Форма № 24

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 СТВОРЕННЯ ПРОГРАМНОЇ КРОСПЛАТФОРМНОЇ СИСТЕМИ ДЛЯ ВІЗУАЛІЗАЦІЇ ДАНИХ МОНІТОРИНГУ У КОНСОЛЬНИХ ЗАСТОСУНКАХ
DEVELOPMENT OF A CROSS-PLATFORM SOFTWARE SYSTEM FOR THE VISUALIZATION OF MONITORING DATA IN CONSOLE APPLICATIONS

Виконав(ла): студент(ка) 4 курсу, групи КНТ-222
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні науки

ШАПОВАЛ М.Р.
(ПРІЗВИЩЕ та ініціали)
Керівник СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)
Рецензент ДЯЧУК Т.С.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

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

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерні науки
(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)

ШАПОВАЛА Максима Руслановича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Створення програмної кросплатформної системи для візуалізації даних моніторингу у консольних застосунках. Development of a Cross-platform Software System for the Visualization of Monitoring Data in Console Applications
керівник проєкту (роботи) к.т.н., доцент, СТЕПАНЕНКО Олександр Олексійович,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, 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. Дослідження методів консольної графіки. 3. Проєктування та реалізація системи 4. Тестування та апробація системи.

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

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	СТЕПАНЕНКО О.О., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

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	Дослідження методів консольної графіки	2 тиждень	Розділ 2
4	Проєктування архітектури системи	2 тиждень	Розділ 3
5	Розробка програмного забезпечення	3-4 тижні	Розділ 3
6	Тестування та апробація системи	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) _____ Максим ШАПОВАЛ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Олександр СТЕПАНЕНКО
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 131 с., 8 табл., 32 рис., 2 дод., 30 джерел.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, КОНСОЛЬНІ ЗАСТОСУНКИ, ВІЗУАЛІЗАЦІЯ ДАНИХ, ДАНІ МОНІТОРИНГУ, КРОСПЛАТФОРМНА СИСТЕМА, WINDOWS GDI, ANSI.

Об’єкт дослідження – процес візуалізації даних моніторингу у консольних застосунках на різних програмних платформах.

Предмет дослідження – методи та програмні засоби кросплатформного графічного виводу у консольному вікні без використання зовнішніх графічних фреймворків.

Мета роботи – зменшення часу та ресурсних витрат на впровадження візуалізації даних моніторингу у консольних застосунках шляхом створення кросплатформної системи з уніфікованим програмним інтерфейсом для платформ Windows та Linux.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, мова програмування C++, середовище розробки Visual Studio Code, компілятор GCC, віртуальна машина VMware Player з Ubuntu 24, персональний комп'ютер з процесором Ryzen 5 7600 під управлінням ОС Microsoft Windows 11.

Результати. Розроблено кросплатформну програмну систему, що дозволяє відображати графіки, діаграми та дані моніторингу безпосередньо у консольному вікні з підтримкою двох бекендів: Windows GDI для локального середовища Windows та ANSI escape-послідовностей для терміналів Linux, у тому числі при роботі через SSH. Уніфікований програмний інтерфейс системи дозволяє використовувати один і той самий код застосунку на обох платформах, що скорочує трудомісткість розробки моніторингових інструментів.

Висновки. Створена система зменшує час впровадження консольної візуалізації за рахунок єдиного API для двох платформ та знижує вимоги до програмного середовища завдяки відсутності зовнішніх графічних залежностей.

Галузь використання – системне адміністрування, моніторинг серверних середовищ, аналіз даних у віддалених обчислювальних системах, розробка консольних інструментів з вбудованою візуалізацією.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 131 pages, 8 tables, 32 figures, 2 appendixes, 30 sources.

SOFTWARE, CONSOLE APPLICATIONS, DATA VISUALIZATION, MONITORING DATA, CROSS-PLATFORM SYSTEM, WINDOWS GDI, ANSI.

Object of study is a process of monitoring data visualization in console applications across different software platforms.

Subject of study is a methods and software tools for cross-platform graphical output in a console window without the use of external graphical frameworks.

The purpose of the work is to reduce the time and resource costs of implementing monitoring data visualization in console applications by creating a cross-platform system with a unified application programming interface for Windows and Linux platforms.

Materials, methods, and tools: structured and object-oriented programming, C++ programming language, integrated development environment Visual Studio Code, GCC compiler, VMware Player virtual machine with Ubuntu 24, personal computer with a Ryzen 5 7600 processor under the control of the Microsoft Windows 11 operating system.

Results. The cross-platform software system is created which displays charts, diagrams, and monitoring data directly in a console window with support for two backends: Windows GDI for the local Windows environment and ANSI escape sequences for Linux terminals, including operation over SSH. The unified application programming interface allows the same application code to be used on both platforms, which reduces the development effort of monitoring tools.

Conclusions A cross-platform software system for monitoring data visualization in console applications has been developed. It reduces the time of implementing console visualization and lowers software environment requirements due to the absence of external graphical dependencies.

The field of use is a system administration, server environment monitoring, data analysis in remote computing systems, development of console tools with built-in visualization.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

GDI – Graphics Device Interface;

IDE – Integrated Development Environment,

OS – Operating System;

SDK – Software Development Kit;

TUI – Terminal User Interface;

VPS – Virtual Private Server;

ООП – об'єктно-орієнтоване програмування;

ПЗ – програмне забезпечення.

ВСТУП

Сучасна розробка програмного забезпечення дедалі частіше передбачає використання віддалених обчислювальних ресурсів – VPS-серверів, хмарних середовищ та виділених машин для аналізу даних, навчання моделей машинного навчання та моніторингу систем. У таких умовах розробник або дослідник взаємодіє з машиною виключно через термінал, без доступу до графічного інтерфейсу. Це породжує практичну потребу у візуалізації даних безпосередньо в консольному середовищі – без запуску окремих графічних застосунків, без передачі зображень по мережі та без встановлення численних залежностей. Існуючі рішення для побудови графіків, такі як matplotlib, gnuplot або засоби веб-дашбордів, орієнтовані на середовища з повноцінним графічним стеком. Їх використання у віддалених серверних середовищах при SSH-підключенні є або неможливим, або вимагає значних налаштувань інфраструктури. Водночас консольне середовище Windows, попри наявність потужного графічного інтерфейсу GDI, практично не використовується для цілей інтерактивної візуалізації даних у терміналі. Оскільки єдиної програмної системи, яка б надавала уніфікований програмний інтерфейс для відображення графіків у консольних застосунках із підтримкою різних платформ та середовищ доступу, на сьогодні не існує у вільному доступі, тема дипломної кваліфікаційної роботи бакалавра є актуальною. Об'єктом дослідження є процес візуалізації даних моніторингу у консольних застосунках на різних програмних платформах. Предметом дослідження є методи та програмні засоби кросплатформного графічного виводу у консольному вікні без використання зовнішніх графічних фреймворків. Метою роботи є зменшення часу та ресурсних витрат на впровадження візуалізації даних моніторингу у консольних застосунках шляхом створення кросплатформної системи з уніфікованим програмним інтерфейсом для платформ Windows та Linux.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати предметну область, дослідити існуючі підходи до консольної та термінальної візуалізації даних і виконати огляд програмних аналогів;

– виконати порівняльний аналіз особливостей платформ Windows та Linux як цільових середовищ для реалізації консольної графіки;

– обґрунтувати вибір технологій реалізації для кожної з цільових платформ та провести експериментальне дослідження обраних методів графічного виво;

– спроєктувати архітектуру системи з підтримкою змінних бекендів виводу на основі абстрактного програмного інтерфейсу;

– реалізувати бекенд Windows GDI для локального середовища та бекенд ANSI escape-послідовностей для віддаленого доступу через SSH;

– реалізувати уніфікований програмний інтерфейс для побудови графіків, діаграм та відображення даних моніторингу у реальному часі;

– провести тестування розробленої системи, оцінити її продуктивність та практичну придатність на обох цільових платформах.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Наведено обсяг з аналізу предметної області, у якому розв'язується задача кросплатформної візуалізації даних моніторингу у консольних застосунках. Розглянуто роль консолі у сучасній розробці програмного забезпечення та її технічну основу як середовища взаємодії з обчислювальною системою. Проаналізовано загальні підходи до візуалізації даних у терміналі – від псевдографіки до сучасних механізмів кольорового виводу. Виконано огляд існуючих програмних аналогів, які реалізують консольну візуалізацію, та визначено їх основні можливості й обмеження. Окрему увагу приділено порівняльному аналізу особливостей операційних систем Windows та Linux як цільових платформ, оскільки саме відмінності їхніх консольних середовищ зумовлюють потребу у кросплатформному рішенні. Розглянуто специфіку даних моніторингу як об'єкта візуалізації та сформульовано вимоги до програмних засобів їх відображення. На основі проведеного аналізу здійснено постановку завдань для розробки власної програмної системи.

1.1 Консольні застосунки у сучасній розробці

Консольні застосунки є однією з найдавніших форм взаємодії людини з обчислювальною технікою. Історично цей спосіб комунікації виник задовго до появи графічних інтерфейсів і пройшов значний шлях еволюції. Першими пристроями введення-виведення для електронно-обчислювальних машин були телетайпи – електромеханічні пристрої, що друкували символи на паперовій стрічці у відповідь на команди оператора на рисунку 1.1 представлено вигляд такого пристою модклі 33 ASR, який використовувався у якостиі ком’ютерного терміналу[1]-[2].

Рисунок 1.1 – Телетайп Model 33 ASR[1]

Згодом, із появою електронно-променевих дисплеїв, телетайпи поступилися місцем відеотерміналам, які забезпечили інтерактивну роботу з символьним поданням інформації безпосередньо на екрані.

Класичним прикладом такого терміналу став VT100 (рисунок 1.2), розроблений компанією Digital Equipment Corporation наприкінці 1970-х років. Саме він закріпив низку стандартів керування курсором, кольоровим виводом та форматуванням тексту, які зберігають свою актуальність донині [2]-[3].

Рисунок 1.2 – Вигляд текстового комп’ютерного терміналу VT100[3]

На особистих комп'ютерах першого покоління, таких як IBM PC, операційна система MS-DOS пропонувала виключно символьний інтерфейс командного рядка[4]. Подальший розвиток операційних систем сімейства UNIX закріпив консольний підхід як основний у середовищі серверного програмного забезпечення та наукових обчислень. Графічні інтерфейси, що набули поширення у 1980-1990 роках, не витіснили консоль, а лише доповнили її, залишивши за нею роль універсального інструменту низькорівневої взаємодії з системою[5].

У сучасній розробці програмного забезпечення консольні застосунки відіграють ключову роль у багатьох предметних галузях. Передусім ідеться про сферу системного адміністрування, де керування серверами здебільшого здійснюється через текстовий інтерфейс. Адміністратор підключається до віддаленої машини за протоколом SSH і виконує необхідні операції за допомогою команд, не маючи при цьому ані потреби, ані можливості використовувати графічну оболонку. Аналогічна ситуація характерна для практики DevOps, де автоматизовані процеси розгортання, конфігурування та моніторингу здійснюються виключно у консольному середовищі[6],[7].

Особливу значущість консольні застосунки мають у системах безперервної інтеграції та доставки (CI/CD), де процеси збирання, тестування та публікації програмного продукту запускаються на віддалених агентах без графічного інтерфейсу[8]. У таких середовищах будь-який інструментарій, що потребує наявності графічної підсистеми, є непридатним до використання. Натомість консольні засоби, які виводять результати у вигляді тексту або форматованих звітів, інтегруються у конвеєр без додаткових налаштувань.

Окремо слід згадати галузь розробки програмного забезпечення з відкритим кодом, у якій значна частина інструментарію – від компіляторів та систем збирання до засобів керування версіями – реалізована саме як консольні застосунки. Це зумовлено низкою практичних переваг такого підходу: швидкістю запуску, можливістю комбінування програм через канали pipes та зручністю автоматизації за допомогою сценаріїв.

Актуальність консольних застосунків у сучасних умовах підтримується кількома фундаментальними чинниками. По-перше, такі програми мають мінімальні вимоги до системних ресурсів – вони не потребують графічної підсистеми, віконного менеджера чи апаратного прискорення відображення. По-друге, консольний інтерфейс однаково функціонує як у локальному, так і у віддаленому режимі: робота через мережу не вимагає окремих протоколів передачі зображення і відбувається у звичайній термінальній сесії. По-третє, текстовий формат виводу легко піддається обробці іншими програмами, що робить консольні застосунки природним складником автоматизованих обчислювальних процесів.

З технічної точки зору взаємодія консольного застосунку з операційною системою побудована на концепції стандартних потоків введення-виведення. Ця концепція, що сягає корінням операційної системи UNIX, передбачає наявність у кожного процесу трьох базових потоків: стандартного потоку введення (stdin), стандартного потоку виведення (stdout) та стандартного потоку повідомлень про помилки (stderr). Стандартний потік введення призначений для отримання даних від користувача або іншого процесу, стандартний потік виведення – для передачі основних результатів роботи програми, а стандартний потік помилок – для виведення діагностичних та аварійних повідомлень окремо від основного результату. Такий поділ потоків дозволяє реалізувати елегантні механізми перенаправлення та комбінування програм без зміни їхнього вихідного коду[7].

У контексті взаємодії з користувачем консольний застосунок виконується у спеціальному середовищі, яке називають терміналом або емулятором терміналу. Власне термінал – це історичне поняття, що походить від часів, коли терміналом називали окремий апаратний пристрій із дисплеєм та клавіатурою. У сучасних операційних системах апаратні термінали поступилися місцем програмним емуляторам – застосункам, які відтворюють поведінку класичних терміналів у вікні графічної оболонки або

безпосередньо у текстовому режимі ядра. До найпоширеніших емуляторів терміналів належать xterm, gnome-terminal та konsole у середовищі Linux, а також Windows Terminal та конхост (conhost) у Microsoft Windows. Емулятор терміналу забезпечує відображення символів, що надходять зі стандартного потоку виведення процесу, обробляє натискання клавіш користувача та передає їх у стандартний потік введення, а також інтерпретує службові послідовності керування – зміну кольору, переміщення курсору, очищення екрана[9].

Таким чином, консольні застосунки залишаються повноцінним і затребуваним класом програмного забезпечення, який відіграє істотну роль у сучасних процесах розробки, адміністрування та експлуатації обчислювальних систем. Їхня технічна простота та універсальність роблять консольне середовище привабливою платформою для широкого кола задач, серед яких – задачі моніторингу та візуалізації даних, що становлять предмет дослідження дипломної роботи бакалавра.

1.2 Підходи до візуалізації у терміналі

Попри те, що консольні застосунки за своєю природою працюють із текстовими даними, протягом усієї історії їх існування розробники прагнули розширити можливості виводу та забезпечити користувача графічним поданням інформації безпосередньо у вікні терміналу. Сформувалися кілька принципово різних підходів до такої візуалізації, кожен з яких має власну сферу застосування, технічні особливості та обмеження. Розгляд цих підходів є необхідним для розуміння того, які саме засоби доступні розробнику консольного програмного забезпечення і яким чином можна реалізувати побудову графіків та діаграм у середовищі без графічного інтерфейсу.

Найдавнішим та найбільш універсальним підходом до візуалізації у терміналі є використання ASCII-псевдографіки. Ця техніка ґрунтується на формуванні зображень із звичайних друкованих символів стандарту ASCII,

таких як скісні риски, тире, вертикальні смуги та крапки, на рисунку 1.3 представлено таблицю з символів та відповідними alt-кодами.

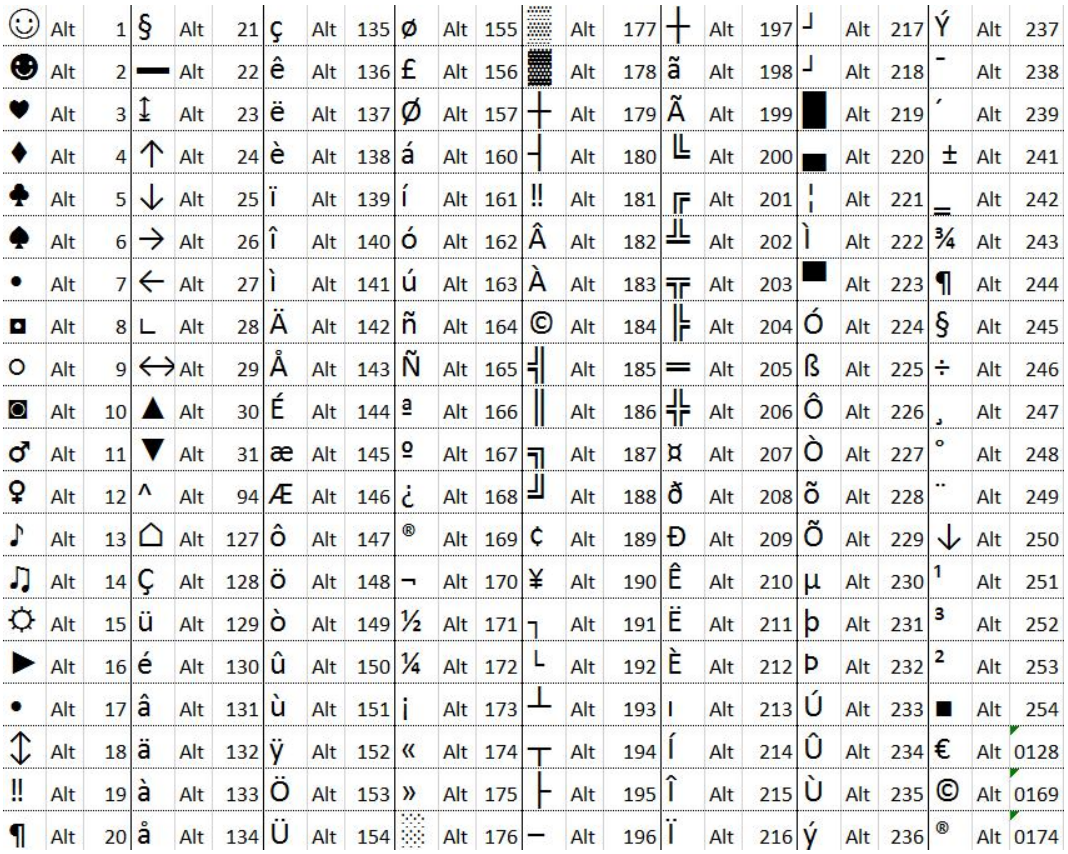


Рисунок 1.3 – Таблиця символів ASCII з Alt-кодами

Поєднуючи ці символи у певному порядку, можна побудувати схематичні зображення прямокутників, ліній, дерев, графіків та інших структур (рисунок 1.4). Перевага такого підходу полягає у його абсолютній сумісності, оскільки набір символів ASCII підтримується кожним без винятку терміналом і кодуванням [10]. Водночас якість зображення залишається доволі низькою, а самі картинки мають характерний грубий вигляд через обмеженість виражальних засобів стандартного набору символів.

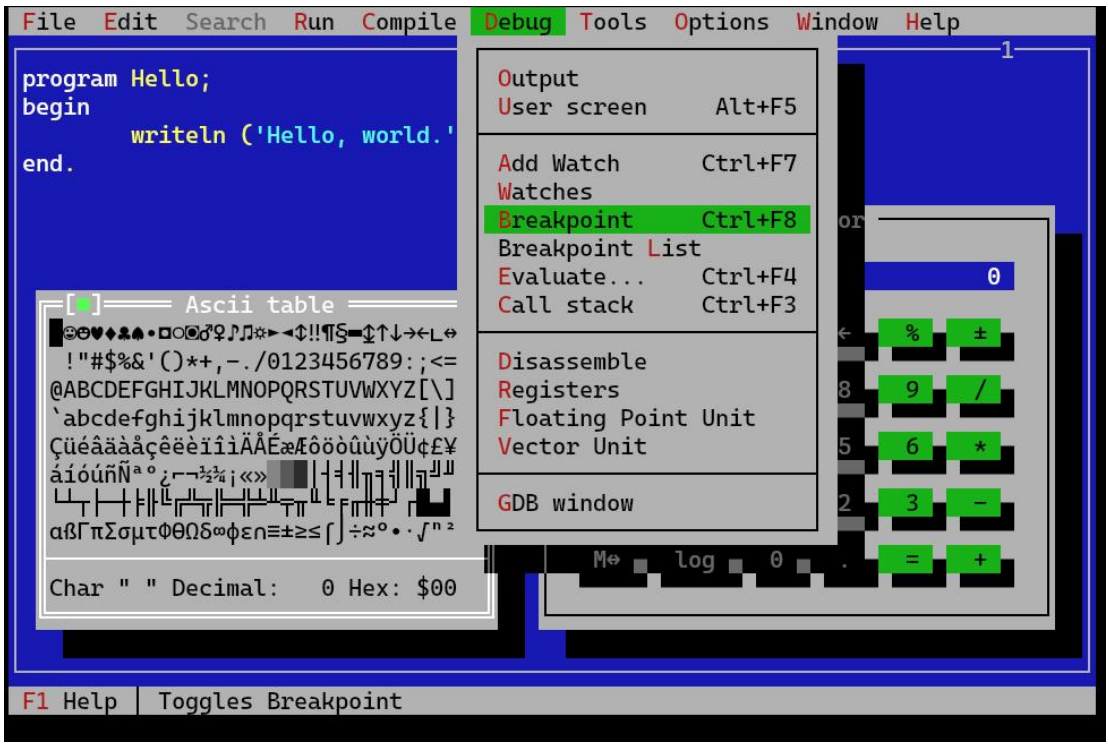


Рисунок 1.4 – Приклад інтерфейсу побущзованим з ASCII

Подальший розвиток підходу пов'язаний із розширеним набором символів кодування CP437, яке набуло поширення в епоху IBM PC та операційної системи MS-DOS. Це кодування містило спеціальні символи рамок, кутів та з'єднань, які дозволяли формувати чітко окреслені вікна, таблиці та меню. Саме завдяки CP437 у текстовому режимі MS-DOS з'явилися повноцінні псевдографічні інтерфейси, що візуально нагадували віконні системи. Хоча сьогодні кодування CP437 поступилося місцем стандарту Unicode, символи рамок із нього успадковані сучасними шрифтами і використовуються у багатьох консольних застосунках [11].

Стандарт Unicode, що став універсальним способом подання символів у сучасних обчислювальних системах, надав значно ширші можливості для псевдографіки. Окрім успадкованих символів рамок, у ньому міститься великий блок графічних символів, серед яких особливе місце посідають блокові символи. Це символи, що візуально являють собою заповнені прямокутні області різної висоти та конфігурації, такі як верхня половина блоку, нижня половина блоку, повний блок, ліва та права половини.

Розміщуючи такі символи поруч, можна досягти ефекту, наближеного до піксельної графіки. Особливо цікавим є використання символів, що поєднують верхню та нижню половини комірки терміналу, оскільки це фактично подвоює вертикальну роздільну здатність зображення[7].

Ще одним різновидом Unicode-псевдографіки є застосування символів Брайля. Хоча первісно вони призначалися для письма для незрячих, кожен такий символ являє собою матрицю з восьми крапок, яка може бути зашифрована у вигляді набору точок. У контексті термінальної графіки це дозволяє відображати у межах однієї комірки терміналу до восьми окремих графічних точок (рисунок 1.5), що значно підвищує деталізацію зображення порівняно з блоковими символами. Деякі сучасні засоби термінальної візуалізації використовують саме Брайль-патерни для побудови ліній та кривих, отримуючи у такий спосіб зображення з відносно високою роздільною здатністю[12].

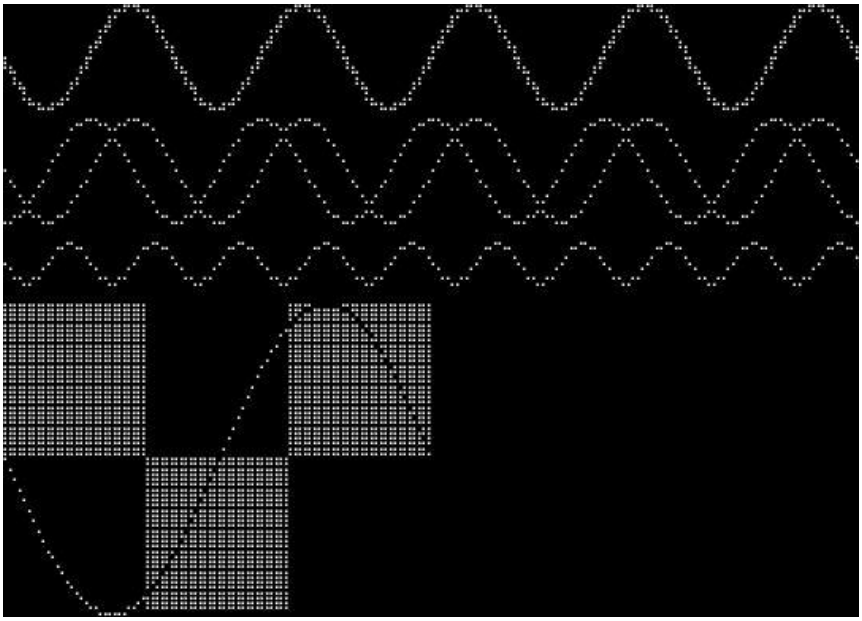


Рисунок 1.5 – Приклад виводу графіки у консолі в Unicode з застовуванням Брайль-патерну[12]

Наступний рівень можливостей термінальної візуалізації забезпечують так звані ANSI escape-послідовності. Йдеться про спеціальні

керівні послідовності, які надсилаються у стандартний потік виведення разом зі звичайним текстом, але інтерпретуються емулятором терміналу не як символи для відображення, а як команди керування. За допомогою escape-послідовностей застосунок може змінювати колір тексту і фону, переміщувати курсор у будь-яку позицію на екрані, очищати екран або окремі його ділянки, а також керувати атрибутами форматування. Такий механізм бере свій початок ще від терміналу VT100 і у сучасному вигляді стандартизований у документі ECMA-48. Підтримка ANSI escape-послідовностей є вбудованою в усі сучасні термінальні емулятори у середовищі Linux, а починаючи з оновлення Windows 10 така підтримка з'явилася й у середовищі Microsoft Windows.

Поєднання ANSI-послідовностей з псевдографічними символами Unicode дозволяє формувати у терміналі повноцінні графічні зображення з кольоровими заливками, плавними переходами та інтерактивним оновленням окремих ділянок екрана. Сучасні емулятори терміналу підтримують режим truecolor, у якому колір задається трьома компонентами RGB у діапазоні від 0 до 255, що відкриває можливості для передачі мільйонів відтінків. Завдяки цьому термінальна графіка перестає виглядати спрощеною і за якістю наближається до растрових зображень малої роздільної здатності. Серед обмежень такого підходу слід відзначити дискретність картинки, оскільки мінімальною одиницею зображення залишається символьна комірка терміналу, розмір якої залежить від обраного шрифту, а також обмеження пов'язані з продуктивністю при значному обсязі оновлюваних даних[13].

Принципово відмінний підхід доступний у середовищі операційної системи Microsoft Windows і ґрунтується на можливості безпосереднього звернення до графічної підсистеми операційної системи. Консольне вікно у Windows за своєю суттю є звичайним вікном графічного інтерфейсу, яке має власний дескриптор вікна і може бути об'єктом малювання за допомогою інтерфейсу Windows GDI. Цей графічний інтерфейс надає набір функцій для роботи з пікселями, лініями, прямокутниками, текстом та іншими

графічними примітивами. Відповідно, застосунок отримує можливість малювати безпосередньо у вікні консолі довільні зображення без обмежень символьної сітки[14].

Підхід на основі Windows GDI забезпечує найвищу якість зображення серед усіх розглянутих, оскільки дозволяє оперувати окремими пікселями та використовувати антиаліасинг для згладжування ліній і шрифтів. Водночас цей підхід має суттєві обмеження. Передусім він є платформозалежним і працює виключно у середовищі Windows. Крім того, він тісно прив'язаний до конкретної архітектури консольного вікна. У сучасних реалізаціях терміналу для Windows, зокрема у Windows Terminal, концепція дескриптора вікна консолі відрізняється від класичної моделі, що ускладнює пряме застосування GDI до таких застосунків. Окремою проблемою є несумісність такого підходу з віддаленими сесіями, оскільки при підключенні через SSH у клієнта немає доступу до графічного контексту серверного вікна.

Узагальнюючи розглянуті підходи, можна виокремити дві магістральні стратегії побудови графіки у консольному середовищі. Перша стратегія базується на текстовому виводі та псевдографічних символах у поєднанні з ANSI escape-послідовностями. Вона є кросплатформною, добре працює у віддалених сесіях та не потребує доступу до графічного інтерфейсу системи, однак має обмежену роздільну здатність через дискретність символьної сітки. Друга стратегія використовує безпосереднє звернення до графічної підсистеми операційної системи. Вона забезпечує піксельну точність та високу якість зображення, але прив'язана до конкретної платформи і не сумісна з режимом віддаленого доступу. Така принципова відмінність двох підходів зумовлює потребу у комбінованому рішенні, яке поєднувало б переваги обох стратегій у межах єдиної програмної системи.

