МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

О.М. Цимбал, А.І. Бронніков

ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ:
C++/CLI в Microsoft Visual Studio

Рекомендовано Вченою радою
Харківського національного університету радіоелектроніки
як навчальний посібник для студентів
спеціальності G7 Автоматизація, комп'ютерно-інтегровані
технології та робототехніка

Харків 2025

УДК 004.4`2

Рецензенти:

В.В. Іващук, д-р техн. наук, доцент (Національний університет біоресурсів і природокористування України);

О.А. Янковський, канд. техн. наук, доцент (Харківський національний університет радіоелектроніки);

М.Ю. Родінко, канд. техн. наук, доцент (Харківський національний університет ім. В.Н. Каразіна).

Рекомендовано Вченою радою
Харківського національного університету радіоелектроніки
(протокол № 7/10 від 29.05.2025 р.)

Цимбал О.М., Бронніков А.І Технології програмування: C++/CLI в Microsoft Visual Studio. – Харків: Видавництво Іванченка І.С., 2025. – 275 с.

ISBN 978-617-8332-86-0
DOI: 10.30837/978-617-8332-86-0

Викладено основи розробки програмного забезпечення у середовищі Microsoft Visual Studio засобами мови програмування C++/CLI для платформи .NET. Розглядається побудова Windows-орієнтованих програм .NET, елементи побудови багатопотокових програм та програм обробки баз даних. Окремо розглядається розробка програм використання тривимірної графіки на основі бібліотеки OpenGL. Теоретичний матеріал супроводжено великою кількістю прикладів практичного застосування технологій програмування.

Рекомендується студентам закладів вищої освіти, які навчаються за спеціальністю G7 «Автоматизація, комп'ютерно-інтегровані технології та робототехінка», також усім зацікавленим у розширенні знань з програмування.

Іл.: 163. Табл.: 13. Бібліогр. наймен.: 14.

УДК 004.4`2

ISBN 978-617-8332-86-0
DOI: 10.30837/978-617-8332-86-0

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

БД	–	база даних
ЕОМ	–	Електронно-обчислювальна машина
ЕК	–	елемент керування
ПЗ	–	програмне забезпечення
ШІ	–	штучний інтелект
СК	–	система координат
API	–	Application programming interface (набір системних функцій програмування інтерфейсу Windows)
CLI	–	Common Language Interface (інтерфейс спільної мови .NET)
CLR	–	Common Language Runtime (набір бібліотек спільної мови .NET)
MFC	–	Microsoft Foundation Classes (базові класи Microsoft)
MSDN	–	Microsoft Developer Network (система документації Microsoft Visual Studio)
OpenGL	–	Open Graphic Library (відкрита графічна бібліотека)

ВСТУП

Здається нещодавно (але минуло вже більше 15 років), як вийшов попередній навчальний посібник «Технології програмування: Visual C++» [1]. З того часу багато чого змінилося і у галузі розробки програмного забезпечення.

Перше за все, окрім персональних та промислових комп'ютерів з'явилося чимало нових типів пристроїв, зокрема смартфони та планшети (таблети), а багато пристроїв, зокрема телевізори, пральні машини, наручні годинники або автомобільні радіоприймачі набули якостей, раніше притаманних лише комп'ютерній техніці. Таким чином коло споживачів такого роду складної побутової або комунікаційної (або іншої) техніки суттєво зросло.

По-друге, з точки зору програмного забезпечення персональних комп'ютерів за цей час відбувся суттєвий зсув платформ цієї розробки: панівна роль Microsoft Windows зникає, натомість зростає кількість робочих станцій на базі Linux та MacOS систем.

По-третє, з точки зору розробника програмного забезпечення можна бачити скорочення кількості розробок мовою C++, обумовлене неймовірним зростом попиту на Інтернет-прикладки, створювані за допомогою достатньо нових мов програмування: C#, Jawa, PHP, Python та інших. У інтернет-конференціях 2023 року досить серйозно дискутується питання «Is C++ dead?» (чи мертвий C++?). Напевно, що прихильники C++ сумуватимуть з такого формулювання, хоча і C# і Jawa можуть вважатися спадкоємцями мов C та C++. Не додають оптимізму і рекомендації розробників не використати бібліотеки C++ для створення програмного забезпечення мобільних пристроїв на базі операційної системи Android.

Проте на таком темному тлі подій навколо мови програмування C++ є і світлі прогалини. Однією з них є поява (вже досить давно – з 2003 року) платформи .NET і підтримка програмування для цієї платформи різновидом C++ під назвою C++/CLI. Звичайно, критики використання мови програмування C++ можуть зауважити, що C# є головним засобом розробки для програмного забезпечення платформи .NET і реінкарнація C++ для завдань програмування в ОС Microsoft Windows є хибним напрямом. Можливо, це і вірно, проте частково – додаткова можливість використання C++ надає можливість використання усіх можливостей платформи .NET, крім того – інтегрування програмного забезпечення для роботи зі стандартними пристроями Windows-комп'ютерів, іншими

пристроями інтегрованих систем, програмування яких, в основному ведеться із залученням C++ .

В основу навчального посібника покладено лекції з дисципліни «Технології програмування комп'ютеризованих систем», прочитані авторами для студентів освітньої програми «Системна інженерія» спеціальності «Автоматизація та комп'ютерно-інтегровані технології» у Харківському національному університеті радіоелектроніки та лекції з дисципліни «System software» (системне програмне забезпечення), прочитані у тому ж університеті для іноземних студентів спеціальності «Комп'ютерна інженерія» у 2012-2025 рр [3].

1 ПЛАТФОРМА .NET, ОСОБЛИВОСТІ ТА ПРИЗНАЧЕННЯ СКЛАДОВИХ

1.1 Платформа .NET та її особливості

Microsoft розробила .NET як безкоштовну та кросплатформну екосистему розробки з відкритим кодом. Її мета – надати інструменти, бібліотеки та середовища виконання, які дозволяють розробникам ефективно створювати різноманітні застосунки, від веб- та мобільних до настільних, хмарних, IoT та ігрових рішень.

Її особливості можуть бути представлені у вигляді схеми (рис. 1.1).

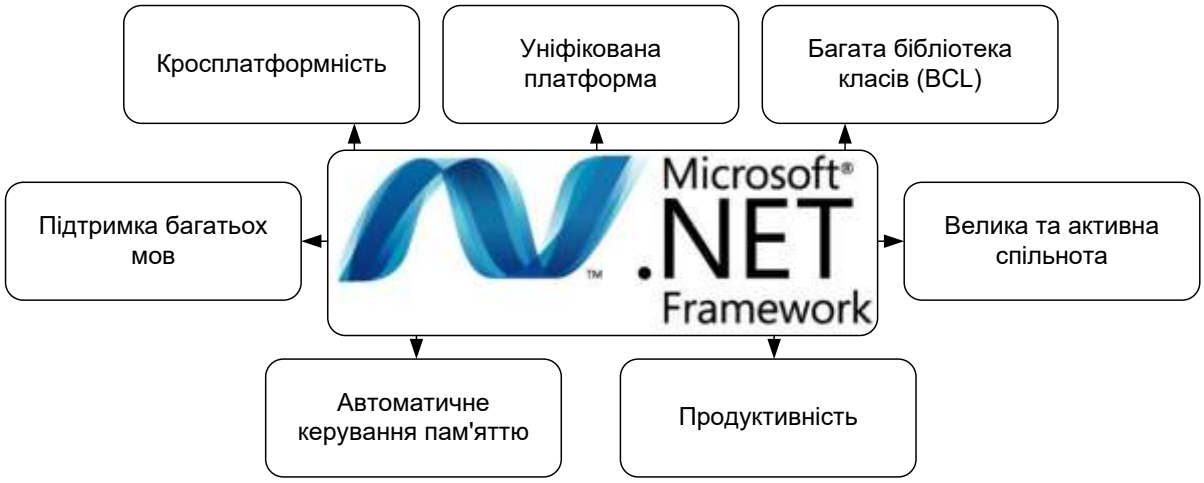


Рисунок 1.1 – Особливості платформи .NET

Розглянемо кожну з цих особливостей.

Першою є кросплатформність. Завдяки кросплатформності .NET, створені на ній програми здатні працювати на різних операційних системах, таких як Windows, macOS і Linux, майже без змін у коді. Це стає можливим завдяки .NET Runtime (особливо CoreCLR у сучасній версії), що приховує відмінності між цими системами. Таким чином, розробники можуть писати код одноразово та використовувати його на багатьох платформах, що полегшує розробку та збільшує потенну кількість користувачів.

Другою є уніфікованість платформи. Microsoft намагається створити в .NET єдину основу для розробки різноманітних застосунків, таких як веб, мобільні, настільні, хмарні та ігри. Замість різних інструментів для кожної потреби, .NET пропонує узгоджений набір технологій. Наприклад, ASP.NET Core застосовується для вебу та API, .NET MAUI – для кросплатформних мобільних і на-

стільних програм, а Unity – для ігор на C#. Це спрощує перехід між різними проєктами, оскільки розробники можуть використовувати вже знайомі підходи та інструменти, що робить процес навчання легшим та підвищує продуктивність.

Третьою є багата бібліотека класів. У .NET існує велика Базова Бібліотека Класів (BCL) – це як величезна колекція вже готових фрагментів коду, які допомагають розробникам швидко створювати програми, не пишучи все з самого початку.

BCL містить інструменти для багатьох поширених задач, таких як:

- опрацювання даних (колекції, файли, потоки, XML, JSON та інше);
- робота з мережею (протоколи HTTP, TCP/IP тощо);
- забезпечення безпеки (шифрування, хешування);
- виконання кількох завдань одночасно;
- аналіз структури коду під час роботи програми.

Завдяки BCL розробка стає швидшою та надійнішою, адже багато рутинних операцій вже реалізовано та протестовано, що дозволяє програмістам зосередитися на головній меті їхнього застосунку.

Четвертою особливістю є підтримка багатьох мов програмування. Однією з важливих рис .NET є те, що Common Language Runtime (CLR) може запускати код, написаний різними мовами програмування. Це можливо тому, що такі мови, як C#, F# та VB.NET, спочатку перетворюються на проміжну мову (IL), яку потім виконує CLR.

Такий підхід дає розробникам свободу вибирати мову, яка найкраще відповідає їхнім потребам або вподобанням, і навіть об'єднувати код, написаний на різних .NET-мовах, в одному проєкті. Наприклад, для об'єктно-орієнтованих завдань можна використовувати C#, а для функціональних – F#.

До основних мов .NET належать:

- C#: найпоширеніша мова в .NET, підтримує різні стилі програмування;
- C++: також як і C#, підтримує різні стилі програмування;
- F#: функціональна мова зі строгим контролем типів, але також підтримує інші підходи;
- VB.NET: сучасна версія Visual Basic, орієнтована на об'єктно-орієнтоване програмування.

Розробники мають більшу свободу у виборі інструментів для створення застосунків на платформі .NET завдяки підтримці багатьох мов програмування, що є дуже важливим у наш час.

П’ятою є автоматичне керування пам’яттю. У .NET пам'ять керується автоматично за допомогою збирача сміття (Garbage collector, GC). Його головне завдання – звільняти пам'ять, яку програма більше не використовує.

Процес виглядає наступним чином:

– коли .NET-застосунок створює об'єкти, CLR виділяє їм місце в керованій пам'яті;

– GC періодично аналізує, які об'єкти в цій пам'яті ще потрібні програмі (на них є посилання), а які вже ні;

– непотрібні об'єкти вважаються "сміттям", і GC звільняє займану ними пам'ять для подальшого використання;

Переваги такого підходу:

– розробникам не потрібно вручну піклуватися про виділення та звільнення пам'яті, що спрощує написання коду та зменшує ймовірність помилок, пов'язаних з пам'яттю;

– застосунки стають стабільнішими та рідше аварійно завершуються через проблеми з пам'яттю.

Хоча GC працює самостійно, розробники можуть давати йому деякі вказівки, але пряме керування зазвичай не потрібне.

Шостою є продуктивність. Одним із важливих напрямків розвитку .NET є забезпечення високої продуктивності застосунків. Цьому сприяє ряд технологій та оптимізацій:

- JIT-компіляція: код проміжною мовою (IL) перетворюється на машинний код безпосередньо перед запуском, що дозволяє оптимізувати його під конкретний процесор;
- AOT-компіляція: у деяких випадках код компілюється в машинний код ще на етапі збірки, що може прискорити запуск програми;
- ефективний збирач сміття (GC): постійно вдосконалюється для зменшення затримок під час звільнення пам'яті, що критично для чутливих до швидкості застосунків;
- внутрішні оптимізації: бібліотеки BCL та CLR містять багато низькорівневих покращень для збільшення швидкодії, включаючи ефективні алгоритми та використання можливостей процесора;
- Span<T> та Memory<T>: забезпечують швидкий та безпечний доступ до послідовних ділянок пам'яті без зайвого копіювання;

– SIMD: підтримка векторних операцій дозволяє одночасно виконувати одну й ту саму дію над багатьма даними, що значно прискорює певні обчислення.

Завдяки цим та іншим покращенням, .NET є продуктивною платформою для розробки різноманітних застосунків, від потужних серверних систем до вимогливих клієнтських програм.

Останньою і однією з найбільших переваг .NET є її велика та діяльна спільнота. Це всесвітнє об'єднання програмістів, шанувальників та компаній, які використовують, підтримують і розвивають цю платформу.

Особливостями є:

– велика кількість учасників: завдяки популярності .NET, до неї входять мільйони розробників з усього світу;

– активна взаємодія: учасники охоче діляться знаннями, допомагають один одному, створюють додаткові бібліотеки, пишуть статті та організовують зустрічі;

– відкритість: оскільки .NET має відкритий код, спільнота може впливати на її розвиток, пропонувати зміни та брати участь у розробці;

– багато навчальних матеріалів: існує безліч посібників, документації, прикладів коду, блогів, форумів (наприклад, Stack Overflow) та відеоуроків, створених спільнотою;

– організованість: спільнота проводить багато місцевих та міжнародних зустрічей, конференцій (наприклад, .NET Conf), що сприяє обміну досвідом та налагодженню зв'язків.

1.2 Складові платформи .NET

Платформа .NET складається з кількох ключових складових, кожна з яких виконує свою певну роль. На рис. 1.2 можна побачити схему складових платформи.

1.2.1 Common Language Runtime (CLR)

Першою складовою є Common Language Runtime (CLR) – це основа платформи .NET, свого роду віртуальна машина, яка керує роботою .NET-програм. Вона приховує відмінності між операційними системами та обладнанням, створюючи кероване середовище для застосунків .NET.

Основні функції CLR включають:

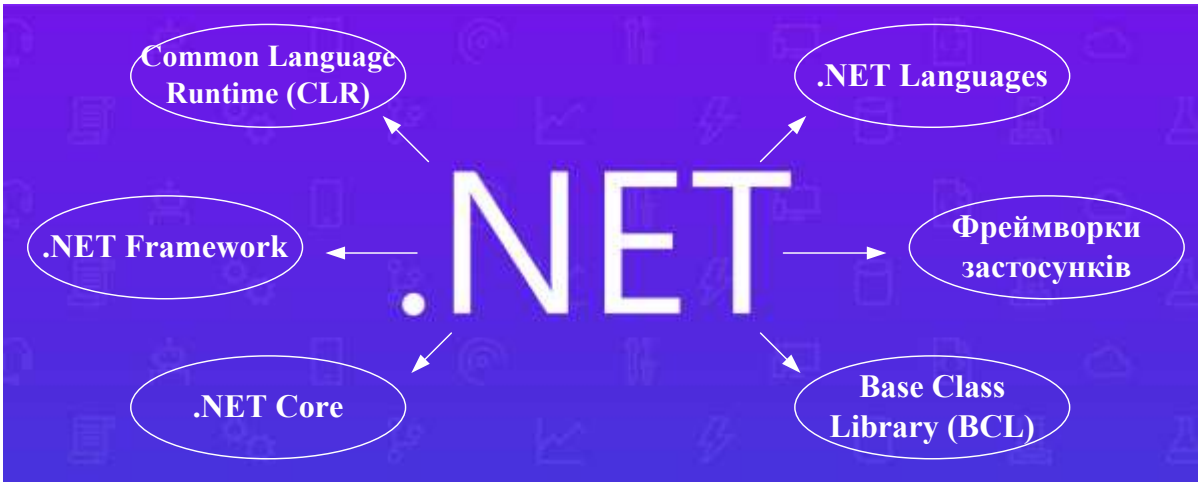


Рисунок 1.2 – Складові платформи .NET

– завантаження та виконання коду – CLR знаходить, завантажує та компілює .NET-асемблеї, часто використовуючи JIT-компіляцію (перетворення IL на машинний код перед першим запуском методу з можливістю кешування). Іноді застосовується AOT-компіляція (компіляція в машинний код під час збірки);

– керування пам'яттю (GC) – CLR автоматично виділяє та звільняє пам'ять за допомогою збирача сміття, що спрощує розробку, усуваючи необхідність ручного керування пам'яттю та зменшуючи ризик помилок;

– безпека типів – CLR перевіряє коректність операцій над даними відповідно до їхніх типів, підвищуючи надійність програм;

– обробка винятків – CLR надає механізм для обробки помилок під час виконання, дозволяючи програмістам коректно реагувати на непередбачувані ситуації;

– управління потоками – CLR дає змогу створювати та керувати потоками для розробки продуктивних та чуйних багатозадачних застосунків;

– взаємодія з ОС – CLR абстрагує низькорівневі деталі ОС, але також дозволяє .NET-коду звертатися до системних функцій при необхідності.

Слід зазначити, CLR – це посередник між .NET-кодом та операційною системою, який бере на себе управління виконанням програми, пам'яттю, безпекою та обробкою помилок. Також існують дві основні реалізації CLR: оригінальна .NET Framework CLR (лише для Windows) та сучасна кросплатформна CoreCLR (основа .NET).

1.2.2 .NET Framework

.NET Framework був початковою платформою Microsoft для розробки застосунків під Windows, започаткованою у 2002 році. Він став основою багатьох

технологій Microsoft і важливим інструментом для розробників, що створювали рішення саме для цієї операційної системи.

Основні риси .NET Framework:

– орієнтація на Windows: на відміну від сучасного .NET, Framework розроблявся насамперед для Windows і тісно з нею інтегрований;

– велика бібліотека класів (BCL): містить безліч готових бібліотек для різних завдань розробки;

– підтримка різних підходів: Framework підтримує WinForms (традиційні настільні програми), WPF (сучасні настільні програми з XAML), ASP.NET (веб-застосунки та сервіси), ADO.NET (робота з даними) та WCF (сервіс-орієнтовані застосунки);

– CLR: як і сучасний .NET, Framework використовує CLR для виконання коду, безпеки та керування пам'яттю;

– зріла екосистема: за роки існування Framework сформувалася велика спільнота, з'явилося багато інструментів та сторонніх бібліотек.

Призначення .NET Framework полягало у спрощенні розробки під Windows, надаючи стандартизоване та кероване середовище для створення надійних, безпечних та швидких програм.

На сьогодні Microsoft зосередилася на розвитку кросплатформного .NET (раніше .NET Core). Версія 4.8 є останньою основною для Framework, і нових великих оновлень не планується, хоча підтримка безпеки та виправлення помилок триватиме. Для нових проєктів, особливо з потребою у кросплатформності, рекомендується використовувати сучасний .NET.

1.2.3 .NET Core (тепер просто .NET)

Сучасний .NET, що об'єднав у собі попередні .NET Framework та .NET Core, є безкоштовною, кросплатформною екосистемою розробки з відкритим кодом від Microsoft. Його створено як наступника .NET Framework з акцентом на роботу на різних операційних системах, високу продуктивність та відповідність сучасним вимогам розробки.

Основні особливості .NET:

– кросплатформність: застосунки .NET можуть працювати на Windows, macOS та Linux майже без змін у коді завдяки оновленій .NET Runtime (CoreCLR);

– висока продуктивність: платформа розробляється з фокусом на швидкодію, включаючи оптимізований CLR, ефективні бібліотеки та підтримку сучасних технік;

– модульність: архітектура .NET дозволяє підключати лише необхідні пакети NuGet, що зменшує розмір застосунку та підвищує його ефективність;

– сучасні підходи до розробки: .NET підтримує ASP.NET Core для вебзастосунків та API, .NET MAUI для кросплатформних мобільних і настільних програм, Blazor для інтерактивних веб-інтерфейсів на C#, а також хмарні технології, контейнеризацію та мікросервіси;

– відкритий код: .NET має відкритий вихідний код на GitHub, що сприяє прозорості та залученню спільноти до його розвитку;

– уніфікована платформа: .NET спрямований на об'єднання розробки різних типів застосунків.

Основне призначення .NET – надати сучасну, продуктивну та кросплатформну основу для створення широкого спектра застосунків, включаючи вебсайти, API, мобільні та настільні програми, хмарні сервіси та ігри (через інтеграцію з Unity).

Еволюційно, після існування окремих .NET Framework (для Windows) та .NET Core (кросплатформний), Microsoft об'єднала їх у єдину платформу під назвою .NET, починаючи з версії 5. Подальші версії (6, 7, 8 тощо) продовжують цю модель. Тому, говорячи про .NET сьогодні, зазвичай мається на увазі саме ця сучасна, кросплатформна реалізація.

1.2.4 Мови програмування (.NET Languages)

Однією з ключових особливостей .NET Common Language Runtime (CLR) є його здатність виконувати код, написаний різними мовами програмування. Це досягається завдяки тому, що ці мови компілюються в проміжну мову (Intermediate Language, IL), яку потім виконує CLR.

Основні мови, які широко використовуються в екосистемі .NET, включають:

– C# (Сі-шарп): потужна та універсальна об'єктно-орієнтована мова від Microsoft, яка також підтримує імперативне та функціональне програмування. C# є основною мовою .NET, що поєднує продуктивність C++ зі зручністю Visual Basic. Вона має строгу типізацію, елегантний синтаксис та багатий набір можливостей, таких як LINQ, асинхронне програмування, узагальнення тощо. C# застосовується для розробки майже всіх типів .NET-застосунків: веб-, настільних, мобільних, ігрових, хмарних, IoT та інших;

– F# (Еф-шарп): функціональна мова зі сильною статичною типізацією, що також підтримує об'єктно-орієнтовані та імперативні підходи. F# відрізняється лаконічністю, ефективністю в обробці даних та паралельному програму-

ванні. Часто використовується у фінансовій, аналітичній та науковій сферах. Застосування включає обробку даних, машинне навчання, веб-розробку (через Fable) та створення надійного паралельного коду;

– VB.NET (Віжуал Бейсік .NET): об'єктно-орієнтована мова, що є еволюцією класичного Visual Basic. Має більш простий синтаксис, знайомий розробникам VB, і підтримує багато сучасних можливостей .NET. VB.NET часто використовується для розробки Windows-застосунків (WinForms, WPF) та веб-проєктів (ASP.NET). Хоча C# є більш популярним для нових розробок, VB.NET залишається важливим для підтримки існуючих систем;

– C++/CLI: це набір розширень мови C++, розроблений Microsoft для полегшення взаємодії між керованим (.NET) та некерованим (традиційним C++) кодом. C++/CLI виступає як міст, дозволяючи .NET-застосункам використовувати існуючі C++ бібліотеки та навпаки. Це особливо корисно при роботі з низькорівневими системними API, застарілим кодом або бібліотеками, написаними на C++. C++/CLI дає змогу створювати обгортки (wrappers) навколо некерованого коду, роблячи його доступним для використання в .NET-середовищі, зберігаючи при цьому продуктивність C++ там, де це необхідно.

Крім цих основних мов, існують й інші, що компілюються в IL, такі як IronPython та IronRuby (реалізації Python та Ruby для .NET).

Вибір мови для розробки на .NET залежить від специфіки проєкту, навичок команди та переваг у парадигмі програмування. C# є найбільш універсальним, F# відмінно підходить для функціонального програмування та обробки даних, VB.NET може бути зручним для тих, хто знайомий з Visual Basic, а C++/CLI незамінний при необхідності інтеграції з некерованим C++ кодом.

1.2.5 Фреймворки застосунків (Application Frameworks)

У .NET існує безліч фреймворків, кожен з яких спеціалізується на розробці певного типу застосунків, надаючи готові інструменти, бібліотеки та шаблони для швидкого та ефективного створення рішень.

Ось деякі з найважливіших фреймворків застосунків у .NET:

– ASP.NET Core: призначений для розробки сучасних веб-застосунків, веб-API, мікросервісів та хмарних рішень. Він є кросплатформним, високопродуктивним, має модульну архітектуру та підтримує MVC і Razor Pages, вбудовані Dependency Injection та Identity, а також SignalR для real-time комунікації. ASP.NET Core є основою для сучасних веб-рішень на .NET;

– .NET MAUI (Multi-platform App UI): використовується для створення кросплатформних настільних (Windows, macOS) та мобільних (Android, iOS) за-

стосунків з єдиної кодової бази на C# та XAML. Він є еволюцією Xamarin.Forms, дозволяючи розробляти UI один раз для різних платформ, зберігаючи доступ до їхніх специфічних можливостей, та підтримує різноманітні макети, елементи керування, навігацію та роботу з даними;

– Windows Presentation Foundation (WPF): створений для розробки багатих та візуально привабливих настільних застосунків під Windows з використанням XAML для інтерфейсу та C# для логіки. Він має потужну систему прив'язки даних, підтримку векторної графіки, анімації, стилів та шаблонів, що дозволяє створювати складні та кастомізовані інтерфейси.

– Windows Forms (WinForms): фреймворк для швидкої розробки традиційних настільних застосунків з графічним інтерфейсом для Windows. Він базується на подіях та візуальних компонентах, які розміщуються на формах. WinForms відрізняється простотою у вивченні та великою бібліотекою готових візуальних елементів керування, що робить його зручним для швидкого створення інтерфейсів користувача для Windows. Хоча WinForms вважається більш зрілою технологією порівняно з WPF, він досі широко використовується для розробки внутрішніх корпоративних застосунків, утиліт та програм, де швидкість розробки та знайомий інтерфейс є пріоритетними;

– Entity Framework Core (EF Core): є об'єктно-реляційним відображенням (ORM) для .NET, що спрощує роботу з базами даних, дозволяючи оперувати даними через об'єкти C# замість SQL. Він підтримує різні типи баз даних, міграції для керування схемою, LINQ для запитів та відстеження змін об'єктів, і є рекомендованим ORM для нових .NET-проєктів;

– ML.NET: фреймворк машинного навчання для .NET, що дозволяє інтегрувати ML-можливості у .NET-застосунки. Він підтримує різні завдання машинного навчання, надає інструменти для підготовки даних, навчання моделей та їх використання.

– Xamarin.Forms (частина .NET MAUI): попередник .NET MAUI, призначений для створення кросплатформних мобільних застосунків (Android, iOS) на C# та XAML. Він багато в чому схожий на .NET MAUI, але орієнтований лише на мобільні платформи, тому для нових проєктів рекомендується використовувати .NET MAUI.

1.3 Windows Forms

Windows Forms (.NET), або скорочено WinForms, є фреймворком графічного інтерфейсу користувача (GUI), що входить до складу .NET. Він використовується для створення традиційних настільних програм, орієнтованих на опера-

ційну систему Windows (рис. 1.3). Більш докладно про Windows Forms можна дізнатися з розділу 3 навчального посібника.

WinForms надає розробникам набір готових візуальних компонентів, таких як кнопки, текстові поля, мітки та списки, які можна розміщувати на формах і програмувати їхню поведінку у відповідь на дії користувача, наприклад, кліки миші або введення тексту.

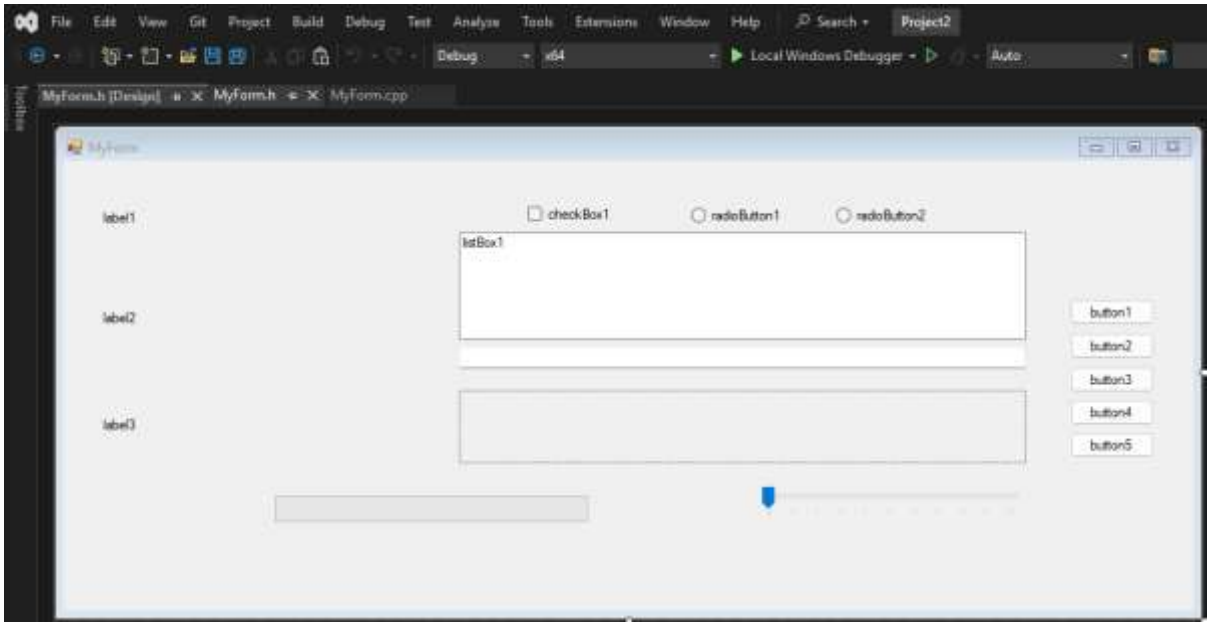


Рисунок 1.3 – Приклад Windows Forms

Основні характеристики WinForms:

– візуальний конструктор: Visual Studio пропонує зручний візуальний дизайнер для WinForms, що дозволяє розробникам перетягувати елементи керування на форму та налаштовувати їхні властивості візуально, без написання коду;

– проста модель обробки подій: взаємодія користувача з елементами інтерфейсу обробляється через події (наприклад, подія "Click" для кнопки). Розробники пишуть обробники подій, щоб визначити реакцію програми на ці дії;

– багатий набір готових компонентів: WinForms містить велику кількість вбудованих елементів керування, що задовольняють більшість стандартних потреб настільних застосунків;

– тісна інтеграція з Windows API: WinForms використовує багато низькорівневих API Windows, забезпечуючи доступ до можливостей операційної системи.

Хоча з появою WPF, яка пропонує сучасніший підхід до створення інтерфейсів з кращою графікою та стилізацією, популярність WinForms дещо зменшилася, він залишається актуальним для таких випадків:

– швидка розробка внутрішніх корпоративних інструментів завдяки простоті використання та візуальному дизайнеру.

– підтримка та оновлення існуючих застосунків, написаних на winforms.

– створення простих застосунків без високих вимог до графіки.

Перевагами використання WinForms є наступні:

– легкість у вивченні та використанні, особливо для початківців або розробників зі знайомством з попередніми GUI-технологіями Windows;

– швидке створення базових інтерфейсів завдяки візуальному дизайнеру;

– велика кількість готових UI-компонентів, що позбавляє від необхідності їхнього самостійного створення;

– нижчі вимоги до системних ресурсів порівняно з WPF;

– зріла технологія з великою кількістю документації та прикладів.

Недоліками WinForms є:

– обмежені можливості кастомізації зовнішнього вигляду компонентів, часто залежні від стандартних стилів Windows;

– складність реалізації сучасних UI/UX патернів, таких як складні анімації та адаптивні макети;

– залежність від операційної системи Windows;

– за замовчуванням менш сучасний зовнішній вигляд порівняно з WPF та іншими сучасними фреймворками;

– гірша підтримка векторної графіки та масштабування на екранах з високою роздільною здатністю порівняно з WPF.

1.4 Висновки по використанню технології .NET

.NET – це потужна та різностороння платформа розробки, яка надає значні переваги для створення найрізноманітніших застосунків. Її здатність працювати на різних операційних системах (Windows, macOS, Linux) розширює охоплення аудиторії. Завдяки уніфікованому підходу, розробники можуть використовувати знайомі інструменти та концепції для різних типів завдань.

Велика бібліотека класів (BCL) суттєво прискорює розробку, надаючи готові рішення для багатьох стандартних операцій. Підтримка кількох мов програмування (C#, F#, VB.NET) забезпечує гнучкість у виборі інструментів, а ав-

томатичне керування пам'яттю робить розробку простішою та підвищує надійність.

.NET демонструє високу продуктивність завдяки таким технологіям, як JIT та AOT компіляція, ефективний збирач сміття та інші оптимізації. Велика та активна спільнота сприяє підтримці, обміну знаннями та постійному розвитку платформи.

Проте, при виборі .NET варто враховувати деякі моменти. Для розробки виключно під Windows .NET Framework залишається актуальним для підтримки наявних проєктів, хоча для нових рекомендується сучасний .NET. Щодо створення графічного інтерфейсу, Windows Forms є простим варіантом для швидкої розробки настільних Windows-застосунків, але може поступатися WPF або кросплатформному .NET MAUI в можливостях кастомізації та підтримці сучасних UI/UX патернів.

Загалом, використання .NET є виправданим для широкого кола проєктів завдяки його кросплатформності, продуктивності, зручності розробки, підтримці сучасних технологій та активній спільноті. Вибір конкретних інструментів всередині .NET (наприклад, ASP.NET Core, .NET MAUI, WPF, WinForms) залежатиме від потреб конкретного проєкту та цільових платформ.

2 ОСОБЛИВОСТІ ІНТЕРФЕЙСУ КОРИСТУВАЧА MICROSOFT VISUAL STUDIO

2.1 Встановлення компонентів Microsoft Visual Studio

Microsoft Visual Studio є інтегрованим середовищем розробника програмного забезпечення, що підтримується компанією Microsoft з моменту появи у 1997 році продукта Visual Studio 97. Серед перших розробок, найбільш популярним був Visual Studio 6.0, який характерно для того часу постачався у пакунку 3,5” дискет.

Сучасний Microsoft Visual Studio, наприклад Visual Studio 2019, підтримує створення багатьох типів проектів декількома мовами програмування, це можна бачити з Рисунка 2.1, що показує вікно Visual Studio Installer.

Встановлення усіх компонентів Microsoft Visual Studio займає десятки гігабайт і, в основному, є зайвим. Слід додавати лише ті компоненти, які є дійсно потрібними для розробки.

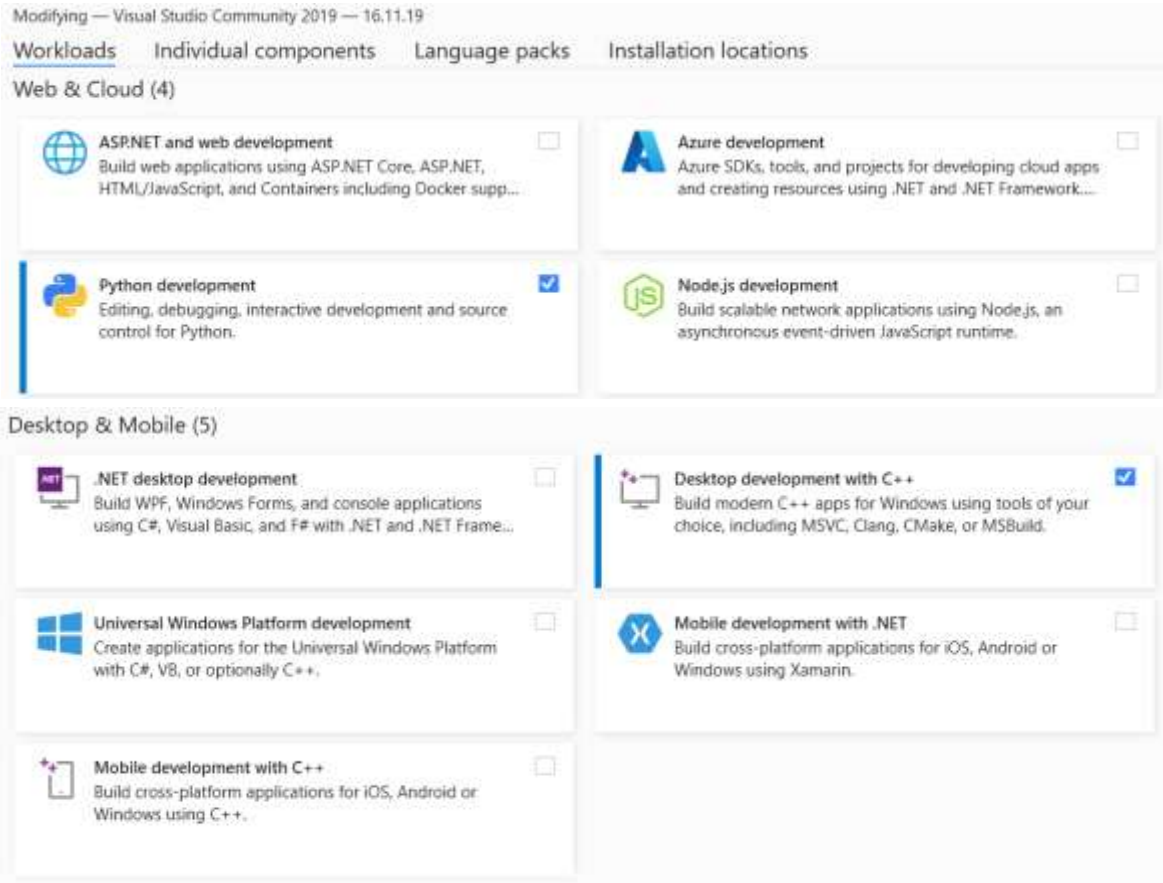


Рисунок 2.1а – Вигляд вікна Visual Studio Installer (верхня частина).
Обрано і встановлено “Python development” та “Desktop development with C++”
(для розробки Python-програм, C++/CLI та MFC-проектів)

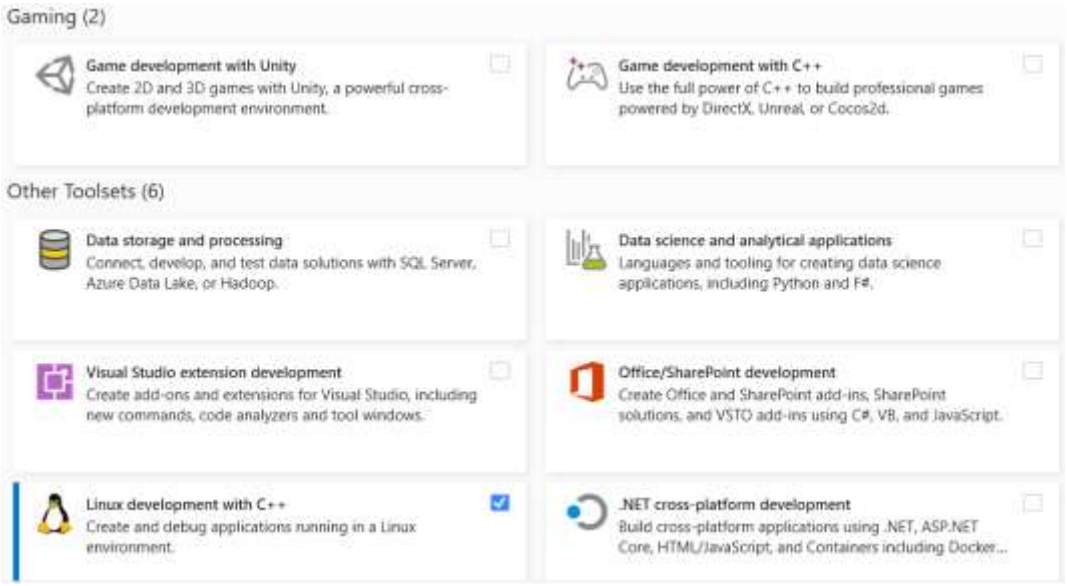


Рисунок 2.1б – Вигляд вікна Visual Studio Installer (нижня частина). Обрано і встановлено “Linux development with C++” (для RaspBerry PI проектів)

Для роботи з окремими типами проектів, зокрема C++/CLI та MFC, необхідні і більш детальні налаштування. Вибір необхідних опцій показано на рис. 2.2.

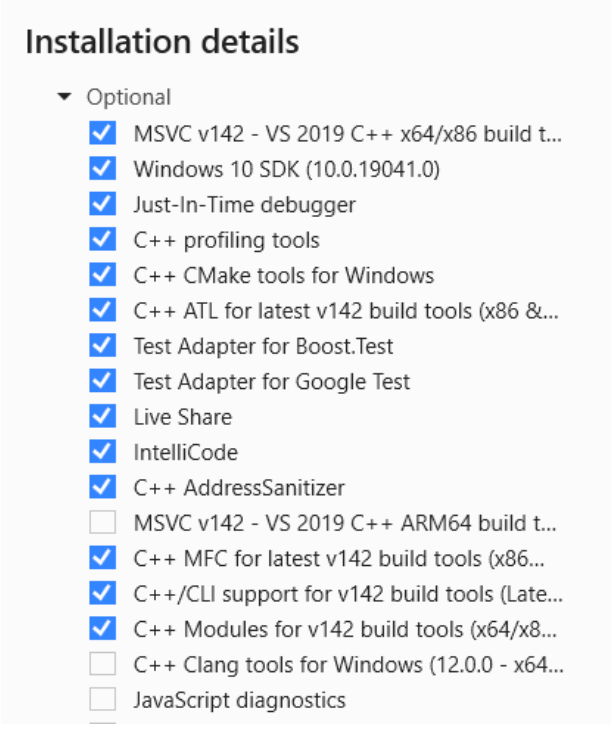


Рисунок 2.2 – Вибір необхідних опцій для проектів C++/CLI та MFC (необхідно обрати “C++/CLI” та “C++ MFC”)

2.2 Створення проектів типу “CLR Empty Project (.NET FrameWork)” мовою C++/CLI у Microsoft Visual Studio

Якщо необхідні компоненти Microsoft Visual Studio встановлено, можна розпочинати створення проектів. Одразу після завантаження, коритувач бачить вікно відкриття та/або створення нових проектів (Рисунок 2.3). В даному вікні користувач бачить список проектів, що нещодавно відкривалися, також, може відкрити проект, створити новий, виконати інші дії.

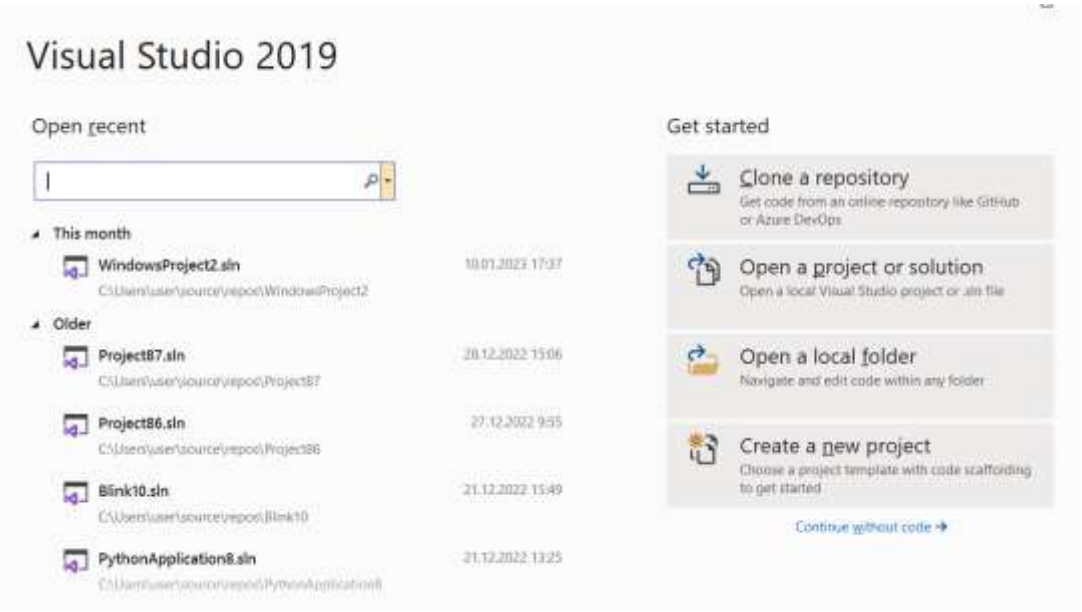


Рисунок 2.3 – Початкове вікно відкриття/створення проектів

Для створення нового проекту обираємо “Create a new project” та переходимо у наступне вікно – рисунок 2.4.

Як видно з рис. 2.4, під час створення нового проекту в лівій панелі вікна користувач бачить перелік типів проектів, що нещодавно створювались у Visual Studio. У правій панелі, користувач може обрати проекти, що відповідають певній мові програмування (зверху обрано “C++”), тип цільової платформи (обрано всі) та обрати окремі типи проектів (“Desktop”, “Mobile”, “IoT” та інші).

Для створення C++/CLI проекту із Window Form оберіть “CLR Empty Project (.NET FrameWork)”, натисніть кнопку “Next”, це викликає вікно конфігурації проекта, показане на Рис. 2.5.

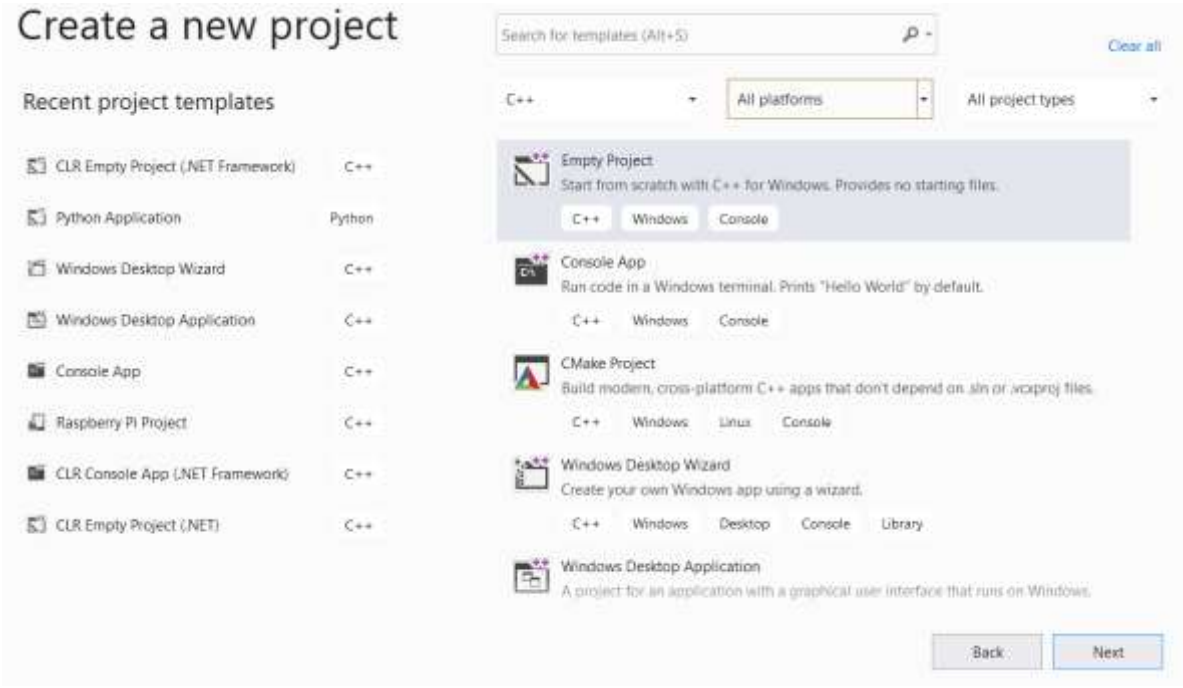


Рисунок 2.4 – Вікно створення нових проектів

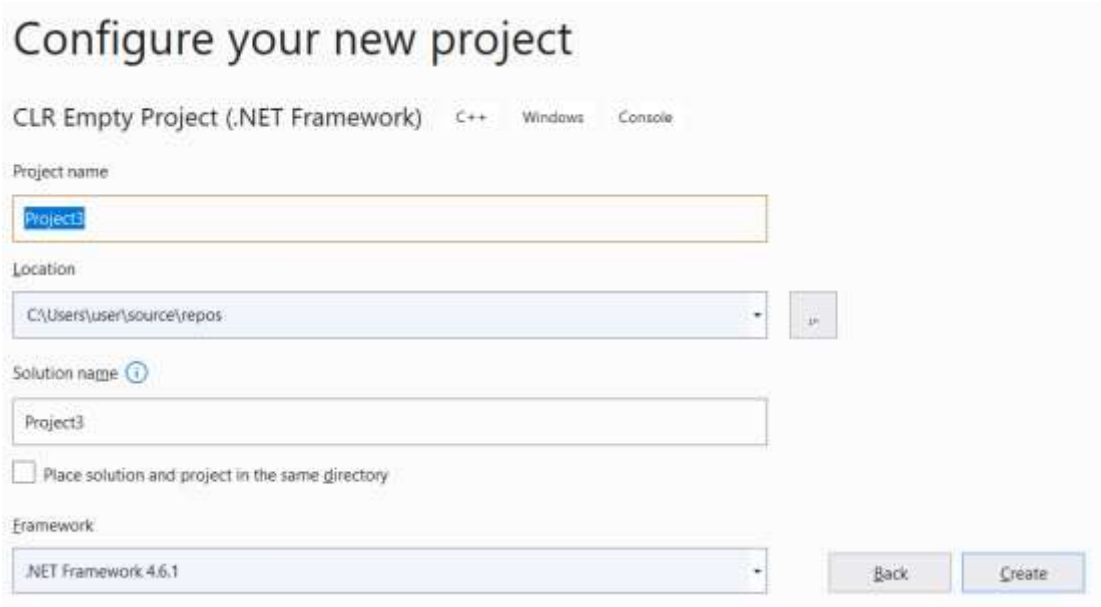


Рисунок 2.5 – Вікно конфігурації нового проекта

Як видно з Рис. 2.5, розробнику пропонується ім’я нового проекта (Project 3), розташування на диску. Відповідно імені проекта задається ім’я рішення (solution). Також користувач може змінити версію .NET Framework. Натискання кнопки “Create” створює новий простір проекта, показаний на Рис. 2.6.

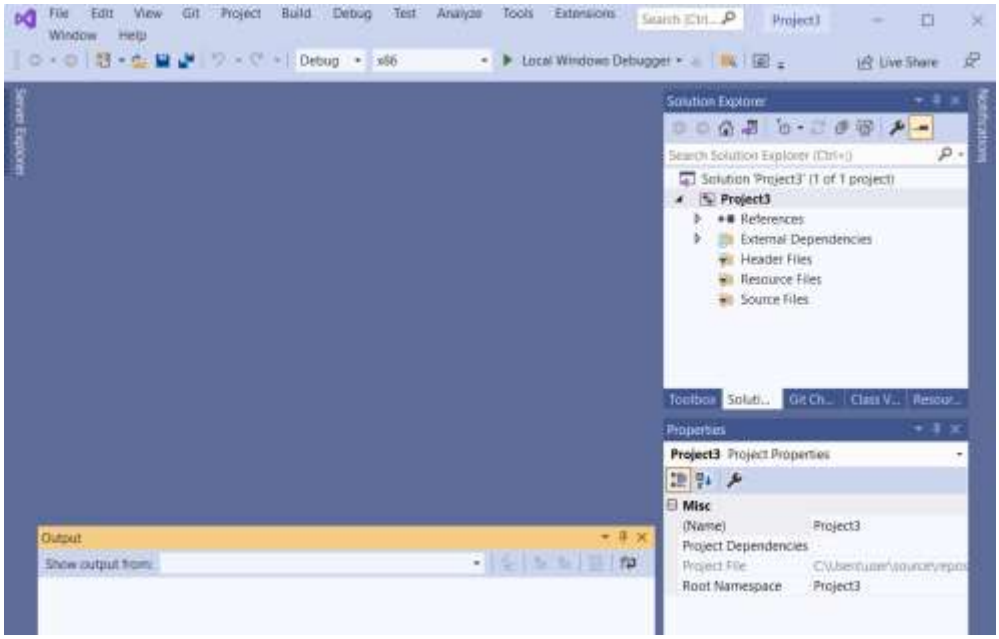


Рисунок 2.6 – Вигляд вікна нового проекту Project3

Створений у такий спосіб проект є порожнім, в ньому поки що немає програмних файлів або ресурсів, проте їх неважко додати.

Для Windows-програми характерне візуальне представлення програми у вигляді вікна певного типу, тому .NET надає можливість додавання Windows Form – віконної форми із можливістю формування інтерфейсу користувача за допомогою елементів керування.

Для додавання Windows Form необхідно у панелі Solution Explorer на обраному імені проекта натиснути праву клавішу миші і додати до проекту (Add) новий елемент (New Item), що відкриває нове вікно (Рисунок 2.7), в якому слід обрати UI (або CLR) в лівій панелі, і Windows Form – у правій.

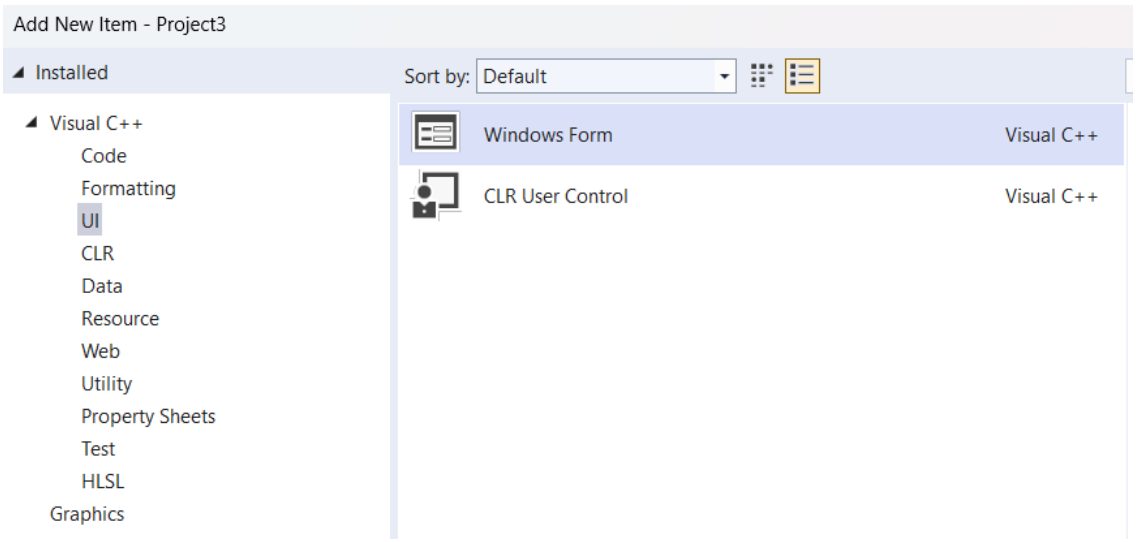


Рисунок 2.7 – Додавання Windows Form у проект

Наступним кроком розробник може побачити баг (похибку) Visual Studio – (показано на Рис. 2.8), коли створена Windows Form не відображається. Помилка пов’язана із масштабуванням екрану Windows (125% замість 100%).

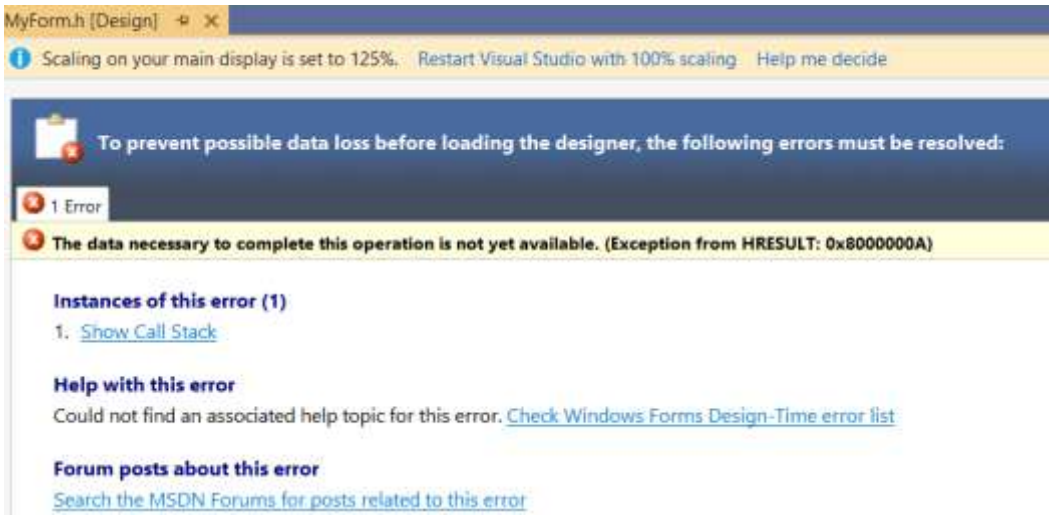


Рисунок 2.8 – Помилка початкового показу Windows Form

Дана помилка присутня в останніх версіях Microsoft Visual Studio. Вона, в основному, вирішується закриттям-відкриттям поточного проекта. Як тільки проект відкрито наново, ця проблема зникає (рис. 2.9).

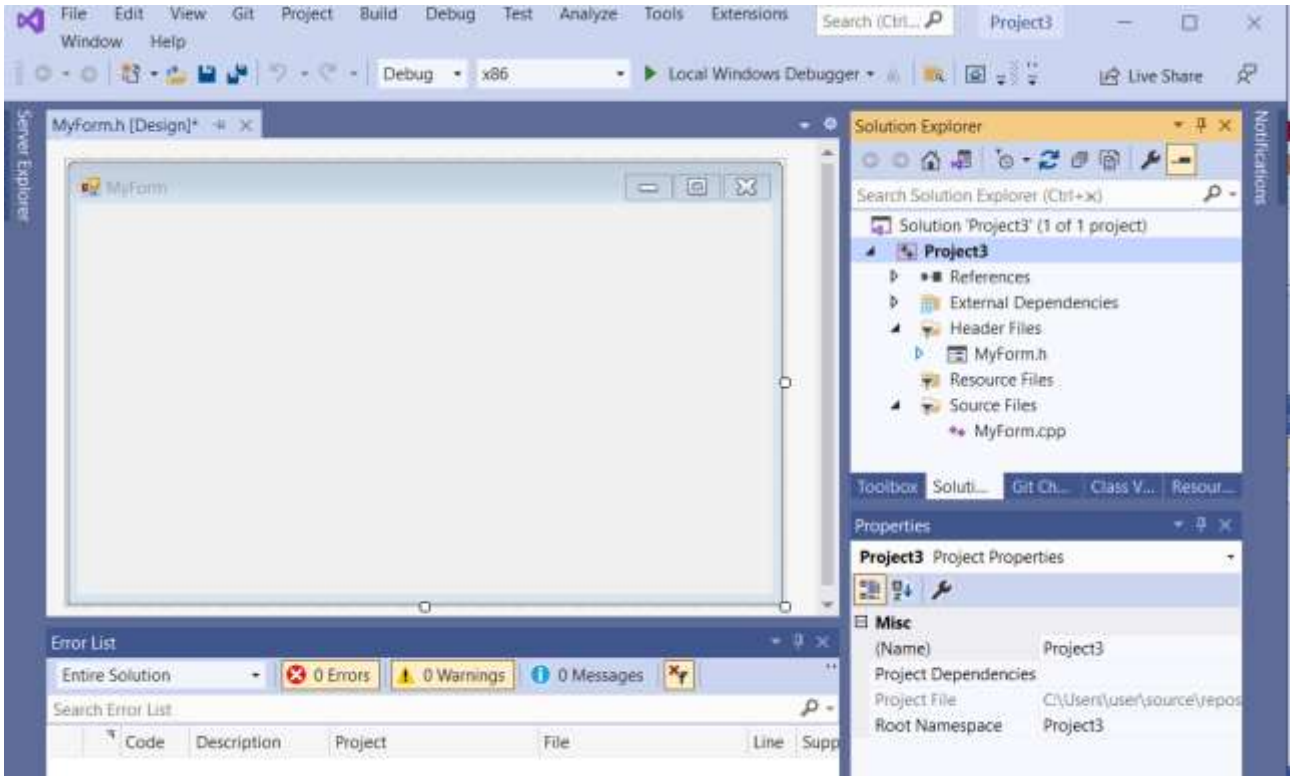


Рисунок 2.9 – Проект Project3 із доданою Windows Form

Як видно з панелі Windows Solution, проект вже містить декілька компонентів та файлів, зокрема MyForm.h і MyForm.cpp. Ці файли відповідають доданій в проект формі, містять її властивості та формуються автоматично. Наша подальша робота буде пов'язана саме з написанням програмного коду в них.

Для запуску проекту на виконання буде необхідно:

– написати код файлу MyForm.cpp;

– визначити опції проекту Project3.

Клікнувши на імені файлу MyForm.cpp, попадаємо в редактор коду. В ньому необхідно написати:

```
#include "MyForm.h"

using namespace System;    // підключення простору імен System
using namespace System::Windows::Forms; // підключення простору імен Windows::Forms

[STAThreadAttribute]                    // встановлення атрибутів однопоточного проекту

void Main(array <String^>^ args)     // функція main із командним рядком args
{
                    Application::EnableVisualStyles();   // вмикання стилів візуалізації
                    Application::SetCompatibleTextRenderingDefault(false); // сумісні
                    //
                    Project3::MyForm form;
                    Application::Run(%form);
}
```

Приклад 2.1 – Вміст файлу MyForm.cpp

Слід трохи пояснити зазначений код. Файл MyForm.cpp починається з підключення MyForm.h – по суті основного файла проекта, який містить більшість інформації і який надалі ми будемо використати для формування коду проекту. Далі – підключаються необхідні простори імен: System – для роботи з класом Application, та Windows::Forms для роботи з класом форми.

[STAThreadAttribute] означає встановлення Single Thread Attribute – тобто атрибутів, що вказують на однопоточність програми. Слід, одразу, зазначити, що існує і режим MTAThreadAttribute – для програм із багатопотоковістю.

Функція Main() проекту визначається в повному стилі – із списком параметрів командного рядка. Тобто, це означає можливість передати масив параметрів args з консольного виклику виконуваного (exe) файла проекту.

Дві функції – EnableVisualStyles() та SetCompatibleTextRenderingDefault() є службовими. Вони забезпечують підтримку стилів візуалізації та встановлення

сумісного режиму відображення тексту. Проте, у більшості випадків, ці функції можна не вказувати, результат роботи функції не зміниться.

Важливими для роботи проекту є два останні рядки. Спочатку у проекті створюється форма form класу MyForm. Потім ця форма передається у функцію Run() класу Application для запуску проекта із заданою формою.

Ще один важливий крок полягає у встановленні властивостей проекту. Для цього на вибраному імені проекту Project3 натискаємо комбінацію клавіш “Alt+Enter” (або треба обрати “Properties” у меню, що з’являється натиском правої клавіші міші). Обираємо Linker->System (Рисунок 2.10а), у правій панелі обираємо Subsystem - Windows.

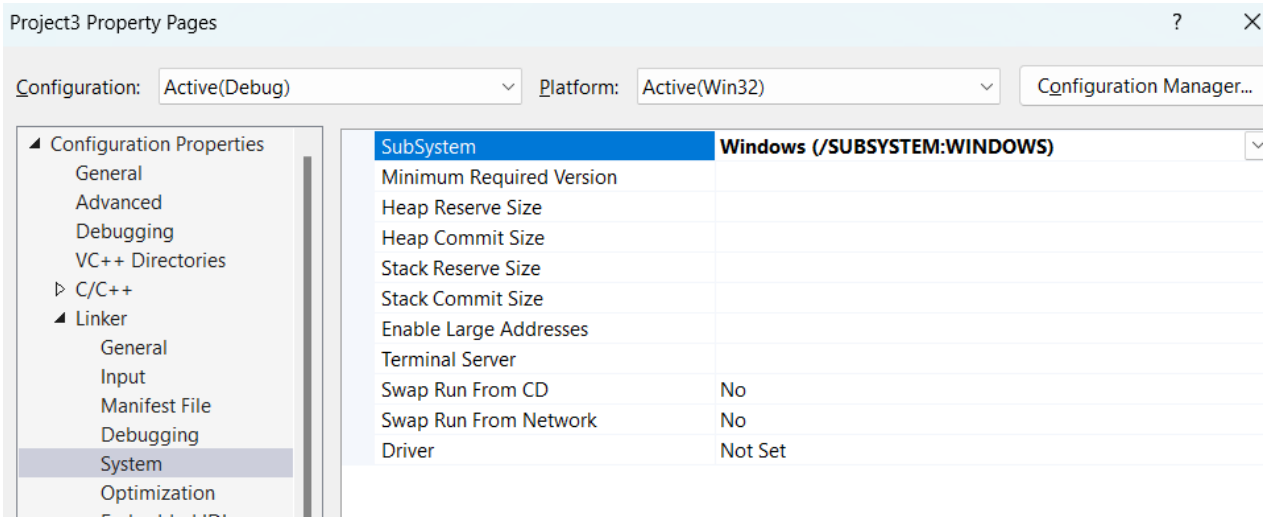


Рисунок 2.10а – Вибір підсистеми проекту

Другою необхідною для визначення властивістю опцією є встановлення точки входу (Entry Point), тобто функції, з якої почнеться виконання проекту. Це в нашому випадку має бути функція Main(), як показано на рис. 2.10б. Зазначимо, що точкою входу може може бути будь-яка функція, наприклад notSoMain() або написана з малої літери main(), проте назва функції у вікні рис. 2.10б має бути тотожною функції з коду файлу MyForm.cpp.

Таким чином, усі кроки зроблено, можна і запускати проект. Натискаємо Ctrl+F5 для запуску проекту. Якщо все зроблено вірно, маємо побачити Windows Form, як це показано на рис 2.11.

Як і будь-яка виконувана програма, Project03 має виконуваний *.exe файл, який знаходиться у каталозі проекту. За замовчанням, цей каталог знаходиться в директорії Users/User, як це показано на Рис. 2.12. Саме в ньому можна знайти файли MyForm.h і MyForm.cpp. Виконуваний модуль знаходиться у каталозі Debug, хоча і на один рівень вище від показаного.

