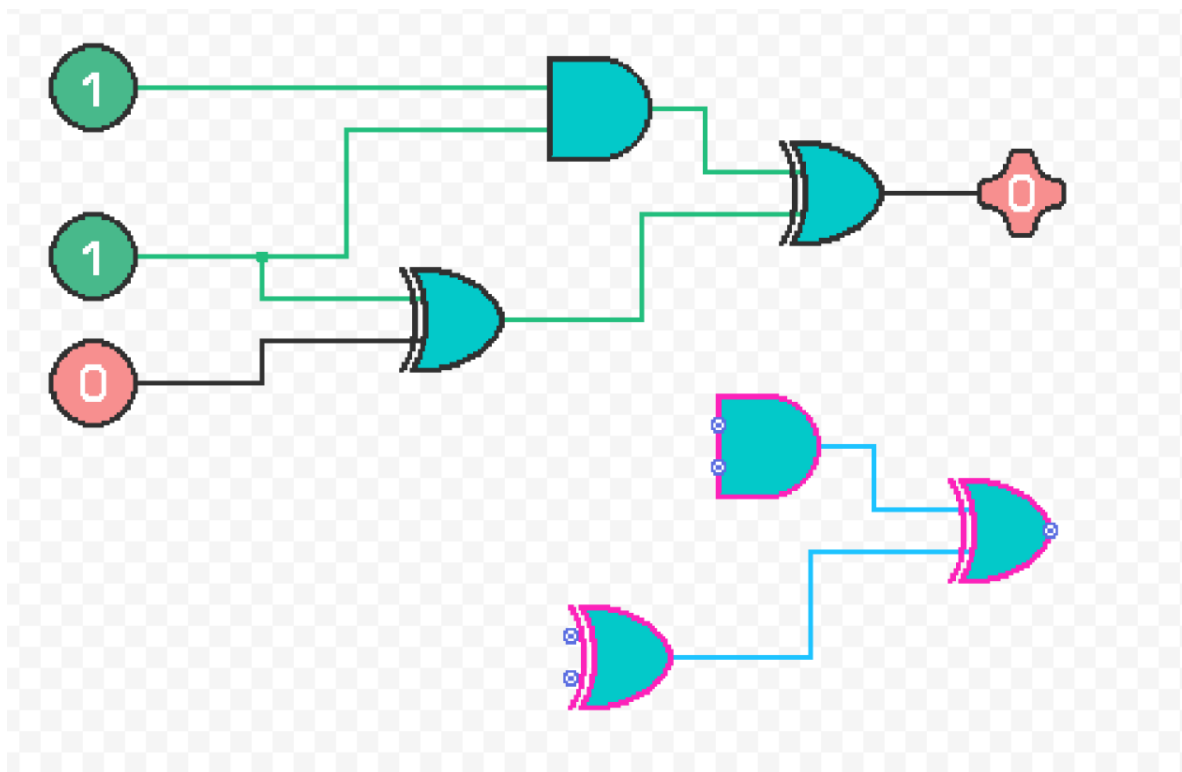


Рисунок 4.5 – Приклад вставки елементів

Для виконання симуляції користувачу потрібно натиснути на клавішу «Enter». Якщо симуляція має декілька кроків, то для переміщення по ним використовуються комбінації «Enter», щоб перейти далі, та «Shift + Enter», щоб перейти назад.

Після створення схеми її можна зберегти в файл. Для цього можна скористатись візуальним меню, або ж комбінацією клавіш «Ctrl + S», яка зберігає проєкт в існуючий файл, якщо він є, або комбінацію клавіш «Ctrl + Shift + S», яка зберігає проєкт завжди в новий файл. Для відкриття проєкту використовується комбінація клавіш «Ctrl + O».

На рис. 4.6 показаний приклад схеми повного додавання двох 4-х бітних бінарних числа.

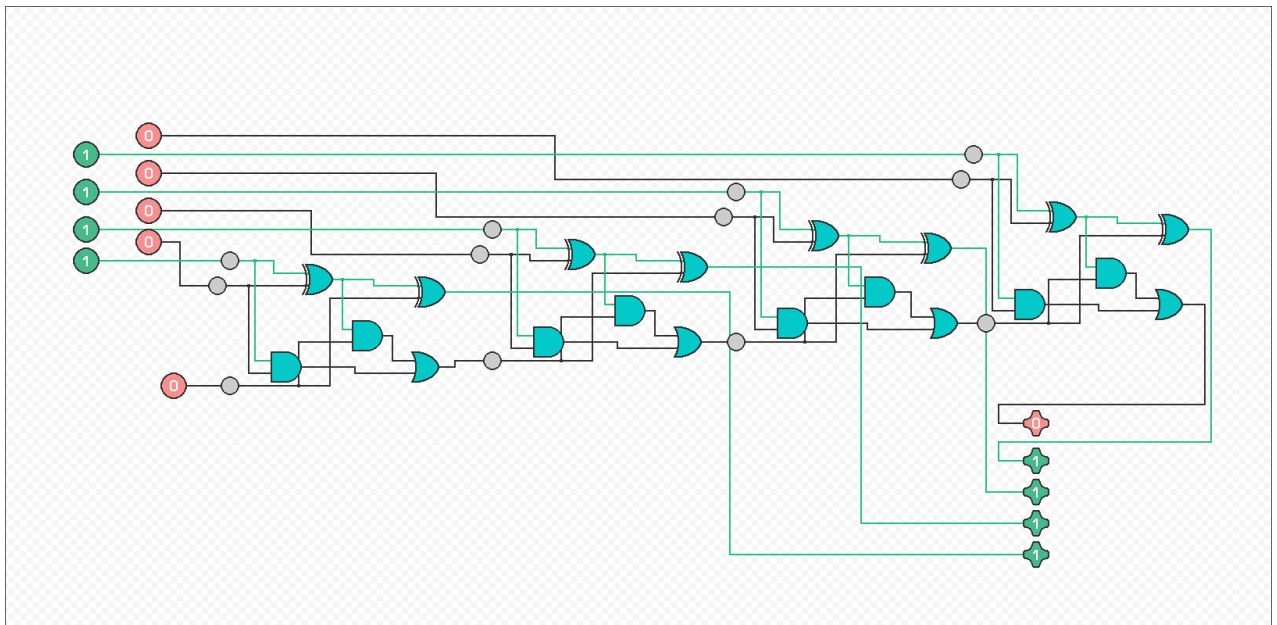


Рисунок 4.6 – Приклад додавання двох 4-х бітних бінарних числа

Логіка симуляції в тому числі підтримує цикли, оскільки вони є основою комірок пам'яті. На рис. 4.7 показано приклад створення тригера, засувки або ж SR-latch.

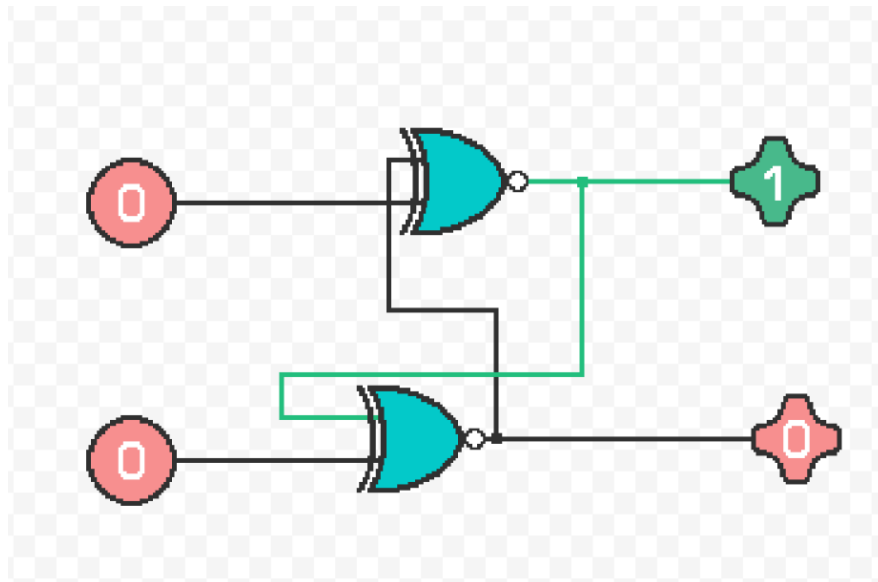


Рисунок 4.7 – Приклад створення SR-latch

Завдяки розробленій системі візуалізації схеми, користувацькому інтерфейсу та нескінченному робочому полю застосунок зменшує трудомісткість проєктування цифрових логічних схем.