1.3 Огляд програмних аналогів

Задача побудови графічних елементів у консольному середовищі не є новою, і протягом останніх десятиліть розробниками створено цілу низку програмних бібліотек та інструментів для її розв'язання. Розгляд цих програмних продуктів дозволяє визначити поточний рівень розвитку галузі, виявити основні підходи до реалізації та окреслити нерозв'язані задачі, які становлять прогалини у наявному програмному забезпеченні. У цьому підрозділі виконано огляд найбільш відомих та активно використовуваних аналогів за такими критеріями як мова реалізації, цільові платформи, тип графіки, відкритість вихідного коду та активність розробки.

Найдавнішою та найбільш відомою бібліотекою для створення текстових інтерфейсів у терміналі є ncurses бібліотека програмування для створення текстових користувацьких інтерфейсів (TUI), що працюють у широкому спектрі терміналів приклад інтерфейсу приведено на рисунку 1.6. Розроблена як вільна реалізація класичної бібліотеки curses із UNIX System, вона стала фактичним стандартом для побудови інтерфейсів командного рядка у середовищі UNIX-подібних операційних систем. Бібліотека написана мовою Сі та надає функції для роботи з вікнами, кольорами, обробкою клавіатурного введення та оптимізованого оновлення екрана. Серед практичних застосувань ncurses можна навести менеджери файлів, текстові редактори, моніторингові утиліти та інтерактивні інсталятори. До переваг ncurses належить її повсюдна доступність у системах GNU/Linux, BSD та macOS, а також стабільність API, що зберігається протягом десятиліть. Водночас її основним обмеженням залишається орієнтація на класичну символьну графіку без сучасних можливостей, таких як truecolor чи піксельне відображення. Підтримка Windows можлива лише через стороннє відгалуження PDCurses, що знижує однорідність кросплатформного коду[16]-[18].

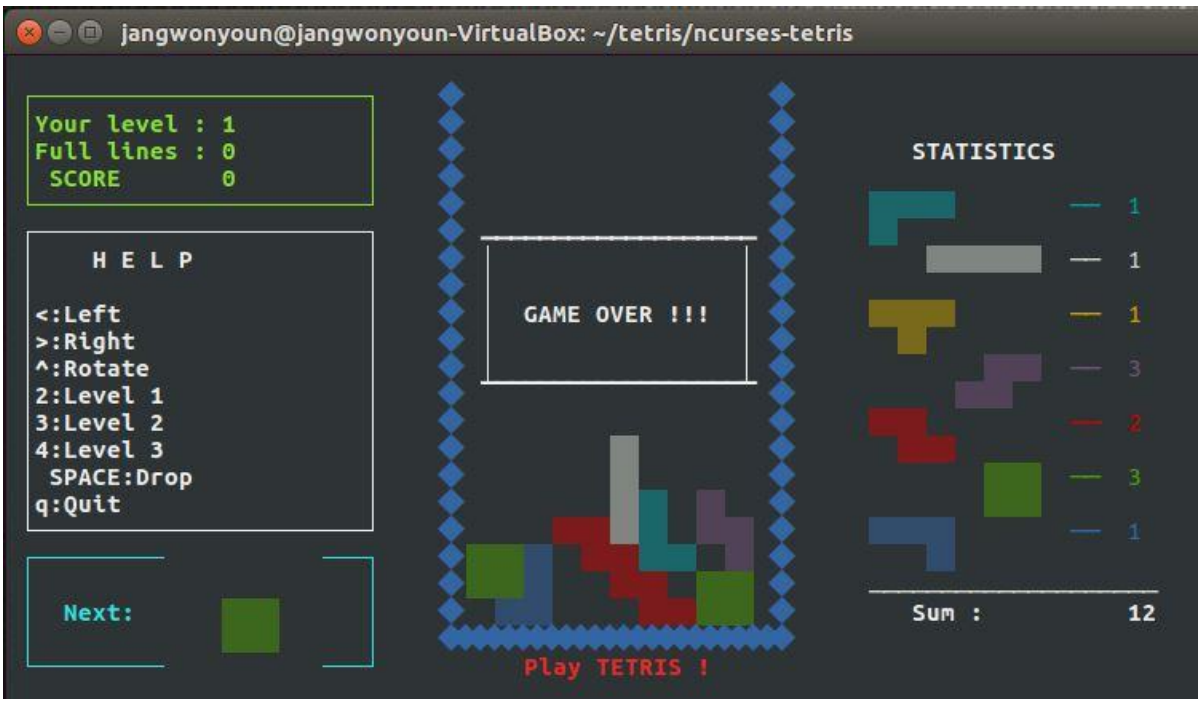


Рисунок 1.6 – Приклад інтерфейсу Ncurses[18]

Сучасною альтернативою ncurses у середовищі C++ є бібліотека FTXUI, що розроблюється Артуром Зонзогні. Це проста кросплатформна C++ бібліотека для побудови термінальних інтерфейсів користувача у функціональному стилі, натхненна підходами React. Бібліотека має сучасний декларативний API, у якому елементи інтерфейсу описуються деревом композицій та оформляються через декоратори. FTXUI підтримує клавіатурно-мишину навігацію, анімації, малювання довільних графічних форм символьними засобами та працює без зовнішніх залежностей. Згідно з даними репозиторію проєкту, актуальна версія датується 2026 роком, що свідчить про активну розробку. Бібліотека поширюється під ліцензією MIT та офіційно підтримує Linux, macOS, Windows і WebAssembly. До її обмежень слід віднести орієнтацію переважно на побудову інтерфейсів користувача, а не на наукову візуалізацію, тому функціональність побудови графіків та діаграм у ній представлена лише базовим набором примітивів[19]. Приклад виводу консольної графіки наведений на рисунку 1.7.

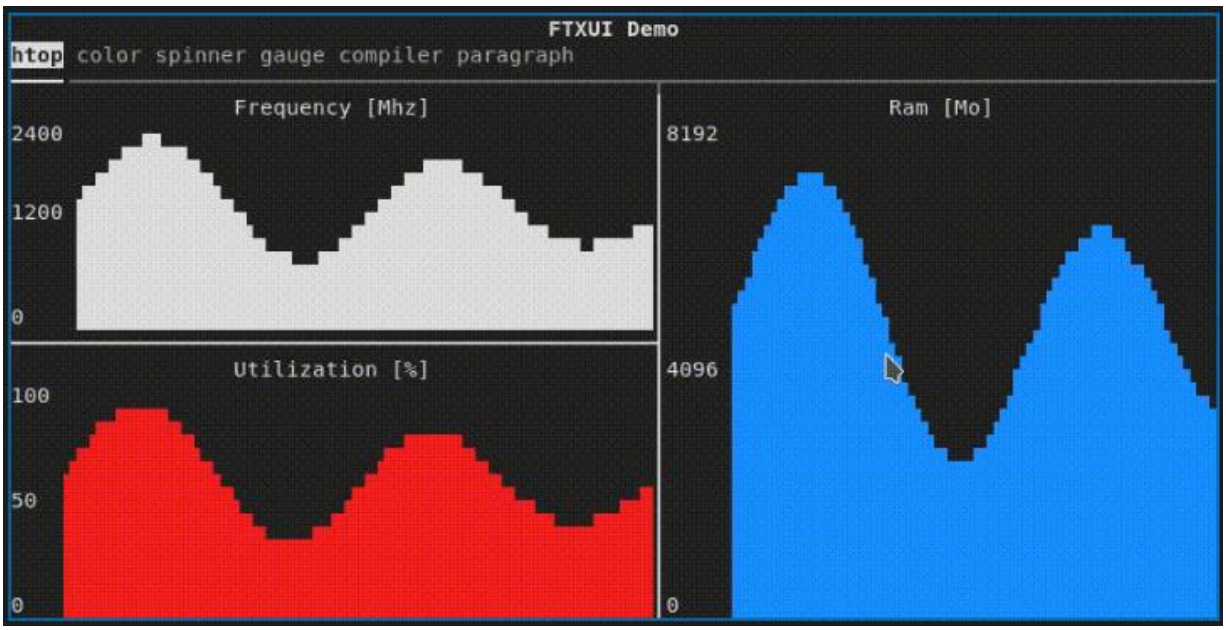


Рисунок 1.7 – Приклад графіки FTXUI[19]

Бібліотека notcurses, розроблена Ніком Блекмоном, позиціонується як спадкоємець ncurses із розширеними можливостями (графічні можливості представлені на рисунку 1.8). За описом авторів, це бібліотека для побудови складних термінальних інтерфейсів на сучасних термінальних емуляторах із підтримкою яскравих кольорів, мультимедіа, потоків та Unicode. Notcurses підтримує 24-розрядний колір, відображення зображень за допомогою протоколів Sixel та Kitty, а також роботу з лінійним кадровим буфером Linux [20].

Рисунок 1.8 – Приклад графіки Notcurses[20]

Бібліотека написана мовою Сі та поширюється під ліцензією Apache 2.0. До переваг можна віднести найвищий серед усіх аналогів рівень виражальних засобів термінальної графіки та орієнтацію на сучасні термінальні емулятори. Водночас вона тісно прив'язана до можливостей конкретного емулятора терміналу й передбачає наявність нестандартних розширень, таких як Sixel або Kitty graphics protocol, яких немає у класичних терміналах та більшості реалізацій консолі Windows.

Серед компактних бібліотек слід відзначити termbox – мінімалістичну Сі-бібліотеку, яка надає набір базових примітивів для виводу символів у комірки терміналу та обробки введення. Termbox свідомо відмовляється від багатьох можливостей ncurses на користь простоти API та малого обсягу коду [21]-[22]. Вона добре підходить для невеликих утиліт, проте не містить готових засобів побудови графіків та діаграм, домонстрація консольної графіки наваедена на рисунку 1.9.

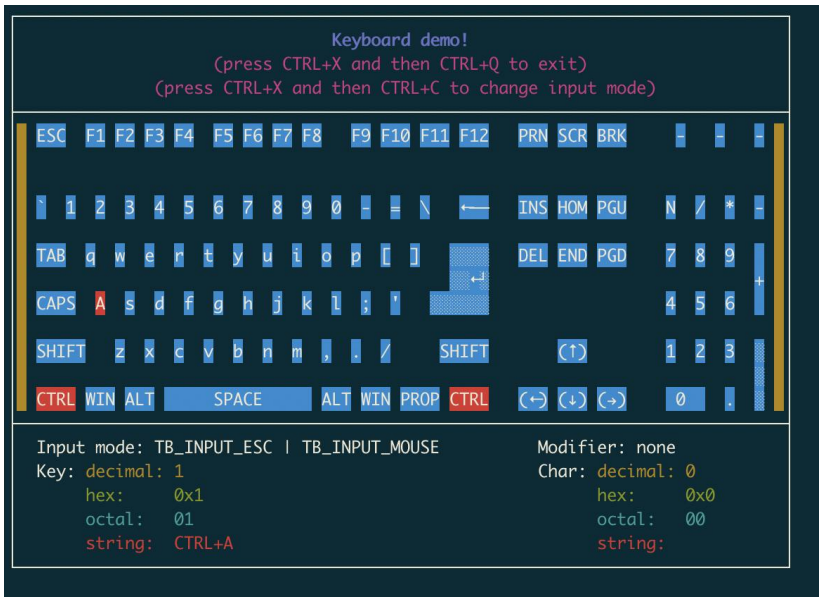


Рисунок 1.9 – Демо графічних можливостей бібліотеки termbox[22]

Окрему категорію аналогів становлять інструменти, реалізовані поверх інтерпретованих мов програмування. Для платформи Node.js популярним рішенням є blessed-contrib, що поєднує бібліотеку blessed для

роботи з терміналом та набір віджетів для побудови дашбордів, серед яких лінійні графіки, стовпчасті діаграми, спарклайни та таблиці (рисунок 1.10). Це дозволяє швидко зібрати функціональний моніторинговий інтерфейс у вікні терміналу, проте така реалізація прив'язана до екосистеми Node.js і потребує наявності інтерпретатора JavaScript у цільовій системі[23].

Рисунок 1.10 – Малювання діаграм у консолі за допомогою blessed-contrib[23]

Для мови Python існує найбільш розвинений набір засобів термінальної візуалізації. Найбільш відомою серед них є бібліотека Rich разом із її продовженням Textual, які надають широкі можливості форматованого виводу, побудови таблиць, прогрес-барів та дашбордів. Спеціально для побудови графіків призначена бібліотека plotext, яка підтримує діаграми розсіювання, лінійні графіки, стовпчасті діаграми, гістограми та навіть відображення зображень, приклад виводу графіки plotext наведено на рисунку 1.11[24].

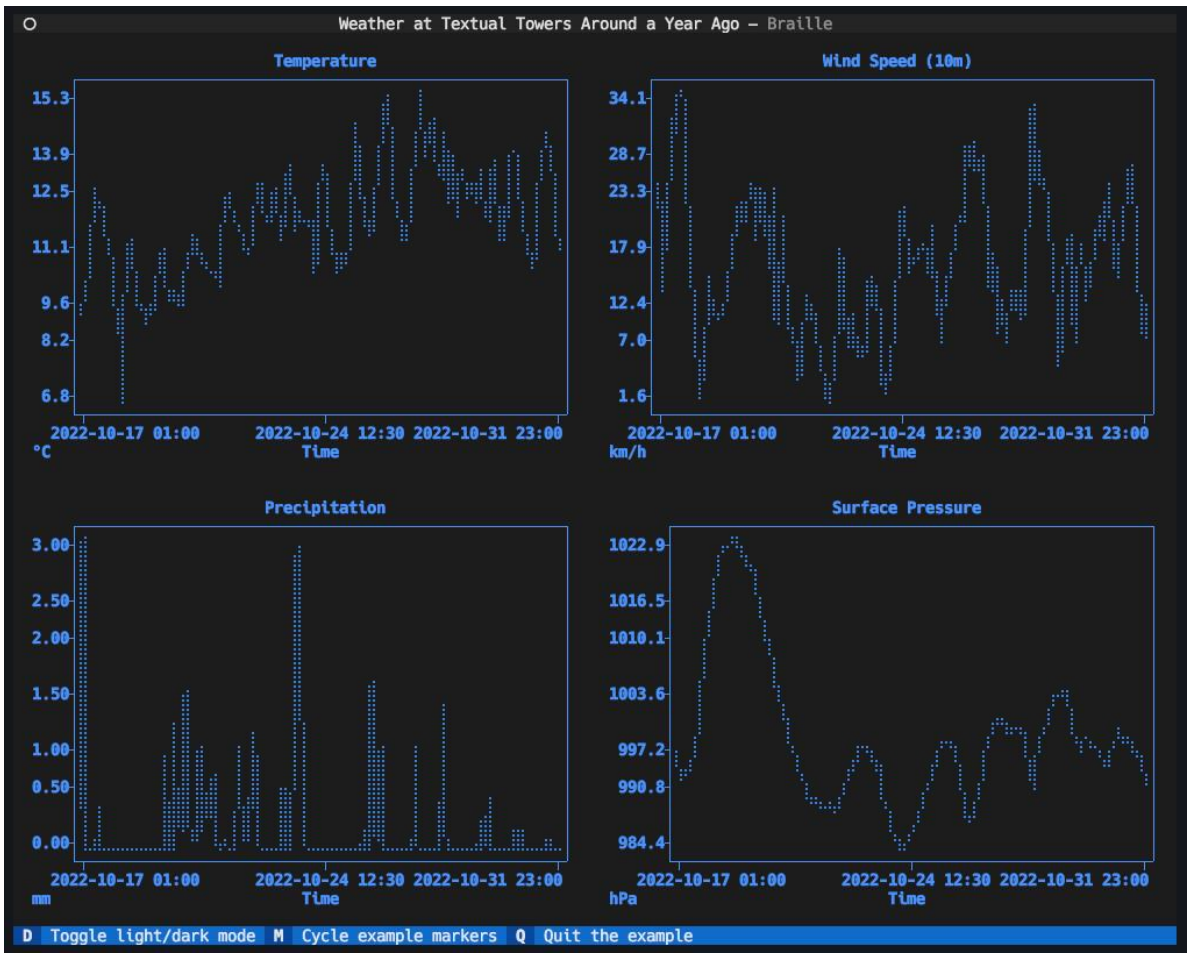


Рисунок 1.11 – Приклад виводу графіки за допомогою біліотеки plotext[24]

Бібліотека активно підтримується і особливо корисна для віддаленої роботи, коли неможливо запустити графічний застосунок. Її API свідомо повторює API matplotlib, що знижує поріг входу для користувачів, знайомих із класичним науковим стеком Python. Серед інших бібліотек цього класу можна назвати asciichart для побудови простих лінійних графіків з ASCII-символів, termplotlib як обгортку над gnuplot, bashplotlib та plotille, яка використовує символи Брайля для підвищення роздільної здатності. CopyProgramming

Також під час дослідження питання пошукові інстумаенти часто видавали программу gnuplot, яка має режим виводу dumb terminal та може формувати ASCII-зображення графіків. Хоча gnuplot не є бібліотекою, його часто використовують у консольних сценаріях моніторингу як зовнішню утиліту, виклик якої виконується з основного застосунку. Перевагою такого

підходу є висока якість зображень та зрілість продукту, проте він вимагає наявності окремого виконуваного файлу gnuplot у системі та не дозволяє інтегрувати побудову графіка безпосередньо у код прикладного застосунку.

Для зведеного аналізу розглянутих аналогів сформовано порівняльну таблицю (табл. 1.1), у якій представлено основні критерії оцінювання.

Таблиця 1.1 – Порівняння програмних аналогів консольної візуалізації

Назва	Платформи	Мова	Тип графіки	Орієнтація	Ліцензія
1	2	3	4	5	6
ncurses	Linux, macOS, BSD; Windows через PDCurses	C	Символьна (TUI)	Інтерфейси	MIT-подібна
FTXUI	Linux, macOS, Windows, WebAssembly	C++	Символьна, Unicode	Інтерфейси	MIT
notcurses	Linux, BSD	C	Символьна, Sixel, Kitty	Мультимедіа TUI	Apache 2.0
termbox	Linux, BSD	C	Символьна	Базові примітиви	MIT
blessed-contrib	Node.js	JavaScript	Символьна, ASCII-графіки	Дашборди	MIT
Rich / Textual	cross-platform	Python	Форматований текст	Інтерфейси	MIT
plotext	cross-platform	Python	ASCII-графіки, Брайль	Графіки даних	MIT
asciichart	cross-platform	Python	ASCII лінії	Прості графіки	MIT
gnuplot (dumb)	Кросплатформно	C	ASCII-графіки	Графіки даних	gnuplot

З отриманої таблиці можна виявити кілька закономірностей у наявному програмному забезпеченні консольної візуалізації. По-перше, існує

чіткий поділ бібліотек на дві групи за призначенням. До першої групи належать рішення, орієнтовані на побудову інтерфейсів користувача загального призначення, такі як ncurses, FTXUI, notcurses та termbox. Вони надають широкі можливості роботи з текстовими елементами, формами та подіями, але не містять спеціалізованих засобів побудови графіків даних. До другої групи належать інструменти, призначені саме для візуалізації числових даних, такі як plotext, asciichart та gnuplot. Вони добре розв'язують задачу побудови графіків, проте обмежені рамками своїх мов реалізації та екосистем.

По-друге, переважна більшість сучасних аналогів реалізована мовами Python та JavaScript, які потребують наявності у системі відповідного інтерпретатора. У контексті задач системного моніторингу та віддаленого адміністрування, де часто потрібно мати самодостатній виконуваний файл без зовнішніх залежностей, таке обмеження є істотним.

По-третє, жоден з розглянутих аналогів не використовує для виводу графіки прямий доступ до графічної підсистеми операційної системи Windows через інтерфейс GDI. Усі без винятку рішення, навіть ті, що офіційно підтримують Windows, побудовані на текстовому виводі через стандартний потік. Як було показано у підрозділі 1.2, такий підхід обмежує роздільну здатність зображення розміром символьної комірки терміналу. Альтернатива у вигляді піксельного малювання у вікні консолі Windows наявними бібліотеками не реалізована.

По-четверте, серед бібліотек для C++ задача побудови саме графіків даних практично не розв'язана. Існуючі рішення цієї мови, такі як FTXUI, орієнтовані на побудову інтерфейсів і не надають готових функцій для лінійних графіків, стовпчастих діаграм та інших типових елементів моніторингу. Для побудови повноцінного дашборду розробнику доводиться самостійно реалізовувати алгоритми відображення даних поверх примітивів цих бібліотек.

Отже, аналіз існуючих програмних аналогів показує, що жоден з них не поєднує одночасно таких властивостей як кросплатформність на рівні платформ Windows та Linux, реалізація мовою C++ без зовнішніх інтерпретаторів, наявність високорівневого API саме для побудови графіків та діаграм, а також можливість використання різних механізмів виводу залежно від середовища роботи (графічний інтерфейс на локальній машині та текстовий вивід у віддаленій сесії). Така потреба у наявному програмному забезпеченні зумовлює доцільність розробки власної програмної системи, постановка завдання якої сформульована у підрозділі 1.6.

1.4 Особливості Windows та Linux як цільових платформ

Згідно з отриманого аналізу вище, можливості реалізації графічного виводу у консольному застосунку істотно залежать від операційної системи, у середовищі якої цей застосунок виконується. Незважаючи на зовнішню схожість консолі Windows та терміналу Linux з погляду користувача, їхні внутрішні архітектури принципово різняться. Розуміння цих відмінностей є необхідним для обґрунтування технічних рішень, що стосуються кросплатформної реалізації виводу графіки. У цьому підрозділі розглянуто архітектурні особливості консольного середовища кожної з двох цільових платформ, а також обмеження, що випливають з їхньої будови.

Консольне середовище в операційній системі Microsoft Windows історично відрізняється від традиційних UNIX-терміналів. На відміну від UNIX-подібних систем, де термінал є абстракцією над символьним пристроєм виводу, у Windows консольне вікно являє собою повноцінний об'єкт графічного інтерфейсу. Воно створюється і керується спеціальним системним процесом, який відповідає за відображення тексту, обробку введення з клавіатури та реалізацію команд керування. У сучасних версіях Windows цю роль виконує процес conhost.exe, що запускається разом із кожним консольним застосунком та обслуговує його взаємодію з

користувачем. Кожен такий процес відкриває звичайне вікно операційної системи, у яке шрифтом фіксованої ширини виводяться символи з потоку виведення застосунку.

З огляду на те, що консольне вікно у Windows є об'єктом графічного інтерфейсу, воно має свій унікальний дескриптор вікна типу HWND. Цей дескриптор використовується системою для адресації вікна під час обробки повідомлень, перемальовування та інших операцій. Через функцію GetConsoleWindow, що належить до Win32 API, прикладний застосунок має змогу отримати дескриптор власного консольного вікна. Це принципово важливий аспект, оскільки наявність HWND відкриває доступ до всіх можливостей графічного інтерфейсу Windows, які стандартно застосовуються до звичайних віконних застосунків. Зокрема, отримавши HWND, застосунок може звернутися до контексту пристрою цього вікна та малювати в ньому довільну графіку.

Універсальним механізмом малювання у графічному інтерфейсі Windows слугує програмний інтерфейс GDI, що входить до складу Win32 API. Цей інтерфейс надає набір функцій для роботи з графічними примітивами, серед яких лінії, прямокутники, еліпси, дуги, текст та растрові зображення. GDI працює з абстрактним поняттям контексту пристрою, яке інкапсулює властивості поверхні малювання та дозволяє писати один і той самий код для виводу як на екран, так і на принтер чи у файл. Підтримка GDI присутня у всіх версіях Windows, починаючи з найдавніших, та залишається стандартним інструментом малювання у класичних настільних застосунках. У контексті консольних застосунків Windows унікальна можливість поєднання GDI з консольним вікном відкриває шлях до піксельного малювання безпосередньо у вікні консолі без створення окремих графічних форм.

Слід окремо зауважити, що в останні роки у середовищі Windows з'явився сучасний емулятор терміналу Windows Terminal, архітектура якого відрізняється від традиційного conhost. Цей емулятор побудований за

моделлю окремого графічного застосунку, що приєднує до себе процеси консольних утиліт як вкладені сесії. У такій архітектурі поняття дескриптора консольного вікна стає менш однозначним, оскільки кожен консольний процес вже не має власного видимого вікна, а є гостем у вікні термінального застосунку. Це потенційно ускладнює пряме застосування GDI до Windows Terminal, однак для класичного conhost механізм залишається повністю працездатним. Поряд з цим, починаючи з версії Windows 10, у системі реалізовано підтримку обробки ANSI escape-послідовностей у потоці виведення консолі. Це означає, що Windows здатна виконувати роль обох середовищ одночасно: класичної консолі з доступом через GDI та сучасного терміналу з підтримкою керівних послідовностей у стилі VT100[25]-[27].

В операційній системі Linux архітектура консольного середовища ґрунтується на принципово інших засадах. Тут немає поняття системного консольного вікна, яке належить процесу. Натомість будь-яка програма, що працює з текстовим введенням-виведенням, взаємодіє з абстрактним пристроєм, відомим як термінал. Цей пристрій реалізований у ядрі операційної системи у вигляді спеціального драйвера. Існують два основні різновиди такого пристрою: фізичний термінал tty, що відповідає сесії на текстовій консолі ядра, та псевдотермінал pty, який емулюється програмно та використовується для організації роботи у графічних емуляторах терміналу або у віддалених сесіях по протоколу SSH.

З погляду застосунку взаємодія з терміналом виглядає як звичайне читання та запис у файлові дескриптори стандартних потоків. Все, що програма виводить у стандартний потік, передається у термінальний пристрій, звідки потрапляє до того процесу, який цей термінал обслуговує. У випадку текстової консолі ядра таким процесом є саме ядро, яке безпосередньо виводить символи на відеоадаптер. У випадку графічного середовища роль обслуговувача виконує програма-емулятор терміналу. Серед найбільш поширених емуляторів терміналу для Linux слід назвати xterm, gnome-terminal, konsole, urxvt, alacritty та kitty. Кожен з них самостійно

інтерпретує вхідний потік символів від застосунку та відображає його у графічному вікні засобами власного коду, незалежного від ядра.

Принциповою особливістю архітектури Linux у цьому контексті є те, що застосунок не має жодного прямого зв'язку з графічним вікном, у якому відображається його вивід. Між програмою та екраном перебуває щонайменше дві ланки: ядерний драйвер псевдотерміналу та процес емулятора. Процес-застосунок не знає, чи його вивід відображається у вікні графічного середовища, чи у текстовій консолі ядра, чи передається через мережу до клієнта SSH на іншому комп'ютері. Усі ці варіанти з погляду застосунку є рівноцінними та виглядають як запис байтів у файловий дескриптор. Така архітектура забезпечує надзвичайну гнучкість і прозорість віддаленої роботи, але водночас принципово виключає можливість прямого графічного малювання у вікні терміналу засобами застосунку.

Єдиним каналом передачі форматування та керування у середовищі Linux є саме потік символів, що надсилається у термінал. Для розширення можливостей цього потоку існує стандарт керівних послідовностей, що бере свій початок від терміналу VT100 і пізніше формалізований у документі ECMA-48. Цей стандарт визначає формат escape-послідовностей, за допомогою яких застосунок може повідомити емулятору терміналу про необхідність змінити колір тексту, перемістити курсор, очистити частину екрана та виконати інші операції форматування. Сучасні емулятори терміналу для Linux підтримують повний набір команд цього стандарту, а також низку розширень, таких як 24-розрядний кольоровий режим. Підтримка ANSI-послідовностей є відповідальністю саме емулятора, тому варіації у поведінці окремих його реалізацій є нормою.

З наведеного опису випливають принципові обмеження кожної з двох розглянутих платформ для задачі графічного виводу у консолі. У середовищі Windows прикладний застосунок має технічну змогу безпосередньо малювати у власному консольному вікні засобами GDI завдяки наявності HWND і доступу до контексту пристрою. Це дозволяє досягти піксельної

точності зображення та якості, порівнянної зі звичайними віконними застосунками. Однак така можливість обмежена локальним середовищем виконання та архітектурою класичного conhost. Вона недоступна при роботі через віддалені сесії, оскільки клієнт не має доступу до серверного графічного контексту, а також може поводитися непередбачувано у сучасних емуляторах терміналу на кшталт Windows Terminal.

У середовищі Linux, навпаки, прямий графічний доступ до вікна терміналу принципово неможливий незалежно від того, локальною чи віддаленою є робота. Усе, що застосунок може передати у термінал, обмежене потоком символів та керівних послідовностей. Це накладає обмеження на роздільну здатність зображень, оскільки мінімальною одиницею виводу є символьна комірка терміналу, розмір якої залежить від обраного шрифту та налаштувань емулятора. Водночас саме така архітектура забезпечує повну прозорість віддаленої роботи через SSH, оскільки байти, що передаються у мережу, не відрізняються від тих, що надсилаються у локальний термінал[28]-[30].