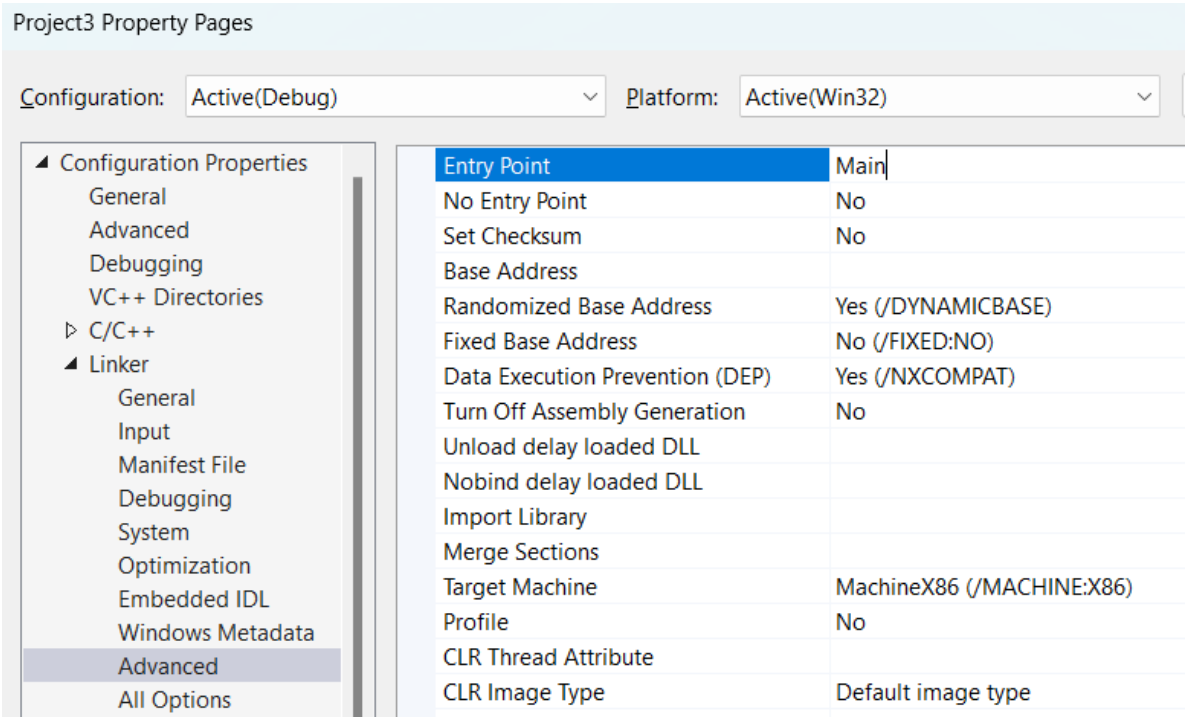


Рисунок 2.10б – Встановлення точки входу (Entry Point) – функції Main()

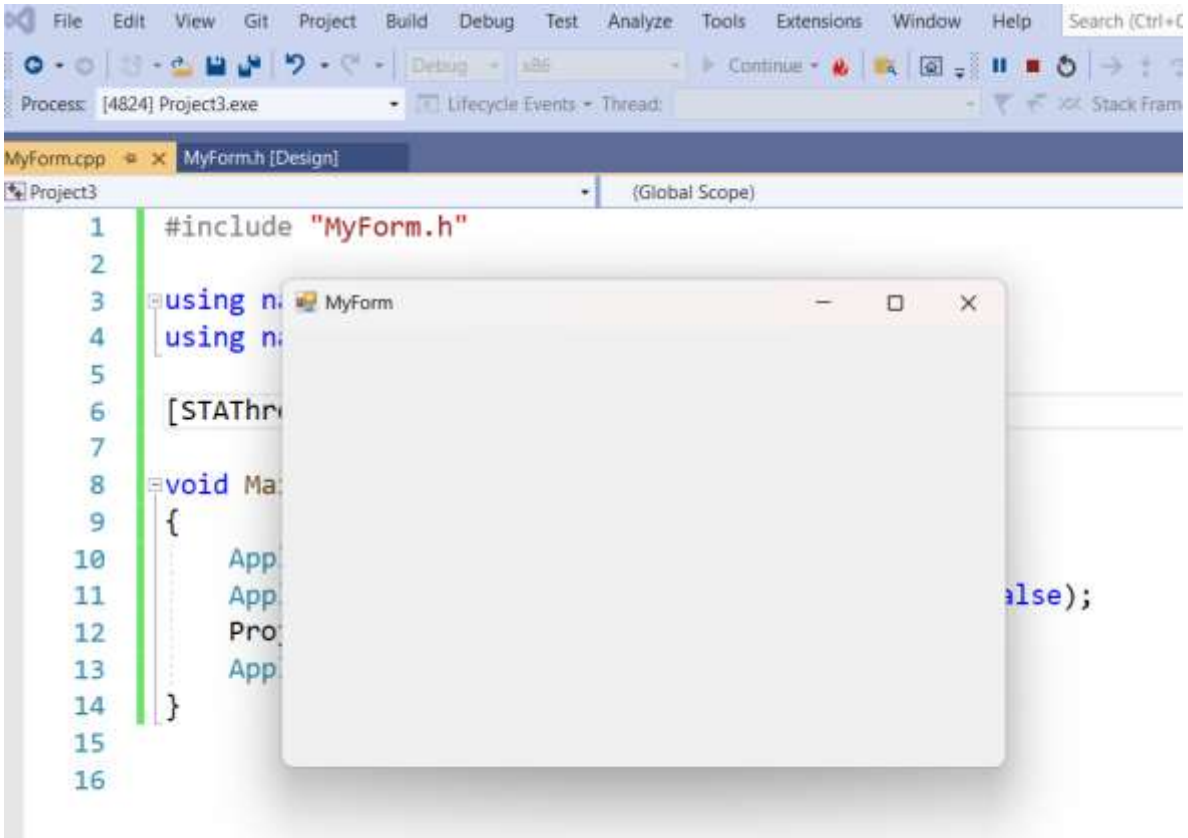


Рисунок 2.11 – Готова запущена проста програма із Windows Form

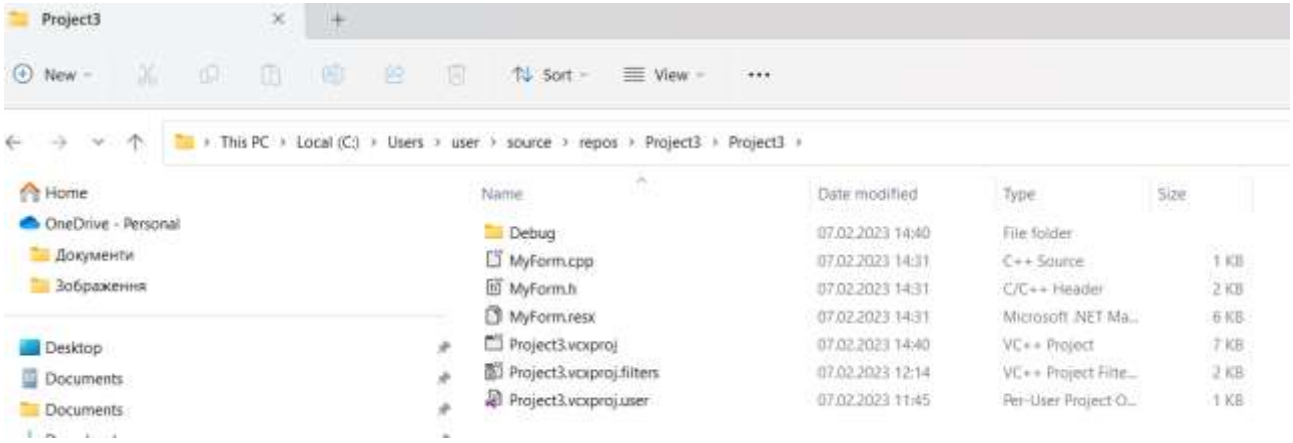


Рисунок 2.12 – Підкаталог Project3/Project3 із файлами проекта

Слід зауважити, що об’єм написаного коду даного проекту є малим і зрозумілим. Проте, це не відміняє великий об’єм роботи, що виконується самим Visual Studio. В реальності, вказаними діями користувача автоматично створюється декілька сторінок програмного коду, що формує файл MyForm.h і означає створення в проекту простору імен Project03, класу MyForm, конструктора класу із викликом функції InitializeCompoment() із автоматично створеним вмістом, що відповідає властимостям і оформленню форми MyForm (приклад 2.2). Нагадаємо, що інший файл проекта MyForm.cpp показано в прикладі 2.1.

```
namespace Project3 {
	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;
	/// <summary>
	/// Summary for MyForm
	/// </summary>
	public ref class MyForm : public System::Windows::Forms::Form
	{public:
		MyForm(void)
		{	InitializeComponent();
			//TODO: Add the constructor code here
		}
	protected:
		/// <summary>
		/// Clean up any resources being used.
		/// </summary>
		~MyForm()
		{
			if (components)
			{
				delete components;
```

```
            }
        }

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->SuspendLayout();
            //
            // MyForm
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(623, 350);
            this->Name = L"MyForm";
            this->Text = L"MyForm";
            this->ResumeLayout(false);

        }
#pragma endregion
    };
}
```

Приклад 2.2 – Повний код файла MyForm.h, сформований Visual Studio

3 ОБРОБКА ПОДІЙ WINDOWS FORM ПРОГРАМИ

3.1 Загальна інформація про обробку подій у C++/CLI-програмі

Windows, як будь-яка операційна система взаємодіє з програмою через механізм обробки подій (інший термін – повідомлення). Події мають різну природу. Це може бути вибір користувачем пункту меню, натискання клавіш клавіатури, переміщення курсору миші екраном вашого комп'ютера, надходження інформації про системні події у мережі тощо.

Програма має „знати" на які події їй слід реагувати, на які – не слід, а якщо необхідна реакція на повідомлення – визначати, яку функцію-обробник слід викликати у відповідь на надходження події. Аналогічний механізм обробки подій (повідомлень) існує і інших Windows-програмах, зокрема, в MFC [1]. Але, якщо в MFC користувач самостійно (в основному) організує обробку повідомлень, в проекті .NET FrameWork ці операції значною мірою автоматизовані.

3.2 Виведення текстової інформації у Windows Form C++/CLI-програми

Обробка подій має бути відчутна не лише для програми, але і для користувача. Тому важливим елементом стає виведення інформації у віконну форму Windows Form, що представляється конкретним об'єктом MyForm.

Текст, який виводится у MyForm, спочатку має бути підготовлений і відформатований функцією Format класу String:

```
String^ Format( String^ format, Object^ arg0 );
String^ Format( String^ format, Object^ arg0, Object^ arg1 );
String^ Format( String^ format, Object^ arg0, Object^ arg1, Object^ arg2 );
String^ Format( String^ format, ... array<Object^>^ args );
```

При цьому параметрами є:

System::String ^ format - форматований рядок тексту.
System::Object ^arg0, arg1, arg2 – перший, другий, третій об'єкти, що додаються в рядок;
array<System::Object^>^ args – список (масив) об'єктів.

Це означає, що кількість аргументів може складати один, два, три чи більше. А випадку великої кількості аргументів доцільно використати їх предста-

влення у вигляді масиву. Ще один момент – характер даних, що форматуються. Це можуть бути дані, фактично, будь-якого типу, про що свідчить тип Object^.

Слід зауважити, що вигляд прототипу функції Format() не дає уявлення про те, як функція буде використатися. Позитивним є те, що у використанні Format() нагадує функцію printf() зі стандартної бібліотеки введення-виведення C/C++, точніше sprintf(), оскільки і в ній результат копіюється в текстовий рядок, також і wsprintf(), що визначається в MFC. Це показує однаковість підходу.

Наприклад, необхідно передачи ціле значення x в текстовий рядок s. В консольних засобах stdio.h та в MFC (afxwin.h) це забезпечиться кодом:

```
char s[100];
int x=10;
sprintf(s,”%d”,x);
```

```
char s[100];
int x=10;
wsprintf(s,”%d”,x);
```

Тобто, змінна x у форматі %d (вставка значень цілого типу) вставляється в рядок тексту s. Якщо ж в текстовий рядок s необхідно передачи дійсне (float) значення x, зміниться лише формат вставлення дійсних даних - %f, хоча в MFC це вже не спрацює з-за обмеженості wsprintf():

```
char s[100];
float y=10.5;
sprintf(s,”%f”,y);
```

Фактично, той же формат роботи функції забезпечується і в C++/CLI:

```
String^ s;
        int x = 10;
s=String::Format("{0}",x);
```

```
String^ s;
float y = 10.5;
s=String::Format("{0}",y);
```

Легко побачити, що спосіб виведення цілих та дійсних значень не відрізняється! Функції Format() зовні все одно, які дані виводити, в обох випадках значення змінних виводяться за допомогою вказання {0} у форматі рядка виведення. Якщо ж вказати {1}, з’явиться помилка, оскільки номер {0} чи {1} означають не формат даних, що виводяться, а лише номер змінної у списку аргументів, коли ми маємо декілька їх.

На жаль, ввівши зазначений код, розробник не побачить нічого на екрані, оскільки дія функції Format() обмежується форматуванням рядка даних, а для виведення необхідні інші функції. Прийшов час розглядіти їх.

Скомбінуємо обидва приклади:

```
String^ s;
        int x = 10;
        float y = 10.5;
s=String::Format("{0} {1}",x,y);      // спочатку виводиться x, потім y
s=String::Format("{1} {0}",x,y);      // спочатку виводиться y, потім x
```

Приклад 3.1 – Приклад коду форматування тексту

Для виведення тексту (а пізніше і графіки) у віконну форму MyForm необхідно:

1. Створити об’єкт графіки, пов’язаний з формою:

```
Graphics^ g = MyForm::CreateGraphics();
```

2. Сформувати текстовий рядок для виведення:

```
String^ s;
int x = 10; float y=-1.234;
s=String::Format("x={0} y={1}",x,y);
```

3. Вивести текстовий рядок у форму згідно заданих параметрів:

```
g->DrawString(s, gcnew Drawing::Font("Arial", 10), Brushes::Black, 20, 20);
```

Повний код, який є реакцією (подією) на кнопку (Рис. 3.1), натиснуту в віконній формі, буде виглядати так (використано динамічне створення об’єкта шрифта Font за допомогою функції gcnew із контролем очищення пам’яті):

```
        private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
        {       Graphics^ g = MyForm::CreateGraphics();
                String^ s;
        int x = 10; float y=-1.234;
        s=String::Format("x={0} y={1}",x,y);
                g->DrawString(s, gcnew Drawing::Font("Arial", 12), Brushes::Black, 20, 20);
}
```

Приклад 3.2 – виведення тексту кнопкою Button1 у віконній формі.

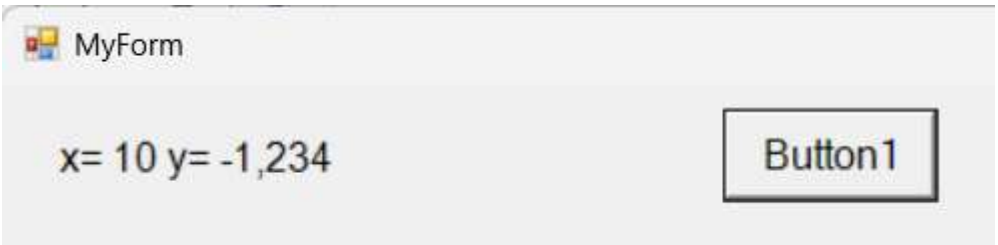


Рисунок 3.1 – Приклад виведення значень змінних x та y в MyForm

Таким чином, безпосереднє виведення текстового рядка забезпечується функцією DrawString(), що належить до широкого кола графічних процедур, які забезпечуть виведення ліній, багатокутників, еліпсів, іншої стандартної графіки. Крім того, це підкреслює універсальність організації виведення графіки будь-якого типу.

DrawString() має декілька варіантів формату, серед них і такий:

```
void DrawString( String^ s, Font^ font, Brush^ brush, float x, float y );
Параметри:
System::String^ s                    // рядок тексту, що відображається.
System.Drawing::Font^ font   // шрифт, яким здійснюється виведення тесту.
System.Drawing::Brush ^brush // пензель, що визначає колір тексту.
System::Single x // x-координата верхнього лівого кута тесту, що виводиться.
System::Single y // y-координата верхнього лівого кута тесту, що виводиться.
```

Перший параметр означає рядок тексту, що планується виводити, і який формується функцією Format() відповідно до прикладу 3.1 вище.

Другий параметр означає визначення типу і розміру шрифта. В динамічному режимі створення і подальшого очищення другий параметр виглядає так:

```
// gc – garbage collection + new - збирання сміття після динамічного створення)
gcnew Drawing::Font("Arial", 10)
```

Третій параметр означає визначення кольору тексту, який обирається зі стандартного списку типу Brushes, наприклад: Brushes::Black або Brushes::RosyBrown.

Четвертий і п’ятий параметри означають координати початку виведення тексту, що, відповідно до визначення системи координат у вікні, обраховуються від лівого верхнього кута (точки (0, 0)) віконної форми MyForm. Хоча відповідно до формату функції, необхідно вказувати значення типу float, в нашому випадку спрацьовують і значення цілого типу – 20, 20.

Слід зазначити, що існують і інші варіанти формату функції DrawString(). Їх відмінність, в основному, полягає у визначенні прямокутної області, що обмежуватиме координати виводу.

3.3 Організаці обробки подій у Windows Form C++/CLI-програмі

3.3.1 Подія “Load”

Оскільки вже відомо, як формувати та виводити інформацію у віконну форму MyForm, можна переходити до розгляду обробки подій.

Як сама віконна форма MyForm, так і будь-який елемент в ній можуть надсилати/відправляти певний список подій (events). Щоб подивитися цей список подій достатньо обрати форму та у панелі “Properties” обрати сторінку “Events”, позначену символом блискавки . Зазвичай, для віконної форми, подією за замовчанням є “Load”, тобто подія, яка відбувається під час завантаження поточного елемента. Вигляд форми і панелі подій показано на рис. 3.2.

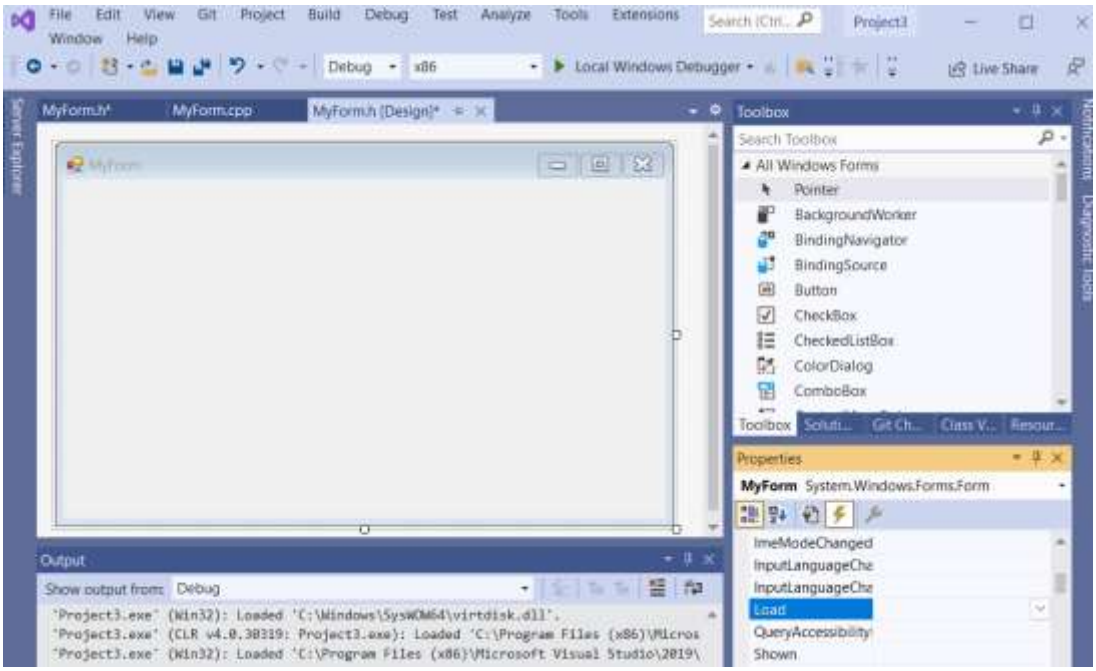


Рисунок 3.2 – Форма MyForm та обрана подія “Load”

Тепер, якщо двічі натиснути на назві події “Load”, в програмі у файлі MyForm.h буде автоматично створено функцію обробник події “Load”:

```
private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e)
{       }
```

Додамо код, який пояснює подію, формує рядок тексту і виводить інформацію у заголовок форми, використовуючи властивість “Text” форми:

```
private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e)
{       String^ s;
        s = String::Format("Відбулася подія завантаження форми");
        MyForm::Text = s; // передача сформованого рядка тексту у заголовок форми   }
```

Приклад 3.3 – Обробник події “Load” для MyForm

Результат обробки події “Load” для MyForm показує Рисунок 3.3.

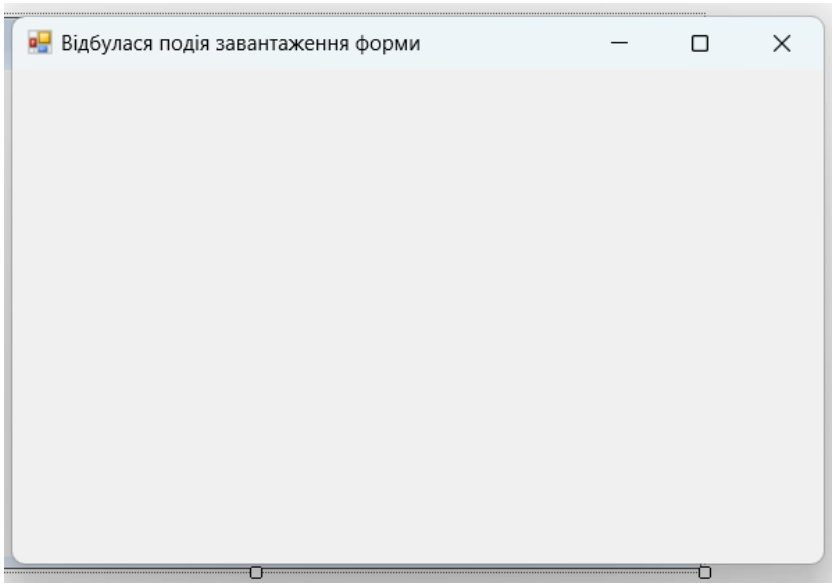


Рисунок 3.3 – Результат події “Load” у MyForm

Зазначимо, що в даному випадку навіть немає необхідності виведення тексту у форму. Більш того, виведення із DrawString() на разі може не відбутися.

3.3.2 Подія “KeyDown”

Стандартною подією у віконній формі є натискання клавіш клавіатури. Для її обробки на сторінці “Events” панелі “Properties” має обиратися “KeyDown”. Обробник події має визначати код і символ натиснутої клавіші.

Код клавіші у обробнику отримується безпосередньо з події натискання клавіші - Keys k = e->KeyCode, де e і є подією (event). Потім цей код передається у формований рядок тексту і виводиться у задані координати віконної форми:

```
private: System::Void MyForm_KeyDown(System::Object^ sender,
System::Windows::Forms::KeyEventArgs^ e)
{ Keys k = e->KeyCode;
  String^ s = String::Format("Натиснута клавіша {0}", k);
  Graphics^ g = MyForm::CreateGraphics();
  g->Clear(MyForm::BackColor);  // очищення у стандартний колір форми
  g->DrawString(s, gcnew Drawing::Font("Arial", 10), Brushes::Red, 20, 20);
}
```

Приклад 3.4 – Обробник події “KeyDown” для MyForm

Результат виконання прикладу 3.4 показано на рис. 3.4.

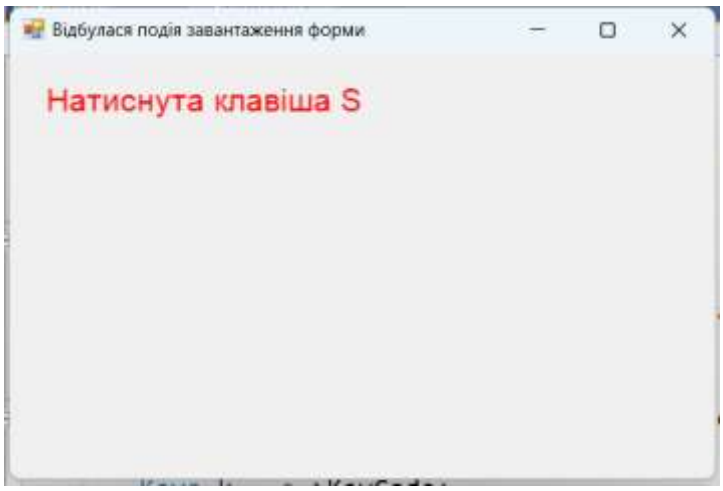


Рисунок 3.4 – Результат події “KeyDown” у MyForm

В деяких випадках подія “KeyDown” спрацьовує лише, коли задано властивість віконної форми “KeyPreview” із значенням “true”.

3.3.3 Подія “Move”

Ще однією стандартною подією у віконній формі є її переміщення на екрані за допомогою миші. Для обробки на сторінці “Events” панелі “Properties” має обиратися “Move”. Обробник події отримує екранні координати форми із властивостей її розташування на робочому столі Windows:

```
int x = MyForm::DesktopLocation.X;
int y = MyForm::DesktopLocation.Y;
```

Решта коду формується у вже відомий спосіб, як показано у Прикладі 3.5:

```
private: System::Void MyForm_Move(System::Object^ sender, System::EventArgs^ e)
{           int x = MyForm::DesktopLocation.X;
            int y = MyForm::DesktopLocation.Y;
            Graphics^ g = MyForm::CreateGraphics( );
            g->Clear(Color::LightGray); // очищення кольору у LightGray
            String^ s;
            s = String::Format("Form position: {0} {1}", x, y);
            g->DrawString(s, gcnew Drawing::Font("Arial",12),Brushes::Black, 5.0, 110.0);
            }
```

Приклад 3.5 – Обробник події “Move” для MyForm

Результат виконання прикладу 3.5 показано на рис. 3.5.

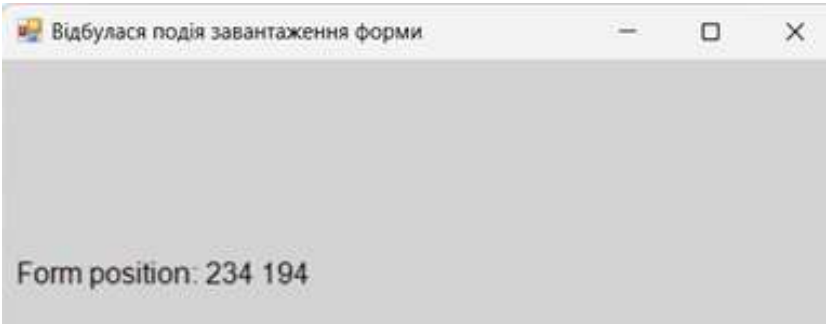


Рисунок 3.5 – Результат події “Move” у MyForm

3.3.4 Подія “Resize”

Подія “Resize” є досить подібною до “Move” і спрацьовує коли користувач змінює розмір віконної форми. Для обробки такої події на сторінці “Events” панелі “Properties” має обиратися “Resize”. Обробник події отримує ширину та висоту форми з її властивостей її розміру та формує рядок тексту з інформацією:

```
private: System::Void MyForm_Resize(System::Object^ sender, System::EventArgs^ e)
{int x = MyForm::Size.Width; int y = MyForm::Size.Height;
 Graphics^ g = MyForm::CreateGraphics();
 g->Clear(Color::LightGray);
 String^ s = String::Format("Form size is changed to: {0} {1}", x, y);
 g->DrawString(s, gcnew Drawing::Font("Arial", 12), Brushes::Black, 5.0, 110.0);
}
```

Приклад 3.6 – Обробник події “Resize” для MyForm

3.3.5 Події натискання кнопки миші

Події натиску кнопки миші спрацьовують коли користувач натискає на якому-небудь об’єкті, що входить до MyForm. Для обробки такої події на сторінці “Events” панелі “Properties” має обиратися дія (“Action”) “Click” або “DoubleClick”, або “MouseClick”,“MouseDoubleClick” (Рис. 3.6). Обробник події отримує позицію курсора на екрані (MousePosition) та віднімає позицію форми на екрані (Location). В результаті ми отримуємо пересування тексту за натиском курсора миші (і доволі незрозумілу позицію, якщо різницю не враховувати):

```
private: System::Void MyForm_Click(System::Object^ sender, System::EventArgs^ e)
{           int x1 = MyForm::MousePosition.X; // поточна X-позиція курсора на екрані
            int y1 = MyForm::MousePosition.Y; // поточна Y-позиція курсора на екрані
            int x2 = MyForm::Location.X; // поточна X-позиція MyForm
            int y2 = MyForm::Location.Y; // поточна Y-позиція MyForm
            Graphics^ g = MyForm::CreateGraphics();
            g->Clear(MyForm::BackColor);
            String^ s;
            s = String::Format("Mouse Click position: {0} {1}", x1-x2, y1-y2);
```

```
            g->DrawString(s, gcnew Drawing::Font("Arial", 12),
Brushes::Black, x1-x2, y1-y2);
}
```

Приклад 3.7 – Обробник події “Click” для MyForm

Код обробника подвійного натискання клавіші миші можна дещо скоротити і представити у спосіб рис. 3.7. В ньому враховується, що змінні-члени MousePosition, Location, BackColor і функція CreateGraphics() належать до MyForm.

```
private: System::Void MyForm_DoubleClick(System::Object^ sender, System::EventArgs^ e)
{           int x = MousePosition.X - Location.X;
            int y = MousePosition.Y - Location.Y;
            Graphics^ g = CreateGraphics();
            g->Clear(MyForm::BackColor);
            String^ s = String::Format("Mouse Double Click position: {0} {1}", x, y);
            g->DrawString(s, gcnew Drawing::Font("Arial", 12), Brushes::Black, x, y);
}
```

Приклад 3.8 – Обробник події “DoubleClick” для MyForm

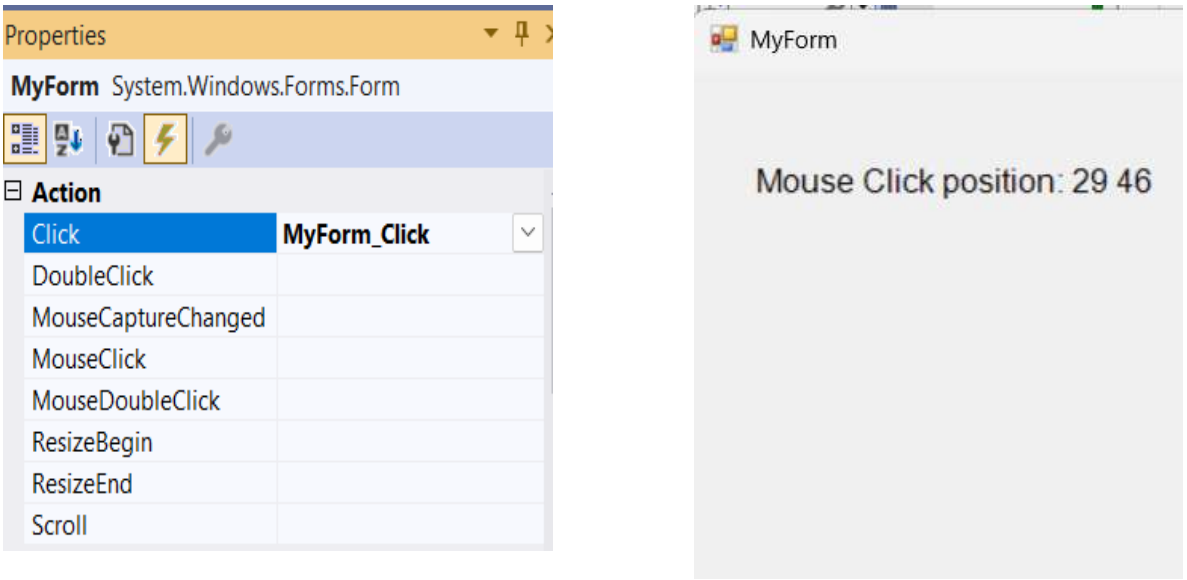


Рисунок 3.6 – Вибір дії із натиску клавіш міші та приклад дії обробника

3.3.6 Подія таймера

Наявність таймера в програмі дозволяє виконувати певні дії синхронно з певними проміжками часу, наприклад, кожну 1 секунду, або кожні 5 хвилин, або декілька годин. Для додавання таймера слід обрати його, як компонент ToolBox і додати його на форму. При йьому таймер з’являється, як додаткова (і невидима під час запуску) властивість MyForm (Рис 3.7). Для першого таймера

форми присвоюється ім’я timer1, але кількість таймерів у програмі не обмежується.

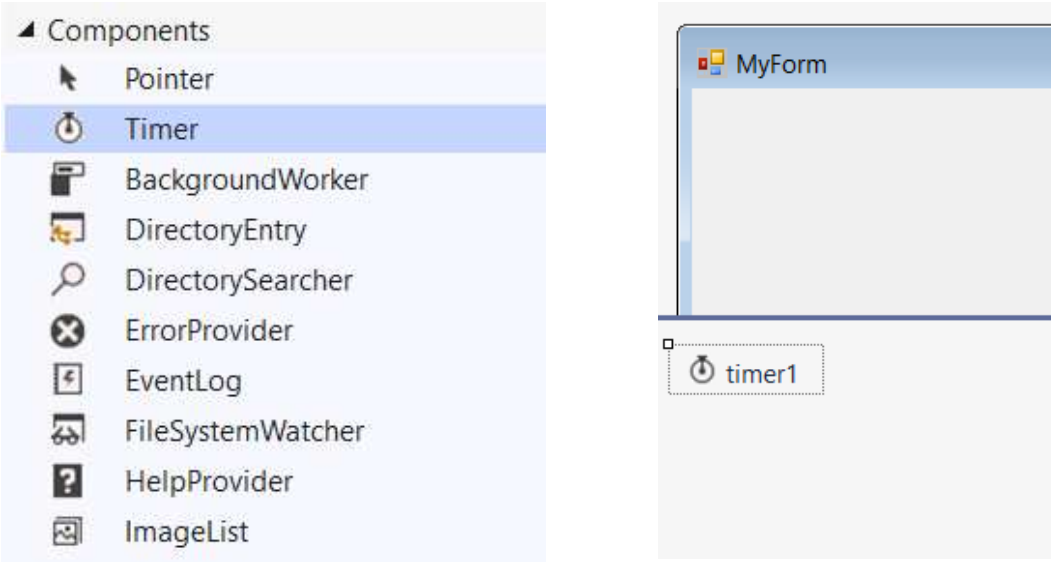


Рисунок 3.9 – Таймер у списку компонентів MyForm і його додавання

Для запуску таймера встановується його інтервал і здійснюється запуск. Це можна зробити, наприклад натиском окремої клавіші “Start Timer”:

```
private: System::Void button1_Click_1(System::Object^ sender, System::EventArgs^ e)
{           timer1->Interval = 1000;
            timer1->Start(); }
```

Приклад 3.9 – Запуск таймера у обробнику події “button1Click” для MyForm

Натискання на іконку таймера (рис. 3.7) забезпечує створення його обробника timer1_Tick(). Приклад нижде показує отримання поточної дати та системного часу (рис. 3.8).

```
private: System::Void timer1_Tick(System::Object^ sender, System::EventArgs^ e)
{Graphics^ g = MyForm::CreateGraphics();
 DateTime localDate = System::DateTime::Now;
String^ s = String::Format("Current Date and Time: {0}", localDate.ToString());
g->Clear(Color::White);
g->DrawString(s, gcnew Drawing::Font("Arial", 12), Brushes::Black, 5.0, 15.0);
}
```

Приклад 3.10 – Обробник події таймера для MyForm

Рисунок 3.8 – Функціонування обробника таймера

Роботу таймера можна зробити більш точною, вказавши дані часу в годинах, звилинах, секундах і, навіть, мілісекундах (рекомендується зменшити інтеовал таймера до 100 мілісекунд) (Рис. 3.9):

```
private: System::Void timer1_Tick(System::Object^ sender, System::EventArgs^ e)
{
        Graphics^ g = MyForm::CreateGraphics();
        DateTime localDate = System::DateTime::Now;
        String^ s = String::Format("Current Date and Time: {0}:{1}:{2}:{3}",
                localDate.Hour, localDate.Minute, localDate.Second,localDate.Millisecond);
        g->Clear(Color::White);
        g->DrawString(s, gcnew Drawing::Font("Arial", 12), Brushes::Black, 5.0, 15.0);
}
```

Приклад 3.11 – Обробник події таймера для MyForm (з точністю до мілісекунд)

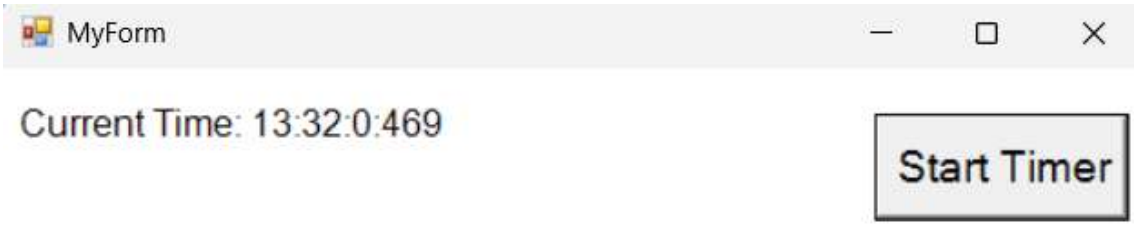


Рисунок 3.9 – Функціонування таймера з точністю до мілісекунд

Додатково зазначимо, що зупинка таймера здійснюється функцію Stop():

```
timer1->Stop();
```

Використання таймера також буде розглядатися в розділах, присвячених використанню 2D і 3D графіки.

3.3.7 Вікна повідомлень в C++/CLI

Операційна система Microsoft Windows забезпечена стандартними засобами інформування користувача про свої події, зокрема запити користувачу про режими роботи програми, помилки, інші повідомлення.

Зазвичай вікна повідомлень виводять інформацію для користувача і чекають на його реакцію. В .NET реалізовано окремий клас MessageBox(), до функцій якого належить створення і відображення вікон повідомлень з необхідним текстом і заголовком, також, з комбінацію визначених піктограм і кнопок. Даний клас у реалізації .NET походить від WinAPI-функції AfxMessageBox і, практично, реалізований у спосіб, аналогічний до MFC.

Для визначення вікна повідомлення задають:

- текст повідомлення;
- заголовок вікна повідомлення;
- стиль піктограм (іконок) вікна повідомлення;
- стиль набору кнопок вікна повідомлення.

Цей набір параметрів можна проілюструвати прикладом:

```
private: System::Void button6_Click(System::Object^ sender, System::EventArgs^ e)
{
        String^ message = "Бажаєте закрити форму";
        String^ caption = "Закриття форми";
        MessageBoxButtons buttons = MessageBoxButtons::OKCancel;
        MessageBoxIcon icon = MessageBoxIcon::Question;
        System::Windows::Forms::DialogResult result;
        result = MessageBox::Show(this, message, caption, buttons, icon);
        if (result == System::Windows::Forms::DialogResult::OK) Close();
}
```

Приклад 3.12 – Обробник з вікном повідомлення MessageBox

В даному випадку спочатку описуються параметри: message – текст у вікні повідомлення “Бажаєте закрити форму”, caption “Закриття форми” – заголовок вікна повідомлення, buttons OKCancel – стиль кнопок, icon – стиль іконок Question, result – варіант відповіді, який перевіряється – натискання ОК. Результатом буде вікно повідомлення, показане на рис. 3.10.

Варіантами оформлення з точки можуть бути комбінації клавіш та іконки, показані на рис. 3.11.

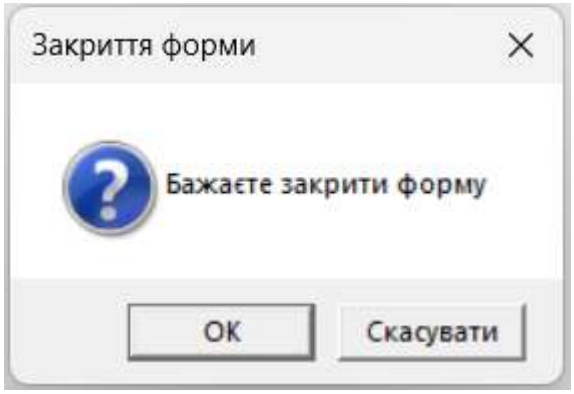
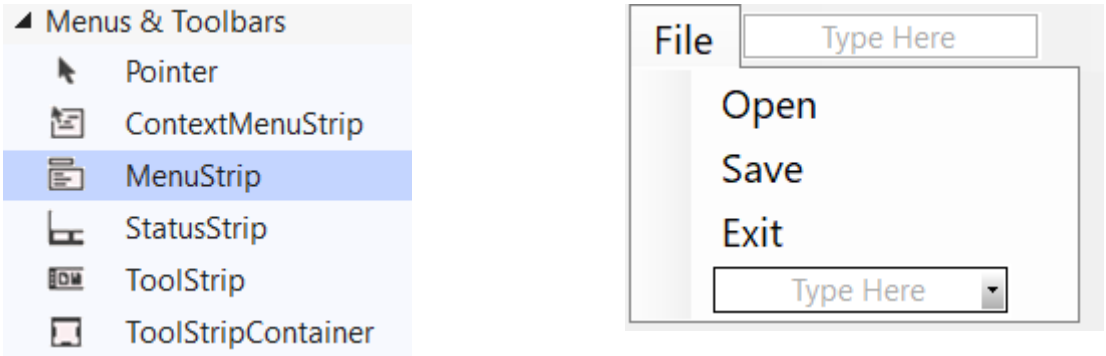


Рисунок 3.10 – Вигляд вікна повідомлення

Рисунок 3.11 – Варіанти налаштування кнопок та іконок вікна повідомлення

3.3.8 Використання меню у Windows Form

Використання меню у програмах .NET залишається актуальним, на що вказує присутність меню в основних програмних продуктах Microsoft, щоправда, основна увага приділяється не текстовим меню, реалізованих елементом MenuStrip, а графічним меню, що задаються елементом ToolStrip. Обидва ці елементи можна знайти у Menu & Toolbars (рис. 3.12).



а) б)

Рисунок 3.12 – Вибір меню в панелі інструментів (а)
та визначення його пунктів (б)

Послідовність додавання текстового меню MenuStrip можна визначити так:

1. вибрати меню MenuStrip у панелі інструментів (Рис. 3.12-а);
2. додати меню у Windows Form (додається у верхній лівий кут форми);
3. визначити пункти меню (Рис. 3.12-б – визначено Open, Save, Exit);
4. визначити код обробників пунктів меню подвійним кліком на кожному пункті і записом коду для кожного обробника.

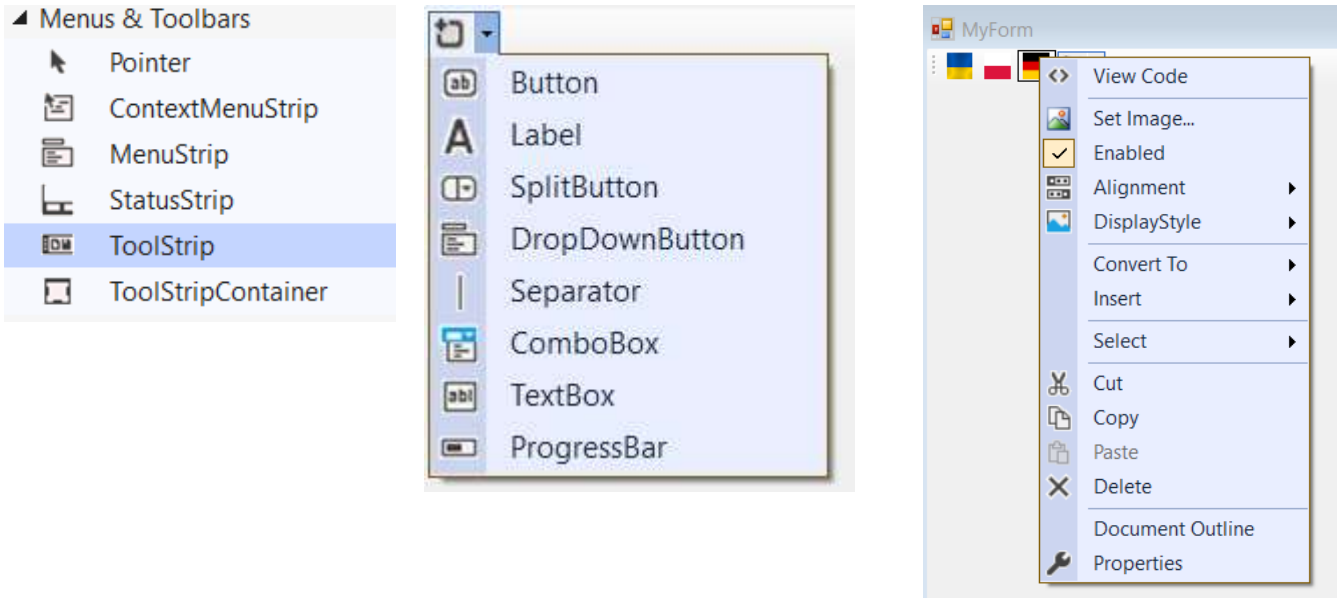
Текст обробників для початку може бути зовсім простим, як показано у прикладі 3.11, зокрема базуватися на вікнах повідомлення MessageBox або на прямих викликах окремих функцій форми, як для пункту «Exit».

```
private: System::Void openToolStripMenuItem_Click(System::Object^ sender, System::EventArgs^ e)
{MessageBox::Show("Open");
}

private: System::Void saveToolStripMenuItem_Click(System::Object^ sender, System::EventArgs^ e)
{       MessageBox::Show("Save");
}
private: System::Void exitToolStripMenuItem_Click(System::Object^ sender, System::EventArgs^ e)
{       Close();
}
```

Приклад 3.13 – Реалізація обробників пунктів меню MenuStrip

Іншим прикладом реалізації меню є графічне меню ToolStrip.



а) б) в)

Рисунок 3.13 – Вибір меню ToolStrip в панелі інструментів (а), визначення типу елемента ((б), обрано Button), та визначення його пунктів зображеннями (в)

Для додавання ToolStrip необхідно:

1. вибрати меню ToolStrip у панелі інструментів (Рис. 3.13-а);
2. додати меню ToolStrip у Windows Form (додається у верхній лівий кут форми);
3. обрати тип елемента меню ToolStrip, наприклад Button (Рис. 3.13-б);
4. визначити зображення пунктів меню ToolStrip, для чого натиснути праву клавішу миші і вибрати SetImage (Рис. 3.13-в – визначено прапорів України, Польші та Німеччини);
5. визначити код обробників пунктів меню ToolStrip подвійним кліком на кожному пункті і записом коду для кожного обробника.

```
private: System::Void toolStripButton1_Click(System::Object^ sender, System::EventArgs^ e)
{
        MessageBox::Show("Україна");
}
private: System::Void toolStripButton2_Click(System::Object^ sender, System::EventArgs^ e)
{
        MessageBox::Show("Polska");
}
private: System::Void toolStripButton3_Click(System::Object^ sender, System::EventArgs^ e)
{
        MessageBox::Show("Deutschland");
}
```

Приклад 3.14 – Реалізація обробників пунктів меню ToolStrip

Таким чином, обидва приклади реалізують певні пункти меню, вибір яких виконується обробниками. При цьому властивості текстового і графічного меню можуть визначатися автоматично і змінюватись користувачем. Назви об’єктів меню та їх окремих пунктів формуються автоматично:

```
private: System::Windows::Forms::MenuStrip^ menuStrip1;
private: System::Windows::Forms::ToolStripMenuItem^ fileToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^ openToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^ saveToolStripMenuItem;
private: System::Windows::Forms::ToolStripMenuItem^ exitToolStripMenuItem;
private: System::Windows::Forms::ToolStrip^ toolStrip1;
private: System::Windows::Forms::ToolStripButton^ toolStripButton1;
private: System::Windows::Forms::ToolStripButton^ toolStripButton2;
private: System::Windows::Forms::ToolStripButton^ toolStripButton3;
```

3.3.9 Контрольні завдання

1. Пояснити окремо роль класів аплікації та форми у створенні Window Forms C++/CLI програми.
2. Пояснити послідовність запуску і функціонування простої Window Forms C++/CLI програми.
3. Пояснити порядок обробки подій у Window Forms C++/CLI - програмах.
1. Пояснити особливості підключення меню до головного вікна програми.
2. Розробити просту MFC-програму. Дослідити вплив параметрів функції *Create()* класу CFrameWnd на параметри головного вікна програми.
3. Розробити просту Window Forms C++/CLI. Реалізувати обробку подій KeyDown, Move, Resize.
4. Розробити просту Window Forms C++/CLI із обробкою повідомлень миші.
5. Розробити просту Window Forms C++/CLI із обробкою події таймера та. Реалізувати три таймери із виведенням часу із різними інтервалами.
6. Реалізувати Window Forms C++/CLI-програму із меню текстового та графічного типу. Реалізувати пункти меню у вигляді виведення вікон повідомлень.

4 СПІЛЬНІ ЕЛЕМЕНТИ КЕРУВАННЯ WINDOWS FORMS

Створюючи графічний інтерфейс (GUI) для настільних Windows-програм на .NET за допомогою C++/CLI, ви використовуєте загальні елементи керування Windows Forms.

Це готові до використання, інтерактивні компоненти, які значно спрощують розробку вікон, діалогів та інших елементів взаємодії з користувачем. Розглядайте їх як набір візуальних інструментів, подібних до кубиків LEGO, які легко комбінувати.

В основі цих елементів лежать керовані об'єкти .NET з простору імен System::Windows::Forms. C++/CLI забезпечує плавний зв'язок між вашим C++ кодом і цими .NET компонентами, надаючи вам переваги розвиненої екосистеми GUI .NET Framework безпосередньо у вашому C++ проєкті.

Ключовими особливостями є їхній наочний вигляд, реакція на дії користувача, гнучкість у налаштуванні (стиль, текст, поведінка), система обробки подій, логічна ієрархічна структура в межах вікон та підтримка засобів доступності.

Серед найпоширеніших елементів керування ви знайдете:

- для відображення інформації та введення даних: Label, TextBox, Button, PictureBox;
- для надання користувачеві можливості вибору: CheckBox, RadioButton, ComboBox, ListBox;
- для організації складних інтерфейсів: Panel, GroupBox, TabControl/TabPage, FlowLayoutPanel, TableLayoutPanel, SplitContainer;
- для представлення структурованих даних: ListView, TreeView, DataGridView;
- допоміжні елементи, такі як ProgressBar, TrackBar, NumericUpDown, DateTimePicker, панелі інструментів (ToolStrip), меню (MenuStrip), смуги стану (StatusStrip) та стандартні діалогові вікна.

Використовуючи ці елементи керування, ви можете зосередитися на логіці вашої програми, а не на ручному створенні кожного візуального компонента з нуля. Ваша робота полягає у виборі, розміщенні, стилізації цих елементів та написанні коду, який реагує на дії користувача через обробники подій.

4.1 Модальні і немодальні діалогові вікна

У розробці GUI на C++/CLI з використанням Windows Forms, модальні та немодальні діалогові вікна слугують важливим механізмом для комунікації з користувачем, візуалізації даних та отримання структурованої інформації. Ці вікна, подібно до основних вікон програми, є об'єктами класу System::Windows::Forms::Form, але використовуються для відображення допоміжного вмісту. Їх ключова відмінність полягає у тому, як вони впливають на виконання програми під час їхньої активності.

4.1.1 Модальне діалогове вікно (метод ShowDialog())

Коли викликається метод ShowDialog() для відображення модального вікна, воно призупиняє будь-яку подальшу взаємодію користувача з головним вікном програми (або іншими вікнами, що працюють в тому ж потоці). Користувач змушений спершу закрити модальне вікно, виконавши необхідні дії, такі як натискання кнопок "OK" або "Скасувати", перш ніж зможе продовжити роботу з основним інтерфейсом програми.

Модальні діалоги є стандартним підходом для:

– збору обов'язкової для подальшої роботи інформації від користувача (наприклад, вибір шляху для збереження файлу, введення логіна та пароля);

– відображення критично важливих повідомлень, що вимагають усвідомленого підтвердження користувачем перед продовженням виконання програми;

– реалізації невеликих, самостійних завдань, які потребують окремого вікна для взаємодії;

Процес закриття модальних діалогових вікон зазвичай ініціюється:

– натисканням кнопок, властивість DialogResult яких має конкретне значення, що сигналізує про вибір користувача (наприклад, System::Windows::Forms::DialogResult::OK для підтвердження, System::Windows::Forms::DialogResult::Cancel для відмови);

– програмним викликом методу Close() для об'єкта діалогової форми.

Після закриття модального вікна, метод ShowDialog() повертає значення властивості DialogResult натиснутої кнопки, надаючи викликаючому коду інформацію про те, як користувач взаємодіяв з діалогом.

Розглянемо приклад створення модального діалогового вікна.

По-перше, треба додати кнопку button з панелі інструментів Toolbox (рис. 4.1), для цього її потрібно перенести на вікно форми за допомогою перетя-

гування мишею (більш детально стосовно елементів керування у підрозділі 4.4), після чого вона з’явиться на формі (рис. 4.2).

Далі потрібно натиснути на кнопку 2 рази, щоб створився відповідний обробник події натискання на неї (рис. 4.3).

Після чого потрібно написати код обробника. Розглянемо приклад такого обробника (рис. 4.4).

```
MyForm^ modalForm = gcnew MyForm();
System::Windows::Forms::DialogResult result = modalForm->ShowDialog();
delete modalForm;
```

Приклад 4.1 – Реалізація обробника створення діалогового вікна

Рядок MyForm^ modalForm = gcnew MyForm(); оголошує змінну з іменем modalForm. Ця змінна є маркером (^), який може посилатися на об’єкт типу MyForm. Створює новий екземпляр класу MyForm у керованій купі (heap) .NET за допомогою gcnew. Це передбачає виділення пам’яті та виклик конструктора MyForm.

Рядок System::Windows::Forms::DialogResult result = modalForm->ShowDialog(); відображає форму, на яку посилається дескриптор modalForm, як модальне діалогове вікно – це означає, що користувач не зможе взаємодіяти з основною програмою, доки не закриє це діалогове вікно.

Останній рядок delete modalForm; необіхний для того, щоб звільнити вже виділену пам’ять для модального діалогового вікна.

Після натиснення кнопки button1, додається ще одне таке саме діалогове вікно.

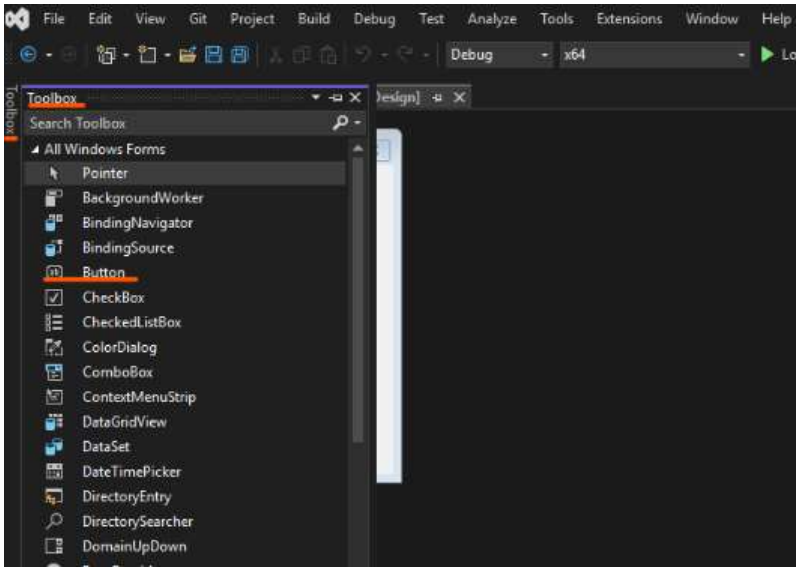


Рисунок 4.1 – Додавання елементу кнопки

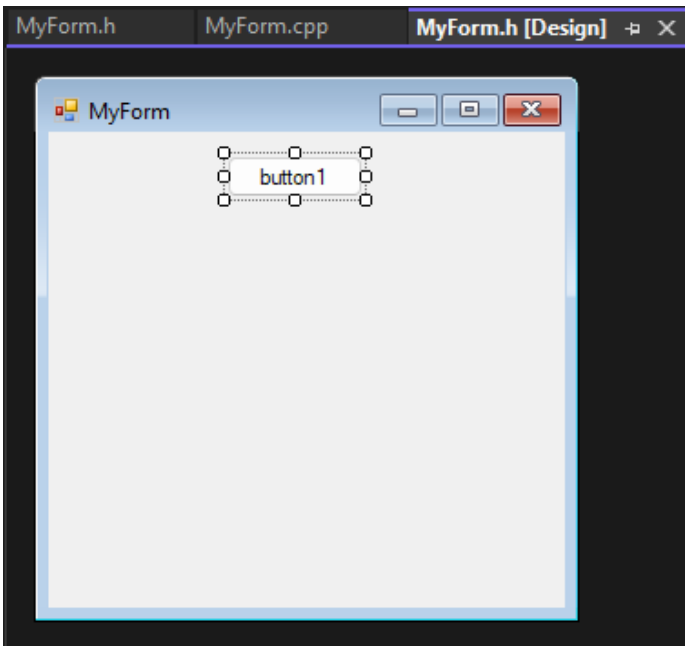


Рисунок 4.2 – Додана кнопка

```
    private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {

    }
    };
}
```

Рисунок 4.3 – Додавання обробника натискання кнопки

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {

    MyForm^ modalForm = gcnew MyForm();
    System::Windows::Forms::DialogResult result = modalForm->ShowDialog();
    delete modalForm; // Important to release managed resources
}
};
```

Рисунок 4.4 – Обробник для модального діалогового вікна

4.1.2 Немодальні діалогові вікна

На відміну від модальних, немодальні діалогові вікна, що відображаються шляхом виклику методу Show(), забезпечують паралельну роботу з основним вікном програми. Користувач зберігає можливість вільного переходу між немодальним діалогом та головним застосунком, а також іншими відкритими немодальними вікнами.

Сфера застосування немодальних діалогів включає:

– відображення допоміжної інформації, такої як контекстна довідка, панелі інструментів або індикатори стану виконання, які можуть бути необхідні користувачеві протягом сеансу роботи з основною програмою;

– візуалізація процесів, що виконуються у фоновому режимі, або динамічних даних, які потребують регулярного оновлення в реальному часі;

– забезпечення можливості багаторазової взаємодії користувача з елементами керування діалогового вікна без його обов'язкового закриття після кожної дії.

Процес закриття немодальних діалогових вікон може бути ініційований наступними способами:

– через стандартний елемент керування закриття вікна (піктограма X).

– шляхом активації елемента керування всередині діалогового вікна, який програмно викликає метод Close().

– за допомогою програмного виклику методу Close() з коду основного вікна програми.

Важливо відзначити, що немодальні діалогові вікна не повертають значення типу DialogResult безпосередньо після закриття. Для отримання даних, що містяться в немодальному діалозі, використовуються наступні підходи:

– надання публічного доступу до внутрішніх елементів керування діалогу (наприклад, об'єктів класів TextBox або ListBox), хоча це може порушувати принципи інкапсуляції;

– реалізація публічних властивостей або методів у класі немодального діалогового вікна, які забезпечують контрольований доступ до необхідних даних;

– використання механізму обробки подій, що дозволяє немодальному діалоговому вікну ініціювати події, які сповіщають основну програму про зміни у його внутрішньому стані.

Розглянемо приклад створення немодального діалогового вікна.

Для цього, проведемо зміну обробнику для діалогового вікна (рис. 4.5).

Спочатку додамо рядок private: MyForm^ nonModalForm = nullptr; – у C++/CLI оголошує приватну змінну-член у класі, яка буде містити дескриптор екземпляра іншого класу форми під назвою MyForm.

Далі пропишемо обробник:

```
if (nonModalForm == nullptr || nonModalForm->IsDisposed) {
        nonModalForm = gcnew MyForm();
        nonModalForm->Show(); }
else {  nonModalForm->Activate(); }
```

Приклад 4.2 – Реалізація обробника створення немодального діалогового вікна

Умова if перевіряє, чи змінна nonModalForm наразі не вказує на дійсний об’єкт MyForm (вона nullptr), чи об’єкт MyForm, на який вона вказувала, уже видалено.

Рядок nonModalForm = gcnew MyForm(); створює новий об’єкт MyForm і призначає його маркер змінній nonModalForm. Це гарантує, що nonModalForm тепер вказує на дійсне, щойно створене немодальне діалогове вікно.

Show() – це метод класу System::Windows::Forms::Form. Він відображає форму як немодальне вікно. Як ми обговорювали, немодальні вікна не блокують взаємодію з основною програмою.

Activate() – також метод класу System::Windows::Forms::Form. Він виводить наявне вікно на передній план і надає йому фокус. Якщо немодальне діалогове вікно було приховане або за іншими вікнами, це зробить його видимим і активним.

```
private: MyForm^ nonModalForm = nullptr;


private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {

    if (nonModalForm == nullptr || nonModalForm->IsDisposed) {
        nonModalForm = gcnew MyForm();
        nonModalForm->Show();
    }
    else {
        nonModalForm->Activate();
    }
}
```

Рисунок 4.5 – Обробник створення немодального діалогового вікна

Основні відмінності між модальними та немодальними діалоговими вікнами представлено у табл. 4.1.

4.2 Обробка повідомлень діалогових вікон та їх ініціалізація

4.2.1 Ініціалізація діалогових вікон

У C++/CLI для Windows Forms процес ініціалізації діалогових вікон передбачає налаштування їхнього первинного стану перед тим, як вони стануть доступними для взаємодії з користувачем. Це охоплює такі завдання, як встановлення вихідних значень для елементів керування, завантаження необхідних даних та загальну підготовку інтерфейсу.

Таблиця 4.1 – Основні відмінності між модальними та немодальними діалоговими вікнами

Особливості	Модальний діалог (ShowDialog())	Немодальний діалог (Show())
Блокування	Блокує взаємодію з власником	Не блокує взаємодію
Потік користувача	Примушує до негайної взаємодії	Дозволяє перемикатися вперед і назад між вікнами
Використання	Основне введення, критичні повідомлення	Допоміжна інформація, поточні завдання
Закриття	Часто за допомогою кнопок DialogResult	Часто за допомогою кнопки закриття або Close()
Значення що повертається	Повертає DialogResult	Зазвичай потрібні публічні властивості/методи/події

Ініціалізація діалогових вікон у C++/CLI для Windows Forms – це поетапний процес, який може бути розбитим на такі кроки:

– візуальне створення макета: за допомогою дизайнера Visual Studio ви оформлюєте зовнішній вигляд діалогового вікна, розміщуючи на ньому потрібні елементи керування (кнопки, поля введення, написи тощо) та налаштовуючи їхні властивості (текст, розміри, розташування, обробники подій). Visual Studio автоматично генерує відповідний C++/CLI код у .h-файлі.

– створення об'єкта діалогу: у тому місці коду, де потрібно показати діалогове вікно, створюється екземпляр його класу. Наприклад, для класу MyForm це виглядає як MyForm^ dialog = gcnew MyForm();.

– додаткове налаштування: перед показом можна програмно змінити властивості створеного об'єкта діалогу, наприклад, встановити заголовок або передати початкові дані:

```
dialog->Text = "Введіть дані";
dialog->someLabel->Text = "Будь ласка, введіть ваше ім'я:";.
```

Для відображення діалогового вікна існує два способи:

– модально (ShowDialog()): діалогове вікно блокує основний потік виконання програми до моменту його закриття користувачем. Це типовий спосіб для отримання вхідних даних або підтвердження дій. Метод повертає

System::Windows::Forms::DialogResult, що вказує на спосіб закриття (наприклад, OK або Cancel).

```
if (dialog->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
// Дії після натискання "OK"
System::String^ enteredText = dialog->someTextBox->Text;
// ... обробка введеного тексту ...
} else {
// Дії, якщо діалог закрито інакше (наприклад, "Cancel")
}
```

– немодально (Show()): діалогове вікно відображається, не блокуючи роботу з іншими вікнами програми.