А завдяки високій швидкодії та наявності великої кількості функцій, які працюють через визначені комбінації клавіш застосунок також гарантує зменшення часу для проєктування цих схем.

4.5 Тестування програми

Для забезпечення якості та надійності розробленої програми була проведена ретельна перевірка його роботи.

Першим етапом тестування є перевірка його функціональності. На цьому етапі було перевірено роботу всіх візуальних елементів, з якими може взаємодіяти користувач, перевірку всіх комбінацій клавіш, а також перевірку роботи симуляції в цілому.

Під час перевірки знайдено одну проблему, коли елементи знаходяться дуже близько один до одного, і при цьому на різних рівнях, що зв'язок між ними показується через додаткове коло.

Ця проблема полягає в тому, яким чином формуються опорні точки при побудові зв'язків та не є критичною для застосунку.

Приклад цієї проблеми показаний на рис. 4.8.

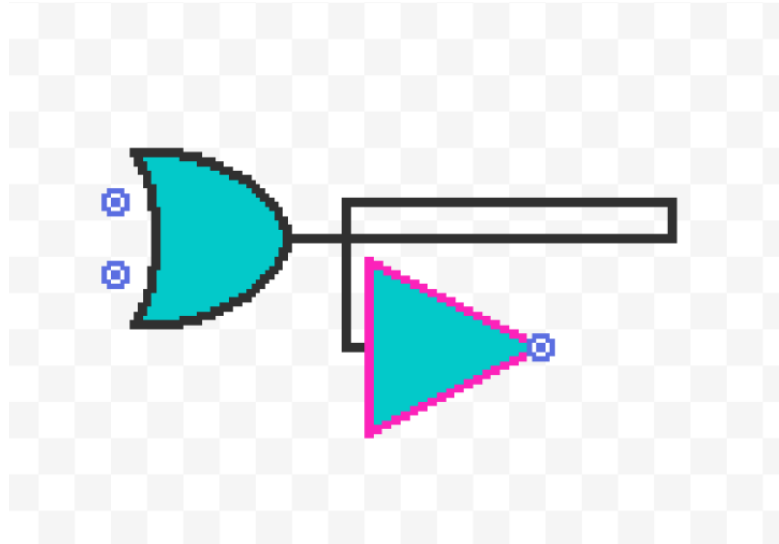


Рисунок 4.8 – Приклад проблеми з візуалізацією зв'язку коли елементи дуже близько один до одного

Наступним етапом є тестування зручності використання програми. На цьому етапі було перевірено весь функціонал на зручність та зрозумілість з точки зору користувача. В результаті перевірок знайдено, що в програмі не вистачає функції відміни дій, тобто undo.

Знайдено, що в програмі не вистачає можливості переміщувати зв'язки вручну, тобто змінювати їх позиції не залежно від позицій елементів, з якими вони пов'язані. Ці недоліки впливають на зручність використання, але є скоріше доповненням до основного функціоналу, а ніж його частиною.

Було знайдено, що при виникненні нескінченного циклу в схемі логіка симуляції припиняє свою роботу на елементах, які йдуть після елементів, які знаходяться в циклі. Ця поведінка є очікуваною і правильною, але може бути не до кінця зрозумілою користувачу. В майбутньому до застосунку можна додати спеціальний зв'язок, який буде означати помилку, і показувати його при нескінченних циклах.

Приклад схеми з нескінченним циклом показаний на рис. 4.9.

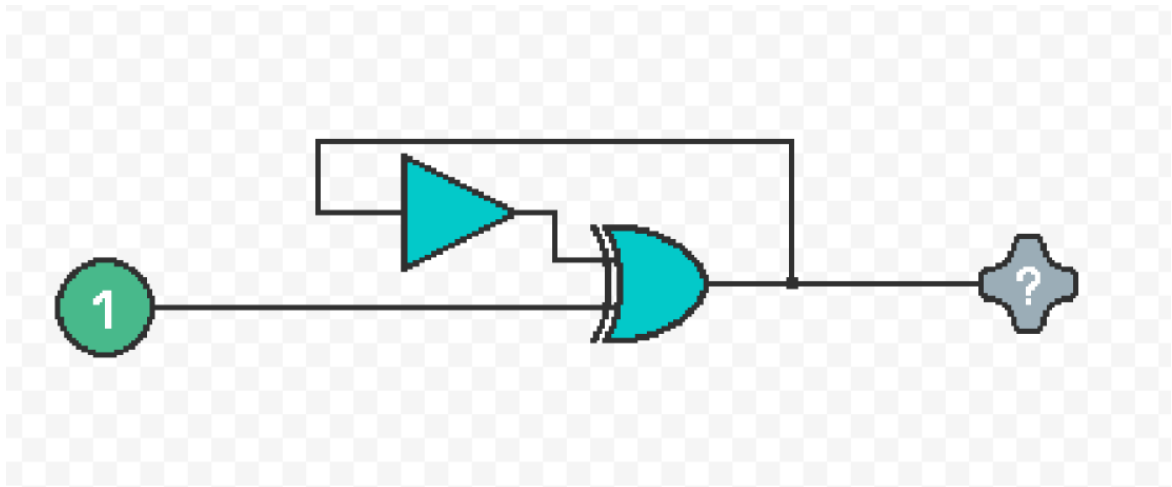


Рисунок 4.9 – Приклад схеми з нескінченним циклом

Під час тестування безпеки було визначено, що застосунок не має доступу до мережі інтернет, а тому не може передавати особисті дані користувача.

Застосунок має можливість читати та змінювати файли в системі, під час тестування було визначено, що він змінює та читає тільки файли, які обирає користувач й не змінює будь-які інші.

Під час перевірки не було знайдено жодного порушення внутрішніх умов застосунку, які повинні бути завжди правдою – інваріантів.

Для використання застосунку його не потрібно запускати від імені адміністратора, а тому він не має доступу до системних налаштувань.

Для тестування сумісності було створено дві скомпільовані версії застосунку:

- перша операційних систем Windows, перевірка відбувалась на Windows 11 версії 23H2(OS Build 22631.3085);
- друга для операційних систем Linux, перевірка відбувалась на системі з Arch Linux з kernel 7.0.3-arch1-1, зі встановленим hyprland та оновленими драйверами nvidia-open.

ВИСНОВКИ

В результаті виконання цієї дипломної роботи було створено кросплатформний застосунок з інтерактивною симуляцією графових моделей, який зменшує час та трудомісткість проектування цифрових логічних схем.

В результаті аналізу предметної області було окреслено головні поняття, визначено об'єкт і предмет дослідження, встановлено мету дослідження, оглянуто існуючі програмні рішення, встановлено вимоги до надійності та проєктованої системи, а також визначені умови роботи та розповсюдження програми.

Обґрунтовано середовища функціонування, розробки та програмні засоби програми, визначено структуру програми та спроєктовано сценарії використання.

Описані всі модулі програми з описом технологій які були обрані під час проєктування та основними рішеннями щодо їх реалізації.