Ключова відмінність двох платформ підсумовується наступним чином. У Windows консольне вікно є об'єктом графічного інтерфейсу з власним дескриптором, що уможливлює прямий графічний доступ. У Linux термінал є абстракцією над символьним пристроєм виводу, і будь-яка взаємодія з ним відбувається виключно через потік символів. Ця різниця має фундаментальний характер та не може бути усунена програмними засобами на рівні застосунку. Вона походить від відмінностей у самій моделі побудови консольного середовища у двох операційних системах та зумовлює необхідність застосування різних технічних підходів для реалізації графічного виводу на кожній з платформ. Принципи побудови такого роздільного підходу, а також спосіб об'єднання двох технічно різних реалізацій у межах єдиного програмного продукту, розглянуто у наступних розділах роботи.

1.5 Дані моніторингу як об'єкт візуалізації

Під моніторингом у контексті дипломної роботи бакалавра є процес неперервного або періодичного спостереження за певними кількісними показниками, що характеризують стан або поведінку об'єкта спостереження. Найбільш відомою сферою застосування моніторингу є відстеження стану обчислювальних систем, у межах якого збираються показники використання процесора, оперативної пам'яті, мережевого трафіку, дискових операцій та інших системних ресурсів. Однак сферою моніторингу не вичерпуються лише системні ресурси, і в сучасних інформаційних системах потреба у відображенні числових даних виникає в значно ширшому колі задач.

До категорії об'єктів моніторингу можна віднести низку різноманітних джерел даних, що відрізняються за своєю природою та способом отримання значень. Першу і найбільш очевидну групу становлять метрики самої операційної системи. У середовищі Linux вони традиційно надаються у вигляді віртуальних файлів файлової системи /proc, а також через утиліти top, vmstat, iostat та ряд інших. У середовищі Windows аналогічні дані доступні через підсистему лічильників продуктивності, відому як Performance Counters і доступну через інтерфейс PDH (Performance Data Helper). Друга група об'єктів моніторингу – це метрики прикладних застосунків. Сюди належать показники, які прикладна програма формує та публікує самостійно, серед яких частота надходження запитів, кількість оброблених транзакцій, тривалість виконання операцій, кількість активних з'єднань та інші характеристики, специфічні для конкретного застосунку.

Третю групу складають метрики зовнішніх систем, з якими взаємодіє застосунок. До них можна віднести показники роботи баз даних, такі як кількість запитів за одиницю часу або середній час відгуку, метрики мережевих сервісів API, стани черг повідомлень у системах обміну даними між компонентами розподіленої системи, показники зовнішніх вебсервісів. Четверта група охоплює дані з апаратних датчиків та сенсорів, що особливо

актуальні для систем інтернету речей та промислової автоматизації. Це можуть бути значення температури, вологості, тиску, швидкості руху, рівня освітлення та подібні фізичні величини. Останнім, до даних моніторингу можна зарахувати показники, що отримуються шляхом аналізу журнальних файлів та структурованих повідомлень, серед яких частота появи подій певного типу, кількість помилок або попереджень за певний інтервал часу.

Ця різноманітність джерел демонструє, що задача збору даних моніторингу принципово відрізняється від задачі їх відображення. Збір даних завжди прив'язаний до конкретного джерела і потребує специфічного коду інтеграції з ним. Натомість відображення даних має універсальний характер і може здійснюватися однаковим чином для довільних числових значень, незалежно від того, з якого джерела вони отримані. Розподіл відповідальності між компонентом, що збирає дані, та компонентом, що їх візуалізує, є природним архітектурним підходом, який дозволяє створити універсальну систему візуалізації, придатну для роботи з найширшим спектром моніторингових сценаріїв. Саме на задачі візуалізації, що приймає на вхід вже підготовлені числові дані, зосереджена розробка, виконана у дипломній роботі.

Незалежно від конкретного джерела, дані моніторингу можна звести до кількох базових типів, кожен з яких має властиві йому особливості подання. Найбільш поширеним типом є часові ряди, послідовності числових значень, прив'язані до моментів часу. До цього типу належать показники навантаження на процесор, обсяги переданого мережею трафіку, частоти надходження запитів та практично всі інші метрики, що змінюються у реальному часі. Другим важливим типом є категоріальні дані, що являють собою набір значень, прив'язаних не до часової шкали, а до дискретної множини категорій. Прикладами таких даних можуть бути обсяги пам'яті, зайняті різними процесами, кількість запитів до різних серверних вузлів, розподіл помилок за типами. Третім типом, що часто супроводжує перші два, є порогові значення. Це опорні рівні, відносно яких оцінюється поточний

стан показника, наприклад, межа критичного завантаження процесора або максимально допустимий час відгуку.

Природа даних безпосередньо визначає вибір типу візуального подання. Для часових рядів традиційно використовують лінійні графіки, на яких горизонтальна вісь відповідає часовій шкалі, а вертикальна – значенню показника. Такі графіки добре передають динаміку змін та дозволяють візуально оцінити тенденції. Для категоріальних даних найбільш придатними є стовпчасті діаграми, що дозволяють порівнювати значення різних категорій за висотою або довжиною стовпців. Для відображення компактних узагальнених тенденцій у обмеженому просторі застосовують спарклайни – мініатюрні графіки без осей та підписів, призначені для розміщення поряд з основним показником. Порогові значення зазвичай зображуються у вигляді горизонтальних опорних ліній, накладених поверх лінійного графіка, або у вигляді кольорових індикаторів, що змінюють відтінок при перевищенні критичного рівня. Для відображення двовимірних розподілів даних застосовуються теплові карти, у яких значення кодуються кольоровою інтенсивністю комірок прямокутної сітки.

Окрім вибору типу візуалізації, специфіка задач моніторингу формує низку вимог до програмних засобів їх реалізації. Першою з цих вимог є підтримка оновлення зображення у реальному часі. Дані моніторингу, як правило, надходять безперервно, і застосунок має відображати їх з мінімальною затримкою, щоб користувач міг своєчасно реагувати на зміни. Це накладає обмеження на швидкість перемальовування екрана та обумовлює потребу в оптимізованих алгоритмах оновлення. Другою вимогою є низьке споживання ресурсів самим засобом візуалізації. Цей пункт особливо актуальний для серверних середовищ, де моніторинговий застосунок не повинен забирати помітну частку обчислювальних потужностей у системи, за якою він наглядає. Споживання процесорного часу та оперативної пам'яті засобом візуалізації має бути несумірно меншим

за рівні, що відображаються на графіках, інакше сам моніторинг ставатиме джерелом викривлення картини.

Третьою важливою вимогою є збереження читабельності зображення на малих розмірах вікна. Консольне середовище, як правило, не передбачає розгортання вікна на повний екран, а у віддалених сесіях SSH розмір терміналу часто обмежений налаштуваннями клієнта. Тому графічні елементи мають правильно масштабуватися і залишатися інформативними при невеликих розмірах. Четвертою вимогою, що випливає з самої природи консольних застосунків, є мінімізація залежностей. Засіб візуалізації повинен працювати у будь-якому консольному середовищі без додаткового встановлення компонентів графічного інтерфейсу, бібліотек відображення зображень чи мережевих сервісів передачі картинки. Дотримання усіх цих вимог є критерієм придатності програмної системи для практичного використання у задачах моніторингу.

Підсумовуючи, дані моніторингу являють собою широкий клас числових показників, що походять з різноманітних джерел і характеризуються кількома базовими типами. Програмна система візуалізації, що розробляється у межах дипломної роботи, орієнтована саме на роботу з вже підготовленими числовими даними і не залежить від способу їх отримання. Це робить її універсальним інструментом, придатним для застосування у широкому колі задач моніторингу – від спостереження за системними ресурсами до відстеження прикладних, мережевих та апаратних показників.

1.6 Висновки за розділом

Підсумовуючи, аналіз предметної можна сформулювати конкретні вимоги до програмної системи, що розроблюється у межах цієї кваліфікаційної роботи. Вимоги визначають основні характеристики готового продукту та задають напрям подальших етапів проєктування, реалізації і

тестування. Узагальнення висновків з підрозділів 1.1-1.5 дає підстави стверджувати, що у наявнних програмнтх забезпеченнях відсутнє рішення, яке поєднувало б реалізацію мовою C++, кросплатформну роботу на платформах Windows та Linux, а також єдиний програмний інтерфейс для побудови графіків і діаграм у консольному середовищі. Це зумовлює доцільність розробки такої системи та формує основні завдання роботи.

Система має забезпечувати кросплатформну роботу на двох цільових платформах Microsoft Windows та GNU/Linux без зміни коду прикладного застосунку, що використовує її засоби. Це передбачає реалізацію двох незалежних механізмів виводу, кожен з яких враховує специфіку відповідної платформи, та об'єднання їх у межах єдиного програмного інтерфейсу.

Для платформи Windows необхідно реалізувати механізм виводу графіки на основі прямого малювання у консольному вікні засобами інтерфейсу GDI. Цей механізм повинен забезпечувати піксельну точність зображення, плавне оновлення без мерехтіння за рахунок подвійної буферизації та коректну поведінку при зміні розмірів вікна.Для платформи Linux необхідно реалізувати механізм виводу на основі ANSI escape-послідовностей у поєднанні з псевдографічними символами Unicode. Реалізація має підтримувати 24-розрядний кольоровий режим сучасних термінальних емуляторів, забезпечувати оптимізоване оновлення екрана через diff-рендер та залишатися працездатною у віддалених сесіях через протокол SSH.

Архітектура системи повинна базуватися на абстрактному програмному інтерфейсі, що приховує від прикладного коду відмінності у механізмах виводу. Такий підхід забезпечує єдність користувацького коду на обох платформах та водночас залишає можливість додавання нових бекендів виводу у майбутньому без зміни існуючої функціональності.

Програмний інтерфейс системи повинен містити функції побудови типових елементів моніторингової візуалізації, серед яких лінійні графіки, стовпчасті діаграми, координатні сітки, осі та порогові лінії. Один і той

самий виклик функцій має давати функціонально еквівалентний результат на обох платформах, незалежно від механізму виводу, що використовується у поточному середовищі.

Система має бути самодостатньою та не потребувати наявності у цільовій системі зовнішніх графічних залежностей, інтерпретаторів або компонентів графічного інтерфейсу. Це забезпечує можливість використання системи на серверах без графічної підсистеми, у віддалених терміналах та у середовищах із обмеженими можливостями встановлення додаткового програмного забезпечення.

Сформульовані вимоги визначають перелік завдань, що мають бути виконані для досягнення поставленої мети роботи. До цих завдань належать дослідження двох методів графічного виводу у консольному середовищі та експериментальна перевірка прототипів на цільових платформах, проєктування архітектури системи з підтримкою змінних бекендів виводу, реалізація бекенду Windows GDI та бекенду ANSI, розробка уніфікованого програмного інтерфейсу візуалізації, а також тестування розробленої системи з оцінкою її продуктивності та практичної придатності.

2 ДОСЛІДЖЕННЯ МЕТОДІВ КОНСОЛЬНОЇ ГРАФІКИ

На етапі проєктування фінальної архітектури програмної системи, необхідно експериментально дослідити кожен з двох підходів до графічного виводу у консольному середовищі, які було теоретично окреслено у першому розділі. В цьому розділі ніведені практичні перевірки можливостей та обмежень двох обраних механізмів виводу: інтерфейсу Windows GDI для платформи Microsoft Windows та ANSI escape-послідовностей у поєднанні з Unicode-псевдографікою для платформи Linux. Перевірка полягає у побудові робочих прототипів, що реалізують типові елементи моніторингової візуалізації засобами кожного з підходів, та у фіксації виявлених під час реалізації технічних особливостей. Отримані результати слугуватимуть основою для побудови архітектури ПЗ.

2.1 Графічний вивід через Windows GDI

Метод графічного виводу через інтерфейс Windows GDI ґрунтується на тій принциповій можливості, що консольне вікно у Microsoft Windows є об'єктом графічного інтерфейсу та має відповідний дескриптор вікна. Цей дескриптор може бути отриманий прикладним застосунком за допомогою функції GetConsoleWindow з Win32 API, після чого вікно стає доступним для використання усіх стандартних механізмів малювання операційної системи. У цьому підрозділі описано принципи реалізації графічного виводу за такою схемою, розглянуто механізм подвійної буферизації та зафіксовано виявлені у процесі експериментальної реалізації обмеження. Повний вихідний код прототипу разом з усіма проведеними експериментами та тестами наведено у Додатку А у файлі testGDI.cpp.

Загальний принцип ініціалізації графічного контексту складається з кількох послідовних кроків, схематично зображених на рисунку 2.1. На першому кроці викликається функція GetConsoleWindow, яка повертає

дескриптор вікна консолі активного процесу типу HWND. На другому кроці отримується контекст пристрою цього вікна за допомогою функції GetDC, що повертає об'єкт типу HDC. Контекст пристрою є абстракцією поверхні малювання та інкапсулює властивості, необхідні для виконання графічних операцій, серед яких поточне перо, кисть, шрифт та режим змішування кольорів. Розмір клієнтської області визначається функцією GetClientRect, що повертає координати її меж у пікселях.

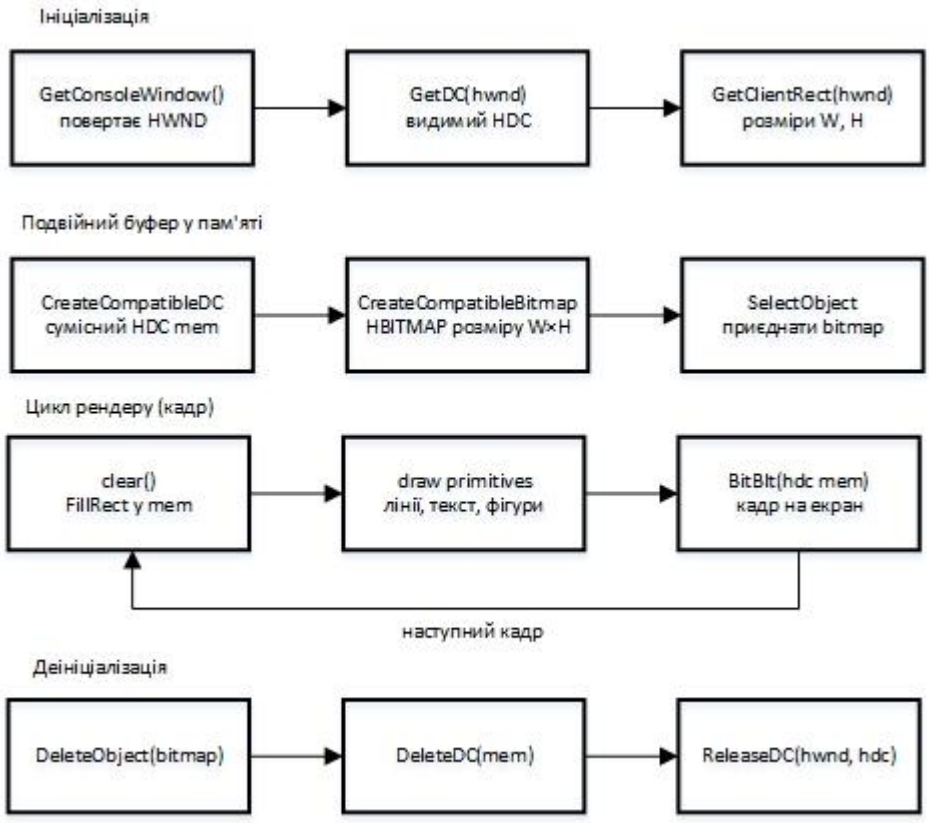


Рисунок 2.1 – Послідовність викликів GDI: від ініціалізації до циклу рендеру

Безпосереднє малювання у видимому контексті пристрою є технічно можливим, проте непридатне для застосунків з частим оновленням зображення. Кожна операція призводить до миттєвого відображення результату на екрані, і при оновленні всього кадру за множинними викликами користувач спостерігає мерехтіння та проміжні стани зображення. Для уникнення цього застосовується класична техніка подвійної буферизації, що передбачає виконання усіх графічних операцій у невидимій області

пам'яті з подальшим одноразовим перенесенням готового кадру у видиме вікно.

Реалізація подвійної буферизації засобами GDI здійснюється через створення сумісного контексту пристрою функцією CreateCompatibleDC та сумісного растрового зображення функцією CreateCompatibleBitmap. Растрове зображення приєднується до сумісного контексту викликом SelectObject. Усі подальші виклики функцій малювання, що передаються у сумісний контекст, не призводять до видимих змін на екрані, оскільки результат накопичується у пам'яті. Перенесення готового кадру у вікно виконується одноразовим викликом функції BitBlt. Завдяки тому, що ця операція виконується одним викликом і реалізована на рівні драйвера відеоадаптера, користувач спостерігає миттєву заміну попереднього кадру на новий без проміжних станів.

Інтерфейс GDI надає широкий набір функцій для роботи з графічними примітивами, серед яких лінії, прямокутники, еліпси, багатокутники та текст. Малювання лінії здійснюється парою функцій MoveToEx та LineTo. Прямокутник з контуром та заливкою виводиться функцією Rectangle, а заповнення області суцільним кольором без контуру забезпечується функцією FillRect. Для виводу довільних багатокутників застосовується функція Polygon, що приймає масив координат вершин. Еліпси та кола малюються функцією Ellipse. Текст виводиться функцією TextOutA з використанням поточного шрифту та кольору, встановленого функцією SetTextColor. Створення власних пер, кистей та шрифтів виконується функціями CreatePen, CreateSolidBrush та CreateFontA[27].

У межах експериментальної реалізації прототипу було побудовано демонстраційний застосунок, що виконує інтерактивне відображення моніторингового дашборда, домостріцію рендеру GDI переставлено на рисунку 2.2.

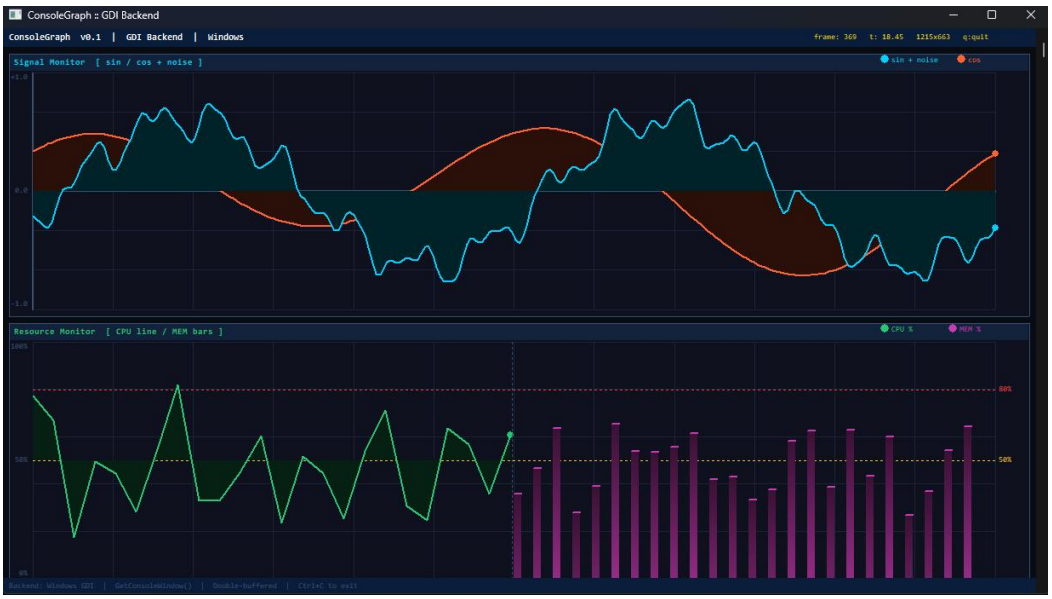


Рисунок 2.2 – Результат рендеру GDI

Звичайний Застосунок реалізує дві логічні панелі – Signal Monitor та Resource Monitor що відповідають типовим сценаріям моніторингової візуалізації. Перша панель демонструє побудову лінійних графіків двох сигналів синусоїдальної природи з накладеним шумом, друга панель – комбіноване відображення часового ряду у вигляді лінійного графіка та категоріальних значень у вигляді стовпчастої діаграми. Обидві панелі містять координатні сітки, осі з підписами, заливку під кривими, легенди та порогові лінії.

У результаті реалізації прототипу вдалося підтвердити повну функціональну придатність методу для розв'язання поставленої задачі. Піксельне малювання забезпечує відсутність будь-яких обмежень, пов'язаних з символьною сіткою. Шрифт виводиться зі згладжуванням за рахунок використання режиму CLEARTYPE_QUALITY. Анімація зображення зі швидкістю тридцяти кадрів за секунду відбувається плавно, без помітних мерехтінь чи розривів. Реакція на зміну розмірів консольного вікна реалізована через періодичну перевірку розміру клієнтської області у головному циклі та повторне створення сумісного растрового зображення відповідного розміру.

Під час практичної реалізації було виявлено низку обмежень, що супроводжують застосування цього методу. Передусім слід зазначити жорстку прив'язку до операційної системи Microsoft Windows, оскільки усі задіяні функції належать виключно до Win32 API і не мають аналогів в інших системах. Жоден рядок коду, написаного на основі цього методу, не може бути перенесений на платформу Linux без повної заміни графічної підсистеми.

Другим суттєвим обмеженням є залежність методу від конкретної реалізації консольного вікна. Метод повністю працездатний у середовищі класичного процесу-обслуговувача consolehost.exe, який використовується у Windows за замовчуванням. У сучасному емуляторі Windows Terminal архітектура розгортання консолі є принципово іншою: окремий консольний процес не має власного видимого вікна, а виступає як вкладена сесія всередині термінального застосунку. У такій конфігурації виклик GetConsoleWindow повертає дескриптор прихованого фонового вікна, малювання у якому не дає видимого результату. Таким чином, метод гарантовано працює лише у класичній консолі conhost.

Третє обмеження пов'язане з необхідністю ретельного керування ресурсами GDI. Кожне створене перо, кисть, шрифт або растрове зображення є системним об'єктом, що залишається у пам'яті до явного звільнення функцією DeleteObject. Створення цих об'єктів у головному циклі без подальшого звільнення призводить до накопичення невідпускних ресурсів і вичерпання обмеженого пулу GDI-об'єктів операційної системи. У реалізації прототипу ця особливість врахована за рахунок симетричного виклику відповідних функцій звільнення після кожного використання тимчасових об'єктів.

Четвертим обмеженням, що випливає з самої природи прямого графічного доступу до вікна, є несумісність методу з режимом віддаленої роботи через протокол SSH. Оскільки малювання здійснюється

безпосередньо у графічному контексті серверного вікна, клієнт SSH не отримує жодних даних про сформоване зображення.

Узагальнюючи описані спостереження, можна стверджувати, що метод графічного виводу через Windows GDI забезпечує найвищу якість зображення серед розглянутих підходів та повністю задовольняє вимоги до моніторингової візуалізації у локальному середовищі класичної консолі Windows. Водночас його обмеженість єдиною платформою, залежність від конкретної реалізації консолі та неможливість роботи у віддалених сесіях не дозволяють використовувати цей метод як єдиний механізм виводу у кросплатформній системі.

2.2 Текстовий вивід через ANSI escape-послідовності

Метод текстового виводу через ANSI escape-послідовності ґрунтується на тому, що сучасні термінальні емулятори інтерпретують спеціальні керівні послідовності у вхідному потоці символів як команди форматування виводу. Цей механізм не вимагає від застосунку прямого доступу до графічного контексту вікна та працює виключно через стандартний потік виведення, що робить його придатним як для локальної роботи, так і для віддалених сесій через протокол SSH. Повний вихідний код прототипу разом з усіма проведеними експериментами наведено у Додатку А у файлі testdiff.cpp.

Принцип роботи ANSI escape-послідовностей полягає у тому, що серед звичайних символів, які надсилаються у потік виведення, можуть зустрічатися спеціальні службові послідовності, що починаються з керівного символа ESC з кодом 0x1B. Емулятор терміналу при отриманні такої послідовності не виводить її як текст, а виконує відповідну дію: переміщує курсор у задану позицію, змінює колір тексту або фону, очищує екран або керує видимістю курсору. Серед найбільш уживаних команд можна назвати позиціонування курсору CUP, керування атрибутами SGR, що змінює колір

символів, а також команди очищення екрана. Сучасні термінальні емулятори підтримують розширений режим відображення кольору, відомий як 24-розрядний truecolor, що задається послідовністю SGR з прапорцем для кольору тексту або для кольору фону, після яких передаються три параметри значень компонентів RGB. Ця можливість дозволяє відтворювати кольорову палітру, порівнянну з повноекранними графічними застосунками[28].

Безпосередній вивід escape-послідовностей у стандартний потік для оновлення кожного символа окремо є непридатним для застосунків з частим оновленням зображення, оскільки повне перемальовування тисяч комірок на кожному кадрі створює значне навантаження на термінальний емулятор та канал передачі даних у віддаленій сесії. Для уникнення цього застосовується підхід, аналогічний за ідеєю до подвійної буферизації у GDI, але реалізований засобами символьного буфера. Прикладний застосунок підтримує у пам'яті двовимірний масив комірок, що містить виведений символ, колір переднього плану та колір фону. Усі операції побудови графічних елементів змінюють вміст цього буфера, не виводячи нічого у потік. Лише після завершення формування кадру виконується процедура передачі вмісту буфера у термінал з застосуванням diff-рендеру, принцип роботи якого схематично зображено на рисунку 2.3.

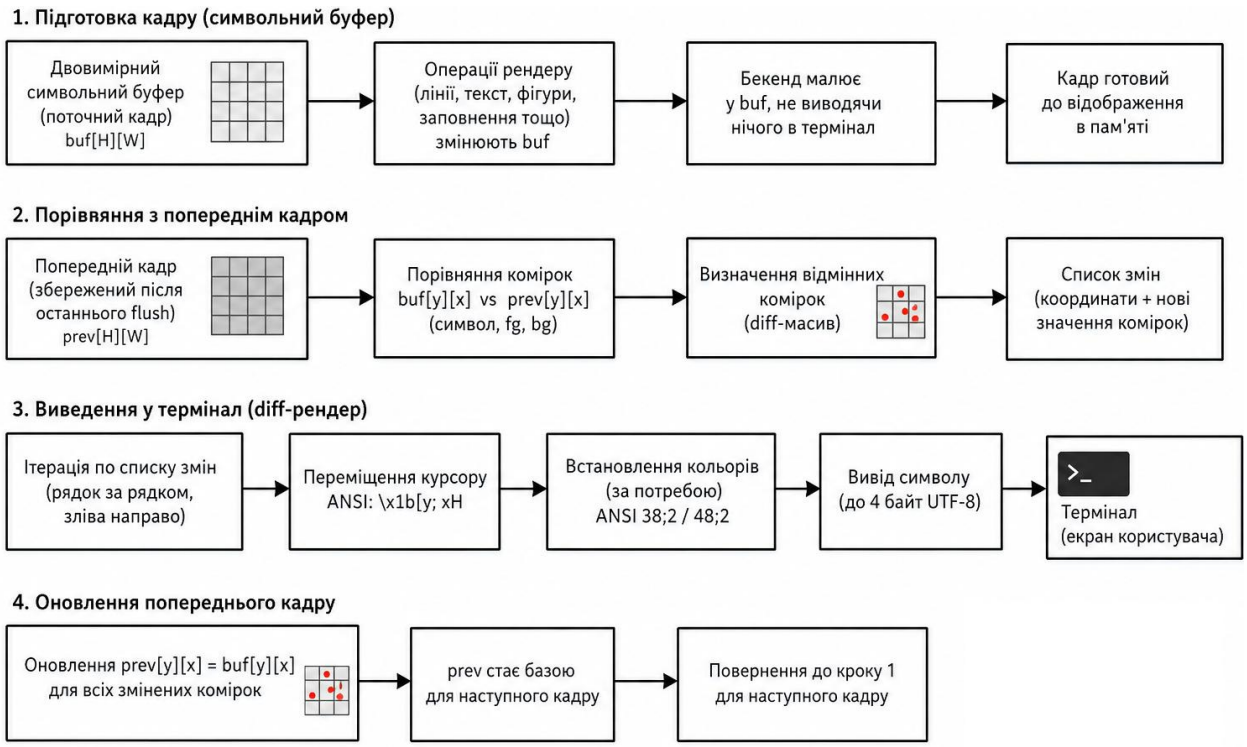


Рисунок 2.3 — Схема роботи, етапи diff-рендеру в ANSI-бекенді

Diff-рендер передбачає порівняння поточного буфера з попереднім та формування escape-послідовностей лише для тих комірок, вміст яких змінився між кадрами. У типових сценаріях моніторингової візуалізації від кадру до кадру змінюється лише невелика частина екрана, що відповідає областям з оновленими графіками, тоді як статичні елементи інтерфейсу залишаються незмінними. Це дозволяє скоротити обсяг даних, що передаються у термінал, у десятки разів порівняно з повним перемальовуванням. Додаткова оптимізація передбачає оновлення атрибутів кольору лише при їх зміні відносно попередньої позиції.

Дискретність символьного представлення є основним обмеженням методу, оскільки мінімальною одиницею виводу залишається одна комірка терміналу. Для часткового подолання цього обмеження застосовується спеціальна техніка, що використовує блокові символи Unicode для імітації пікселів. Зокрема, символи верхньої та нижньої половини блоку дозволяють розділити кожну комірку терміналу на дві логічні точки, виведення яких з

різними кольорами фактично подвоює вертикальну роздільну здатність зображення.