```
dialog->Show();
// Код, що виконується відразу після показу діалогу
```

– реагування на події: для забезпечення інтерактивності необхідно обробляти події, що відбуваються в діалоговому вікні (наприклад, кліки на кнопках, зміни в текстових полях). Обробники подій можна підключити візуально в дизайнері або програмно. Visual Studio створить заготовки функцій обробників у .cpp-файлі, де потрібно буде реалізувати їхню логіку;

– очищення ресурсів: після закриття діалогового вікна важливо звільнити зайняті ним ресурси. Оскільки використовуються керовані об'єкти (gcnew), автоматичне звільнення пам'яті забезпечується збирачем сміття. Проте, у випадках використання некерованих ресурсів (що рідко трапляється у простих діалогових вікнах), може знадобитися реалізація шаблону Dispose.

4.2.2 Обробка повідомлень діалогових вікон

Функція MessageBox::Show() у C++/CLI для Windows Forms є статичним методом класу System::Windows::Forms::MessageBox. Її основне призначення – відображення стандартного діалогового вікна з повідомленням для користувача. Це простий та зручний спосіб інформувати користувача про події, попереджати про можливі проблеми або запитувати підтвердження дій.

Статичний метод Show() класу System::Windows::Forms::MessageBox у C++/CLI Windows Forms призначений для виведення на екран стандартного модального діалогового вікна з метою комунікації з користувачем. Він забезпечує простий та ефективний механізм для відображення інформаційних повідомлень, попереджень про помилки або запитів на підтвердження певних дій.

При використанні MessageBox::Show() створюється та відображається діалогове вікно, яке може містити такі елементи: основний текст повідомлення для користувача, заголовок діалогового вікна, стандартний набір інтерактивних кнопок (наприклад, "OK", "Скасувати", "Так", "Ні"), графічну іконку для візуального представлення типу повідомлення (за бажанням), кнопку, яка буде активною при відкритті діалогу (за потреби), додаткові налаштування відображення та посилання на вікно-власник.

При відображенні модального діалогового вікна користувач бачить:

- основний зміст повідомлення, призначений для ознайомлення;
- текст у заголовку вікна, що ідентифікує його призначення;
- стандартні елементи керування у вигляді кнопок (наприклад, "OK", "Скасувати"), що надають варіанти дій;
- за потреби, графічне зображення (іконку), що візуально класифікує повідомлення (наприклад, як помилку чи попередження);
- можливість попереднього вибору однієї з кнопок за замовчуванням;
- додаткові параметри, що можуть змінювати спосіб відображення та поведінку діалогу;
- необов'язкове визначення вікна-власника, що може вплинути на розміщення діалогу відносно інших вікон програми.

Функція MessageBox::Show() має кілька перевантажень, які дозволяють викликати її з різним набором параметрів:

```
static System::Windows::Forms::DialogResult Show(System::String^ text);
static System::Windows::Forms::DialogResult Show(System::String^ text, System::String^ caption);
static System::Windows::Forms::DialogResult Show(System::String^ text, System::String^ caption, System::Windows::Forms::MessageBoxButtons buttons);
static System::Windows::Forms::DialogResult Show(System::String^ text, System::String^ caption, System::Windows::Forms::MessageBoxButtons buttons, System::Windows::Forms::MessageBoxIcon icon);
static System::Windows::Forms::DialogResult Show(System::String^ text, System::String^ caption, System::Windows::Forms::MessageBoxButtons buttons, System::Windows::Forms::MessageBoxIcon icon, System::Windows::Forms::MessageBoxDefaultButton defaultButton).
```

Функція MessageBox::Show() приймає наступні аргументи:

- text (System::String^): обов'язковий, представляє текстове повідомлення, яке побачить користувач у діалоговому вікні;
- caption (System::String^): Необов'язковий, задає текст у заголовку вікна повідомлення. Якщо його не вказати, буде використано стандартний заголовок (назва вашої програми або слово "Повідомлення");
- buttons (System::Windows::Forms::MessageBoxButtons): необов'язковий, визначає, які кнопки будуть присутні в діалоговому вікні (наприклад, лише

"OK", "OK" та "Скасувати", "Так" та "Ні"). Використовуються значення з переліченння MessageBoxButtons, а за замовчуванням відображається лише кнопка "OK";

– icon (System::Windows::Forms::MessageBoxIcon): необов'язковий, вказує, яку іконку відобразити поруч із повідомленням (наприклад, іконку помилки, попередження, питання, інформації). Використовуються значення з переліченння MessageBoxIcon, а за замовчуванням іконка не відображається;

– defaultButton (System::Windows::Forms::MessageBoxDefaultButton): необов'язковий, визначає, яка з кнопок у діалоговому вікні отримає початковий фокус клавіатури. Використовуються значення з переліченння MessageBoxDefaultButton, а за замовчуванням фокус отримує перша кнопка;

– options (System::Windows::Forms::MessageBoxOptions): необов'язковий, надає додаткові можливості для налаштування відображення діалогового вікна, наприклад, для підтримки мов з написанням справа наліво;

– owner (System::Windows::Forms::IWin32Window^): необов'язковий, дозволяє вказати вікно, яке буде власником цього діалогового вікна. Це може вплинути на розташування діалогу та його поведінку щодо інших вікон програми.

Наведемо приклади використання. Просте повідомлення з кнопкою "OK":

```
MessageBox::Show("Операцію виконано успішно.");
```

Повідомлення з заголовком:

```
MessageBox::Show("Сталася помилка при збереженні файлу.", "Помилка");
```

Повідомлення з кнопками "Так" та "Ні" та іконкою питання:

```
if (MessageBox::Show("Ви впевнені, що хочете видалити цей елемент?", "Підтвердження", MessageBoxButtons::YesNo, MessageBoxIcon::Question) == System::Windows::Forms::DialogResult::Yes) {
    // Код для видалення елемента }
```

Повідомлення з кнопками "OK" та "Скасувати" та іконкою попередження:

```
if (MessageBox::Show("Деякі дані будуть втрачені. Продовжити?", "Попередження", MessageBoxButtons::OKCancel, MessageBoxIcon::Warning) == System::Windows::Forms::DialogResult::OK) {
    // Код для продовження з втратою даних }
```

4.3 Використання діалогових вікон як головних

Застосування діалогових вікон як основних у C++/CLI для Windows Forms – це хоч і не стандартний, але здійсненний спосіб організації програми, де замість звичного головного вікна (Form) використовуються діалоги. Діалогові вікна, які зазвичай слугують для короткочасного обміну даними з користувачем, можуть стати центральним елементом програми, особливо якщо її логіка побудована навколо одноразових або послідовних інтеракцій.

Один із простих методів реалізації такого підходу полягає у створенні екземпляра потрібного діалогового вікна у функції Main() та його модальному відображенні за допомогою ShowDialog(). У цьому сценарії закриття діалогового вікна призведе до завершення роботи всієї програми. Для цього потрібно модифікувати файл .cpp, а саме функцію Main, де вставити наступні рядки коду (рис. 4.6):

```
Project2::MyForm^ mainDialog = gcnew Project2::MyForm();
Application::Run(mainDialog);
```

Або:

```
Project2::MyForm^ mainDialog = gcnew Project2::MyForm();
mainDialog->ShowDialog();.
```

```
#include "MyForm.h"

using namespace System; // підключення простору імен System
using namespace System::Windows::Forms; // підключення простору імен Windows::Forms

[STAThreadAttribute]        // встановлення атрибутів однопоточного проекту

void Main(array <String^>^ args)    // функція main із командним рядком args
{
    Application::EnableVisualStyles();  // вмикання стилів візуалізації
    Application::SetCompatibleTextRenderingDefault(false); // сумісні

    /*Project2::MyForm form;
    Application::Run(%form);*/

    Project2::MyForm^ mainDialog = gcnew Project2::MyForm();
    Application::Run(mainDialog);
}
```

Рисунок 4.6 – Змінений код функції Main

Якщо ваша програма складається з кількох послідовних кроків, кожен з яких представлений діалоговим вікном, ви можете відображати їх один за одним у функції Main() або в обробниках подій, тоді код матиме наступний вигляд:

```
// Program.cpp

#include "Dialog1.h" // створений окремо діалог 1
#include "Dialog2.h" // створений окремо діалог 1

using namespace System;
using namespace System::Windows::Forms;

[STAThreadAttribute]
int main(array<System::String ^> ^args) {
   Application::EnableVisualStyles();
   Application::SetCompatibleTextRenderingDefault(false);
   Dialog1^ dialog1 = gcnew Dialog1();
   if (dialog1->ShowDialog() == DialogResult::OK) {
      // Отримання даних з dialog1
      String^ dataFromDialog1 = dialog1->someTextBox->Text;
      delete dialog1; // Очищення ресурсів
      Dialog2^ dialog2 = gcnew Dialog2(dataFromDialog1); // Передача даних у наступний діалог
      if (dialog2->ShowDialog() == DialogResult::OK) {
         // Отримання даних з dialog2
         String^ finalResult = dialog2->resultLabel->Text;
         MessageBox::Show("Результат: " + finalResult);
      }
      delete dialog2; // Очищення ресурсів
   }
   delete dialog1; // Очищення ресурсів, якщо перший діалог не був успішно закритий
   return 0; }
```

Приклад 4.3 – Реалізація діалогових вікон як головних

Ключовими ідеями, які слід врахувати, при використанні діалоговий вікон як основних елементів інтерфейсу є наступниі:

– завершення програми: після закриття головного діалогового вікна, яке було відкрито в модальному режимі, робота програми завершується;

– керування через події: уся поведінка та логіка програми будуть визначатися реакцією на події, що відбуваються в межах цих діалогових вікон (наприклад, кліки на кнопках, введення тексту);

– обмежені можливості: діалогові вікна зазвичай мають менший набір стандартних елементів інтерфейсу порівняно зі звичайними вікнами, наприклад, у них може не бути головного меню чи панелі інструментів;

– вплив на користувацький досвід: орієнтація лише на діалогові вікна може зробити інтерфейс менш гнучким і звичним для користувачів, які очікують стандартної структури застосунку Windows. Важливо ретельно оцінити, чи відповідає такий підхід потребам вашої програми;

– налаштування DialogResult: для кнопок, які відповідають за закриття діалогового вікна (наприклад, "OK", "Скасувати" або інші), правильно встановлено властивість DialogResult. Це необхідно для визначення результату виклику

методу ShowDialog(). Налаштувати цю властивість можна у візуальному редакторі Visual Studio;

– очищення ресурсів: хоча збірка сміття автоматично звільняє пам'ять, яку займають діалогові вікна, рекомендується явно видаляти їхні екземпляри за допомогою delete dialog; після використання для більш передбачуваного керування ресурсами.

4.4 Діалогові вікна та стандартні елементи керування

Розробимо класифікацію загальних елементів керування.

Елементи керування, що відповідають за основний функціонал введення та виведення даних є наступними:

– Label: відображає статичний текст, доступний лише для читання, для інформації чи інструкцій;

– TextBox: дозволяє користувачам вводити та редагувати однорядковий або багаторядковий текст;

– Button: запускає дію, коли натискає користувач;

– PictureBox: відображає зображення в різних форматах.

Елементи, що відповідають за керування вибором:

– CheckBox: дозволяє користувачам вибирати або скасовувати окремі параметри (у групі можна вибрати кілька варіантів);

– RadioButton: дозволяє користувачам вибрати один варіант із взаємовиключної групи;

– ComboBox: представляє розкривний список, де користувачі можуть вибрати один елемент, часто з частиною тексту, яку можна редагувати;

– ListBox: відображає список елементів, що дозволяє користувачам вибрати один або кілька елементів.

Для того, щоб керувати контейнером існують такі елементи (приймають участь у організації або групуванні інших елементів керування):

– Panel: основний контейнер, який може групувати інші елементи керування та забезпечувати прокручування;

– GroupBox: подібно до панелі, але з рядком заголовка для візуального групування пов’язаних елементів керування;

– TabControl та TabPage: створює інтерфейс із вкладками для організації вмісту на окремих сторінках;

– FlowLayoutPanel: впорядковує елементи керування в напрямку потоку (горизонтально чи вертикально), за потреби робить перенесення їх на наступний рядок/стовпець;

– TableLayoutPanel: впорядковує елементи керування у структурі, подібній до сітки, визначеній рядками та стовпцями;

– SplitContainer: дозволяє користувачеві змінювати розмір двох панелей, що містяться, за допомогою рухомої панелі.

Елементи керування відображенням даних є наступні:

– ListView: відображає список елементів із необов'язковими стовпцями для відображення деталей. Підтримує різні режими перегляду (список, деталі, іконки);

– TreeView: відображає ієрархічні дані в деревоподібній структурі з розгортаними та згортаними вузлами;

– DataGridView: надає сітку з можливістю налаштування для відображення та редагування табличних даних.

До інших дуже корисних елементів керування можна віднести:

– ProgressBar: візуально вказує на хід тривалої операції;

– TrackBar: дозволяє користувачам вибирати значення в межах діапазону, пересуваючи повзунок;

– NumericUpDown: дозволяє користувачам вводити чи вибирати числове значення в заданому діапазоні за допомогою кнопок збільшення/зменшення;

– DateTimePicker: надає інтерфейс, схожий на календар, для вибору дат і часу;

– ToolStrip & ToolStripItem (включаючи ToolStripButton, ToolStripLabel тощо): створює настроювані панелі інструментів із кнопками, мітками, розкривними меню тощо;

– MenuStrip & ToolStripMenuItem: створює традиційні панелі меню програми;

– StatusStrip & ToolStripStatusLabel: відображає інформацію про стан у нижній частині вікна.

Також слід зазначити, що існують і додаткові елементи керування діалоговим вікном, такі як OpenFileDialog, SaveFileDialog, FolderBrowserDialog, ColorDialog, FontDialog – попередньо створені діалогові вікна для таких типових завдань, як відкриття/збереження файлів, вибір папок, кольорів і шрифтів.

Далі розглянемо кожен з цих елементів окремо.

4.4.1 Елемент Label

Робота з елементом Label у Windows Forms на C++/CLI передбачає кілька кроків. Спочатку Label додається на форму візуально у Visual Studio Designer шляхом перетягування з панелі елементів та налаштування його розміру і положення (рис. 4.7). Потім його зовнішній вигляд та поведінка конфігуруються через вікно властивостей (рис. 4.8) або програмно, змінюючи такі параметри, як текст (Text), автоматичне змінення розміру (AutoSize), вирівнювання тексту (TextAlign), шрифт (Font), кольори (ForeColor, BackColor), стиль межі (BorderStyle), видимість (Visible), розташування (Location), розмір (Size), прив'язка до країв (Anchor) та спосіб прикріплення (Dock). Для динамічної зміни властивостей Label у коді використовується його ім'я. Хоча Label є статичним елементом, він успадковує події, які за потреби можна обробляти.

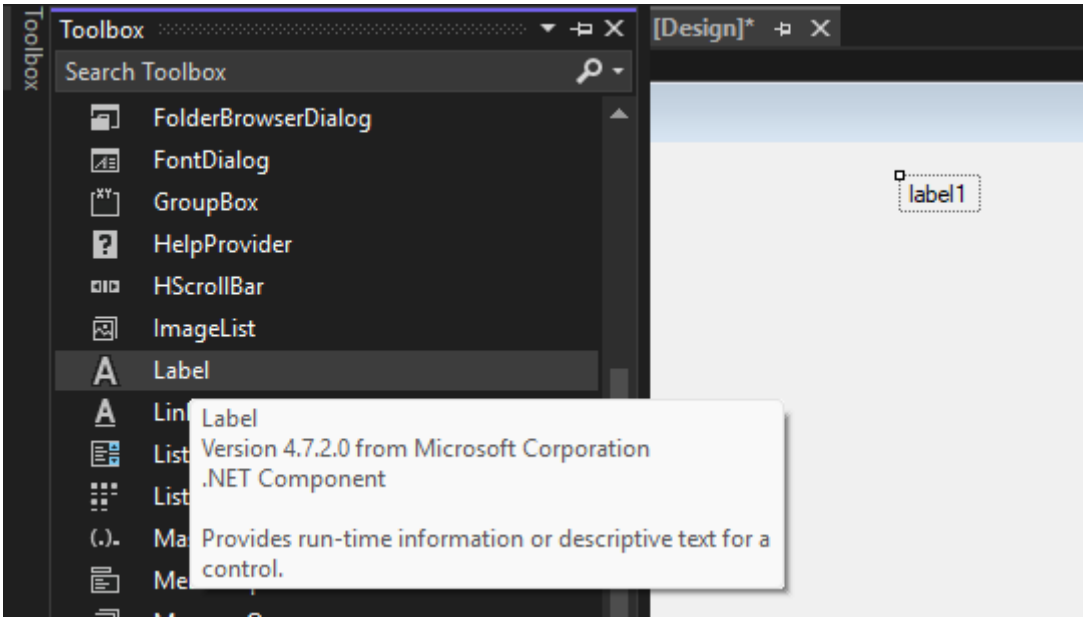


Рисунок 4.7 – Додавання елементу Label

Також програмна зміна тексту Label вилгядає наступним чином:

```
this->label1->Text = "Новий текст мітки";
```

Вирівнювання тексту по центру:

```
this->label1->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;
```

Зміна шрифту:

```
this->label1->Font = gcnew System::Drawing::Font("Arial", 12, System::Drawing::FontStyle::Bold);
```

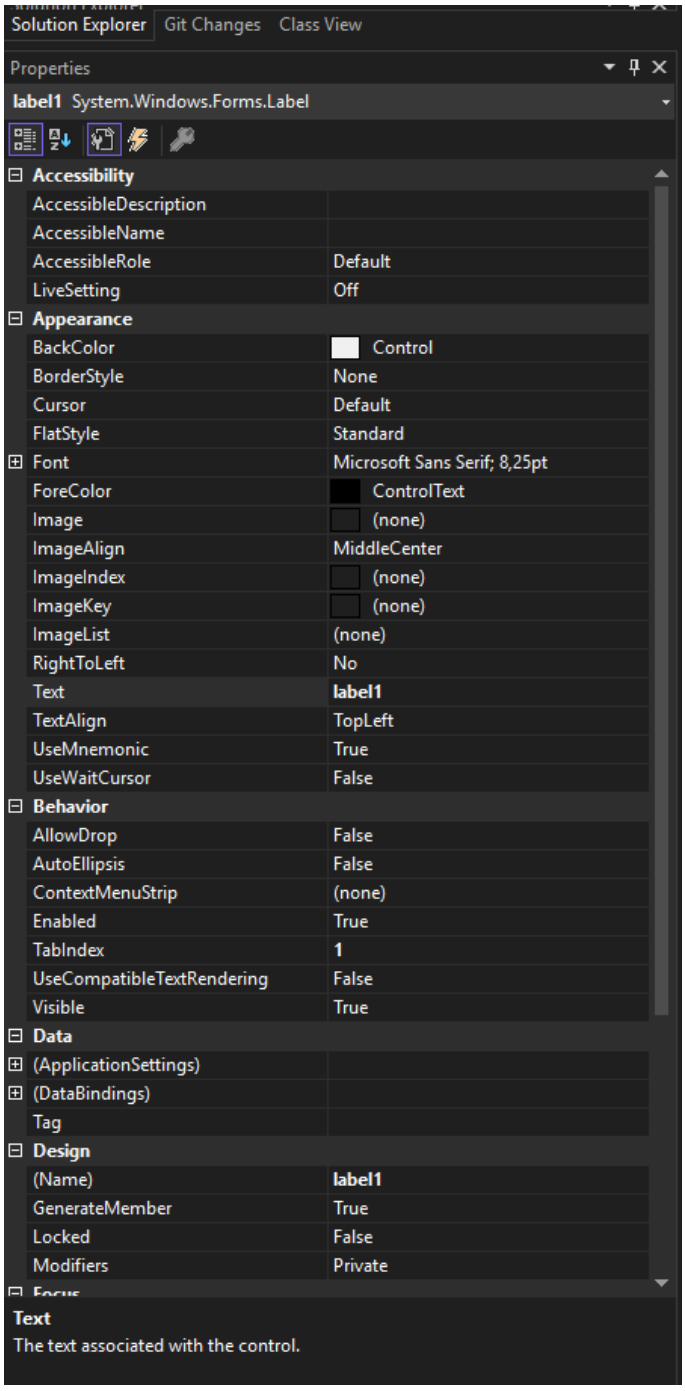


Рисунок 4.8 – Властивості елементу Label

Зміна кольору тексту на червоний:

```
this->label1->ForeColor = System::Drawing::Color::Red;
```

Зміна кольору фону на жовтий:

```
this->label1->BackColor = System::Drawing::Color::Yellow;
```

Встановлення одинарної межі:

```
this->label1->BorderStyle = System::Windows::Forms::BorderStyle::FixedSingle;
```

Для отримання тексту з Label можна використати наступний код:

```
System::String^ currentText = this->label1->Text;
```

Або перевірити, чи відображається Label:

```
if (this->label1->Visible) { // ... виконати якісь дії ... }.
```

4.4.2 Елемент TextBox

Для забезпечення можливості введення та редагування тексту в C++/CLI застосунках Windows Forms використовується елемент TextBox. Додавання TextBox на форму здійснюється у візуальному конструкторі Visual Studio шляхом перетягування з панелі елементів та налаштування його початкових розмірів і положення (рис. 4.9). Подальше налаштування контролу відбувається через зміну його властивостей у вікні Properties або програмно (рис. 4.10). Ключові властивості включають текст (Text), режим відображення кількох рядків (Multiline), наявність смуг прокрутки (ScrollBars), можливість редагування (ReadOnly), відображення символів пароля (PasswordChar), обмеження довжини введеного тексту (MaxLength), вирівнювання тексту (TextAlign), параметри шрифту (Font), кольори тексту та фону (ForeColor, BackColor), стиль межі (BorderStyle), стан активності (Enabled), видимість (Visible), координати розташування (Location), розміри (Size), правила прив'язки до контейнера (Anchor) та спосіб його прикріплення (Dock).

Команда отримання тексту з TextBox:

```
System::String^ enteredText = this->textBox1->Text;.
```

Встановлення тексту в TextBox:

```
this->textBox1->Text = "Початкове значення";.
```

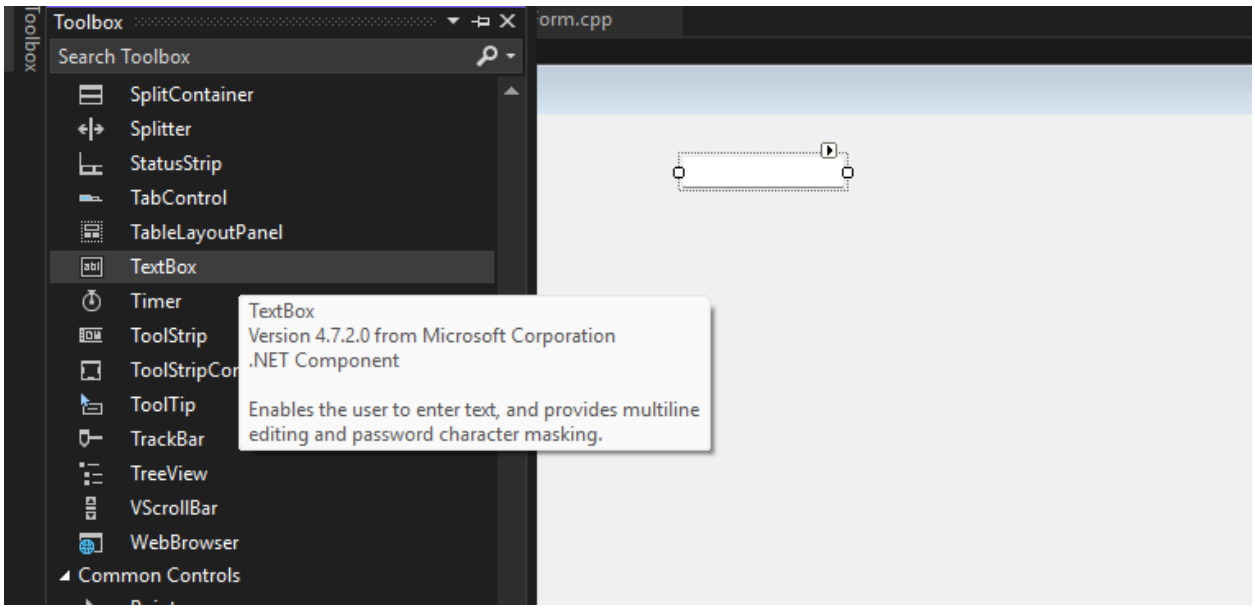


Рисунок 4.9 – Додавання елементу TextBox

Рисунок 4.10 – Властивості елементу TextBox

TextBox має кілька корисних подій, які ви можете обробляти для реагування на дії користувача (рис. 4.11). Для того, щоб подивитися на всі можливі події, потрібно натиснути на клавішу блискавки у вікні Properties, після чого, обравши подію, потрібно натиснути праворуч від неї 2 рази, щоб створився відповідний обробник.

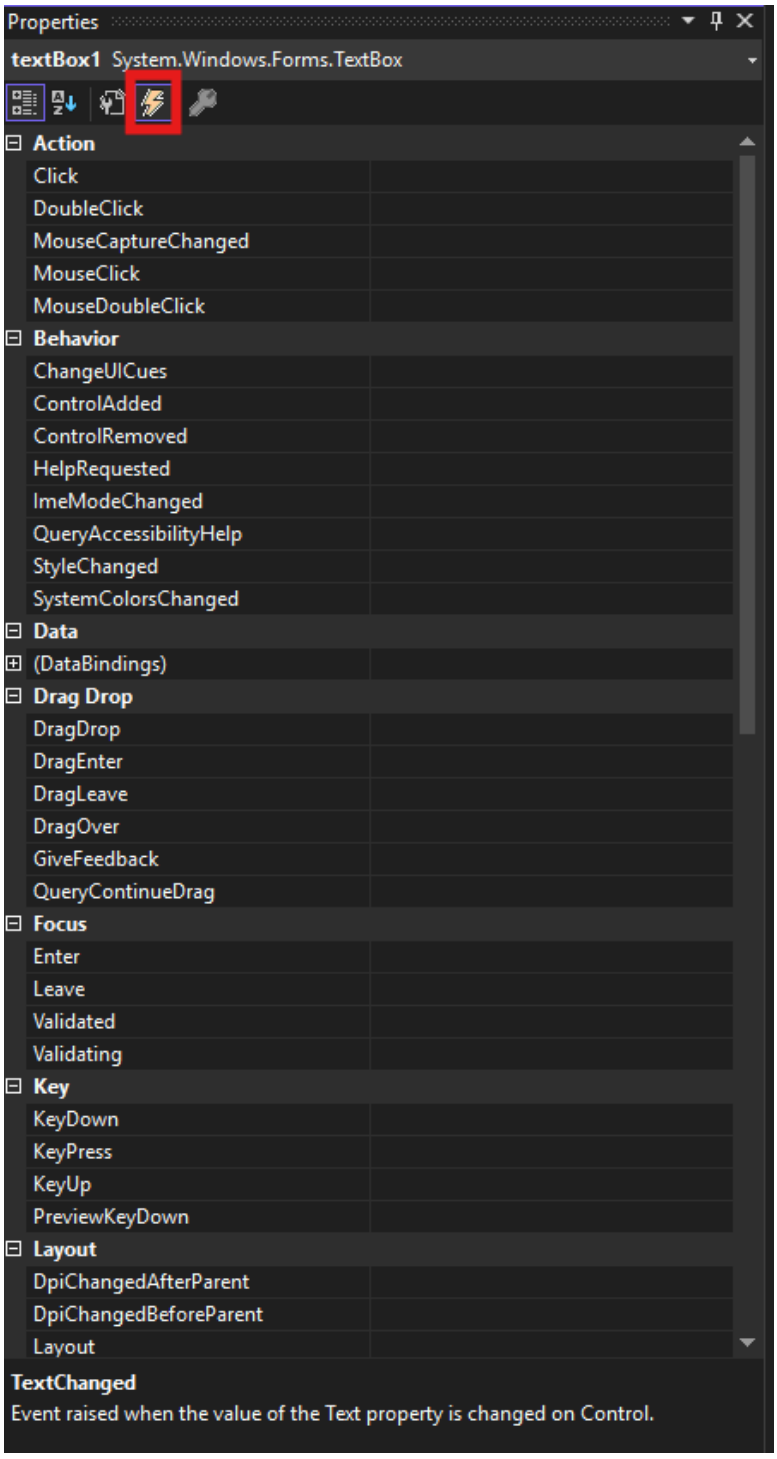


Рисунок 4.11 – Події для TextBox

Одна з таких корисних подій TextChanged – відбувається при кожній зміні тексту в TextBox користувачем або програмно. Це дуже часто використовувана подія для відстеження введення:

```
private: System::Void textBox1_TextChanged(System::Object^ sender, System::EventArgs^ e) {
   System::String^ currentText = this->textBox1->Text;
   // ... виконати якісь дії на основі введеного тексту ... }
```

Приклад 4.4 – Реалізація події TextChanged

KeyPress – відбувається при натисканні клавіші, коли TextBox має фокус. Надає інформацію про натиснуту клавішу і дозволяє відмінити її обробку:

```
private: System::Void textBox1_KeyPress(System::Object^ sender, System::Windows::Forms::KeyPressEventArgs^ e) {
   if (e->KeyChar == (wchar_t)Keys::Enter) {
      // Обробка натискання Enter
      System::String^ enteredText = this->textBox1->Text;
      // ... виконати якісь дії ...
      e->Handled = true; // Заборонити подальшу обробку Enter (наприклад, додавання нового рядка в однорядковому TextBox)  } }
```

Приклад 4.4 – Реалізація події KeyPress

Також існують і інші події, наприклад, KeyDown та KeyUp – ці події фіксують моменти натискання та відпускання клавіш відповідно, коли TextBox є активним для введення. Вони надають розширені відомості про натиснуту клавішу, включаючи її віртуальний код.

Події GotFocus та LostFocus. Подія GotFocus відбувається, коли елемент TextBox стає активним і готовим до введення (отримує фокус). Подія LostFocus відбувається, коли TextBox втрачає фокус введення, наприклад, при переході до іншого елемента керування.

Наприклад, нехай існують елементи Label та TextBox. Необхідно встановити текст з TextBox у поле Label, тоді обробник матиме такий вигляд:

```
private: System::Void buttonProcess_Click(System::Object^ sender, System::EventArgs^ e) {
   System::String^ inputText = this->textBoxInput->Text;
   this->labelOutput->Text = "Ви ввели: " + inputText; }
```

4.4.3 Елемент Button

Для створення інтерактивних елементів керування у C++/CLI застосунках Windows Forms використовується кнопка (Button). Її розміщують на формі у ві-

зуальному конструкторі Visual Studio, перетягуючи з панелі елементів та визначаючи початкові розміри та положення (рис. 4.12). Подальше налаштування здійснюється через вікно Properties (рис 4.13) або програмно, де можна змінити текст кнопки (Text), її стан активності (Enabled), видимість (Visible), розташування (Location), розміри (Size), правила прив'язки (Anchor), спосіб прикріплення (Dock), параметри шрифту (Font), кольори (ForeColor, BackColor), зображення (Image), спосіб його відображення відносно тексту (ImageAlign, TextImageRelation), стиль (FlatStyle), можливість навігації за допомогою Tab (TabStop, TabIndex) та результат для діалогових вікон (DialogResult).

Основна дія, пов'язана з кнопкою, обробляється через подію Click. Ця подія генерується при кліку миші на кнопці або при натисканні клавіш Enter чи пробіл, коли кнопка є активною. Саме в обробнику події Click розміщується код, який визначає реакцію програми на натискання кнопки. Окрім цієї ключової події, кнопка також підтримує інші події (рис. 4.14), успадковані від Control, такі як події миші (MouseEnter, MouseLeave, MouseDown, MouseUp) та події фокуса й клавіатури (GotFocus, LostFocus, KeyDown, KeyUp), які можуть бути корисними для розширення функціональності.

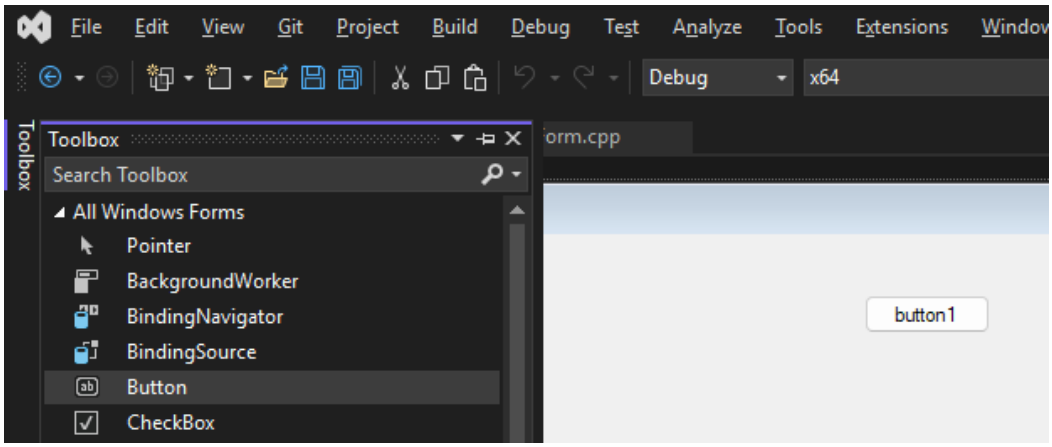


Рисунок 4.12 – Додавання елементу Button

Наприклад, є форма з текстовим полем (textBox1) та кнопкою (buttonAction). При натисканні на кнопку вміст текстового поля відобразиться у вікні повідомлення:

```
private: System::Void buttonAction_Click(System::Object^ sender, System::EventArgs^ e) {
  System::String^ inputText = this->textBox1->Text;
  System::Windows::Forms::MessageBox::Show("Введено: " + inputText);}
```

Приклад 4.5 – Реалізація події натискання кнопки

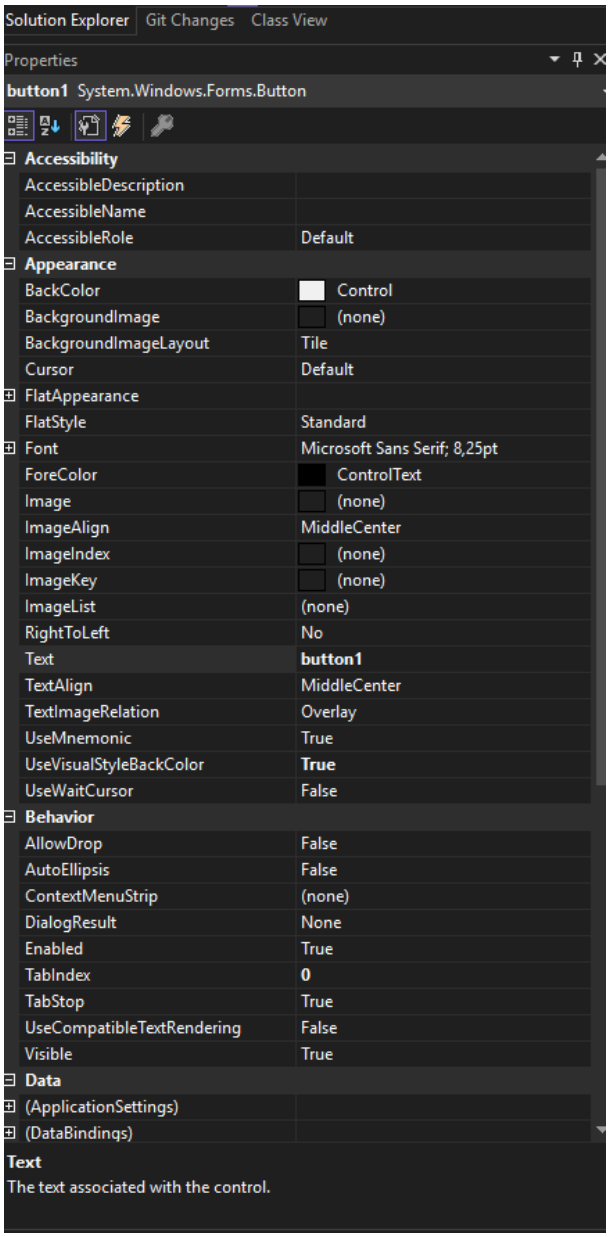


Рисунок 4.13 – Властивості елементу Button

4.4.4 Елемент PictureBox

Для відображення зображень у Windows Forms застосунках на C++/CLI використовується елемент Picture Box. Його розміщують на формі у візуальному конструкторі Visual Studio, перетягуючи з панелі елементів та визначаючи початкові розміри та положення (рис. 4.15). Налаштування Picture Box включає встановлення зображення через властивість Image (можливе завантаження з файлу або ресурсу), вибір режиму його масштабування та розташування за допомогою SizeMode (наприклад, Normal, StretchImage, AutoSize, CenterImage, Zoom), визначення стилю межі (BorderStyle), кольору фону (BackColor), стану

активності (Enabled), видимості (Visible), координат розташування (Location), розмірів (Size), правил прив'язки до контейнера (Anchor) та способу його прикріплення (Dock). Події показані на рис. 4.16.

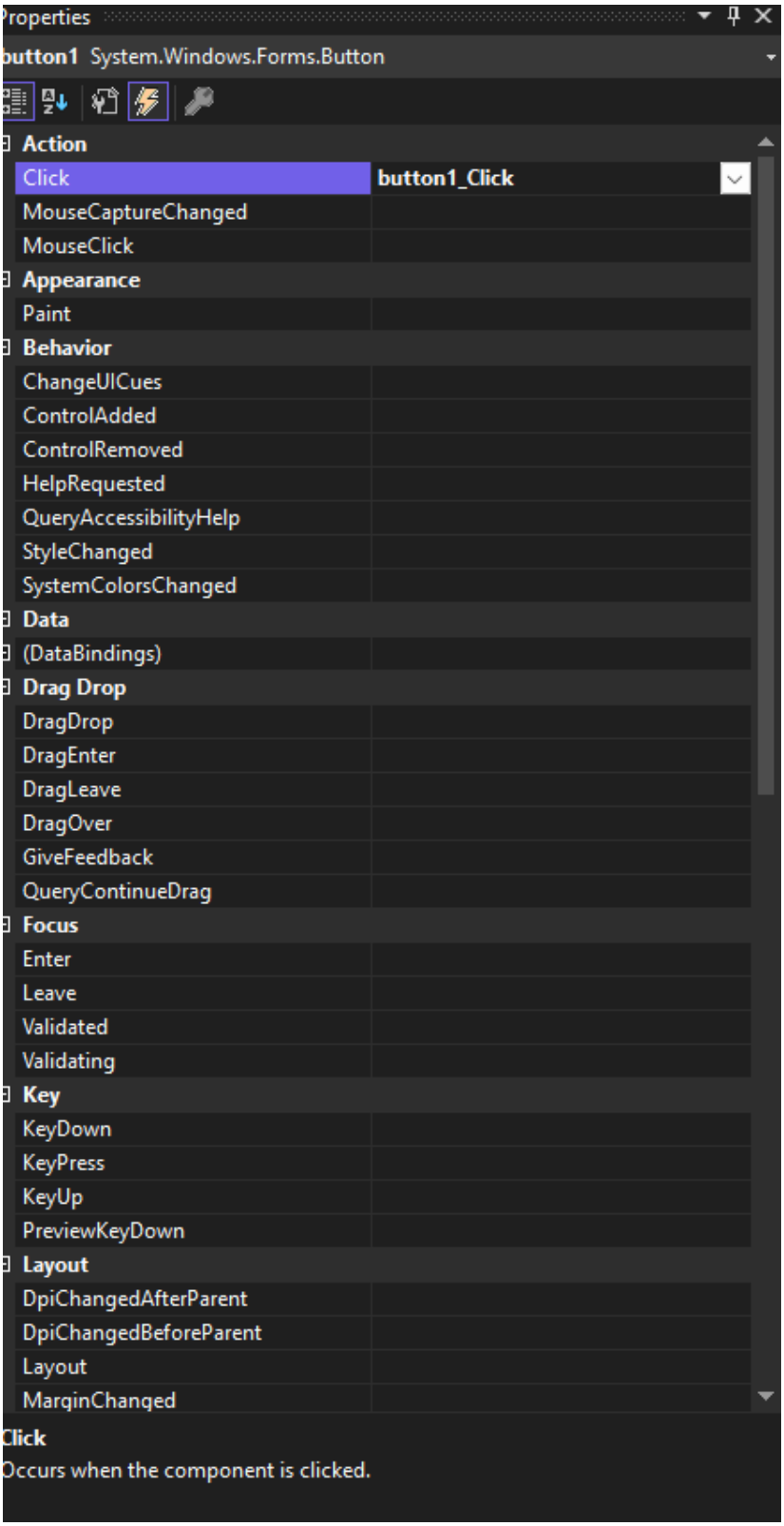


Рисунок 4.14 – Події для Button

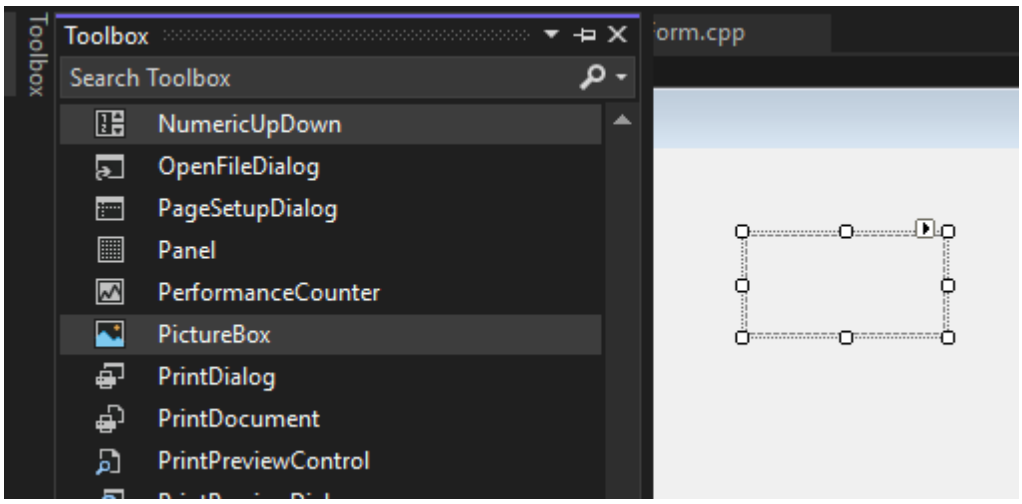


Рисунок 4.15 – Додавання елементу PictureBox

Image встановлює або отримує зображення, яке відображається в Picture Box. Ви можете завантажити зображення з файлу, ресурсу або іншого джерела. Завантаження зображення з файлу відбувається наступним чином:

```
this->pictureBox1->Image = System::Drawing::Image::FromFile("диск:\\шлях\\до\\зображення.png");
```

Встановлення зображення з ресурсу (припустимо, ресурс називається myImage):

```
this->pictureBox1->Image = Properties::Resources::myImage;
```

Наприклад, існує форма, з кнопкою (buttonLoad) та Picture Box (pictureBoxDisplay). При натисканні кнопки потрібно відобразити зображення в Picture Box:

```
private: System::Void buttonLoad_Click(System::Object^ sender, System::EventArgs^ e) {
        this->pictureBoxDisplay->Image = System::Drawing::Image::FromFile("диск:\\шлях\\до\\зображення.png");
        this->pictureBoxDisplay->SizeMode = System::Windows::Forms::PictureBoxSizeMode::Zoom;    }
```

Приклад 4.6 – Реалізація події додавання зображення у Picture Box

4.4.5 Елемент CheckBox

У Windows Forms на C++/CLI елемент Check Box надає користувачеві можливість вибору або скасування опції, відображаючись як квадратик з підписом. Встановлений прапорець позначається галочкою. Додається на форму через Visual Studio Toolbox (рис. 4.17), а його поведінка та зовнішній вигляд на-

лаштовуються за допомогою властивостей, таких як текст підпису, стан встановлення (Checked, CheckState), підтримка невизначеного стану (ThreeState), зовнішній вигляд (Appearance), вирівнювання тексту (TextAlign), шрифт і колір тексту (Font, ForeColor), колір фону (BackColor), стан активності (Enabled), видимість (Visible), розташування (Location), розміри (Size), прив'язка (Anchor), прикріплення (Dock), можливість отримання фокуса Tab (TabStop) та порядок переходу Tab (TabIndex).

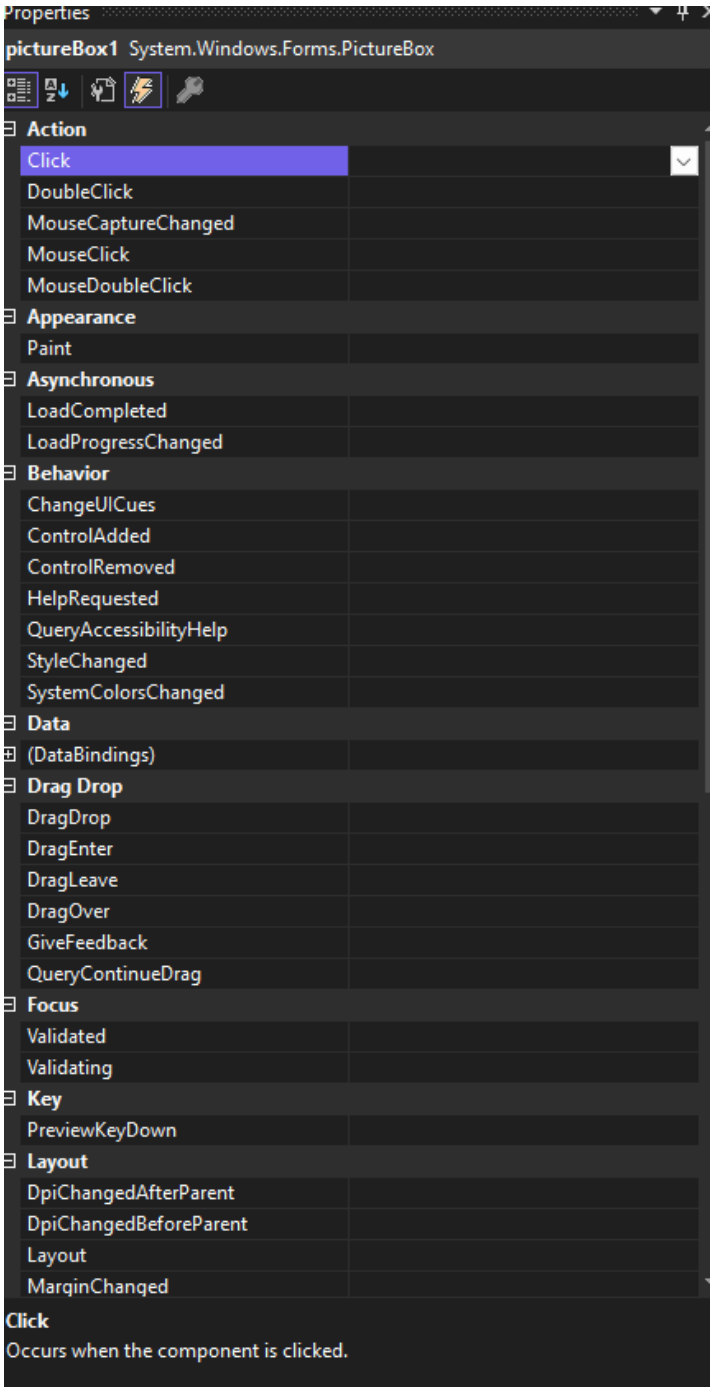


Рисунок 4.16 – Події для PictureBox

Властивість Text встановлює або отримує текст підпису, який відображається поруч із прапорцем:

```
this->checkBox1->Text = "Увімкнути цю опцію";.
```

Властивість Checked – логічне значення, яке визначає, чи встановлено прапорець. true, якщо прапорець встановлено (позначено), і false в іншому випадку:

```
if (this->checkBox1->Checked) {
   // ... виконати дії, якщо прапорець встановлено ...
} else {
   // ... виконати дії, якщо прапорець не встановлено ... }
```

Також є можливість встановлення стану прапорця програмно:

```
this->checkBox1->Checked = true; // Встановити прапорець
this->checkBox1->Checked = false; // Зняти прапорець.
```

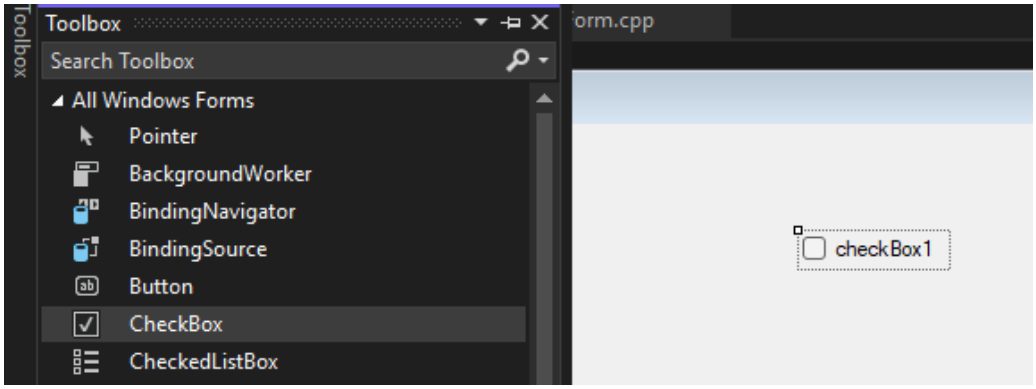


Рисунок 4.17 – Додавання елементу CheckBox

Розглянемо основні події цього елементу.

Першою подією є CheckedChanged, що відбувається, коли значення властивості Checked змінюється. Це відбувається як при взаємодії користувача, так і при програмній зміні значення:

```
private: System::Void checkBox1_CheckedChanged(System::Object^ sender, System::EventArgs^ e) {
   if (this->checkBox1->Checked) {
      System::Windows::Forms::MessageBox::Show("Прапорець встановлено!");
   } else {
     System::Windows::Forms::MessageBox::Show("Прапорець знято!"); }}
```

Приклад 4.7 – Реалізація події CheckedChanged

Або у випадку, коли значення властивості CheckState змінюється (тобто при переході між Checked, Unchecked та Indeterminate):

```
private: System::Void checkBox1_CheckStateChanged(System::Object^ sender, System::EventArgs^ e) {
   switch (this->checkBox1->CheckState) {
   case System::Windows::Forms::CheckState::Checked:
      System::Windows::Forms::MessageBox::Show("Стан: Встановлено");
      break;
   case System::Windows::Forms::CheckState::Unchecked:
      System::Windows::Forms::MessageBox::Show("Стан: Не встановлено");
      break;
   case System::Windows::Forms::CheckState::Indeterminate:
      System::Windows::Forms::MessageBox::Show("Стан: Невизначено");
      break;    } }
```

Приклад 4.8 – Реалізація події з властивістю CheckState

Наприклад, існує форма з CheckBox (checkBoxOption) та Label (labelStatus). необхідно відображати в Label поточний стан CheckBox при кожній зміні:

```
private: System::Void checkBoxOption_CheckedChanged(System::Object^ sender, System::EventArgs^ e) {
   if (this->checkBoxOption->Checked) {
      this->labelStatus->Text = "Опцію увімкнено";
   } else {
      this->labelStatus->Text = "Опцію вимкнено";    } }
```

Приклад 4.9 – Реалізація події з CheckBox та Label

4.4.6 Елемент RadioButton

У Windows Forms на C++/CLI елемент RadioButton дозволяє користувачеві вибирати єдину опцію з групи взаємовиключних варіантів. RadioButton, об'єднані спільним батьківським контейнером (зазвичай Form або GroupBox), працюють як єдиний набір, де лише один елемент може бути обраним одночасно. Візуально RadioButton представлений кружечком з підписом, а обраний стан позначається заповненим кружечком. Додається на форму через Visual Studio Toolbox (рис. 4.18), а його поведінка та зовнішній вигляд налаштовуються за допомогою різноманітних властивостей, включаючи текст підпису, стан обрання (Checked), зовнішній вигляд (Appearance), вирівнювання тексту (TextAlign), шрифт та колір тексту (Font, ForeColor), колір фону (BackColor), стан активності (Enabled), видимість (Visible), розташування (Location), розміри (Size), прив'язка (Anchor), прикріплення (Dock), можливість отримання фокуса Tab (TabStop) та порядок переходу Tab (TabIndex).

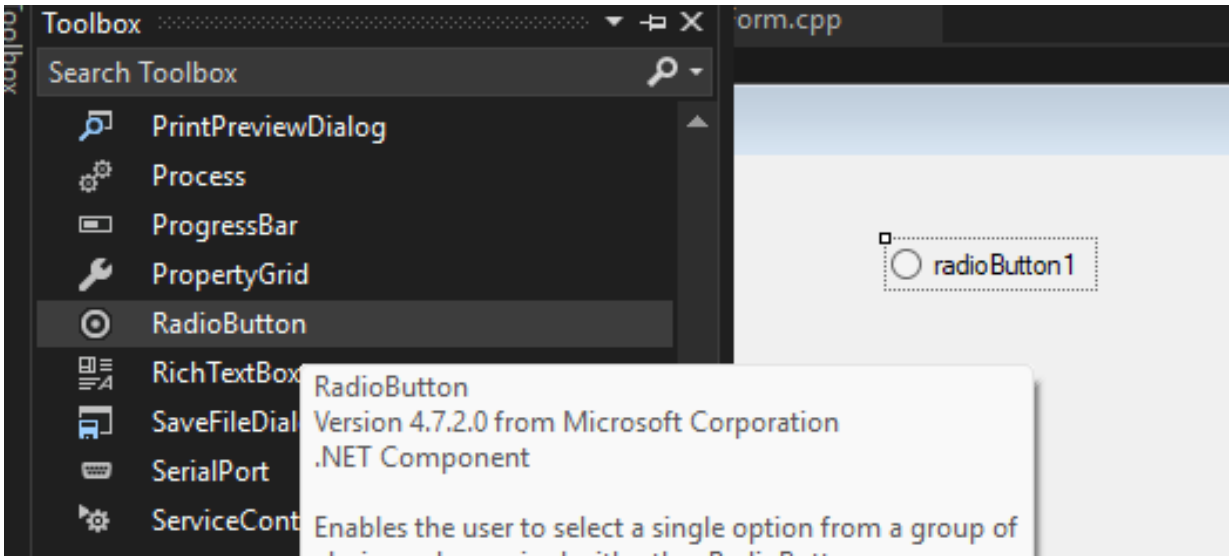


Рисунок 4.18 – Додавання елементу RadioButton

Основною подією RadioButton є CheckedChanged, що відбувається, коли значення властивості Checked змінюється. Це відбувається як при взаємодії користувача (виборі іншого RadioButton у групі або безпосередньому кліку на RadioButton), так і при програмній зміні значення. Зазвичай ви обробляєте цю подію, щоб визначити, яку опцію обрав користувач:

```
private: System::Void radioButton1_CheckedChanged(System::Object^ sender, System::EventArgs^ e) {
   if (this->radioButton1->Checked) {
      System::Windows::Forms::MessageBox::Show("Обрано опцію 1"); } }

private: System::Void radioButton2_CheckedChanged(System::Object^ sender, System::EventArgs^ e) {
   if (this->radioButton2->Checked) {
     System::Windows::Forms::MessageBox::Show("Обрано опцію 2");} }
```

Щоб RadioButton працювали як набір, де можна обрати лише одну опцію, їх необхідно помістити в один контейнер. Цим контейнером може бути або сама форма (Form) – тоді всі RadioButton на ній будуть об'єднані в одну групу, або елемент GroupBox. Розміщення RadioButton у GroupBox створює окрему, незалежну групу. Наприклад, якщо у вас є форма з двома RadioButton для вибору статі (Чоловіча та Жіноча) та кнопкою Підтвердити, розміщення цих RadioButton безпосередньо на формі або всередині GroupBox гарантує, що користувач зможе вибрати лише одну стать:

```
// У файлі .h вашої форми
private: System::Windows::Forms::RadioButton^ radioButtonMale;
private: System::Windows::Forms::RadioButton^ radioButtonFemale;
```

```
private: System::Windows::Forms::Button^ buttonSubmit;

private: System::Void buttonSubmit_Click(System::Object^ sender, System::EventArgs^ e) {
   System::String^ selectedGender = "";
   if (this->radioButtonMale->Checked) {  selectedGender = "Чоловіча";}
    else if (this->radioButtonFemale->Checked) {
      selectedGender = "Жіноча"; }
   else { selectedGender = "Стать не обрано";}
   System::Windows::Forms::MessageBox::Show("Обрана стать: " + selectedGender); }
```

Приклад 4.10 – Реалізація події з RadioButton

4.4.7 Елемент ComboBox

Для надання користувачеві можливості вибору зі списку або введення власного значення у C++/CLI застосунках Windows Forms використовується елемент ComboBox. Його розміщують на формі у візуальному конструкторі Visual Studio, перетягуючи з панелі елементів та визначаючи початкові розміри та положення (рис. 4.19). Налаштування ComboBox включає керування списком елементів через колекцію Items (додавання, видалення, зміна), отримання та встановлення тексту в полі (Text), доступ до обраного елемента (SelectedItem) та його індексу (SelectedIndex), вибір стилю відображення випадаючого списку (DropDownStyle), сортування елементів (Sorted), обмеження кількості видимих елементів (MaxDropDownItems), налаштування шрифту та кольорів (Font, ForeColor, BackColor), керування станом активності (Enabled) та видимості (Visible), визначення розташування (Location) та розмірів (Size), налаштування правил прив'язки до контейнера (Anchor) та способу його прикріплення (Dock), а також керування можливістю отримання фокуса клавішею Tab (TabStop) та порядком переходу між елементами (TabIndex).

Властивість Items – колекція елементів, які відображаються у випадаючому списку. Ви можете додавати, видаляти та змінювати елементи цієї колекції як у Designer, так і програмно:

```
this->comboBox1->Items->Add("Елемент 1");
this->comboBox1->Items->Add("Елемент 2");
this->comboBox1->Items->AddRange(gcnew cli::array<System::Object^> { "Елемент 3", "Елемент 4" });
```

Отримання кількості елементів:

```
int itemCount = this->comboBox1->Items->Count;
```

Отримання елемента за індексом:

```
System::String^ firstItem = safe_cast<System::String^>(this->comboBox1->Items[0]);
```

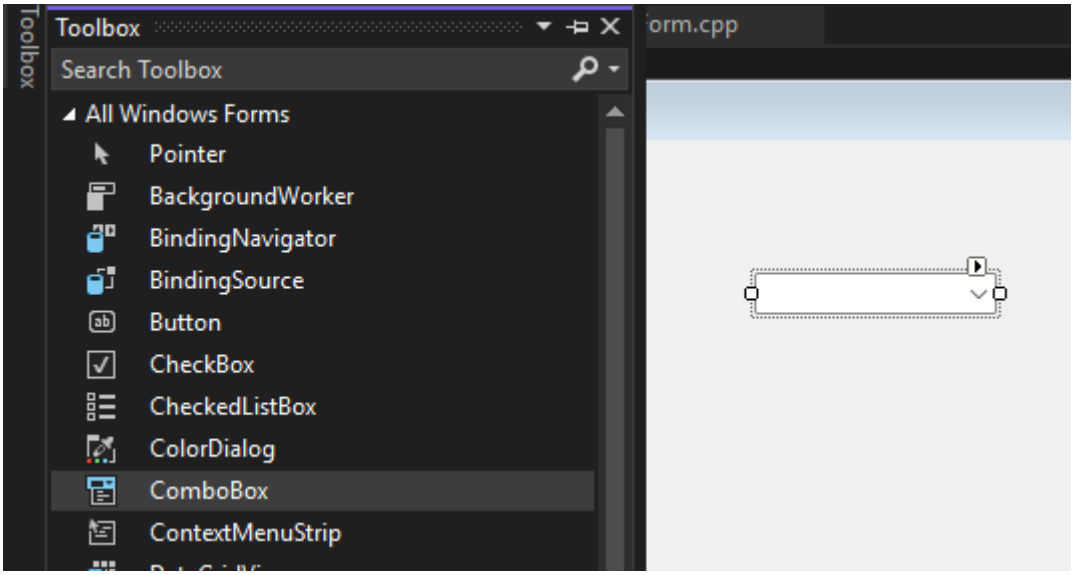


Рисунок 4.19 – Додавання елементу ComboBox

Розглянемо приклад з основними подіями елементу:

```
private: System::Void comboBoxCities_SelectedIndexChanged(System::Object^ sender, System::EventArgs^ e) {
    if (this->comboBoxCities->SelectedIndex != -1) {
        this->labelSelectedCity->Text = safe_cast<System::String^>(this->comboBoxCities->SelectedItem);    } }
```

Також у конструкторі форми або в події Load потрібно додати:

```
public: MyForm1() {
    InitializeComponent();
    this->comboBoxCities->Items->AddRange(gcnew cli::array<System::Object^> { "Київ", "Харків", "Львів",
"Одеса" });
    this->comboBoxCities->SelectedIndex = 0; }  // Вибрати перший елемент за замовчуванням
```

Приклад 4.11 – Реалізація події з ComboBox

4.4.8 Елемент ListBox

Для відображення списку елементів з можливістю вибору одного або кількох у C++/CLI застосунках Windows Forms використовується елемент ListBox. На відміну від ComboBox, список ListBox є постійно видимим. Його розміщують на формі у візуальному конструкторі Visual Studio, перетягуючи з панелі елементів та визначаючи початкові розміри та положення (рис. 4.20). Налаштування ListBox включає керування списком елементів через колекцію Items (додавання, видалення, зміна), отримання та встановлення обраного елемента

(SelectedItem) та його індексу (SelectedIndex), доступ до множинного вибору через колекції SelectedItems та SelectedIndices, визначення режиму вибору (SelectionMode), увімкнення сортування (Sorted), керування відображенням смуг прокрутки (ScrollAlwaysVisible), налаштування шрифту та кольорів (Font, ForeColor, BackColor), керування станом активності (Enabled) та видимості (Visible), визначення розташування (Location) та розмірів (Size), налаштування правил прив'язки до контейнера (Anchor) та способу його прикріплення (Dock), а також керування можливістю отримання фокуса клавішею Tab (TabStop) та порядком переходу між елементами (TabIndex).

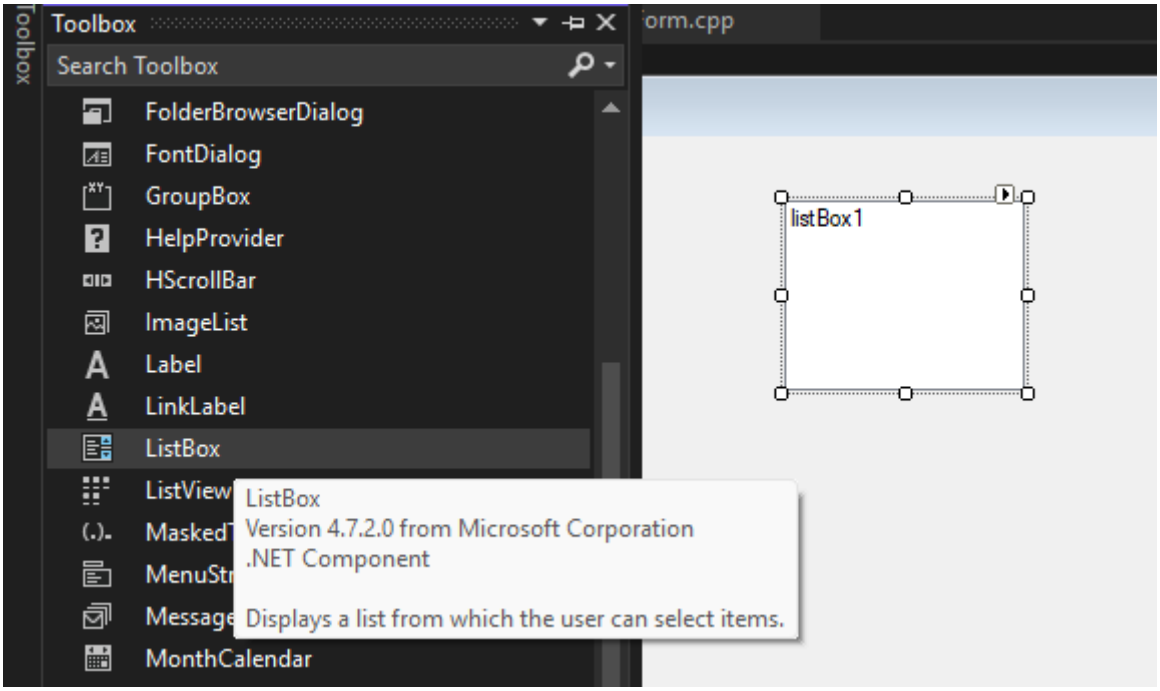


Рисунок 4.20– Додавання елементу ListBox

У цей елемент тако можна додавати елементи програмно:

```
this->listBox1->Items->Add("Елемент A");
this->listBox1->Items->Add("Елемент B");
this->listBox1->Items->AddRange(gcnew cli::array<System::Object^> { "Елемент C", "Елемент D" });
```

Для отримання кількості елементів є спеціальна функція:

```
int itemCount = this->listBox1->Items->Count;
```

Щоб отримати елемент за індексом, потрібно:

```
System::String^ firstItem = safe_cast<System::String^>(this->listBox1->Items[0]);
```

Розглянемо приклад з основними подіями елементу. Наприклад, існує ListBox (listBoxProducts) зі списком продуктів та кнопка (buttonDetails). При виборі продукту в ListBox і натисканні кнопки необхідно відобразити деталі цього продукту:

```
private: System::Void buttonDetails_Click(System::Object^ sender, System::EventArgs^ e) {
    if (this->listBoxProducts->SelectedItem != nullptr) {
        System::String^ selectedProduct = safe_cast<System::String^>(this->listBoxProducts->SelectedItem);
        System::Windows::Forms::MessageBox::Show("Деталі для: " + selectedProduct + " ...");
    } else {
        System::Windows::Forms::MessageBox::Show("Будь ласка, виберіть продукт зі списку.");    } }
```

Приклад 4.12 – Реалізація події з ListBox

Перед цим також потрібно у конструкторі форми або в події Load додати наступне:

```
public: MyForm1() {
    InitializeComponent();
    this->listBoxProducts->Items->AddRange(gcnew cli::array<System::Object^> { "Яблуко", "Банан", "Апельсин", "Груша" }); }
```

4.4.9 Елемент Panel

Елемент Panel у C++/CLI для Windows Forms слугує контейнером для об'єднання інших контролів, таких як кнопки та текстові поля, з метою структурування інтерфейсу користувача. Розміщується на формі за допомогою Visual Studio Toolbox (рис. 4.21). Його функціональність налаштовується через ряд властивостей: стиль межі (BorderStyle), колір фону (BackColor), фонове зображення з налаштуваннями (BackgroundImage, BackgroundImageLayout), увімкнення прокрутки при необхідності (AutoScroll), керування активністю дочірніх елементів (Enabled), контроль видимості групи елементів (Visible), визначення положення (Location) та розмірів (Size), прив'язка до країв контейнера (Anchor), спосіб прикріплення (Dock), вплив на порядок табуляції (TabIndex, хоча рідко отримує фокус сам - TabStop). Дочірні елементи керуються через колекцію Controls.

Panel є корисним для логічного групування елементів, керування їх видимістю, забезпечення прокрутки для великих груп, застосування єдиного візуального стилю та побудови складних макетів інтерфейсу.

Розглянемо форму, на якій для зручного керування параметрами програми використовується контейнер Panel (panelSettings), що об'єднує кілька елементів

CheckBox. Завдяки Panel, група налаштувань візуально відокремлюється, що покращує сприйняття інтерфейсу. Більше того, керуючи властивістю Visible Panel, можна одним дією відобразити або сховати всю секцію налаштувань:

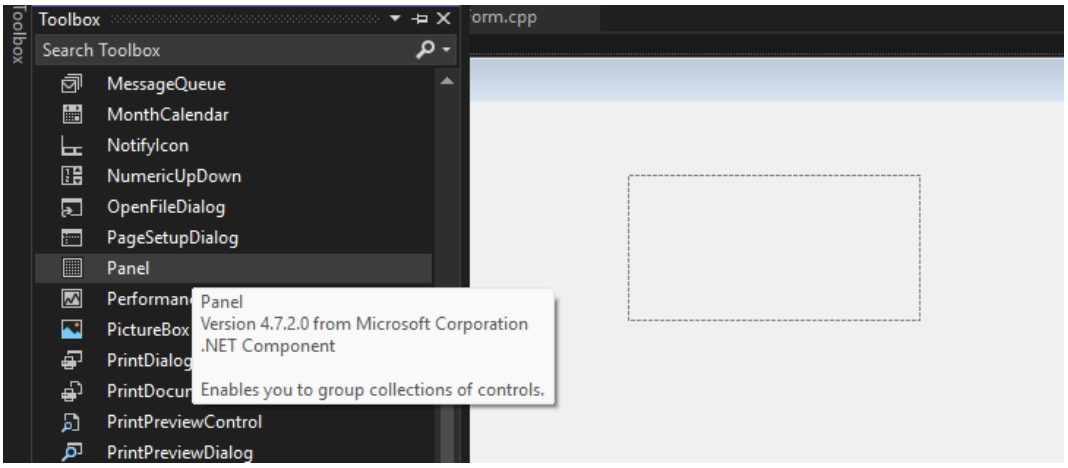


Рисунок 4.21– Додавання елементу Panel