Розроблено інструкцію з експлуатація для користувачів та проведено повноцінне тестування застосунку, в результаті якого було визначено, що застосунок відповідає всім поставленим умовам і не має дефектів в його роботі, але має простір для покращення в майбутньому.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Сорока К. Основи теорії систем і системного аналізу: Навч. посібник. Харків : ХНАМГ, 2004. 291 с.
2. Радельчук Г.І. Порівняльний аналіз сучасних методологій розробки програмного забезпечення. URL: https://www.rusnauka.com/40_OINBG_2014/Informatica/3_182487.doc.htm (дата звернення: 20.04.2026).
3. Ousterhout J. K. A Philosophy of Software Design. Yaknyam Press, 2018. 190 p.
4. What is graph-based data model. ITU Online Training. URL: <http://www.ituonline.com/tech-definitions/what-is-graph-based-data-model> (date of access: 25.04.2026).
5. Grokking Algorithms: An illustrated guide for programmers and other curious people. Manning Publications, 2016. 256 с.
6. Logic circuit diagrams. Ada Computer Science. URL: https://adacomputerscience.org/concepts/boolean_draw_circuit_diagram (date of access: 25.04.2026).
7. Logic gates. Ada Computer Science. URL: https://adacomputerscience.org/concepts/boolean_logic_gates (date of access: 25.04.2026).
8. Al-Busaidi S., Ahmad A. Free Open Source Software Logisim – A Perfect Tool for Teaching and Learning of Digital Logic Circuit Design Course – Experience and Status. 2019.
9. Burch C. Logisim. URL: <https://cburch.com/logisim/qanda.html> (date of access: 26.04.2026).
10. Neemann H. Digital. URL: https://wcours.gel.ulaval.ca/2021/a/GIF1002/default/6travaux/Doc_English2019.pdf (date of access: 26.04.2026).

11. Digital. GitHub, Digital: A digital logic designer and circuit simulator. URL: <https://github.com/hneemann/Digital> (date of access: 26.04.2026).
12. Tynjala J. Logicly: a logic gate simulator built in Flex. Josh Talks Flash Archive. URL: <https://joshblog.net/2008/11/03/logicly-a-logic-gate-simulator-built-in-flex/> (date of access: 26.04.2026).
13. Tynjala J. Logicly. A logic circuit simulator for Windows and macOS - logic gates, flip-flops, computer architecture, electronics, integrated circuits. URL: <https://logic.ly> (date of access: 28.04.2026).
14. Invariant. MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Glossary/Invariant> (date of access: 26.04.2026).
15. Target S. Thoughts on Go vs. Rust vs. Zig. URL: <https://sinclairtarget.com/blog/2025/08/thoughts-on-go-vs.-rust-vs.-zig> (date of access: 26.04.2026).
16. Zig, Rust, Go?. DEV Community. URL: https://dev.to/dev_tips/zig-rust-go-i-tried-3-low-level-languages-and-heres-what-im-sticking-with-4gpp (date of access: 26.04.2026).
17. Pike R. The Best Programming Advice I Ever Got. InformIT. URL: <https://www.informit.com/articles/article.aspx?p=1941206> (date of access: 02.05.2026).
18. Top 10 Programmers in the World of All Time. The Crazy Programmer. URL: <https://www.thecrazyprogrammer.com/2014/02/the-top-10-greatest-programmers-in-the-world-of-all-time.html> (date of access: 03.05.2026).
19. Ebitengine - A dead simple 2D game engine for Go. Ebitengine Documentation. URL: <https://ebitengine.org> (date of access: 02.05.2026).
20. Cruces N. Zenity dialogs for Golang, Windows, macOS. GitHub. URL: <https://github.com/ncruces/zenity> (date of access: 02.05.2026).
21. Aseprite. Aseprite - Animated sprite editor & pixel art tool. URL: <https://www.aseprite.org> (date of access: 05.05.2026).
22. Results from the 2025 Go Developer Survey. The Go Programming Language. URL: <https://go.dev/blog/survey2025> (date of access: 02.05.2026).

23. Go Wiki: Editors and IDEs for Go. The Go Programming Language. URL: <https://go.dev/wiki/IDEsAndTextEditorPlugins> (date of access: 02.05.2026).
24. GoLand features. JetBrains. URL: <https://www.jetbrains.com/go/features> (date of access: 05.05.2026).
25. Lyu Y. NeoVim setup for golang programming. Medium. URL: <https://medium.com/@yanglyu5201/neovim-setup-for-golang-programming-68ebf59336d9> (date of access: 05.05.2026).
26. Raiborde J. Excessive Extensions. Medium. URL: <https://joshraiborde.medium.com/excessive-extensions-b2683ce7235e> (date of access: 05.05.2026).
27. Shader. Ebitengine. URL: <https://ebitengine.org/en/documents/shader.html> (date of access: 08.05.2026).
28. Routing Orthogonal Diagram Connectors in JavaScript. Medium. URL: <https://medium.com/swlh/routing-orthogonal-diagram-connectors-in-javascript-191dc2c5ff70> (date of access: 03.05.2026).

ДОДАТОК А
ФРАГМЕНТ ТЕКСТУ ПРОГРАМИ

Файл main.go:

```
package main

import "github.com/ltlaitoff/logical/ui"

func main() {
    ui.Run()
}
```

Файл app\entities\base.go:

```
package entities

type Size struct {
    W float32
    H float32
}

type Position struct {
    X float32
    Y float32
}

type InputNodeInfo struct {
    Variant    NodeVariant
    Position   Position
    Size       Size
}

// NOTE: Temporally data structure, should be removed in future
type InputData struct {
    Links [][]NodeId
    Nodes map[NodeId]InputNodeInfo
}

// Tick represent one state of application. Each new tick can be generated
// based on previous tick state
// Application calculated state, like active linkSprite, state of nodes and
// other
// is saved in **Ticks**
// Zero-tick contains only sources
type Tick struct {
    // Sequence of nodes, that should be processed to get new tick information
    Queue []NodeId

    // Series of binary active [entities.Link] in current tick
    // If link is not in set, then it's not active
    // If link was active in previous tick it will be active in next too,
    // FUTURE: Application do not support loops right now
    ActiveLinks map[LinkId]bool

    // Set of already linkSprite, that has some real value, that we can use
    // If link is not processed, then value it have was not calculated
    //
    // If link is not processed, then Node, which depend on this link will
    // not be processed too
    ProcessedLinks map[LinkId]bool

    // Represent a state of consumers
    // We need this to correctly show it to the user, because it's only one
    // element that can switch
    // Variable has value as int, where:
    // - 0 - Not defined. Default value which go will get if key is not in
    map
```

```

        // - 1 - Active
        // - 2 - Not active
        ActiveConsumers map[NodeId]int
    }

    type Ticks = map[int]Tick

Файл app\entities\link.go:
package entities

import (
    "slices"
)

type Coordinates struct {
    X0 int
    Y0 int
    X1 int
    Y1 int
}

type LinkVectors struct {
    Changed      bool
    Coordinates []Coordinates
}

// LinkId describes unique link identifier
type LinkId string

type LinkState string

type Link struct {
    Id LinkId

    FromNodeId NodeId
    ToNodeId   NodeId

    FromPosition Position
    ToPosition   Position
    Vectors      LinkVectors
}

type Links = map[LinkId]*Link
type LinkStates map[LinkId]LinkState

const (
    LinkStateDefault LinkState = "DEFAULT"
    LinkStateActive   LinkState = "ACTIVE"
    LinkStateSelected LinkState = "SELECTED"
)

type LinkStateVectors struct {
    State      LinkState
    Coordinates []Coordinates
}

func (l *Link) RecalculatePositions(nodes Nodes) {
    fromNode := nodes[l.FromNodeId]
    toNode   := nodes[l.ToNodeId]

    l.calculatePositionsFrom(fromNode)
    l.calculatePositionsTo(toNode)
}