У межах експериментальної реалізації прототипу побудовано демонстраційний застосунок, функціонально еквівалентний прототипу для платформи Windows. Прототип скомпільовано компілятором GCC та запущено у терміналі віртуальної машини з Linux. Результат рендеру наведено на рисунку 2.4.

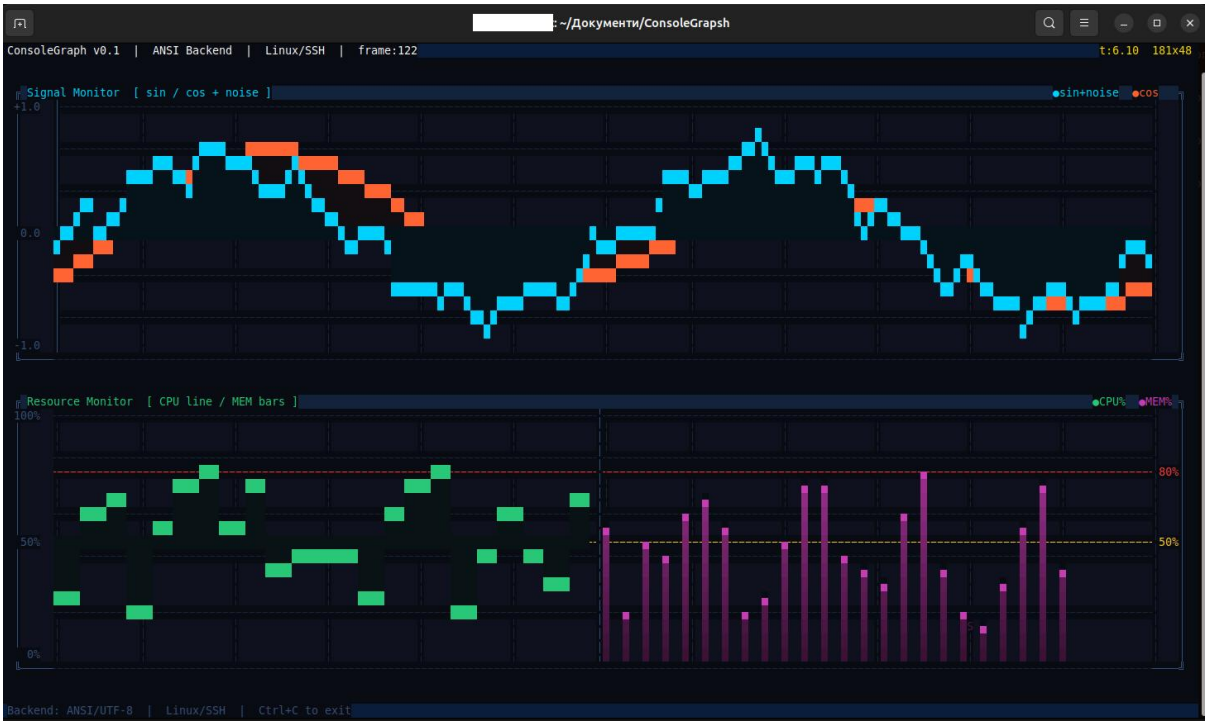


Рисунок 2.4 – Результат рендеру ANSI-бекенду в терміналі Linux

Програма реалізує ті самі дві логічні панелі, що й прототип на платформі Windows: Signal Monitor з лінійними графіками синусоїдальних сигналів та Resource Monitor з комбінованим відображенням лінійного графіка та стовпчастої діаграми. Обидві панелі містять координатні сітки, осі з підписами, легенди, порогові лінії та заливку під кривими. Уся графіка побудована блоковими символами Unicode з використанням 24-розрядного кольорового режиму.

У результаті реалізації прототипу вдалося підтвердити функціональну придатність методу для розв'язання поставленої задачі. Усі типові елементи

моніторингового дашборда успішно відтворюються засобами символьного виводу, кольорова палітра truecolor забезпечує візуальну виразність зображення, а оновлення екрана відбувається у реальному часі без помітних затримок завдяки застосуванню diff-рендеру. Метод працює однаково як у локальному термінальному емуляторі, так і у віддалених сесіях через SSH.

Поряд з підтвердженням працездатності методу було виявлено низку обмежень. Першим і найбільш фундаментальним є дискретність розміру комірки терміналу. Незважаючи на застосування блокових символів для подвоєння вертикальної роздільної здатності, мінімальна одиниця зображення залишається помітно більшою за окремий піксель, що призводить до східчастого вигляду похилих ліній. Другим обмеженням є залежність якості зображення від ширини шрифту терміналу: зміна шрифту може спричинити викривлення пропорцій графіка та появу видимих проміжків між суміжними блоковими символами. Третім обмеженням є залежність від коректної підтримки кодування UTF-8 термінальним емулятором, оскільки блокові символи Unicode потребують повної підтримки багатобайтового кодування на всьому шляху проходження даних. Четвертим обмеженням є варіативність поведінки різних термінальних емуляторів у частині підтримки розширених команд ANSI та truecolor, що змушує прикладний застосунок орієнтуватися на найбільш поширений спільний підмножину можливостей.

Узагальнюючи описані спостереження, можна стверджувати, що метод текстового виводу через ANSI escape-послідовності забезпечує надійну та переносиму реалізацію консольної візуалізації, яка працює у будь-якому сучасному термінальному емуляторі. Водночас його обмеженість символьною сіткою терміналу не дозволяє досягти такої ж якості зображення, як при прямому піксельному малюванні.

2.3 Висновки за розділом

Експериментальна реалізація прототипів двох механізмів графічного виводу у консольному середовищі, основу для прийняття архітектурного рішення щодо подальшої побудови програмної системи. Метод графічного виводу через інтерфейс Windows GDI підтвердив свою повну функціональну придатність для відтворення моніторингових візуалізацій з піксельною точністю у локальному середовищі класичної консолі Windows. Метод текстового виводу через ANSI escape-послідовності продемонстрував стабільну роботу у термінальних емуляторах Linux. Однак жоден з цих методів не виявився універсальним: кожен має чітко окреслену сферу застосування і власні обмеження, що унеможливлюють його використання поза цією сферою.

Для систематизації виявлених у процесі дослідження властивостей двох методів сформовано порівняльну таблицю 2.1, у якій представлено зіставлення за основними критеріями оцінювання.

Аналіз представлених у таблиці характеристик дозволяє зробити кілька важливих висновків щодо вибору механізму виводу. Метод Windows GDI забезпечує найвищу якість зображення серед розглянутих підходів та найкращу інтеграцію з нативним середовищем класичної консолі Windows. Однак він принципово не може використовуватись на платформі Linux і у віддалених сесіях, що становлять одну з найважливіших цільових сфер застосування системи. Метод ANSI escape-послідовностей, навпаки, забезпечує переносимість і надійну роботу у будь-якому сучасному термінальному середовищі, у тому числі через мережу, але не дозволяє досягти такої ж якості зображення, як піксельне малювання.

Таблиця 2.1 – Порівняння методів графічного виводу

Критерій	Windows GDI	ANSI escape-послідовності
1	2	3
Цільова платформа	Microsoft Windows	Linux, macOS, BSD, Windows
Тип виводу	Піксельна графіка	Символьна псевдографіка
Якість зображення	Висока, обмежена розміром вікна у пікселях	Дискретна, обмежена розміром комірки терміналу
Згладжування шрифтів	Підтримується через ClearType	Залежить від налаштувань емулятора
Кольорова модель	RGB 24-bit	RGB 24-bit (truecolor)
Робота у віддаленій сесії SSH	Неможлива	Повна підтримка
Сумісність з Windows Terminal	Обмежена	Повна
Залежність від кодування	Відсутня	Потребує UTF-8
Залежність від системного API	Win32 API	Стандартні потоки введення-виведення

Таким чином, жоден із досліджених методів окремо не задовольняє повного переліку вимог, сформульованих до програмної системи у підрозділі 1.6. Для одночасного забезпечення високої якості зображення у локальному середовищі Windows та переносимості на платформу Linux із підтримкою віддаленої роботи необхідне поєднання обох методів у межах єдиного програмного продукту. Такий підхід дозволяє використовувати на кожній платформі найбільш придатний для неї механізм виводу, не накладаючи на користувача системи додаткових обмежень.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Вибір технологічного стеку

Вибір технологій реалізації системи визначається вимогами кросплатформності, продуктивності та мінімізації зовнішніх залежностей,

Мова програмування C++. Мова забезпечує прямий доступ до системних API обох цільових платформ, що необхідно для реалізації Windows GDI бекенду через Win32 API та ANSI бекенду через системні виклики Linux. C++ підтримує об'єктно-орієнтоване програмування з механізмом поліморфізму, який є основою архітектурного рішення з абстрактним інтерфейсом IBackend. Використовується стандарт C++17 як достатньо сучасний для застосування контейнерів стандартної бібліотеки та водночас підтримуваний усіма цільовими компіляторами без додаткових налаштувань.

Компілятор GCC обрано як основний для обох платформ. На Linux він є стандартним системним компілятором, доступним за замовчуванням у більшості дистрибутивів. На Windows використовується через дистрибутив MinGW-w64, що надає повний доступ до Win32 API, необхідного для роботи GDI-бекенду. Використання одного й того самого компілятора для обох платформ забезпечує однакову поведінку коду та усуває розбіжності, пов'язані з особливостями різних компіляторів.

Система збирання CMake є кросплатформним генератором проєктних файлів і дозволяє описати процес збирання один раз для обох платформ. Конфігураційний файл CMakeLists.txt визначає вихідні файли, цільові платформи, параметри компіляції та умовну компіляцію відповідного бекенду залежно від операційної системи.

Інструмент збирання Ninja використовується як низькорівнева система збирання, що працює у парі з CMake. На відміну від класичних утиліт make, Ninja орієнтована на максимальну швидкість виконання

збирання за рахунок мінімалістичного формату файлів збирання та паралельного виконання.

IDE Visual Studio Code. VS Code обрано як кросплатформне середовище розробки, що однаково працює на Windows та Linux. Воно інтегрується з CMake, GCC через стандартні розширення, надає засоби налагодження для обох платформ та зручне керування проєктом. Використання одного середовища на обох платформах спрощує процес розробки і тестування системи у різних середовищах.

Описаний технологічний стек забезпечує єдиний процес розробки і збирання системи на двох цільових платформах, мінімальну кількість зовнішніх залежностей та повну переносимість вихідного коду.

3.2 Організація збірки та керування залежностями

Для реалізації віддаленої роботи системи через захищений канал використано бібліотеку libssh2, що реалізує клієнтську частину протоколу SSH-2. Бібліотека libssh2 потребує криптографічного бекенду, оскільки не виконує криптографічні операції самостійно. Для цієї функції обрано бібліотеку mbedTLS, що відзначається малим розміром та підтримкою статичного лінкування на обох цільових платформах.

Обидві бібліотеки лінкуються статично. Це забезпечує, що скомпільована бібліотека системи містить увесь необхідний код і не потребує наявності зовнішніх динамічних бібліотек у системі кінцевого користувача. Прикладний застосунок, що використовує розроблювану бібліотеку, отримує єдиний самодостатній модуль без додаткових зовнішніх залежностей.

Несумісність скомпільованих бінарних бібліотек між операційними системами є принциповою проблемою при розробці кросплатформної системи. Об'єктний код, зібраний під Windows, не може бути використаний під Linux і навпаки, оскільки формати об'єктних файлів цих платформ різняться. Для розв'язання цієї проблеми зовнішні залежності постачаються у

складі проєкту у вигляді вихідних кодів, а їх компіляція виконується автоматично під цільову платформу засобами системи збирання CMake. Конфігураційний сценарій визначає поточну операційну систему, перевіряє наявність раніше зібраних залежностей за маркером і за його відсутності виконує послідовну збірку mbedTLS та libssh2 з розміщенням отриманих статичних бібліотек у каталозі залежностей проєкту. Перенесення системи на іншу платформу зводиться до копіювання каталогу проєкту. Повторні збірки на тій самій платформі використовують уже скомпільовані залежності.

Файлову структуру проєкту наведено на рисунку 3.1.

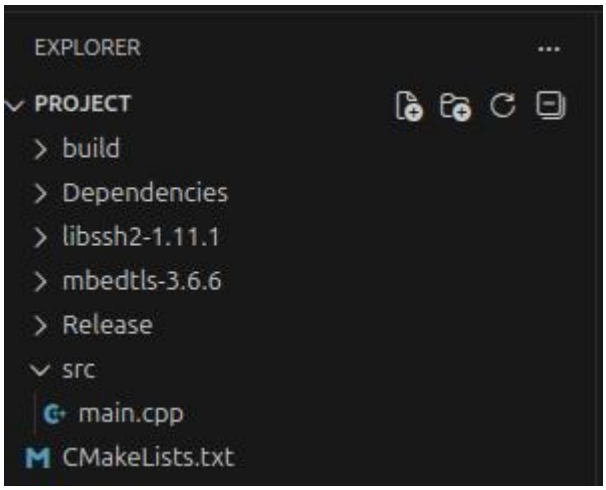


Рисунок 3.1 – Файлова структура проєкту

Каталог libssh2-1.11.1 містить вихідні коди бібліотеки libssh2, каталог mbedtls-3.6.6 містить вихідний коди криптографічної бібліотеки mbedTLS. Каталог Dependencies призначений для розміщення скомпільованих під поточну платформу статичних бібліотек та заголовних файлів. Каталог src містить вихідний код системи, каталог Release – результат компіляції. Файл CMakeLists.txt описує процес збирання, включно з автоматичною компіляцією залежностей.

Для перевірки коректності організації збірки та лінкування залежностей написано тестову програму, що виконує ініціалізацію бібліотеки libssh2, створення сесії SSH з використанням криптографічного бекенду

mbedTLS та звільнення ресурсів. Результат компіляції та виконання у середовищі Microsoft Windows наведено на рисунку 3.2.

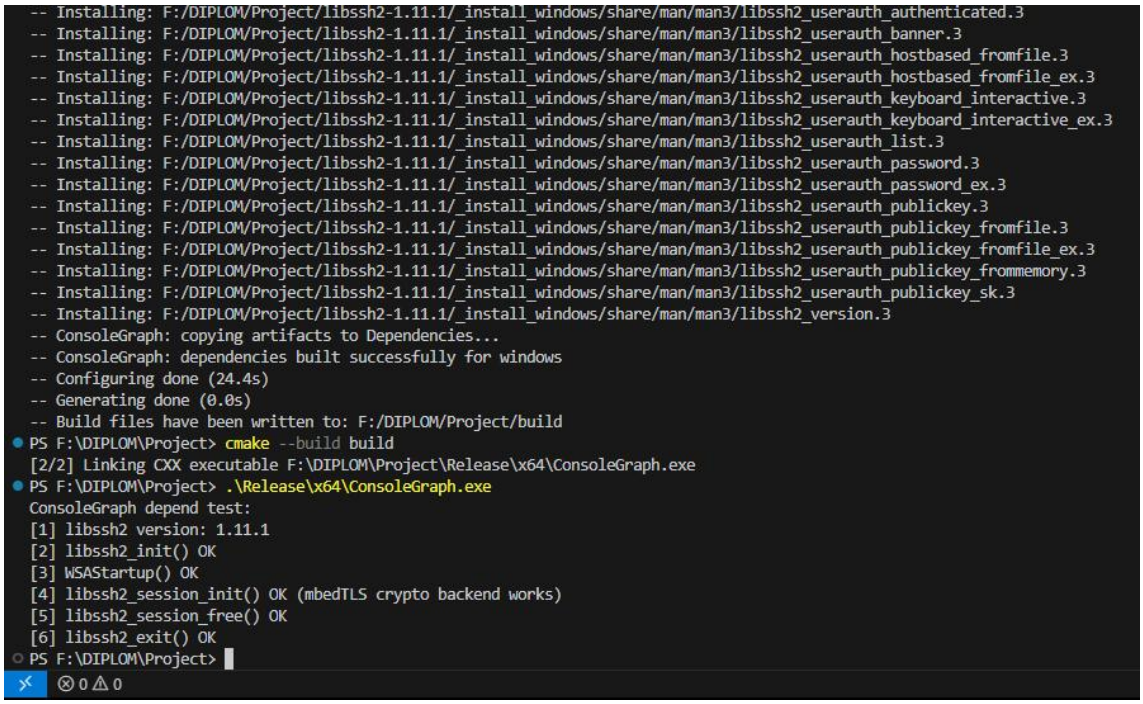


Рисунок 3.2 – Результат компіляції та виконання у середовищі Windows

Аналогічну перевірку виконано у середовищі Linux. Після копіювання каталогу проєкту на цільову систему та запуску збирання залежності автоматично перекомпільовано під платформу Linux, після чого зібрано програму. Результат компіляції та виконання у середовищі Linux наведено на рисунку 3.3.

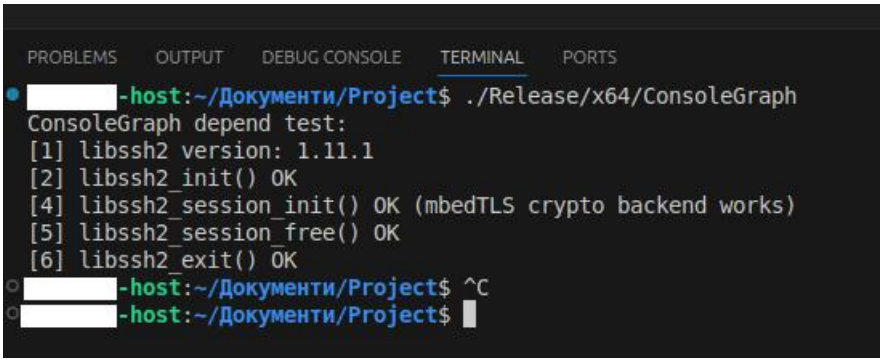


Рисунок 3.3 – Результат компіляції та виконання у середовищі Linux

Успішне виконання програми на обох цільових платформах підтверджує коректність організації збірки та керування залежностями. Технологічний стек і система збирання підготовлені для реалізації програмних компонентів системи.

3.3 Архітектура системи

На рисунку 3.4 представлена загальна схема функціонування розроблюваного програмного забезпечення.

Система побудована за клієнт-серверною моделлю і включає дві основні сторони взаємодії. На віддаленому сервері VPS під управлінням Linux виконується прикладний застосунок користувача, який збирає та обробляє дані моніторингу і використовує бібліотеку ConsoleGraph для побудови візуалізації. На локальній машині під управлінням Windows працює утиліта-клієнт consolegraph.exe, що забезпечує перегляд результатів роботи віддаленого застосунку у власному консольному вікні. З'єднання між клієнтом і сервером встановлюється через шифрований канал SSH.

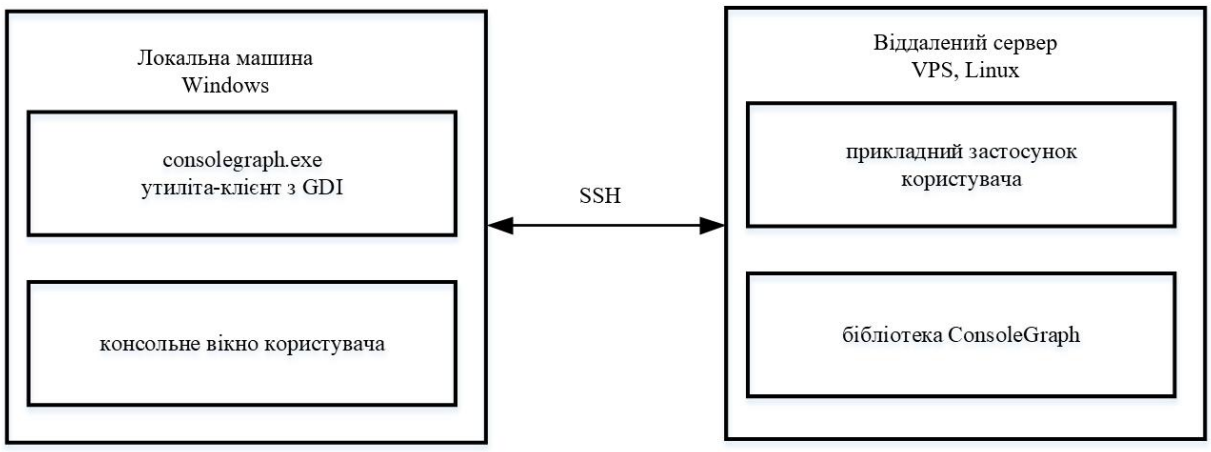


Рисунок 3.4 – Загальна структура системи ConsoleGraph

Окрім описаного режиму роботи з віддаленим сервером, бібліотека підтримує і локальний режим без використання мережі, у якому вивід виконується безпосередньо у консольне вікно Windows або термінал Linux

залежно від платформи запуску. Внутрішня будова системи представлена на рисунку 3.5.

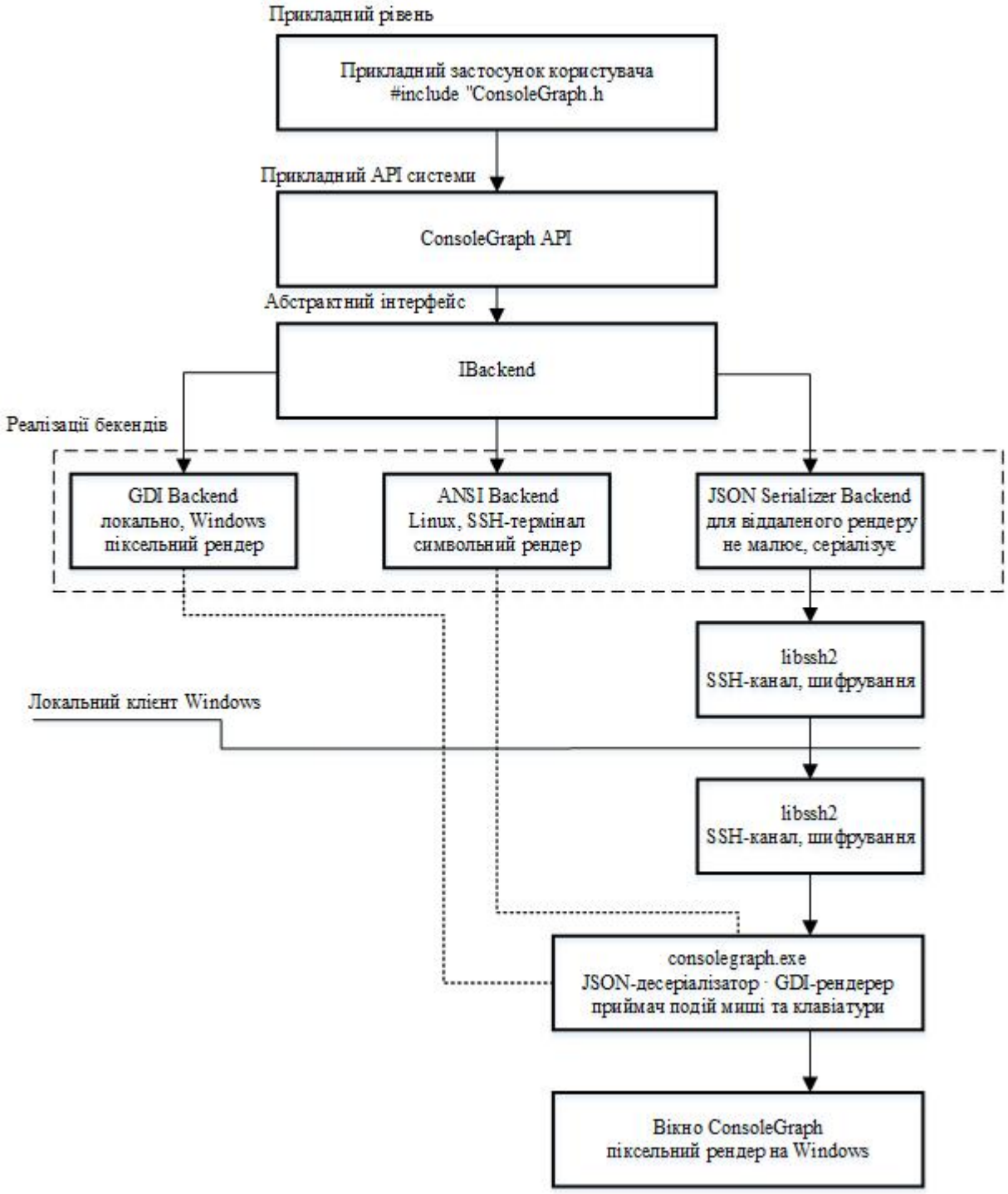


Рисунок 3.5 – Загальна архітектура системи ConsoleGraph

Архітектура поділяється на чотири логічні шари: прикладний рівень, шар прикладного API, шар абстракції бекендів та шар реалізацій бекендів.

До складу системи також входить транспортна підсистема та клієнтська утиліта віддаленого рендеру.

Прикладний застосунок користувача є зовнішнім по відношенню до системи компонентом, що використовує її для побудови візуалізації власних даних моніторингу. Розробник підключає заголовний файл бібліотеки, лінкується зі статичним об'єктним файлом і отримує доступ до її функціональності.

Прикладний API є шаром високорівневих функцій побудови типових елементів моніторингової візуалізації. Цей шар не залежить від платформи і не звертається безпосередньо до системних викликів операційної системи, а виконує усі графічні операції через виклик методів абстрактного інтерфейсу нижнього шару.

Абстрактний інтерфейс IBackend є центральним архітектурним елементом системи, що забезпечує її кросплатформність. Інтерфейс визначає мінімальний набір базових графічних операцій, через які виражаються усі високорівневі функції прикладного API. Конкретний спосіб виконання цих операцій залежить від обраної реалізації бекенду.

GDI-бекенд є реалізацією інтерфейсу IBackend, що використовується у режимі локального запуску на платформі Microsoft Windows. Цей компонент забезпечує піксельне малювання у консольному вікні засобами Win32 API.

ANSI-бекенд є реалізацією інтерфейсу IBackend для платформи Linux та для випадків роботи через звичайний термінальний клієнт. Цей компонент забезпечує символьний вивід у термінал засобами стандартного потоку виведення.

JSON Serializer Backend є третьою реалізацією інтерфейсу IBackend, що принципово відрізняється від двох попередніх відсутністю фактичного малювання. Цей компонент серіалізує кожен виклик методу інтерфейсу у текстове повідомлення формату JSON та передає його у транспортну

підсистему. Активується у випадку, коли застосунок виявляє підключення через утиліту ConsoleGraph.

Транспортна підсистема забезпечує шифрований канал передачі даних між сервером і клієнтом. Реалізована через бібліотеку libssh2, що статично лінкується у склад системи з обох сторін з'єднання, що відображено на рисунку 3.4 двома окремими блоками. Канал є двостороннім і використовується для передачі команд малювання від сервера до клієнта та подій введення у зворотному напрямку.

Утиліта consolegraph є самостійним виконуваним застосунком, що запускається на локальній машині для роботи з віддаленим сервером. Її функціональність полягає у встановленні SSH-з'єднання, прийомі JSON-повідомлень з каналу та їх відтворенні у власному вікні засобами вбудованого GDI-рендеру. Утиліта також обробляє події вікна та пересилає їх на сервер, забезпечуючи двосторонню взаємодію.

3.4 Алгоритм роботи системи

Виконання алгориму (рисунок 3.6) починається з блоку 1, у якому здійснюється ініціалізація бібліотеки та підготовка її внутрішніх структур до подальшої роботи. На цьому етапі бібліотека визначає параметри середовища виконання – змінні оточення, тип термінала, розміри вихідної області й готується до вибору активного бекенду.

У блоці 2 виконується перевірка наявності ознак запуску застосунку через утиліту consolegraph. Перевірка здійснюється через зчитування спеціальної змінної середовища, яка встановлюється утилітою при відкритті SSH-сесії з сервером. Якщо змінну виявлено, керування передається на блок 6. У протилежному випадку керування передається на блок 3, де визначається цільова платформа виконання.