```
private: System::Void buttonToggleSettings_Click(System::Object^ sender, System::EventArgs^ e) {
    this->panelSettings->Visible = !this->panelSettings->Visible; }
```

Приклад 4.13 – Реалізація події з Panel

4.4.10 Елемент GroupBox

У Windows Forms на C++/CLI елемент GroupBox є контейнером для групування інших елементів керування, подібний до Panel, але з вбудованою рамкою та заголовком. Додається на форму через Visual Studio Toolbox (рис. 4.22). Його основні властивості включають текст заголовка (Text), колір фону (BackColor), фонове зображення (BackgroundImage з його розміщенням BackgroundImageLayout), стан активності (Enabled), видимість (Visible), розташування (Location), розміри (Size), прив'язку (Anchor), прикріплення (Dock), порядок табуляції (TabIndex), можливість отримання фокуса Tab (TabStop), колекцію дочірніх елементів (Controls), шрифт та колір заголовка (Font, ForeColor).

GroupBox використовується для візуального групування логічно пов'язаних елементів, надання їм описового заголовка, керування станом групи та організації складних форм.

4.4.11 Елементи TabControl та TabPage

Елементи TabControl та TabPage у C++/CLI для Windows Forms використовуються для створення інтерфейсів з вкладками, де великий обсяг інформації

або елементів керування розділяється на кілька панелей, доступних через вкладки (рис. 4.23). TabControl є основним контейнером, а TabPage – окремою панеллю вмісту. Додавання TabControl на форму та керування TabPage (додавання, вибір, налаштування тексту, фону, розміщення елементів) здійснюється у візуальному конструкторі Visual Studio або програмно через код C++/CLI. Ключові властивості TabControl включають керування сторінками (TabPages), визначення поточної обраної сторінки за індексом (SelectedIndex) або об'єктом (SelectedTab), налаштування розташування (Alignment) та зовнішнього вигляду вкладок (Appearance), керування розмірами вкладок (SizeMode, ItemSize), стилізацію тексту вкладок (Font, ForeColor) та області вкладок (BackColor), а також керування станом активності (Enabled) та відображенням (Visible).

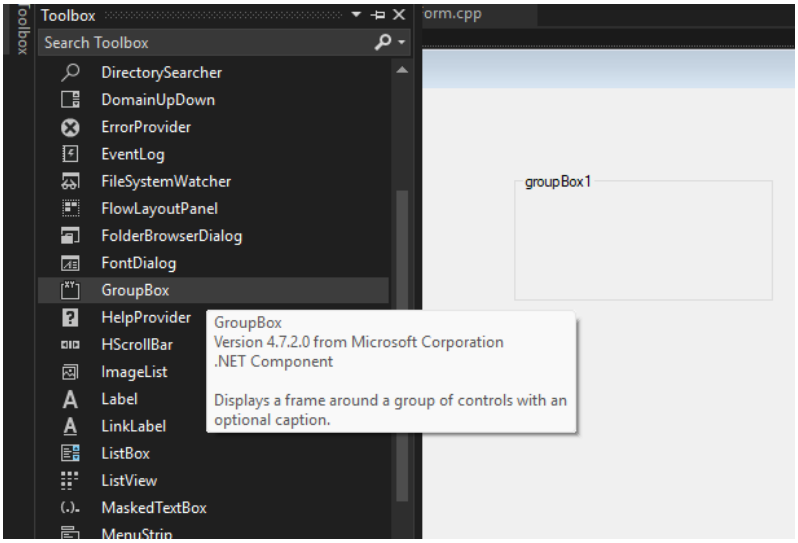


Рисунок 4.22– Додавання елементу GroupBox

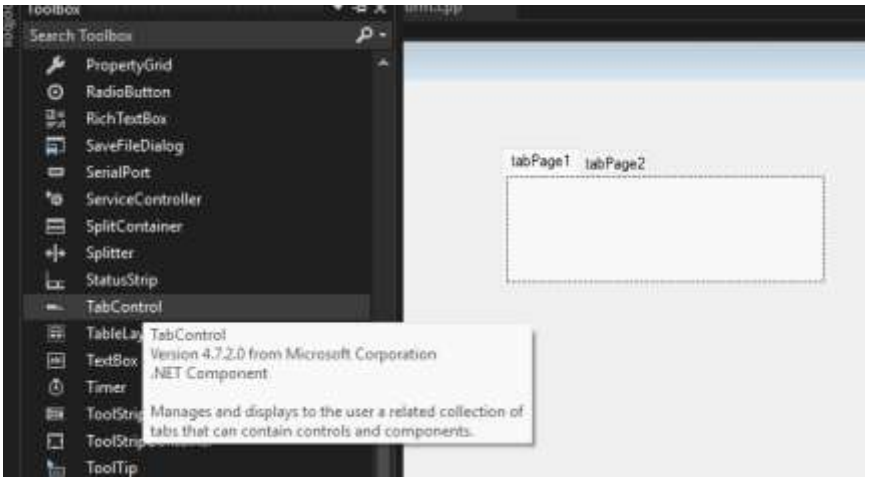


Рисунок 4.23– Додавання елементу TabControl з TabPage

Є можливість створювати екземпляри класу TabPage та додавати їх до колекції TabPages елемента TabControl, наприклад, створення нової TabPage:

```
System::Windows::Forms::TabPage^ newTabPage = gcnew System::Windows::Forms::TabPage();
newTabPage->Text = "Додаткові параметри";
newTabPage->BackColor = System::Drawing::Color::LightGreen;.
```

А також можливість створення та розміщення елемента керування на новій сторінці:

```
System::Windows::Forms::Label^ newLabel = gcnew System::Windows::Forms::Label();
newLabel->Text = "Це додаткові параметри.";
newLabel->newLabel->Location = System::Drawing::Point(10, 10);
newTabPage->Controls->Add(newLabel);.
```

Для того щоб додати сторінки до TabControl, наобхідно наступне:

```
this->tabControl1->TabPages->Add(newTabPage);.
```

Якщо потрібно отрмати або встановити індекс поточної обраної вкладки (починаючи з 0), є відповідні команди:

```
int currentTabIndex = this->tabControl1->SelectedIndex;
this->tabControl1->SelectedIndex = 1;
```

Також є можливість отримати або встановити поточну обрану TabPage:

```
System::Windows::Forms::TabPage^ currentTabPage = this->tabControl1->SelectedTab;
if (currentTabPage != nullptr) {
   System::String^ tabText = currentTabPage->Text;
   // ... виконати дії з обраною сторінкою ... }

this->tabControl1->SelectedTab = this->tabPage2;.
```

4.4.12 Елемент FlowLayoutPanel

Елемент FlowLayoutPanel у C++/CLI для Windows Forms забезпечує динамічне розміщення дочірніх елементів керування в заданому порядку та напрямку (горизонтально або вертикально) з автоматичним перенесенням при досягненні меж контейнера. Це робить його зручним для створення макетів, що гнучко реагують на зміни розміру або вмісту (рис. 4.24). Налаштування FlowLayoutPanel включає визначення напрямку розміщення (FlowDirection), увімкнення або вимкнення перенесення елементів (WrapContents), налаштуван-

ня автоматичного розміру контейнера (AutoSize, AutoSizeMode), встановлення внутрішніх відступів (Padding) та стилю межі (BorderStyle), а також використання стандартних властивостей. Порядок розміщення елементів залежить від порядку їх додавання до колекції Controls. Для точнішого керування розміщенням дочірніх елементів використовуються властивості FlowBreak (примусовий перехід) та Margin (зовнішні відступи). FlowLayoutPanel знаходить застосування при створенні панелей інструментів, кнопкових панелей, відображенні елементів з динамічною кількістю та розробці адаптивних інтерфейсів.

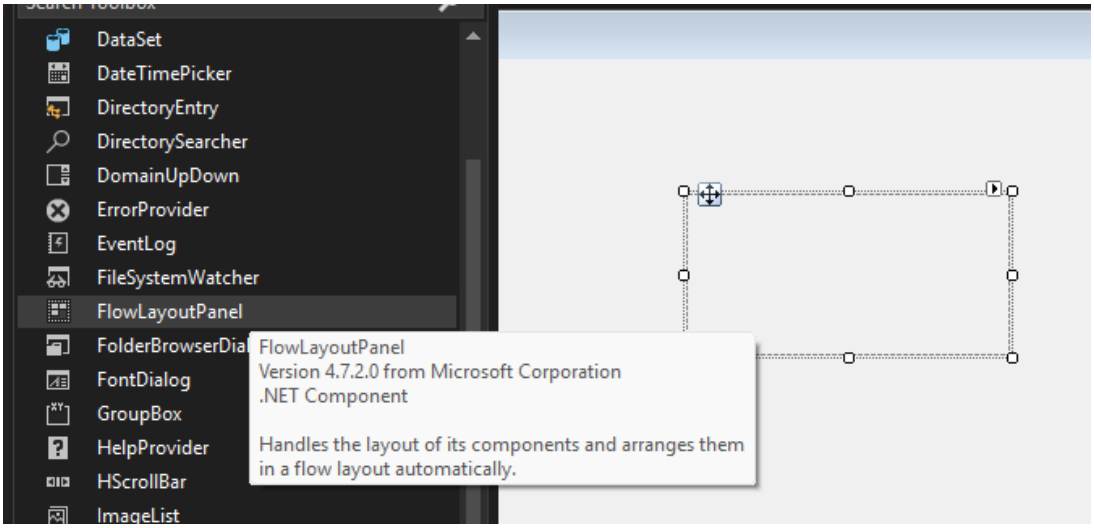


Рисунок 4.24 – Додавання елементу FlowLayoutPanel

Наприклад, існує панель інструментів з кількома кнопками. Ці кнопки розміщено у FlowLayoutPanel з властивістю FlowDirection встановленою в LeftToRight. Якщо вікно стане занадто вузьким, кнопки автоматично перенесуться на наступний рядок, забезпечуючи зручний інтерфейс навіть при зміні розміру вікна. Для цього створимо кількох кнопок:

```
System::Windows::Forms::Button^ button1 = gcnew System::Windows::Forms::Button();
button1->Text = "Кнопка 1";
System::Windows::Forms::Button^ button2 = gcnew System::Windows::Forms::Button();
button2->Text = "Кнопка 2";
System::Windows::Forms::Button^ button3 = gcnew System::Windows::Forms::Button();
button3->Text = "Кнопка 3";
```

Далі додамо ці кнопки до FlowLayoutPanel:

```
this->flowLayoutPanel1->Controls->Add(button1);
this->flowLayoutPanel1->Controls->Add(button2);
this->flowLayoutPanel1->Controls->Add(button3);
```

Останньою дією є встановлення напрямку розміщення:

```
this->flowLayoutPanel1->FlowDirection = System::Windows::Forms::FlowDirection::LeftToRight;
```

4.4.13 Елемент TableLayoutPanel

Елемент TableLayoutPanel у C++/CLI для Windows Forms надає потужний механізм для організації дочірніх елементів керування у вигляді табличної структури з чітким вирівнюванням по рядках і стовпцях (рис. 4.25). Гнучкість налаштування розмірів рядків і стовпців (фіксовані, автоматичні, відсоткові) дозволяє створювати як статичні, так і адаптивні макети. Розміщення елементів у TableLayoutPanel здійснюється візуально у Designer або програмно з можливістю задання їхнього положення в сітці та охоплення кількох клітинок. Керування властивостями TableLayoutPanel (кількість рядків/стовпців, стилі розмірів, межі) та дочірніх елементів (прив'язка, прикріплення) забезпечує точне компонування інтерфейсу. TableLayoutPanel є незамінним інструментом для розробки структурованих форм, адаптивних інтерфейсів та сітчастих макетів.

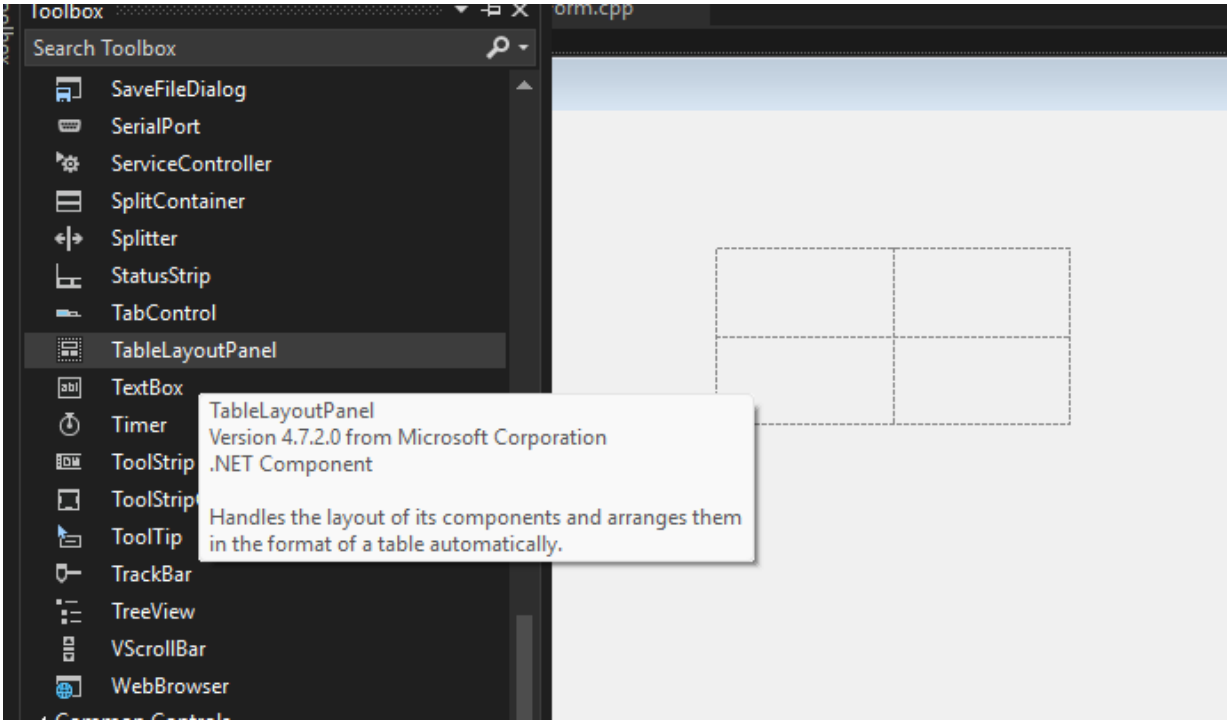


Рисунок 4.25 – Додавання елементу TableLayoutPanel

Є можливість керувати структурою таблиці через властивості RowCount та ColumnCount, а також через колекції RowStyles та ColumnStyles:

```
this->tableLayoutPanel1->RowCount = 3;
this->tableLayoutPanel1->ColumnCount = 2;
```

```
        this->tableLayoutPanel1->RowStyles->Add(gcnew
System::Windows::Forms::RowStyle(System::Windows::Forms::SizeType::AutoSize));

        this->tableLayoutPanel1->ColumnStyles->Add(gcnew
System::Windows::Forms::ColumnStyle(System::Windows::Forms::SizeType::Percent, 50.0f));.
```

При додаванні елемента керування до колекції Controls TableLayoutPanel, можна вказати його рядок та стовпець за допомогою методу SetCellPosition() або властивостей Row та Column. Для охоплення кількох рядків або стовпців використовуйте методи SetRowSpan() та SetColumnSpan() або властивості RowSpan та ColumnSpan:

```
System::Windows::Forms::Button^ newButton = gcnew System::Windows::Forms::Button();
newButton->Text = "Кнопка в (1, 0)";

this->tableLayoutPanel1->Controls->Add(newButton, 0, 1); // (control, column, row)

System::Windows::Forms::Button^ wideButton = gcnew System::Windows::Forms::Button();
wideButton->Text = "Широка кнопка";
this->tableLayoutPanel1->Controls->Add(wideButton, 0, 0);
this->tableLayoutPanel1->SetColumnSpan(wideButton, 2);.
```

Приклад 4.14 – Реалізація події з TableLayoutPanel

4.4.14 Елемент SplitContainer

У Windows Forms на C++/CLI елемент SplitContainer надає користувачеві можливість розділяти область вікна або контейнера на дві незалежні панелі зі змінним розміром, перетягуючи роздільник між ними. Це зручно для відображення пов'язаного контенту (наприклад, навігації та деталей) або для налаштування розмірів різних частин інтерфейсу. SplitContainer додається на форму через Visual Studio Toolbox (рис. 4.26). Його основні властивості включають орієнтацію (Orientation), доступ до панелей (Panel1, Panel2), положення роздільника (SplitterDistance), товщину роздільника (SplitterWidth), можливість переміщення роздільника (IsSplitterFixed), фіксовану панель (FixedPanel), стиль межі (BorderStyle), колір роздільника (SplitterBackColor) та стандартні властивості. Дочірні елементи розміщуються безпосередньо на панелях. Основні події – це переміщення роздільника (SplitterMoved) та його переміщення в процесі (SplitterMoving).

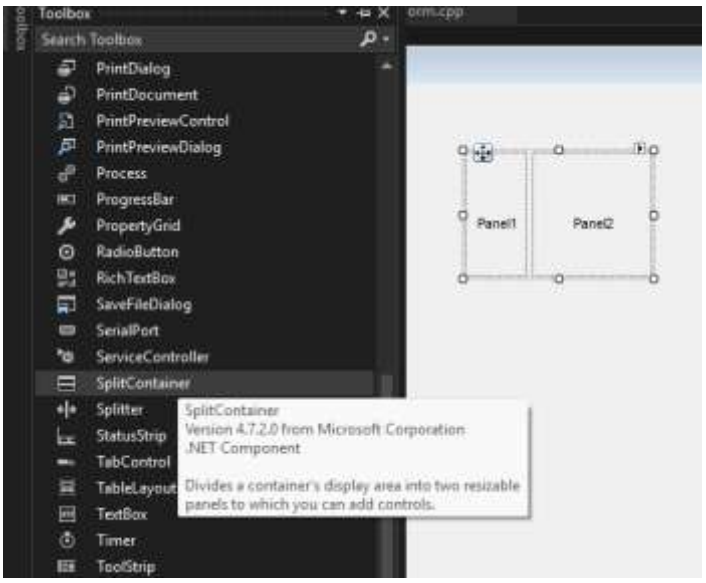


Рисунок 4.26 – Додавання елементу SplitContainer

Розглянемо деякий функціонал. Для додавання кнопки на першу панель необхідно виконати наступну команду:

```
System::Windows::Forms::Button^ button1 = gcnew System::Windows::Forms::Button();
button1->Text = "Кнопка 1";
this->splitContainer1->Panel1->Controls->Add(button1);
```

Щоб отримати колір фону другої панелі, можна виконати наступну команду:

```
System::Drawing::Color panel2Color = this->splitContainer1->Panel2->BackColor;.
```

4.4.15 Елемент ListView

У C++/CLI для Windows Forms елемент ListView є універсальним контролом для відображення колекції елементів у різних форматах (великі/малі значки, список, таблиця з колонками, плитки). Кожен елемент може містити текст, значок та підпорядковані елементи, що відображаються у стовпцях. ListView часто використовується для представлення структурованих даних, таких як файли, завдання або результати пошуку. Додається на форму через Visual Studio Toolbox (рис. 4.27). Його зовнішній вигляд та поведінка налаштовуються за допомогою різноманітних властивостей, включаючи режим відображення (View), колекцію стовпців (Columns), колекцію елементів (Items), списки зображень (SmallImageList, LargeImageList), режими вибору (MultiSelect, FullRowSelect), відображення сітки (GridLines), сортування (Sorting), вибрані елементи (SelectedItem, SelectedItems, SelectedIndex, SelectedIndices), прокрутку

(Scrollable), стиль заголовків (HeaderStyle), редагування міток (LabelEdit), зміну порядку стовпців (AllowColumnReorder), віртуальний режим (VirtualMode) та стандартні властивості. ListView також має ряд важливих подій, таких як зміна вибору (SelectedIndexChanged), активація елемента (ItemActivate), клік на заголовку стовпця (ColumnClick), редагування міток (BeforeLabelEdit, AfterLabelEdit) та перетягування елементів (ItemDrag).

Розглянемо деякий фунціонал. Наприклад, встановимо режиму відображення Деталі:

```
this->listView1->View = System::Windows::Forms::View::Details;
```

Для додавання стовпців існують такі команди:

```
this->listView1->Columns->Add("Назва", 150);
this->listView1->Columns->Add("Розмір", 100);
this->listView1->Columns->Add("Дата змінення", 150);
```

Створення нового елемента у ListView:

```
System::Windows::Forms::ListViewItem^ item1 = gcnew System::Windows::Forms::ListViewItem("Документ.docx");
item1->SubItems->Add("25 КБ");
item1->SubItems->Add("2024-04-23");.
```

Приклад 4.15 – Реалізація події з ListView

Додавання елемента до ListView:

```
this->listView1->Items->Add(item1);.
```

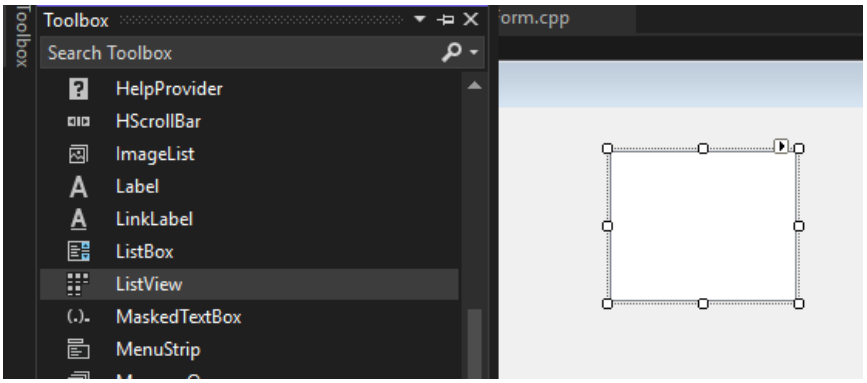


Рисунок 4.27 – Додавання елементу ListView

4.4.16 Елемент TreeView

У C++/CLI для Windows Forms елемент TreeView використовується для відображення ієрархічних даних у вигляді дерева, де кожен вузол може мати текст, значок та дочірні вузли. TreeView часто застосовується для візуалізації файлових систем, організаційних структур або навігаційних меню. Додається на форму через Visual Studio Toolbox (рис. 4.27). Його поведінка та зовнішній вигляд налаштовуються за допомогою властивостей, включаючи колекцію вузлів (Nodes), списки зображень (ImageList), відображення ліній (ShowRootLines, ShowPlusMinus, ShowLines), візуалізацію при наведенні (HotTracking), редагування міток (LabelEdit), прокрутку (Scrollable), вибраний вузол (SelectedNode), відображення зображень вибраного вузла (SelectedImageIndex, SelectedImageKey), роздільник шляхів (PathSeparator), виділення всього рядка (FullRowSelect), відступ (Indent), сортування (Sort, NodesSort), підказки (ShowNodeToolTips) та стандартні властивості. TreeView також має ряд важливих подій, що виникають при виборі, розгортанні/згортанні вузлів, редагуванні міток та діях миші.

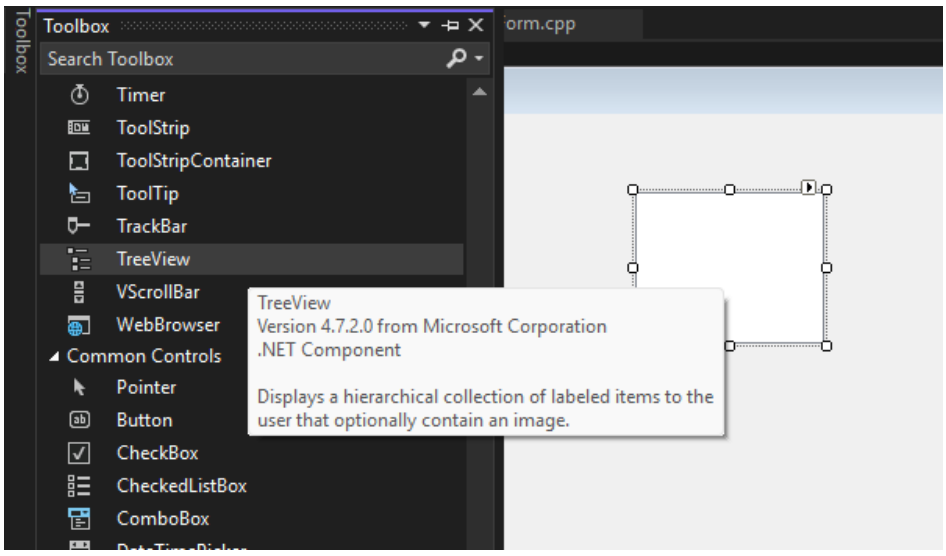


Рисунок 4.27 – Додавання елементу TreeView

Розглянемо функціонал. По-перше, потрібно створити кореневий вузол:

```
System::Windows::Forms::TreeNode^ rootNode = gcnew System::Windows::Forms::TreeNode("Корінь");
```

Далі є функції для створення дочірніх вузлів:

```
System::Windows::Forms::TreeNode^ childNode1 = gcnew System::Windows::Forms::TreeNode("Дочірній 1");
System::Windows::Forms::TreeNode^ childNode2 = gcnew System::Windows::Forms::TreeNode("Дочірній 2");
```

Після чого потрібно додати дочірні вузли до кореневого:

```
rootNode->Nodes->Add(childNode1);
rootNode->Nodes->Add(childNode2);
```

Останньою дією є додавання кореневого вузла до TreeView:

```
this->treeView1->Nodes->Add(rootNode);.
```

4.4.17 Елемент DataGridView

У C++/CLI для Windows Forms елемент DataGridView є потужним та гнучким інструментом для відображення та редагування табличних даних. Він пропонує широкі можливості налаштування зовнішнього вигляду, поведінки та взаємодії з даними, що робить його придатним для роботи з різноманітними наборами даних, від простих таблиць до складних структур з різними типами стовпців, форматуванням та інтерактивністю. DataGridView додається на форму через Visual Studio Toolbox (рис. 4.28). Його основні властивості включають джерело даних (DataSource), колекцію стовпців (Columns), колекцію рядків (Rows), відображення заголовків (ColumnHeadersVisible, RowHeadersVisible), дозволи на додавання/видалення/зміну розміру рядків/стовпців (AllowUserToAddRows, AllowUserToDeleteRows, AllowUserToResizeColumns, AllowUserToResizeRows), режим вибору (SelectionMode), множинний вибір (MultiSelect), режим редагування (EditMode), автоматичне змінення розміру рядків/стовпців (AutoSizeColumnsMode, AutoSizeRowsMode), стилі меж (BorderStyle, CellBorderStyle тощо), кольори та стилі відображення (BackgroundColor, DefaultCellStyle тощо) та стандартні властивості. DataGridView також має велику кількість подій для обробки дій користувача та змін даних, таких як кліки на клітинках/заголовках, зміна значень, початок/завершення редагування, додавання/видалення рядків, сортування, помилки даних та зміна вибору.

Для цього елементу є можливість призначити DataTable як джерело даних:

```
System::Data::DataTable^ dataTable = gcnew System::Data::DataTable("МоїДані");
dataTable->Columns->Add("Ім'я", System::Type::GetType("System.String"));
dataTable->Columns->Add("Вік", System::Type::GetType("System.Int32"));
dataTable->Rows->Add(cli::array<System::Object^>{ "Іван", 30 });
dataTable->Rows->Add(cli::array<System::Object^>{ "Марія", 25 });
this->dataGridView1->DataSource = dataTable;.
```

Приклад 4.16 – Реалізація події з DataGridView

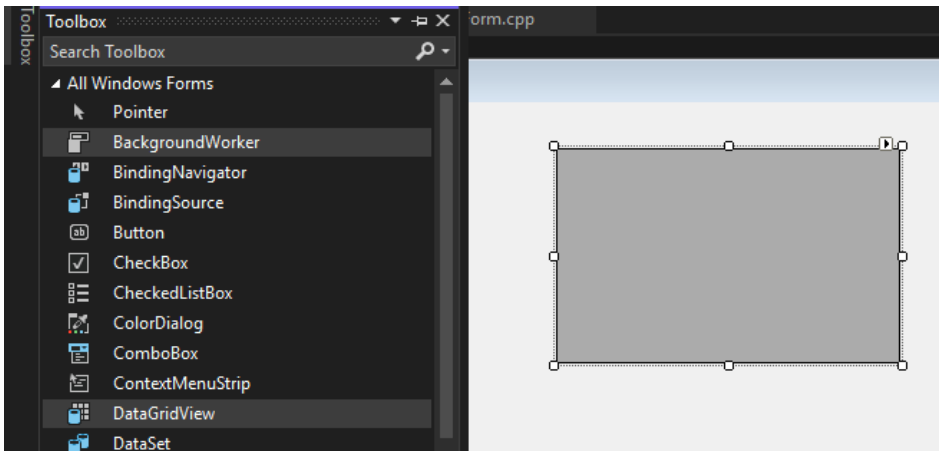


Рисунок 4.28 – Додавання елементу DataGridView

Для додавання текстового стовпця:

```
System::Windows::Forms::DataGridViewTextBoxColumn^ nameColumn = gcnew System::Windows::Forms::DataGridViewTextBoxColumn();
nameColumn->HeaderText = "Ім'я користувача";
nameColumn->DataPropertyName = "Ім'я"; // Відповідність стовпцю в DataSource
this->dataGridView1->Columns->Add(nameColumn);
```

Видалення стовпця за назвою:

```
this->dataGridView1->Columns->Remove("Вік");.
```

Додавання нового порожнього рядка:

```
this->dataGridView1->Rows->Add();
```

Доступ до значення клітинки:

```
if (this->dataGridView1->Rows->Count > 0) {
   System::String^ name = safe_cast<System::String^>(this->dataGridView1->Rows[0]->Cells["Ім'я"]->Value); }
```

Більш детально про використання цього блоку можна буде дізнатися з розділу, присвяченому роботі з базами даних.

4.4.18 Елемент ProgressBar

У C++/CLI для Windows Forms елемент ProgressBar візуально відображає прогрес тривалої операції за допомогою заповненої смуги, що поступово збільшується. Це допомагає користувачам відстежувати хід виконання та розуміти, що програма працює. ProgressBar додається на форму через Visual Studio

Toolbox (рис. 4.29). Його поведінка та вигляд налаштовуються за допомогою властивостей, таких як мінімальне (Minimum) та максимальне (Maximum) значення прогресу, поточне значення (Value), величина кроку (Step), стиль відображення (Style: блоки, суцільна смуга, анімована смуга Marquee з регульованою швидкістю MarqueeAnimationSpeed), орієнтація (Orientation: горизонтальна або вертикальна) та стандартні властивості. Для керування прогресом програмно використовуються методи PerformStep(), Increment() та Reset().

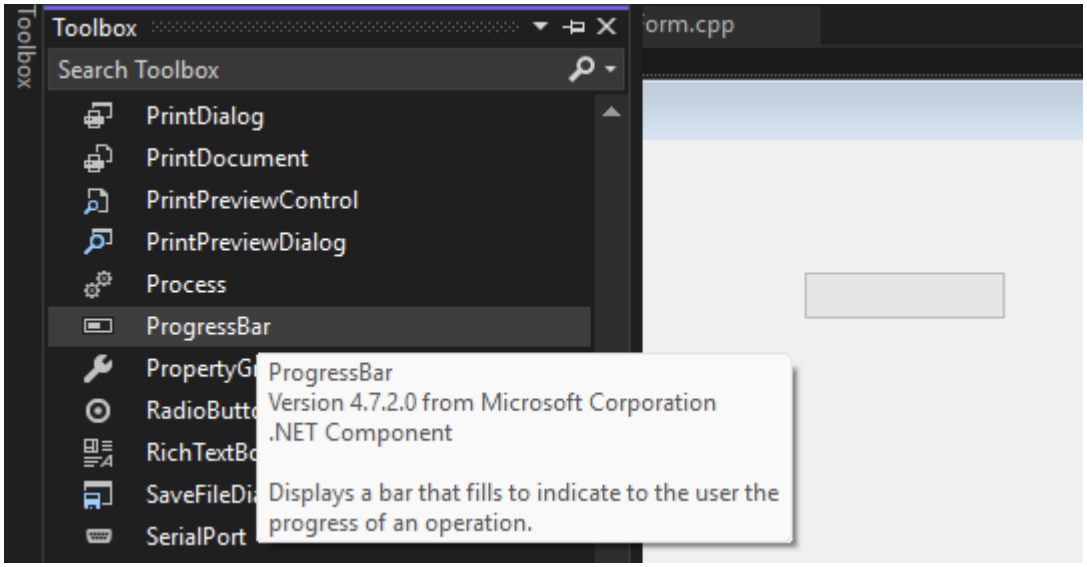


Рисунок 4.29 – Додавання елементу ProgressBar

Для цього елементу можна встановити наступні властивості. По-перше, встановлення мінімального, поточного та максимального значення, а також кроку:

```
this->progressBar1->Minimum = 0;
this->progressBar1->Value = 50;
this->progressBar1->Maximum = 200;
this->progressBar1->Step = 5;
```

Розглянемо приклад. Виконується процес копіювання файлів. ProgressBarбуде використовуватись для відображення відсотка скопійованих даних. Властивість Minimum буде 0, Maximum – загальний розмір файлів (змінна totalBytesToCopy яка зберігає цю величину), а властивість Value буде збільшуватися в міру копіювання кожного фрагмента файлу.

Спочатку потрібно встановити діапазон прогресу:

```
this->progressBar1->Minimum = 0;
this->progressBar1->Maximum = totalBytesToCopy;
```

Після чого у функції оновлювати прогрес під час копіювання:

```
void UpdateProgress(int bytesCopied) {
   if (this->progressBar1->Value + bytesCopied <= this->progressBar1->Maximum) {
      this->progressBar1->Value += bytesCopied;
   } else {
      this->progressBar1->Value = this->progressBar1->Maximum;    } }
```

Приклад 4.17 – Реалізація події з ProgressBar

4.4.19 Елемент TrackBar

У C++/CLI для Windows Forms елемент TrackBar надає користувачеві можливість інтерактивно вибирати числове значення в заданому діапазоні шляхом переміщення повзунка вздовж шкали. Він часто використовується для налаштування параметрів, таких як гучність або яскравість. TrackBar додається на форму через Visual Studio Toolbox (рис. 4.30). Його поведінка та вигляд налаштовуються за допомогою властивостей, включаючи мінімальне (Minimum) та максимальне (Maximum) значення діапазону, поточне значення (Value), частоту відображення поділок (TickFrequency), розташування поділок (TickStyle), величину малих (SmallChange) та великих (LargeChange) кроків зміни значення, орієнтацію (Orientation: горизонтальна або вертикальна), підтримку відображення справа наліво (RightToLeftLayout, RightToLeft), автоматичне змінення розміру (AutoSize) та стандартні властивості. Основні події включають зміну значення (ValueChanged) та процес прокрутки (Scroll).

Основні властивості елементу: мінімальне та максимальне, початкового значення, поділок через кожні декілька одиниць, встановлення величини малого та великого кроку, автоматичного змінення розміру відповідно:

```
this->trackBar1->Minimum = 10;
this->trackBar1->Maximum = 100;
this->trackBar1->Value = 50;
this->trackBar1->TickFrequency = 5;
this->trackBar1->SmallChange = 2;
this->trackBar1->LargeChange = 10;
this->trackBar1->AutoSize = false;
```

ValueChanged є основною подією і відбувається після зміни значення властивості Value в результаті переміщення повзунка користувачем або програмно і записується у нашому випадку у змінну currentValue. Це найбільш часто використовувана подія для отримання поточного значення TrackBar:

```
private: System::Void trackBar1_ValueChanged(System::Object^ sender, System::EventArgs^ e) {
```

```
int currentValue = this->trackBar1->Value; }
```

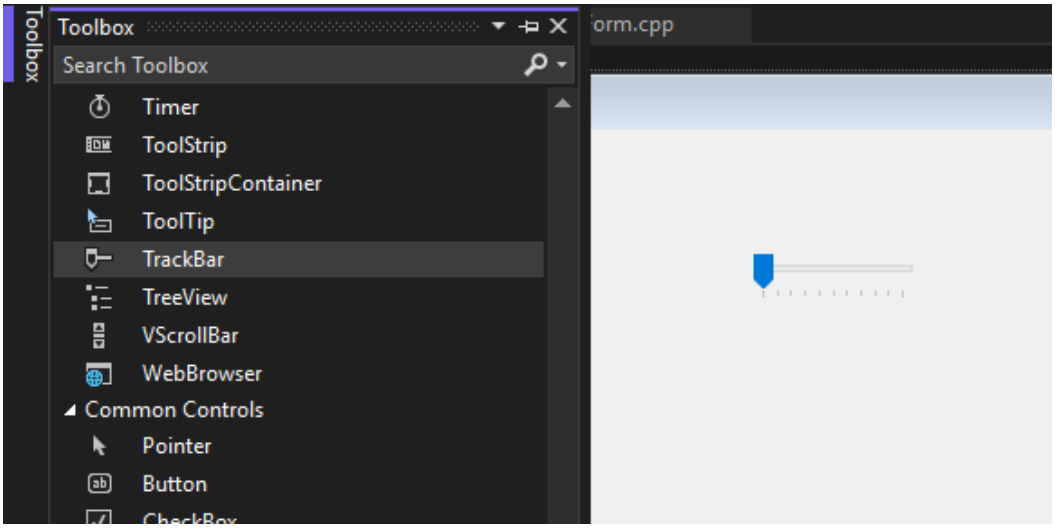


Рисунок 4.30 – Додавання елементу TrackBar

4.4.20 Елемент NumericUpDown

NumericUpDown у C++/CLI для Windows Forms є контролом для введення та вибору числових значень з визначеного діапазону. Користувач може змінювати значення за допомогою спеціальних кнопок або безпосередньо вводити його з клавіатури. Цей елемент керування широко використовується для завдання числових параметрів у програмі. Для використання NumericUpDown його необхідно додати на форму через Visual Studio Toolbox (рис. 4.31). Налаштування включає визначення меж діапазону (мінімальне та максимальне значення), встановлення початкового значення, задання величини кроку зміни, кількості знаків після коми, відображення роздільників тисяч, вирівнювання тексту, заборону введення з клавіатури, вибір стилю межі, відображення в шістнадцятковому форматі та розташування кнопок керування. Основна подія, яка сигналізує про зміну значення, – це ValueChanged.

Основними властивостями є мінімальне та максимальне, початкового значення, поділок через кожні декілька одиниць, знаків після коми відповідно:

```
this->numericUpDown1->Minimum = 1;
this->numericUpDown1->Maximum = 500;
this->numericUpDown1->Value = 10;
this->numericUpDown1->Increment = 5;
this->numericUpDown1->DecimalPlaces = 2;.
```

Основною подією є ValueChanged – відбувається після зміни значення властивості Value користувачем (через кнопки або введення з клавіатури). Та-

кож можливо використати стандартні події клавіатури (KeyDown, KeyPress, KeyUp), які можуть бути використані для обробки введення користувача. Це найбільш часто використовувана подія для отримання поточного значення:

```
private: System::Void numericUpDown1_ValueChanged(System::Object^ sender, System::EventArgs^ e) {
    Decimal currentValue = this->numericUpDown1->Value; }
```

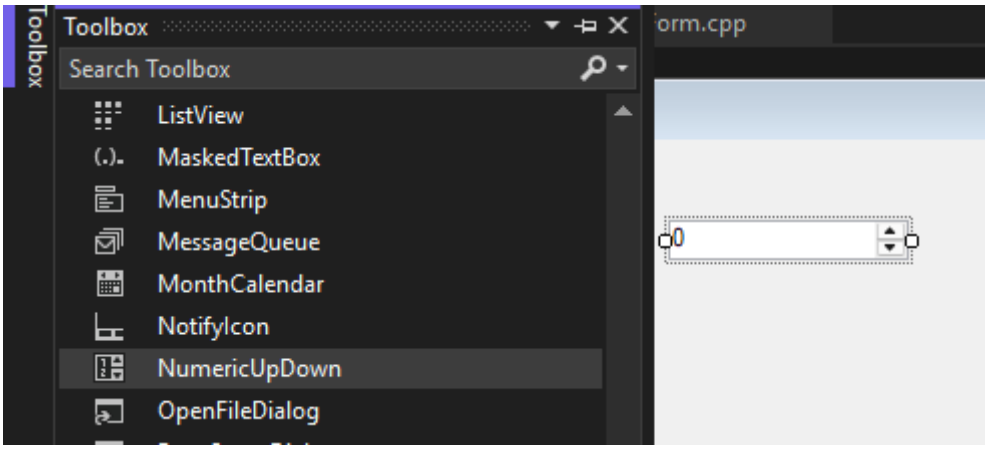


Рисунок 4.31 – Додавання елементу NumericUpDown

4.4.21 Елемент DateTimePicker

DateTimePicker у C++/CLI для Windows Forms є елементом керування, призначеним для зручного вибору дати та часу. Користувач може використовувати розкривний календар або безпосередньо вводити значення. Цей контрол широко застосовується для завдань, пов'язаних з вибором часових параметрів. Для використання DateTimePicker його необхідно додати на форму через Visual Studio Toolbox (рис. 4.32). Налаштування включає визначення формату відображення (включаючи можливість задання власного формату), відображення елемента у вигляді кнопок замість календаря, встановлення меж допустимих дат, відображення прапорця для позначення вибраного значення, налаштування вирівнювання календаря та використання стандартних властивостей. Основна подія, що сигналізує про зміну вибраної дати/часу, – це ValueChanged.

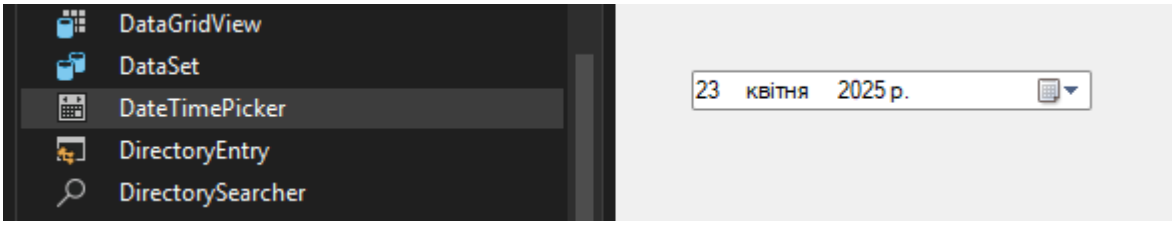


Рисунок 4.31 – Додавання елементу DateTimePicker

Для отримання вибраної дати та часу потрібно:

```
System::DateTime selectedDateTime = this->dateTimePicker1->Value;
```

Встановлення певної дати та часу:

```
this->dateTimePicker1->Value = System::DateTime(2025, 12, 25);
```

Відображення тільки часу:

```
this->dateTimePicker1->Format = System::Windows::Forms::DateTimePickerFormat::Time;
```

Використання користувацького формату:

```
this->dateTimePicker1->Format = System::Windows::Forms::DateTimePickerFormat::Custom;
this->dateTimePicker1->CustomFormat = "dd.MM.yyyy HH:mm";
```

Відображення у вигляді кнопок вгору-вниз:

```
this->dateTimePicker1->ShowUpDown = true;
```

Встановлення мінімальної та максимальної дати:

```
this->dateTimePicker1->MinDate = System::DateTime(2020, 1, 1);
this->dateTimePicker1->MaxDate = System::DateTime(2030, 12, 31);
```

Подія ValueChanged відбувається після зміни значення властивості Value користувачем (через розкривний календар або введення):

```
private: System::Void dateTimePicker1_ValueChanged(System::Object^ sender, System::EventArgs^ e) {
   System::DateTime newDateTime = this->dateTimePicker1->Value; }
```

Також є додаткові події, такі як:

– CloseUp: Відбувається після закриття розкривного календаря;

– DropDown: Відбувається при розкритті календаря;

– CheckStateChanged: Відбувається при зміні стану прапорця (якщо ShowCheckBox має значення true).

4.4.22 Елементи ToolStrip & ToolStripItem

ToolStrip у C++/CLI для Windows Forms (рис. 4.32) являє собою панель інструментів, яка слугує контейнером для різноманітних елементів керування ToolStripItem, таких як кнопки (ToolStripButton), мітки (ToolStripLabel), поля введення (ToolStripTextBox), розкривні списки (ToolStripComboBox), роздільники (ToolStripSeparator) та кнопки з розкривними меню (ToolStripDropDownButton, ToolStripSplitButton). Ці панелі інструментів використовуються для організації швидкого доступу до ключових функцій програми, часто розташовуючись у верхній частині вікна. ToolStripItem є базовим типом для всіх елементів, що розміщуються на ToolStrip. Для додавання ToolStrip на форму потрібно скористатися Visual Studio Designer, знайшовши його в розділі "Menus & Toolbars" та перетягнувши на потрібне місце. Потім елементи ToolStripItem додаються безпосередньо на створену панель інструментів.

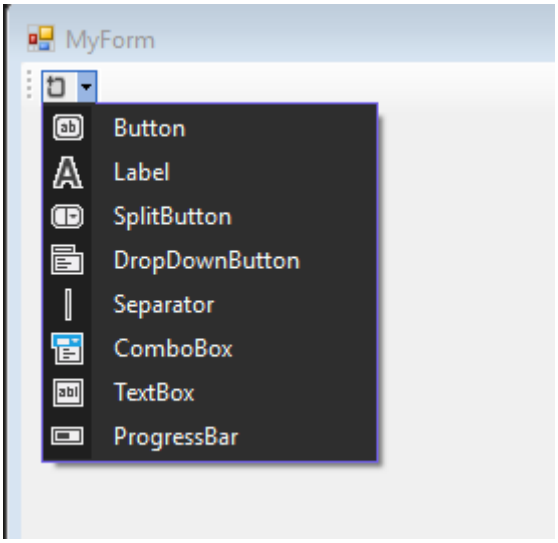


Рисунок 4.32 – Додавання елементу ToolStrip

Основними властивостями є наступні. Перше це створення нової кнопки:

```
System::Windows::Forms::ToolStripButton^ newButton = gcnew System::Windows::Forms::ToolStripButton("Створити");
newButton->Image = System::Drawing::Image::FromFile("create.png");
newButton->DisplayStyle = System::Windows::Forms::ToolStripItemDisplayStyle::ImageAndText;
newButton->Click += gcnew System::EventHandler(this, &MyForm::newButton_Click);
```

Далі це додавання кнопки на ToolStrip:

```
this->toolStrip1->Items->Add(newButton);
```

Додавання роздільника:

```
this->toolStrip1->Items->Add(gcnew System::Windows::Forms::ToolStripSeparator());
```

Додавання мітки:

```
this->toolStrip1->Items->Add(gcnew System::Windows::Forms::ToolStripLabel("Статус:"));.
```

Також є можливість динамічно додавати елементи на ToolStrip у коді C++/CLI:

```
System::Windows::Forms::ToolStripLabel^ statusLabel = gcnew System::Windows::Forms::ToolStripLabel("Готово");
this->toolStrip1->Items->Add(statusLabel);

System::Windows::Forms::ToolStripButton^ exitButton = gcnew System::Windows::Forms::ToolStripButton("Вихід");
exitButton->Click += gcnew System::EventHandler(this, &MyForm::exitButton_Click);
this->toolStrip1->Items->Add(exitButton);
```

Приклад 4.18 – Реалізація події з ToolStrip & ToolStripItem

Елементи ToolStripItem генерують ряд подій для обробки взаємодії користувача: Click відбувається при натисканні, TextChanged – при зміні тексту (для текстових полів та комбобоксів), SelectedIndexChanged – при зміні вибраного елемента в комбобоксах та розкривних меню, а DropDownOpening і DropDownClosed сигналізують про відкриття та закриття розкривних меню.

4.4.23 Елементи MenuStrip та ToolStripMenuItem

У C++/CLI для Windows Forms MenuStrip є контейнером для створення головного меню вікна, яке зазвичай розташовується у верхній частині (рис. 4.34). Він містить пункти меню верхнього рівня, представлені класом ToolStripMenuItem. ToolStripMenuItem може мати текст, зображення, гарячі клавіші, підменю та обробники подій. Для додавання MenuStrip на форму використовується Visual Studio Designer, де його можна знайти в розділі "Menus & Toolbars" та перетягнути у верхню частину форми. Пункти меню ToolStripMenuItem додаються безпосередньо в конструкторі MenuStrip. Основні властивості MenuStrip включають колекцію пунктів меню (Items), пункт для відображення списку MDI-вікон (MdiWindowListItem), відображення підказок (ShowItemToolTips) та стандартні властивості.

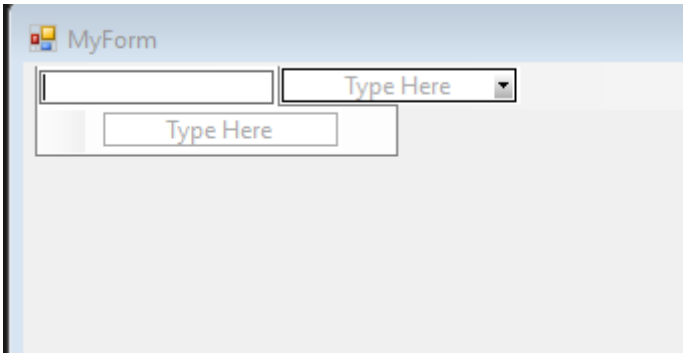


Рисунок 4.34 – Зовнішній вигляд елементу MenuStrip та ToolStripMenuItem

Для створення нового пункту меню верхнього рівня є функція:

```
System::Windows::Forms::ToolStripMenuItem^ fileMenuItem = gcnew System::Windows::Forms::ToolStripMenuItem("Файл");
```

Для того, щоб додати пункту меню до MenuStrip:

```
this->menuStrip1->Items->Add(fileMenuItem);
```

Для того, щоб створити підменю:

```
System::Windows::Forms::ToolStripMenuItem^ saveMenuItem = gcnew System::Windows::Forms::ToolStripMenuItem("Зберегти");
saveMenuItem->Click += gcnew System::EventHandler(this, &MyForm::saveMenuItem_Click);

System::Windows::Forms::ToolStripMenuItem^ exitMenuItem = gcnew System::Windows::Forms::ToolStripMenuItem("Вихід");
exitMenuItem->Click += gcnew System::EventHandler(this, &MyForm::exitMenuItem_Click);
```

Для програмного додавання підпунктів до пункту Файл:

```
fileMenuItem->DropDownItems->Add(saveMenuItem);
fileMenuItem->DropDownItems->Add(gcnew System::Windows::Forms::ToolStripSeparator());
fileMenuItem->DropDownItems->Add(exitMenuItem);
```

Найголовнішою подією є Click, що відбувається при виборі (натисканні) пункту меню. Саме в обробнику цієї події зазвичай виконується основна логіка, пов'язана з пунктом меню:

```
private: System::Void saveMenuItem_Click(System::Object^ sender, System::EventArgs^ e) {
   // Логіка збереження файлу тощо }
```

Також є додаткові події:

– DropDownOpening: Виникає безпосередньо перед відкриттям підменю;

– DropDownClosed: Виникає після закриття підменю.

4.4.24 StatusStrip та ToolStripStatusLabel

У C++/CLI для Windows Forms StatusStrip є контейнером, який зазвичай розташовується внизу вікна та призначений для відображення інформації про стан програми, фонові процеси, повідомлення та індикатори. Він може містити різні елементи керування, серед яких ToolStripStatusLabel – текстова мітка для виведення статусної інформації з можливістю форматування та відображення значків. Додається StatusStrip на форму через Visual Studio Toolbar з розділу Menus & Toolbars шляхом перетягування в нижню частину вікна (рис. 4.35). Основні властивості StatusStrip включають колекцію елементів (Items), закріплення внизу вікна (Dock), відображення ручки зміни розміру (SizingGrip), показ підказок (ShowItemToolTips) та стандартні властивості. ToolStripStatusLabel має такі важливі властивості, як текст (Text), зображення (Image), спосіб відображення (DisplayStyle), підказка (ToolTipText), межі (BorderSides, BorderStyle), автоматичне розтягування (Spring) та вирівнювання тексту (TextAlign).

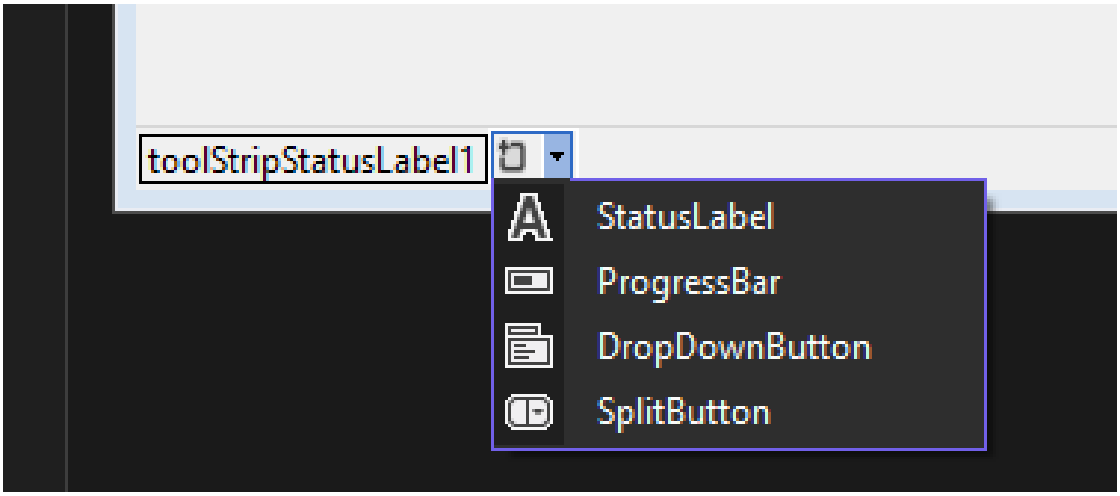


Рисунок 4.35 – Налаштування елементів StatusStrip та ToolStripStatusLabel

Можна додавати властивості, наприклад, створення нової мітки статусу:

```
System::Windows::Forms::ToolStripStatusLabel^ statusLabel = gcnew System::Windows::Forms::ToolStripStatusLabel("Готово");
```

Або додавання мітки на StatusStrip:

```
this->statusStrip1->Items->Add(statusLabel);
```

Чи додавання індикатора прогресу:

```
System::Windows::Forms::ToolStripProgressBar^ progressBar = gcnew System::Windows::Forms::ToolStripProgressBar();
progressBar->Value = 50;
this->statusStrip1->Items->Add(progressBar);
```

Також можна додавати мітки статусу та інші елементи на StatusStrip у коді C++/CLI програмно. По-перше, створення нової мітки статусу:

```
System::Windows::Forms::ToolStripStatusLabel^ messageLabel = gcnew System::Windows::Forms::ToolStripStatusLabel("Операція завершена");
this->statusStrip1->Items->Add(messageLabel);
```

По-друге, оновлення тексту існуючої мітки статусу:

```
if (this->statusStrip1->Items->Count > 0) {
   safe_cast<System::Windows::Forms::ToolStripStatusLabel^>(this->statusStrip1->Items[0])->Text = "Виконується завантаження..."; }.
```

4.4.25 Елементи OpenFileDialog, SaveFileDialog, FolderBrowserDialog, ColorDialog та FontDialog

У C++/CLI для розробки інтерфейсу Windows Forms доступні стандартні вікна, які спрощують типові дії користувача, як-от робота з файлами (відкриття, збереження), вибір папок, кольорів і шрифтів (рис. 4.36). також слід зазначити, що при додаванні вони є невидимі і з’являться під створеною формою.

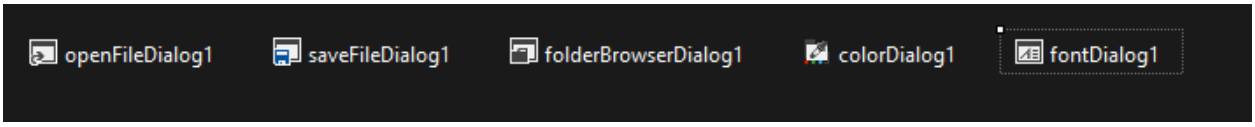


Рисунок 4.36 – Діалоги, додані до форми

OpenFileDialog – надає можливість обрати один або декілька файлів для подальшої роботи:

– відображає стандартне вікно "відкрити";

– дозволяє обмежувати види файлів для відображення за допомогою фільтрів (наприклад, лише текстові файли .txt або зображення .jpg і .jpeg);

– надає повну інформацію про обрані файли, включаючи їхній шлях і назву;

– може перевіряти, чи існують вказані файли та шляхи до них;

– підтримує вибір кількох файлів одночасно;

Основними властивостями є:

– Filter: задає або повертає рядок, що визначає фільтри для відображення типів файлів;

– FileName: встановлює або отримує ім'я файлу, обраного користувачем;

– FileNames: повертає масив з іменами всіх обраних файлів (актуально, якщо Multiselect має значення true);

– InitialDirectory: визначає початкову папку, яка відкривається при відображенні вікна;

– Title: задає або отримує заголовок вікна;

– Multiselect: вказує, чи дозволено користувачеві вибирати кілька файлів одночасно (значення true або false);

– CheckFileExists: визначає, чи буде вікно перевіряти наявність обраного файлу;

– CheckPathExists: визначає, чи буде вікно перевіряти наявність обраного шляху.

Основним методом є ShowDialog(), що відображає діалогове вікно. Він повертає значення типу System::Windows::Forms::DialogResult, яке вказує, яку кнопку натиснув користувач (наприклад, OK або Cancel).

SaveFileDialog – дозволяє користувачеві вказати місцезнаходження та ім'я для збереження файлу:

– відображає стандартне вікно Зберегти як;

– дозволяє фільтрувати типи файлів для збереження;

– може попереджати про спробу перезаписати вже існуючий файл;

– надає інформацію про обраний файл (шлях та ім'я);

– може автоматично додавати розширення до імені файлу.

Основними властивостями є:

– Filter: задає або повертає рядок фільтра для типів файлів, які можна зберегти;

– FileName: встановлює або отримує ім'я файлу, введене користувачем;

– InitialDirectory: визначає початкову папку для відображення;

– Title: задає або отримує заголовок вікна;

– OverwritePrompt: вказує, чи потрібно виводити попередження перед перезаписом існуючого файлу;

– AddExtension: визначає, чи буде вікно автоматично додавати розширення до імені файлу, якщо його не ввів користувач;

– DefaultExt: задає розширення файлу за замовчуванням;

– CreatePrompt: вказує, чи потрібно запитувати підтвердження перед створенням нового файлу.

Основним методом є ShowDialog(), що показує діалогове вікно та повертає значення типу System::Windows::Forms::DialogResult.

FolderBrowserDialog – надає можливість користувачеві обрати папку (директорію):

– відображає спеціалізоване вікно для навігації та вибору папок.

– дозволяє задати початкову папку для перегляду.

– може відображати кнопку для створення нової папки.

– повертає шлях до обраної папки.

Основними властивостями є:

– SelectedPath: встановлює або отримує шлях до обраної папки;

– RootFolder: визначає кореневу папку, з якої починається перегляд структури папок;

– Description: задає текст опису, який відображається у верхній частині вікна;

– ShowNewFolderButton: вказує, чи відображати кнопку "Створити нову папку".

Основним методом є ShowDialog() для відображення діалогового вікна та повернення значення типу System::Windows::Forms::DialogResult.

ColorDialog – дозволяє користувачеві вибрати колір з доступної палітри або створити власний:

– відображає стандартне вікно "колір";

– надає можливість вибору основних та розширених кольорів;

– забезпечує доступ до характеристик обраного кольору (наприклад, значення rgb, hsl);

– може дозволяти користувачеві визначати власні кольори.

Основними властивостями є:

– Color: встановлює або отримує обраний колір;

– AllowFullOpen: вказує, чи може користувач розгортати розділ "Визначити користувацький колір";

– FullOpen: визначає, чи буде розділ "Визначити користувацький колір" відкритий за замовчуванням при відображенні вікна;

– AnyColor: вказує, чи відображаються всі доступні кольори в базовій палітрі;

– SolidColorOnly: визначає, чи може користувач вибирати лише суцільні кольори;

– CustomColors: задає або отримує масив користувацьких кольорів, які відображаються у вікні.

Основний метод є ShowDialog() – діалогове вікно та повертає значення типу System::Windows::Forms::DialogResult.

FontDialog дозволяє користувачеві вибрати шрифт, його накреслення та розмір:

– відображає стандартне вікно "шрифт";

– дозволяє вибирати сімейство шрифтів, стиль (наприклад, жирний, курсив) та розмір;

– надає можливість попереднього перегляду обраного шрифту;

– може обмежувати список доступних шрифтів за їхнім типом (наприклад, показувати лише векторні шрифти).

Основними властивостями є:

– Font: встановлює або отримує обраний шрифт;

– FontMustExist: вказує, чи має обраний шрифт обов'язково бути встановленим у системі;

– ShowEffects: визначає, чи відображаються параметри ефектів (наприклад, підкреслення, закреслення);

– ShowColor: вказує, чи відображається параметр вибору кольору шрифту;

– Color: встановлює або отримує колір шрифту, обраний користувачем (якщо ShowColor має значення true);

– MinSize, MaxSize: задають мінімальний та максимальний розміри шрифту, які може вибрати користувач;

– FixedPitchOnly: вказує, чи відображаються лише шрифти з фіксованим кроком;

– AllowScriptChange: визначає, чи може користувач змінювати набір символів (скрипт).

Основним методом є метод ShowDialog(), що показує діалогове вікно та повертає значення типу System::Windows::Forms::DialogResult.

4.5 Приклади з елементами керування

4.5.1 Перший приклад з основними елементами

Першим кроком є створення форми та додавання на неї елементів керування з Toolbox (рис. 4.37):

label – 3 шт.;

checkbox – 1 шт.;

radioButton – 2 шт.;

listBox – 1 шт.;

textbox – 1 шт.;

pictureBox – 1 шт.;

button – 4 шт.;

progressBar – 1 шт.;

trackBar – 1 шт.;

openFileDialog – 1 шт.;

colorDialog – 1 шт.;

fontDialog – 1 шт.;

Всі елементи на формі додані без зміни назви, що можна було легше відстежити, що і де знаходиться.

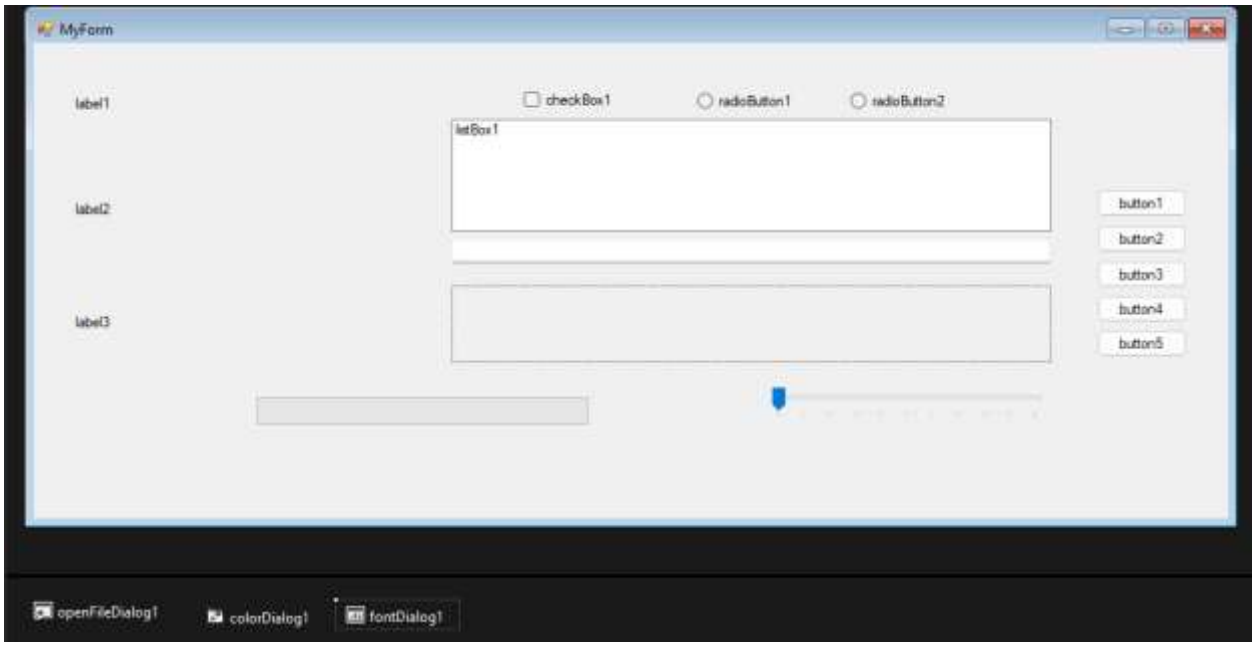


Рисунок 4.37 – Зовнішній вигляд створеної форми

Для кожної з кнопок, а також до елементу trackBar було додано обробники. До кнопок – Click (двічі натиснувши на кнопку), до trackBar – обравши події (значок блискавки у Properties).

Розглянемо кожний з обробників.

Для trackBar він виглядатиме наступним чином:

```
private: System::Void trackBar1_Scroll(System::Object^ sender, System::EventArgs^ e) {
        progressBar1->Value = trackBar1->Value * 10;}
```

Ми беремо поточне значення Value від trackBar1, перемножуємо на 10 та ставимо поточним значенням для progressBar1.

Для кнопки button1 обробник виглядає наступним чином:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
        if (openFileDialog1->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
                String^ selectedFile = openFileDialog1->FileName;
                textBox1->Text = selectedFile; }
```

В даному випадку відкривається openFileDialog1, де обирається файл. Після вибору файлу та натиснення клавіші ОК, шлях до цього файлу встановлюється в якості тексту (Text) для textBox1 (рис. 4.28-4.29).