```

```

        //if l.Vectors.Changed == false {
        l.calculateVectors(fromNode, toNode)
        //} else {
        //    l.updateLinkFromToVectors()
        //}
    }

    func (l *Link) calculatePositionsFrom(fromNode *Node) {
        x0 := fromNode.Position.X + fromNode.Size.W/2.0
        y0 := fromNode.Position.Y + fromNode.Size.H/2.0
        l.FromPosition = Position{X: x0, Y: y0}
    }

    func (l *Link) calculatePositionsTo(toNode *Node) {
        x1 := toNode.Position.X + toNode.Size.W/2.0

        inputsCount := len(toNode.Inputs)
        if inputsCount < NodeVariantConnections[toNode.Variant] {
            inputsCount = NodeVariantConnections[toNode.Variant]
        }

        linkIndex := float32(slices.Index(toNode.Inputs, l.Id))
        oneInputSize := toNode.Size.H / float32(inputsCount)

        y1 := toNode.Position.Y + oneInputSize/2.0 + oneInputSize*linkIndex
        l.ToPosition = Position{X: x1, Y: y1}
    }

    func (l *Link) Copy() *Link {
        copyLink := Link{
            Id:          l.Id,
            FromNodeId:   l.FromNodeId,
            ToNodeId:     l.ToNodeId,
            FromPosition: l.FromPosition,
            ToPosition:   l.ToPosition,
        }

        return &copyLink
    }
}

Файл app/entities/linkCoordinates.go:
package entities

import (
    "container/heap"
    "math"
    "slices"
    "sort"
)

type PointTwo struct {
    x int
    y int
}

type BlockGrid map[int]map[int]Coordinates

const gridSpace = 4

//func (l *Link) updateLinkFromToVectors() {
//    from := PointTwo{
//        x: int(math.Floor(float64(l.FromPosition.X))),
//        y: int(math.Floor(float64(l.FromPosition.Y))),

```

```

//      }
//      to := PointTwo{
//          x: int(math.Floor(float64(l.ToPosition.X))),
//          y: int(math.Floor(float64(l.ToPosition.Y))),
//      }
//
//      l.Vectors.Coordinates[0].X0 = from.x
//      l.Vectors.Coordinates[0].Y0 = from.y
//      l.Vectors.Coordinates[len(l.Vectors.Coordinates)-1].X1 = to.x
//      l.Vectors.Coordinates[len(l.Vectors.Coordinates)-1].Y1 = to.y
//  }

func (l *Link) calculateVectors(fromNode *Node, toNode *Node) {
    fromNodeSize := getNodeSize(fromNode)
    toNodeSize := getNodeSize(toNode)

    gSize := &Coordinates{
        X0: min(fromNodeSize.X0, toNodeSize.X0) - gridSpace,
        Y0: min(fromNodeSize.Y0, toNodeSize.Y0) - gridSpace,
        X1: max(fromNodeSize.X1, toNodeSize.X1) + gridSpace,
        Y1: max(fromNodeSize.Y1, toNodeSize.Y1) + gridSpace,
    }

    //debugRect(gSize.X0-1, gSize.Y0-1, gSize.X1+1, gSize.Y1+1, debugColor)

    grid := generateGrid(l, fromNodeSize, toNodeSize, gSize)
    //debugGrid(grid)

    points := generatePoints(grid, fromNodeSize, toNodeSize)
    fromPoint := PointTwo{
        x:      int(math.Floor(float64(fromNode.Position.X +
fromNode.Size.W))),
        y: int(math.Floor(float64(l.FromPosition.Y))),
    }
    toPoint := PointTwo{
        x: int(math.Floor(float64(toNode.Position.X - 1))),
        y: int(math.Floor(float64(l.ToPosition.Y))),
    }
    points = append(points, fromPoint, toPoint)
    //debugPoints(points)

    graph, fromGraphNode, toGraphNode := createGraph(points, fromNodeSize,
toNodeSize, fromPoint, toPoint)

    //fmt.Println("From point", fromPoint)
    //fmt.Println("To point", toPoint)
    graphPath := dijkstra(graph, fromGraphNode, toGraphNode)
    path := []PointTwo{}

    path = append(path, PointTwo{
        x: int(math.Floor(float64(l.FromPosition.X))),
        y: int(math.Floor(float64(l.FromPosition.Y))),
    })

    path = append(path, graphPath...)
    path = append(path, PointTwo{
        x: int(math.Floor(float64(l.ToPosition.X))),
        y: int(math.Floor(float64(l.ToPosition.Y))),
    })

    simplified := simplifyPath(path)
    l.Vectors = LinkVectors{
        Changed:      false,
        Coordinates: pathToCoordinates(simplified),
    }
}

```

```

    }
}

func getNodeSize(node *Node) *Coordinates {
    return &Coordinates{
        X0: int(math.Floor(float64(node.Position.X))),
        Y0: int(math.Floor(float64(node.Position.Y))),
        X1: int(math.Floor(float64(node.Position.X + node.Size.W))),
        Y1: int(math.Floor(float64(node.Position.Y + node.Size.H))),
    }
}

func generateGrid(
    link *Link,
    fromNodeSize *Coordinates,
    toNodeSize *Coordinates,
    gSize *Coordinates,
) BlockGrid {
    verticals := []int{
        fromNodeSize.X0,
        fromNodeSize.X1,
        toNodeSize.X0,
        toNodeSize.X1,
    }
    horizontals := []int{
        fromNodeSize.Y0,
        int(math.Floor(float64(link.FromPosition.Y))),
        fromNodeSize.Y1,

        toNodeSize.Y0,
        int(math.Floor(float64(link.ToPosition.Y))),
        toNodeSize.Y1,
    }
    sort.Ints(verticals)
    sort.Ints(horizontals)

    grid := make(BlockGrid)

    lastX := gSize.X0
    lastY := gSize.Y0
    col := 0
    row := 0

    for _, y := range horizontals {
        for _, x := range verticals {
            if _, ok := grid[row]; !ok {
                grid[row] = make(map[int]Coordinates)
            }

            grid[row][col] = Coordinates{X0: lastX, Y0: lastY, X1: x, Y1:
y}

                col++
                lastX = x
            }

            grid[row][col] = Coordinates{X0: lastX, Y0: lastY, X1: gSize.X1,
Y1: y}

            lastX = gSize.X0
            lastY = y

            row++
            col = 0
        }
    }
}

```

```

lastX = gSize.X0

for _, x := range verticals {
    if _, ok := grid[row]; !ok {
        grid[row] = make(map[int]Coordinates)
    }

    grid[row][col] = Coordinates{X0: lastX, Y0: lastY, X1: x, Y1:
gSize.Y1}
    col++
    lastX = x
}

grid[row][col] = Coordinates{X0: lastX, Y0: lastY, X1: gSize.X1, Y1:
gSize.Y1}

return grid
}

func generatePoints(
    grid BlockGrid,
    fromNodeSize *Coordinates,
    toNodeSize *Coordinates,
) []PointTwo {
    maxRow := len(grid) - 1
    maxCol := len(grid[0]) - 1

    points := make([]PointTwo, 0)

    for row := range grid {
        firstRow := row == 0
        lastRow := row == maxRow

        for col := range grid[row] {
            el := grid[row][col]

            x0 := el.X0
            x1 := el.X1 - 1
            y0 := el.Y0
            y1 := el.Y1 - 1

            centerX := x0 + (x1-x0)/2
            centerY := y0 + (y1-y0)/2
            center := PointTwo{x: centerX, y: centerY}

            topLeft := PointTwo{x: x0, y: y0}
            top := PointTwo{x: centerX, y: y0}
            topRight := PointTwo{x: x1, y: y0}

            bottomLeft := PointTwo{x: x0, y: y1}
            bottom := PointTwo{x: centerX, y: y1}
            bottomRight := PointTwo{x: x1, y: y1}

            left := PointTwo{x: x0, y: centerY}
            right := PointTwo{x: x1, y: centerY}

            firstCol := col == 0
            lastCol := col == maxCol

            tl := firstRow && firstCol
            tr := firstRow && lastCol
            bl := lastRow && firstCol
            br := lastRow && lastCol