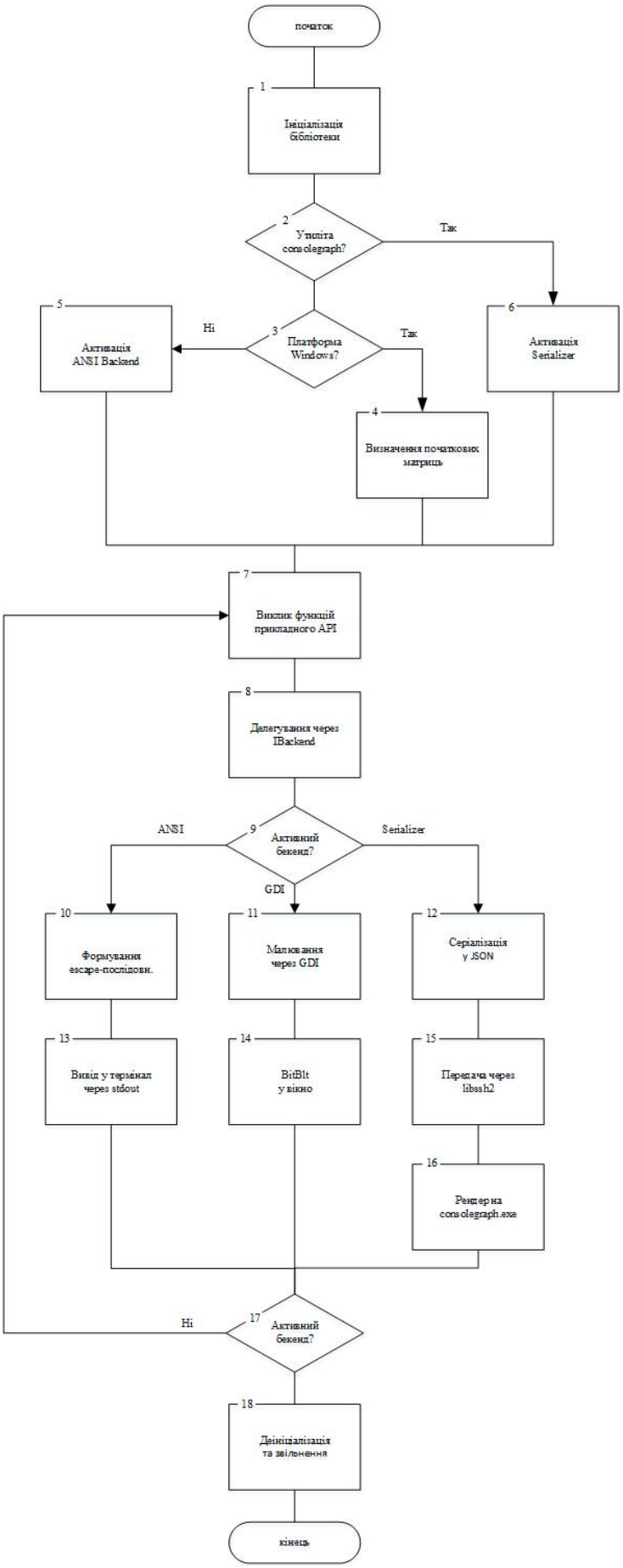


Рисунок 3.6 – Алгоритм функціювання системи

Блок 3 містить розгалуження за платформою. На етапі компіляції бібліотеки за допомогою умовної компіляції визначається, яка з двох платформ є цільовою для поточної збірки. У залежності від результату активується відповідний локальний бекенд – GDI Backend для Windows у блоці 4 або ANSI Backend для Linux у блоці 5. Блок 6, який є альтернативною гілкою з блоку 2, активує JSON Serializer Backend для режиму віддаленого рендеру.

Усі три гілки активації бекендів сходяться у точці входу циклу рендеру, що починається з блоку 7. У цьому блоці прикладний застосунок користувача викликає функції прикладного API для побудови чергового кадру візуалізації. Виклики API не залежать від того, який бекенд активовано, що забезпечує єдність прикладного коду на усіх платформах.

Блок 8 виконує делегування викликів API до активного бекенду через абстрактний інтерфейс IBackend. На цьому рівні відбувається ключовий момент кросплатформної архітектури: одні й ті самі виклики методів інтерфейсу виконуються різними способами залежно від того, яка реалізація IBackend є активною.

Блок 9 містить розгалуження за активним бекендом, що визначає подальшу гілку обробки кадру. Кожна з трьох гілок реалізує власний механізм виводу.

Гілка ANSI Backend починається з блоку 10, у якому виклики методів IBackend перетворюються на ANSI escape-послідовності з використанням блокових символів Unicode для зображення графіки. Сформовані послідовності у блоці 13 передаються у стандартний потік виведення stdout, звідки потрапляють у термінальний емулятор для відображення.

Гілка GDI Backend починається з блоку 11, у якому виклики методів IBackend виконуються засобами Win32 API у сумісному контексті пристрою у пам'яті. У блоці 14 готовий кадр одноразово переноситься у видиме консольне вікно операцією BitBlt, що забезпечує плавну анімацію без мерехтіння.

Гілка JSON Serializer Backend починається з блоку 12, у якому виклики методів IBackend упаковуються у текстові повідомлення формату JSON з ідентифікатором операції та її параметрами. У блоці 15 сформовані повідомлення передаються через шифрований канал libssh2 до клієнтської утиліти. У блоці 16 утиліта consolegraph.exe на стороні клієнта приймає повідомлення, виконує їх десеріалізацію та малює зображення засобами власного GDI-рендеру у локальному вікні Windows.

Усі три гілки сходяться у блоці 17, що містить перевірку умови завершення роботи. Якщо застосунок продовжує виконання, керування повертається до блоку 7 для побудови наступного кадру. У типовому сценарії моніторингу цикл рендеру повторюється з частотою тридцяти кадрів за секунду, забезпечуючи оновлення зображення у реальному часі.

При виявленні умови завершення керування передається у блок 18, де виконується деініціалізація бібліотеки та звільнення усіх системних ресурсів, виділених під час роботи. Для GDI-бекенду це включає звільнення створених GDI-об'єктів, для ANSI-бекенду – повернення терміналу у вихідний стан, для Serializer-бекенду – закриття SSH-каналу. Після завершення деініціалізації виконання алгоритму завершується.

3.5 Розробка програмних компонентів системи

Виходячи з архітектурного рішення, описаного у попередніх підрозділах, програмне забезпечення системи реалізоване у вигляді двох окремих компіляційних одиниць – статичної бібліотеки libConsoleGraph та виконуваної утиліти consolegraph. Бібліотека лінкується безпосередньо у прикладний код користувача і надає функції побудови візуалізації, тоді як утиліта є самостійним застосунком-клієнтом для віддаленого рендеру через мережу. Розробку обох компонентів виконано мовою C++ з застосуванням принципів об'єктно-орієнтованого програмування та поліморфізму.

На рисунку 3.7 представлено UML-діаграму класів бібліотеки libConsoleGraph, що відображає її внутрішню структуру, ієрархію класів та зв'язки між ними.

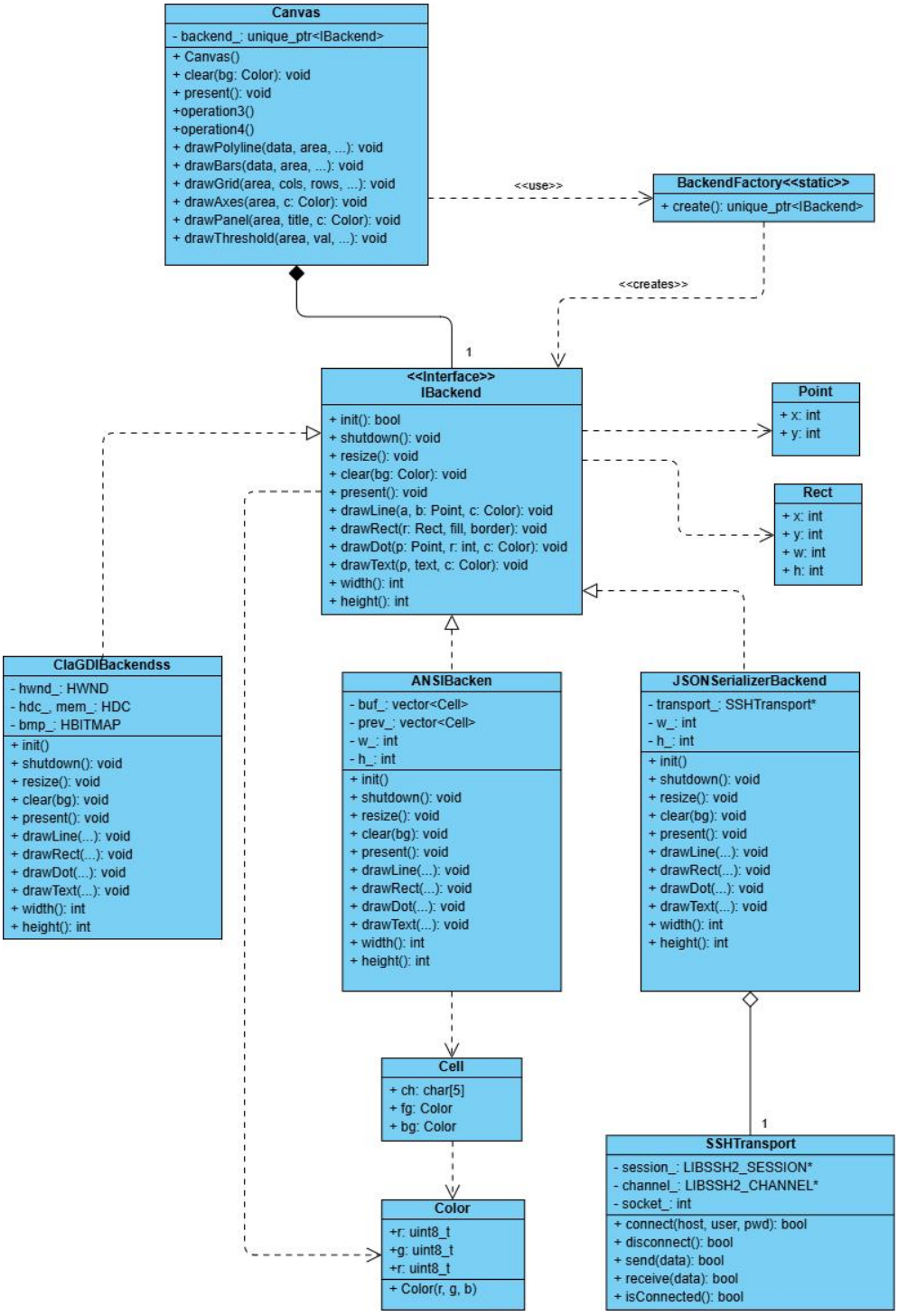


Рисунок 3.7 – UML-діаграма класів бібліотеки libConsoleGraph

Бібліотека містить кілька груп класів. До групи прикладного інтерфейсу належить клас Canvas, що виступає фасадом і надає прикладному коду набір високорівневих функцій побудови графіків та діаграм. До групи абстрактного шару входить інтерфейс IBackend, що визначає базовий контракт операцій рендеру, незалежний від платформи. Групу реалізацій утворюють три класи бекендів: GDIBackend для піксельного малювання у консольному вікні Windows, ANSIBackend для символьного виводу у термінал Linux та JSONSerializerBackend для серіалізації викликів у JSON-протокол при роботі з утилітою віддаленого рендеру. До групи транспорту належить клас SSHTransport, що інкапсулює роботу з шифрованим каналом libssh2. Окрему групу складають допоміжні структури Color, Point, Rect та Cell, що використовуються як параметри методів інтерфейсу та внутрішні структури даних бекендів. Вибір активної реалізації бекенду здійснюється класом BackendFactory, що містить статичний метод створення відповідного об'єкта залежно від середовища виконання.

Центральним класом прикладного шару системи є Canvas. Він приховує від прикладного коду подробиці роботи з конкретною платформою і надає єдиний набір функцій побудови типових елементів моніторингової візуалізації. Перелік методів класу Canvas з описом їх призначення наведено у таблиці 3.1.

Таблиця 3.1 – Методи класу Canvas

Метод	Параметри	Опис методу
1	2	3
Canvas()	-	Конструктор. Викликає BackendFactory::create() для отримання активного бекенду залежно від середовища виконання
clear()	bg: Color	Очищує поверхню рендеру кольором фону
drawPolyline()	data, area, yMin, yMax, c	Будує лінійний графік за переданим масивом числових значень у прямокутній області

Продовження таблиці 3.1

1	2	3
drawBars()	data, area, maxVal, c	Будує стовпчасту діаграму за переданим масивом значень
drawGrid()	area, cols, rows, c	Малює координатну сітку з заданою кількістю стовпців і рядків
drawAxes()	area, c	Малює координатні осі з підписами у прямокутній області
drawPanel()	area, title, c	Оформлює логічну панель з заголовком і рамкою
drawThreshold()	area, val, yMax, c, label	Малює горизонтальну порогову лінію з підписом значення
width()	-	Повертає поточну ширину поверхні рендеру у логічних одиницях
height()	-	Повертає поточну висоту поверхні рендеру у логічних одиницях
present()	-	Передає сформований кадр у вивід через активний бекенд

Ключовим елементом архітектури, що забезпечує її кросплатформність, є абстрактний інтерфейс IBackend. Цей інтерфейс визначає мінімальний набір базових графічних операцій, через які реалізуються усі високорівневі функції класу Canvas. Завдяки тому, що Canvas взаємодіє з бекендом виключно через цей інтерфейс, конкретна реалізація може бути змінена без модифікації прикладного коду. Перелік методів інтерфейсу IBackend наведено у таблиці 3.2.

Таблиця 3.2 – Методи інтерфейсу IBackend

Метод	Параметри	Опис методу
1	2	3
init()	-	Ініціалізація ресурсів бекенду перед початком роботи

Продовження таблиці 3.2

1	2	3
shutdown()	-	Звільнення ресурсів бекенду при завершенні роботи
resize()	-	Реакція на зміну розмірів вихідної області
resize()	-	Реакція на зміну розмірів вихідної області
clear()	bg: Color	Очищення поверхні кольором фону
present()	-	Передача сформованого кадру у вивід
drawLine()	a: Point, b: Point, c: Color	Малювання лінії між двома точками заданим кольором
drawRect()	area, c	Малювання прямокутника з заливкою і контуром
drawText()	area, title, c	Виведення текстового рядка у заданій позиції
width()	-	Повертає ширину вихідної області у логічних одиницях
height()	-	Повертає висоту вихідної області у логічних одиницях
drawDot()	p: Point, r: int, c: Color	Малювання точки заданого радіуса

Серед трьох реалізацій інтерфейсу IBackend особливий інтерес становить клас JSONSerializerBackend, що принципово відрізняється від двох інших. На відміну від GDIBackend та ANSIBackend, які виконують фактичне малювання у локальному середовищі, JSONSerializerBackend перетворює кожен виклик методу на текстове JSON-повідомлення і передає його у віддалену клієнтську утиліту через канал SSH. Цей клас агрегує об'єкт SSHTransport для забезпечення мережевої передачі. Перелік методів класу JSONSerializerBackend наведено у таблиці 3.3.

Таблиця 3.3 – Методи класу JSONSerializerBackend

Метод	Параметри	Опис методу
1	2	3
JSONSerializerBackend()	transport: SSHTransport*	Конструктор. Зберігає вказівник на об'єкт транспорту, через який будуть передаватися повідомлення
init()	-	Виконує початковий обмін повідомленнями для встановлення параметрів сесії з клієнтом
shutdown()	-	Надсилає повідомлення про завершення сесії і закриває канал
resize()	-	Надсилає клієнту інформацію про новий розмір вихідної області
clear()	bg: Color	Серіалізує операцію очищення у JSON та надсилає через транспорт
present()	-	Надсилає команду відображення сформованого кадру на стороні клієнта
drawLine()	a, b, c	Серіалізує параметри лінії у JSON-повідомлення формату {op: "line", x1, y1, x2, y2, r, g, b}
drawRect()	r, fill, border	Серіалізує параметри прямокутника у JSON-повідомлення
drawDot()	p, r, c	Серіалізує параметри точки у JSON-повідомлення
drawText()	p, text, c	Серіалізує текстове повідомлення з позицією та кольором у JSON
height()	-	Повертає висоту поверхні рендеру, отриману під час handshake з клієнтом

Продовження таблиці 3.3

1	2	3

height()	-	Повертає висоту поверхні рендеру, отриману під час handshake з клієнтом
sendCommand()	json: string	Приватний допоміжний метод, що надсилає сформоване JSON-повідомлення через об'єкт SSHTransport
width()	-	Повертає ширину поверхні рендеру, отриману під час handshake з клієнтом

На рисунку 3.8 представлено програмні компоненти утиліти UML-діаграму класів утиліти consolegraph, що працює як клієнт для віддаленого рендеру.

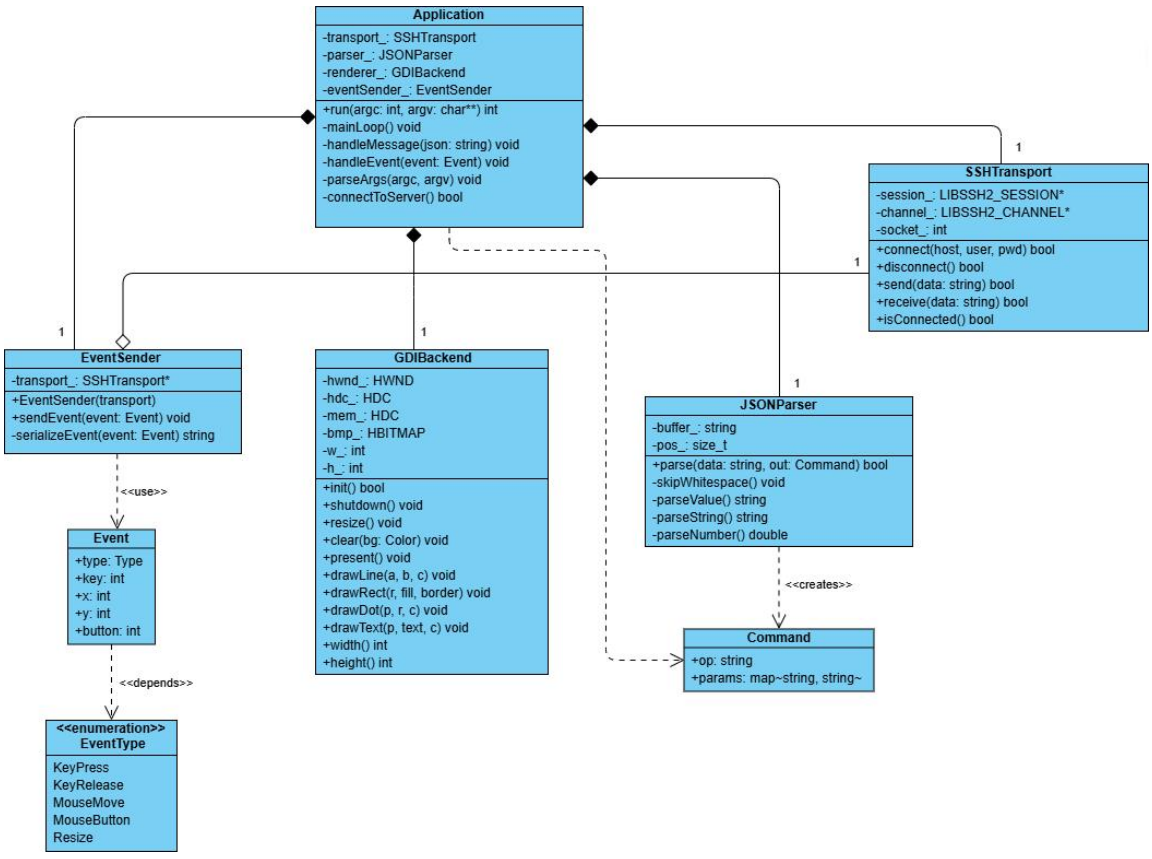


Рисунок 3.8 – UML-діаграма класів утиліти consolegraph

Утиліта містить кілька власних класів, специфічних для її функціональності, та використовує два класи з бібліотеки. Власні класи утиліти включають Application як головний координаційний клас, JSONParser

для десеріалізації повідомлень з мережі, EventSender для пересилання подій вікна на сервер, а також структури Event і Command для подання подій та команд відповідно. Імпортованими з бібліотеки є SSHTransport для встановлення SSH-з'єднання та GDIBackend для рендеру у локальному вікні Windows. Використання бібліотечних класів напряму, а не їх дублювання у коді утиліти, демонструє гнучкість архітектурного рішення з абстракцією платформи: один і той самий клас GDIBackend забезпечує піксельний рендер як при локальному запуску застосунку, так і у клієнтській утиліті при віддаленій роботі.

Центральним класом утиліти є Application, що координує роботу всіх її компонентів. Цей клас встановлює SSH-з'єднання з сервером, організовує цикл прийому повідомлень з каналу, делегує їх десеріалізацію парсеру та передає десеріалізовані команди у GDI-рендерер. Окрім цього, він обробляє події вікна та передає їх через EventSender назад на сервер. Перелік методів класу Application наведено у таблиці 3.4.

Таблиця 3.4 – Методи класу Application

Метод	Параметри	Опис методу
1	2	3
run()	argc: int, argv: char**	Точка входу утиліти. Виконує парсинг аргументів командного рядка, встановлення SSH-з'єднання та запуск головного циклу
mainLoop()	-	Головний цикл утиліти. Чекає на повідомлення з каналу, обробляє події вікна, оновлює рендер
handleMessage()	json: string	Обробляє отримане з каналу JSON-повідомлення: викликає парсер та виконує десеріалізовану команду через GDIBackend

Продовження таблиці 3.4

1	2	3

parseArgs()	argc, argv	Розбирає аргументи командного рядка: адресу сервера, шлях до застосунку, параметри підключення
handleEvent()	event: Even	Обробляє подію введення з боку користувача та передає її через EventSender на сервер

Класом, що відповідає за десеріалізацію повідомлень з мережі, є JSONParser. Він приймає рядок JSON-повідомлення, перевіряє його синтаксичну коректність та повертає структуру Command з ідентифікатором операції та її параметрами. Парсер реалізований без використання сторонніх бібліотек для забезпечення мінімізації залежностей утиліти. Перелік методів класу JSONParser наведено у таблиці 3.5.

Таблиця 3.5 – Методи класу JSONParser

Метод	Параметри	Опис методу
1	2	3
parse()	data: string, out: Command	Виконує синтаксичний розбір JSON-рядка та заповнює структуру Command. Повертає true при успішному парсингу
skipWhitespace()	-	Приватний метод. Пропускає пробільні символи у вхідному потоці
parseValue()	-	Приватний метод. Розбирає значення поля JSON: рядок, число, об'єкт або масив
parseString()	-	Приватний метод. Розбирає рядкове значення JSON з обробкою екранованих символів
parseNumber()	-	Приватний метод. Розбирає числове значення JSON

За пересилання подій вікна на сервер відповідає клас EventSender. Він приймає об'єкт події від класу Application, серіалізує його у JSON-повідомлення та надсилає через об'єкт SSHTransport. На сервері відповідні

події читаються прикладним застосунком через інтерфейс читання з SSHTransport. Перелік методів класу EventSender наведено у таблиці 3.6.

Таблиця 3.6 – Методи класу EventSender

Метод	Параметри	Опис методу
EventSender()	transport: SSHTransport*	Конструктор. Зберігає вказівник на об'єкт транспорту для подальшого надсилання повідомлень
sendEvent()	event: Event	Серіалізує подію у JSON та надсилає її через об'єкт SSHTransport на сервер
serializeEvent()	event: Event	Приватний метод. Перетворює структуру Event у текстове JSON-представлення з типом події та її параметрами

Описана структура програмних компонентів забезпечує чітке розділення відповідальностей між класами та можливість незалежної розробки і тестування кожного з них. Бібліотека і утиліта компілюються незалежно одна від одної, а їх взаємодія відбувається виключно через JSON-протокол поверх SSH-каналу.

4 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

4.1 Вибір технологічного стеку

Метою тестування є перевірка відповідності розробленої системи вимогам, сформульованим у підрозділі 1.6, а саме кросплатформності, наявності уніфікованого програмного інтерфейсу, самодостатності щодо зовнішніх залежностей та працездатності механізму віддаленого рендеру через захищений канал.

Тестування проводилося у середовищі, що складається з трьох обчислювальних вузлів. Першим вузлом є локальна машина під управлінням Microsoft Windows 11, на якій виконується утиліта-клієнт. Другим вузлом є віртуальна машина під управлінням Ubuntu, розгорнута засобами VMware Player на тій самій фізичній машині. Третім вузлом є віддалений сервер VPS під управлінням Ubuntu, доступний через мережу Інтернет. Така конфігурація дозволяє перевірити роботу системи як у локальному віртуалізованому середовищі, так і на реальному віддаленому сервері.

Для тестування використовується прикладний застосунок dashboard, що входить до складу проєкту і використовує бібліотеку ConsoleGraph. Застосунок будує моніторинговий дашборд з двох панелей: панелі Signal Monitor з лінійними графіками сигналів та панелі Resource Monitor з лінійним графіком, пороговими лініями та стовпчастою діаграмою. Вихідний код застосунку є однаковим для усіх платформ і режимів роботи, що дозволяє використовувати його для перевірки уніфікованості програмного інтерфейсу.

Перевірка системи виконувалася за такими сценаріями.

Перший сценарій передбачає складання системи на обох цільових платформах. Перевіряється коректність автоматичної компіляції зовнішніх залежностей під відповідну операційну систему та збирання бібліотеки, прикладного застосунку і утиліти.

Другий сценарій передбачає локальний запуск прикладного застосунку безпосередньо на платформі складання. На платформі Windows перевіряється робота GDI-бекенду, на платформі Linux – робота ANSI-бекенду.

Третій сценарій передбачає віддалений запуск прикладного застосунку на сервері з відображенням результату на клієнтській машині Windows. Цей сценарій перевіряється у двох варіантах підключення. У першому варіанті підключення виконується звичайним SSH-клієнтом без використання утиліти, при якому активується ANSI-бекенд і графіка відображається у вікні термінала засобами символьного виводу. У другому варіанті підключення виконується через утиліту consolegraph, при якому активується бекенд серіалізації, дані передаються у форматі JSON через канал SSH і відтворюються на клієнті засобами GDI з піксельною якістю. Обидва варіанти перевіряються спочатку з використанням локальної віртуальної машини Ubuntu, а потім з використанням віддаленого сервера VPS.

Окремо перевіряється самодостатність скомпільованого застосунку щодо зовнішніх залежностей. Зібраний на платформі Linux застосунок разом з бібліотекою переноситься на віддалений сервер без перенесення будь-яких додаткових бібліотек, після чого запускається на сервері. Успішний запуск підтверджує, що статичне лінкування забезпечує самодостатність застосунку і відсутність потреби у встановленні зовнішніх компонентів на цільовому сервері.

Передача подій введення у напрямку від клієнта до сервера окремо не тестувалася, оскільки реалізація каналу є симетричною і використовує той самий механізм SSHTransport, що й передача команд малювання у зворотному напрямку. Працездатність каналу в одному напрямку з високою імовірністю свідчить про його працездатність у зворотному напрямку через ідентичність програмної реалізації.

4.2 Складання системи на цільових платформах

Перед перевіркою функціональності системи виконано її складання на обох цільових платформах. Структуру каталогів проєкту наведено на рисунку 4.1.

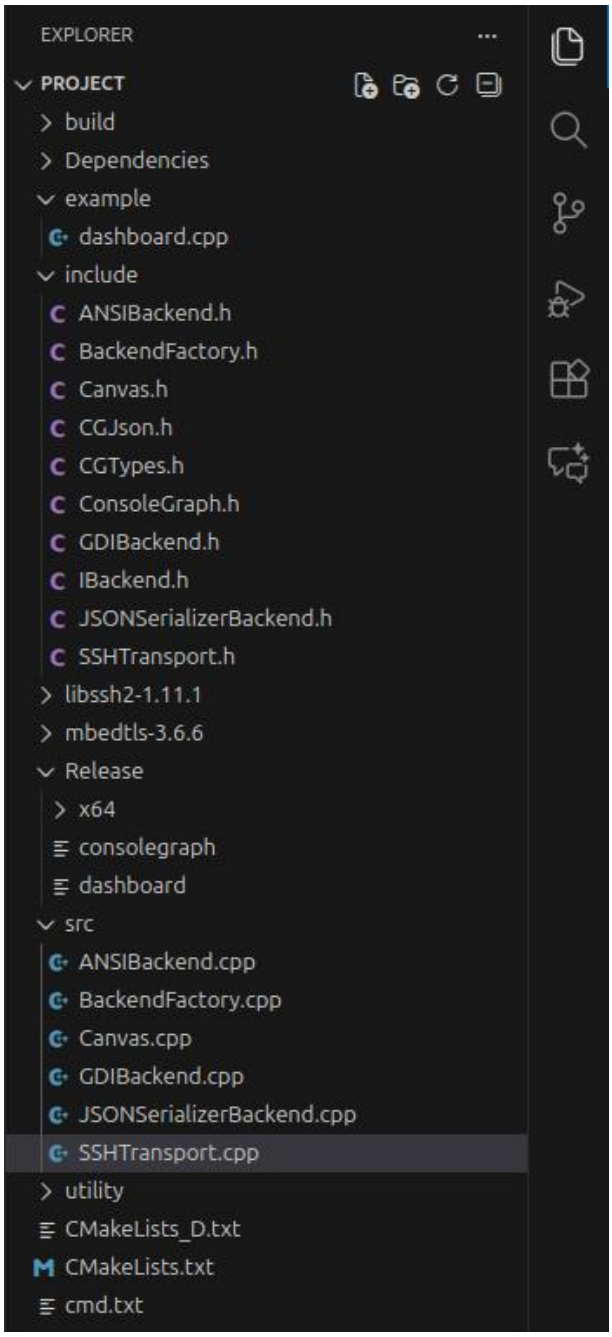


Рисунок 4.1 – Структура каталогів проєкту

Каталог include містить заголовні файли бібліотеки, каталог src – реалізації її класів, каталог example є прикладним застосунком dashboard, каталог utility містить вихідний код утиліти-клієнта. Файл CMakeLists.txt

описує процес складання усіх трьох цілей проєкту: бібліотеки, прикладного застосунку та утиліти.