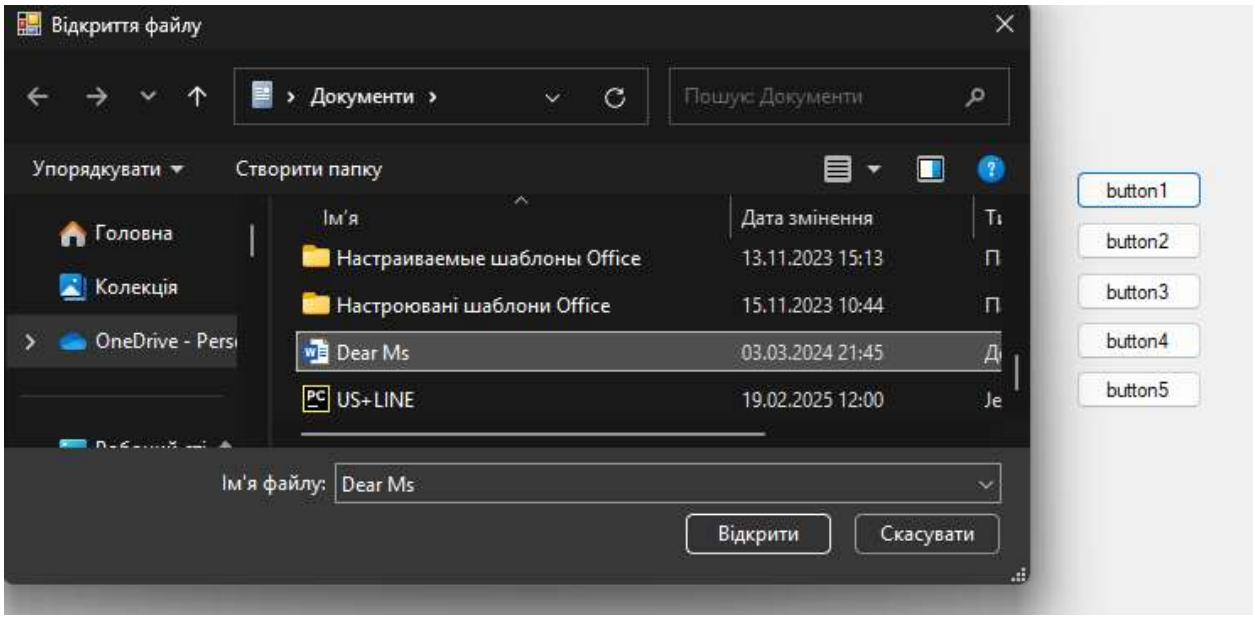


Рисунок 4.28 – Відкриття openFileDialog1

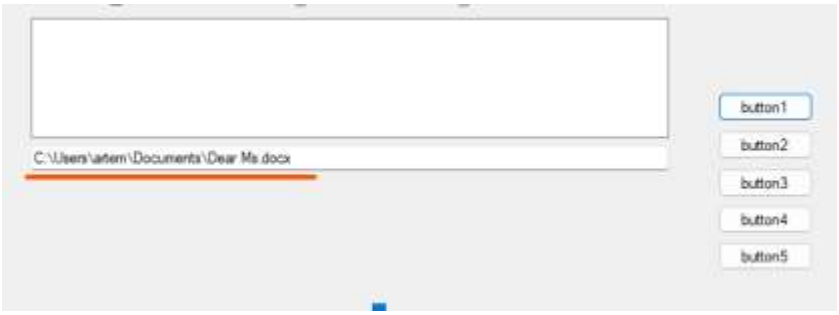


Рисунок 4.29 – Встановлення шляху до файлу як текст у textBox1

Обробник кнопки button2 матиме наступний вигляд:

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e) {
        if (colorDialog1->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
                Color selectedColor = colorDialog1->Color;
                pictureBox1->BackColor = selectedColor;} }
```

Під час натискання 2 кнопки відкривається colorDialog1, де після обирання кольору і натиснення клавіші ОК, цей колір стає кольором фону pictureBox1 (рис. 4.30-4.31).

Рисунок 4.30 – Відкриття colorDialog1

Рисунок 4.31 – Встановлення значення з colorDialog1 у pictureBox1->BackColor

Обробник кнопки button3 матиме наступний вигляд:

```
private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e) {
            if (fontDialog1->ShowDialog() == System::Windows::Forms::DialogResult::OK) {
                        System::Drawing::Font^ selectedFont = fontDialog1->Font;
                        textBox1->Font = selectedFont;        } }
```

Натискання 3 кнопки приводить до відкривання fontDialog1, де після обирання шрифту та його налаштувань і натиснення кнопки ОК, ці налаштування шрифту застосовуються до властивості шрифту у textBox1 (рис. 4.32-4.33).

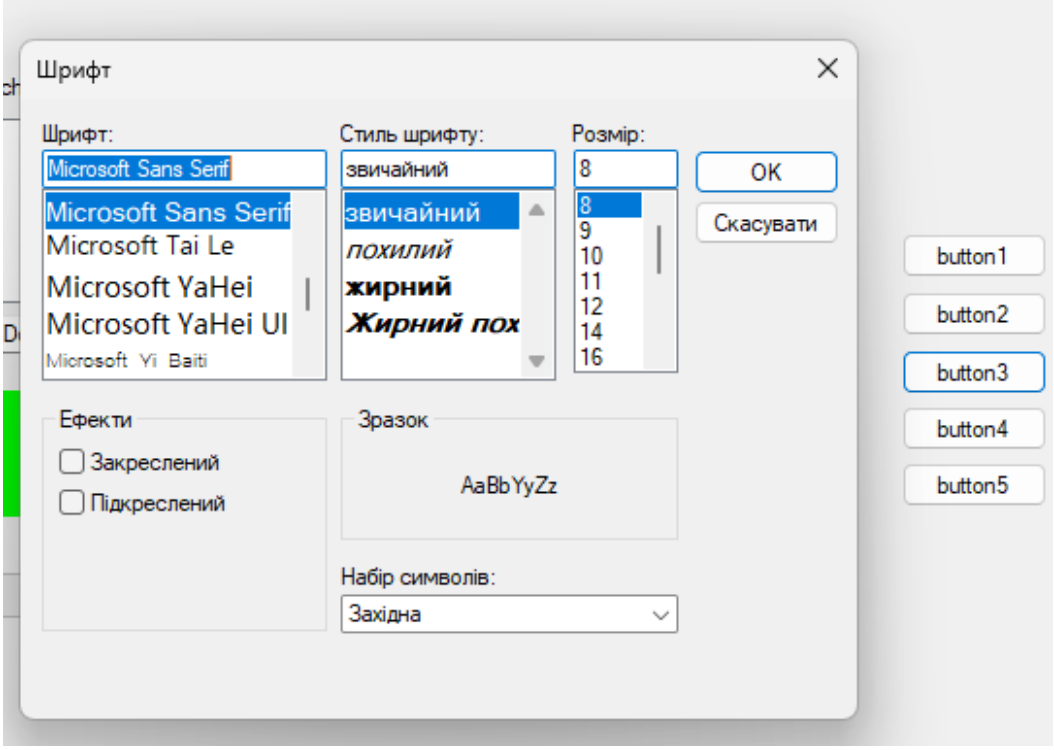


Рисунок 4.32 – Відкриття fontDialog1

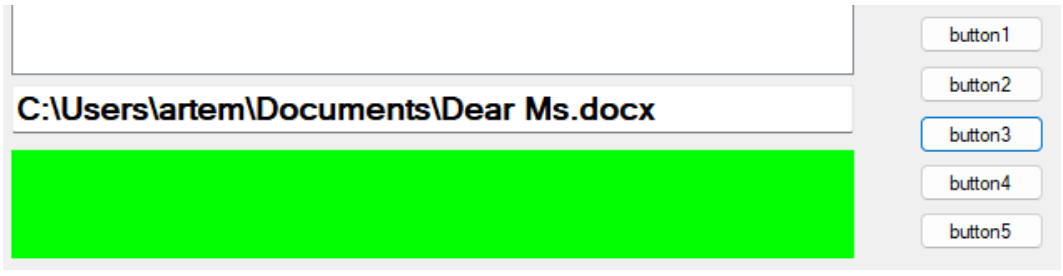


Рисунок 4.33 – Застосування параметрів шрифту до властивості у textBox1

Розглянемо обробник події натиснення на 4 кнопку (button4):

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e) {
        String^ textToAdd = textBox1->Text;
        if (radioButton1->Checked) {
                listBox1->Items->Add(textToAdd->ToUpper());
                label2->Text = textToAdd;  }
        else if (radioButton2->Checked) {
                listBox1->Items->Add(textToAdd->ToLower());
                label3->Text = textToAdd;  }
        else {
                listBox1->Items->Add(textToAdd);
                MessageBox::Show("Choose register", "Warning!"); }
        MessageBox::Show("You pressed button 4!", "Message");
        label1->Text = "Button 4 pressed"; }
```

По-перше, обирається текст з елементу textBox1, далі відбувається перевірка, яка з радіокнопок (radioButton) обрана.

У випадку, якщо вибране 1 значення, то значення тексту з textBox1 додається у listBox1, при чому відбувається зміна його на верхній регістр (textToAdd->ToUpper();), а також він додається у значення тексту label2 (рис. 4.34).

У випадку, якщо вибране 2 значення, то значення тексту з textBox1 додається у listBox1, при чому відбувається зміна його на нижній регістр (textToAdd-> ToLower ();), а також він додається у значення тексту label3 (рис. 4.35).

У випадку якщо не обрано жодного зі значень radioButton, то текст у listBox1 додається за замовчуванням, а також створюється діалогове вікно MessageBox з повідомленням Choose register (рис. 4.36).

Після цього створюється ще одне діалогове вікно MessageBox з повідомленням You pressed button 4, а також у label1 додається текст Button 4 pressed (рис. 4.37-4.38).

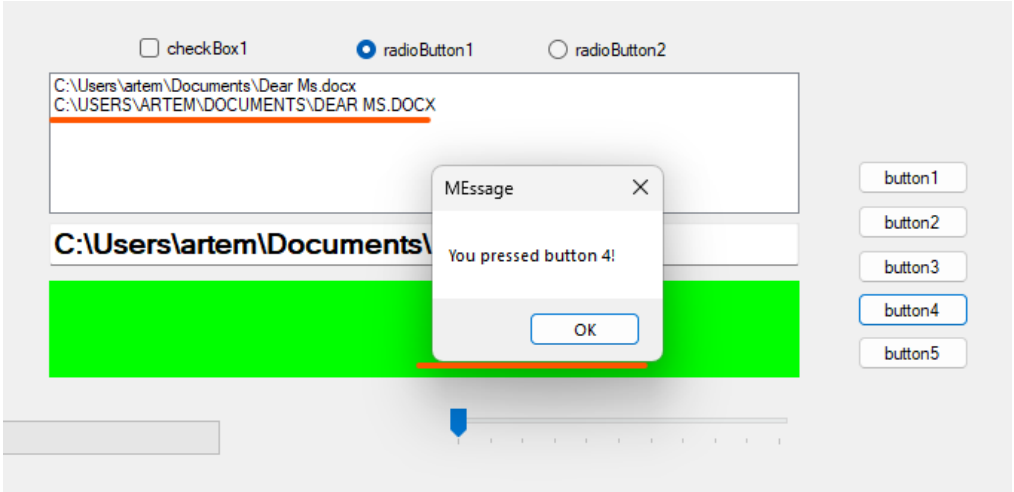


Рисунок 4.34 – Подія з додаванням тексту і значенням radioButton1

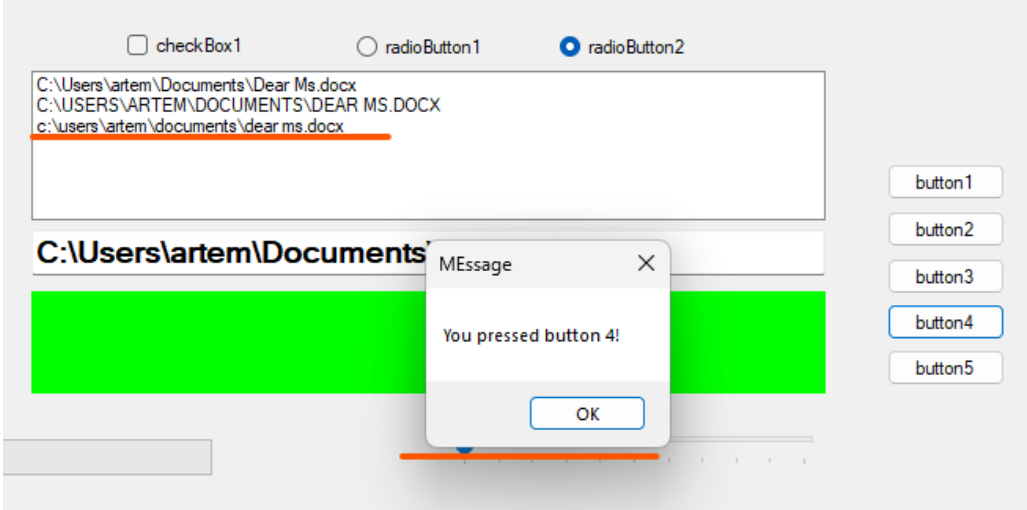


Рисунок 4.35 – Подія з додаванням тексту і значенням radioButton1

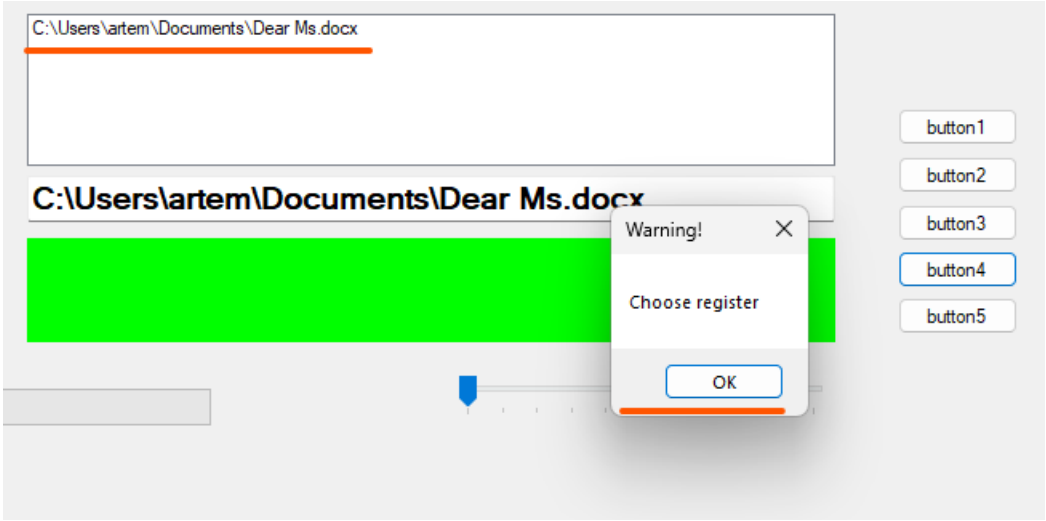


Рисунок 4.36 – Подія з додаванням тексту і не обраним значенням radioButton

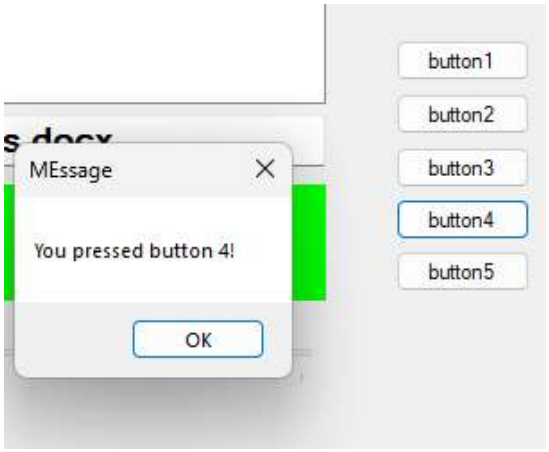


Рисунок 4.37 – Додатковий MessageBox

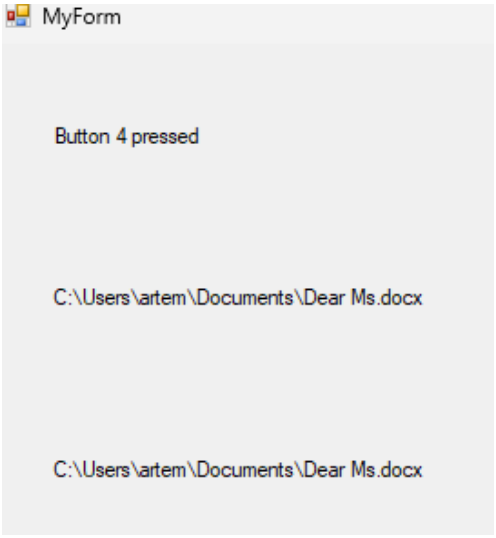


Рисунок 4.38 – Додавання у label1 тексту Button 4 pressed

Останньою є подія настискання кнопки 5, яка матиме наступний вигляд:

```
private: System::Void button5_Click(System::Object^ sender, System::EventArgs^ e) {
            if (checkBox1->Checked) {
              listBox1->Items->Add("Element added by button 5 (checked)"); }
            else {
                        listBox1->Items->Add("Element added by button 5 (not checked)");          }
            progressBar1->Value = (progressBar1->Value + 10) % (progressBar1->Maximum + 1); }
```

Ця подія буде мати наступний функціонал. Спочатку перевіряється, чи стоїть прапорець у checkBox1. Якщо він стоїть, то у listBox1 додається текст Element added by button 5 (checked) (рис. 4.39), якщо прапорець не стоїть, то listBox1 додається текст Element added by button 5 (not checked) (рис. 4.40).

Також під час кожного натиснення значення progressBar1 змінюється відповідно до такого запису:

```
progressBar1->Value + 10) % (progressBar1->Maximum + 1).
```

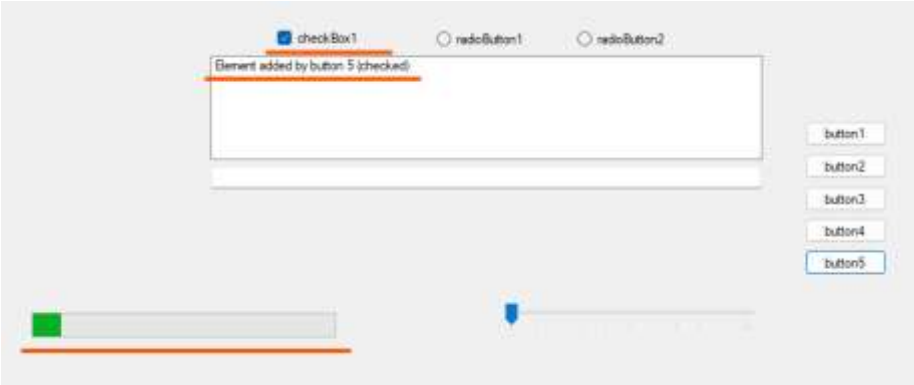


Рисунок 4.39 – Подія з перевіреним (активним) прапорцем checkBox1

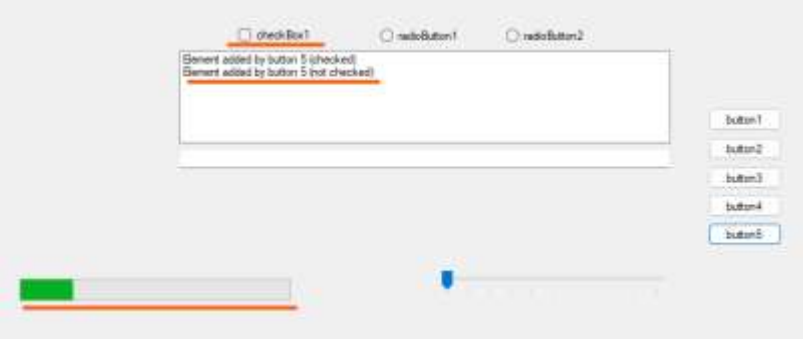


Рисунок 4.40 – Подія з неперевіреним (неактивним) прапорцем checkBox1

4.5.2 Другий приклад

Розглянемо ще один приклад створення форми. Приблизний вигляд представлено на рис. 4.41.

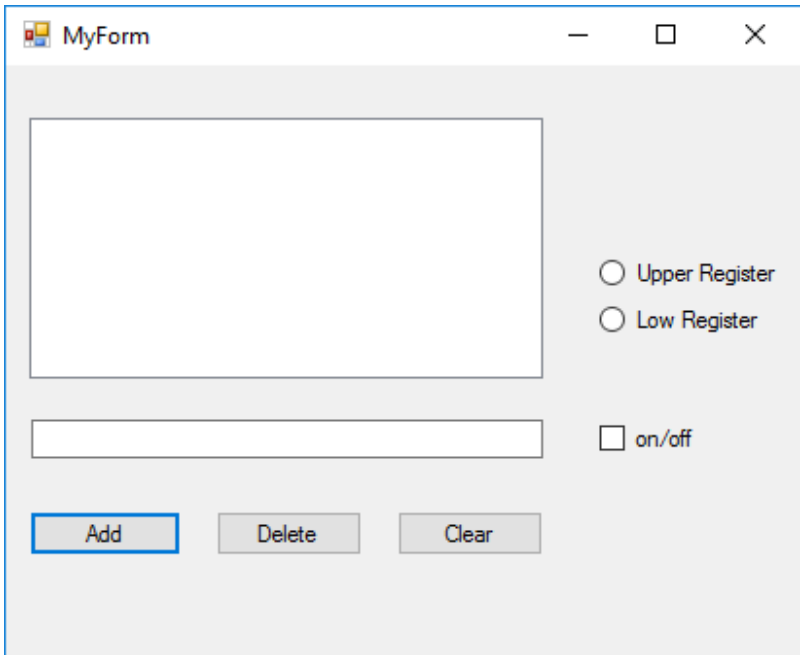


Рисунок 4.41 – Форма, що розроблюється

Вона складається з таких елементів:

– 3 елемента Button;
– 2 елемента radioButton;
– 1 елемента checkBox;
– 1 елемент textBox;
– 1 елемент listBox.

Додамо обробники для всіх кнопок та елементів.

Обробник кнопки Add має наступний вигляд:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{ String ^s;
 s=textBox1->Text;
 if (s->Length)
            {if(radioButton1->Checked) s = s->ToUpper();
             if(radioButton2->Checked) s = s->ToLower();
             listBox1->Items->Add(s); }
else MessageBox::Show("No text to add"); }
```

Спочатку створюється рядок, в який потім записується значення, додане у textBox1. Потім відбувається перевірка – якщо є якийсь текст, то відбувається наступна перевірка за допомогою radioButton. У видпаку якщо обрана 1 радіокнопка, то текст додається у елемент listBox1 великими літерами (ToUpper), якщо обрана друга радіокнопка – малими (ToLower). У випадку коли тексту у textBox1 немає, то видається повідомлення про помилку – No text to add.

Обробник другої кнопки (Delete) має наступний вигляд:

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e)
{ int i = listBox1->SelectedIndex;
if (i == -1)MessageBox::Show("Нема обраних елементів");
else listBox1->Items->RemoveAt(i); }
```

За допомогою властивості SelectedIndex отримуємо поточний індекс елементу з listBox1 (обраний рядок), також перевіряємо, чи обрані якісь елементи у listBox1, якщо ні, то видається повідомлення, що такі елементи не обрані. Якщо текст є і отриманий індекс цього елементу, то виконується метод RemoveAt(i), де i – індекс елементу у listBox1.

Обробник третьої кнопки (Clear) має наступний вигляд:

```
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e)
{int i = listBox1->Items->Count;
 for (int j = 0; j < i; j++)listBox1->Items->RemoveAt(0); }
```

Спочатку отримуємо кількість елементів у listBox1 за допомогою властивості Count, далі за допомогою циклу видаляємо елементі, кількість яких і є кількості елементів у listBox1. Також слід зауважити, що елементи видаляються зверху (з індексом 0) за допомогою RemoveAt(0).

Останньою є подія на checkBox1, яка має наступний вигляд:

```
private: System::Void checkBox1_CheckedChanged(System::Object^                sender, System::EventArgs^  e) {
```

```
bool chState = checkBox1->Checked::get();
if (chState)
{       button1->Enabled::set(true);
        button2->Enabled::set(true);
        button3->Enabled::set(true);
}
else {  button1->Enabled::set(false);
        button2->Enabled::set(false);
        button3->Enabled::set(false);           }   }
```

Спочатку перевіряється стан checkBox1 за допомогою Checked::get(). Якщо він повертає значення true, то всі 3 кнопки стають активними за допомогою Enabled::set(true);, якщо прапорець не стоїть, то всі кнопки є неактивними (Enabled::set(false);).

4.6 Контрольні завдання

1. Пояснити відмінності у властивостях та порядку створення діалогових вікон модального та немодального типу.
2. Пояснити порядок створення та використання ресурсів у .NET-програмах.
3. Пояснити особливості обробки повідомлень діалогових вікон.
4. Пояснити можливість використання діалогового вікна як головного вікна програми.
5. Пояснити порядок ініціалізації стандартних елементів керування у діалогових вікнах.
6. Розробити .NET -програму, у якій діалогове вікно реалізувати головним вікном програми.
7. Розробити .NET -програму діалогового типу із реалізацією списку, поля редагування, кнопок додавання та вилучення елементів зі списку.
8. Розробити .NET -програму із реалізацією обробників вертикальної та горизонтальної смуг прокручування.
9. Розробити .NET -програму діалогового типу із реалізацією списку, поля редагування, кнопок додавання та вилучення елементів зі списку, контрольних перемикачів та селекторних кнопок.
10. Пояснити назву “спільні елементи керування”.
11. Пояснити порядок використання панелей інструментів.
12. Пояснити порядок використання спливаючих підказок.
13. Пояснити порядок використання стрілок, регуляторів, індикаторів.
14. Пояснити порядок побудови програми із вікном анімації.

15.Пояснити порядок побудови програми із вікном анімації.
16.Пояснити порядок використання списків зображень.
17.Пояснити порядок використання вікон перегляду дерев.
18.Пояснити порядок використання закладок.
19. Пояснити порядок використання вікон перегляду списків.
20. Розробити програму із використанням панелі інструментів у діалоговому вікні.
21. Розробити програму, що у головному вікні реалізує елементи керування типу стрілка, регулятор, індикатор.
22. Розробити програму, що у головному вікні реалізує регулятор гучності звуку.
23. Розробити програму, що реалізує вікно перегляду дерев і пов'язує його з іменами файлів певного логічного диску вашого комп'ютера.
24. Розробити програму, що у головному вікні об'єднує вікно перегляду дерев і вікно перегляду списків, забезпечуючи одночасне зв'язане редагування обох структур подання даних, у якості прикладу множини даних оберіть певний каталог файлів на логічному диску комп'ютера.

5 РОБОТА З ГРАФІКОЮ

5.1 Загальні особливості використання графічних об’єктів

Робота з графікою у .NET не є складною, у порівнянні, наприклад, з MFC. В принципі, .NET і C++/CLI використовують схему подібного типу, коли із об’єктом Windows Form (або для MFC об’єктом CFrameWnd або CDialog) зв’язується спеціальний об’єкт, який для .NET є об’єктом графіки (в MFC його роль виконує контекст пристрою). Необхідність об’єкта графіки в .NET навіть простіше пояснити за вимогу контекста пристрою в MFC.

Об’єкт графіки і виведення у форму вже згадувалося раніше (приклад 3.2), і прощено це означає три кроки: створення об’єкта графіки, форматування тексту, виведення тексту у задані координати:

```
Graphics^ g = MyForm::CreateGraphics();
g->DrawString(“My Sample Text”, gcnew Drawing::Font("Arial", 10), Brushes::Black, 20, 20);
```

Подібна схема буде використана і для інших графічних функцій.

5.2 Виведення зображень

Найпростіший варіант виведення зображень реалізує функція

```
void DrawImage( Image^ image, int x, int y );
void DrawImage( Image^ image, Point point );
```

Її параметрами є об’єкт зображення, який необхідно вивести, і координати лівого верхнього кута зображення в системі координат форми. Координати початку виведення зображення можна задати окремими x і y або розмістити ті ж x і y всередині точки point. Для зручності, можна використати також версії функції DrawImage з float координатами і точкою.

Нагадуємо, що відлік системи координат у формі також розпочинається з її лівого верхнього кута.

Якщо у формі задати кнопку button1 з назвою «Image» і в теці проєкту мати файл зображення «LandRoverDefender.png», обробник цієї кнопки може виглядати відповідно до Прикладу 5.1, а результат виведення – видповідно до рис. 5.1:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
	Graphics^ g = MyForm::CreateGraphics();
	Image^ newImage = Image::FromFile("LandRoverDefender.png");
	g->DrawImage(newImage,0,0);
}
```

Приклад 5.1 – Обробник з виведенням зображення

Рисунок 4.1 – Виведення зображення в форму

Аналогічно, функція DrawImage() реалізує виведення файлів форматів bmp, jpg, gif, tiff. В цілому, DrawImage() має близько 25 варіантів реалізації, що відрізняються типами даних координат та різним розмірам зображення, що виводиться. Наприклад, якщо необхідно звузити зображення до вказаних розмірів 250x150:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
	Graphics^ g = MyForm::CreateGraphics();
	Image^ newImage = Image::FromFile("LandRoverDefender.png");
	g->DrawImage(newImage,0,0, 250, 150);
}
```

Приклад 5.2 – Обробник з виведенням зображення з розмірами 250x150

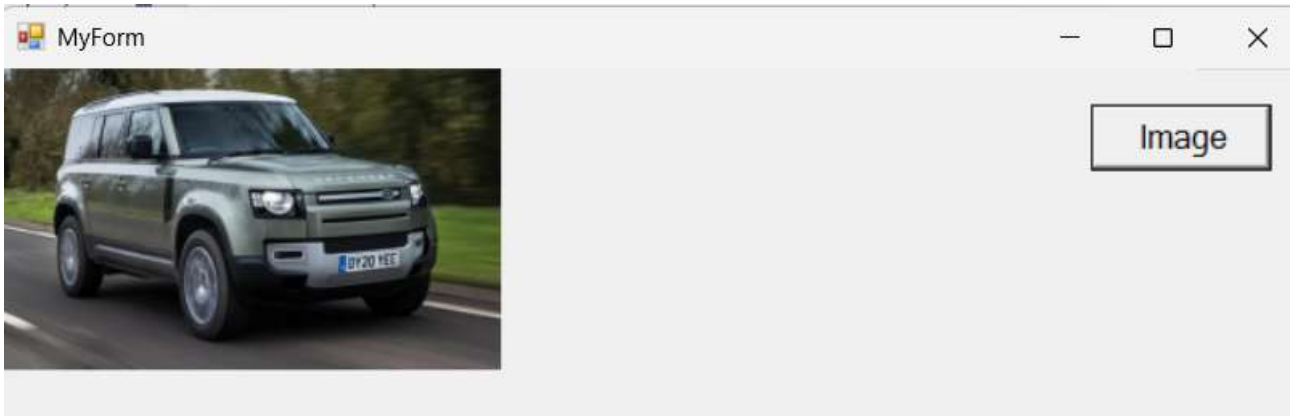


Рисунок 5.2 – Виведення зображення зі вказаними розмірами в форму

5.3 Відображення ліній

Для виведення ділянок прямих діній використовується функція DrawLine(), яка відображає лінію, яка поєднує дві точки, задані координатними парами, причому x- і y-координати або задаються окремо, або формують точки. Підтримуються значення цілого та плаваючого типу:

```
void DrawLine( Pen^ pen, int x1, int y1, int x2, int y2 );
void DrawLine( Pen^ pen, Point pt1, Point pt2 );
void DrawLine( Pen^ pen, PointF pt1, PointF pt2 );
void DrawLine( Pen^ pen, float x1, float y1, float x2, float y2 );
```

Характерною рисою усіх функцій вище і нижче є динамічне використання об’єнтів пера (Pen), яке створюється і видаляється динамічно. Для об’єкта пера має вказуватися колір (їх визначено багато) і товщина у пікселях.

Приклад 4.3 показую код обробника з трьома незалежними лініями різних кольорів та товщині, а Рисунок 4.3 ілюструє результат застосування обробника.

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        int x1=50, y1=50, x2=50, y2=250, x3=250, y3=250, x4=250, y4=50;
        g->DrawLine(gcnew Drawing::Pen(Color::Red,15), x1, y1, x2, y2);
        g->DrawLine(gcnew Drawing::Pen(Color::Green,10), x2, y2, x3, y3);
        g->DrawLine(gcnew Drawing::Pen(Color::Blue, 5), x3, y3, x4, y4);
}
```

Приклад 5.3 – Виведення різних за координатами, кольором і товщиною ліній

Рисунок 5.3 – Виведення трьох незалежних ліній

Для формування зв’язаних ліній .NET використає функцію DrawLines:

```
void DrawLines( Pen^ pen, array<Point>^ points );
void DrawLines( Pen^ pen, array<PointF>^ points);
```

Результатом застосування DrawLines є сформована масивом точок послідовність зв’язаних ліній (ламана лінія), причому на N точок будується N-1 ліній. Наприклад, задані на рис. 5.3 точки із доданою 5-ю точкою можуть сформувати масив:

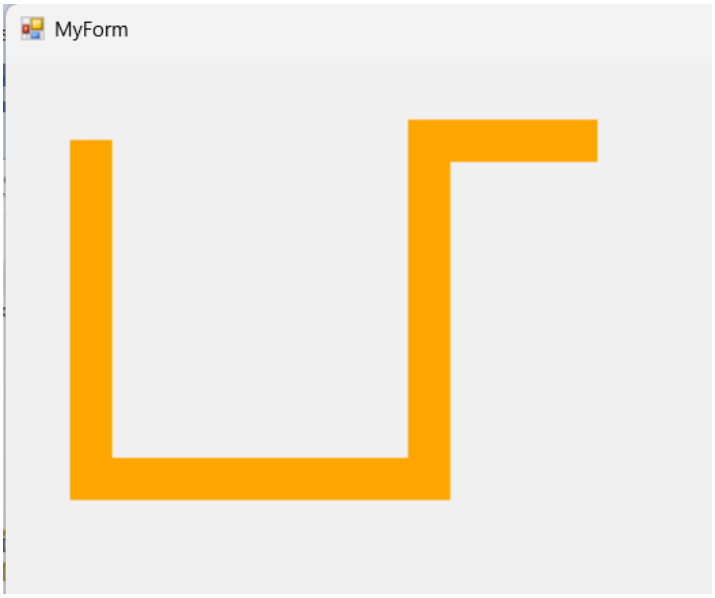
```
int x1 = 50, y1 = 50, x2 = 50, y2 = 250,
        x3 = 250, y3 = 250, x4 = 250, y4 = 50, x5 =350, y5 = 50;
array <Point>^ points =
{Point(x1,y1), Point(x2,y2), Point(x3,y3) ,Point(x4,y4) ,Point(x5,y5)};
```

Таким чином код для ламаної ліній виглядатиме так:

```
private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e)
{
 Graphics^ g = MyForm::CreateGraphics();
 int x1 = 50, y1 = 50, x2 = 50, y2 = 250,
    x3 = 250, y3 = 250, x4 = 250, y4 = 50, x5 =350, y5 = 50;
 array <Point>^ points =
{Point(x1,y1), Point(x2,y2), Point(x3,y3) ,Point(x4,y4) ,Point(x5,y5)};
 g->DrawLines(gcnew Drawing::Pen(Color::Orange, 25), points);          }
```

Приклад 5.4 – Формування зв’язаної послідовності ліній

Результат застосування прикладу показує Рис. 4.4.



Приклад 5.4 – Застосування функції DrawLines()

Обмеженням застосування функції DrawLines() є неможливість формування ділянок ламаної різними кольорами. Для різнокольорової ліній необхідно формувати окремі частини.

Логічним продовженням DrawLines() є функція формування полігону DrawPolygon(), яка приймає ті ж аргументи – перо і масив точок:

```
void DrawPolygon( Pen^ pen, array<Point>^ points );
void DrawPolygon( Pen^ pen, array<PointF>^ points );
```

Приклад 4.5 демонструє обробник з формуванням полігону, а Рис. 4.5 – результат його виконання.

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{Graphics^ g = MyForm::CreateGraphics();
        array <Point>^ points =
        {Point(50,50), Point(50,250), Point(250,250),
         Point(250,50),Point(350,50),Point(350,100)};
         g->DrawPolygon(gcnew Drawing::Pen(Color::Black, 10), points);
}
```

Приклад 5.5 – Формування полігона

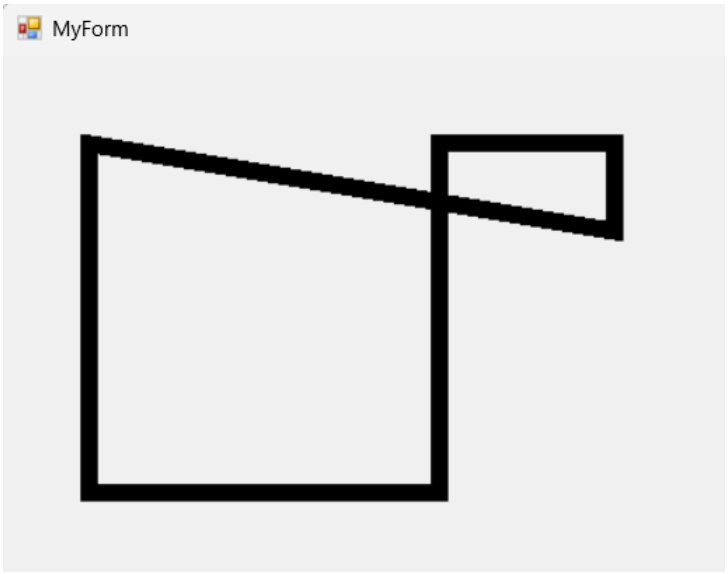


Рисунок 5.5 – Застосування функції DrawPolygon()

Розвитком функції DrawPolygon() є DrawClosedCurve(), результатом якої є перетворення полігона на плавний апроксимований об’єкт.

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{
 Graphics^ g = MyForm::CreateGraphics();
 array <Point>^ points =
 {Point(50,50), Point(50,250), Point(250,250) ,Point(250,50) ,Point(350,50)};
 g->DrawClosedCurve(gcnew Drawing::Pen(Color::BlueViolet, 25), points);
}
```

Приклад 5.6 – Формування замкненої апроксимованої кривої

Рисунок 5.6 – Застосування функції DrawClosedCurve()

Розглядаючи лінії криволінійного типу слід вказати і на функцію формування Безье-сплайнів DrawBezier(). Для формування таких ліній використається інший підбір аргументів – чотири точки сплайна (можуть бути як точками, так і окремими координатами x і y).

```
void DrawBezier( Pen^ pen, Point pt1, Point pt2, Point pt3, Point pt4 );
void DrawBezier( Pen^ pen, PointF pt1, PointF pt2, PointF pt3, PointF pt4 );
```

Спочатку вказується перша точка, потім друга і третя (що є контрольними), кінцева точка. Примітно, що сплайн майже ніколи не проходить через контрольні точки, що забезпечують витягування сплайна в необхідному напрямку.

Приклад 5.7 демонструє обробник з формуванням сплайна, а рис. 5.7 – результат його виконання.

```
private: System::Void button6_Click(System::Object^ sender, System::EventArgs^ e)
{
        Graphics^ g = MyForm::CreateGraphics();
        Pen^ blackPen = gcnew Pen(Color::DarkGray, 5.0f);
        Point start = Point(100, 100), control1 = Point(200, 10),
                control2 = Point(350, 50), end = Point(500, 100);
        g->DrawBezier(blackPen, start, control1, control2, end);
}
```

Приклад 5.7 – Формування сплайна функцією DrawBezier()

Рисунок 5.7 – Застосування функції DrawBezier()

Функція формування окремих сплайнів має продовження у функції DrawBeziers(), яка замість формування одного сплайна формує список сплайнів, на основі масиву точок. При цьому, перші чотири точки масиву є основою для побудови 1-го сплайна; 2-й сплайн використовує кінцеву точку 1-го сплайна, як

початкову; 3-й сплайн використовує кінцеву точку 2-го сплайна, як свою початкову і т.д.

Приклад 5.8 демонструє обробник з формуванням списку сплайнів, а рис. 5.8 – результат його виконання.

```
private: System::Void button7_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        Pen^ blackPen = gcnew Pen(Color::Green, 7.0f);
        Point start1 = Point(100, 100), control1 = Point(200, 10),
                control2 = Point(350, 50), start2 = Point(500, 100),
                control3 = Point(350, 150), control4 = Point(200, 150),
        end = Point(100, 100);
        array <Point>^ ptsOfSplines =
{start1,control1,control2, start2,control3, control4, end };
        g->DrawBeziers(blackPen, ptsOfSplines);
}
```

Приклад 5.8 – Формування сплайна функцією DrawBezier()

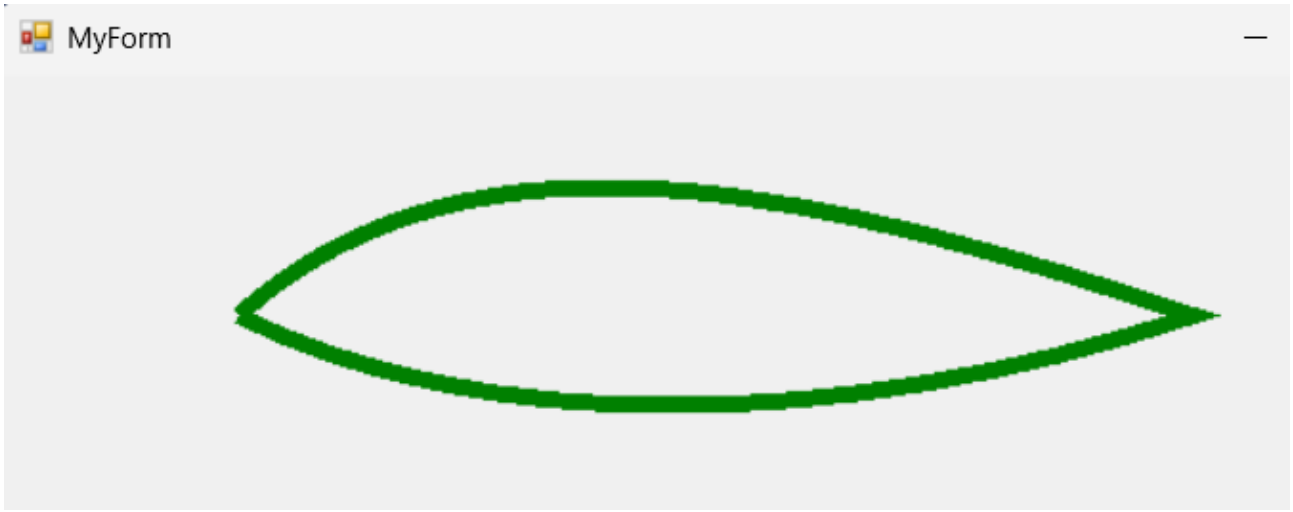


Рисунок 5.7 – Побудова двох зв’язаних сплайнів функцією DrawBeziers()

5.4 Відображення прямокутників

У .NET прямокутник відображається функцією DrawRectangle(), як об’єкт, який визначається початковою точкою (верхній лівий кут), шириною та висотою або прямокутною областю типу Rectangle, що містить ті ж початкові координати, ширину і висоту. Прямокутник зображується об’єктом пера Pen:

```
void DrawRectangle( Pen^ pen, int x, int y, int width, int height );
void DrawRectangle( Pen^ pen, float x, float y, float width, float height );
void DrawRectangle( Pen^ pen, Rectangle rect );
```

Приклад 5.9 і рис. 5.9 ілюструють побудову двох прямокутників – на основі параметрів типу float. Для червоного прямокутника початкова координата, ширина і висота вказані окремо; для синього – всередині об'єкта Rectangle:

```
        private: System::Void button8_Click(System::Object^ sender, System::EventArgs^ e)
        {Graphics^ g = MyForm::CreateGraphics();
float x = 10.0F, y = 10.0F, width = 240.0F, height = 200.0F;
g->DrawRectangle(gcnew Pen(Color::OrangeRed, 6.5f), x, y, width, height);
g->DrawRectangle(gcnew Pen(Color::Blue, 6.5f),Rectangle(70.0F,60.0F,120.0F,100.0F));
        }
```

Приклад 5.9 – Формування прямокутників з параметрами типу float

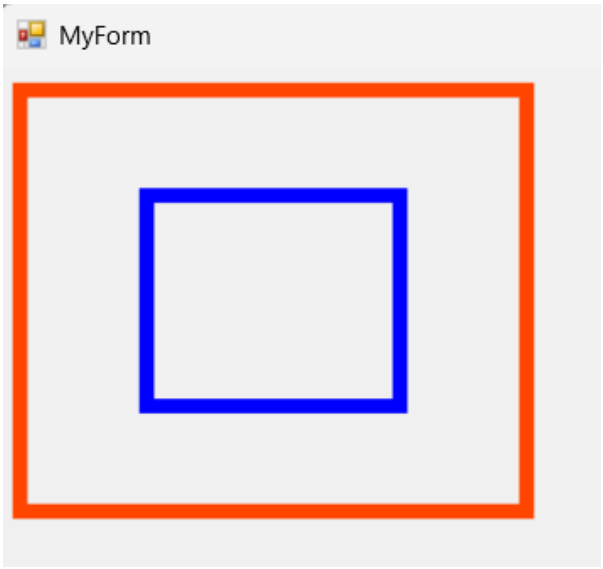


Рисунок 5.9 – Побудова прямокутників функцією DrawRectangle()

У випадку, коли необхідно побудувати декілька прямокутників одного кольору, корисною є функція DrawRectangles(), будує список прямокутників на основі масиву об'єктів Rectangle (показано у прикладі 5.10 та на рис. 5.10). Слід зазначити, що спосіб задання декількох об'єктів списком є надзвичайно зручним.

```
private: System::Void button9_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        array <Rectangle>^ rects = {Rectangle(40.0F,50.0F,150.0F,120.0F),
        Rectangle(100.0F,80.0F,150.0F,120.0F),Rectangle(160.0F,110.0F,150.0F,120.0F) };
        g->DrawRectangles(gcnew Pen(Color::Green, 7.5f), rects);
}
```

Приклад 5.10 – Формування списку прямокутників

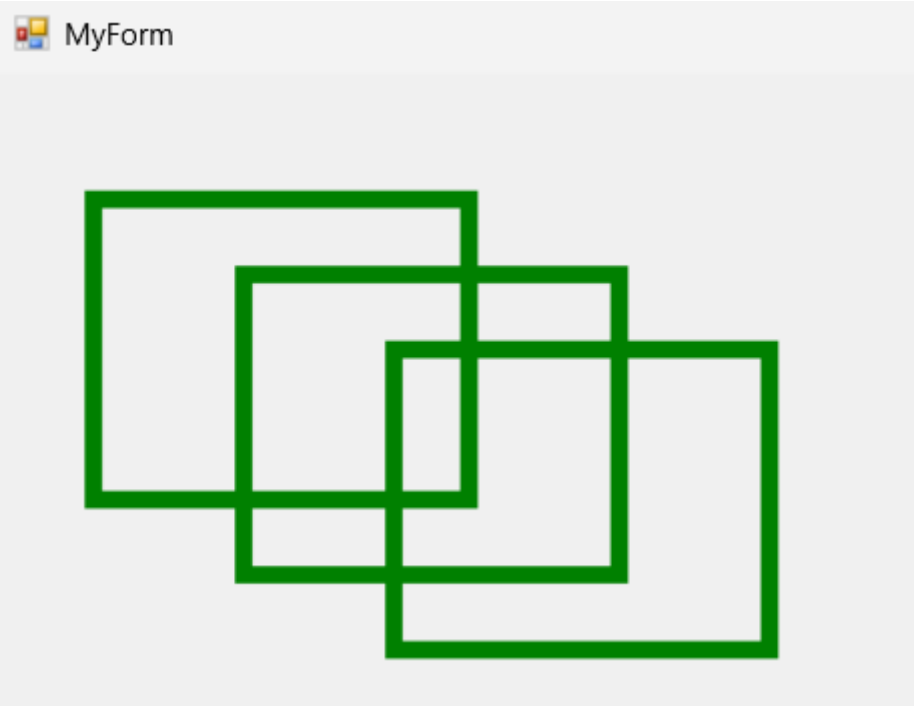


Рисунок 5.10 – Побудова списку прямокутників функцією DrawRectangles()

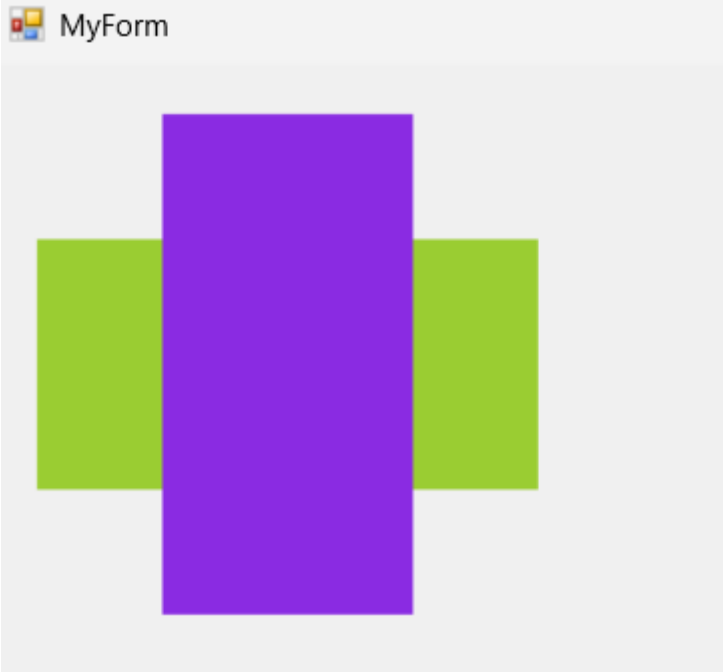
Якщо ж необхідно заповнити кольором внутрішню частину прямокутника, в нагоді стає функція FillRectangle(), яка для побудови об’єкта використає ті ж координати, ширину і висоту, що і DrawRectangles(). Особливістю FillRectangle() (як і інших Fill-функцій) є використання об’єкту SolidBrush() для визначення кольору заповнення об’єкта. Приклад 5.11 показує рис. 5.11.

```
void FillRectangle( Brush^ brush, int x, int y, int width, int height );
void FillRectangle (Brush^ brush, Rectangle rect );
```

```
private: System::Void button10_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        SolidBrush^ yellBrush = gcnew SolidBrush(Color::YellowGreen);
        SolidBrush^ blBrush = gcnew SolidBrush(Color::BlueViolet);
        int x = 20; int y = 70; int width = 200; int height = 100;
        Rectangle rect1 = Rectangle(x, y, width, height),
                  rect2 = Rectangle(x+50, y-50, height, width);
        g->FillRectangle(yellBrush, rect1);
        g->FillRectangle(blBrush, rect2);
}
```

Приклад 5.11 – Приклад заповнених перпендикулярних прямокутників

Як видно з прикладу і рисунка, прямокутники не є прозорими. Додатково зазначимо, що можливість побудови декількох прямокутників закладена в функцію FillRectangles(), що є аналогічною DrawRectangles().



Приклад 5.11 – Приклад із заповненими прямокутниками

Одразу зазначимо, що, крім FillRectangle(), заповнені версії об’єктів є для полігона – FillPolygon, замкненої кривої FillClosedCurve(), еліпса – FillEllipse(), сектора – FillPie(), тобто для усіх замкнених об’єктів.

5.5 Відображення еліпсів, дуг та секторів

Криволінійні об’єкти, зокрема еліпси та їх складові у формі дуг та секторів є досить складними для формального опису координатним способом. Проте, для їх відображення існують прості прийоми. Зокрема, будь-який еліпс відображається як об’єкт, вписаний у прямокутник. При цьому, для опису еліпса і функції DrawEllipse() вказуються ті ж самі параметри, що і для прямокутника: перо, початкова точка (верхній лівий кут прямокутника), ширина і висота (або всі вказані параметри вписуються в об’єкт Rectangle).

```
void DrawEllipse( Pen^ pen, int x, int y, int width, int height );
void DrawEllipse( Pen^ pen, float x, float y, float width, float height );
void DrawEllipse( Pen^ pen, Rectangle rect );
void DrawEllipse( Pen^ pen, RectangleF rect );
```

Приклад 5.12 і рис. 5.12 показують приклад побудови еліпса і прямокутника з однаковими координатами.

```
private: System::Void button11_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        float x = 10.0F, y = 10.0F, width = 240.0F, height = 200.0F;
        g->DrawRectangle(gcnew Pen(Color::Red, 4.5f), x, y, width, height);
        g->DrawEllipse(gcnew Pen(Color::Green, 4.5f), x, y, width, height);
}
```

Приклад 5.12 – Обробник із прямокутником і визначеним за його координатами еліпсом

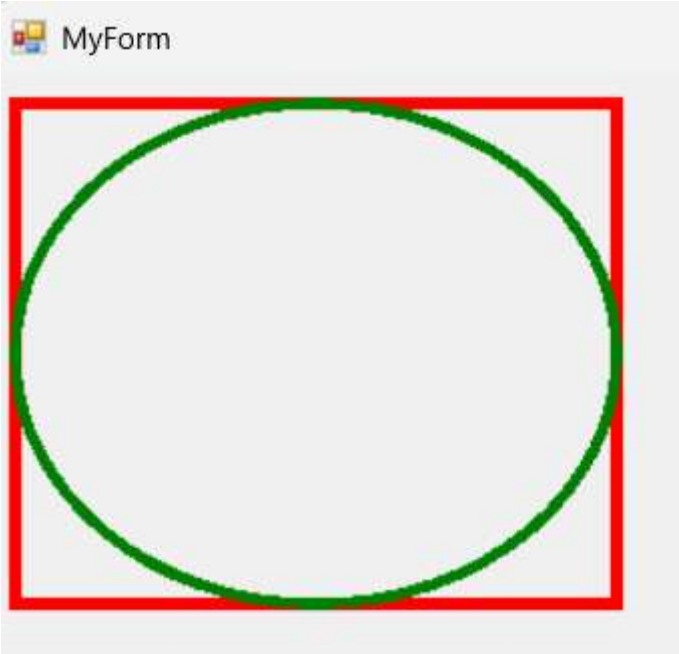


Рисунок 5.12 – Приклад із прямокутником і вписаним до нього еліпсом

Дуга (Arc) є складовою еліпса і відображає частину його криволінійної форми. Для відображення дуги у функції DrawArc() використається той самий принцип вписаного у прямокутник еліпса – задаються координати початкової точки, ширина і висота базового прямокутника, до яких додаються початкова і кінцева точка дуги, як частини еліпса:

```
void DrawArc( Pen^ pen, int x, int y, int width, int height,
int startAngle, int sweepAngle );
void DrawArc( Pen^ pen, float x, float y, float width,
float height, float startAngle, float sweepAngle );
void DrawArc( Pen^ pen, Rectangle rect, float startAngle, float sweepAngle );
void DrawArc( Pen^ pen, RectangleF rect, float startAngle, float sweepAngle );
```

де startAngle є початковим кутом дуги, sweepAngle – її кутовою довжиною. Обидва параметри задаються у градусах і відраховуються в системі координат центра еліпса відносно горизонтальної осі OX, традиційно спрямованої праворуч.

Приклад 5.13 і рис. 5.13 показують побудову двох дуг на основі кола, як еліпса, вписаного в квадрат:

```
private: System::Void button12_Click(System::Object^ sender, System::EventArgs^ e)
{
        Graphics^ g = MyForm::CreateGraphics();
        float x = 10.0F, y = 10.0F, width = 200.0F, height = 200.0F;
        g->DrawArc(gcnew Pen(Color::Red, 8.5f), x, y, width, height, 0.0,90.0);
        g->DrawArc(gcnew Pen(Color::Blue, 10.5f), x, y, width, height, 180.0, 90.0);
}
```

Приклад 5.13 – Обробник із двома дугами на основі кола

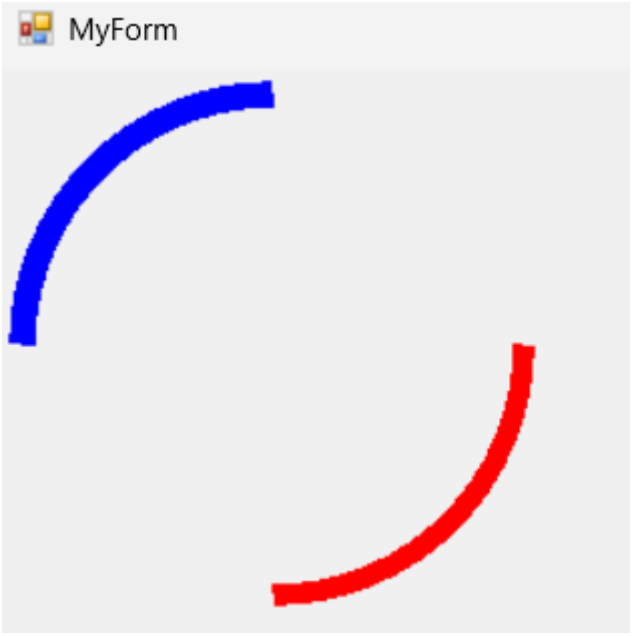


Рисунок 5.13 – Відображення двох дуг на основі кола

Сектор (Pie) є складовою еліпса і також відображає частину його криволінійної форми. Для відображення сектора у функції DrawPie() використається той самий принцип вписаного у прямокутник еліпса – задаються координати початкової точки, ширина і висота базового прямокутника, до яких додаються початкова і кінцева точка дуги сектора, як частини еліпса. Візуально Pie відрізняється від дуги наявністю двох радіусів, що з'єднують початок і кінець дуги з центром уявного еліпса.

```
void DrawPie( Pen^ pen, int x, int y, int width, int height, int startAngle, int sweepAngle );
void DrawPie( Pen^ pen, float x, float y, float width, float height, float startAngle, float sweepAngle );
```

```
void DrawPie( Pen^ pen, Rectangle rect, float startAngle, float sweepAngle );
void DrawPie( Pen^ pen, RectangleF rect, float startAngle, float sweepAngle );
```

Аналогічно сформована і функція заповненого сектора FillPie():

```
void FillPie( Brush^ brush, int x, int y, int width, int height, int startAngle, int sweepAngle );
void FillPie( Brush^ brush, float x, float y, float width, float height, float startAngle, float sweepAngle );
void FillPie( Brush^ brush, Rectangle rect, float startAngle, float sweepAngle );
void FillPie( Brush^ brush, RectangleF rect, float startAngle, float sweepAngle );
```

Приклад 5.14 і рисунок 5.14 показують побудову двох протилежних заповнених секторів на основі кола, як еліпса, вписаного в квадрат:

```
private: System::Void button13_Click(System::Object^ sender, System::EventArgs^ e)
{Graphics^ g = MyForm::CreateGraphics();
 float x = 10.0F, y = 10.0F, width = 200.0F, height = 200.0F;
 g->FillPie(gcnew SolidBrush(Color::Green), x, y, width, height, 90.0, 90.0);
 g->FillPie(gcnew SolidBrush(Color::Yellow), x, y, width, height, 270.0, 90.0);
}
```

Приклад 5.14 – Обробник із двома заповненими секторами

Рисунок 5.14 – Прикдад функції FillPie() для двох заповнених секторів

5.6 Приклад програми зі стрілковим годинником

Використання розглянутих вище графічних функції демонструє програма зі стрілковим годинником. Дана програма має враховувати наступне:

1. Моделювання стрілкового годинника має перетворювати цифровий час комп’ютера у геометричні координати струлок годинника;

2. Зображення стрілок годинника має використати полярну систему координат;

3. Центр полярної системи координат має бути розташований у центрі циферблата;

4. Програма має враховувати проміжні стани руху годинної (пересування кожні 15 хвилин) та хвилинної (кожні 30 секунд) стрілок годинника.

Запуск таймера годинника можна реалізувати під час запуску форми (як у прикладі 5.15) або натиском клавіші на формі, або в інший спосіб.

```
private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e)
{
        timer1->Interval = 1000;
        timer1->Start();
}
```

Приклад 5.15 – Запуск таймера у обробнику завантаження форми

Для програми зі стрілковим годинником обробник таймера реалізує графічне відображення циферблату і стрілок годинника відповідно до поточного часу в системі, фактично передворюючи час в координати стрілок годинника. Реалізація такої функції таймера показана у прикладі 5.16.

Основні дії функції таймера:

- отримання поточного системного часу;
- очищення циферблата від попереднього зображення стрілок;
- виведення поточного значення часу текстом у формі;
- визначення об’єктів пера для стрілок годинника;
- відображення циферблату та міток годин;
- формування поправок проміжних станів годинної та хвилинної стрілок;
- відображення стрілок у полярній системі координат із початком у центрі циферблату.

```
private: System::Void timer1_Tick(System::Object^ sender, System::EventArgs^ e)
{
 DateTime localDate = System::DateTime::Now;
 int hour = localDate.Hour, mint = localDate.Minute, secd = localDate.Second;
 Graphics^ g = MyForm::CreateGraphics();
 g->Clear(Color::White);
 String^ s = String::Format("{0}", localDate.ToString());
 g->DrawString(s, gcnew Drawing::Font("Arial", 10), Brushes::Black, 5.0, 5.0);
 Pen^ redPen = gcnew Pen(Color::Red, 4.0f);
 Pen^ bluePen = gcnew Pen(Color::Blue, 6.0f);
 Pen^ greenPen = gcnew Pen(Color::Green, 8.0f);
 double pi = System::Math::PI;
```

```
g->DrawEllipse(redPen, 10, 10, 400, 400);
for (int k = 0; k < 12; k++)
        g->DrawEllipse(redPen, 200 + 180 * System::Math::Cos(pi * k / 6),
                               200 + 180 * System::Math::Sin(pi * k / 6), 10, 10);
int i = 0, j = 0;
if (secd >= 30)i = 1; if (mint >= 15)j = 1; if (mint >= 30)j = 2;
if (mint >= 45)j = 3;  if (hour >= 12)hour = hour - 12;
hour = hour - 3; mint = mint - 15; secd = secd - 15;
double Hr = pi * hour / 6 + pi * j / 16;
double Mn = pi * mint / 30 + pi * i / 60;
g->DrawLine(redPen, 205, 205, 205 + 160 * System::Math::Cos(pi * secd / 30),
                    205 + 160 * System::Math::Sin(pi * secd / 30));
g->DrawLine(bluePen, 205, 205, 205 + 120 * System::Math::Cos(Mn),
                    205 + 120 * System::Math::Sin(Mn));
g->DrawLine(greenPen, 205, 205, 205 + 80 * System::Math::Cos(Hr),
                    205 + 80 * System::Math::Sin(Hr));
}
```

Приклад 5.16 – Реалізація обробника таймера для стрілкового годинника

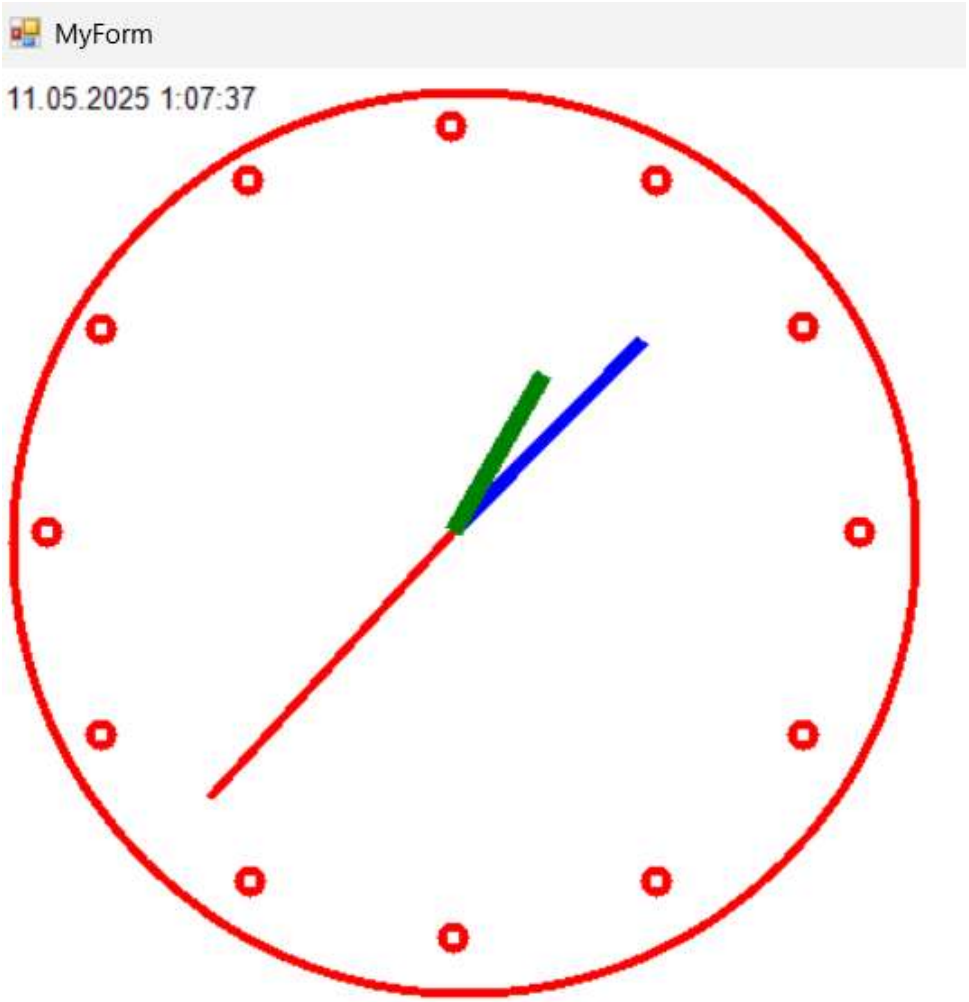


Рисунок 5.15 – Вигляд програми зі стрілковим годинником

5.7 Додаткові функції роботи з графікою

Додатковою функцією роботи з графікою є ExcludeClip(), яка визначає певну частину Windows Form забороненою для графіки. Тобто, певна визначена прямокутна площина форми буде вільною від графіки. ЇЇ формат:

```
void ExcludeClip( Rectangle rect ); // rect – прямокутна виключна область
```

Приклад 5.17 і рисунок 5.16 показують наявність прямокутної площини, вільної від діагональних ліній:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
        Graphics^ g = MyForm::CreateGraphics();
        Rectangle exRect = Rectangle(75, 75, 150, 300);
        g->ExcludeClip(exRect);
        g->DrawLine(gcnew Drawing::Pen(Color::Green, 10), 0, 0, 300, 300);
        g->DrawLine(gcnew Drawing::Pen(Color::Blue, 10), 300, 0, 0, 300);
}
```

Приклад 5.17 – Обробник з визначенням області, вільної від графіки

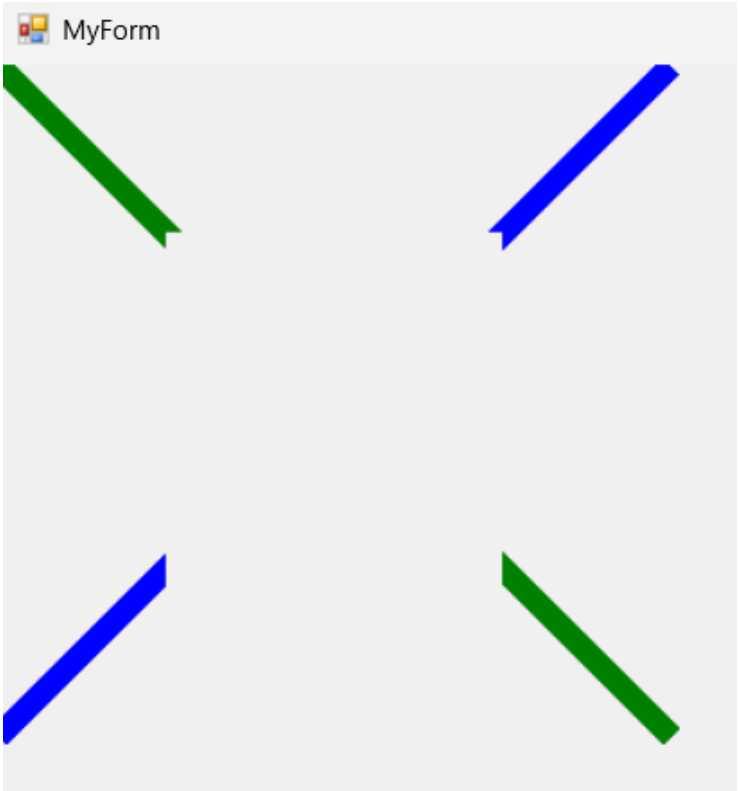


Рисунок 5.16 – Прикдад застосування функції ExcludeClip(), що вирізає прямокутну область перетину двох ліній

Ще однією допоміжною функцією є CopyFromScreen(), яка здійснює копіювання заданої області екрана ОС Windows у задані координати Windows Form програми. Її формат:

void CopyFromScreen(Point upperLeftSource, Point upperLeftDestination,
Size blockRegionSize);

де upperLeftSource – координата точки верхнього лівого кута початкового прямокутника (область екрану); upperLeftDestination - координата точки верхнього лівого кута початкового прямокутника (область вікна); blockRegionSize – розмір області.