```

```

        if t1 || tr || bl || br {
            points = append(points,
                topLeft,
                topRight,
                bottomLeft,
                bottomRight,
            )
        } else if firstRow {
            points = append(points,
                topLeft,
                top,
                topRight,
            )
        } else if lastRow {
            points = append(points,
                bottomLeft,
                bottom,
                bottomRight,
            )
        } else if firstCol {
            points = append(points,
                topLeft,
                left,
                bottomLeft,
            )
        } else if lastCol {
            points = append(points,
                topRight,
                right,
                bottomRight,
            )
        } else {
            points = append(points,
                topLeft, topRight, bottomLeft, bottomRight,
                top, bottom, left, right,
                center,
            )
        }
    }
}

deleteSpace := 1
points = slices.DeleteFunc(points, func(p PointTwo) bool {
    if fromNodeSize.X0-deleteSpace <= p.x && p.x <= fromNodeSize.X1-
1+deleteSpace &&
        fromNodeSize.Y0-deleteSpace <= p.y && p.y <= fromNodeSize.Y1-
1+deleteSpace {
        return true
    }

    if toNodeSize.X0-deleteSpace <= p.x && p.x <= toNodeSize.X1-
1+deleteSpace &&
        toNodeSize.Y0-deleteSpace <= p.y && p.y <= toNodeSize.Y1-
1+deleteSpace {
        return true
    }

    return false
})

slices.SortFunc(points, func(a PointTwo, b PointTwo) int {
    if a.x != b.x {
        return a.x - b.x
    }
})

```

```

        }

        return a.y - b.y
    })

    points = slices.Compact(points)
    return points
}

//func debugGrid(grid BlockGrid) {
//    for y := range grid {
//        for x := range grid[y] {
//            cell := grid[y][x]
//
//            clr := color.RGBA{R: 0, G: 0, B: 0, A: 255}
//
//            if y%2 == 0 {
//                clr.R = 255
//            }
//            if x%2 == 0 {
//                clr.B = 255
//            }
//            if y%2 == 0 && x%2 == 0 {
//                clr.G = 255
//            }
//
//            debugRect(
//                cell.X0,
//                cell.Y0,
//                cell.X1,
//                cell.Y1,
//                clr,
//            )
//        }
//    }
//}

//func debugPoints(points []PointTwo) {
//    for _, point := range points {
//        debugRect(point.x, point.y, point.x, point.y, color.RGBA{R: 0, G:
//255, B: 0, A: 50})
//    }
//}

type NodeWeight struct {
    weight int
    node   *GraphNode
}

type NodeNeighbors struct {
    top      NodeWeight
    right    NodeWeight
    bottom   NodeWeight
    left     NodeWeight
}

type GraphNode struct {
    point      PointTwo
    neighbors  NodeNeighbors
}

type Graph struct {
    nodes []*GraphNode
}

```

```

func createGraph(
    points []PointTwo,
    fromNodeSize *Coordinates,
    toNodeSize *Coordinates,
    fromPoint PointTwo,
    toPoint PointTwo,
) (*Graph, *GraphNode, *GraphNode) {
    pointsMap := make(map[int]map[int]*GraphNode)
    nodes := make([]*GraphNode, 0)

    minX, minY, maxX, maxY := math.MaxInt, math.MaxInt, math.MinInt,
math.MinInt

    for _, point := range points {
        x, y := point.x, point.y

        minX = min(minX, x)
        minY = min(minY, y)
        maxX = max(maxX, x)
        maxY = max(maxY, y)

        if _, ok := pointsMap[y]; !ok {
            pointsMap[y] = make(map[int]*GraphNode)
        }

        graphNode := &GraphNode{
            point: point,
            neighbors: NodeNeighbors{
                top:    NodeWeight{weight: math.MaxInt, node: nil},
                right:   NodeWeight{weight: math.MaxInt, node: nil},
                bottom: NodeWeight{weight: math.MaxInt, node: nil},
                left:    NodeWeight{weight: math.MaxInt, node: nil},
            },
        }
        nodes = append(nodes, graphNode)
        pointsMap[y][x] = graphNode
    }

    type Direction struct {
        dir string
        x    int
        y    int
    }
    directions := []Direction{
        {x: 0, y: 1, dir: "top"},
        {x: 0, y: -1, dir: "bottom"},
        {x: 1, y: 0, dir: "right"},
        {x: -1, y: 0, dir: "left"},
    }

    fromNode, ok := pointsMap[fromPoint.y][fromPoint.x]
    if !ok {
        panic("Something went wrong with getting from graph node from points
map")
    }
    toNode, ok := pointsMap[toPoint.y][toPoint.x]
    if !ok {
        panic("Something went wrong with getting to graph node from points
map")
    }

    for _, point := range points {
        x, y := point.x, point.y
        node, ok := pointsMap[y][x]

```

```

        if !ok {
            panic("Something went wrong")
        }

        for _, direction := range directions {
            if x == fromPoint.x && y == fromPoint.y && direction.dir !=
"right" {
                continue
            }
            if x == toPoint.x && y == toPoint.y && direction.dir != "left"
{
                continue
            }

            nx := x
            ny := y

            minDistance := math.MaxInt
            var minDistanceNode *GraphNode = nil

            for minX <= nx && nx <= maxX && minY <= ny && ny <= maxY {
                nx = nx + direction.x
                ny = ny + direction.y

                if fromNodeSize.X0 <= nx && nx <= fromNodeSize.X1 &&
fromNodeSize.Y0 <= ny && ny <= fromNodeSize.Y1 {
                    break
                }
                if toNodeSize.X0 <= nx && nx <= toNodeSize.X1 &&
toNodeSize.Y0 <= ny && ny <= toNodeSize.Y1 {
                    break
                }

                extraDistance := 0
                if nx == fromPoint.x && ny == fromPoint.y &&
direction.dir != "left" {
                    extraDistance = 2
                }
                if nx == toPoint.x && ny == toPoint.y && direction.dir
!= "right" {
                    extraDistance = 2
                }

                yPoints, ok := pointsMap[ny]
                if !ok {
                    continue
                }
                neighborNode, ok := yPoints[nx]
                if !ok {
                    continue
                }

                distance := int(math.Abs(float64(x-
nx))+math.Abs(float64(y-ny))) + extraDistance
                if minDistance > distance {
                    minDistance = distance
                    minDistanceNode = neighborNode
                }
            }

            if minDistance == math.MaxInt || minDistanceNode == nil {
                continue
            }

```

```

weight      :=      NodeWeight{weight:      minDistance,      node:
minDistanceNode}
weightForNeighbor := NodeWeight{weight: minDistance, node:
node}

        if direction.dir == "top" {
            node.neighbors.top = weight
            minDistanceNode.neighbors.bottom = weightForNeighbor
        }
        if direction.dir == "bottom" {
            node.neighbors.bottom = weight
            minDistanceNode.neighbors.top = weightForNeighbor
        }
        if direction.dir == "right" {
            node.neighbors.right = weight
            minDistanceNode.neighbors.left = weightForNeighbor
        }
        if direction.dir == "left" {
            node.neighbors.left = weight
            minDistanceNode.neighbors.right = weightForNeighbor
        }
    }
}

return &Graph{nodes: nodes}, fromNode, toNode
}

type DjikstraStep struct {
    distance      int
    node          *GraphNode
    previousPoints []PointTwo
    direction     string

    index int
}

type PriorityQueue []*DjikstraStep

func (pq PriorityQueue) Len() int { return len(pq) }

func (pq PriorityQueue) Less(i, j int) bool {
    return pq[i].distance < pq[j].distance
}

func (pq PriorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
    pq[i].index = i
    pq[j].index = j
}

func (pq *PriorityQueue) Push(x any) {
    n := len(*pq)
    item := x.(*DjikstraStep)
    item.index = n
    *pq = append(*pq, item)
}

func (pq *PriorityQueue) Pop() any {
    old := *pq
    n := len(old)
    item := old[n-1]
    old[n-1] = nil
    item.index = -1
    *pq = old[0 : n-1]
}