Складання на платформі Windows виконано компілятором GCC з дистрибутиву MinGW-w64 за допомогою системи збирання CMake та інструмента Ninja. Результат складання наведено на рисунку 4.2.

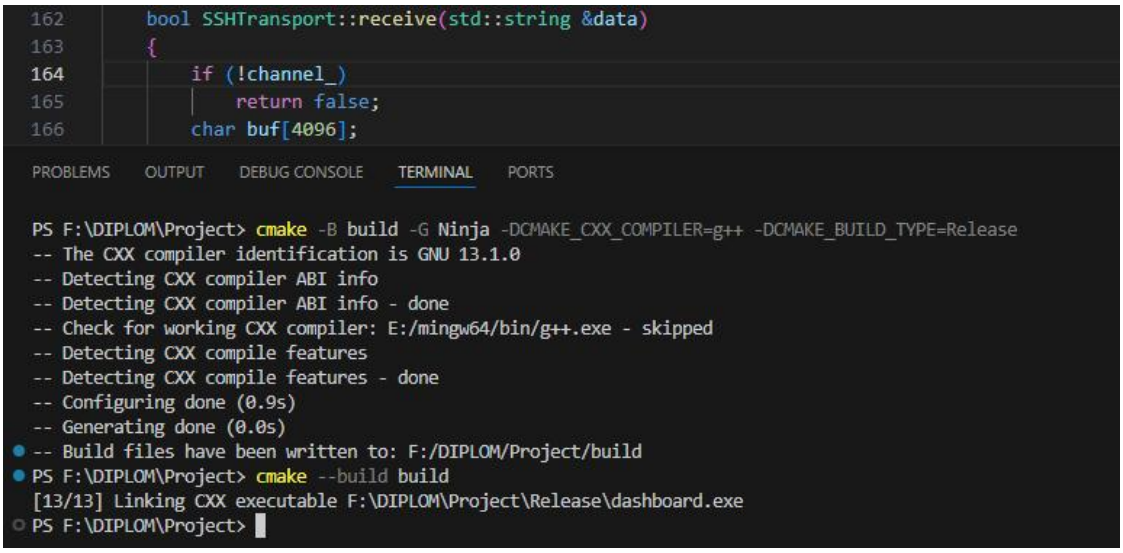


Рисунок 4.2 – Результат складання системи на платформі Windows

На етапі конфігурації CMake визначив компілятор та підготував файли збирання. На етапі компіляції послідовно зібрано бібліотеку, прикладний застосунок та утиліту. Повідомлення про успішне лінкування виконуваних файлів підтверджує коректне складання усіх компонентів системи під платформу Windows.

Аналогічне складання виконано на платформі Linux у середовищі віртуальної машини Ubuntu. Результат наведено на рисунку 4.3.

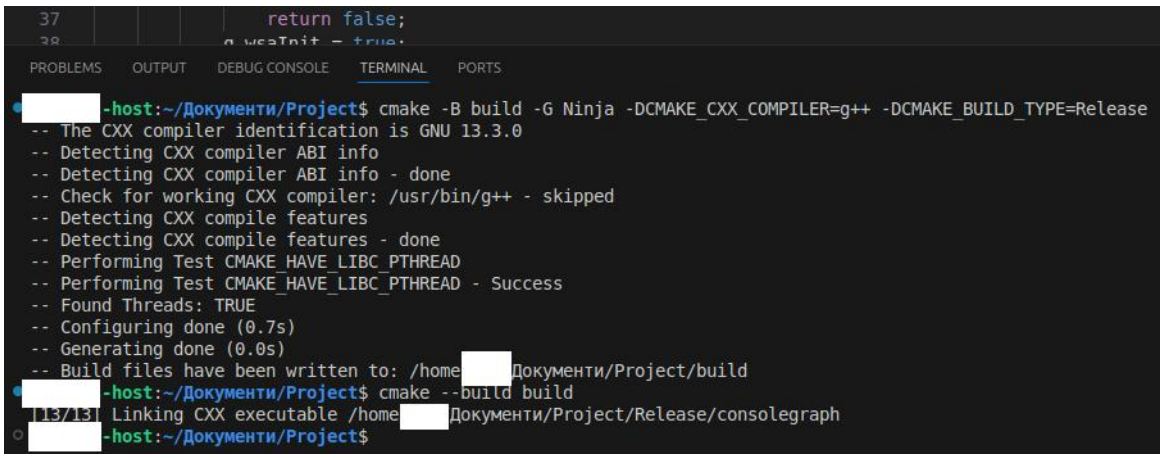


Рисунок 4.3 – Результат складання системи на платформі Linux

CMake визначив цільову платформу як Linux та підключив системну бібліотеку потоків, необхідну для роботи залежностей на цій платформі. Складання усіх трьох цілей завершено успішно. Відмінності у складанні між платформами зводяться до автоматичного вибору відповідного бекенду та системних бібліотек засобами CMake.

4.3 Перевірка локального режиму роботи

Локальний режим роботи передбачає запуск прикладного застосунку безпосередньо на тій платформі, де його було складено, без використання мережі та утиліти-клієнта. У цьому режимі бібліотека самостійно обирає бекенд відповідно до платформи виконання.

Запуск застосунку dashboard на платформі Windows активує GDI-бекенд. Результат роботи наведено на рисунку 4.4.

Рисунок 4.4 – Локальний запуск на платформі Windows (GDI-бекенд)

Застосунок відобразив дашборд у вікні консолі засобами піксельного малювання. Усі елементи відтворено з піксельною точністю, лінії згладжені, оновлення зображення відбувається без помітних затримок.

Запуск того самого застосунку на платформі Linux у віртуальній машині Ubuntu активує ANSI-бекенд. Результат роботи наведено на рисунку 4.5.

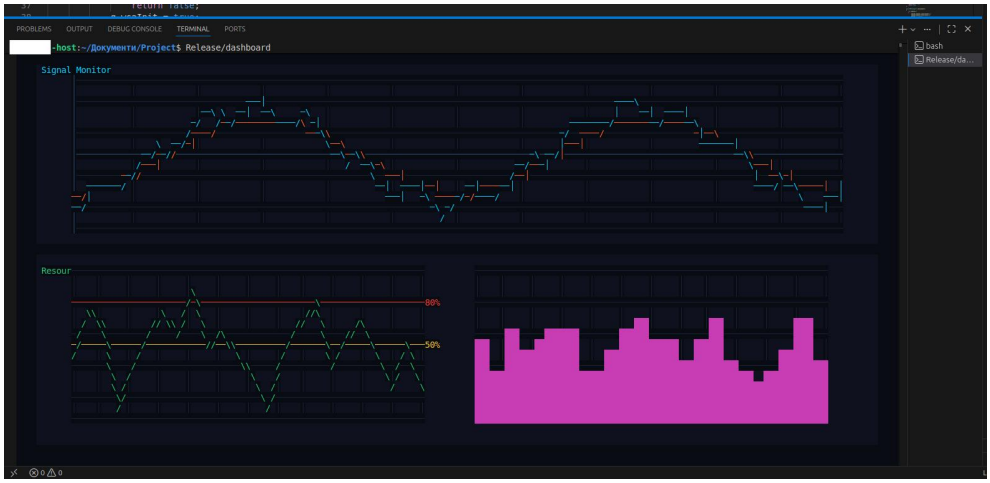


Рисунок 4.5 – Локальний запуск на платформі Linux (ANSI-бекенд)

Застосунок відобразив той самий дашборд засобами символьного виводу з використанням блокових символів Unicode. Структура зображення

відповідає варіанту на платформі Windows, відмінність полягає лише у способі формування зображення, що зумовлено дискретністю символьної сітки термінала.

Вихідний код застосунку при запуску на обох платформах є абсолютно однаковим, вибір механізму виводу здійснено системою автоматично.

4.4 Перевірка віддаленого режиму роботи

Віддалений режим роботи передбачає виконання прикладного застосунку на сервері під управлінням Linux з відображенням результату на клієнтській машині Windows через канал SSH. Перевірку виконано у двох варіантах підключення для двох типів серверів: локальної віртуальної машини Ubuntu та віддаленого сервера VPS.

Першим перевірено підключення до віртуальної машини Ubuntu звичайним SSH-клієнтом без використання утиліти. У цьому варіанті бібліотека на сервері визначає підключення як термінальне і активує ANSI-бекенд. Результат наведено на рисунку 4.6.

Рисунок 4.6 –Підключення до віртуальної машини звичайним SSH-клієнтом

Застосунок, запущений на віртуальній машині, відобразив дашборд у вікні термінала клієнта засобами символьного виводу. Дані передавалися через стандартний потік виведення SSH-сесії без використання спеціалізованої утиліти.

Другим перевірено підключення до тієї самої віртуальної машини через утиліту consolegraph. У цьому варіанті бібліотека визначає підключення через утиліту і активує бекенд серіалізації. Результат наведено на рисунку 4.7.

Рисунок 4.7 – Підключення до віртуальної машини через утиліту

Застосунок на віртуальній машині серіалізував виклики методів у JSON-повідомлення та передав їх через канал SSH. Утиліта на клієнті десеріалізувала повідомлення і відтворила зображення засобами GDI з піксельною якістю. Результат візуально відповідає локальному запуску на платформі Windows, незважаючи на те, що застосунок фактично виконувався на віддаленій віртуальній машині під управлінням Linux.

Аналогічну перевірку виконано з використанням віддаленого сервера VPS, доступного через мережу Інтернет. Перед перевіркою зібраний на платформі Linux застосунок разом з бібліотекою перенесено на сервер без перенесення будь-яких додаткових бібліотек чи компонентів рисунок 4.8.

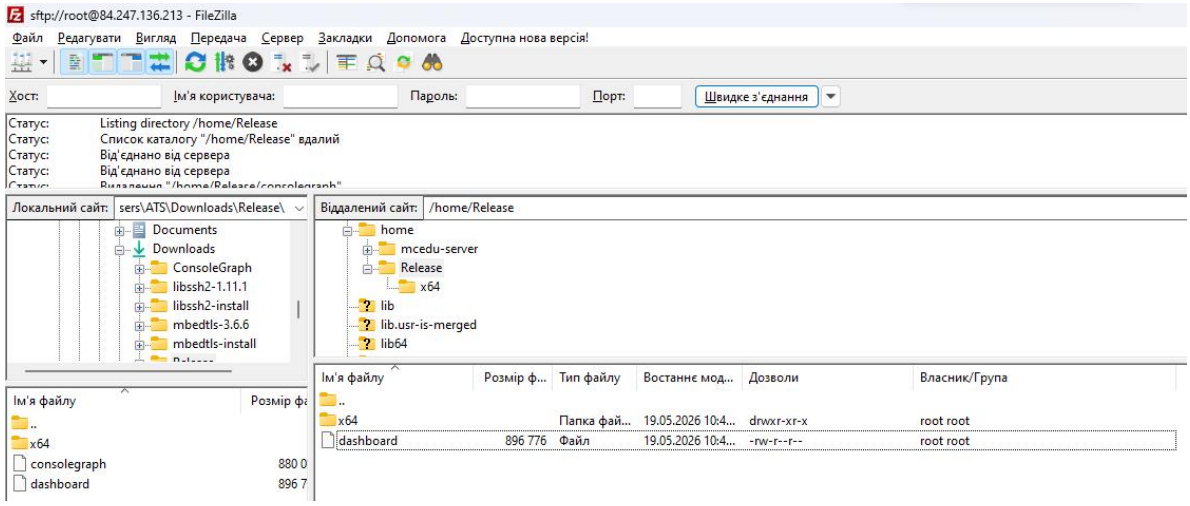


Рисунок 4.8 – Розташування застосунку на VPS

Успішний запуск застосунку на сервері підтвердив самодостатність системи: статичне лінкування забезпечує відсутність потреби у встановленні зовнішніх залежностей на цільовому сервері.

Підключення до сервера VPS звичайним SSH-клієнтом активує ANSI-бекенд. При підключенні до того самого сервера через утиліту consolegraph активувало бекенд серіалізації з відтворенням зображення засобами GDI на клієнті. Результат наведено на рисунку 4.9.

Рисунок 4.9 – Підключення до сервера VPS через утиліту

Результат роботи на віддаленому сервері VPS повністю відповідає результату з локальною віртуальною машиною як у термінальному, так і в утилітному варіанті підключення. Це підтверджує, що система коректно функціонує не лише у локальному віртуалізованому середовищі, а й на реальному віддаленому сервері, доступному через мережу Інтернет.

Виконані перевірки підтверджують працездатність усіх трьох бекендів системи та механізму віддаленого рендеру. Той самий прикладний застосунок, без жодних змін у вихідному коді, працює у локальному режимі на двох платформах, у термінальному режимі через звичайний SSH та у режимі піксельного рендеру через утиліту, що підтверджує виконання вимог щодо кросплатформності, уніфікованого програмного інтерфейсу та підтримки віддаленої роботи.

ВИСНОВКИ

У кваліфікаційній дипломній роботі бакалавра розв'язано задачу створення програмної кросплатформної системи для візуалізації даних моніторингу у консольних застосунках. Поставлену мету щодо зменшення часу та ресурсних витрат на впровадження візуалізації даних моніторингу шляхом створення кросплатформної системи з уніфікованим програмним інтерфейсом для платформ Windows та Linux досягнуто.

У результаті аналізу предметної області встановлено, що наявні програмні засоби консольної візуалізації не поєднують одночасно реалізацію мовою C++, кросплатформну роботу на платформах Windows та Linux і єдиний програмний інтерфейс для побудови графіків та діаграм. Переважна більшість аналогів реалізована інтерпретованими мовами і потребує наявності інтерпретатора у цільовій системі, а жоден з розглянутих засобів не використовує прямий доступ до графічної підсистеми Windows. Це обґрунтувало доцільність розробки власної системи.

Експериментально досліджено два методи графічного виводу у консольному середовищі. Метод на основі Windows GDI підтвердив придатність для піксельного відображення у локальному середовищі Windows, метод на основі ANSI escape-послідовностей – для символьного виводу у терміналах Linux, у тому числі через віддалені сесії SSH. Встановлено, що жоден з методів не є універсальним, що зумовило потребу у комбінованому архітектурному рішенні.

Спроєктовано архітектуру системи на основі абстрактного програмного інтерфейсу IBackend, що приховує від прикладного коду відмінності у механізмах виводу. Реалізовано три бекенди: GDI-бекенд для платформи Windows, ANSI-бекенд для платформи Linux та бекенд серіалізації для віддаленого рендеру через канал SSH. Розроблено уніфікований програмний інтерфейс у вигляді класу-фасаду з функціями

побудови лінійних графіків, стовпчастих діаграм, координатних сіток, осей та порогових ліній.

Реалізовано механізм віддаленого рендеру, за якого застосунок, що виконується на віддаленому сервері Linux, передає серіалізовані у форматі JSON команди малювання через шифрований канал SSH на клієнтську утиліту, де вони відтворюються засобами GDI з піксельною якістю. Зовнішні залежності системи постачаються у вигляді вихідних кодів і автоматично компілюються під цільову платформу засобами системи збирання, а статичне лінкування забезпечує самодостатність системи без потреби у встановленні зовнішніх компонентів.

Тестування системи на двох цільових платформах підтвердило виконання усіх поставлених вимог. Один і той самий код прикладного застосунку без жодних змін дає функціонально еквівалентний результат у локальному режимі на Windows та Linux, у термінальному режимі через звичайний SSH і у режимі піксельного рендеру через утиліту. Працездатність системи підтверджено як на локальній віртуальній машині, так і на реальному віддаленому сервері VPS.

Практичне значення отриманих результатів полягає у зменшенні часу впровадження консольної візуалізації за рахунок єдиного програмного інтерфейсу для двох платформ, що усуває потребу у розробці окремих рішень під кожну операційну систему. Відсутність зовнішніх графічних залежностей знижує вимоги до програмного середовища і дозволяє застосовувати систему на серверах без графічної підсистеми та у віддалених термінальних сесіях.

Розроблена система може бути використана у задачах системного адміністрування, моніторингу серверних середовищ, аналізу даних у віддалених обчислювальних системах та розробці консольних інструментів з вбудованою візуалізацією. Подальший розвиток системи можливий за рахунок додавання нових бекендів виводу, зокрема для платформи macOS,

без зміни існуючої функціональності, що забезпечується обраним архітектурним рішенням.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Teleprinter (date of access: 19.05.2026).

2 A brief history of terminal emulators. *Charm.* URL: https://charm.land/blog/intro-to-terminals/ (date of access: 19.05.2026).

3 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/VT100 (date of access: 19.05.2026).

4 Windows Command-Line: The Evolution of the Windows Command-Line. Windows Command Line. URL: https://devblogs.microsoft.com/commandline/windows-command-line-the-evolution-of-the-windows-command-line/ (date of access: 19.05.2026).

5 The evolution of the command line interface (CLI). *Contentstack.* URL: https://www.contentstack.com/blog/tech-talk/the-evolution-of-command-line-interface-cli-a-historical-insight (date of access: 19.05.2026).

6 Introduction to CLI Tools and Importance in DevOps. *KodeKloud*. URL: https://notes.kodekloud.com/docs/Rust-Programming/Building-Command-Line-Tools/Introduction-to-CLI-Tools-and-Importance-in-DevOps/page (date of access: 19.05.2026).

7 Jr S. W. E. Linux command line: a complete introduction. No Starch Press, Incorporated, 2012. 480 p.

8 Sabbir Saleh M. A systematic literature review on continuous integration and deployment (CI/CD) for secure cloud computing. *arXiv.org e-Print archive*. URL: https://arxiv.org/pdf/2506.08055 (date of access: 19.05.2026).

9 Delony D. From teletype to terminal window: the 3 eras of unix terminals. *How-To Geek.* URL: https://www.howtogeek.com/the-three-eras-of-unix-terminals/ (date of access: 19.05.2026).

10 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/ASCII_art (date of access: 19.05.2026).

11 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.*

URL: https://en.wikipedia.org/wiki/Code_page_437 (date of access: 19.05.2026).

12 GitHub - asciimoo/drawille: pixel graphics in terminal with unicode braille characters. *GitHub.* URL: https://github.com/asciimoo/drawille (date of access: 19.05.2026).

13 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/ANSI_escape_code (date of access: 19.05.2026).

14 URL: https://croakingkero.com/tutorials/drawing_pixels_win32_gdi/ (дата звернення: 19.05.2026).

15 Windows GDI - Win32 apps. Microsoft Learn: *Build with answers in reach*. URL: https://learn.microsoft.com/en-us/windows/win32/api/_gdi/ (date of access: 19.05.2026).

16 Contributors to Wikimedia projects. *Wikipedia, the free encyclopedia.* URL: https://en.wikipedia.org/wiki/Ncurses (date of access: 19.05.2026).

17 GitHub - mirror/ncurses: ncurses Git mirror. *GitHub*. URL: https://github.com/mirror/ncurses (date of access: 19.05.2026).

18 A gentle introduction to ncurses for the terminally impatient. *Hackaday.* URL: https://hackaday.com/2025/06/17/a-gentle-introduction-to-ncurses-for-the-terminally-impatient/ (date of access: 19.05.2026).

19 GitHub - arthursonzogni/ftxui: :computer: c++ functional terminal user interface. :heart:. *GitHub*. URL: https://github.com/ArthurSonzogni/FTXUI (date of access: 19.05.2026).

20 Notcurses is still alive for ramping up "terminal bling" with complex tuis - phoronix. Linux Hardware Reviews & Performance Benchmarks, *Open-Source News - Phoronix*. URL: https://www.phoronix.com/news/Notcurses-3.0.10-Release (date of access: 19.05.2026).

21 GitHub - nsf/termbox: library for writing text-based user interfaces. *GitHub*. URL: https://github.com/nsf/termbox (date of access: 19.05.2026).

22 Archived Yu-Jie Lin. Termbox: library for writing text-based user interfaces for C and Python, 2013. *YouTube.* URL:

https://www.youtube.com/watch?v=AaXbS9tz63E (date of access: 19.05.2026).

23 Naveh Y. Blessed-contrib. URL: https://www.npmjs.com/package/@blessed/blessed-contrib (date of access: 19.05.2026).

24 GitHub - Textualize/textual-plotext: A Textual widget wrapper library for Plotext. *GitHub.* URL: https://github.com/Textualize/textual-plotext (date of access: 19.05.2026).

25 Solomon D., Yosifovich P., Ionescu A. Windows internals, part 1: system architecture, processes, threads, memory management, and more. Microsoft Press, 2017. 800 p.

26 Petzold C. Programming windows. 5th ed. Microsoft Press, 2002. 1122 p. URL: https://www.cl72.org/100winProg/Charles%20Petzold%20-%20Programming%20Windows%20-%205th%20Ed.pdf (date of access: 29.05.2026).