Приклад 5.18 і рисунок 5.17 показують можливості функції CopyFromScreen().

```
private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e)
{
	Graphics^ g = MyForm::CreateGraphics();
	Point pointSource = Point(0, 0);
	Point pointDestination = Point(0, 0);
	Drawing::Size areaSize = Drawing::Size(300, 200);
	g->CopyFromScreen(pointSource, pointDestination, areaSize);
}
```

Приклад 5.18 – Обробник з функцією CopyFromScreen()

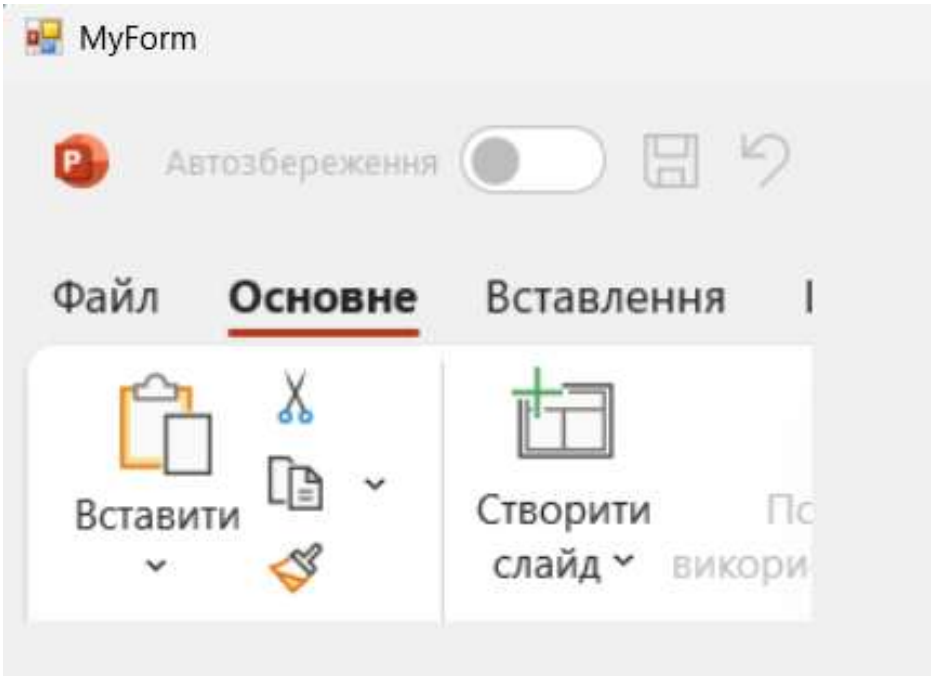


Рисунок 5.17 – Копіювання вернього лівого кута вікна PowerPoint у MyForm

5.8 Функції формування списку графічних об’єктів (Graphical Path)

У графічних системах, наприклад AutoCAD, використається метод, коли можна сформувати список об’єктів різного типу (лінії, дуги, смуги, текст і т.п.) і потім використати його, як єдине ціле. Подібна річ реалізована в .NET, хоча і має певні обмеження реалізації. Це функція відображення графічного шляху:

```
void DrawPath( Pen^ pen, GraphicsPath^ path );
```

DrawPath() сворює графічний шлях, складений з окремих об’єктів і відображає його із використанням певного пера (одного визначеного кольору).

Для використання Graphical Path підключається додатковий простір імен:

```
using namespace System::Drawing::Drawing2D;
```

Спочатку оголошується список графічних об’єктів Graphical Path, а потім до списку додаються необхідні об’єкти: прямі та криві лінії, полігони, дуги, сектора, криві, прямокутники, еліпси, інші шляхи (див. Приклад 4.18 і Рис. 4.18):

```
private: System::Void button6_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        // створення графічного списку (шляху)
        GraphicsPath^ graphPath = gcnew GraphicsPath;
        // додавання об'єктів список графічних об'єктів (шлях)
        graphPath->AddEllipse(50, 20, 160, 100);
        graphPath->AddEllipse(240, 20, 160, 100);
        graphPath->AddLine(220, 100, 220, 200);
        graphPath->AddLine(220, 200, 240, 200);
        graphPath->AddPie(Rectangle(120, 180, 200, 100),0,180);
        Pen^ blackPen = gcnew Pen(Color::Black, 3.0f);
        // Виведення графічного списку у форму.
        g->DrawPath(blackPen, graphPath);
}
```

Приклад 5.18 – Обробник з побудовою графічного шляху

5.9 Функції трансформації системи координат

У програмі зі стрілковим годинником (розділ 4.6) для відображення міток на циферблаті і визначення позицій стрілок необхідно переносити центр системи координат у центр циферблата. У прикладі 4.16 це робиться додаванням координат центру до відповідних координат стрілок, але цей спосіб дещо ускладнює програму і не є наочним. Більш зручним способом є зміна системи координат на рівні функцій самої платформи .NET. Розглянемо ці функції.

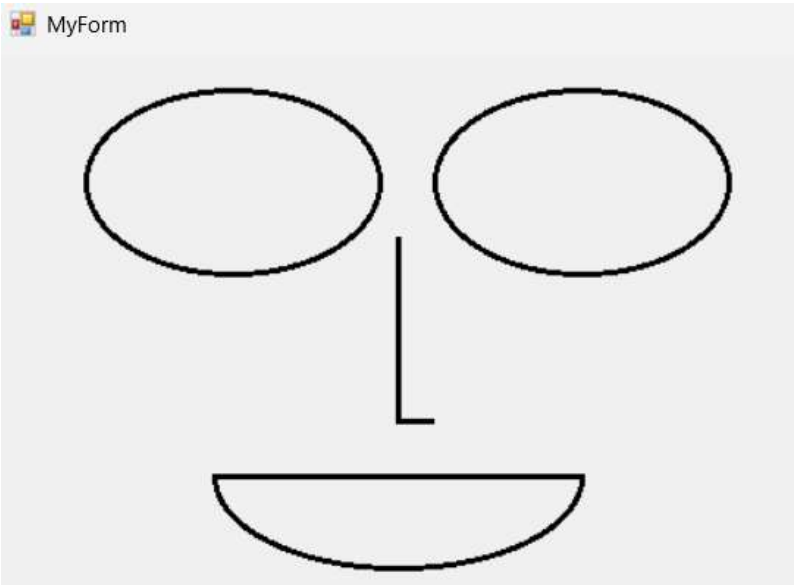


Рисунок 5.18 – Відображення заданого графічного шляху

а) перенесення центру системи координат реалізується TranslateTransform(), яка змінює початок системи координат на основі перемноження поточної матриці з матрицею перетворення, здійснює паралельний перенесення методами простору імен Graphics. Матриця переносу задається координатами нової системи координат

```
void TranslateTransform( float dx, float dy);
```

де dx і dy – x і y координати паралельного переносу, тобто координати, до яких переноситься центр системи координат програми.

Наприклад, необхідно відобразити декілька еліпсів, розташованих з певним зсувом по координаті x. В нагоді стає функція TranslateTransform():

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        Pen^ blackPen = gcnew Pen(Color::Brown, 3.0f);
        int x = 20, y = 20, width = 200, height = 100;
        for (int i = 0; i < 5; i++)
        {       g->TranslateTransform(25.0F + (float)i, 0.0F);
                g->DrawEllipse(blackPen, x, y, width, height);
        }
}
```

Приклад 5.19 – Обробник з функцією TranslateTransform()

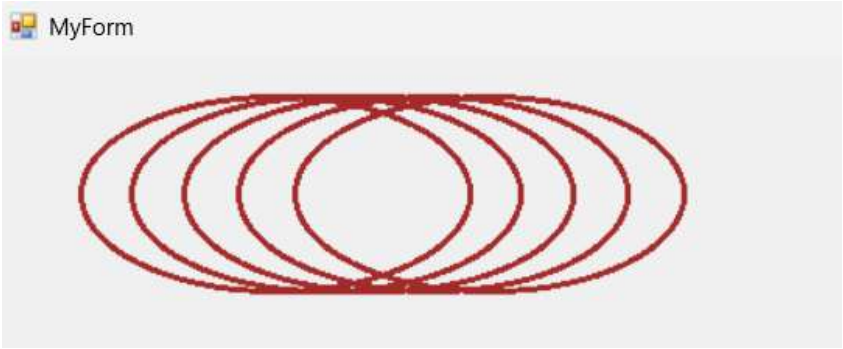


Рисунок 5.19 – Виведення еліпсів із використанням TranslateTransform():

б) обертання системи координат навколо її центра реалізується RotateTransform():

```
void RotateTransform( float angle ); // angle – кут обернення в градусах.
```

Слід зазначити, що без цієї функції неможливо зобразити під певним кутом прямокутник, еліпс, або зображення.

Приклад 4.20 та Рис. 4.20 демонструють поворот еліпсів на заданий кут.

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        Pen^ blackPen = gcnew Pen(Color::Blue, 3.0f);
        int x = 100; int y = 20;
        int width = 200; int height = 100;
        for (int i = 0; i < 5; i++)
        { g->RotateTransform((float)i * 4.0F);
          g->DrawEllipse(blackPen, x, y, width, height);
        }
}
```

Приклад 5.20 – Обробник з функцією RotateTransform()

Рисунок 5.20 – Поворот еліпсів із використанням RotateTransform()

в) зміна масштабу системи координат реалізується ScaleTransform():

Функція ScaleTransform() забезпечує зміну масштабу шляхом використання матриці перетворення простору імен Graphics:

```
void ScaleTransform( float sx, float sy );  // sx і sy коефіцієнти масштабування по осях


private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e)
{       Graphics^ g = MyForm::CreateGraphics();
        Pen^ blackPen = gcnew Pen(Color::Black, 2.0f);
        int x = 10; int y = 20;
        int width = 40; int height = 40;
        for (int i = 0; i < 3; i++)
        {       g->ScaleTransform(2.0F, 2.0F);
                g->DrawRectangle(blackPen, x, y, width, height);
        }
}
```

Приклад 5.21 – Обробник з функцією ScaleTransform()

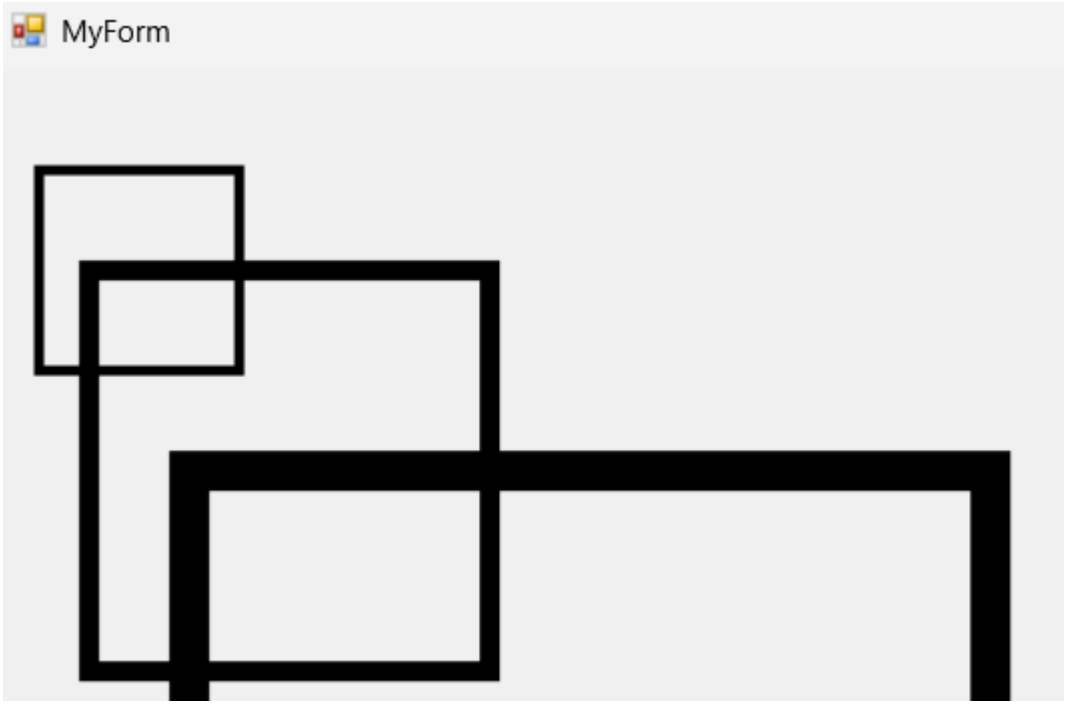


Рисунок 5.21 – Виведення прямокутників із зростаючим масштабом

5.10 Приклад програми з транформацією системи координат

Для демонстрації функцій перетворення системи координат розглянемо програму, яка імітує роботи маніпуляційного робота, що обслуговує певні робочі місця. При цьому програма зображає роботизовану систему у позиції «вигляд зверху», а сама роботизована система складається з маніпулятора, зобра-

женого прямокутниками його суглобів та кола в основі; прямокутників, що зображають обслуговувані робочі місця, що і показано на рис. 5.22.

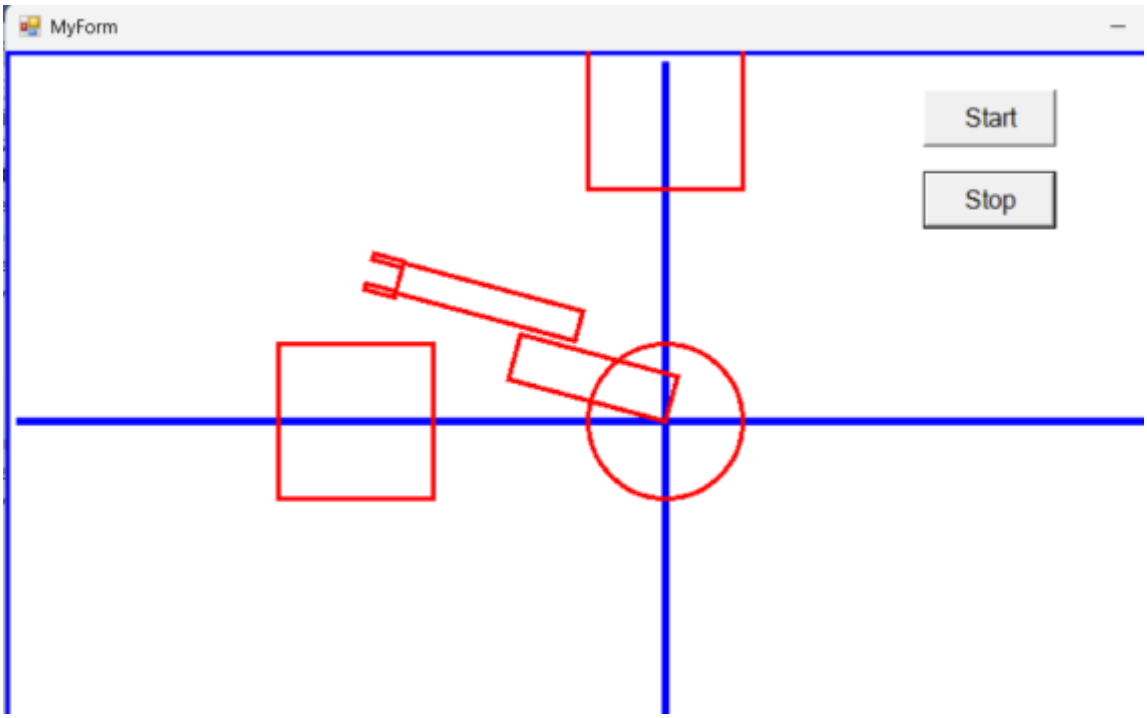


Рисунок 5.22 – Вигляд програми симуляції роботизованої системи

Для забезпечення роботи програми у базовому проекті необхідно:

а) визначити глобальні змінні (в верхій чатині файлу MyForm.h), відповідні за кут обертання, пов’язаний з таймером і напрям обертання (за/проти годинниковою стрілкою):

```
int iTimer;  bool clockFlag = true;
```

б) додати у MyForm таймер, запуск і зупинку яких забезпечити у обробниках кнопок Start і Stop форми:

```
        private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{timer1->Interval = 200;
         timer1->Start();
}

        private: System::Void button5_Click(System::Object^ sender, System::EventArgs^ e)
        {timer1->Stop();
        }
```

в) реалізувати обробник таймера:

```
private: System::Void timer1_Tick(System::Object^ sender, System::EventArgs^ e)
{
 if (iTimer >= 0 && clockFlag)iTimer++;
 if (iTimer >= 0 && !clockFlag)iTimer--;
 if (iTimer == 0) { clockFlag = true; System::Threading::Thread::Sleep(500); }
 if (iTimer == 90) { clockFlag = false; System::Threading::Thread::Sleep(500); }
 Graphics^ g = MyForm::CreateGraphics();
 Rectangle rect(0, 0, MyForm::Size.Width, MyForm::Size.Height);
 g->Clear(Color::White);
 Pen^ bluePen = gcnew Pen(Color::Blue, 5.0f);
 g->DrawRectangle(bluePen, rect);
Point centerPt;
centerPt.X = (rect.Right - rect.Left) / 2;
centerPt.Y = (rect.Bottom - rect.Top) / 2;
// центр системи
g->DrawLine(bluePen, centerPt.X, rect.Top + 7, centerPt.X, rect.Bottom - 7);
g->DrawLine(bluePen, rect.Left + 7, centerPt.Y, rect.Right - 7, centerPt.Y);
// місця обслуговування
Pen^ redPen = gcnew Pen(Color::Red, 3.0f);
g->DrawRectangle(redPen, centerPt.X - 50, centerPt.Y - 250, 100, 100);
g->DrawRectangle(redPen, centerPt.X - 250, centerPt.Y - 50, 100, 100);
// основа робота у сигляді кола
Pen^ blackPen = gcnew Pen(Color::Black, 2.0f);
int x = 100; int y = 20;
int width = 200; int height = 100;
g->DrawEllipse(redPen, centerPt.X - 50, centerPt.Y - 50, 100, 100);
// перенесення системи координат у центр системи
g->TranslateTransform(centerPt.X, centerPt.Y);
// обертання системи координат
g->RotateTransform((float)iTimer + 180.0);
// плече маніпулятора
g->DrawRectangle(redPen, 0, 0, 105, 30);
// лікоть маніпулятора
g->DrawRectangle(redPen, 70, 35, 120, 20);
// пальці схвату маніпулятора
g->DrawRectangle(redPen, 190, 32, 20, 4);
g->DrawRectangle(redPen, 190, 52, 20, 4);
}
```

Приклад 5.22 – Обробник з побудовою моделі роботизованої системи

Система розпочинає роботу і зупиняється таймером, роботі якого відповідає лічильник iTimer. iTimer відповідає куту повороту маніпулятора і зростає під час руху за годинниковою стрілкою і зменшується під час руху проти годинникової стрілки. У вертикальному і горизонтальному положеннях маніпулятора (коли iTimer==90 та iTimer==0) відбувається переключення напрямку руху (за/проти годинникової стрілки) і встановлюється пауза руху на 500 мілісекунд.

5.11 Контрольні завдання

1. Пояснити порядок створення та використання курсорів, піктограм та пензлів фону у .NET-програмах.
2. Пояснити порядок виведення растрових зображень у головне вікно програми.
3. Пояснити основні функції визначення режимів виведення тексту на екран.
4. Пояснити відмінності у застосування системних та логічних шрифтів.
5. Пояснити доцільність використання віртуальних вікон.
6. Пояснити порядок створення та застосування пір’я та пензлів.
7. Розробити .NET -програму із визначеними курсором, піктограмою та пензлем.
8. Розробити .NET -програму із визначенням властивостей системних шрифтів та виведенням текстів у головне вікно програми. Використайте шрифти ANSI_FIXED_FONT, ANSI_VAR_FONT.
9. Розробити .NET -програму із завантаженням та виведенням логічних шрифтів.
10. Розробити .NET -програму із реалізацію конструкцій віртуального вікна. Здійсніть виведення зображень у головне вікно програми.
11. Розробити .NET -програму із використанням виведення графічних функцій через віртуальне вікно. Реалізувати зміну пір’я та пензлів за допомогою меню головного вікна програми.
12. Реалізувати програму із стрілковим годинником.

6 СТВОРЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ВИКОРИСТАННЯМ MYSQL WORKBENCH C++/ CLI .NET

6.1 Загальні відомості про MySQL

MySQL – одна з найвідоміших технологій у сучасній екосистемі великих даних. Часто звана найпопулярнішою базою даних і в даний час користується широким і ефективним використанням незалежно від галузі, ясно, що будь-хто, хто пов'язаний з корпоративними даними або загальними IT, повинен прагнути хоча б базового знайомства з MySQL.

За допомогою MySQL навіть ті, хто погано знайомий з реляційними системами, можуть відразу створювати швидкі, потужні та безпечні системи зберігання даних. Програмний синтаксис та інтерфейси MySQL також є ідеальними воротами у широкий світ інших популярних мов запитів та структурованих сховищ даних.

MySQL – це система управління реляційними базами даних (СУРБД), розроблена Oracle і заснована на мові структурованих запитів (SQL).

База даних є структурованим набором даних. Це може бути будь-що, від простого списку покупок до картинної галереї або місця для зберігання величезних обсягів інформації в корпоративній мережі. Зокрема, реляційна база даних — це цифрове сховище, яке збирає дані та організує їх відповідно до реляційної моделі. У цій моделі таблиці складаються з рядків та стовпців, а відносини між елементами даних мають сувору логічну структуру. СУРБД – це просто набір програмних інструментів, що використовуються для реалізації, управління та запитів до такої бази даних.

MySQL є невід'ємною частиною багатьох найбільш популярних програмних стеків для створення та підтримки всього, від клієнтських веб-додатків до потужних керованих даними послуг B2B. Його відкритий вихідний код, стабільність та багатий набір функцій у поєднанні з постійним розвитком та підтримкою з боку Oracle означають, що критично важливі для Інтернету організації, такі як Facebook, Flickr, Twitter, Wikipedia та YouTube використовують серверні частини MySQL.

Оскільки MySQL найбільш широко використовується в багатьох галузях, бізнес-користувачі, від веб-майстрів-початківців до досвідчених менеджерів, повинні прагнути зрозуміти його основні характеристики. Ухвалення рішення про використання цієї технології та ефективне інформування про неї почина-

ється з огляду базової доступності, структури, філософії та зручності використання MySQL.

Хоча MySQL часто асоціюється з Інтернет-додатками або веб-службами, вона була розроблена так, щоб бути повністю сумісною з іншими технологіями та архітектурами. СУРБД працює на всіх основних обчислювальних платформах, включаючи операційні системи на основі Unix, такі як безліч дистрибутивів Linux або Mac OS та Windows.

Клієнт-серверна архітектура MySQL означає, що вона може підтримувати різні серверні частини та різні програмні інтерфейси. Дані можуть бути безпосередньо перенесені з MySQL у її відгалуження (наприклад, MariaDB), а також у більшість інших СУБД завдяки архітектурній та мовній схожості.

Затверджені Oracle та сторонні інструменти міграції також дозволяють MySQL переміщати дані в широкий набір загальних систем зберігання та з них, незалежно від того, чи вони призначені для локального або хмарного використання. MySQL можна розгортати у віртуалізованих середовищах, розподілених або централізованих, і навіть існує у вигляді автономних бібліотек, що переносяться для цілей навчання, тестування або невеликих додатків.

Широка сумісність MySQL з усіма цими іншими системами та програмним забезпеченням робить її особливо практичним вибором РСУБД у більшості ситуацій.

Основний чинник, який відрізняє реляційні бази даних з інших цифрових сховищ, у тому, як дані організовані високому рівні. Бази даних, такі як MySQL, містять записи в декількох окремих та строго закодованих таблицях, на відміну від єдиного всеосяжного репозиторію або колекцій напівструктурованих чи неструктурованих документів.

Це дозволяє РСУБД краще оптимізувати такі дії, як пошук даних, оновлення інформації чи складніші дії, такі як агрегування. Логічна модель визначається для вмісту бази даних, описуючи, наприклад, допустимі значення в окремих стовпцях, характеристики таблиць і уявлень або те, як пов'язані індекси з двох таблиць.

Реляційні моделі залишаються популярними з кількох причин. Вони надають користувачам інтуїтивно зрозумілі, декларативні мови програмування — по суті, повідомляючи базі даних, який результат потрібен, мовою, близькою або принаймні зрозумілою письмовій англійській мові, замість того, щоб ретельно кодувати кожен крок процедури, що веде до цього результату. . Це переносить більшу частину роботи на механізми РСУБД і SQL, забезпечуючи

найкраще дотримання логічних правил і заощаджуючи цінні ресурси та робочу силу.

Будь-яка фізична або юридична особа може вільно використовувати, модифікувати, публікувати та розширювати кодову базу Oracle з відкритим кодом MySQL. Програмне забезпечення випущено під Стандартною громадською ліцензією GNU (GPL).

Для коду MySQL, який необхідно інтегрувати або включити до комерційної програми (або якщо програмне забезпечення з відкритим вихідним кодом не є пріоритетом), підприємства можуть придбати версію з комерційною ліцензією у Oracle.

Знову ж таки, ці опції надають організаціям додаткову гнучкість після ухвалення рішення про роботу з MySQL. Публічний і заснований на співтоваристві характер випусків з відкритим вихідним кодом збагачує документацію MySQL та культуру онлайн-підтримки, а також гарантує, що стійкі або нещодавно розроблені можливості ніколи не відхиляються надто далеко від поточних потреб користувачів.

Хоча реляційна природа MySQL і жорсткі структури зберігання, що випливають з неї, можуть здатися обмежуючими, таблична парадигма, мабуть, найбільш інтуїтивно зрозуміла і, в кінцевому рахунку, забезпечує більшу зручність використання.

Насправді MySQL робить багато поступок у підтримці максимально широкого розмаїття структур даних, від стандартних, але багатих логічних, числових, літерно-цифрових типів, типів дати та часу до більш сучасних JSON або геопросторових даних. Крім простих типів даних і розширеного набору вбудованих функцій, екосистема MySQL також включає безліч інструментів, що спрощують все, від управління сервером до створення звітів і аналізу даних.

Незалежно від всеосяжної архітектури СУБД, користувачі завжди можуть знайти функцію MySQL, що дозволяє їм моделювати і кодувати дані за своїм бажанням. MySQL залишається однією з найпростіших технологій баз даних для вивчення та використання.

6.2 Встановлення MySQL Workbench та створення бази даних

По-перше, потрібно зайти на сайт MySQL за посиланням https://dev.mysql.com/downloads та завантажити файл MySQL Installer for Windows (рис. 6.1).

Далі, після встановлення MySQL Installer, користуючись меню, вам потрібно встановити наступні елементи (рис. 6.2), серед яких:

– MySQL Workbench;

– MySQL for Visual Studio;

– Connector/NET;

– MySQL Server.

Рисунок 6.1 – Завантаження файлу MySQL Installer for Windows

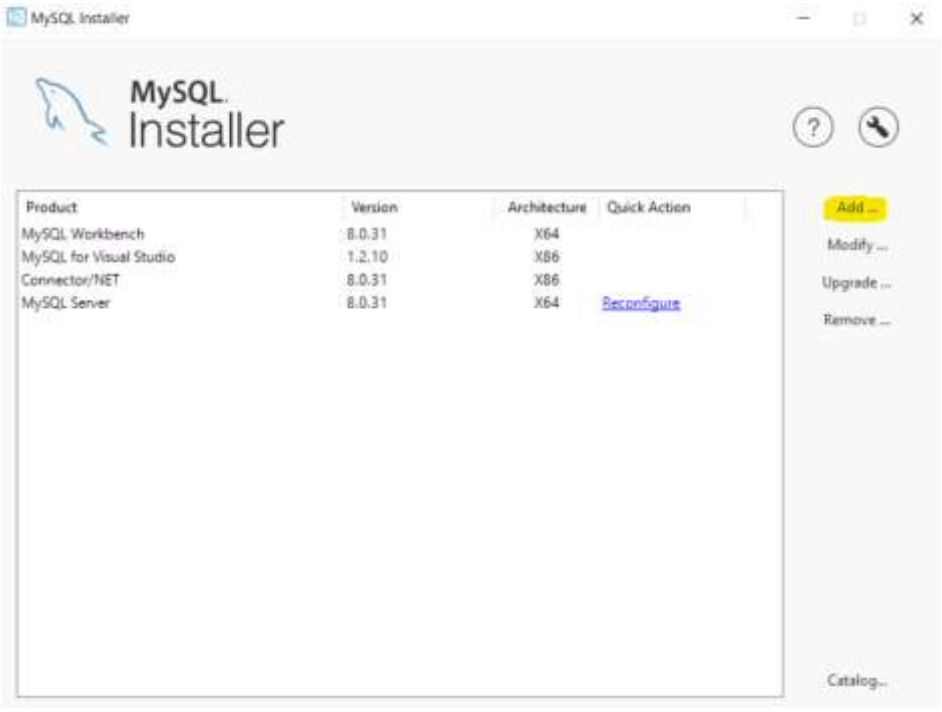


Рисунок 6.2 – Встановлення продуктів за допомогою MySQL Installer

Також є можливість у будь-який час змінити налаштування, натиснувши кнопку Add. При встановленні MySQL Server (рис. 6.3) потрібно встановити параметри паролю для адміну (root) у відповідному вікні (рис. 6.4).

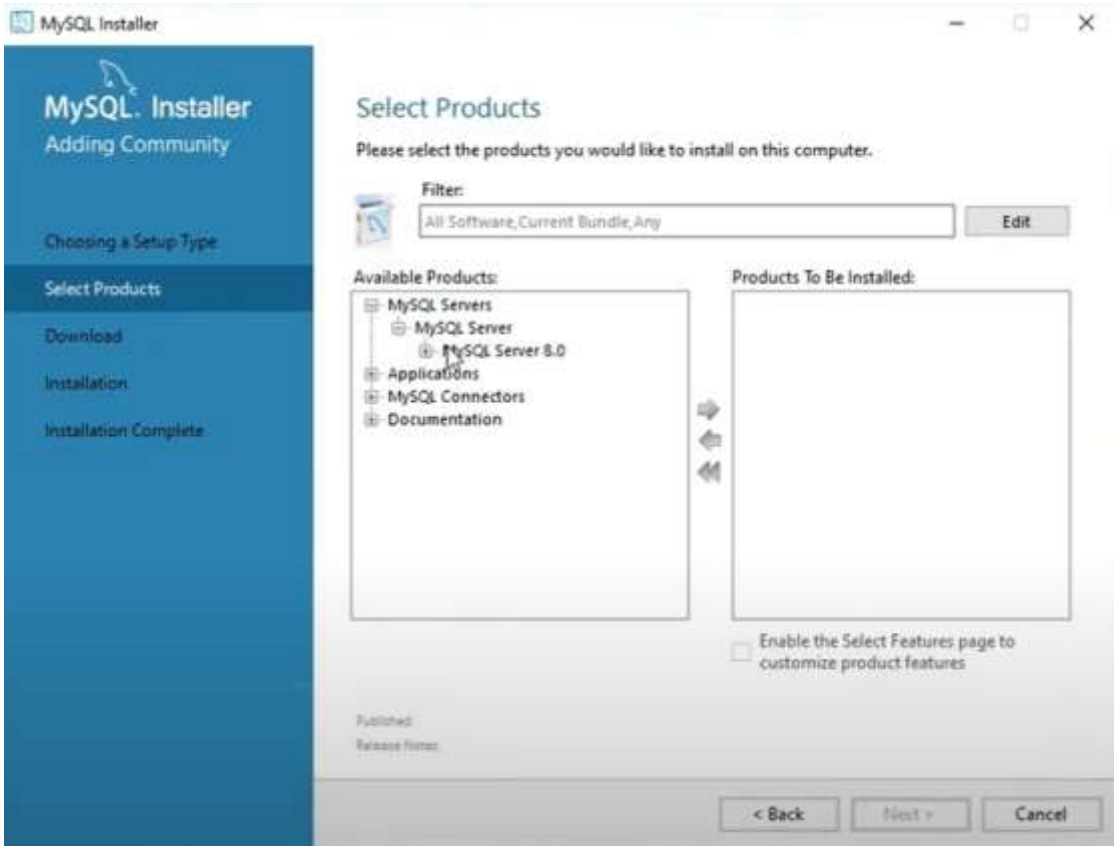


Рисунок 6.3 – Встановлення MySQL Server

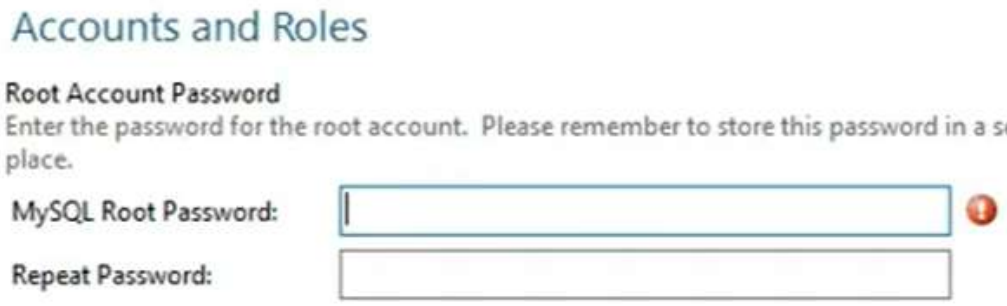


Рисунок 6.4 – Налаштування паролю для root у MySQL Server

Після встановлення MySQL Workbench відкриється головне вікно, в якому потрібно створити нове з’єднання, натиснувши знак плюс (рис. 6.5).

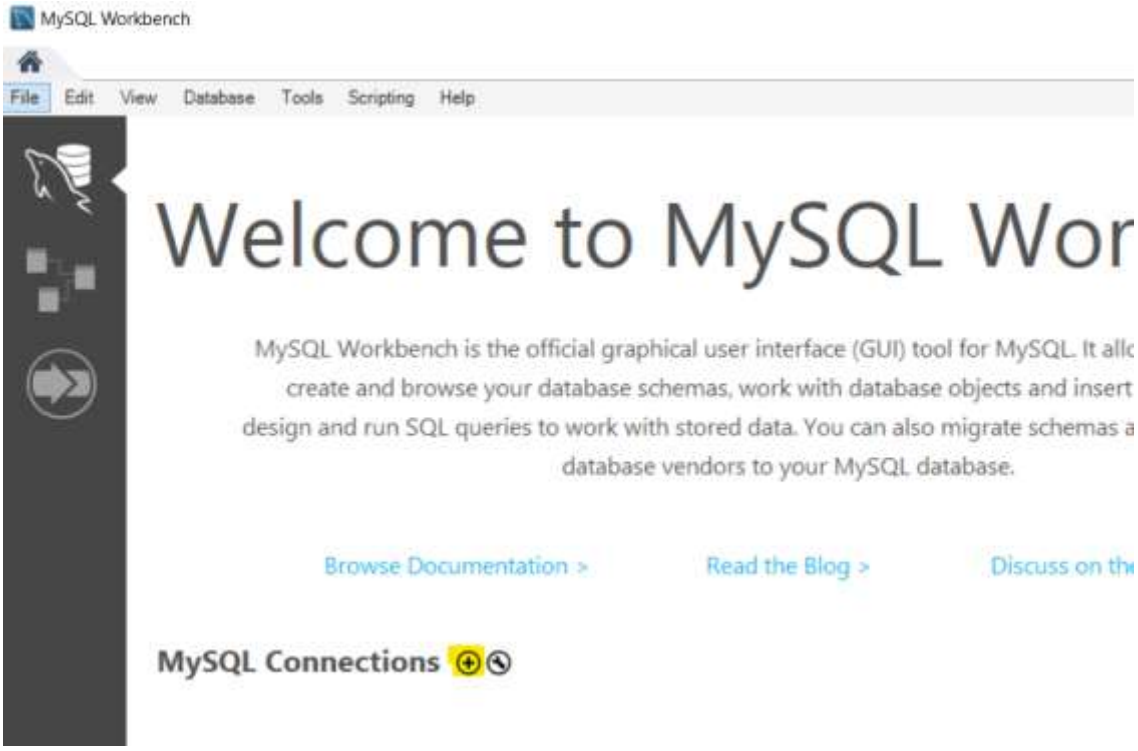


Рисунок 6.5 – Головне вікно MySQL Workbench

Після натиснення відкриється вікно, в кому потрібно заповнити параметр його назви, як показано на рис. 6.6.

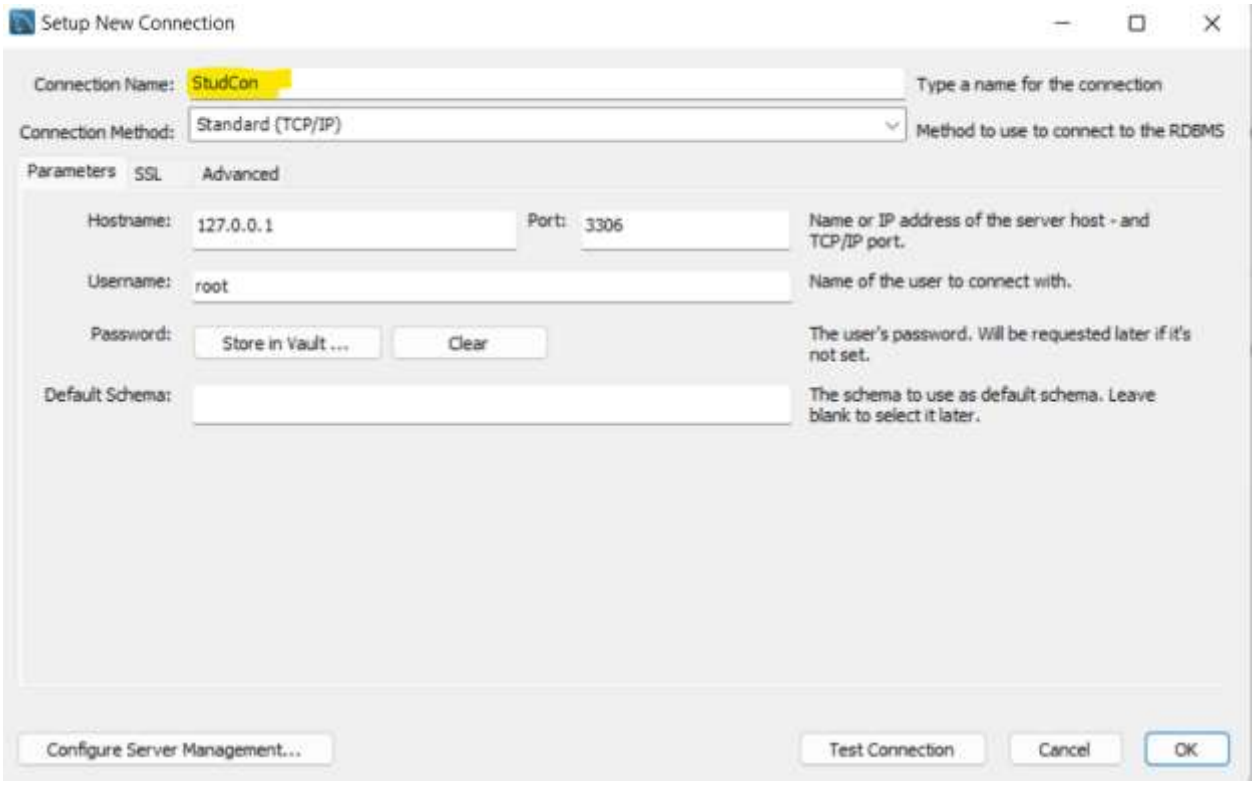


Рисунок 6.6 – Створення нового з’єднання

Після цього воно з’явиться внизу, як показано на рис. 6.7.

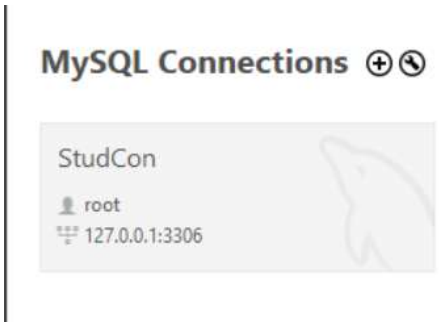


Рисунок 6.7 – Створене з’єднання

Для під’єднання потрібно натиснути на нього та ввести пароль, що було задано на кроці створення у MySQL Server.

Після цього відкриється вікно зі схемою або базою даних (БД) за замовчуванням. Її можна видалити. Для створення нової потрібно натиснути Create a new schema in the connected server (рис. 6.8).

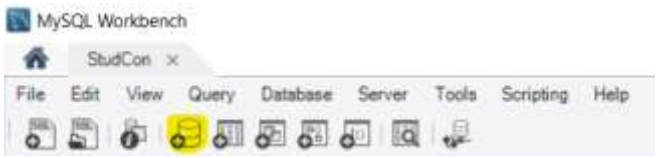


Рисунок 6.8 – Створення нової схеми

Далі потрібно вказати ім’я для БД, а також виконати SQL-запит (рис. 6.9).

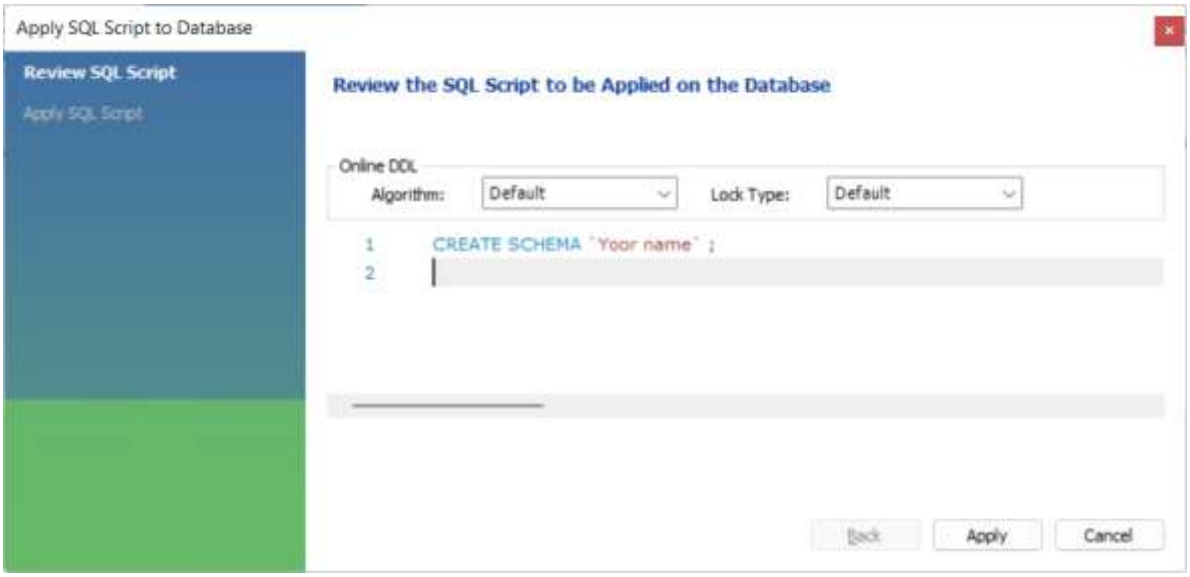


Рисунок 6.9 – SQL-запит для створення схеми

Далі у навігаторі (рис. 6.10) натиснути правою кнопкою миші на назві схеми правою кнопкою миші, обрати опцію Set as Default Schema, потім створити нову таблицю, натиснувши на Tables правою кнопкою та обравши Create Table.

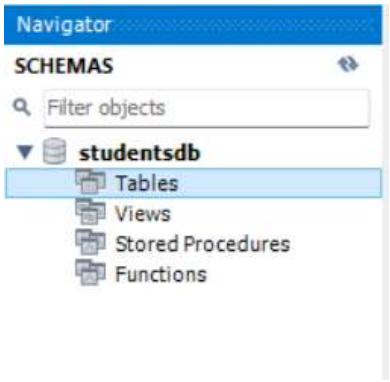


Рисунок 6.10 – Навігатор схеми

Для створення таблиці потрібно вказати назви полів (Column Name), типи даних (Datatype) та первинний ключ (PK) (рис. 6.11).

Потім натиснути кнопку Apply та виконати SQL-запит на створення таблиці (рис. 6.12).

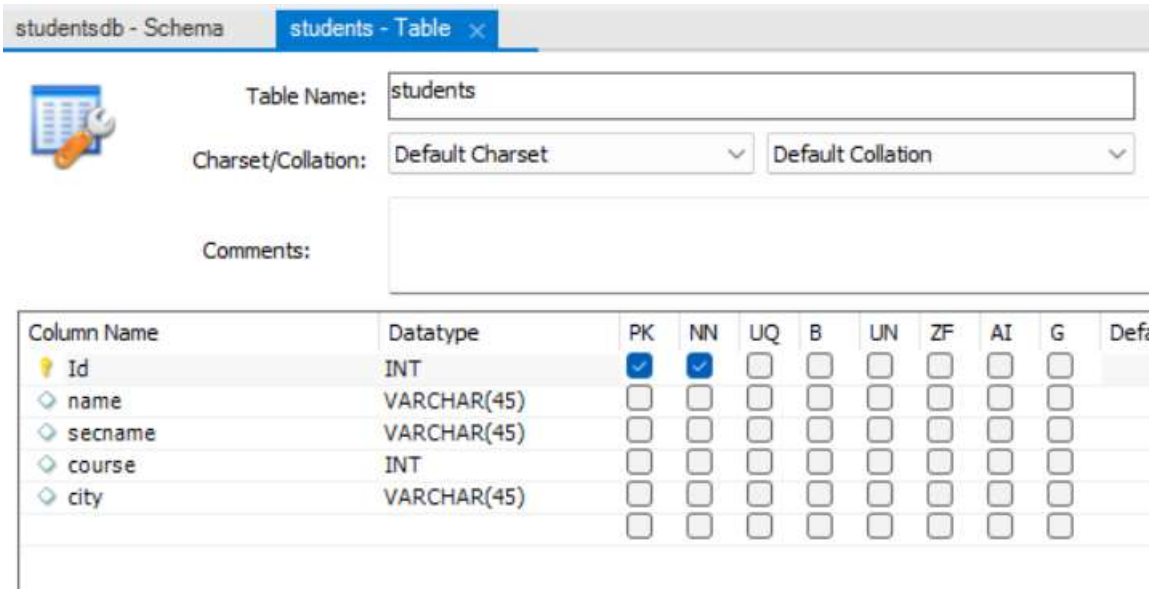


Рисунок 6.11 – Налаштування та створення таблиці

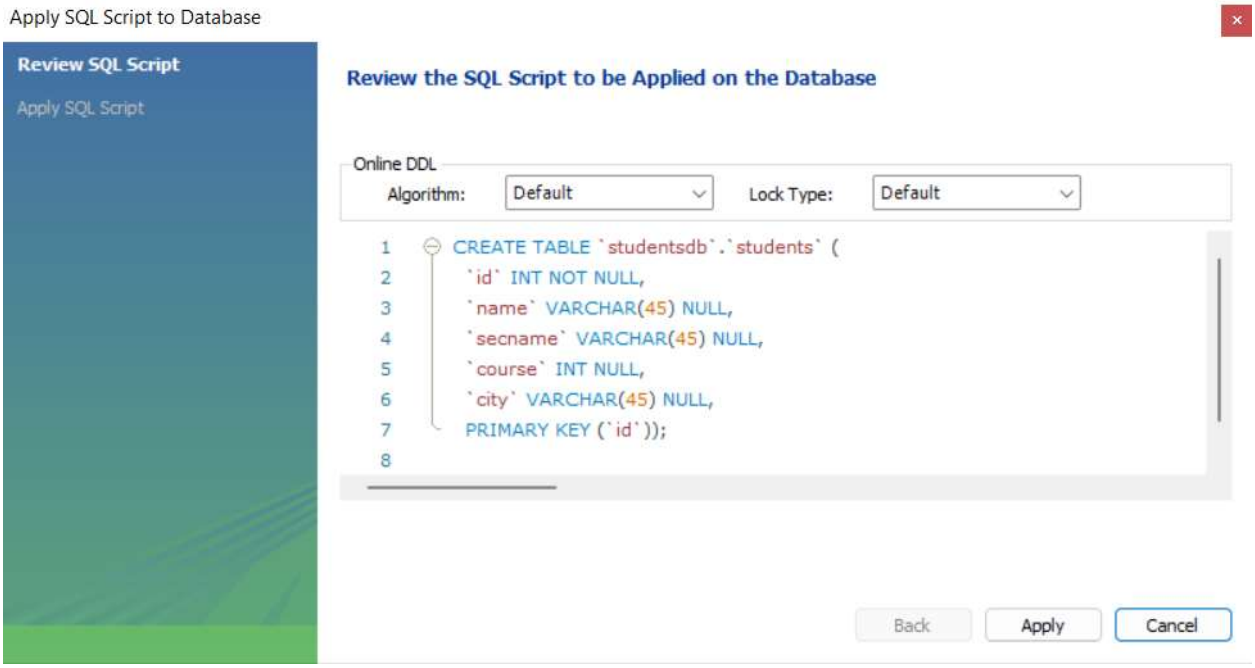


Рисунок 6.12 – Виконання запиту на створення таблиці

Після виконання SQL-запиту таблиця з’явиться у вкладці Tables (рис. 6.13).

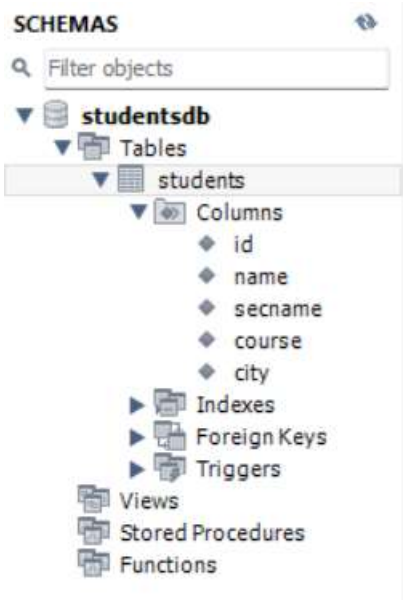


Рисунок 6.13 – Створена таблиця

Далі натискаємо правою кнопкою миші по створеній таблиці та обираємо Select Rows – Limit 1000 та заповнюємо таблицю даними (рис. 6.14), після чого натискаємо кнопку Apply та виконуємо SQL-запит (рис. 6.15).

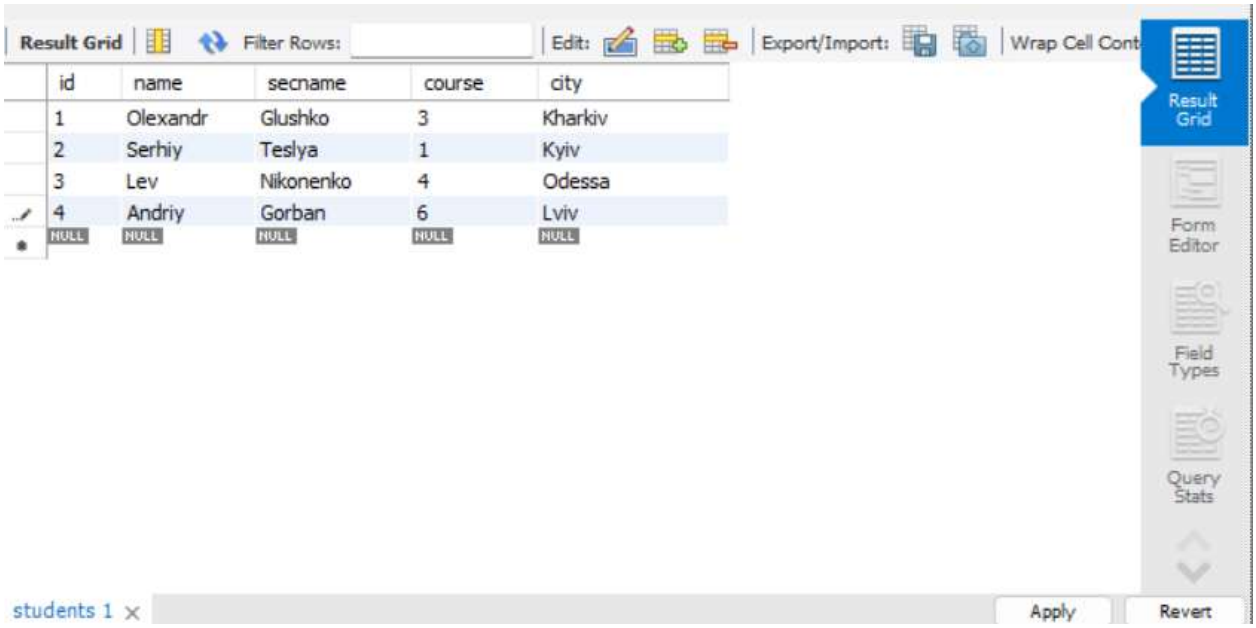


Рисунок 6.14 – Заповнення таблиці

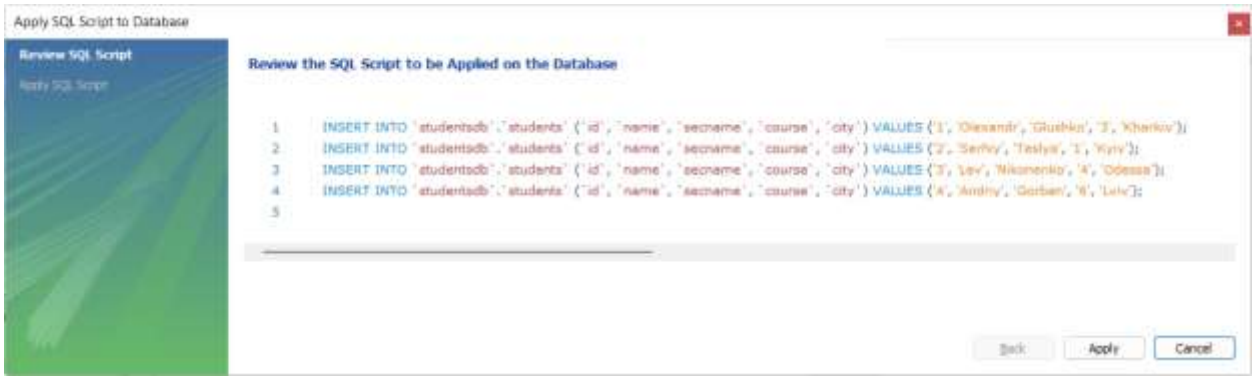


Рисунок 6.15 – SQL-запит на заповнення таблиці

6.3 Створення пустого проєкту та встановлення відповідних бібліотек

Перш за все потрібно створити проєкт CLR Empty Project (.NET Framework) під будь-якою назвою, наприклад DatabaseProject.

Далі створюємо форму MyForm.h та встановлюємо всі необхідні параметри для компіляції пустого .NET-проєкту.

На форму додаємо елементи для під’єднання до бази даних та роботи з нею (рис. 5.15):

– 6 кнопок (Button) – для додавання, оновлення, видалення, перезавантаження, видалення даних та виходу;

– 5 надписів (Label) та полів (TextBox) для роботи з базою даних;

– елемент DataGridView – для того, щоб виводити інформацію з бази даних, а також для слідкуванням за нею.

Далі потрібно зайти у посилання (References) вашого проєкту у Оглядачі рішень (Solution Explorer) (рис. 6.16).

Натиснути по ньому правою кнопкою миші та обрати опцію Add Reference. Потім обрати Assemblies (Збірки) – Extensions (Розшинення) – MySql.Data (рис. 6.17) та натиснути клавішу підтвердження.

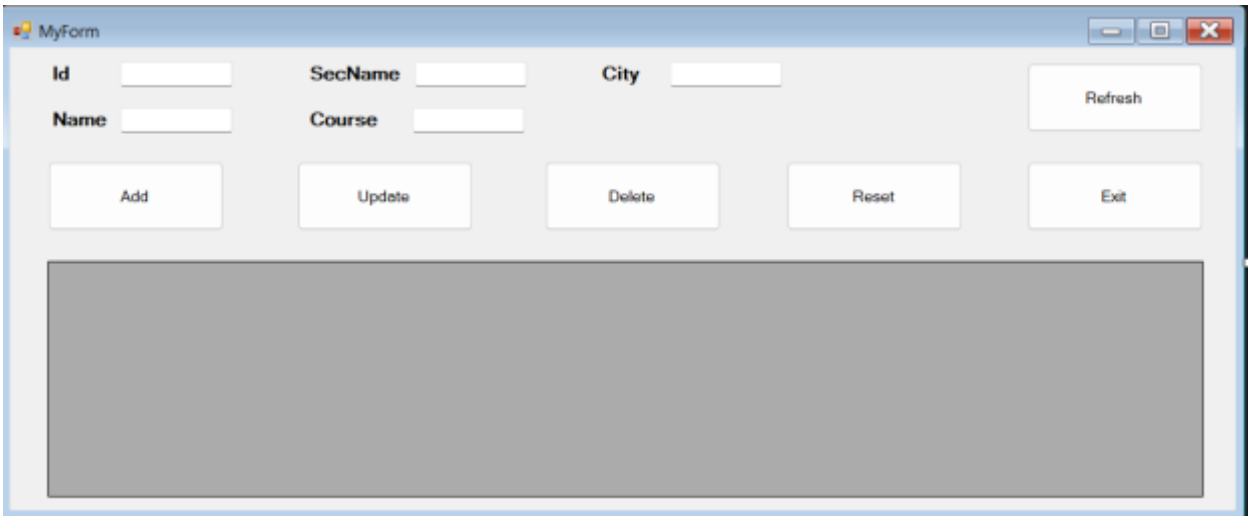


Рисунок 6.16 – Додавання елементів для роботи з базою даних

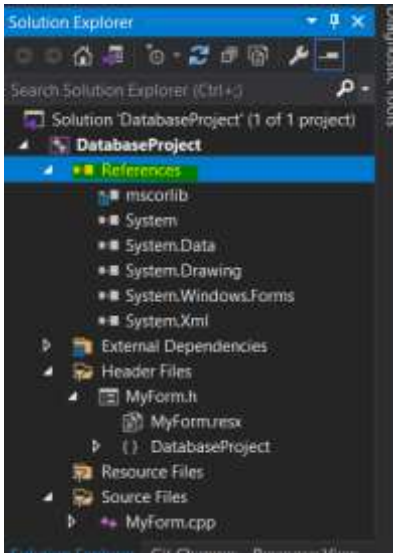


Рисунок 6.17 – References вашого проєкту у Solution Explorer

У разі, якщо цього розширення немає, його потрібно обрати через кнопку Browse за шляхом C:\Program Files (x86)\MySQL\Connector NET 8.0\Assemblies та обрати відповідну опцію до вашого .NET Framework та обрати файл MySql.Data.dll. Якщо виникає помилка, змінити .NET Framework Target version на v4.5.2. Потім підключаємо бібліотеку:

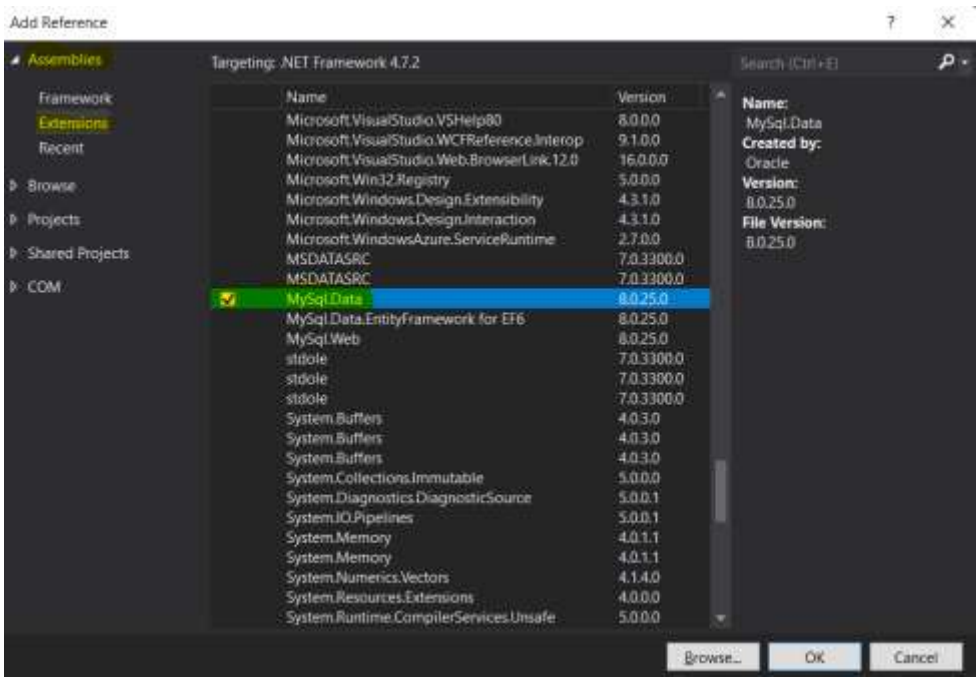


Рисунок 6.18 – Розширення для використання MySQL

using namespace MySql::Data::MySqlClient;

Потім у класі форми створюємо змінні (рис. 6.18):

```
MySqlConnection^ sqlConn = gcnew MySqlConnection();
        MySqlCommand^ sqlCmd = gcnew MySqlCommand();
        DataTable^ sqlDt = gcnew DataTable();
        MySqlDataAdapter^ sqlDtA = gcnew MySqlDataAdapter();
        MySqlDataReader^ sqlRd;
```

```
using namespace MySql::Data::MySqlClient;

/// <summary>
/// Summary for MyForm
/// </summary>
public ref class MyForm : public System::Windows::Forms::Form
{

    MySqlConnection^ sqlConn = gcnew MySqlConnection();
    MySqlCommand^ sqlCmd = gcnew MySqlCommand();
    DataTable^ sqlDt = gcnew DataTable();
    MySqlDataAdapter^ sqlDtA = gcnew MySqlDataAdapter();
    MySqlDataReader^ sqlRd;
```

Рисунок 6.19 – Створення змінних

Перша змінна відповідає за з'єднання з БД, друга – за виконання запитів до БД, третя – представляє одну таблицю даних у пам'яті, четверта – представляє набір команд даних і підключення до бази даних, які використовуються для заповнення набору даних і оновлення бази даних MySQL, п'ята – надає засоби читання прямого потоку рядків із бази даних MySQL.

Додаємо обробники до кнопок керування. Першою є кнопка виходу з програми. Для цього натискаємо двічі на кнопку Exit у дизайнері форм і записуємо туди відповідний код (рис. 6.20):

```
System::Windows::Forms::DialogResult iExit;
iExit = MessageBox::Show("Confirm your exit!", "Data Entry Form",
MessageBoxButtons::YesNo,MessageBoxIcon::Question);
if (iExit == System::Windows::Forms::DialogResult::Yes)
{ Application::Exit(); }.
```

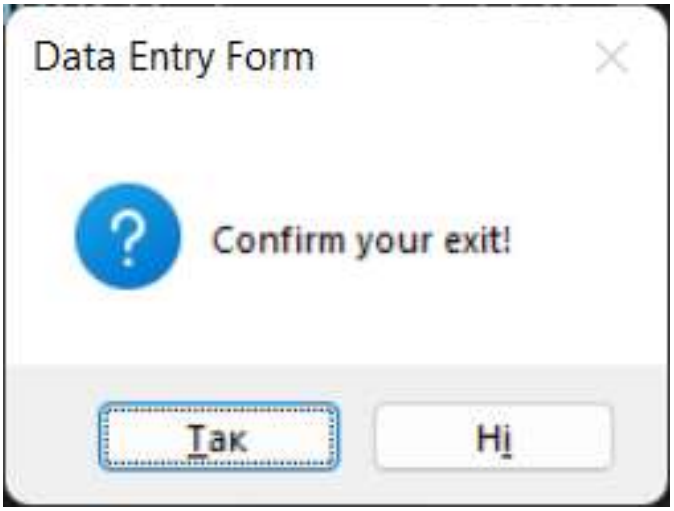


Рисунок 6.20 – Діалогове вікно виходу

Створимо функцію, що буде при запуску програмного забезпечення виводити інформацію з бази даних, наприклад MembershipDB() (рис. 6.21):

```
private: System::Void MembershipDB() {
sqlConn->ConnectionString = "datasource=localhost;
port=3306;username=root;password=12345;database=studentsdb";
sqlConn->Open();
sqlCmd->Connection = sqlConn;
sqlCmd->CommandText = "SELECT * FROM students";
sqlRd = sqlCmd->ExecuteReader();
sqlDt->Load(sqlRd);
sqlRd->Close();
sqlConn->Close();
dataGridView1->DataSource = sqlDt; }
```

Основними параметрами цієї функції є:

– рядок з параметрами з'єднання (джерело даних (datasource), порт (port), ім'я користувача (username), пароль (password) та назва бази даних (database));

– відкриття з'єднання з БД;

– виконання команд (запитів до БД);

– завантаження інформації;

– закриття з'єднання з БД;

– виведення інформації у dataGridView1.