```

```

    return item
}

func dijkstra(graph *Graph, fromNode *GraphNode, toNode *GraphNode) []PointTwo
{
    distances := make(map[*GraphNode]int)
    distancesPaths := make(map[*GraphNode][]PointTwo)
    for _, node := range graph.nodes {
        distances[node] = math.MaxInt
        distancesPaths[node] = make([]PointTwo, 0)
    }
    queue := make(PriorityQueue, 0)

    distances[fromNode] = 0
    queue = append(queue, &DijkstraStep{
        distance: 0,
        node: fromNode,
        previousPoints: []PointTwo{fromNode.point},
        direction: "right",
    })
    heap.Init(&queue)

    for len(queue) > 0 {
        step := heap.Pop(&queue).(*DijkstraStep)
        distance, node, previousPoints := step.distance, step.node,
step.previousPoints

        if distance > distances[node] {
            continue
        }

        directions := []string{"top", "right", "bottom", "left"}
        neighbors := []NodeWeight{
            node.neighbors.top,
            node.neighbors.right,
            node.neighbors.bottom,
            node.neighbors.left,
        }
        for i, neighbor := range neighbors {
            if neighbor.node == nil || neighbor.weight == math.MaxInt {
                continue
            }

            extraWeight := 0
            if step.direction != "" && step.direction != directions[i] {
                extraWeight = 1
            }

            neighborDistance := distance + neighbor.weight + extraWeight
            if neighborDistance >= distances[neighbor.node] {
                continue
            }

            distances[neighbor.node] = neighborDistance

            newPath := make([]PointTwo, len(previousPoints))
            copy(newPath, previousPoints)
            newPath = append(newPath, neighbor.node.point)

            distancesPaths[neighbor.node] = newPath
            heap.Push(&queue, &DijkstraStep{
                distance: neighborDistance,
                node: neighbor.node,
                previousPoints: newPath,
            })
        }
    }
}

```

```

                direction:      directions[i],
            })
        }
    }

    return distancesPaths[toNode]
}

func simplifyPath(path []PointTwo) []PointTwo {
    if len(path) <= 2 {
        return path
    }

    result := []PointTwo{path[0]}

    for i := 1; i < len(path)-1; i++ {
        prev := path[i-1]
        current := path[i]
        next := path[i+1]

        if prev.x == current.x && current.x == next.x {
            continue
        }
        if prev.y == current.y && current.y == next.y {
            continue
        }

        result = append(result, path[i])
    }

    result = append(result, path[len(path)-1])
    return result
}

func pathToCoordinates(path []PointTwo) []Coordinates {
    if len(path) < 2 {
        return nil
    }

    result := make([]Coordinates, len(path)-1)
    for i := 0; i < len(path)-1; i++ {
        result[i] = Coordinates{
            X0: path[i].x,
            Y0: path[i].y,
            X1: path[i+1].x,
            Y1: path[i+1].y,
        }
    }
    return result
}

```

Файл app/entities/logic.go:

```
package entities
```

```
import (
    "errors"
    "maps"
    "slices"
)
```

```
// TODO: In future we should create module for run all logic
// Because right now we can add and delete nodes, and all should work correctly
func RunTick(
```

```

    p *Project,
) {
    prevTick := p.ticksMap[p.currentTick]

    tick := Tick{
        Queue:          make([]NodeId, 0),
        ActiveLinks:     maps.Clone(prevTick.ActiveLinks),
        ProcessedLinks:  maps.Clone(prevTick.ProcessedLinks),
        ActiveConsumers: maps.Clone(prevTick.ActiveConsumers),
    }

    queue := prevTick.Queue
    iteration := 0
    maxIterations := 100

    for len(queue) > 0 {
        // TODO: Implement an error on max iterations for the user!
        if iteration >= maxIterations {
            return
        }

        iteration++
        node := p.Nodes[queue[0]]
        queue = queue[1:]

        result, err := calculateResult(&tick, node)
        if err != nil {
            continue
        }
        if node.Not {
            result = !result
        }

        addOutputsToQueue(&tick, p, node.Outputs, result, &queue)
    }

    p.ticksMap[p.currentTick+1] = tick
}

func addOutputsToQueue(
    tick *Tick,
    project *Project,
    outputs []LinkId,
    isActive bool,
    queue *[]NodeId,
) {
    for _, link := range outputs {
        sameValue := tick.ActiveLinks[link] == isActive
        initialized := tick.ProcessedLinks[link]

        if sameValue && initialized {
            continue
        }

        tick.ActiveLinks[link] = isActive
        tick.ProcessedLinks[link] = true

        toNodeId := project.Links[link].ToNodeId

        if !slices.Contains(*queue, toNodeId) {
            *queue = append(*queue, toNodeId)
        }
    }
}

```

```
func calculateResult(tick *Tick, node *Node) (bool, error)
```

ДОДАТОК Б
СЛАЙДИ ПРЕЗЕНТАЦІЇ

Національний університет «Запорізька політехніка»
Кафедра програмних засобів

РОЗРОБЛЕННЯ КРОСПЛАТФОРМНОЇ ПРОГРАМИ ДЛЯ ПРОЄКТУВАННЯ ТА СИМУЛЯЦІЇ ГРАФОВИХ МОДЕЛЕЙ

Виконав: Іван ЩЕДРОВСЬКИЙ, студент групи КНТ-113сп
Керівник: Тетяна ФЕДОРОНЧАК, к.т.н., доцент

Рисунок Б.1 – Слайд №1

ОБ'ЄКТ ТА ПРЕДМЕТ:

Об'єкт дослідження — це процес розроблення кросплатформного програмного забезпечення на основі графових моделей

Предмет дослідження — це методи та програмні засоби представлення, обходу графових моделей і симуляції сигналів у вузлах графу

Рисунок Б.2 – Слайд №2

МЕТА РОБОТИ:

Метою дипломної роботи є зменшення часу та трудомісткості проектування цифрових логічних схем шляхом розроблення кросплатформного інструменту з інтерактивною симуляцією графових моделей

3

Рисунок Б.3 – Слайд №3

ЗАВДАННЯ РОБОТИ:

- Виконати аналіз предметної області систем проектування та симуляції графових моделей
- Спроекувати програмне забезпечення для проектування та симуляції логічних схем
- Розробити програмне забезпечення
- Виконати тестування розробленого програмного забезпечення

4

Рисунок Б.4 – Слайд №4

Огляд існуючих програмних рішень:

Критерій порівняння	Існуючі програми		
	Logisim	Digital	Logicly
Відкритий програмний код	Так	Так	Ні
Вартість	Безплатний	Безплатний	Комерційний
Ціль створення	Проектування великих схем	Проектування великих схем	Навчання студентів
Зручність інтерфейсу	Низька	Низька	Висока
Кількість елементів	Велика	Велика	Середня
Зручність створення зв'язків	Низька	Низька	Висока
Робоча область	Обмежена	Необмежена, з можливістю переміщення	Необмежена, з можливістю переміщення

5

Рисунок Б.5 – Слайд №5

Обґрунтування вибору програмних засобів:

Параметр порівняння	Мова програмування		
	Go	Rust	Zig
1	2	3	4
Складність вивчення	Середня	Дуже висока	Висока
Рік випуску	2009	2010	2016
Швидкодія	Висока	Висока	Висока
Швидкість розробки	Висока	Середня	Середня
Управління пам'яттю	Автоматичне, через збирач сміття	Автоматичне, через Borrow Checker	Ручне
Документація	Повноцінна	Повноцінна	Майже відсутня
Розробник	Компанія <u>Google</u>	Некомерційна організація <u>Rust Foundation</u>	Некомерційна організація <u>Zig Software Foundation</u>