27 Windows graphics programming: Win32 GDI and DirectDraw. Upper Saddle River, NJ : Prentice Hall, 2001. 1234 p.

28 Rubini A., Corbet J. Linux device drivers, 3rd edition. 3rd ed. Sebastopol, CA : O'Reilly Media, Inc., 2005. 636 p.

29 Cesati M., Bovet D. P. Understanding the linux kernel. 3rd ed. Sebastopol, CA : O'Reilly Media, Inc., 2005. 942 p.

30 The Linux command line. San Francisco : No Starch Press, 2011. 480 p.

ДОДАТОК А

Текст програми

А.1 Текст файлу main.cpp

```
#define NOMINMAX
#include <boost/asio.hpp>
#include <boost/beast.hpp>
#include <boost/filesystem.hpp>

#include <windows.h>
#include <stdio.h>
#include <math.h>
#include <vector>
#include <string>
#include <algorithm>

#include <iostream>

extern "C" WINBASEAPI HWND WINAPI GetConsoleWindow();

#define PI 3.14
struct Color { int r, g, b; };

constexpr Color COL_BG     = {10,  12,  20};
constexpr Color COL_GRID   = {25,  38,  58};
constexpr Color COL_AXIS   = {55,  80, 120};
constexpr Color COL_SIN    = {0,  210, 255};
constexpr Color COL_COS    = {255, 100,  50};
constexpr Color COL_CPU    = {40,  200, 120};
constexpr Color COL_MEM    = {200,  60, 180};
constexpr Color COL_TEXT   = {180, 200, 230};
constexpr Color COL_TITLE  = {255, 255, 255};
constexpr Color COL_WARN   = {255,  60,  60};
constexpr Color COL_CAUTION= {255, 200,  50};
constexpr Color COL_ACCENT = {255, 220,   0};

COLORREF rgb(Color c) { return RGB(c.r, c.g, c.b); }

struct GDICtx {
    HWND    hwnd;
    HDC     hdc;
    HDC     mem;
    HBITMAP bmp;
    int     W, H;

    bool init() {
```

```
    hwnd = GetConsoleWindow();
    if (!hwnd) return false;

    RECT rc;
    GetClientRect(hwnd, &rc);
    W = rc.right  - rc.left;
    H = rc.bottom - rc.top;

    hdc = GetDC(hwnd);
    mem = CreateCompatibleDC(hdc);
    bmp = CreateCompatibleBitmap(hdc, W, H);
    SelectObject(mem, bmp);
    return true;
}

void resize() {
    RECT rc;
    GetClientRect(hwnd, &rc);
    int nw = rc.right - rc.left;
    int nh = rc.bottom - rc.top;
    if (nw == W && nh == H) return;
    W = nw; H = nh;
    DeleteObject(bmp);
    DeleteDC(mem);
    mem = CreateCompatibleDC(hdc);
    bmp = CreateCompatibleBitmap(hdc, W, H);
    SelectObject(mem, bmp);
}

void clear() {
    HBRUSH br = CreateSolidBrush(rgb(COL_BG));
    RECT r = {0, 0, W, H};
    FillRect(mem, &r, br);
    DeleteObject(br);
}

void present() {
    BitBlt(hdc, 0, 0, W, H, mem, 0, 0, SRCCOPY);
}

void free() {
    DeleteObject(bmp);
    DeleteDC(mem);
    ReleaseDC(hwnd, hdc);
}
```

```
};

void gdiLine(GDICtx& g, int x0, int y0, int x1, int y1,
        Color c, int w = 1, int style = PS_SOLID) {
   HPEN pen = CreatePen(style, w, rgb(c));
   HGDIOBJ o = SelectObject(g.mem, pen);
   MoveToEx(g.mem, x0, y0, NULL);
   LineTo  (g.mem, x1, y1);
   SelectObject(g.mem, o);
   DeleteObject(pen);
}

void gdiRect(GDICtx& g, int x, int y, int w, int h,
        Color fill, Color border, int bw = 1) {
   HBRUSH br  = CreateSolidBrush(rgb(fill));
   HPEN   pen = CreatePen(PS_SOLID, bw, rgb(border));
   HGDIOBJ ob = SelectObject(g.mem, br);
   HGDIOBJ op = SelectObject(g.mem, pen);
   Rectangle(g.mem, x, y, x + w, y + h);
   SelectObject(g.mem, ob);
   SelectObject(g.mem, op);
   DeleteObject(br);
   DeleteObject(pen);
}

void gdiText(GDICtx& g, int x, int y, const char* s,
        Color c, int size = 13, bool bold = false) {
   SetBkMode   (g.mem, TRANSPARENT);
   SetTextColor(g.mem, rgb(c));
   HFONT f = CreateFontA(size, 0, 0, 0,
                bold ? FW_BOLD : FW_NORMAL,
                0, 0, 0, DEFAULT_CHARSET,
                OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                CLEARTYPE_QUALITY,
                DEFAULT_PITCH | FF_DONTCARE, "Consolas");
   HGDIOBJ o = SelectObject(g.mem, f);
   TextOutA(g.mem, x, y, s, (int)strlen(s));
   SelectObject(g.mem, o);
   DeleteObject(f);
}

void gdiDot(GDICtx& g, int x, int y, int r, Color c) {
   HPEN   pen = CreatePen(PS_SOLID, 1, rgb(c));
   HBRUSH br  = CreateSolidBrush(rgb(c));
   HGDIOBJ op = SelectObject(g.mem, pen);
```

```
    HGDIOBJ ob = SelectObject(g.mem, br);
    Ellipse(g.mem, x-r, y-r, x+r, y+r);
    SelectObject(g.mem, op);
    SelectObject(g.mem, ob);
    DeleteObject(pen);
    DeleteObject(br);
}


void drawGrid(GDICtx& g, int ox, int oy, int ow, int oh,
              int cols = 10, int rows = 8) {
    for (int i = 0; i <= cols; i++)
        gdiLine(g, ox + i*ow/cols, oy, ox + i*ow/cols, oy+oh, COL_GRID);
    for (int j = 0; j <= rows; j++)
        gdiLine(g, ox, oy + j*oh/rows, ox+ow, oy + j*oh/rows, COL_GRID);
}

void drawAxes(GDICtx& g, int ox, int oy, int ow, int oh) {
    int mid = oy + oh/2;
    gdiLine(g, ox, mid, ox+ow, mid, COL_AXIS, 2);
    gdiLine(g, ox, oy,  ox,    oy+oh, COL_AXIS, 2);

    gdiLine(g, ox+ow-8, mid-4, ox+ow, mid, COL_AXIS, 2);
    gdiLine(g, ox+ow-8, mid+4, ox+ow, mid, COL_AXIS, 2);
}


void drawPolyline(GDICtx& g,
                  const std::vector<double>& data,
                  int ox, int oy, int ow, int oh,
                  double yMin, double yMax,
                  Color c, int lw = 2) {
    int n = (int)data.size();
    if (n < 2) return;

    auto mapX = [&](int i) { return ox + (int)((double)i / (n-1) * ow); };
    auto mapY = [&](double v) {
        double norm = (v - yMin) / (yMax - yMin);
        norm = std::max(0.0, std::min(1.0, norm));
        return oy + oh - (int)(norm * oh);
    };


    for (int i = 0; i < n-1; i++) {
        int x0 = mapX(i),   y0 = mapY(data[i]);
```

```
        int x1 = mapX(i+1), y1 = mapY(data[i+1]);
        int base = oy + oh/2;

        POINT pts[4] = {{x0,y0},{x1,y1},{x1,base},{x0,base}};
        HBRUSH br = CreateSolidBrush(RGB(c.r/6, c.g/6, c.b/6));
        HPEN   np = CreatePen(PS_NULL, 0, 0);
        HGDIOBJ ob = SelectObject(g.mem, br);
        HGDIOBJ op = SelectObject(g.mem, np);
        Polygon(g.mem, pts, 4);
        SelectObject(g.mem, ob);
        SelectObject(g.mem, op);
        DeleteObject(br);
        DeleteObject(np);
    }


    HPEN pen = CreatePen(PS_SOLID, lw, rgb(c));
    HGDIOBJ op = SelectObject(g.mem, pen);
    MoveToEx(g.mem, mapX(0), mapY(data[0]), NULL);
    for (int i = 1; i < n; i++)
        LineTo(g.mem, mapX(i), mapY(data[i]));
    SelectObject(g.mem, op);
    DeleteObject(pen);


    gdiDot(g, mapX(n-1), mapY(data[n-1]), 4, c);
}



void drawBars(GDICtx& g,
              const std::vector<double>& data,
              int ox, int oy, int ow, int oh,
              double maxVal, Color c, int offset = 0) {
    int n  = (int)data.size();
    int bw = ow / n;
    int pw = std::max(2, bw/2 - 2);
    int base = oy + oh;

    for (int i = 0; i < n; i++) {
        double norm = std::max(0.0, std::min(1.0, data[i] / maxVal));
        int bh = (int)(norm * oh);
        int bx = ox + i * bw + offset;
        int by = base - bh;
```

```
        for (int s = 0; s < bh; s++) {
            double t = (double)s / oh;
            Color lc = {
                (int)(c.r * (0.3 + 0.7*t)),
                (int)(c.g * (0.3 + 0.7*t)),
                (int)(c.b * (0.3 + 0.7*t))
            };
            gdiLine(g, bx, by+s, bx+pw, by+s, lc);
        }
        gdiLine(g, bx, by, bx+pw, by, c, 2);
    }
}

void drawPanel(GDICtx& g, int x, int y, int w, int h,
               const char* title, Color titleColor) {
    HBRUSH br = CreateSolidBrush(RGB(15, 18, 30));
    RECT r = {x, y, x+w, y+h};
    FillRect(g.mem, &r, br);
    DeleteObject(br);

    gdiLine(g, x,   y,   x+w, y,   COL_AXIS, 1);
    gdiLine(g, x,   y,   x,   y+h, COL_AXIS, 1);
    gdiLine(g, x+w, y,   x+w, y+h, COL_AXIS, 1);
    gdiLine(g, x,   y+h, x+w, y+h, COL_AXIS, 1);

    HBRUSH hdr = CreateSolidBrush(RGB(20, 35, 60));
    RECT hr = {x+1, y+1, x+w-1, y+20};
    FillRect(g.mem, &hr, hdr);
    DeleteObject(hdr);

    gdiText(g, x+6, y+4, title, titleColor, 12, true);
}

void drawThreshold(GDICtx& g, int ox, int oy, int ow,
                   double val, double yMax, Color c,
                   int plotH, const char* label) {
    int y = oy + plotH - (int)(val / yMax * plotH);
    gdiLine(g, ox, y, ox+ow, y, c, 1, PS_DOT);
    gdiText(g, ox+ow+4, y-7, label, c, 11);
}

static double noise(double t) {
    return sin(t*7.3)*0.15 + sin(t*13.1)*0.09 + cos(t*23.7)*0.04;
}
```

```
int main()
{

    std::cout << "Boost version: " << BOOST_VERSION << std::endl;
    HANDLE hCon = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_CURSOR_INFO ci = {1, FALSE};
    SetConsoleCursorInfo(hCon, &ci);
    SetConsoleTitleA("ConsoleGraph :: GDI Backend");

    GDICtx g;
    if (!g.init()) {
        printf("GetConsoleWindow() failed.\n");
        return 1;
    }

    const int N = 256;
    double t = 0.0;

    std::vector<double> sig1(N), sig2(N);
    std::vector<double> cpu(24), mem(24);

    srand(42);
    for (int i = 0; i < 24; i++) {
        cpu[i] = 20 + rand() % 60;
        mem[i] = 30 + rand() % 40;
    }

    COORD cur = {0, 9999};
    SetConsoleCursorPosition(hCon, cur);

    int frame = 0;

    for (;;) {
        DWORD t0 = GetTickCount();

        g.resize();

        const int PAD  = 8;
        const int TH   = 22;
        int totalW = g.W - PAD*2;
        int panelH = (g.H - TH - PAD*3) / 2;

        for (int i = 0; i < N; i++) {
            double x = (double)i / N * 4 * PI;
```

```
    sig1[i] = sin(x + t) * 0.75 + noise(x + t);
    sig2[i] = cos(x + t * 0.65) * 0.6 + sin(x*0.5 + t) * 0.25;
}
for (int i = 0; i < 24; i++) {
    cpu[i] += sin(t*2.0 + i*0.7) * 0.9;
    mem[i] += cos(t*1.4 + i*1.1) * 0.6;
    cpu[i] = std::max(5.0, std::min(95.0, cpu[i]));
    mem[i] = std::max(8.0, std::min(80.0, mem[i]));
}

g.clear();

{
    HBRUSH hb = CreateSolidBrush(RGB(10, 30, 60));
    RECT hr = {0, 0, g.W, TH};
    FillRect(g.mem, &hr, hb);
    DeleteObject(hb);

    gdiText(g, PAD, 4,
            "ConsoleGraph  v0.1  |  GDI Backend  |  Windows",
            COL_TITLE, 13, true);

    char info[128];
    snprintf(info, sizeof(info),
             "frame: %d   t: %.2f   %dx%d   q:quit",
             frame, t, g.W, g.H);
    SIZE sz; GetTextExtentPoint32A(g.mem, info, (int)strlen(info), &sz);
    gdiText(g, g.W - sz.cx - PAD, 4, info, COL_ACCENT, 11);
}

int p1x = PAD;
int p1y = TH + PAD;
int p1w = totalW;
int p1h = panelH;
int plotPad = 28;
int plotX = p1x + plotPad;
int plotY = p1y + 22;
int plotW = p1w - plotPad - 40;
int plotH = p1h - 30;

drawPanel(g, p1x, p1y, p1w, p1h,
          "Signal Monitor  [ sin / cos + noise ]", COL_SIN);
drawGrid (g, plotX, plotY, plotW, plotH, 12, 6);
drawAxes (g, plotX, plotY, plotW, plotH);
```

```
gdiText(g, p1x+2, plotY,          "+1.0", COL_AXIS, 10);
gdiText(g, p1x+2, plotY+plotH/2-6, " 0.0", COL_AXIS, 10);
gdiText(g, p1x+2, plotY+plotH-12, "-1.0", COL_AXIS, 10);

drawPolyline(g, sig2, plotX, plotY, plotW, plotH, -1.2, 1.2, COL_COS, 2);
drawPolyline(g, sig1, plotX, plotY, plotW, plotH, -1.2, 1.2, COL_SIN, 2);

gdiDot(g, p1x + p1w - 170, p1y + 6, 5, COL_SIN);
gdiText(g, p1x + p1w - 162, p1y + 1, "sin + noise", COL_SIN, 11);
gdiDot(g, p1x + p1w -  80, p1y + 6, 5, COL_COS);
gdiText(g, p1x + p1w -  72, p1y + 1, "cos", COL_COS, 11);

int p2x = PAD;
int p2y = TH + PAD*2 + panelH;
int p2w = totalW;
int p2h = panelH;
int p2plotX = p2x + plotPad;
int p2plotY = p2y + 22;
int p2plotW = p2w - plotPad - 40;
int p2plotH = p2h - 30;

drawPanel(g, p2x, p2y, p2w, p2h,
        "Resource Monitor  [ CPU line / MEM bars ]", COL_CPU);
drawGrid (g, p2plotX, p2plotY, p2plotW, p2plotH, 12, 5);

gdiText(g, p2x+2, p2plotY,           "100%", COL_AXIS, 10);
gdiText(g, p2x+2, p2plotY+p2plotH/2-6, " 50%", COL_AXIS, 10);
gdiText(g, p2x+2, p2plotY+p2plotH-12,  "  0%", COL_AXIS, 10);

drawThreshold(g, p2plotX, p2plotY, p2plotW,
            80, 100, COL_WARN, p2plotH, "80%");
drawThreshold(g, p2plotX, p2plotY, p2plotW,
            50, 100, COL_CAUTION, p2plotH, "50%");

int halfW = p2plotW / 2 - 4;
drawPolyline(g, cpu, p2plotX, p2plotY, halfW, p2plotH,
           0, 100, COL_CPU, 2);

drawBars(g, mem, p2plotX + halfW + 4, p2plotY,
        halfW, p2plotH, 100, COL_MEM);

gdiLine(g, p2plotX + halfW + 2, p2plotY,
          p2plotX + halfW + 2, p2plotY + p2plotH,
          COL_AXIS, 1, PS_DOT);
```

```
        gdiDot(g, p2x + p2w - 170, p2y + 6, 5, COL_CPU);
        gdiText(g, p2x + p2w - 162, p2y + 1, "CPU %", COL_CPU, 11);
        gdiDot(g, p2x + p2w -  90, p2y + 6, 5, COL_MEM);
        gdiText(g, p2x + p2w -  82, p2y + 1, "MEM %", COL_MEM, 11);


        {
           HBRUSH sb = CreateSolidBrush(RGB(10, 30, 60));
           RECT sr = {0, g.H-18, g.W, g.H};
           FillRect(g.mem, &sr, sb);
           DeleteObject(sb);
           gdiText(g, PAD, g.H-15,
                 "Backend: Windows GDI  |  GetConsoleWindow()  |  Double-buffered
|  Ctrl+C to exit",
                 {55, 80, 120}, 11);
        }

        g.present();

        DWORD elapsed = GetTickCount() - t0;
        if (elapsed < 33) Sleep(33 - elapsed);

        t += 0.05;
        frame++;
    }

    g.free();
    return 0;
}
```

А.2 Текст файлу testdiff.cpp

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <vector>
#include <string>
#include <algorithm>
#include <sys/ioctl.h>
#include <unistd.h>
#include <termios.h>

#define PI 3.14
```

```
struct Color { int r, g, b; };

constexpr Color COL_BG      = {10,  12,  20};
constexpr Color COL_GRID    = {25,  38,  58};
constexpr Color COL_AXIS    = {55,  80, 120};
constexpr Color COL_SIN     = {0,  210, 255};
constexpr Color COL_COS     = {255, 100,  50};
constexpr Color COL_CPU     = {40,  200, 120};
constexpr Color COL_MEM     = {200,  60, 180};
constexpr Color COL_TEXT    = {180, 200, 230};
constexpr Color COL_TITLE   = {255, 255, 255};
constexpr Color COL_WARN    = {255,  60,  60};
constexpr Color COL_CAUTION = {255, 200,  50};
constexpr Color COL_ACCENT  = {255, 220,   0};

struct Cell {
    char ch[5];
    Color fg, bg;
    bool operator==(const Cell& o) const {
        return memcmp(this, &o, sizeof(Cell)) == 0;
    }
};

struct ANSICtx {
    int W, H;
    std::vector<Cell> buf, prev;

    void init() {
        getSize();
        buf .assign(W * H, {{" "}, COL_TEXT, COL_BG});
        prev.assign(W * H, {{"?"}, {0,0,0},  {0,0,0}});
        printf("\x1b[2J\x1b[?25l");
        fflush(stdout);
    }

    void getSize() {
        struct winsize ws;
        ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws);
        W = ws.ws_col > 0 ? ws.ws_col : 120;
        H = ws.ws_row > 0 ? ws.ws_row : 40;
    }

    void resize() {
        int nw, nh;
```

```
    struct winsize ws;
    ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws);
    nw = ws.ws_col > 0 ? ws.ws_col : 120;
    nh = ws.ws_row > 0 ? ws.ws_row : 40;
    if (nw == W && nh == H) return;
    W = nw; H = nh;
    buf .assign(W * H, {{" "}, COL_TEXT, COL_BG});
    prev.assign(W * H, {{"?"}, {0,0,0},  {0,0,0}});
    printf("\x1b[2J");
}

Cell& at(int x, int y) { return buf[y * W + x]; }

void set(int x, int y, const char* ch, Color fg, Color bg = COL_BG) {
    if (x < 0 || x >= W || y < 0 || y >= H) return;
    Cell& c = at(x, y);
    strncpy(c.ch, ch, 4); c.ch[4] = 0;
    c.fg = fg; c.bg = bg;
}

void clear() {
    for (auto& c : buf) { strcpy(c.ch, " "); c.fg = COL_TEXT; c.bg = COL_BG; }
}

void present() {
    Color lastFg = {-1,-1,-1}, lastBg = {-1,-1,-1};
    for (int y = 0; y < H; y++) {
        for (int x = 0; x < W; x++) {
            Cell& c = at(x, y);
            Cell& p = prev[y * W + x];
            if (c == p) continue;
            printf("\x1b[%d;%dH", y + 1, x + 1);
            if (c.fg.r != lastFg.r || c.fg.g != lastFg.g || c.fg.b != lastFg.b) {
                printf("\x1b[38;2;%d;%d;%dm", c.fg.r, c.fg.g, c.fg.b);
                lastFg = c.fg;
            }
            if (c.bg.r != lastBg.r || c.bg.g != lastBg.g || c.bg.b != lastBg.b) {
                printf("\x1b[48;2;%d;%d;%dm", c.bg.r, c.bg.g, c.bg.b);
                lastBg = c.bg;
            }
            printf("%s", c.ch);
            p = c;
        }
    }
    fflush(stdout);
```

```
    }

    void free() {
        printf("\x1b[0m\x1b[2J\x1b[H\x1b[?25h");
        fflush(stdout);
    }
};

void gdiLine(ANSICtx& g, int x0, int y0, int x1, int y1,
             Color c, int w = 1, int style = 0)
{
    int dx = abs(x1-x0), dy = abs(y1-y0);
    int sx = x0 < x1 ? 1 : -1;
    int sy = y0 < y1 ? 1 : -1;
    int err = dx - dy;
    const char* ch = (dy > dx) ? " │ " : "─";
    if (dx > 0 && dy > 0) ch = (sx == sy) ? "\\" : "/";
    if (style != 0) ch = "--";
    while (true) {
        g.set(x0, y0, ch, c);
        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x0 += sx; }
        if (e2 <  dx) { err += dx; y0 += sy; }
    }
}

void gdiText(ANSICtx& g, int x, int y, const char* s,
             Color c, int size = 0, bool bold = false)
{
    for (int i = 0; s[i]; i++) {
        char ch[2] = {s[i], 0};
        g.set(x + i, y, ch, c);
    }
}

void gdiDot(ANSICtx& g, int x, int y, int r, Color c) {
    g.set(x, y, "●", c);
}

void gdiRect(ANSICtx& g, int x, int y, int w, int h,
             Color fill, Color border, int bw = 1)
{
    for (int row = y; row < y+h; row++)
        for (int col = x; col < x+w; col++)
```

```
            g.set(col, row, " ", fill, fill);
}

void drawGrid(ANSICtx& g, int ox, int oy, int ow, int oh,
              int cols = 10, int rows = 8)
{
    for (int i = 0; i <= cols; i++) {
        int x = ox + i * ow / cols;
        for (int row = oy; row < oy+oh; row++)
            g.set(x, row, "┆", COL_GRID);
    }
    for (int j = 0; j <= rows; j++) {
        int y = oy + j * oh / rows;
        for (int col = ox; col < ox+ow; col++)
            g.set(col, y, "--", COL_GRID);
    }
}

void drawAxes(ANSICtx& g, int ox, int oy, int ow, int oh)
{
    int mid = oy + oh / 2;
    for (int col = ox; col < ox+ow; col++) g.set(col, mid, "─", COL_AXIS);
    for (int row = oy; row < oy+oh; row++) g.set(ox,  row, " │", COL_AXIS);
    g.set(ox+ow-1, mid, "►", COL_AXIS);
}

void drawPolyline(ANSICtx& g,
                  const std::vector<double>& data,
                  int ox, int oy, int ow, int oh,
                  double yMin, double yMax,
                  Color c, int lw = 1)
{
    int n = (int)data.size();
    if (n < 2) return;

    static const char* blocks[] = {" ","▁","▂","▃","▄","▅","▆","▇","█"};

    for (int px = 0; px < ow; px++) {
        int idx = (int)((double)px / ow * (n - 1));
        double v = data[std::min(idx, n-1)];
        double norm = (v - yMin) / (yMax - yMin);
        norm = std::max(0.0, std::min(1.0, norm));
        int top = oy + oh - 1 - (int)(norm * (oh - 1));
        int mid = oy + oh / 2;
```

```
        int fill_from = std::min(top, mid);
        int fill_to   = std::max(top, mid);
        for (int row = fill_from; row <= fill_to; row++) {
            Color dim = {c.r/5, c.g/5, c.b/5};
            g.set(ox + px, row, "░", dim);
        }
        g.set(ox + px, top, "█", c);
    }
}

void drawBars(ANSICtx& g,
              const std::vector<double>& data,
              int ox, int oy, int ow, int oh,
              double maxVal, Color c, int offset = 0)
{
    int n  = (int)data.size();
    int bw = std::max(1, ow / n);

    for (int i = 0; i < n && i * bw < ow; i++) {
        double norm = std::max(0.0, std::min(1.0, data[i] / maxVal));
        int bh   = (int)(norm * (oh - 1));
        int base = oy + oh - 1;

        for (int row = 0; row < bh; row++) {
            double t = (double)row / oh;
            Color lc = {(int)(c.r*(0.3+0.7*t)), (int)(c.g*(0.3+0.7*t)),
(int)(c.b*(0.3+0.7*t))};
            g.set(ox + i * bw, base - row, "▓", lc);
        }
        g.set(ox + i * bw, base - bh, "▄", c);
    }
}

void drawPanel(ANSICtx& g, int x, int y, int w, int h,
               const char* title, Color titleColor)
{
    Color dark = {15, 18, 30};
    for (int row = y; row < y+h; row++)
        for (int col = x; col < x+w; col++)
            g.set(col, row, " ", dark, dark);

    for (int col = x+1; col < x+w-1; col++) {
        g.set(col, y,     "═", COL_AXIS);
        g.set(col, y+h-1, "═", COL_AXIS);
    }
```

```
    for (int row = y+1; row < y+h-1; row++) {
        g.set(x,     row, "│", COL_AXIS);
        g.set(x+w-1, row, "│", COL_AXIS);
    }
    g.set(x,     y,     "╔", COL_AXIS);
    g.set(x+w-1, y,     "╗", COL_AXIS);
    g.set(x,     y+h-1, "╚", COL_AXIS);
    g.set(x+w-1, y+h-1, "╝", COL_AXIS);

    Color hdrBg = {20, 35, 60};
    for (int col = x+1; col < x+w-1; col++)
        g.set(col, y, " ", titleColor, hdrBg);
    gdiText(g, x+2, y, title, titleColor);
}

void drawThreshold(ANSICtx& g, int ox, int oy, int ow,
                   double val, double yMax, Color c,
                   int plotH, const char* label)
{
    int y = oy + plotH - 1 - (int)(val / yMax * (plotH - 1));
    for (int col = ox; col < ox+ow; col++)
        g.set(col, y, "--", c);
    gdiText(g, ox + ow + 1, y, label, c);
}

static double noise(double t) {
    return sin(t*7.3)*0.15 + sin(t*13.1)*0.09 + cos(t*23.7)*0.04;
}

int main()
{
    ANSICtx g;
    g.init();

    const int N = 200;
    double t = 0.0;

    std::vector<double> sig1(N), sig2(N);
    std::vector<double> cpu(24), mem(24);

    srand(42);
    for (int i = 0; i < 24; i++) {
        cpu[i] = 20 + rand() % 60;
        mem[i] = 30 + rand() % 40;
    }
```

```
    int frame = 0;

    for (;;) {
        g.resize();

        const int PAD = 2;
        const int TH  = 1;
        int panelH = (g.H - TH - PAD*3) / 2;

        for (int i = 0; i < N; i++) {
            double x = (double)i / N * 4 * PI;
            sig1[i] = sin(x + t) * 0.75 + noise(x + t);
            sig2[i] = cos(x + t * 0.65) * 0.6 + sin(x*0.5 + t) * 0.25;
        }
        for (int i = 0; i < 24; i++) {
            cpu[i] += sin(t*2.0 + i*0.7) * 0.9;
            mem[i] += cos(t*1.4 + i*1.1) * 0.6;
            cpu[i] = std::max(5.0,  std::min(95.0, cpu[i]));
            mem[i] = std::max(8.0,  std::min(80.0, mem[i]));
        }

        g.clear();

        for (int col = 0; col < g.W; col++)
            g.set(col, 0, " ", COL_TITLE, {10,30,60});
        char hdr[256];
        snprintf(hdr, sizeof(hdr),
                 " ConsoleGraph v0.1  |  ANSI Backend  |  Linux/SSH  |  frame:%d",
frame);
        gdiText(g, 0, 0, hdr, COL_TITLE);
        char info[64];
        snprintf(info, sizeof(info), "t:%.2f  %dx%d ", t, g.W, g.H);
        gdiText(g, g.W - (int)strlen(info), 0, info, COL_ACCENT);

        int p1x = PAD, p1y = TH + PAD;
        int p1w = g.W - PAD*2, p1h = panelH;
        int ppad = 6;
        int plotX = p1x + ppad, plotY = p1y + 1;
        int plotW = p1w - ppad - 5, plotH = p1h - 2;

        drawPanel(g, p1x, p1y, p1w, p1h,
                  "Signal Monitor  [ sin / cos + noise ]", COL_SIN);
        drawGrid (g, plotX, plotY, plotW, plotH, 12, 6);
        drawAxes (g, plotX, plotY, plotW, plotH);
```

```
        gdiText(g, p1x, plotY,          "+1.0", COL_AXIS);
        gdiText(g, p1x, plotY+plotH/2,   " 0.0", COL_AXIS);
        gdiText(g, p1x, plotY+plotH-1,   "-1.0", COL_AXIS);

        drawPolyline(g, sig2, plotX, plotY, plotW, plotH, -1.2, 1.2, COL_COS);
        drawPolyline(g, sig1, plotX, plotY, plotW, plotH, -1.2, 1.2, COL_SIN);

        gdiDot(g, p1x+p1w-20, p1y, 1, COL_SIN);
        gdiText(g, p1x+p1w-19, p1y, "sin+noise", COL_SIN);
        gdiDot(g, p1x+p1w-8,  p1y, 1, COL_COS);
        gdiText(g, p1x+p1w-7,  p1y, "cos", COL_COS);

        int p2x = PAD, p2y = TH + PAD*2 + panelH;
        int p2w = g.W - PAD*2, p2h = panelH;
        int p2plotX = p2x + ppad, p2plotY = p2y + 1;
        int p2plotW = p2w - ppad - 5, p2plotH = p2h - 2;

        drawPanel(g, p2x, p2y, p2w, p2h,
                "Resource Monitor  [ CPU line / MEM bars ]", COL_CPU);
        drawGrid (g, p2plotX, p2plotY, p2plotW, p2plotH, 12, 5);

        gdiText(g, p2x, p2plotY,            "100%", COL_AXIS);
        gdiText(g, p2x, p2plotY+p2plotH/2,   " 50%", COL_AXIS);
        gdiText(g, p2x, p2plotY+p2plotH-1,   "  0%", COL_AXIS);

        drawThreshold(g, p2plotX, p2plotY, p2plotW, 80, 100, COL_WARN,
p2plotH, "80%");
        drawThreshold(g, p2plotX, p2plotY, p2plotW, 50, 100, COL_CAUTION,
p2plotH, "50%");

        int halfW = p2plotW / 2 - 2;
        drawPolyline(g, cpu, p2plotX,              p2plotY, halfW, p2plotH, 0, 100,
COL_CPU);
        drawBars     (g, mem, p2plotX + halfW+2, p2plotY, halfW, p2plotH, 100,
COL_MEM);

        for (int row = p2plotY; row < p2plotY+p2plotH; row++)
           g.set(p2plotX + halfW + 1, row, "┆", COL_AXIS);

        gdiDot(g, p2x+p2w-14, p2y, 1, COL_CPU); gdiText(g, p2x+p2w-13, p2y,
"CPU%", COL_CPU);
        gdiDot(g, p2x+p2w-7,  p2y, 1, COL_MEM); gdiText(g, p2x+p2w-6,  p2y,
"MEM%", COL_MEM);
```

```
        for (int col = 0; col < g.W; col++)
            g.set(col, g.H-1, " ", COL_AXIS, {10,30,60});
        gdiText(g, 1, g.H-1,
                "Backend: ANSI/UTF-8 | Linux/SSH | Ctrl+C to exit",
                COL_AXIS);

        g.present();

        usleep(50000);
        t += 0.05;
        frame++;
    }

    g.free();
    return 0;
}
```

A.3 Текст файлу ANSIBackend.cpp

```
#include "ANSIBackend.h"

#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>

#ifndef _WIN32
#include <sys/ioctl.h>
#include <unistd.h>
#endif

namespace cg {

void ANSIBackend::queryTerminalSize() {
#ifndef _WIN32
    struct winsize ws;
    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws) == 0 &&
        ws.ws_col > 0 && ws.ws_row > 0) {
        W_ = ws.ws_col;
        H_ = ws.ws_row;
        return;
    }
#endif
    W_ = 120;
```

```
    H_ = 40;
}


bool ANSIBackend::init() {
    queryTerminalSize();
    Cell empty;
    empty.fg = palette::TEXT;
    empty.bg = bg_;
    buf_.assign(static_cast<size_t>(W_) * H_, empty);

    Cell dirty;
    dirty.ch[0] = '\1';
    prev_.assign(static_cast<size_t>(W_) * H_, dirty);

    std::printf("\x1b[2J\x1b[?25l");
    std::fflush(stdout);
    return true;
}

void ANSIBackend::shutdown() {
    std::printf("\x1b[0m\x1b[2J\x1b[H\x1b[?25h");
    std::fflush(stdout);
}

void ANSIBackend::resize() {
    int oldW = W_, oldH = H_;
    queryTerminalSize();
    if (W_ == oldW && H_ == oldH) return;

    Cell empty;
    empty.fg = palette::TEXT;
    empty.bg = bg_;
    buf_.assign(static_cast<size_t>(W_) * H_, empty);

    Cell dirty;
    dirty.ch[0] = '\1';
    prev_.assign(static_cast<size_t>(W_) * H_, dirty);

    std::printf("\x1b[2J");
    std::fflush(stdout);
}

Cell& ANSIBackend::at(int x, int y) {
    return buf_[static_cast<size_t>(y) * W_ + x];
```

```
}

void ANSIBackend::setCell(int x, int y, const char* ch,
                          Color fg, Color bg) {
    if (x < 0 || x >= W_ || y < 0 || y >= H_) return;
    Cell& c = at(x, y);
    std::strncpy(c.ch, ch, 4);
    c.ch[4] = 0;
    c.fg = fg;
    c.bg = bg;
}

int ANSIBackend::nx(double x) const {
    int v = static_cast<int>(x * (W_ - 1) + 0.5);
    return std::max(0, std::min(W_ - 1, v));
}

int ANSIBackend::ny(double y) const {
    int v = static_cast<int>(y * (H_ - 1) + 0.5);
    return std::max(0, std::min(H_ - 1, v));
}

// frame
void ANSIBackend::clear(Color bg) {
    bg_ = bg;
    for (auto& c : buf_) {
        c.ch[0] = ' '; c.ch[1] = 0;
        c.fg = palette::TEXT;
        c.bg = bg;
    }
}

void ANSIBackend::present() {
    Color lastFg{1, 2, 3}, lastBg{4, 5, 6};
    bool  haveColor = false;

    for (int y = 0; y < H_; ++y) {
        for (int x = 0; x < W_; ++x) {
            Cell& c = at(x, y);
            Cell& p = prev_[static_cast<size_t>(y) * W_ + x];
            if (c == p) continue;

            std::printf("\x1b[%d;%dH", y + 1, x + 1);
            if (!haveColor || c.fg != lastFg) {
                std::printf("\x1b[38;2;%d;%d;%dm", c.fg.r, c.fg.g, c.fg.b);
```

```
            lastFg = c.fg;
        }
        if (!haveColor || c.bg != lastBg) {
            std::printf("\x1b[48;2;%d;%d;%dm", c.bg.r, c.bg.g, c.bg.b);
            lastBg = c.bg;
        }
        haveColor = true;
        std::printf("%s", c.ch);
        p = c;
      }
   }
   std::fflush(stdout);
}

//parts
void ANSIBackend::drawLine(Point a, Point b, Color c, int ) {
   int x0 = nx(a.x), y0 = ny(a.y);
   int x1 = nx(b.x), y1 = ny(b.y);

   int dx = std::abs(x1 - x0), dy = std::abs(y1 - y0);
   int sx = x0 < x1 ? 1 : -1;
   int sy = y0 < y1 ? 1 : -1;
   int err = dx - dy;

   const char* ch = (dy > dx) ? "\u2502" : "\u2500"; //
   if (dx > 0 && dy > 0) ch = (sx == sy) ? "\\" : "/";

   while (true) {
      setCell(x0, y0, ch, c, bg_);
      if (x0 == x1 && y0 == y1) break;
      int e2 = 2 * err;
      if (e2 > -dy) { err -= dy; x0 += sx; }
      if (e2 <  dx) { err += dx; y0 += sy; }
   }
}

void ANSIBackend::drawRect(Rect r, Color fill, Color ) {
   int x0 = nx(r.x),         y0 = ny(r.y);
   int x1 = nx(r.x + r.w),   y1 = ny(r.y + r.h);
   for (int yy = y0; yy <= y1; ++yy)
      for (int xx = x0; xx <= x1; ++xx)
         setCell(xx, yy, " ", fill, fill);
}

void ANSIBackend::drawDot(Point p, int , Color c) {
```

```
    setCell(nx(p.x), ny(p.y), "\u25CF", c, bg_);
}

void ANSIBackend::drawText(Point p, const std::string& text, Color c, int ) {
    int x = nx(p.x), y = ny(p.y);
    for (size_t i = 0; i < text.size() && x + (int)i < W_; ++i) {
        char ch[2] = { text[i], 0 };
        setCell(x + static_cast<int>(i), y, ch, c, bg_);
    }
}