Рисунок 6.21 – Завантаження інформації з БД при запуску програмного забезпечення

При натисненні на кнопку Reset відбувається очищення всієї інформації з полів TextBox, початковий стан представлено на рис. 5.62, кінцевий – на рис. 6.23:

```
try
        {       txtId->Text = "";
                txtName->Text = "";
                txtSecName->Text = "";
                txtCourse->Text = "";
                txtCity->Text = "";         }
        catch (Exception^ ex)
        {               MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo,
MessageBoxIcon::Information);     }.
```

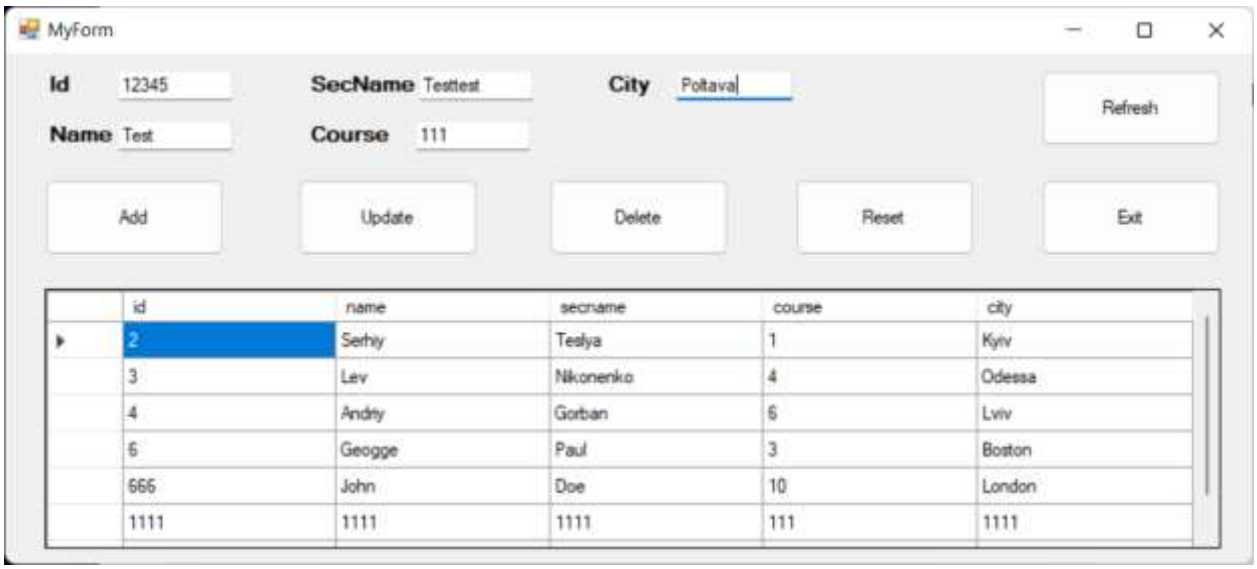


Рисунок 6.22– Стан з введеними даними у поля

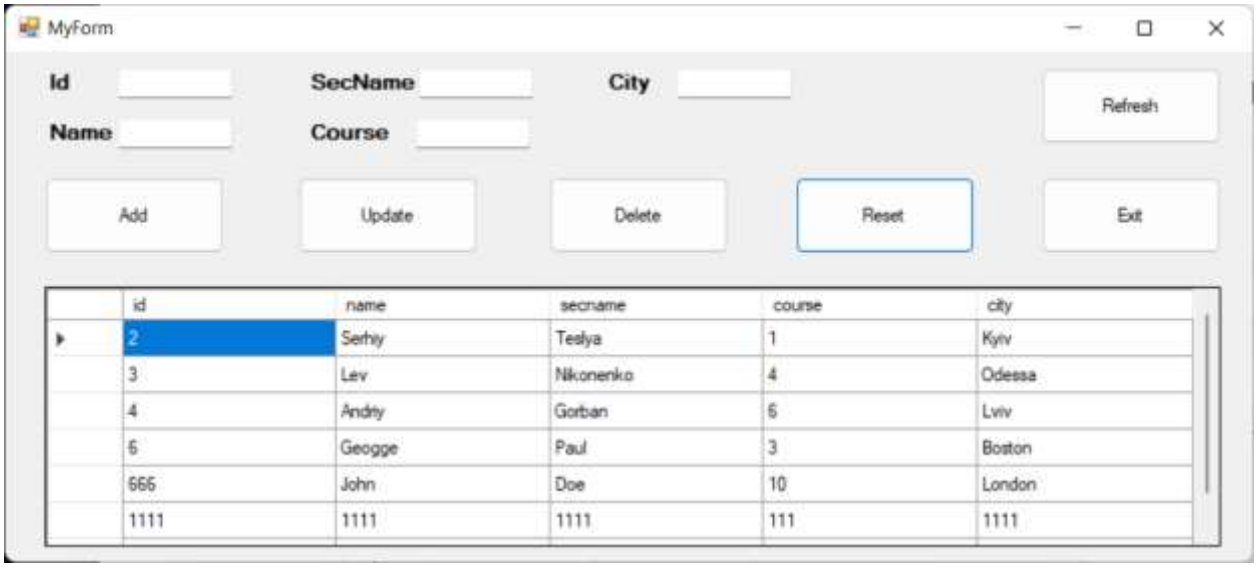


Рисунок 6.22– Стан після натиснення кнопки Reset

Для оновлення бази даних у DataGridView додамо нову функцію RefreshDB() (рис. 5.23), а також подію на кнопку Refresh (рис. 5.24) з наступним текстом, де буде виконано оновлення всієї бази даних відразу після натиснення кнопки:

```
try
        { sqlConn->ConnectionString = "datasource=localhost;
port=3306;username=root;password=12345;database=studentsdb";
          sqlCmd->Connection = sqlConn;
          MySqlDataAdapter^ sqlDtA = gcnew MySqlDataAdapter("SELECT * FROM students",
sqlConn);
          DataTable^ sqlDt = gcnew DataTable();
          sqlDtA->Fill(sqlDt);
```

```
                dataGridView1->DataSource = sqlDt;          }
            catch (Exception^ ex)
            {       MessageBox::Show(ex->Message,   "Error   Message",   MessageBoxButtons::YesNo,
MessageBoxIcon::Information);       }

    private: System::Void btnRefresh_Click(System::Object^ sender, System::EventArgs^ e) {    RefreshDB(); }
```

```
private: System::Void RefreshDB() {
    try
    {

        sqlConn->ConnectionString = "datasource=localhost; port=3306;username=root;password=12345;database=studentsdb";
        sqlCmd->Connection = sqlConn;

        MySqlDataAdapter^ sqlDtA = gcnew MySqlDataAdapter("SELECT * FROM students", sqlConn);
        DataTable^ sqlDt = gcnew DataTable();
        sqlDtA->Fill(sqlDt);
        dataGridView1->DataSource = sqlDt;


    }
    catch (Exception^ ex)
    {
        MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo, MessageBoxIcon::Information);
    }
}
```

Рисунок 6.23 – Функція оновлення інформації у DataGridView

```
private: System::Void btnRefresh_Click(System::Object^ sender, System::EventArgs^ e) {
    RefreshDB();
}
```

Рисунок 6.24 – Подія натискання на кнопку Refresh

Для додавання інформації до бази даних створимо подію на натиснення кнопки Add (рис. 6.25) та після виконання цієї функції отримаємо наступні стани роботи програми (рис. 6.26-6.27):

```
private: System::Void btnAdd_Click(System::Object^ sender, System::EventArgs^ e) {
        sqlConn->ConnectionString                    =                    "datasource=localhost;
port=3306;username=root;password=12345;database=studentsdb";
        sqlConn->Open();
        sqlCmd->Connection = sqlConn;
        try
        { sqlCmd->CommandText = "INSERT INTO students(id,name,secname,course,city)"+
                "VALUES('"+ txtId->Text + "', '" + txtName->Text+"', '" +txtSecName->Text+ "', '" + txtCourse->Text +
"', '" +txtCity->Text + "')";
                sqlCmd->ExecuteNonQuery();
                sqlConn->Close();
                MembershipDB();
                RefreshDB();       }
        catch (Exception^ ex)
        {       MessageBox::Show(ex->Message,   "Error   Message",   MessageBoxButtons::YesNo,
MessageBoxIcon::Information);       }
                }
```

```
private: System::Void btnAdd_Click(System::Object^ sender, System::EventArgs^ e) {

    sqlConn->ConnectionString = "datasource=localhost; port=3306;username=root;password=12345;database=studentsdb";
    sqlConn->Open();
    sqlCmd->Connection = sqlConn;
    try
    {
        sqlCmd->CommandText = "INSERT INTO students(id,name,secname,course,city)"+
        "VALUES('"+ txtId->Text + "', '" + txtName->Text+"', '" +txtSecName->Text+ "', '" + txtCourse->Text + "', '" +txtCity->Text + "')";
        sqlCmd->ExecuteNonQuery();
        sqlConn->Close();
        MembershipDB();
        RefreshDB();
    }
    catch (Exception^ ex)
    {       MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo, MessageBoxIcon::Information);  }
}
```

Рисунок 6.25 – Подія натискання на кнопку Add

Рисунок 6.26 – Стан перед натисканням кнопки Add

Рисунок 6.27 – Стан після натискання кнопки Add

Аналогічним чином відбувається і створення події на кнопку Update, попередні і змінені стани, представлено на рис. 6.28-6.29:

```
private: System::Void btnUpdate_Click(System::Object^ sender, System::EventArgs^ e) {
        try
        {       sqlConn->ConnectionString = "datasource=localhost;
port=3306;username=root;password=12345;database=studentsdb";
                sqlConn->Open();
                String^ Id = txtId->Text;
                String^ Name = txtName->Text;
                String^ SecName = txtSecName->Text;
                String^ Course = txtCourse->Text;
                String^ City = txtCity->Text;

                sqlCmd->CommandText = "UPDATE students SET id = '" + Id + "', name ='" + Name + "',
secname = '"
                        + SecName + "', course = '" + Course + "', city = '" + City + "'WHERE id = '"+ Id +"'";
                sqlCmd->ExecuteNonQuery();
                sqlConn->Close();
                MembershipDB();
                RefreshDB();    }
        catch (Exception^ ex)
        {       MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo,
MessageBoxIcon::Information);    }
        }.
```

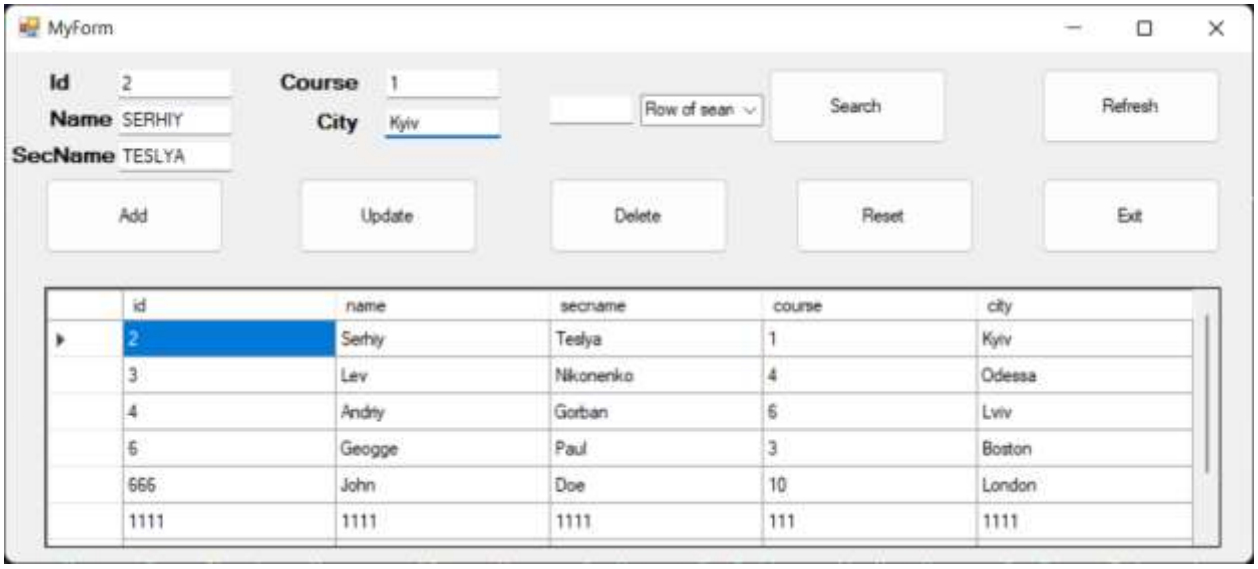


Рисунок 6.28 – Попередній стан перед натисканням кнопки Update

Також для зручності зміни параметрів при функції Update додамо метод для кожного рядка у DataGridView, що допоможе вставляти автоматично у поля інформацію з бази даних при натисканні зліва на рядок (рис. 6.30-6.31). Для цього використаємо подію dataGridView1_CellClick:

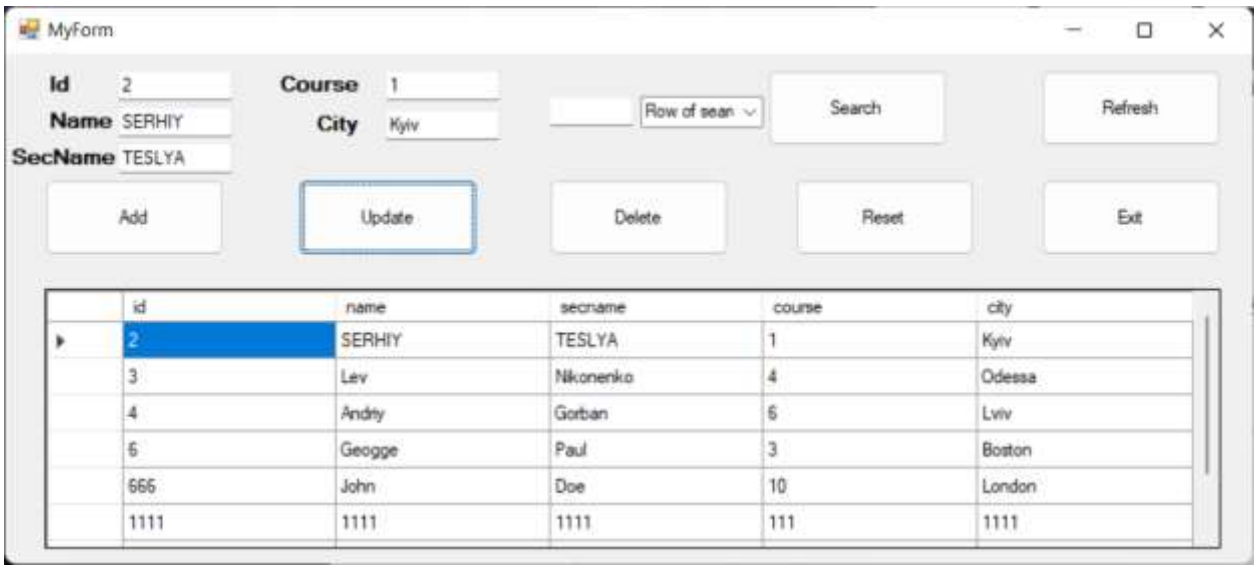


Рисунок 6.29 – Стан після натискання кнопки Update

```
private: System::Void dataGridView1_CellClick(System::Object^ sender,
System::Windows::Forms::DataGridViewCellEventArgs^ e) {
        txtId->Text = dataGridView1->SelectedRows[0]->Cells[0]->Value->ToString();
        txtName->Text = dataGridView1->SelectedRows[0]->Cells[1]->Value->ToString();
        txtSecName->Text = dataGridView1->SelectedRows[0]->Cells[2]->Value->ToString();
        txtCourse->Text = dataGridView1->SelectedRows[0]->Cells[3]-> Value->ToString();
        txtCity->Text = dataGridView1->SelectedRows[0]->Cells[4]->Value->ToString(); }
```

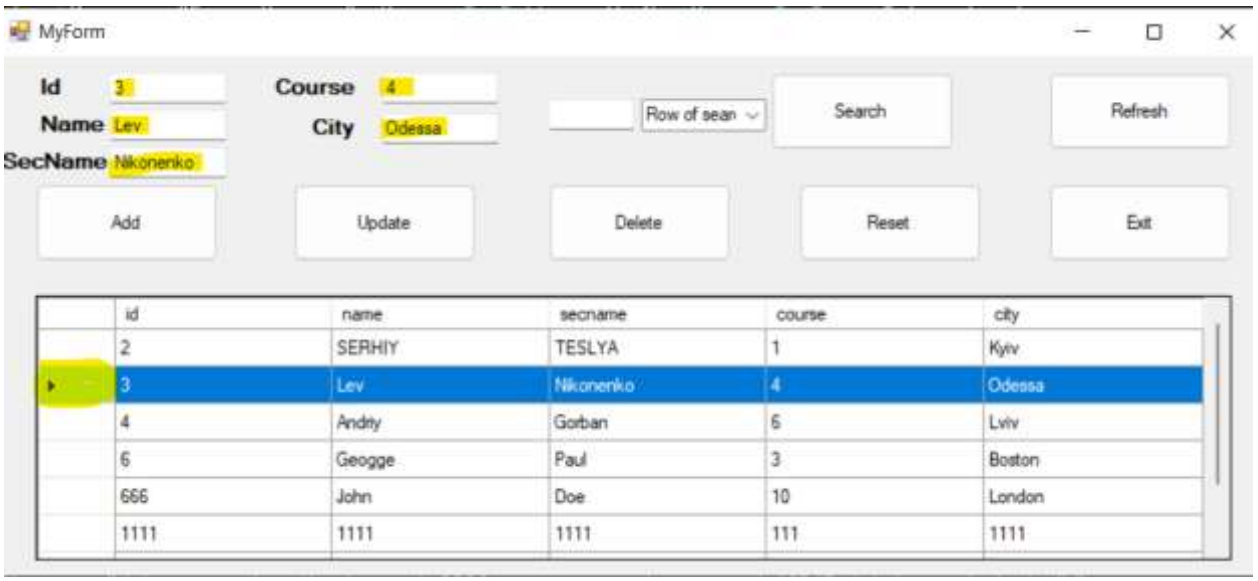


Рисунок 6.30 – Перший стан роботи функції dataGridView1_CellClick

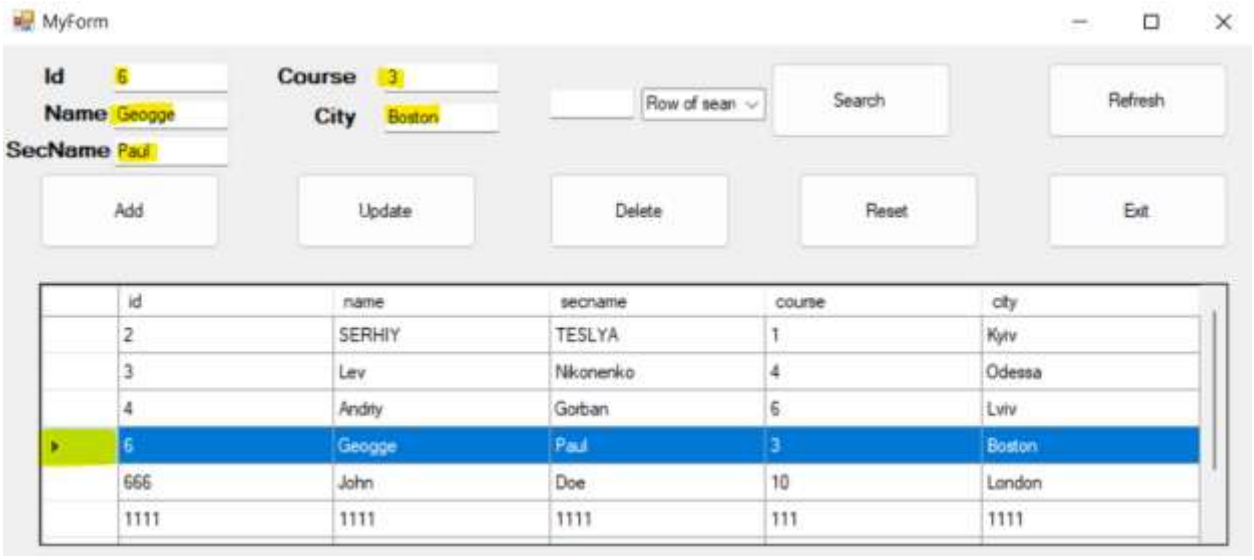


Рисунок 6.31 – Другий стан роботи функції dataGridView1_CellClick

Також потрібно змінити налаштування параметру AutoSizeColumnsMode на Fill (рис. 6.32).

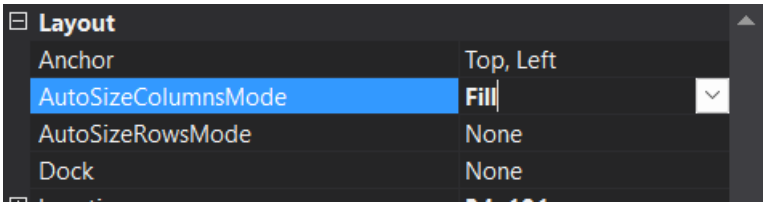


Рисунок 6.32 – Зміна налаштування параметру AutoSizeColumnsMode

Також додамо обробник для конопки Delete, що відповідає за видалення обраного елементу за полем Id:

```
private: System::Void btnDelete_Click(System::Object^ sender, System::EventArgs^ e) {
        try
        { sqlConn->ConnectionString = "datasource=localhost;
port=3306;username=root;password=12345;database=studentsdb";
        sqlCmd->Connection = sqlConn;
        String^ Id = txtId->Text;
        MySqlCommand^ sqlCmd = gcnew MySqlCommand("DELETE FROM students WHERE id = "
+ Id + "", sqlConn);
        sqlConn->Open();
        sqlRd = sqlCmd->ExecuteReader();
        MessageBox::Show("Record Deleted", "Error Message", MessageBoxButtons::YesNo,
MessageBoxIcon::Information);
        sqlConn->Close(); }
    catch (Exception^ ex)
    { MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo,
MessageBoxIcon::Information); }
    RefreshDB(); }
```

Як ви могли помітити, було додано ще одне поле ListBox, ComboBox та кнопка Search (рис. 6.33). Ця функція буде відповідати за пошук інформації у dataGridView.

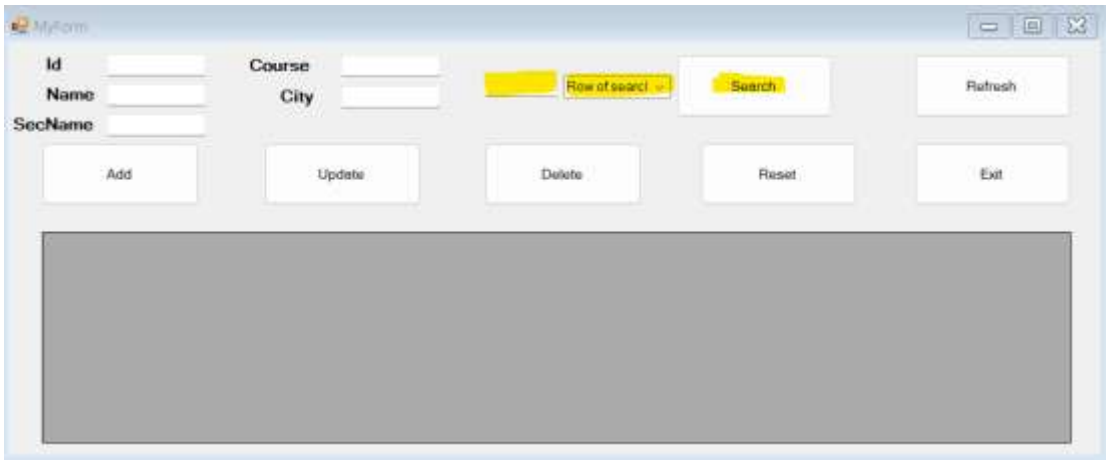


Рисунок 6.33 – Форма з новими елементами

Для елементу ComboBox було додано параметри з назвами стовпців dataGridView (рис. 6.34-6.35).

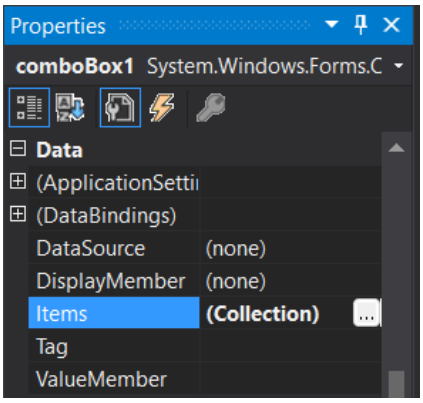


Рисунок 6.34 – Параметр Items ComboBox

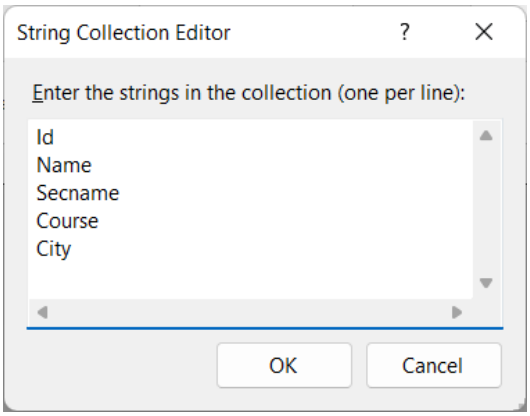


Рисунок 6.34 – Колекція Items ComboBox

Також на кнопку Search було додано обробник пошуку елементів у dataGridView (рис. 6.35-6.36):

```
private: System::Void btnSearch_Click(System::Object^ sender, System::EventArgs^ e) {
        try
        {       DataView^ dv = sqlDt->DefaultView;
                dv->RowFilter = String::Format("CONVERT({0}, System.String) LIKE '%{1}%'",comboBox1->SelectedItem->ToString(), txtSearch->Text);
                dataGridView1->DataSource = dv->ToTable();
        }
        catch (Exception^ ex)
        {       MessageBox::Show(ex->Message, "Error Message", MessageBoxButtons::YesNo, MessageBoxIcon::Information);       }}
```

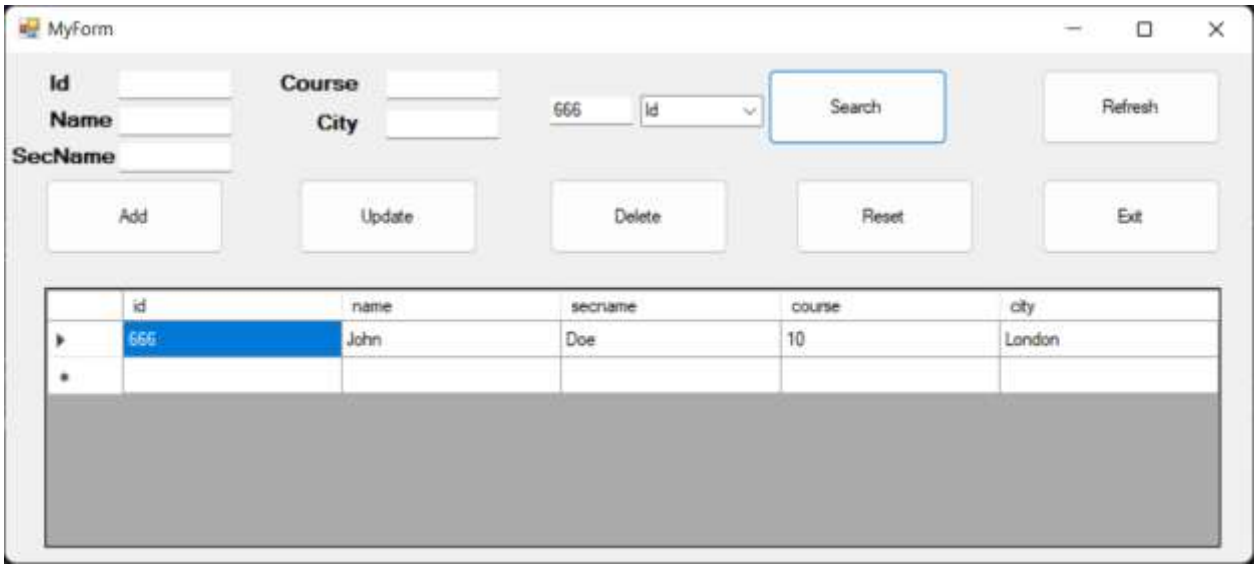


Рисунок 6.35 – Фільтр-пошук за Id, який дорівнює 666

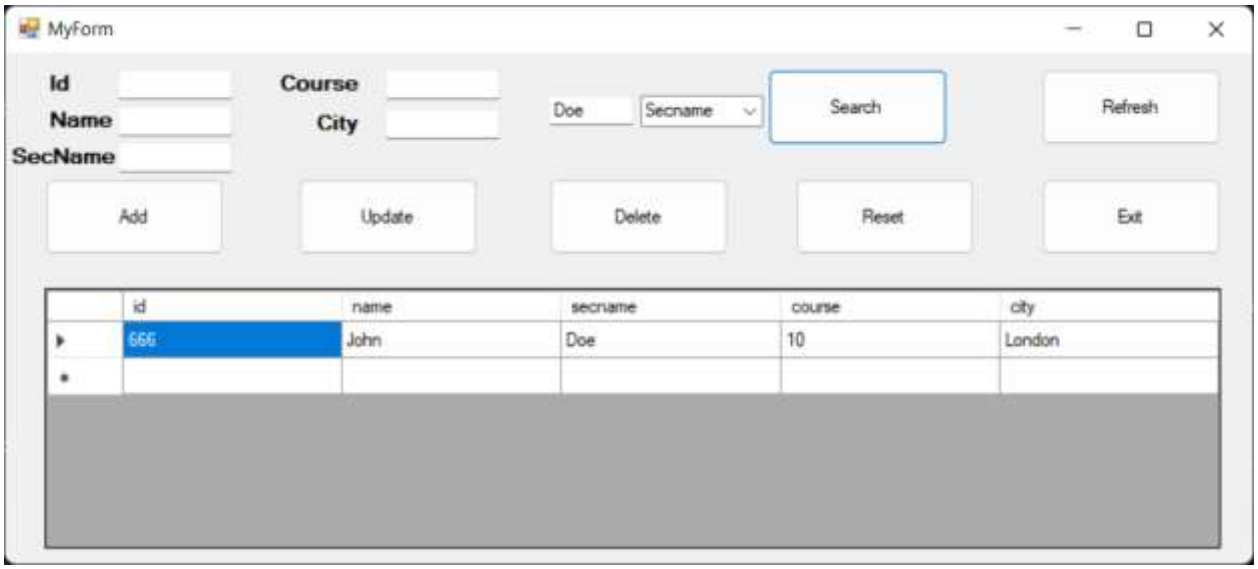


Рисунок 6.36 – Фільтр-пошук за Secname, який дорівнює Doe

Було використано команду перетворення числа на рядок у фільтрі за допомогою команди CONVERT({0}, System.String).

Також використано DataView RowFilter, що є дуже цікавим при написанні. Якщо назва стовпця містить будь-який із цих спеціальних символів ~ ( ) # \ / = > < + - * % & | ^ ' " [ ], ви повинні взяти назву стовпця в квадратні дужки [ ]. Якщо ім'я стовпця містить праву дужку] або зворотну скісну риску \, екрануйте її за допомогою зворотної скісної риски (\] або \\), наприклад:

```
dataView->RowFilter = "id = 10"; // без спеціального символу в назві стовпця "id"
dataView->RowFilter = "$id = 10"; // без спеціального символу в назві стовпця "$id"
dataView->RowFilter = "[#id] = 10"; // спеціальний символ "#" у назві стовпця "#id"
dataView->RowFilter = "[[id\]] = 10"; // спеціальні символи в назві стовпця "[id]".
```

Рядкові значення взяті в одинарні лапки " ". Якщо рядок містить одинарні лапки ', лапки потрібно подвоїти:

```
dataView->RowFilter = "Name = 'John'" // рядкове значення
dataView->RowFilter = "Name = 'John "A"" // рядок з одинарними лапками "John 'A'"
dataView->RowFilter = String.Format("Name = '{0}'", "John 'A'".Replace("'", "''"));
```

Числові значення не укладені в жодні символи. Значення мають бути такими ж, як і результат методу int->ToString() або float->ToString() для інваріантної або англійської абетки.

```
dataView->RowFilter = "Year = 2008" // ціле значення
dataView->RowFilter = "Ціна = 1199,9" // значення з плаваючою точкою
dataView->RowFilter = String.Format( CultureInfo.InvariantCulture.NumberFormat,        "Price = {0}", 1199.9f);
```

Значення дати обмежуються символами # #. Формат дати такий самий, як і результат методу DateTime->ToString() для інваріантної або англійської культури.

```
dataView->RowFilter = "Date = #12/31/2008#" // значення дати (час 00:00:00)
dataView->RowFilter = "Дата = #2008-12-31#" // також цей формат підтримується
dataView->RowFilter = "Дата = #12/31/2008 16:44:58#" // значення дати та часу
dataView->RowFilter = String.Format( CultureInfo.InvariantCulture.DateTimeFormat, "Date = #{0}#", new DateTime(2008, 12, 31, 16, 44, 58));
```

Оператори «рівно», «не рівно», «менше», «більше» використовуються для включення лише значень, які відповідають виразу порівняння. Ви можете використовувати такі оператори = <> < <= > >=.

Порівняння рядків залежить від культури, воно використовує властивість CultureInfo із DataTable.Locale пов’язаної таблиці (dataView->Table->Locale). Якщо властивість не встановлено явно, його значенням за замовчуванням є DataSet->Locale (а його значенням за замовчуванням є поточна культура системи Thread->CurrentThread->CurrentCulture).

```
dataView->RowFilter = "Num = 10" // число дорівнює 10
dataView->RowFilter = "Дата < #1/1/2008#" // дата менша за 1/1/2008
dataView->RowFilter = "Name <> 'John'" // рядок не дорівнює 'John'
dataView->RowFilter = "Name >= 'Jo'" // порівняння рядків.
```

Оператор IN використовується для включення лише значень зі списку. Ви можете використовувати оператор для всіх типів даних, таких як числа або рядки:

```
dataView->RowFilter = "Id IN (1, 2, 3)" // цілі значення
dataView->RowFilter = "Price IN (1.0, 9.9, 11.5)" // плаваючі значення
dataView->RowFilter = "Name IN ('Джон', 'Джим', 'Том')" // рядкові значення
dataView->RowFilter = "Date IN (#12/31/2008#, #1/1/2009#)" // значення дати і часу
dataView->RowFilter = "Id NOT IN (1, 2, 3)" // значення не зі списку.
```

Оператор LIKE використовується для включення лише значень, які відповідають шаблону із символами підстановки. Символом узагальнення є * або %, він може бути на початку шаблону '*value, у кінці value*' або на обох '*value*'. Символ підстановки в середині патерну 'va*lue' заборонений.

```
dataView->RowFilter = "Name LIKE 'j*'" // значення, що починаються з 'j'
dataView->RowFilter = " Name LIKE '%jo%'" // значення, що містять 'jo'
dataView->RowFilter = "Name NOT LIKE 'j*'" // значення, які не починаються з 'j'.
```

Якщо шаблон у реченні LIKE містить будь-який із цих спеціальних символів * % [ ], ці символи мають бути екрановані в дужках [ ], наприклад [*], [%], [[] або []].

```
dataView->RowFilter = "Name LIKE '[*]*'" // значення, що починаються з '*'
dataView->RowFilter = " Name LIKE '[[]*'" // значення, що починаються з '['.
```

Логічні оператори AND, OR і NOT використовуються для об’єднання виразів. Оператор NOT має пріоритет над оператором AND і має пріоритет над оператором OR.

Оператор AND має перевагу над оператором OR, потрібні дужки:

```
dataView->RowFilter = "City = Tokio I (Age < 20 OR Age > 60)";
```

Наступні приклади роблять те саме:

```
dataView->RowFilter = " City <> Tokio TA City <> ' Paris '";
dataView->Refilter = "NOT City = Tokio AND NOT City = ' Paris '";
dataView->RowFilter = "NOT (City = Tokio OR City = ' Paris ')";
dataView->RowFilter = " City NOT IN B (Tokio, ' Paris ')";
```

Арифметичними операторами є додавання +, віднімання -, множення *, ділення / і модуль %.

```
dataView->RowFilter = "MotherAge - Age < 20"; // люди з молодою мамою
dataView->RowFilter = " Age % 10 = 0"; // люди з десятирічним днем народження.
```

6.4 Контрольні завдання

1. Пояснити схему взаємодії .NET-програми з базою даних.
2. Пояснити програмні особливості відкриття бази даних.
3. Пояснити реалізацію функції обміну програми з БД.
4. Пояснити реалізацію функції виведення даних, наведеної у даному розділі.
5. Розробити програму, що забезпечує ведення бази даних про поточну успішність студентів.
6. Розробити програму, що забезпечує реалізацію запитів до бази даних із відомостями про країни світу на основі програми, наведеної у даному розділі.

7 ВИКОРИСТАННЯ ПОТОКОВОЇ БАГАТОЗАДАЧНОСТІ

7.1 Багатозадачність та її основні риси

Багатозадачність завжди вважалася необхідною рисою операційних систем. Колись, у період панування операційної системи MS-DOS, наявність графічної оболонки DosShell, яка начебто підтримувала псевдо-багатозадачність вважалося серйозним чинником, хоча, пам'ятається авторові так і не вдалося упевнитися у справжньому одночасному виконанні декількох програм за допомогою цієї оболонки.

З тих пір (початок 90-х років) комп'ютери стали швидшими на багато порядків. Наприклад така ПЕОМ, як "Іскра 1030.11" (до речі – IBM-сумісний комп'ютер) мала тактову частоту процесора у 4.77 МГц, оперативну пам'ять у 640 кілобайт та жорсткий диск у 10 мегабайт. Комп'ютер, на якому складався посібник [1] – мав процесор Intel Celeron 2200 МГц, оперативну пам'ять 384 мегабайти і жорсткий диск 20 гігабайт, теперішній посібник – на ноутбуці із процесором Intel Core I5 (8^{th} Gen), оперативною пам'яттю 8 гігабайт і двома SSD-дисками (256 і 512 Гб). Ці ПЕОМ розділяє 40 років і величезний прорив у розвитку обчислювальної техніки. Так що комп'ютерна техніка продовжує стрімко розвиватися, що не можна сказати про загальний процес людства, який вможе в один момент знищуватися війною.

Зростання потужності комп'ютерів надало і нові можливості з точки зору багатозадачності. Щоправда, справжня багатозадачність залишається можливою тільки у разі дійсно паралельного виконання декількох обчислювальних операцій. Звичайний персональний комп'ютер 2025 року 2005 року має лише один центральний процесор і за один такт виконує одну просту обчислювальну операцію. Це означає, що дві або більше задач мають виконуватися щонайменше у різні такти роботи процесора у певній послідовності і казати про "справжню" багатозадачність не доводиться.

Дійсна багатозадачність можлива лише у багатопроцесорних системах. Але й те, що центральний процесор під орудою операційної системи має змогу планувати виконання декількох обчислювальних операцій у певній послідовності є суттєвим здобутком і є однією з форм багатозадачності.

На сьогодні багатозадачність не є примхою, а конче необхідною рисою функціонування програмного забезпечення. Справді, зараз досить важко уявити ситуацію, коли немає можливості одночасно працювати з текстом у програмі Microsoft Word, користуватися файловим менеджером, графічним редактором,

одночасно переключатися між декількома Інтернет-сторінками – якщо, звісно, потужність вашого комп’ютера це дозволяє.

Таким чином, якщо комп’ютер має один центральний процесор, функції багатозадачності забезпечуються операційною системою. Microsoft Windows має два типи багатозадачності. В основі першого типу – поняття процесу (process), в основі другого – поняття потоку (thread).

Більшість розробників цілком правильно розуміють концепцію багатозадачності як здатність комп'ютерів виконувати більше однієї програми чи процесу одночасно. Однак, для частини з них, багатопотоковість може бути чужим терміном [2]. Багато програмістів не мали жодних причин програмувати в багатопоточному режимі. Фактично, для деяких мов програмування немає способу виконувати багатопотокове програмування, не проходячи через деякі дуже заплутані програмні рішення.

Отже, що таке багатопотокове програмування? Розробник може думати про це, як про багатозадачність на рівні програми. Програма має два варіанти виконання. Перший варіант – запустити себе в одному потоці виконання. У цьому методі виконання програма послідовно дотримується логіки програми від початку до кінця. Можна думати про цей метод виконання, як про однопотоковий. Другий варіант полягає в тому, що програма може розбити себе на кілька потоків виконання або, іншими словами, розділити програму на кілька сегментів (з початковою та кінцевою точками) та запустити деякі з них одночасно (одночасно). Це те, що більш відоме як багатопотокова робота. Слід зазначити, однак, що кінцевий результат як однопотокової, так і багатопотокової програми буде однаковим.

Звичайно, якщо у вас однопроцесорна машина, справжня паралельність неможлива, оскільки лише одна команда може виконуватися одночасно через процесор. (Завдяки технологіям Hyper-Threading або Multi-Core від Intel Corporation можете виконувати більше однієї команди одночасно на одномупроцесорі, але це тема для окремого розгляду [2-3]) Це важлива концепція, яку слід зрозуміти, оскільки багато програмістів помилково думають, що якщо вони розіб'ють обчислювально пов'язану частину програми на дві частини та запустять їх у двох потоках виконання, програма виконуватиметься швидше.

Насправді все навпаки — це займе більше часу. Причина полягає в тому, що для програми виконується такий самий обсяг коду, плюс необхідно додати додатковий час для обробки обміну контекстом потоку (регістри процесора, стек тощо). Тож з якої причини програмісти мають використовували багатопоточність для однопроцесорного комп'ютера, якщо це займає більше часу, ніж

однопоточність? Причина полягає в тому, що за правильного використання багатопоточність може забезпечити кращий час відгуку, пов'язаний з вводом/виводом, а також краще використання процесора [2].

Ключовим моментом правильного використання багатопоточності є типи команд, які виконують потоки. Обчислювально пов'язані потоки (тобто потоки, які виконують багато обчислень) отримують дуже мало користі, коли справа доходить до багатопоточності, оскільки вони вже працюють понаднормово, намагаючись власноруч виконатися. Багатопоточність уповільнює цей тип потоків. Потоки вводу/виводу, з іншого боку, отримують значний прибуток.

Цей прибуток найбільш очевидний у двох областях: краща реакція та використання процесора.

Усі розробники стикалися з програмою, яка, здавалося, зупинилася або зависла, а потім раптово повернулася до життя. Зазвичай причиною цього є те, що програма виконує обчислювально пов'язану ділянку коду. І, оскільки багатопоточність не виконувалася, не було передбачено циклів процесора для взаємодії користувача з комп'ютером. Додаючи багатопоточність, можна мати один потік, який виконував обчислювальну обмежену область, а інший обробляв взаємодію з користувачем. Наявність потоку вводу/виводу дозволяє користувачеві продовжувати працювати, поки процесор пробивається через обчислювальний обмежений потік. Щоправда, сам обчислювальний обмежений потік виконуватиметься довше, але оскільки користувач може продовжувати працювати, цей незначний проміжок часу зазвичай не має значення.

Потоки вводу/виводу відомі тим, що марнують цикли процесора. Люди, принтери, жорсткі диски, монітори, тощо є дуже повільними порівняно з процесором. Потоки вводу/виводу витрачають значну частину свого часу просто на очікування, нічого не роблячи. Таким чином, багатопоточність дозволяє процесору використовувати цей втрачений час [2-3].

Багатозадачність на основі процесів підтримується вже з перших версій Windows. Процесом є окрема програма, що виконується операційною системою. У багатозадачності такого типу дві або більше програм можуть виконуватися одночасно.

Потік є частиною процесу, що виконується. Кожен процес складається щонайменше з одного потоку, але може містити два чи більше потоків. За потокової багатозадачності декілька частин однієї програми можуть виконуватися одночасно. Потокова багатозадачність підтримується Windows починаючи з Windows ’95 та Windows NT (до речі, Microsoft Visual C++ 1.0 не містить підтримки потокової багатозадачності).

Потокова багатозадачність надає змогу розробляти ефективні програми шляхом розділення їх на окремі виконувані блоки і керування ходом виконання усієї програми в цілому. Із використанням потокової багатозадачності виникає необхідність використання спеціального механізму, який дозволяв би контролювати виконання потоків (і процесів) у чітко визначений спосіб. Цей механізм називається синхронізацією [2, 11-14].

7.2 Багатопотоковість у .NET

В .NET визначено один простір імен, необхідний для обробки потоків:

System::Threading

Різні завдання багатозадачності і багатопотоковості визначають, який із класів необхідно використовувати. Багато класів надають різні способи виконання однієї й тієї ж дії, зазвичай відрізняючись ступенем контролю.

Розглянемо список основних класів у просторі імен System::Threading:

Таблиця 7.1 – Основні класи простору імен System::Threading

Клас	Коментар
1	2
AutoResetEvent	повідомляє потік, що очікує, про те, що сталася подія. Клас використовується, щоб дозволити зв'язок між потоками за допомогою сигналізації. Зазвичай використовуються для потоків, яким потрібен ексклюзивний доступ
Interlocked	дозволяє атомарні операції зі статичною змінною, яка є спільною для потоків
ManualResetEvent	повідомляє один або кілька потоків про те, що сталася подія. Цей клас використовується для забезпечення зв'язку між потоками за допомогою сигналізації. Зазвичай цей клас використовується для сценаріїв, де один потік має завершитися, перш ніж інші потоки зможуть продовжити роботу.
Monitor	надає механізм синхронізації доступу до об'єктів шляхом блокування доступу до блоку, який зазвичай називають критичним розділом. Хоча потік володіє блокуванням об'єкта, жоден інший потік не може отримати це блокування

1	2
Mutex	надає спосіб синхронізації, який вирішує проблему двох або більше потоків, яким потрібен доступ до спільного ресурсу одночасно. Він гарантує, що лише один потік одночасно використовуватиме ресурс. Цей клас подібний за функціональністю до Monitor, за винятком того, що Mutex дозволяє міжпроцесну синхронізацію
ReaderWriterLock	дозволяє одному записувачу та кільком читачам доступ до ресурсу. У будь-який момент часу він дозволяє або одночасний доступ на читання для кількох потоків, або доступ на запис до одного потоку
Semaphore	обмежує кількість потоків, які можуть отримати доступ до певного системного ресурсу
Thread	основний клас для створення потоку для виконання частини програмного коду
ThreadPool	надає доступ до пулу потоків, що підтримуються системою
WaitHandle	дозволяє отримувати або звільняти ексклюзивний доступ до спільного системного ресурсу

З попереднього списку класів видно, що бібліотека класів .NET Framework надає два способи створення потоків:

– Thread;
– ThreadPool.

Різниця між ними залежить головним чином від того, чи необхідно підтримувати об'єкт Thread або чи є необхідність виконання цих операцій самою ОС. Фактично, майже однакових результатів можна досягти за допомогою будь-якого з цих методів.

Пул потоків – це набір робочих потоків, які ефективно виконують асинхронні зворотні виклики від імені програми. Пул потоків в основному використовується для зменшення кількості потоків програми та забезпечення управління робочими потоками. Програми можуть ставити робочі елементи в чергу, пов'язувати роботу з дескрипторами очікування, автоматично ставити в чергу на основі таймера та прив'язуватися до вводу-виводу.

Наступні програми можуть отримати користь від використання пулу потоків:

1) Програма з високим рівнем паралельності, яка може асинхронно відправляти велику кількість невеликих робочих елементів (таких як розподілений пошук по індексу або мережевий ввід/вивід).

2) Програма, яка створює та знищує велику кількість потоків, кожен з яких виконується протягом короткого часу. Використання пулу потоків може зменшити складність керування потоками та накладні витрати, пов'язані зі створенням та знищенням потоків.

3) Програма, яка обробляє незалежні робочі елементи у фоновому режимі та паралельно (наприклад, завантаження кількох вкладок).

4) Програма, яка повинна виконувати ексклюзивне очікування об'єктів ядра або блокувати вхідні події об'єкта. Використання пулу потоків може зменшити складність керування потоками та підвищити продуктивність, зменшуючи кількість перемикачів контексту.

5) Програма, яка створює власні потоки-очікувачі для очікування подій.

7.3 Створення та використання робочих потоків

7.3.1 Загальна інформація

Потокова багатозадачність надає розробнику програмного забезпечення нові можливості із контролювання виконання окремих частин програми. У багатопотоковій програмі один потік можна зв’язати, наприклад, із сортуванням даних програми, другий – із збиранням інформації про віддалений ресурс проекту, третій – із організацією інтерфейсу користувача.

Усі процеси складаються щонайменше із одного потоку виконання – головного потоку. Після виклику головного потоку, до нього можна під’єднати виконання додаткових потоків. При цьому і головний і додаткові потоки виконуватимуться не послідовно (як функції програми), а паралельно. У цьому, власне, полягає потокова багатозадачність.

Кожен процес надає ресурси, необхідні для виконання програми [14]. Процес має віртуальний адресний простір, виконуваний код, відкриті дескриптори системних об'єктів, контекст безпеки, унікальний ідентифікатор процесу, змінні середовища, клас пріоритетів, мінімальний та максимальний розміри робочого набору та принаймні один потік виконання. Кожен процес запускається з одного потоку, який часто називають основним потоком, але він може створювати додаткові потоки з будь-якого зі своїх потоків.

Потік – це сутність у процесі, виконання якої можна запланувати. Усі потоки процесу спільно використовують свій віртуальний адресний простір та си-

стемні ресурси. Крім того, кожен потік підтримує обробники винятків, пріоритет планування, локальне сховище потоків, унікальний ідентифікатор потоку та набір структур, які система використовуватиме для збереження контексту потоку, доки його не буде заплановано. Контекст потоку включає набір машинних регістрів потоку, стек ядра, блок середовища потоку та стек користувача в адресному просторі процесу потоку. Потоки також можуть мати власний контекст безпеки, який можна використовувати для імітації клієнтів.

Microsoft Windows підтримує витісняючу багатозадачність, яка створює ефект одночасного виконання кількох потоків з кількох процесів. На багатопроцесорному комп'ютері система може одночасно виконувати стільки потоків, скільки процесорів на комп'ютері [14].

Об'єкт завдання дозволяє керувати групами процесів як єдиним цілим. Об'єкти завдань – це іменовані, захищені та спільно використовувані об'єкти, які контролюють атрибути пов'язаних з ними процесів. Операції, що виконуються над об'єктом завдання, впливають на всі процеси, пов'язані з об'єктом завдання.

Програма може використовувати пул потоків для зменшення кількості потоків програми та забезпечення управління робочими потоками. Програми можуть ставити в чергу робочі елементи, пов'язувати роботу з дескрипторами очікування, автоматично ставити в чергу на основі таймера та прив'язуватися до вводу-виводу.

Планування в режимі користувача (UMS) – це легкий механізм, який програми можуть використовувати для планування власних потоків. Програма може перемикатися між потоками UMS у режимі користувача без залучення системного планувальника та відновлювати контроль над процесором, якщо потік UMS блокується в ядрі. Кожен потік UMS має власний контекст потоку, замість того, щоб спільно використовувати контекст потоку одного потоку. Можливість перемикатися між потоками в режимі користувача робить UMS ефективнішим, ніж пули потоків, для короткочасних робочих елементів, які потребують мало системних викликів.

Волокно (fiber) – це одиниця виконання, яку програма має планувати вручну [14]. Волокна виконуються в контексті потоків, які їх планують. Кожен потік може планувати кілька волокон. Загалом, волокна не надають переваг над добре розробленою багатопотоковою програмою. Однак використання волокон може спростити перенесення програм, які були розроблені для планування власних потоків.

У .NET (як раніше і в MFC) визначаються два типи потоків: інтерфейсні і робочі. Інтерфейсні потоки характерні тим, що є здатними приймати та обробляти повідомлення. Тут слід згадати найпростішу C++/CLI-програму. Вона, як пам’ятаєте, починається зі створення об’єкта форми і передачі форми об’єкту Application для запуску функцією Run(). Це і є початок головного потоку програми.

Робочі потоки, на відміну від інтерфейсного, не приймають і не обробляють повідомлень. Вони спрямовуються на забезпечення виконання додаткових шляхів виконання окремих завдань інтерфейсного потоку. Додатково зазначимо, що на рівні WinAPI немає різниці між робочими і інтерфейсними потоками.

Використання багатопотокового режиму роботи програми потребуватиме підключення простору імен Threading та інших заголовочних файлів.

7.3.2 Реалізація багатопотоковості стандартними компонентами

Компонент BackgroundWorker надає можливість виконувати трудомісткі операції асинхронно («у фоні») у потоці, відмінному від основного потоку інтерфейсу користувача вашої програми. Щоб використовувати BackgroundWorker, необхідно просто вказати йому, який трудомісткий робочий метод виконувати у фоні, а потім викликати метод RunWorkerAsync(). Викликаючий потік продовжує працювати нормально, поки робочий метод виконується асинхронно. Коли метод завершується, BackgroundWorker сповіщає викликаючий потік, запускаючи подію RunWorkerCompleted, яка за бажанням містить результати операції.

Компонент BackgroundWorker доступний з панелі інструментів, на вкладці «Компоненти». Щоб додати BackgroundWorker до форми, перетягніть компонент BackgroundWorker на форму. Він з'явиться в переліку компонентів, а його властивості відображатимуться у вікні «Властивості».

Щоб розпочати асинхронну операцію, використовується метод RunWorkerAsync(). RunWorkerAsync() приймає необов'язковий параметр об'єкта, який можна використовувати для передачі аргументів вашому робочому методу. Клас BackgroundWorker надає подію DoWork(), до якої ваш робочий потік приєднаний через обробник подій DoWork().

Обробник подій DoWork() приймає параметр DoWorkEventArgs, який має властивість Argument. Ця властивість отримує параметр від RunWorkerAsync і може бути передана робочому методу, який буде викликано в обробнику подій DoWork.

Для реалізації режиму багатопотоковості зробимо наступні кроки:

1. Побудуємо стандартний C++/CLI-проект з формою MyForm і необхідними налаштуваннями.

2. У панелі інструментів ToolBox оберемо підсписок “Components” і додамо у форму два потоки backGroundWorker1() і backGroundWorker2(), як показано на Рис. 7.1:

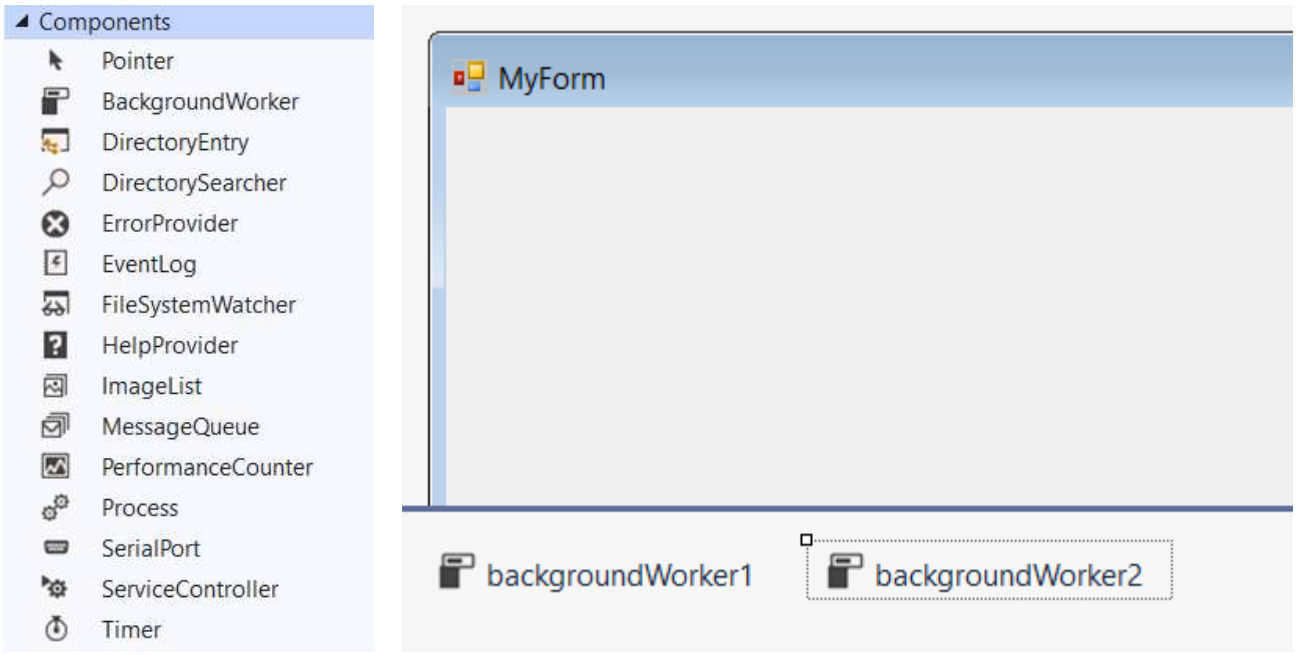


Рисунок 7.1 – Додавання компонентів BackgroundWorker до MyForm

3. Подвійними натисками на backGroundWorker1() і backGroundWorker2() створимо код фонових потоків, які виводитимуть у форму прямокутники, що рухатимуться зліва – праворуч, у наступному вигляді:

```
private:  System::Void  backgroundWorker1_DoWork(System::Object^  sender,  System::ComponentModel::DoWorkEventArgs^ e)
      {     Graphics^ g = CreateGraphics();
            SolidBrush^ greenBrush = gcnew SolidBrush(Color::Green);
            for (int i = 0; i < 20; i++)
            {     Rectangle r0(10 + 25 * (i-1), 10, 150, 110);
                  g->FillRectangle(gcnew SolidBrush(MyForm::BackColor),r0);
                  Rectangle r(10 + 25 * i, 10, 150, 110);
                  g->FillRectangle(greenBrush,r);
                  System::Threading::Thread::Sleep(300);
            }
      }

private:  System::Void  backgroundWorker2_DoWork(System::Object^  sender,  System::ComponentModel::DoWorkEventArgs^ e)
      {     Graphics^ g = CreateGraphics();
            SolidBrush^ blueBrush = gcnew SolidBrush(Color::Blue);
            for (int i = 0; i < 20; i++)
            {     Rectangle r0(10 + 15 * (i-1), 210, 150, 110);
                  g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
                  Rectangle r(10 + 15 * i, 210, 150, 110);
```

```
                g->FillRectangle(blueBrush, r);
                System::Threading::Thread::Sleep(225);
            }
        }
```

4. У формі MyForm реалізуємо кнопку button1, натиск якої забезпечуватиме запуск потоків функцією RunWorkerAsync():

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{
backgroundWorker1->RunWorkerAsync();
      backgroundWorker2->RunWorkerAsync();
}
```

5. Результатом виконання потоків стане виведення горизонтально рухомих прямокутників, відповідних потокам backGroundWorker1() і backGroundWorker2(), як це показано на Рис. 7.2:

Рисунок 7.2 – Реалізація двох потоків із виведенням прямокутників

Аналогічно можна реалізувати і більшу кількість потоків, якщо це необхідно в програмі.

7.3.3 Реалізація багатопотоковості за допомогою потокових об’єктів

Спосіб реалізації багатопотоковості, показаний в п. 7.1 і який реалізується в Visual Studio 2022 не описано в літературі [2, 3, 7]. Натомість там пропонується використання класичних методів простору імен ::Threading.

При цьому, потокові функції додаються в C++/CLI-проект у якості статичних (це треба додатково вказати) методів MyForm:

```
public:
	static void Thread01(Object^ data);
	static void Thread02(Object^ data);
```

Самі статичні методи (аналогічні потокам у 7.1) реалізуються у наступний спосіб:

```
void Project7::MyForm::Thread01(Object^ data)
{	MyForm^ form = (MyForm^)data;
	Graphics^ g = form->CreateGraphics();
	SolidBrush^ greenBrush = gcnew SolidBrush(Color::Green);
	for (int i = 0; i < 20; i++)
	{	Rectangle r0(10 + 25 * (i - 1), 10, 100, 100);
		g->FillRectangle(gcnew SolidBrush(form->BackColor), r0);
		Rectangle r(10 + 25 * i, 10, 100, 100);
		g->FillRectangle(greenBrush, r);
		System::Threading::Thread::Sleep(300);

	}
}

void Project7::MyForm::Thread02(Object^ data)
{	MyForm^ form = (MyForm^)data;
	Graphics^ g = form->CreateGraphics();
	SolidBrush^ blueBrush = gcnew SolidBrush(Color::Blue);
	for (int i = 0; i < 20; i++)
	{	Rectangle r0(10 + 15 * (i - 1), 120, 100, 100);
		g->FillRectangle(gcnew SolidBrush(form->BackColor), r0);
		Rectangle r(10 + 15 * i, 120, 100, 100);
		g->FillRectangle(blueBrush, r);
		System::Threading::Thread::Sleep(225);
	}
}
```

Виклик таких потоків, як і раніше можна забезпечити з обробника кнопки форми, хоча самі виклики не є надто наочними. Слід окремо зазначити спосіб, яким передається параметри у потоки – через метод Start(), чиїм параметром стає this (покажчик на форму MyForm). Покажчик на форму попадає в потік в якості параметру Object^ data, тобто без точного вказання типу параметра, але надалі перетворюється у тип MyForm, завдяки чому забезпечується повний доступ до графіки форми MyForm.

```
	private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
	{
	Threading::Thread^ newThread1 = gcnew Threading::Thread(gcnew Thread-
ing::ParameterizedThreadStart(MyForm::Thread01));
	newThread1->Start(this);

	Threading::Thread^ newThread2 = gcnew Threading::Thread(gcnew Thread-
ing::ParameterizedThreadStart(MyForm::Thread02));
	newThread2->Start(this);
	}
```

Практичним недоліком такої реалізації є визначення потоків безпосередньо в просторі проєкту. Недолік виправляється, якщо виконати наступні дії:

1. оголосити потокові функції Thread3() та Thread4() глобальними – на початку файла MyForm.h:

```
using namespace System;
void Thread03(Object^ data);
void Thread04(Object^ data);
```

2. реалізувати потоки як глобальні функції (поза простором імен проєкту, в кінці файла MyForm.h), одночасно вказавши на використання цього простору, наприклад:

```
using namespace Project7;

void Thread03(Object^ data)
{	MyForm^ form = (MyForm^)data;
	Graphics^ g = form->CreateGraphics();
	SolidBrush^ oredBrush = gcnew SolidBrush(Color::OrangeRed);
	for (int i = 0; i < 20; i++)
	{
		Rectangle r0(10 + 25 * (i - 1), 230, 100, 100);
		g->FillRectangle(gcnew SolidBrush(form->BackColor), r0);
		Rectangle r(10 + 25 * i, 230, 100, 100);
		g->FillRectangle(oredBrush, r);
		System::Threading::Thread::Sleep(150);

	}
}

void Thread04(Object^ data)
{	MyForm^ form = (MyForm^)data;
	Graphics^ g = form->CreateGraphics();
	SolidBrush^ bluevBrush = gcnew SolidBrush(Color::BlueViolet);
	for (int i = 0; i < 20; i++)
	{
		Rectangle r0(10 + 25 * (i - 1), 340, 100, 100);
		g->FillRectangle(gcnew SolidBrush(form->BackColor), r0);
		Rectangle r(10 + 25 * i, 340, 100, 100);
		g->FillRectangle(bluevBrush, r);
		System::Threading::Thread::Sleep(300);

	}
}
```

3. Здійснити виклик потоків з обробника однієї з клавіш форми MyForm:

```
	private: System::Void button3_Click(System::Object^ sender, System::EventArgs^ e)
	{
	Threading::Thread^ newThread3 = gcnew Threading::Thread(gcnew Thread-
ing::ParameterizedThreadStart(Thread03));
		newThread3->Start(this);

	Threading::Thread^ newThread4 = gcnew Threading::Thread(gcnew Thread-
ing::ParameterizedThreadStart(Thread04));
		newThread4->Start(this);
	}
```

7.4 Керування пріоритетами процесів та потоків

Кожен процес та потік має певні характеристики пріоритету. Ці характеристики визначають рівень першочерговості виконання процесу або потоку у порівнянні з іншими процесами та потоками, що одночасно виконуються операційною системою.

Microsoft Windows (як і інші операційні системи) має спеціальну системну програму – Диспетчер завдань (Task Manager), завданням якої є відображення стану програм, що запущені користувачем або автоматично використовуються операційною системою. Одночасно ця програма дозволяє контролювати стан завантаженості процесора, взаємодію з локальною мережею, стан користувачів системи тощо. Серед можливостей Task Manager – і контролювання рівня пріоритету процесів. Для отримання (і зміни) пріоритету програми достатньо перейти на сторінку “Processes”, обрати необхідну програму та, натиснувши праву клавішу миші увійти у спливаюче меню, де обрати “Set Priority”, як і показано на рис. 7.3.

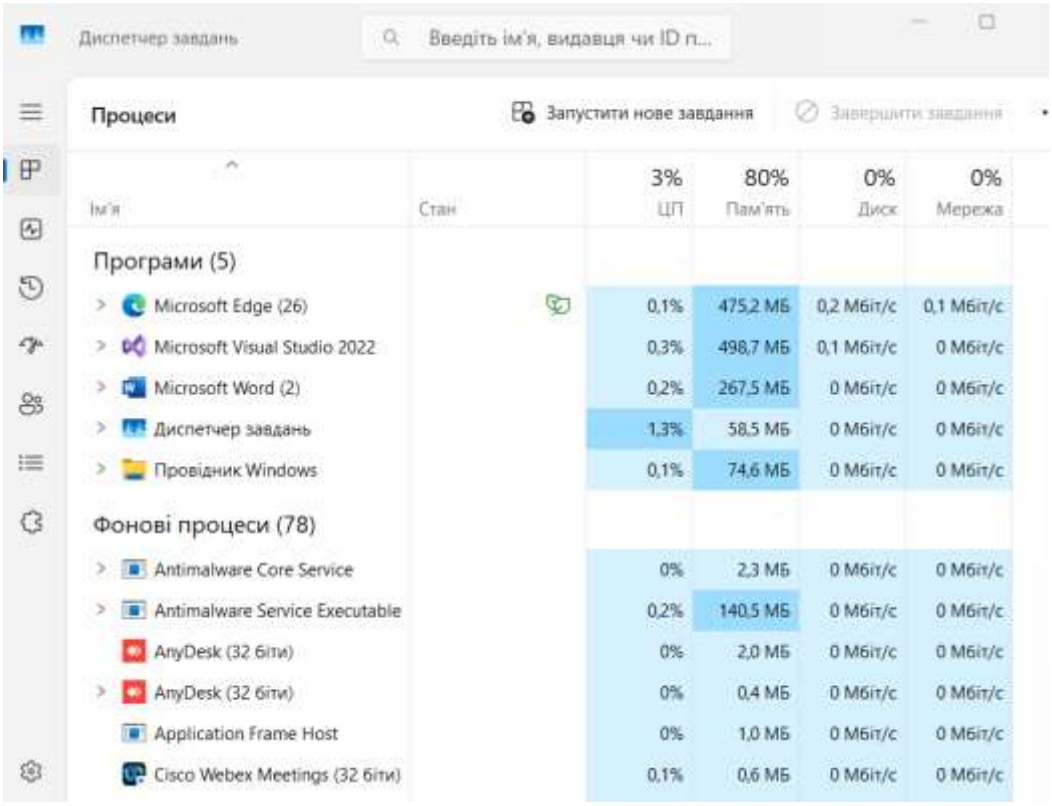


Рисунок 7.3 – Вигляд вікна програми Task Manager

Аналогічно до процесів, кожен потік має свої власні рівні пріоритетів. Рівень пріоритету є комбінацією двох значень: загального значення класу пріоритету процесу і значення пріоритету потоку, відносно пріоритету процесу.

Реальний пріоритет потоку є сумою пріоритету класу процесу і рівня пріоритету потоку. Пріоритет процесу вказує, скільки робочого часу процесора має витрачатися на його потреби (чим більший пріоритет – тим більше часу має витрачатися, чим менший – тим менше часу).

За допомогою спеціальних функцій WinAPI можна отримати інформацію про клас пріоритету процесу і встановити нове значення класу пріоритету. Ці можливості надаються функціями *GetPriorityClass()* і *SetPriorityClass()*:

```
DWORD GetPriorityClass(HANDLE hProcess);
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
```

де *hProcess* – дескриптор процесу, а *dwPriorityClass* – значення класу пріоритету процесу. Значення пріоритетів займають таблицю 7.1.

Таблиця 7.1 – Значення пріоритетів процесів

Значення	Коментар
HIGH_PRIORITY_CLASS	Встановлюється для процесів, виконання яких є критичним в часі і має здійснюватися першочергово у порівнянні з процесами із значеннями пріоритету NORMAL або IDLE. Приклад – Windows Task Manager. Слід обережно використовувати цей рівень, бо його використання займатиме основні ресурси центрального процесора системи.
IDLE_PRIORITY_CLASS	Використовується в разі простою системи. Процес такого рівня може перериватися будь-яким іншим процесом більш високого рівня пріоритету. Прикладом є програми – зберігачі екрана (screen savers).
NORMAL_PRIORITY_CLASS	Рівень пріоритету за замовчуванням. Характеризує процеси без спеціальних вимог.
REALTIME_PRIORITY_CLASS	Застосовується для процесу, що має найвищий у системі пріоритет і здатний переривати роботу усіх інших процесів, включно із процесами операційної системи. Забезпечує захоплення усіх ресурсів центрального процесора.

Додатково зазначимо, що в операційній системі Windows 2000 з'явилися два нових класи пріоритетів: BELOW_NORMAL_PRIORITY_CLASS (нижчий від нормального) та ABOVE_NORMAL_PRIORITY_CLASS (вищий за нормальний).

Встановлення рівня пріоритету процесу можна реалізувати, зокрема, у будь-якій функції форми, наприклад, натискання кнопки 2:

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e)
{
 HANDLE proc = GetCurrentProcess();  // отримання дескриптора процесу
 SetPriorityClass(proc, IDLE_PRIORITY_CLASS); // встановлення рівня пріоритету
}
```

Результат застосування функцій зміни пріоритету процесу легко проконтролювати у диспетчері завдань Microsoft Windows (Рис. 7.4), де це показано для проєкту Project12.