6

Рисунок Б.6 – Слайд №6

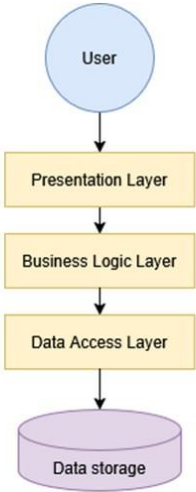
Обґрунтування вибору середовища розробки системи:

7

Параметр порівняння	Середовище розробки		
	GoLand	Visual Studio Code	Neovim
Мета створення середовища	Розробка програм виключно на мові програмування Go	Розробка програм на всіх мовах програмування	Розробка програм на всіх мовах програмування
Швидкодія	Висока	Висока	Найкраща
Налаштування для роботи	Майже не є потрібні	Потрібно встановити необхідні розширення для мови Go	Потрібно встановити велику кількість розширень для повноцінної роботи
Рівень кастомізації	Низький	Середній	Дуже високий
Кількість доповнень	Середня	Висока	Дуже висока
Автоматична синхронізація між іншими середовищами розробки	Вбудована, через персональний аккаунт	Не вбудована, можлива через додаткові розширення	Не вбудована, можлива через додаткові розширення
Власник	Компанія JetBrains	Компанія Microsoft	Немає, безкоштовний відкритий проєкт

Рисунок Б.7 – Слайд №7

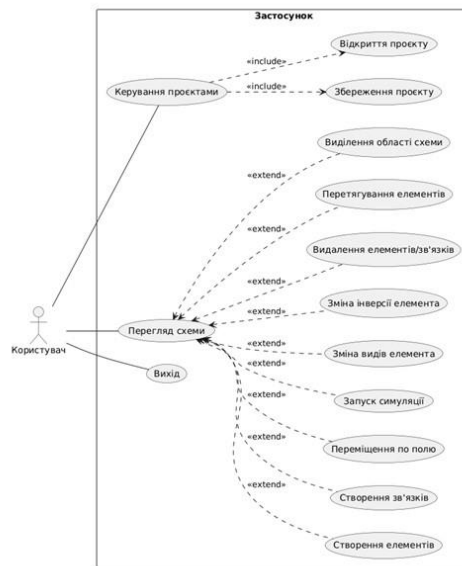
Структурна схема програми:



8

Рисунок Б.8 – Слайд №8

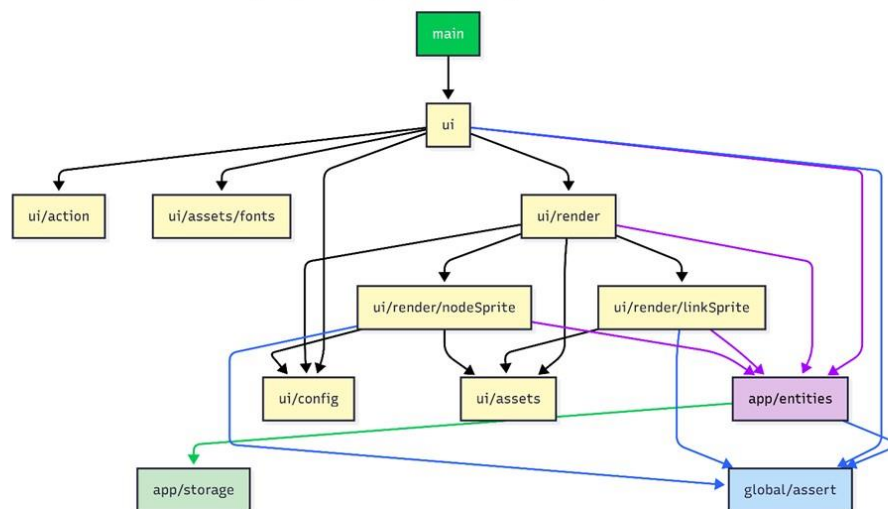
Діаграма прецедентів



9

Рисунок Б.9 – Слайд №9

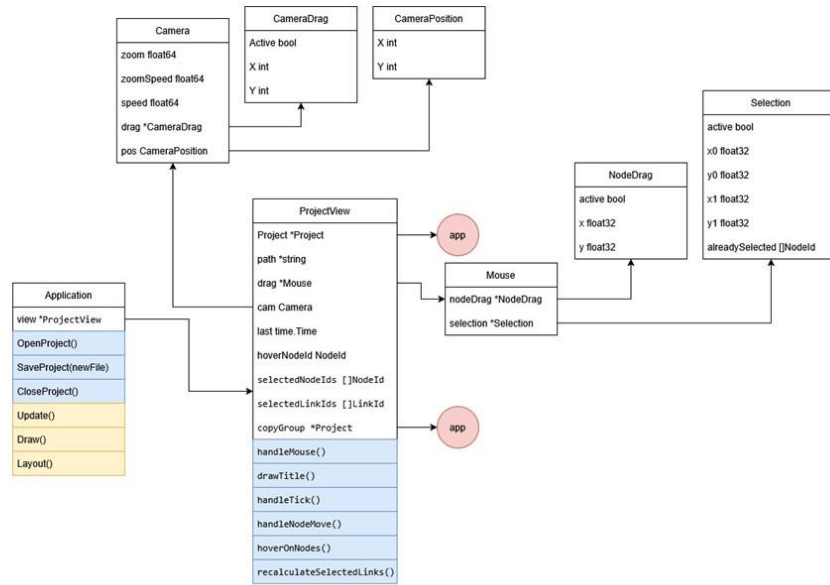
Структура програми:



10

Рисунок Б.10 – Слайд №10

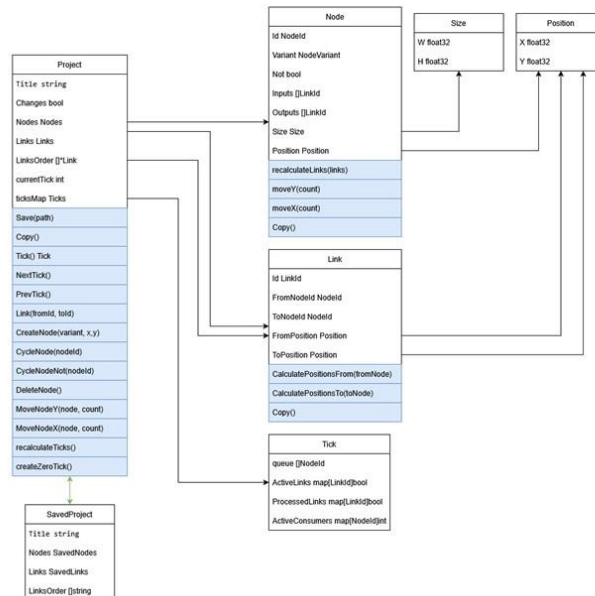
Моделі даних, шар візуалізації:



11

Рисунок Б.11 – Слайд №11

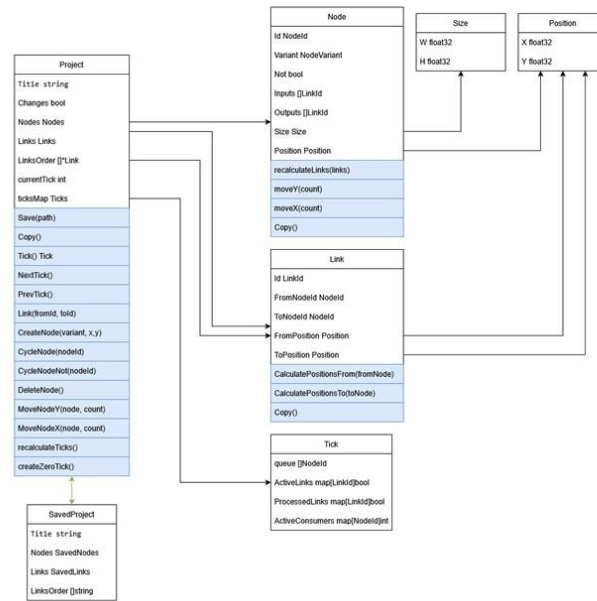
Моделі даних, шар застосунку:



12

Рисунок Б.12 – Слайд №12

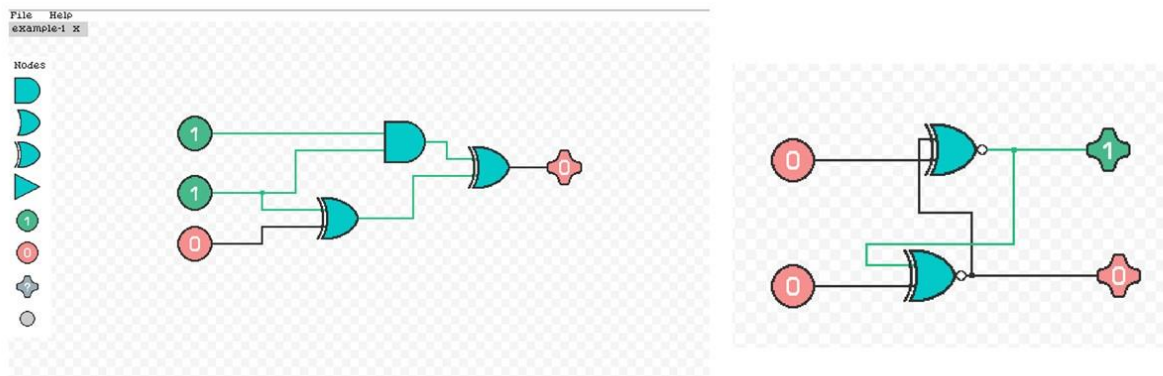
Моделі даних, шар застосунку:



13

Рисунок Б.13 – Слайд №13

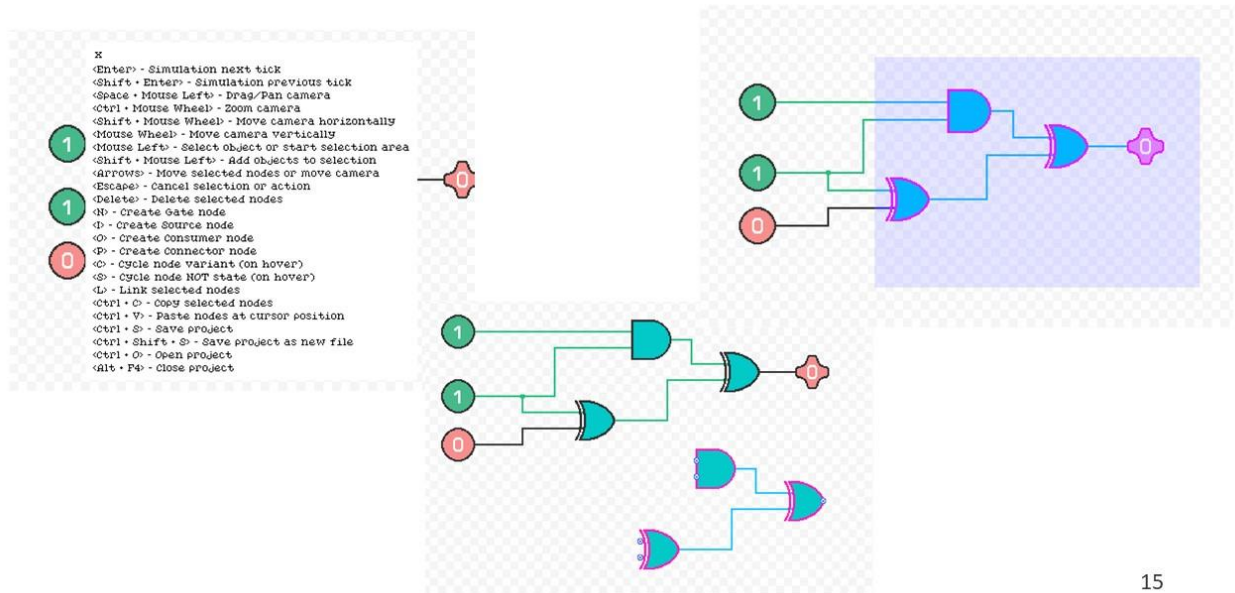
Робота з програмою. Приклади схем:



14

Рисунок Б.14 – Слайд №14

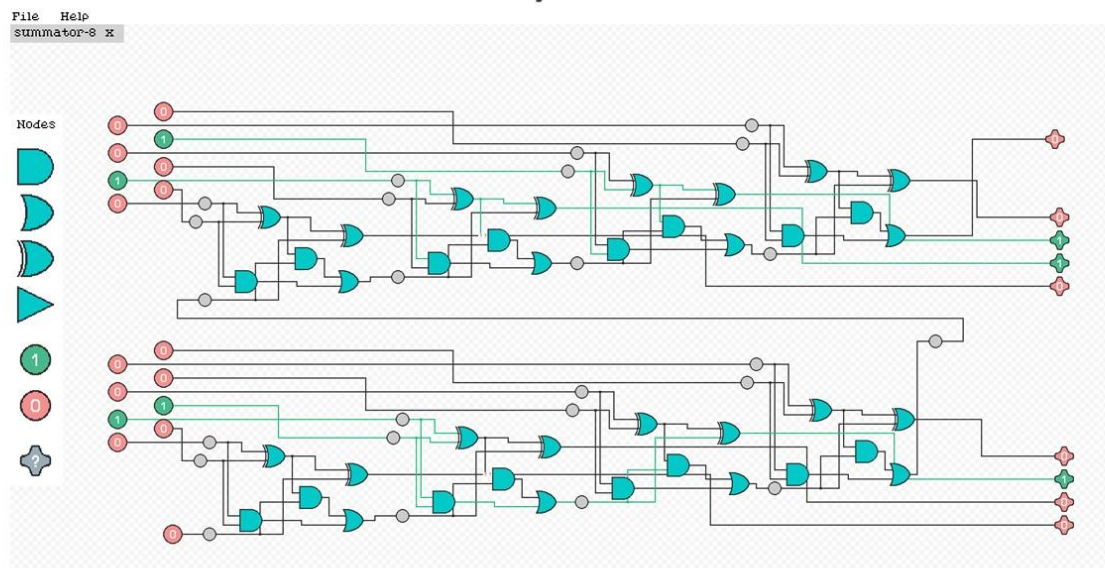
Робота з програмою. Додаткові можливості:



15

Рисунок Б.15 – Слайд №15

Тестування:



16

Рисунок Б.16 – Слайд №16

Висновки:

В результаті виконання цієї дипломної роботи було створено кросплатформний застосунок з інтерактивною симуляцією графових моделей, який зменшує час та трудомісткість проектування цифрових логічних схем.

В результаті аналізу предметної області було окреслено головні поняття, визначено об'єкт і предмет дослідження, встановлено мету дослідження, оглянуто існуючі програмні рішення, встановлено вимоги до надійності та проектованої системи, а також визначені умови роботи та розповсюдження програми.

Обґрунтовано середовища функціонування, розробки та програмні засоби програми, визначено структуру програми та спроектовано сценарії використання.

Описані всі модулі програми з описом технологій які були обрані під час проектування та основними рішеннями щодо їх реалізації.

Розроблено інструкцію з експлуатація для користувачів та проведено повноцінне тестування застосунку, в результаті якого було визначено, що застосунок відповідає всім поставленим умовам і не має дефектів в його роботі, але має простір для покращення в майбутньому.

17

Рисунок Б.17 – Слайд №17