}
```

А.4 Текст файлу BackendFactory.cpp

```
#include "BackendFactory.h"
#include "ANSIBackend.h"
#include "JSONSerializerBackend.h"

#ifdef _WIN32
#include "GDIBackend.h"
#endif

#include <cstdlib>

namespace cg {

std::unique_ptr<IBackend> BackendFactory::create(SSHTransport* transport) {

    const char* client = std::getenv("CONSOLEGRAPH_CLIENT");
    if (client && transport) {
        return std::unique_ptr<IBackend>(
            new JSONSerializerBackend(transport));
    }

#ifdef _WIN32
    return std::unique_ptr<IBackend>(new GDIBackend());
#else
    return std::unique_ptr<IBackend>(new ANSIBackend());
#endif
}

}
```

A.5 Текст файлу GDIBackend.cpp

```
#include "GDIBackend.h"

#ifdef _WIN32

#include <algorithm>

extern "C" WINBASEAPI HWND WINAPI GetConsoleWindow();

namespace cg {

static COLORREF toCOLORREF(Color c) { return RGB(c.r, c.g, c.b); }

int GDIBackend::px(double x) const {
    int v = static_cast<int>(x * (W_ - 1) + 0.5);
    return std::max(0, std::min(W_ - 1, v));
}
int GDIBackend::py(double y) const {
    int v = static_cast<int>(y * (H_ - 1) + 0.5);
    return std::max(0, std::min(H_ - 1, v));
}

bool GDIBackend::init() {
    hwnd_ = GetConsoleWindow();
    if (!hwnd_) return false;

    RECT rc;
    GetClientRect(hwnd_, &rc);
    W_ = rc.right - rc.left;
    H_ = rc.bottom - rc.top;
    if (W_ <= 0) W_ = 800;
    if (H_ <= 0) H_ = 600;

    hdc_ = GetDC(hwnd_);
    mem_ = CreateCompatibleDC(hdc_);
    bmp_ = CreateCompatibleBitmap(hdc_, W_, H_);
    SelectObject(mem_, bmp_);
    return true;
}

void GDIBackend::shutdown() {
    if (bmp_) DeleteObject(bmp_);
    if (mem_) DeleteDC(mem_);
```

```
    if (hdc_) ReleaseDC(hwnd_, hdc_);
    bmp_ = nullptr; mem_ = nullptr; hdc_ = nullptr;
}

void GDIBackend::resize() {
    RECT rc;
    GetClientRect(hwnd_, &rc);
    int nw = rc.right - rc.left;
    int nh = rc.bottom - rc.top;
    if (nw == W_ && nh == H_) return;
    if (nw <= 0 || nh <= 0) return;

    W_ = nw; H_ = nh;
    DeleteObject(bmp_);
    DeleteDC(mem_);
    mem_ = CreateCompatibleDC(hdc_);
    bmp_ = CreateCompatibleBitmap(hdc_, W_, H_);
    SelectObject(mem_, bmp_);
}

void GDIBackend::clear(Color bg) {
    HBRUSH br = CreateSolidBrush(toCOLORREF(bg));
    RECT r = { 0, 0, W_, H_ };
    FillRect(mem_, &r, br);
    DeleteObject(br);
}

void GDIBackend::present() {
    BitBlt(hdc_, 0, 0, W_, H_, mem_, 0, 0, SRCCOPY);
}

void GDIBackend::drawLine(Point a, Point b, Color c, int thickness) {
    HPEN pen = CreatePen(PS_SOLID, thickness, toCOLORREF(c));
    HGDIOBJ old = SelectObject(mem_, pen);
    MoveToEx(mem_, px(a.x), py(a.y), nullptr);
    LineTo(mem_, px(b.x), py(b.y));
    SelectObject(mem_, old);
    DeleteObject(pen);
}

void GDIBackend::drawRect(Rect r, Color fill, Color border) {
    HBRUSH br = CreateSolidBrush(toCOLORREF(fill));
    HPEN   pn = CreatePen(PS_SOLID, 1, toCOLORREF(border));
    HGDIOBJ ob = SelectObject(mem_, br);
    HGDIOBJ op = SelectObject(mem_, pn);
```

```
    Rectangle(mem_, px(r.x), py(r.y),
              px(r.x + r.w), py(r.y + r.h));
    SelectObject(mem_, ob);
    SelectObject(mem_, op);
    DeleteObject(br);
    DeleteObject(pn);
}

void GDIBackend::drawDot(Point p, int radius, Color c) {
    HPEN   pn = CreatePen(PS_SOLID, 1, toCOLORREF(c));
    HBRUSH br = CreateSolidBrush(toCOLORREF(c));
    HGDIOBJ op = SelectObject(mem_, pn);
    HGDIOBJ ob = SelectObject(mem_, br);
    int x = px(p.x), y = py(p.y);
    Ellipse(mem_, x - radius, y - radius, x + radius, y + radius);
    SelectObject(mem_, op);
    SelectObject(mem_, ob);
    DeleteObject(pn);
    DeleteObject(br);
}

void GDIBackend::drawText(Point p, const std::string& text,
                          Color c, int size) {
    SetBkMode(mem_, TRANSPARENT);
    SetTextColor(mem_, toCOLORREF(c));
    HFONT f = CreateFontA(size, 0, 0, 0, FW_NORMAL,
                          0, 0, 0, DEFAULT_CHARSET,
                          OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
                          CLEARTYPE_QUALITY,
                          DEFAULT_PITCH | FF_DONTCARE, "Consolas");
    HGDIOBJ old = SelectObject(mem_, f);
    TextOutA(mem_, px(p.x), py(p.y),
             text.c_str(), static_cast<int>(text.size()));
    SelectObject(mem_, old);
    DeleteObject(f);
}

}

#endif
```

А.6 Текст файлу consolegraph.cpp

```
#include "Application.h"
#include "CGTypes.h"

#ifdef _WIN32
#include "GDIBackend.h"
#include <windows.h>
#define SLEEP_MS(x) Sleep(x)
#else
#include "ANSIBackend.h"
#include <unistd.h>
#define SLEEP_MS(x) usleep((x) * 1000)
#endif

#include <cstdio>
#include <cstring>
#include <string>

namespace cg {

//
bool Application::parseArgs(int argc, char** argv) {
    for (int i = 1; i < argc; ++i) {
        std::string a = argv[i];
        if (a == "-ssh" && i + 1 < argc) {
            std::string target = argv[++i];
            size_t at = target.find('@');
            if (at == std::string::npos) return false;
            user_ = target.substr(0, at);
            std::string hp = target.substr(at + 1);
            size_t colon = hp.find(':');
            if (colon != std::string::npos) {
                host_ = hp.substr(0, colon);
                port_ = std::atoi(hp.substr(colon + 1).c_str());
            } else {
                host_ = hp;
            }
        } else if (a[0] != '-') {
            appPath_ = a;
        }
    }
    if (host_.empty() || user_.empty()) return false;
```

```
    // r password
    const char* envPwd = std::getenv("CONSOLEGRAPH_PASSWORD");
    if (envPwd) {
        password_ = envPwd;
    } else {
        std::printf("Password for %s@%s: ", user_.c_str(), host_.c_str());
        std::fflush(stdout);
        char buf[256] = {0};
        if (std::fgets(buf, sizeof(buf), stdin)) {
            password_ = buf;
            while (!password_.empty() &&
                   (password_.back() == '\n' || password_.back() == '\r'))
                password_.pop_back();
        }
    }
    return true;
}

bool Application::connectToServer() {

    std::string cmd;
    if (!appPath_.empty()) {
        cmd = "CONSOLEGRAPH_CLIENT=1 " + appPath_;
    }
    if (!transport_.connect(host_, port_, user_, password_, cmd)) {
        std::printf("Connection failed\n");
        return false;
    }
    return true;
}

void Application::handleCommand(const json::Object& cmd) {
    std::string op = cmd.getStr("op");
    if (op.empty() || !renderer_) return;

    auto W = renderer_->width();
    auto H = renderer_->height();
    auto NX = [&](double x){ return x; };
    auto NY = [&](double y){ return y; };
    (void)W; (void)H;

    if (op == "clear") {
        renderer_->clear({ (uint8_t)cmd.getInt("r"),
                           (uint8_t)cmd.getInt("g"),
                           (uint8_t)cmd.getInt("b") });
```

```
    } else if (op == "present") {
        renderer_->present();
    } else if (op == "line") {
        renderer_->drawLine(
            { NX(cmd.getNum("x1")), NY(cmd.getNum("y1")) },
            { NX(cmd.getNum("x2")), NY(cmd.getNum("y2")) },
            { (uint8_t)cmd.getInt("r"), (uint8_t)cmd.getInt("g"),
              (uint8_t)cmd.getInt("b") },
            (int)cmd.getInt("t", 1));
    } else if (op == "rect") {
        renderer_->drawRect(
            { cmd.getNum("x"), cmd.getNum("y"),
              cmd.getNum("w"), cmd.getNum("h") },
            { (uint8_t)cmd.getInt("fr"), (uint8_t)cmd.getInt("fg"),
              (uint8_t)cmd.getInt("fb") },
            { (uint8_t)cmd.getInt("br"), (uint8_t)cmd.getInt("bg"),
              (uint8_t)cmd.getInt("bb") });
    } else if (op == "dot") {
        renderer_->drawDot(
            { cmd.getNum("x"), cmd.getNum("y") },
            (int)cmd.getInt("rad", 1),
            { (uint8_t)cmd.getInt("r"), (uint8_t)cmd.getInt("g"),
              (uint8_t)cmd.getInt("b") });
    } else if (op == "text") {
        renderer_->drawText(
            { cmd.getNum("x"), cmd.getNum("y") },
            cmd.getStr("s"),
            { (uint8_t)cmd.getInt("r"), (uint8_t)cmd.getInt("g"),
              (uint8_t)cmd.getInt("b") },
            (int)cmd.getInt("sz", 12));
    } else if (op == "shutdown") {
        renderer_->shutdown();
    }
}

void Application::mainLoop() {
    std::string chunk;
    while (transport_.isConnected()) {
        if (!transport_.receive(chunk)) break;
        if (!chunk.empty()) {
            auto cmds = parser_.feed(chunk);
            for (auto& c : cmds) handleCommand(c);
        }
        SLEEP_MS(5);
    }
```

```
}

int Application::run(int argc, char** argv) {
    if (!parseArgs(argc, argv)) {
        std::printf("Usage: consolegraph -ssh user@host[:port] "
                    "[/path/to/application]\n");
        return 1;
    }

#ifdef _WIN32
    renderer_.reset(new GDIBackend());
#else
    renderer_.reset(new ANSIBackend());
#endif
    renderer_->init();

    if (!connectToServer()) return 1;
    eventSender_.reset(new EventSender(&transport_));

    mainLoop();

    renderer_->shutdown();
    transport_.disconnect();
    return 0;
}

}


int main(int argc, char** argv) {
    cg::Application app;
    return app.run(argc, argv);
}
```

A.7 Текст файлу CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(ConsoleGraph LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(DEPS "${CMAKE_SOURCE_DIR}/Dependencies")
```

```
set(LIBSSH2_INC "${DEPS}/libssh2/include")
set(LIBSSH2_LIB "${DEPS}/libssh2/lib/libssh2.a")

set(MBEDTLS_INC    "${DEPS}/mbedtls/include")
set(MBEDTLS_LIB    "${DEPS}/mbedtls/lib/libmbedtls.a")
set(MBEDX509_LIB   "${DEPS}/mbedtls/lib/libmbedx509.a")
set(MBEDCRYPTO_LIB "${DEPS}/mbedtls/lib/libmbedcrypto.a")
set(EVEREST_LIB    "${DEPS}/mbedtls/lib/libeverest.a")
set(P256M_LIB      "${DEPS}/mbedtls/lib/libp256m.a")

set(LINK_DEPS ${LIBSSH2_LIB} ${MBEDTLS_LIB}
            ${MBEDX509_LIB} ${MBEDCRYPTO_LIB})
if(EXISTS "${EVEREST_LIB}")
   list(APPEND LINK_DEPS ${EVEREST_LIB})
endif()
if(EXISTS "${P256M_LIB}")
   list(APPEND LINK_DEPS ${P256M_LIB})
endif()

set(LIB_SOURCES
   src/ANSIBackend.cpp
   src/GDIBackend.cpp
   src/JSONSerializerBackend.cpp
   src/SSHTransport.cpp
   src/BackendFactory.cpp
   src/Canvas.cpp
)

add_library(ConsoleGraph STATIC ${LIB_SOURCES})

target_include_directories(ConsoleGraph PUBLIC
   ${CMAKE_SOURCE_DIR}/include
   ${LIBSSH2_INC}
   ${MBEDTLS_INC}
)

target_link_libraries(ConsoleGraph PUBLIC ${LINK_DEPS})

if(WIN32)
   target_link_libraries(ConsoleGraph PUBLIC
      ws2_32 bcrypt crypt32 gdi32 user32)
else()
   find_package(Threads REQUIRED)
   target_link_libraries(ConsoleGraph PUBLIC Threads::Threads)
endif()
```

```
add_executable(dashboard example/dashboard.cpp)
target_link_libraries(dashboard PRIVATE ConsoleGraph)
set_target_properties(dashboard PROPERTIES
   RUNTIME_OUTPUT_DIRECTORY "${CMAKE_SOURCE_DIR}/Release")

set(UTIL_SOURCES
   utility/Application.cpp
   utility/JSONParser.cpp
   utility/EventSender.cpp
)

add_executable(consolegraph ${UTIL_SOURCES})
target_include_directories(consolegraph PRIVATE
   ${CMAKE_SOURCE_DIR}/utility)
target_link_libraries(consolegraph PRIVATE ConsoleGraph)
set_target_properties(consolegraph PROPERTIES
   RUNTIME_OUTPUT_DIRECTORY "${CMAKE_SOURCE_DIR}/Release")
```

ДОДАТОК Б

Слайди презентації

Дипломна кваліфікаційна робота бакалавра

Тема:”Створення програмної кросплатформної системи для візуалізації даних моніторингу у консольних застосунках”

Виконав ст. гр. КНТ-222 Максим ШАПОВАЛ

Керівник доц. Олександр СТЕПАНЕНКО

1

Рисунок Б.1 – Вступнйий слайд Шаповал М.Р

Об’єкт дослідження – процес візуалізації даних моніторингу у консольних застосунках на різних програмних платформах.

Предмет дослідження – методи та програмні засоби кросплатформного графічного виводу у консольному вікні без використання зовнішніх графічних фреймворків.

Мета роботи – зменшення часу та ресурсних витрат на впровадження візуалізації даних моніторингу у консольних застосунках шляхом створення кросплатформної системи з уніфікованим програмним інтерфейсом для платформ Windows та Linux.

2

Рисунок Б.2 – Слайд мети роботи

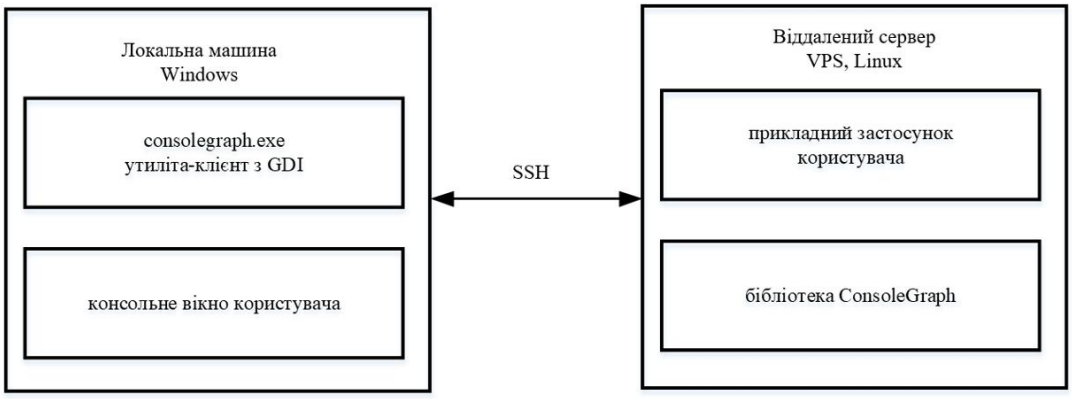
Порівняння методів графічного виводу

Критерій	Windows GDI	ANSI escape-послідовності
Цільова платформа	Microsoft Windows	Linux, macOS, BSD, Windows
Тип виводу	Піксельна графіка	Символьна псевдографіка
Якість зображення	Висока, обмежена розміром вікна у пікселях	Дискретна, обмежена розміром комірки терміналу
Згладжування шрифтів	Підтримується через ClearType	Залежить від налаштувань емулятора
Кольорова модель	RGB 24-bit	RGB 24-bit (truecolor)
Робота у віддаленій сесії SSH	Неможлива	Повна підтримка
Сумісність з Windows Terminal	Обмежена	Повна
Залежність від кодування	Відсутня	Потребує UTF-8
Залежність від системного API	Win32 API	Стандартні потоки введення-виведення

3

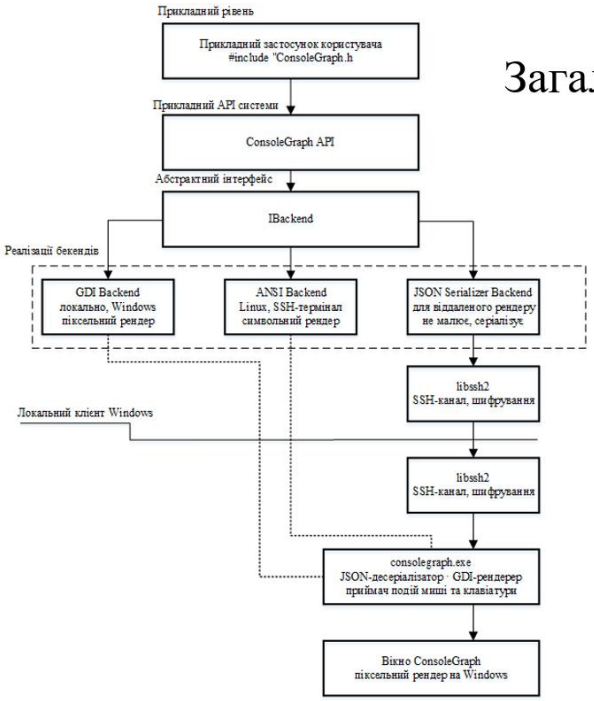
Рисунок Б.3 – Порівняння методів графічного виводу

Загальна структура системи



4

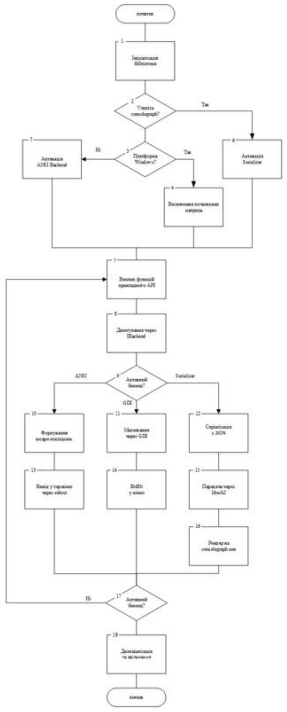
Рисунок Б.4 – Загальна структура системи



Загальна архітектура системи

5

Рисунок Б.5 – Загальна архітектура системи



Алгоритм роботи системи

6

Рисунок Б.6 – Алгоритм роботи системи

UML-діаграма класів бібліотеки

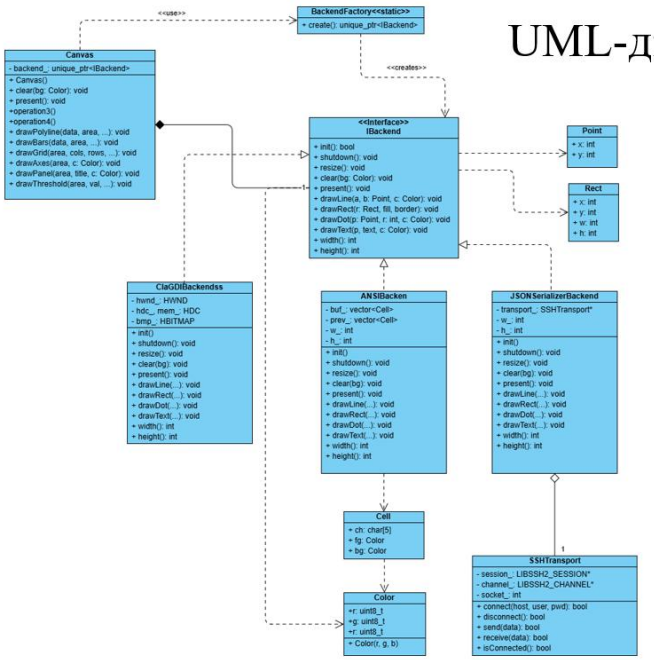


Рисунок Б.7 – UML-діаграма класів бібліотеки

UML-діаграма класів утиліти

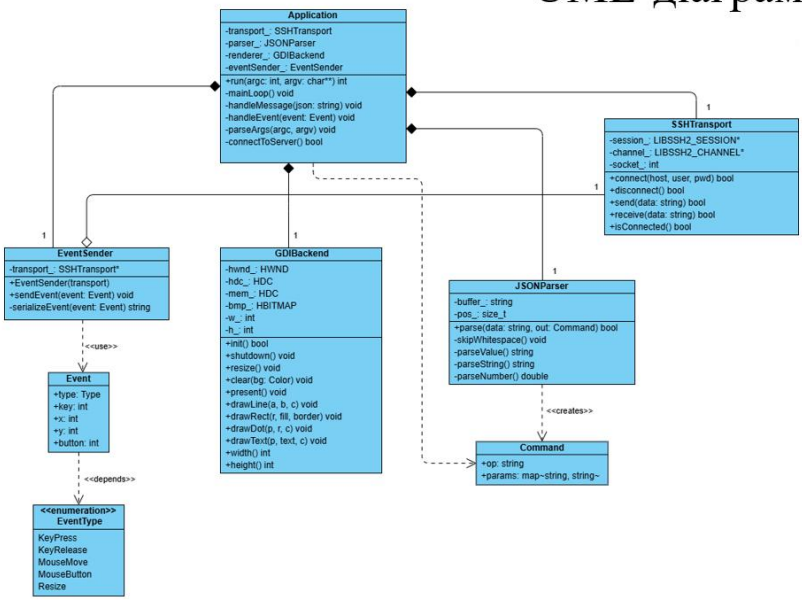
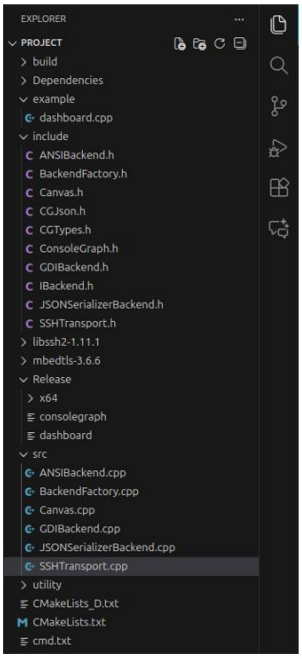


Рисунок Б.8 – UML-діаграма класів утиліти



Структура каталогів проєкту

9

Рисунок Б.9 – Структура каталогів проєкту

Результат складання системи на платформі Windows

```
162     bool SSHTransport::receive(std::string &data)
163     {
164         if (!channel_)
165             return false;
166         char buf[4096];
```

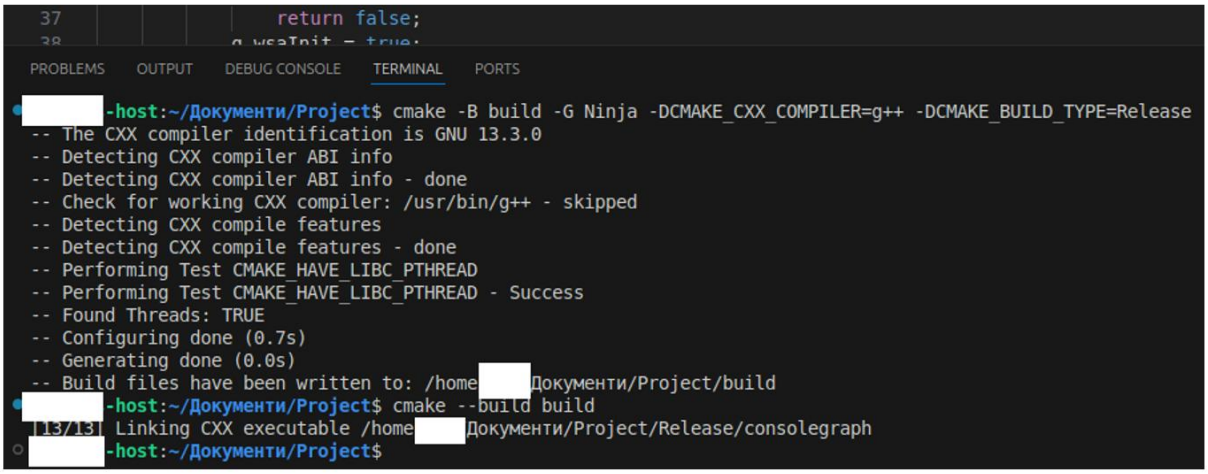
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS F:\DIPLOM\Project> cmake -B build -G Ninja -DCMAKE_CXX_COMPILER=g++ -DCMAKE_BUILD_TYPE=Release
-- The CXX compiler identification is GNU 13.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: E:/mingw64/bin/g++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.9s)
-- Generating done (0.0s)
-- Build files have been written to: F:/DIPLOM/Project/build
PS F:\DIPLOM\Project> cmake --build build
[13/13] Linking CXX executable F:\DIPLOM\Project\Release\dashboard.exe
PS F:\DIPLOM\Project>
```

10

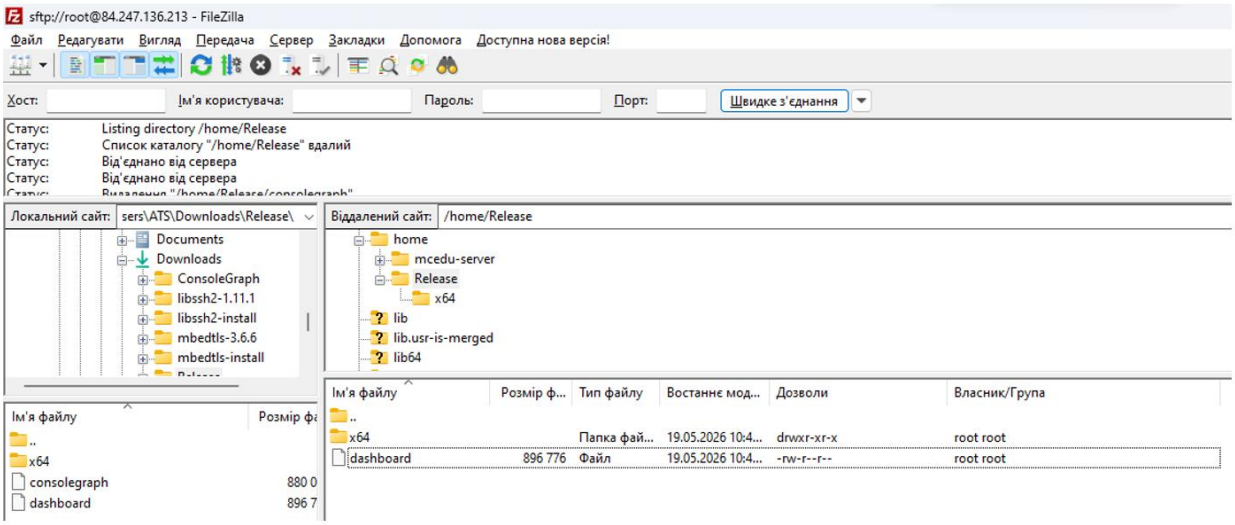
Рисунок Б.10 – Результат складання системи на платформі Windows

Результат складання системи на платформі Linux



11

Рисунок Б.11 – Результат складання системи на платформі Linux



12

Рисунок Б.12 – Розташування застосунку на VPS

13

Рисунок Б.13 – Підключення до сервера VPS

14

Рисунок Б.14 – Підключення до сервера VPS через утиліту

ВИСНОВКИ

Розроблено кросплатформну систему ConsoleGraph для візуалізації даних моніторингу у консольних застосунках Windows та Linux.

Реалізовано:
-Уніфікований програмний інтерфейс мовою C++
-Три бекенди виводу: GDI (Windows), ANSI (Linux), JSON Serializer (віддалений рендер)
-Утиліту-клієнт consolegraph для роботи з віддаленими серверами через SSH
-Власний механізм автоматичної збірки залежностей під цільову платформу

Підтверджено:
-Один і той самий код працює на Windows і Linux без змін
-Самодостатність системи без зовнішніх залежностей
-Працездатність на реальному VPS-сервері через мережу Інтернет

15

Рисунок Б.15 – Висновки