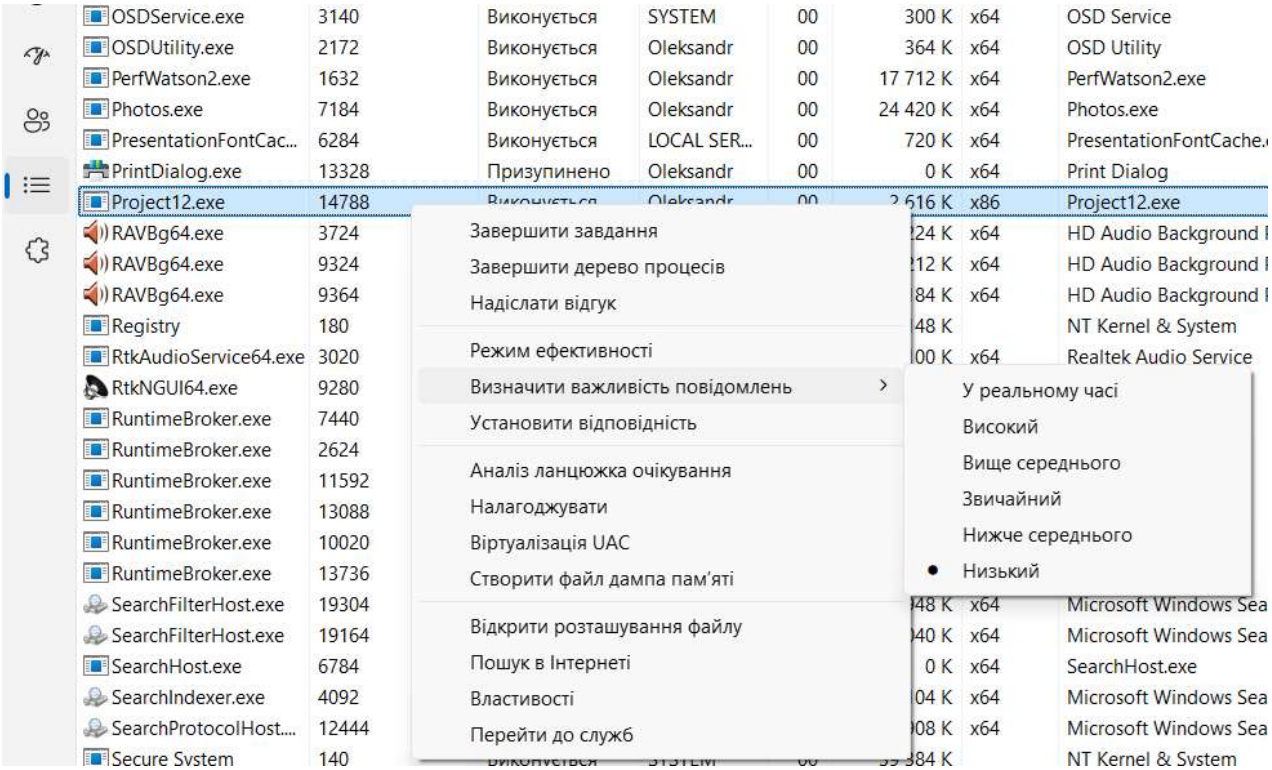


Рисунок 7.4 – Перевірка результату функції зміни пріоритету процесу у програмі Task Manager

Потік, як частина процесу, має своє значення пріоритету. Пріоритет потоку означає скільки часу центрального процесора системи займається у загальному часі процесу. За замовчуванням, усі потоки створюються із нормальним (THREAD_PRIORITY_NORMAL) рівнем пріоритету.

Пріоритет потоку може бути встановлено або отримано за допомогою функцій SetThreadPriority() та GetThreadPriority():

```
BOOL SetThreadPriority (int nPriority);      // встановлює пріоритет
int GetThreadPriority ();                    // повертає значення пріоритету
```

де *nPriority* – встановлюваний рівень пріоритету, що має приймати такі значення:

```
THREAD_PRIORITY_TIME_CRITICAL          // найвищий рівень
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_IDLE                   // найнижчий рівень
```

Зміст значень, перелічених від найвищого пріоритету до найнижчого, легко зрозуміти. Зазначимо, що WinAPI надає ширші можливості у встановленні пріоритетів. Зокрема, MSDN дає інформацію про 31 рівень пріоритетів потоків та процесів.

7.5 Синхронізація та об’єкти синхронізації

7.5.1 Загальні риси синхронізації

Багатозадачність і, включно, багатопотоковість є корисними явищами, що можуть суттєво поліпшити роботу програміста. Але така корисність не завжди є беззаперечною. Зокрема, можна легко уявити ситуацію, коли декілька потоків прагнуть отримати доступ до одного й того самого файла і одночасно здійснюють записи у цей файл, зовсім не прагнучи координувати свої дії. Або, коли одночасно завантажити на програвання комп’ютером декілька різних звукових кліпів – слухач почує їх одночасно (можливо, якщо програвати звукові файли різними програмами), але навряд чи отримає задоволення. У таких випадках багатозадачність стає на заваді цілісності зберігання або виведення інформації.

Синхронізація є спеціальним засобом організації доступу декількох потоків програми до спільних ресурсів. Уже зазначалося, що відсутність такого механізму може призводити до небажаних та непередбачуваних результатів. .NET забезпечує створення спеціальних об’єктів синхронізації за допомогою відповідних класів.

Типова багатопотокова програма містить ресурси, що мають розподілятися між потоками – розподілені ресурси. Під час роботи з розподіленими ресурсами кожен потік може знаходитися у двох сталих ситуаціях:

1) потік може виконуватися або бути готовим до виконання;

2) потік може бути заблокованим – його виконання може бути призупиненим доки не буде звільнено необхідний ресурс або не відбудеться певна програмна подія.

7.5.2 Об’єкти синхронізації

.NET та Win32 підтримують 4 типи об’єктів синхронізації. Усі з них ґрунтуються на понятті семафора.

1. Класичний семафор (classic semaphore) – об’єкт синхронізації, що дозволяє обмеженій кількості потоків доступ до розподіленого ресурсу. При цьому доступ до ресурсу або обмежується повністю (тоді тільки один потік або процес матиме доступ до ресурсу продовж певного проміжку часу) або дозволяється для обмеженої кількості потоків чи процесів. Програмно семафор реалізується у вигляді лічильника, що зменшується, коли потік (процес) отримує семафор та збільшується, коли потік (процес) звільнює його.

2. Виключний семафор (mutex semaphore) – об’єкт синхронізації, що забезпечує повне обмеження доступу до спільного ресурсу. Дозволяє певному визначеному потоку здійснити виключний доступ до ресурсу. Виключні семафори використовуються, коли модифікація або інший вид керування даними мають забезпечуватися тільки одним потоком.

3. Подія (або об’єкт події event) – об’єкт синхронізації, що здійснює повідомлення одним потоком до іншого потоку про настання певної програмної події. Використовується для блокування доступу до ресурсу, доки один потік не повідомить, що спільний ресурс може бути використано. Об’єкт події є корисним у випадку, коли потокові необхідно повідомити про дозвіл на виконання завдання.

4. Критичний розділ (critical section) – об’єкт синхронізації, що забезпечує виключне використання визначеного спільного ресурсу або частини коду програми одним потоком (процесом) та блокування усіх інших потоків (процесів). Критичні розділи є корисними, коли тільки один потік має виконувати певну функцію програми, модифікувати дані або інші контрольовані ресурси. Цей об’єкт синхронізації в .NET замінено.

7.5.3 Класи синхронізації

Класи об’єктів синхронізації використовуються, коли для гарантування цілісності розподіленого ресурсу, програма має контролювати доступ до такого ресурсу. Класи забезпечення доступу використовуються для отримання доступу до цих керованих ресурсів. Для визначення, який клас синхронізації необхідно використовувати, MSDN рекомендує отримати відповіді на ряд запитань:

1) чи має програма очікувати на певну подію, перед тим як отримати доступ до ресурсу (наприклад, спочатку дані мають бути отримані з послідовного порту і лише потім записуватися у файл), якщо так – оберіть клас події;

2) чи може більше одного потоку всередині однієї програми одночасно отримати доступ до ресурсу, якщо так – оберіть класичний сефамор;

3) чи може більше однієї програми використовувати ресурс (наприклад DLL – динамічно зв’язувану бібліотеку), якщо так – використайте виключний семафор.

7.5.4 Використання виключного семафора Mutex

Коли двом або більше потокам потрібно одночасно отримати доступ до спільного ресурсу, системі потрібен механізм синхронізації, щоб гарантувати, що лише один потік одночасно використовує цей ресурс. Mutex є об’єктом синхронізації, який надає виключний доступ до спільного ресурсу лише одному потоку. Якщо потік отримує Mutex, другий потік, який хоче отримати цей виключний семафор, призупиняється, доки перший потік не звільнить його.

Виключні семафори бувають двох типів: локальні, які не мають назви, та іменовані системні виключні семафори. Локальний виключний семафор існує лише в межах вашого процесу. Його може використовувати будь-який потік у вашому процесі, який має посилання на об'єкт Mutex, що представляє виключний семафор. Кожен неіменований об'єкт Mutex представляє окремий локальний виключний семафор.

Іменовані системні виключні семафори є видимими в усій операційній системі та можуть використовуватися для синхронізації дій процесів. Можна створити об'єкт Mutex, який представляє іменований системний виключний семафор, за допомогою конструктора, який приймає ім'я. Об'єкт операційної системи може бути створений одночасно або він може існувати до створення об'єкта Mutex. Можна також створити кілька об'єктів Mutex, які представляють один і той самий іменований системний виключний семафор, і використовувати метод OpenExisting, щоб відкрити існуючий іменований системний виключний семафор [11].

Приклад, що реалізує керування виключним семафором роботи двох потоків може бути зроблено виконанням наступних кроків:

1. Для реалізації прикладу з виключним семафором Mutex додати у Windows Form програми два фонових потоки backgroundWorker3() та backgroundWorker4(), в нижній частині форми MyForm, як це показано на Рис. 7.5:

Рисунок 7.5 – Додавання третього та четвертого потоків у форму MyForm

2. Запуск потоків реалізувати у обробнику кнопки на формі MyForm:

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{     backgroundWorker3->RunWorkerAsync();
            backgroundWorker4->RunWorkerAsync();
}
```

Як видно з обробника, перший запускається третій потік, потім – четвертий.

3. Для реалізації виключного семафора оголосити об’єкт класу Mutex у класі форми MyForm (показано частину класу MyForm):

```
public ref class MyForm : public System::Windows::Forms::Form
{
 public:
      MyForm(void)
      {
      InitializeComponent();
      }
      static Mutex ^mutex1 = gcnew Mutex(); // оголошення виключного семафора
///// продовження коду класу MyForm
};
```

4. Подвійним натиском на компоненти потоків у MyForm створити потокові функції backgroundWorker3() та backgroundWorker4(), в яких реалізувати блокування потоку виключним семафором за допомогою методу WaitOne() і його вивільнення методом ReleaseMutex().

Як видно коду прикладу, у потоках реалізується завантаження зображень автомобілів "Lotus.png" і "Ferrari.png" (файли необхідно скопіювати у теку проєкту). Потік, який запускається першим (дивіться п. 2), викликом mutex1-

>WaitOne() блокує доступ до виведення у форму і звільняє доступ викликом mutex1->ReleaseMutex(). Таким чином, Lotus проходить шлях першим, Ferrari – другим (Рис. 7.6).

```
      private:  System::Void  backgroundWorker3_DoWork(System::Object^  sender,  Sys-
tem::ComponentModel::DoWorkEventArgs^ e)
      {     Graphics^ g = CreateGraphics();
            Image^ image1 = Image::FromFile("Lotus.png"); // розмір 175x100
            mutex1->WaitOne();
            for (int i = 0; i < 60; i++)
            {     Rectangle r0(10 + 5 * (i - 1), 10, 175, 100);
                  g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
                  g->DrawImage(image1, 10 + 5 * i,10);
                  System::Threading::Thread::Sleep(5);
            }
            mutex1->ReleaseMutex();
      }
      private:  System::Void  backgroundWorker4_DoWork(System::Object^  sender,  Sys-
tem::ComponentModel::DoWorkEventArgs^ e)
      {     Graphics^ g = CreateGraphics();
            Image^ image2 = Image::FromFile("Ferrari.png"); // розмір 214x100
            mutex1->WaitOne();
            for (int i = 0; i < 50; i++)
            {     Rectangle r0(10 + 5 * (i - 1), 120, 214, 100);
                  g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
                  g->DrawImage(image2, 10 + 5 * i, 120);
                  System::Threading::Thread::Sleep(6);
            }
            mutex1->ReleaseMutex();
      }
```

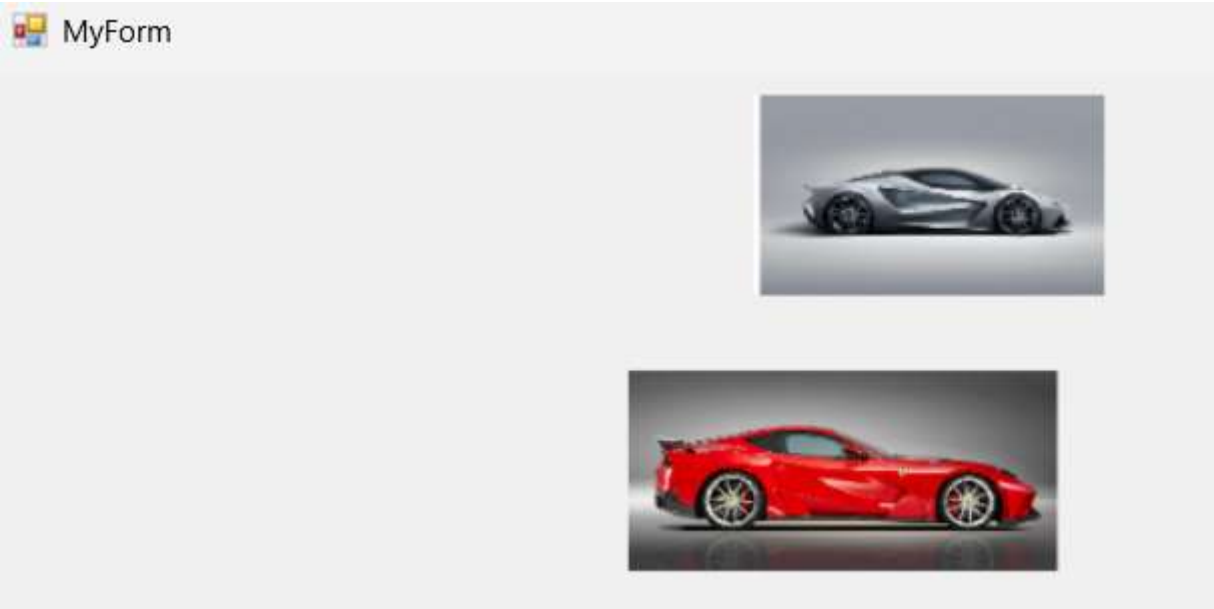


Рисунок 7.6 – Результат прикладу з об’єктом синхронізації Mutex

7.5.5 Використання класичного семафора Semaphore

На відміну від виключного семафора, класичний семафор Semaphore здатен забезпечувати багатопотоковий режим для вказаної кількості потоків. При цьому у конструкторі об’єкта класу Semaphore вказується початкова кількість потоків-учасників та їх максимальна кількість.

Клас Semaphore використовують для керування доступом до пулу ресурсів. Потоки входять до семафора, викликаючи метод WaitOne, який успадковується від класу WaitHandle, та звільняють семафор, викликаючи метод Release [12].

Лічильник семафора зменшується щоразу, коли потік входить до семафора, та збільшується, коли потік звільняє семафор. Коли лічильник дорівнює нулю, наступні запити блокуються, доки інші потоки не звільнять семафор. Коли всі потоки звільнили семафор, лічильник має максимальне значення, вказане під час створення семафора.

Потік може входити до семафора кілька разів, багаторазово викликаючи метод WaitOne. Щоб звільнити деякі або всі ці записи, потік може кілька разів викликати перевантаження методу Release() без параметрів або може викликати перевантаження методу Release(Int32), яке визначає кількість записів, які потрібно звільнити.

Клас Semaphore не застосовує ідентифікацію потоків до викликів WaitOne або Release. Програміст зобов'язаний забезпечити, щоб потоки не звільняли семафор занадто багато разів. Наприклад, припустимо, що семафор має максимальну кількість два, і що потік A та потік B обидва входять до семафора. Якщо помилка програмування в потоці B призводить до того, що він двічі викликає Release, обидва виклики завершуються успішно. Кількість семафорів повна, і коли потік A врешті-решт викликає Release, виникає виняток SemaphoreFullException.

Семафори бувають двох типів [12]: локальні семафори та іменовані системні семафори. Якщо об'єкт Semaphore створюється за допомогою конструктора, який приймає ім'я, він пов'язується з семафором операційної системи з цим ім'ям. Іменовані системні семафори видимі в усій операційній системі і можуть використовуватися для синхронізації дій процесів. Можна створити кілька об'єктів Semaphore, які представляють один і той самий іменований системний семафор, і використовувати метод OpenExisting, щоб відкрити існуючий іменований системний семафор.

Локальний семафор існує лише у поточному процесі. Його може використовувати будь-який потік процесу, який має посилання на локальний об'єкт Semaphore. Кожен об'єкт Semaphore є окремим локальним семафором.

Приклад, що реалізує керування класичним семафором роботи двох потоків може бути зроблено виконанням наступних кроків:

1. Для реалізації прикладу з виключним семафором Semaphore додати у Windows Form програми два фонових потоки backgroundWorker5() та backgroundWorker6() у спосіб вже показаний на Рис. 7.4.
2. Запуск потоків реалізувати у обробнику кнопки на формі MyForm:

```
private: System::Void button4_Click(System::Object^ sender, System::EventArgs^ e)
{     backgroundWorker5->RunWorkerAsync();
            backgroundWorker6->RunWorkerAsync();
}
```

Як видно з обробника, перший запускається потік-5, потім – потік-6.

3. Для реалізації виключного семафора оголосити об’єкт класу Semaphore у класі форми MyForm (показано частину класу MyForm):

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
      MyForm(void)
      {InitializeComponent();
      }
      // оголошення класичного семафора
      static Semaphore^ semaphore1 = gcnew Semaphore(1,2);
///// продовження коду класу MyForm
};
```

4. Подвійним натиском на компоненти потоків у MyForm створити потокові функції backgroundWorker5() та backgroundWorker5(), в яких реалізувати блокування потоку класичним семафором за допомогою метода WaitOne() і його вивільнення методом Release().

Як видно коду прикладу, у потоках реалізується завантаження зображень літаків "An225.png" і "A340.png" (файли необхідно скопіювати у теку проєкту). Потік, який запускається першим (дивіться п. 2), викликом semaphore1->WaitOne() блокує доступ до виведення у форму і звільняє доступ викликом semaphore1->Release(). Таким чином, Ан-225 проходить шлях першим, А340 – другим (Рис. 7.7).

Проте, якщо у конструкторі класичного семафора вказати

```
static Semaphore^ semaphore1 = gcnew Semaphore(2,2);
```

відбудеться паралельне виведення обох рисунків, що свідчить про надання дозволу виведення одразу для двох потоків:

```
private: System::Void backgroundWorker5_DoWork(System::Object^ sender, System::ComponentModel::DoWorkEventArgs^ e)
{
	Graphics^ g = CreateGraphics();
	Image^ image1 = Image::FromFile("An225.png");
	semaphore1->WaitOne();
	for (int i = 0; i < 60; i++)
	{
		Rectangle r0(10 + 5 * (i - 1), 10, 450, 120); // розмір 450x120
		g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
		g->DrawImage(image1, 10 + 5 * i, 10);
		System::Threading::Thread::Sleep(5);
	}
	semaphore1->Release();
}
private: System::Void backgroundWorker6_DoWork(System::Object^ sender, System::ComponentModel::DoWorkEventArgs^ e)
{
	Graphics^ g = CreateGraphics();
	Image^ image1 = Image::FromFile("A340.png"); // розмір 410x150
	semaphore1->WaitOne();
	for (int i = 0; i < 60; i++)
	{
		Rectangle r0(10 + 5 * (i - 1), 140, 410, 150);
		g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
		g->DrawImage(image1, 10 + 5 * i, 140);
		System::Threading::Thread::Sleep(5);
	}
	semaphore1->Release();
	}
```

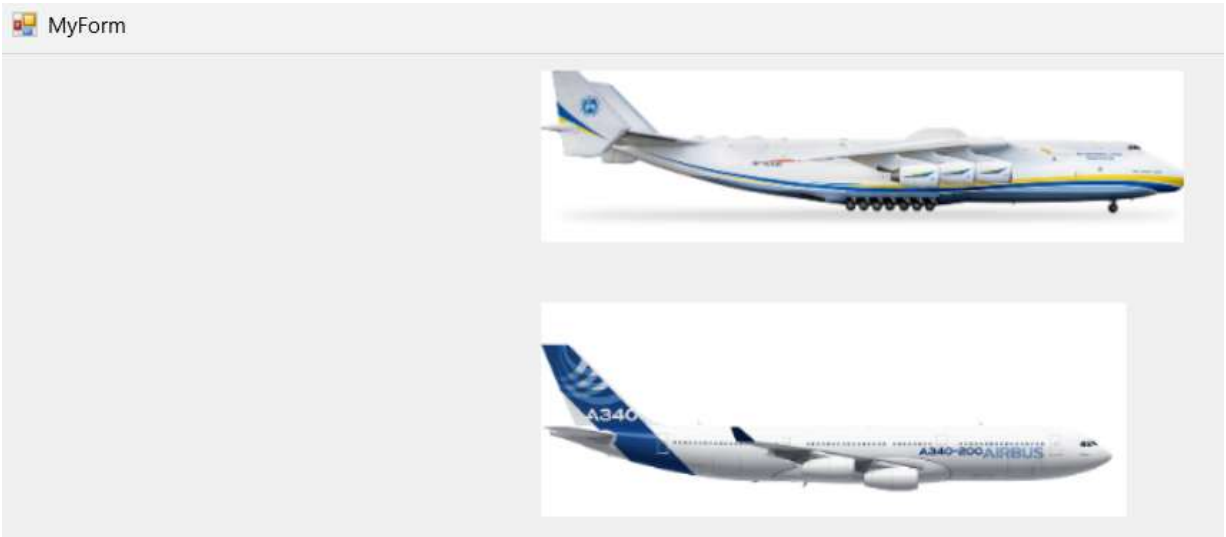


Рисунок 7.7 – Результат прикладу з об’єктом синхронізації Semaphore

7.5.6 Використання об’єкта події EventWaitHandle

Ідея використання об’єкта події не є новою і передбачає повідомлення одним потоком до другого інформації про настання певної програмної події, яка зазвичай призводить до запуску бо зупинки іншого потоку [1].

Напевно, щоб уникнути термінологічної плутанини, розробники .NET видозмінили назву об’єкта події, визначивши клас EventWaitHandle.

Клас EventWaitHandle дозволяє потокам взаємодіяти один з одним за допомогою сигналів [13]. Зазвичай один або кілька потоків блокуються на EventWaitHandle, доки нерозблокований потік не викличе метод Set(), звільняючи один або кілька заблокованих потоків. Потік може сигналізувати EventWaitHandle, а потім блокуватися на ньому, викликаючи статичний (спільний у Visual Basic) метод WaitHandle.SignalAndWait().

Поведінка EventWaitHandle, про який було повідомлено, залежить від режиму його скидання. EventWaitHandle, створений з прапорцем EventResetMode.AutoReset, автоматично скидається після сигналізації, після звільнення одного потоку, що очікує. EventWaitHandle, створений з прапорцем EventResetMode.ManualReset, залишається сигналізованим, доки не буде викликано його метод Reset().

Події автоматичного скидання надають ексклюзивний доступ до ресурсу. Якщо подія автоматичного скидання сигналізується, коли жодні потоки не очікують, вона залишається сигналізованою, доки потік не спробує її зачекати. Подія звільняє потік і негайно скидає його, блокуючи наступні потоки.

Події ручного скидання подібні до шлюзів. Коли подія не сигналізується, потоки, які її очікують, блокуються. Коли подія сигналізується, всі потоки, що очікують, звільняються, і подія залишається сигналізованою (тобто наступні очікування не блокуються), доки не буде викликано її метод Reset(). Події ручного скидання корисні, коли один потік повинен завершити дію, перш ніж інші потоки зможуть продовжити роботу [13].

Об'єкти EventWaitHandle можна використовувати зі статичними (спільними у Visual Basic) методами WaitHandle.WaitAll та WaitHandle.WaitAny.

У прикладі, що реалізує керування роботою потоків об’єктом події, потік-8 очікує подію у заблокованому стані, який задано

```
event1->WaitOne();
```

а потік-9 повідомляє методом Set() про настання дозвільної події.

Приклад може бути зроблено виконанням наступних кроків:

1. Для реалізації прикладу з об’єктом події EventWaitHandle додати у Windows Form програми два фонових потоки backgroundWorker7() та backgroundWorker8().
2. Запуск потоків реалізувати у обробнику кнопки на формі MyForm:

```
private: System::Void button6_Click(System::Object^ sender, System::EventArgs^ e)
{
	backgroundWorker7->RunWorkerAsync();
	backgroundWorker8->RunWorkerAsync();
}
```

3. Для реалізації виключного семафора оголосити об’єкт класу EventWaitHandle у класі форми MyForm (показано частину класу MyForm):

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
	MyForm(void)
	{InitializeComponent();
	}
// оголошення об’єкта події
static EventWaitHandle^ event1 =
gcnew EventWaitHandle(false, EventResetMode::AutoReset);
///// продовження коду класу MyForm
};
```

4. Подвійним натиском на компоненти потоків у MyForm створити потокові функції backgroundWorker7() та backgroundWorker8().

Як видно коду прикладу, із досягненням індексу циклу i=5, потік-7 встановлює подію, що для потоку-8 означає його розблокування (Рис. 7.8).

```
	private: System::Void backgroundWorker7_DoWork(System::Object^ sender, System::ComponentModel::DoWorkEventArgs^ e)
	{
		Graphics^ g = CreateGraphics();
		SolidBrush^ greenBrush = gcnew SolidBrush(Color::Green);
		for (int i = 0; i < 20; i++)
		{
			Rectangle r0(10,10 + 25 * (i - 1), 100, 100);
			g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
			if (i == 5) event1->Set(); // встановлення дозвільної події
			Rectangle r(10, 10 + 25 * i, 100, 100);
			g->FillRectangle(greenBrush, r);
			System::Threading::Thread::Sleep(120);
		}
	}
	private: System::Void backgroundWorker8_DoWork(System::Object^ sender, System::ComponentModel::DoWorkEventArgs^ e)
	{
		Graphics^ g = CreateGraphics();
		SolidBrush^ redBrush = gcnew SolidBrush(Color::Red);
		event1->WaitOne(); // встановлення очікування події
		for (int i = 0; i < 20; i++)
		{
```

```
                Rectangle r0(120, 10 + 15 * (i - 1), 100, 100);
                g->FillRectangle(gcnew SolidBrush(MyForm::BackColor), r0);
                Rectangle r(120, 10 + 15 * i, 100, 100);
                g->FillRectangle(redBrush, r);
                System::Threading::Thread::Sleep(230);
        }
}
```

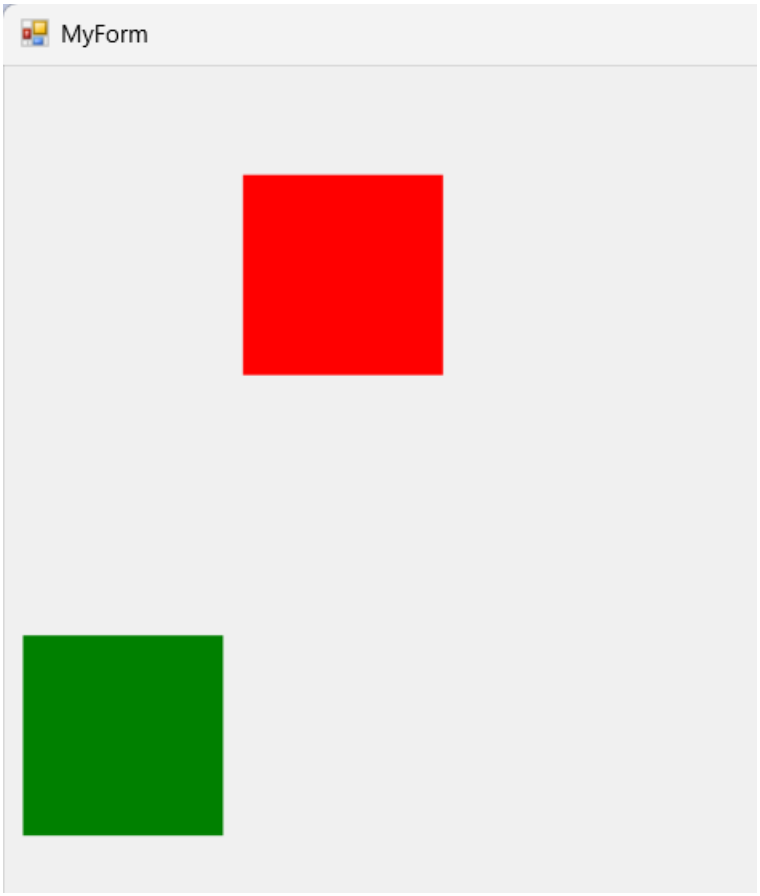


Рисунок 7.8 – Результат прикладу з об’єктом синхронізації EventWaitHandle із вертикальним рухом прямокутників в потоках

7.6 Розробка програми із функціями управління процесами ОС

7.6.1 Опис функцій управління роботою процесів

Розробка наступної програми має реалізувати деякі функції диспетчера завдань ОС Microsoft Windows. Для її реалізації необхідно додати заголовкові файли:

```
#include <windows.h>
#include <TlHelp32.h>
```

Спочатку, розглянемо функції Windows API, які можуть реалізувати необхідні властивості програми.

1. функція відкриття існуючого об’єкта процесу – забезпечує відкриття існуючого (запущеного в ОС) процесу для керування його режимами роботи:

```
HANDLE OpenProcess (DWORD dwDesiredAccess,  // прапорець доступу
                    // параметр дескриптора спадкоємності (0/1)
                    BOOL bInheritHandle,
                    DWORD dwProcessId       // ідентифікатор процесу
                   );
```

Прапорець доступу	Коментар
PROCESS_ALL_ACCESS	Усі можливі права доступу для об’єкта процесу
PROCESS_CREATE_PROCESS	Необхідно для створення процесу
PROCESS_CREATE_THREAD	Необхідно для створення потоку
PROCESS_TERMINATE	Необхідно для переривання процесу

2. функція безумовного переривання процесу та усіх його потоків – забезпечує зупинку отриманого процесу:

```
BOOL WINAPI TerminateProcess(_In_ HANDLE hProcess, _In_ UINT uExitCode );
      //    hProcess – дескриптор процесу, що має перериватися
      // uExitCode – код виходу.
```

Завершення процесу має такі результати:

– будь-які потоки, що залишилися в процесі, позначаються для завершення;

– будь-які ресурси, виділені процесом, звільняються;

– усі об'єкти ядра закриваються;

– код процесу видаляється з пам'яті;

– встановлюється код виходу процесу;

– об'єкт процесу сигналізується.

3. Функція отримання списку процесів, модулів, потоків

```
      HANDLE WINAPI CreateToolhelp32Snapshot
      (_In_ DWORD dwFlags ,_In_ DWORD th32ProcessID);
      //    dwFlags    визначає    тип    списку    –    TH32CS_SNAPPROCESS,
TH32CS_SNAPMODULE,
      // TH32CS_SNAPTHREAD.
      // dwProcessId задає ідентифікатор процесу, для якого створюється спи-
сок // (0 – для поточного процесу).
      PROCESSENTRY32 – вхід до списку процесів, що функціонують в системі.
```

4. Process32First() – функція отримання інформації про перший процес у списку;
5. Process32Next() – функція отримання інформації про наступний процес у списку.

7.6.2 Порядок розробки програми

1. Побудувати стандартний C++/CLI-проект з формою MyForm і необхідними налаштуваннями.

2. Додати у форму MyForm об’єкти списку ListBox та кнопки «Stop Process», як показано на Рис. 7.9.

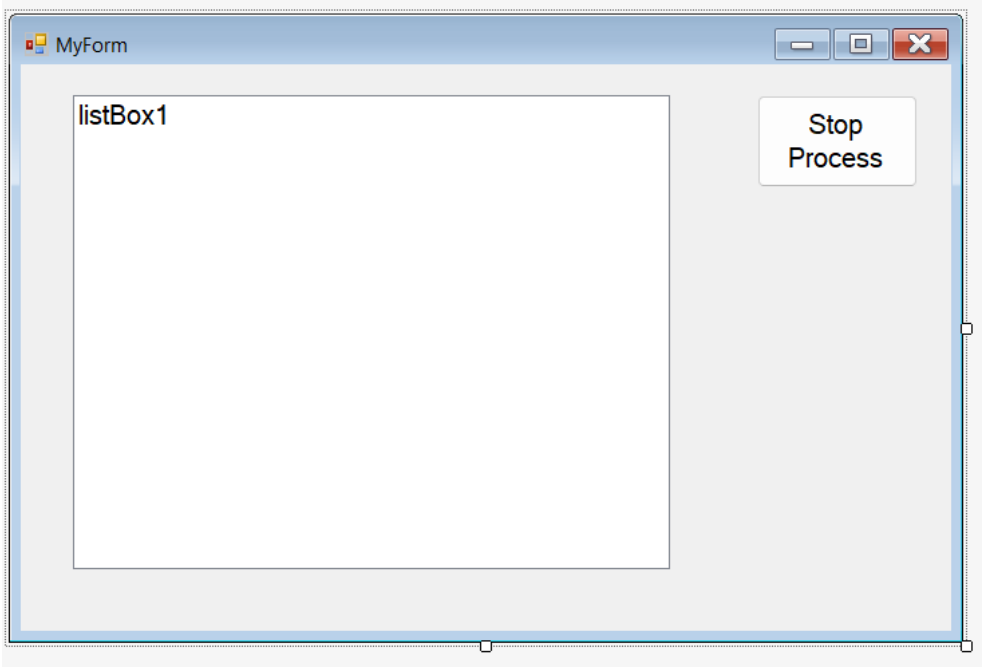


Рисунок 7.9 – Вигляд форми для програми керування процесами

3. У обробнику події завантаження форми Load реалізувати заповнення списку ListBox1 переліком активних процесів (програм) операційної системи:

```
private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e)
       {
       HANDLE snapshot
              = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
       PROCESSENTRY32 process;
       ZeroMemory(&process, sizeof(PROCESSENTRY32));
       process.dwSize = sizeof(PROCESSENTRY32);
       if (Process32FirstW(snapshot, &process))
       {      do
              { String^ processName = gcnew String(process.szExeFile);
                listBox1->Items->Add(processName);
              } while (Process32NextW(snapshot, &process));
       }
       CloseHandle(snapshot);
    }
```

4. У обробнику кнопки “Stop Process” реалізувати зупинку та видалення обраного (якщо такий є) у списку ListBox процеса шляхом пошуку у переліку активних процесів такого, що відповідає обраному імені, вибором ідентифікатора процеса, і, якщо, ідентифікатор є ненульовим – зупинка і видалення процеса застосуванням функції TerminateProcess():

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
	String^ processName = listBox1->SelectedItem->ToString();
	DWORD pid = 0;
	if (processName)
	{
		HANDLE snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
		PROCESSENTRY32 process;
		ZeroMemory(&process, sizeof(PROCESSENTRY32));
		process.dwSize = sizeof(PROCESSENTRY32);
		if (Process32First(snapshot, &process))
		{
			do {
				if (gcnew String(process.szExeFile) == processName)
				{
				 pid = process.th32ProcessID; // отримання ID процесу
				 break;
				}
			} while (Process32Next(snapshot, &process));
		}
		CloseHandle(snapshot);
		if (pid != 0)
		{
			HANDLE p = OpenProcess(PROCESS_TERMINATE, false, pid);
			TerminateProcess(p, 0);
			MessageBox::Show("Process "+ processName+ " is stopped",
				"Process Stop", MessageBoxButtons::OK,
				MessageBoxIcon::Information);
		}
		else MessageBox::Show("Process"+ processName+" isn't found",
			"Process Not Stop", MessageBoxButtons::OK,
			MessageBoxIcon::Error);
	}
}
```

Результати роботи програми показано на Рис. 7.10.

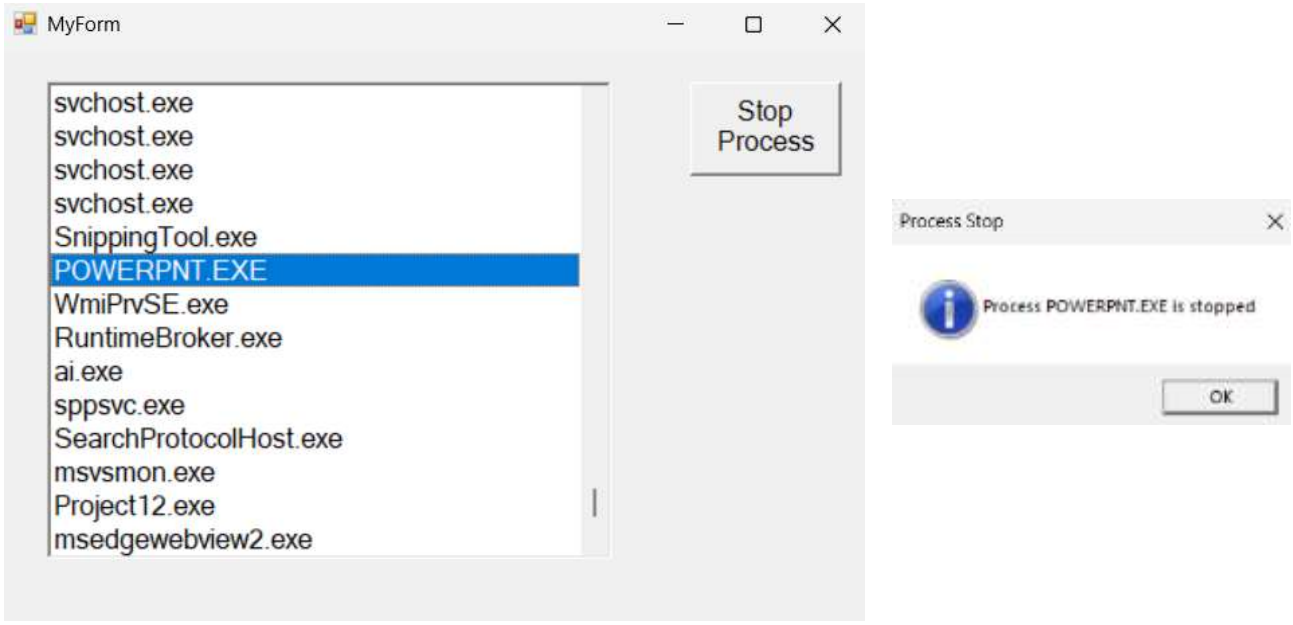


Рисунок 7.10 – Вибір процесу та його зупинка

7.7 Контрольні завдання

1. Пояснити терміни «багатозадачність» та «багатопотоковість», їх спільні та відмінні риси.
2. Пояснити порядок створення та використання робочих потоків.
3. Пояснити можливість встановлення значення пріоритету потоку та процесу.
4. Пояснити термін “синхронізація”.
5. Пояснити відмінності об’єктів синхронізації різних типів.
6. Реалізувати програму із звичайним та виключним семафорами, пояснити порядок її функціонування.
7. Реалізувати програму із об’єктом події пояснити порядок її функціонування.
8. Реалізувати програму із виведенням у форму списку запущених в ОС процесів та функцією зупинки процесів.

8 ВИКОРИСТАННЯ ГРАФІЧНОЇ БІБЛІОТЕКИ OPENGL

8.1 Загальні особливості OpenGL

Мабуть, будь-якому користувачу персональних комп'ютерів, що не обмежується Microsoft Word та Excel, а ще й захоплюється комп'ютерними іграми, відомі такі словосполучення, як OpenGL та Direct3D. Саме OpenGL та Direct3D певний час були основними технологіями розробки ігрового програмного забезпечення в галузі тривимірної графіки.

Більш того, саме комп'ютерні ігри є найбільшою рушійною силою розвитку апаратного забезпечення персональних комп'ютерів, бо саме комп'ютерні ігри, а також системи моделювання робочих міст оператора (наприклад авіаційні, або інші тренажери) є найбільш вибагливими з точки зору забезпечення швидкості обчислень, обробки та виведення аудіо-візуальної інформації. Збільшення складності та якості зображень тягне за собою збільшення потреб та апаратного забезпечення, що у свою чергу надає нові і нові можливості. Сегмент ринку комп'ютерних технологій, пов'язаний із комп'ютерними іграми залишається одним з найбільш прибуткових і динамічних.

З точки зору користувача в іграх немає різниці, яка використовується технологія програмування, важливим є оптимальне налаштування відеопристроїв комп'ютера і апаратна підтримка тієї чи іншої графічної технології. Зазначимо, що, на відміну від OpenGL, Direct3D є частиною більш загальної технології DirectX, що включає не тільки роботу із графічною інформацією, а й забезпечення інших мультимедійних засобів, у тому числі пов'язаних із введенням та виведення звукової та іншої інформації. Але, якщо DirectX – витвір компанії Microsoft і функціонує лише у Microsoft Windows, OpenGL підтримується і іншими операційними системами.

OpenGL є графічною бібліотекою, що надає програмний інтерфейс забезпечення машинної графіки (близько 250 команд), які використовуються для визначення об'єктів і операцій, необхідних під час створення тривимірного графічного програмного забезпечення.

OpenGL розроблено у як низькорівневий апаратно-незалежний інтерфейс, що реалізується рядом різних апаратних платформ. З цієї причини у OpenGL не включені команди для роботи з вікнами або забезпечення введення користувача. Також бібліотека не містить команди високого рівня для опису моделей тривимірних об'єктів. Навпаки, необхідна модель створюється з обмеженого набору геометричних примітивів: точок, ліній, багатокутників.

На базі OpenGL пізніше були створені більш складні бібліотеки:

- бібліотека утиліт OpenGL (GLU – OpenGL Utility Library), яка включає широкі можливості моделювання (поверхні другого порядку та сплайни);
- бібліотека FSG (Fahrenheit Scene Graph).

До основних можливостей OpenGL належить [5]:

- відображення каркасних моделей;
- підтримка ілюзії перспективи та глибини простору, атмосферних ефектів (туман, сніг);
- підтримка згладжування вершин сцен;
- підтримка сцен з плоским зафарбовуванням без використанням освітлення;
- підтримка сцен з освітленням та рівномірним зафарбовуванням;
- накладення тіней та текстур;
- моделювання розмитості відображення рухомих об’єктів (motion blur);
- підтримка ефектів глибини різко зображеного простору.

Основними графічними операціями OpenGL є:

- створення форм з графічних примітивів, і таким чином, створення математичних описів об’єктів (примітивами є лінії, багатокутники, зображення);
- упорядкування об’єктів у тривимірному просторі та вибір бажаної точки розташування камери для перегляду скомпонованої сцени;
- обчислення кольорів усіх об’єктів, де кольори у явний вигляд задаються програмою, визначаються, виходячи з умов освітлення, отримуються шляхом накладання текстур або комбінацією трьох зазначених дій;
- перетворення математичного опису об’єктів та пов’язаної з ними інформації щодо кольору об’єктів у пікселі на екрані (процес, що називають растеризацією).

8.2 Конвеєр візуалізації OpenGL

Оскільки графічна система OpenGL забезпечує вирішення великої кількості завдань, структура програми, що використовує OpenGL, може бути досить складною для розуміння. Насправді, основні завдання програми полягають у тому, щоб ініціювати певні стани, що забезпечують керування візуалізацією та визнати чи, які саме об’єкти мають бути візуалізовані.

Щоб надалі не заплутувати читача, оголосимо певні визначення OpenGL згідно з [6].

Візуалізацією є процес, під час якого комп’ютер створює зображення з моделей. У свою чергу, моделі або об’єкти створюються з графічних примітивів – точок, ліній та багатокутників, що визначаються їх вершинами.

Кінцеве візуалізоване зображення складається з пікселів, що виводяться на екран. Піксель є найменшим видимим елементом, що може створюватися апаратними засобами відображення.

Інформація про пікселі (наприклад, колір) організовується у вигляді бітових площин. Бітова площина є областю пам'яті, що містить один біт інформації для кожного пікселя на екрані (наприклад, із кольором пікселя). Бітові площини організуються, у свою чергу у буфер кадру, що містить усю інформацію, необхідну дисплею для того, щоб забезпечити керування кольором та яскравістю усіх пікселів на екрані.

З програмної точки зору, OpenGL є набором команд, що визначають координати об'єктів, забезпечують встановлення параметрів та їх перетворення. У будь-якому випадку команди OpenGL мають однаковий та визначений порядок виконання операцій обробки, що називається конвеєром візуалізації OpenGL. Цей порядок є практично повною структурою OpenGL і зображений на рис. 8.1.

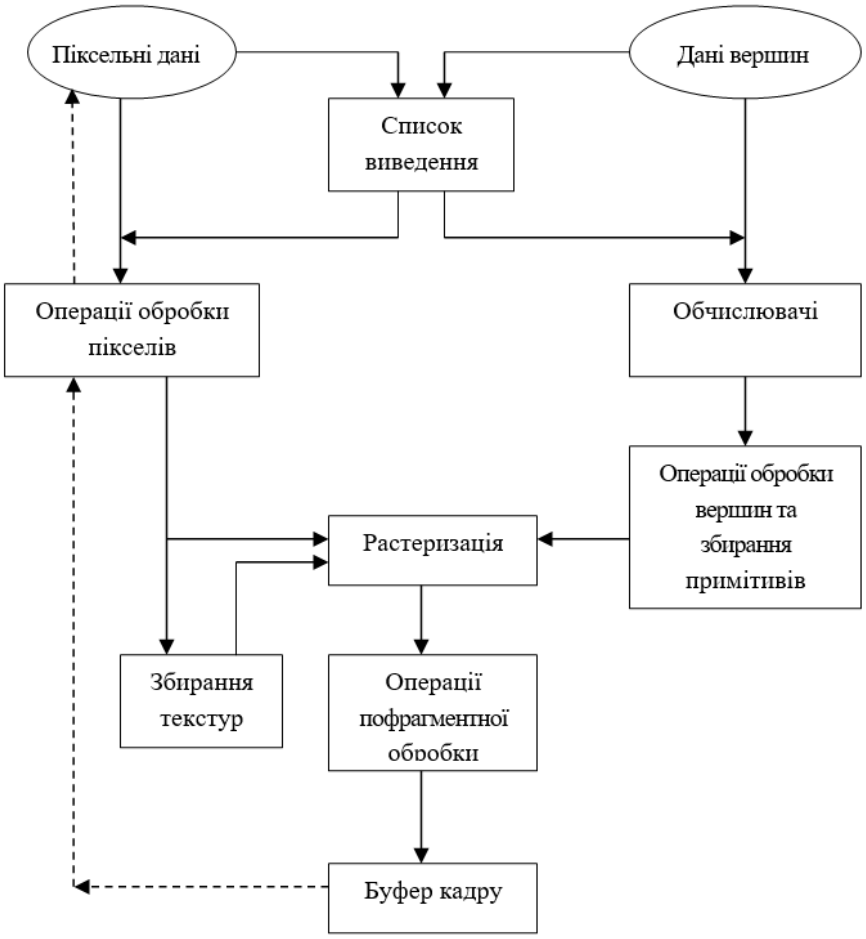


Рисунок 8.1 – Схема конвеєру візуалізації OpenGL

Тепер стисло розшифруємо окремі блоки цієї схеми [5].

1. Будь-які дані OpenGL можуть зберігатися у списках виведення для поточного або подальшого використання.

2. Усі графічні примітиви описуються вершинами. Параметричні криві та поверхні можуть спочатку описуватися контрольними точками та поліноміальними функціями, що називаються базовими функціями. Обчислювачі надають методи отримання вершин, що використовуються для подання поверхні за контрольними точками. Таким методом є поліноміальне відображення, що може формувати нормаль до поверхні, координати текстури, тексту та значення просторових координат за контрольними точками.

3. Наступною операцією для вершин є стадія операцій обробки вершин, що перетворює вершини у примітиви. Певні типи даних вершин (просторові координати) перетворюються у матриці чисел із плаваючою точкою розміром 4x4. Просторові координати проектуються з положення у тривимірному просторі у положення на екрані. Якщо програма використовує текстури, на цій стадії можуть генеруватися та перетворюватися координати текстур. Якщо використовується освітлення, також використовується обчислення параметрів освітлення, нормалі до поверхонь, положення джерел освітлення, властивості матеріалу та інша інформація освітлення.

4. Основною частиною збирання примітивів є операція відсікання, що полягає у вилученні тих частин зображення, що виходять за межі напівпростору, визначеного певною площиною. Під час відсікання точок, вершини можуть пропускатися або відкидатися; під час відсікання ліній або багатокутників можуть додаватися додаткові вершини у залежності від того, як відсікається лінія або багатокутник. Інколи збирання примітивів супроводжується перспективним поділом, що забезпечує зменшення віддалених геометричних об'єктів. Результатом етапу є закінчені геометричні примітиви, що складаються з перетворених та відсічених вершин та пов'язаних з ними значеннями кольору, глибини та інколи координати текстур.

5. Піксельні дані крокують своїм власним шляхом: з визначених масивів пам'яті пікселі спочатку розпаковуються у компоненти певного формату. Потім дані масштабуються, змішуються, оброблюються за допомогою елементів відображення. Опісля цього, результати фіксуються та записуються в область пам'яті, виділену під текстури, або передаються на стадію растеризації. Якщо піксельні дані зчитуються з буфера кадру, виконуються операції передавання пікселя (масштабування, зміщення, відображення та фіксація). Насамкінець, отримані результати запаковуються у відповідний формат та повертаються у певний формат системної пам'яті.

6. За допомогою операцій збирання текстур OpenGL- програми можуть накладати зображення на геометричні об’єкти, що робить їх більш реалістичними. Якщо програма використовує декілька зображень текстури, можна використовувати текстурні об’єкти та здійснювати перехід від одного текстурного об’єкта до іншого. Можна також здійснювати впорядкування текстур.

7. Растеризацією є перетворення геометричних або піксельних даних у фрагменти. Кожен квадратний фрагмент відповідає пікселю у буфері кадру. Штрихування ліній та багатокутників, ширина ліній, розмір точок, модель замальовування та обчислення покриття, необхідні для підтримки згладжування, враховуються як вершини, що з’єднуються у лінії, або як внутрішні пікселі, розраховані для зарисованого багатокутника. Значення кольору та глибини призначаються для кожного квадратного фрагмента.

8. Перед фактичним збереженням у буфері кадру виконується ряд операцій. Результатом операцій є змінення фрагментів або навіть їх відкидання. Усі операції можна увімкнути або вимкнути. Першою операцію є накладення текстур. Вона полягає у тому, що тексель (елемент текстури) генерується у пам’яті текстур для кожного фрагмента і застосовується для конкретного фрагмента. Після цього можуть застосовуватися операції обчислення туману, що супроводжуються тестом ножиць, альфа-тестом, тестом трафарету, тестом буферу глибини. Також можуть виконуватися операції змішування кольорів, псевдозмішування (розмиття) кольорів для передавання напівтонів, логічної обробки та маскування за допомогою бітової маски. Нарешті, повністю оброблений фрагмент виводиться у відповідний буфер, де він перетворюється у піксель і досягає свого кінцевого розташування.

Буфер кадру відіграє особливу роль. Дані буфера кадру можуть записуватися та зчитуватися функціями OpenGL, однак його настроювання відбувається за допомогою операційної системи, у якій використовується OpenGL.

8.3 Взаємодія Windows та OpenGL у .NET -програмах

Хоча команди OpenGL можуть виконуватися у різних операційних системах, написання програм OpenGL у спосіб, наприклад, зазначений у [4] для проектів Visual C++ консольного типу (аналогічний організації DOS-програм), не завжди є прийнятним. Організація OpenGL-програм для операційної системи Microsoft Windows має свої особливості.

Як вже зазначалося у розділі 2, Windows-програми для виведення інформації графічного або текстового вмісту використовують контексти пристроїв.

Інтерфейс графічних пристроїв Windows GDI забезпечує керування контекстами цих пристроїв. Технологія OpenGL не співпрацює зі стандартними контекстами пристроїв Windows, натомість використовує так званий контекст візуалізації (rendering context) [3].

Під час роботи з контекстом пристрою для вибору нового інструмента відображення (пера або пензля) використовується GDI-функція *SelectObject()*. Обраний інструмент залишається активним до вибору наступного інструмента.

За аналогією з контекстом пристрою, контекст візуалізації забезпечує обробку графічного стану. Технологія OpenGL використовує контекст візуалізації для обробки інформації, необхідної для коректного екранного подання сцени у заданому вікні. У OpenGL має бути присутнім спеціальний механізм взаємозв'язку вихідної інформації з контекстом пристроїв Windows. Шляхом розміщення визначених ділянок інформації у структурі контексту візуалізації, OpenGL дозволяє оновлювати графічний стан вікна під час роботи з операційною системою Windows. Ці взаємини описуються рисунком 8.2.

Рисунок 8.2 – Відношення у системі контекст пристрою-контекст візуалізації

Контексти візуалізації є багатопотоковими, тобто один спільний контекст візуалізації може використовуватися декількома потоками. У структурі програми може використовуватися декілька контекстів візуалізації, з яких обирається необхідний. Однак один потік має функціонувати в межах активного контексту візуалізації.

OpenGL-команди не потребують дескрипторів або покажчиків на контексти візуалізації незалежно від того, який контекст візуалізації є активним. Контекст візуалізації є практично однаковим для різних OpenGL-команд.

У процесі роботи з OpenGL контекст візуалізації спочатку створюється, потім ініціюється, обирається поточним, використовується, відключається та вилучається. Така послідовність не означає необхідність її виконання під час кожного оновлення графічної сцени. У реальних OpenGL-програмах створений та ініційований контекст візуалізації надалі багаторазово використовується у роботі поточної програми. При цьому активізація та відключення контексту виконуються багаторазово, а вилучення здійснюється один раз – після закінчення роботи програми.

Використання контекстів візуалізації забезпечується такими WinAPI-функціями:

```
HGLRC wglCreateContext(HDC  hdc );                  // створення контексту візуалізації
BOOL wglDeleteContext(HGLRC  hglrc );          // вилучення контексту візуалізації
BOOL wglMakeCurrent(HDC hdc, HGLRC  hglrc );  // активізація контексту візуалізації
```

У наведених функціях *hdc* – дескриптор контексту GDI, *hglrc* – дескриптор контексту візуалізації OpenGL.

8.4 Структура PIXELFORMATDESCRIPTOR

В основі контекстного відображення знаходиться поняття формату пікселя. Формат пікселя Windows не завжди узгоджується з OpenGL. У процесі ініціалізації OpenGL ці формати необхідно узгодити. Інформація щодо взаємодії OpenGL та конкретного системного пристрою Windows міститься у структурі PIXELFORMATDESCRIPTOR, що описується нижче:

```
typedef struct tagPIXELFORMATDESCRIPTOR {
    WORD  nSize;            // визначає розмір структури, має дорівнювати
                            // sizeof (PIXELFORMATDESCRIPTOR)
    WORD  nVersion;         // версія структури (має бути 1.0)
    DWORD dwFlags;          //  набір бітів, що характеризують властивості буфера пікселів
    BYTE  iPixelType;       // тип піксельних даних
    BYTE  cColorBits;       // кількість бітових площин у кожному буфері кольору
    BYTE  cRedBits;         // кількість бітових площин червоного у кожному буфері RGBA
    BYTE  cRedShift;        // зсув від початку кількості бітових площин червоного у кожному
                            // буфері RGBA
    BYTE  cGreenBits;       // кількість бітових площин зеленого у кожному буфері RGBA
    BYTE  cGreenShift;      // зсув від початку кількості бітових площин зеленого у кожному
                            // буфері RGBA
    BYTE  cBlueBits;        // кількість бітових площин синього у кожному буфері RGBA
    BYTE  cBlueShift;       // зсув від початку кількості бітових площин синього у кожному
                            // буфері RGBA
    BYTE  cAlphaBits;       // кількість бітових площин альфа у кожному буфері RGBA
    BYTE  cAlphaShift;      // зсув від початку кількості бітових площин альфа у кожному буфері RGBA
```

```
    BYTE  cAccumBits;        // загальна кількість бітових площин у буфері акумулятора
    BYTE  cAccumRedBits;     // загальна кількість бітових площин червоного у буфері акумулятора
    BYTE  cAccumGreenBits;   // загальна кількість бітових площин зеленого у буфері акумулятора
    BYTE  cAccumBlueBits;    // загальна кількість бітових площин синього у буфері акумулятора
    BYTE  cAccumAlphaBits;   // загальна кількість бітових площин альфа у буфері акумулятора
    BYTE  cDepthBits;        // розмір буфера глибини (вісь Z)
    BYTE  cStencilBits;      // розмір буфера трафарету
    BYTE  cAuxBuffers;       // кількість допоміжних буферів (не підтримується)
    BYTE  iLayerType;        // ігнорується
    BYTE  bReserved;         // кількість площин переднього та заднього плану
    DWORD dwLayerMask;       // ігнорується
    DWORD dwVisibleMask;     // індекс або колір прозорості нижньої площини
    DWORD dwDamageMask;      // ігнорується
} PIXELFORMATDESCRIPTOR;
```

Структура PIXELFORMATDESCRIPTOR використовується для запису значень, до яких програма OpenGL має налаштуватися. До таких параметрів належить подвійна буферизація (використання двох відеобуферів для підтримки мультиплікації без мерехтіння) та пікселі типу RGBA (червоний, зелений, синій та альфа). Після заповнення, структура передається GDI-функції і дозволяє використовувати можливості поточної системи. Контекст візуалізації створюється функцією *wglCreateContext()*.

Окремі елементи структури PIXELFORMATDESCRIPTOR мають значну кількість значень. Значення поля *dwFlags* наводяться у таблиці 8.1.

Таблиця 8.1 – Значення поля *dwFlags* структури PIXELFORMATDESCRIPTOR

Значення поля	Коментар
PFD_DRAW_TO_WINDOW	дозволено рисування у вікні або на поверхні пристрою
PFD_DRAW_TO_BITMAP	дозволено рисування у бітовий масив у пам’яті
PFD_SUPPORT_GDI	підтримується рисування GDI (не може використовуватися одночасно із PFD_DOUBLEBUFFER)
PFD_SUPPORT_OPENGL	підтримується рисування OpenGL
PFD_GENERIC_ACCELERATED	підтримується драйверами із прискоренням
PFD_GENERIC_FORMAT	формат пікселів підтримується програмною реалізацією GDI

Продовження таблиці 8.1

Значення поля	Коментар
PFD_NEED_PALETTE	для керування палітрою використовуються пікселі RGBA
PFD_NEED_SYSTEM_PALETTE	використовується системою із спеціальним обладнанням OpenGL із однією палітрою пристрою
PFD_DOUBLEBUFFER	підтримується режим подвійної буферизації
PFD_STEREO	стереоскопічний буфер (не реалізований)
PFD_SWAP_LAYER_BUFFERS	показує, чи може пристрій міняти місцями окремі рівні з форматами пікселів

Значення поля *iPixelType* наводяться у таблиці 8.2.

Таблиця 8.2 – Типові значення поля *iPixelType* структури PIXELFORMATDESCRIPTOR

Значення поля	Коментар
PFD_TYPE_RGBA	колір кожного пікселя визначається чотирма значеннями: червоним, зеленим, синім та альфа
PFD_TYPE_COLORINDEX	колір кожного пікселя визначається індексом у спеці-альній таблиці кольорів

8.5 Особливості організації OpenGL-орієнтованої .NET-програми

У літературі з OpenGL звичайно створення MFC-програм звичайно базується на проектах, створюваних за допомогою MFC Application Wizard, що в основному передбачає роботу із класами вигляду (view). В [1] побудова OpenGL-програми ґрунтується на структурі проекту типу Win32 Application. Наше завдання – розлянути можливість створення OpenGL-програми на основі проекту .NET (C++/CLI).

Розглянемо порядок внесення змін у початковий проект .NET (C++/CLI), послідовність побудови якого описана в Розділі 2:

1. До класу MyForm проекту додати дескриптори віконної форми та контексту візуалізації OpenGL:

```
public ref class MyForm : public System::Windows::Forms::Form
{
public:
        MyForm(void)
```

```
        {
            InitializeComponent();
        }
        HDC m_hDC;      // дескриптор вікна
HGLRC m_hrc; // дескристор контексту візуалізації OpenGL
////////
};
```

2. Реалізувати подію Shown() форми MyForm у наступний спосіб:

```
private: System::Void MyForm_Shown(System::Object^ sender, System::EventArgs^ e)
{
        PIXELFORMATDESCRIPTOR pfd;
        pfd.dwFlags = PFD_DOUBLEBUFFER;
        m_hDC = GetDC((HWND)this->Handle.ToPointer());
        int nPixelFormat = ChoosePixelFormat(m_hDC, &pfd);
        SetPixelFormat(m_hDC, nPixelFormat, &pfd);
        m_hrc = wglCreateContext(m_hDC);
        wglMakeCurrent(m_hDC, m_hrc);
        this->MyForm_SizeChanged(this, e);
}
```

3. Реалізувати подію SizeChanged() форми MyForm:

```
private: System::Void MyForm_SizeChanged(System::Object^ sender, System::EventArgs^ e)
{
        int cx = MyForm::Size.Width;
        int cy = MyForm::Size.Height;
        BOOL bResult = wglMakeCurrent(m_hDC, m_hrc);
        GLdouble gldAspect = (GLdouble)cx / (GLdouble)cy;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(30.0, gldAspect, 1.0, 10.0);
glViewport(0, 0, cx, cy);
        wglMakeCurrent(NULL, NULL);
}
```

4. Реалізувати подію Paint() форми MyForm():

```
private: System::Void MyForm_Paint(System::Object^ sender, System::Windows::Forms::PaintEventArgs^ e)
        {
            wglMakeCurrent(m_hDC, m_hrc);
            GLInit();
            OnOpenGL();
            SwapBuffers(m_hDC);
            wglMakeCurrent(NULL, NULL);
        }
```

5. У форму MyForm додати елементи HScrollBar і VScrollBar та реалізувати їх обробники в наступний спосіб:

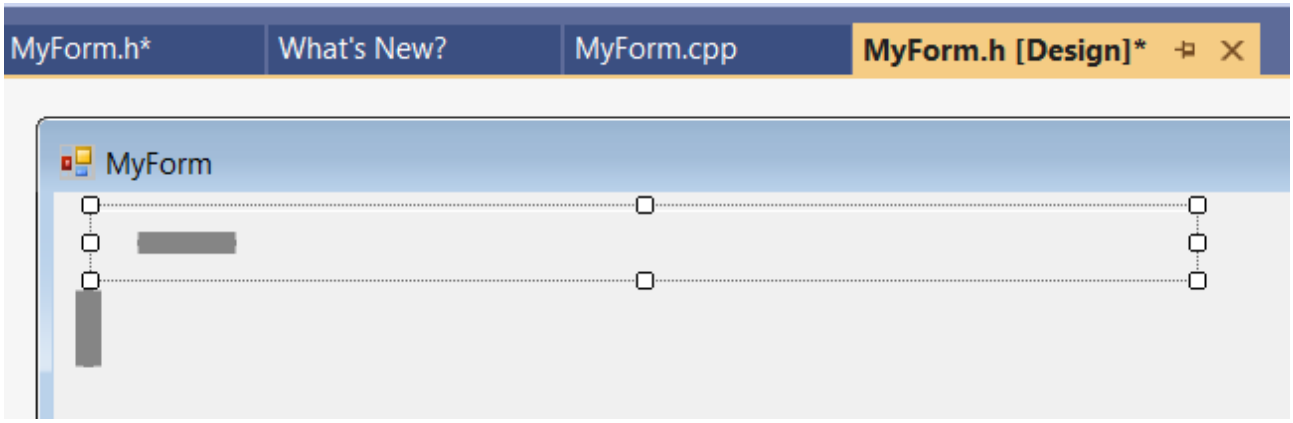


Рисунок 8.3 – Додавання елементів прокрутки (для перегляду 3D-об’єктів)

```
private: System::Void hScrollBar1_Scroll(System::Object^ sender,
System::Windows::Forms::ScrollEventArgs^ e)
{ hspos = e->NewValue;
MyForm::Invalidate();
//this->MyForm_Paint(this, k);
}
private: System::Void vScrollBar1_Scroll(System::Object^ sender,
System::Windows::Forms::ScrollEventArgs^ e)
{ vspos = e->NewValue;
MyForm::Invalidate();
//this->MyForm_Paint(this, k);
}
```

6. Визначити глобально (після основного коду проєкту за поза його фігурними дужками – внизу файла MyForm.h) функцію ініціалазації OpenGL-програми:

```
void GLInit(void)
{
GLdouble marengo[3] = { 1.0, 1.0, 0.0 };
glClearColor(1.0, 1.0, 1.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -5.0);
glColor3dv(marengo);
glScalef(1.0, 1.0, 1.0);
}
```

7. Також визначити глобально функцію формування графічної сцени OpenGL-програми:

```
void OnOpenGL(void)
```

```
{
                glRotated(360.0 * hspos / 100, 0, 1, 0);
                glRotated(360.0 * vspos / 100, 1, 0, 0);
                glColor3f(0.0, 0.0, 1.0); //  синій колір
                gluCylinder(gluNewQuadric(), 1.5, 1.5, 10.0, 10, 10); // циліндр
}
```

8. Визначити глобальні змінні OpenGL-програми, вставивши їх на початок файлу MyForm.h:

```
#pragma comment(lib,"user32.lib")
#pragma comment(lib,"gdi32.lib")
#pragma comment(lib,"glu32.lib")
#pragma comment(lib,"opengl32.lib")
#pragma comment (lib,"legacy_stdio_definitions.lib")

#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>
```

В результаті виконання усіх перелічених кроків користувач має отримати в формі MyForm зображення повернутого до користувача циліндра (визначеного у п.8.), як це показує Рис. 8.4.

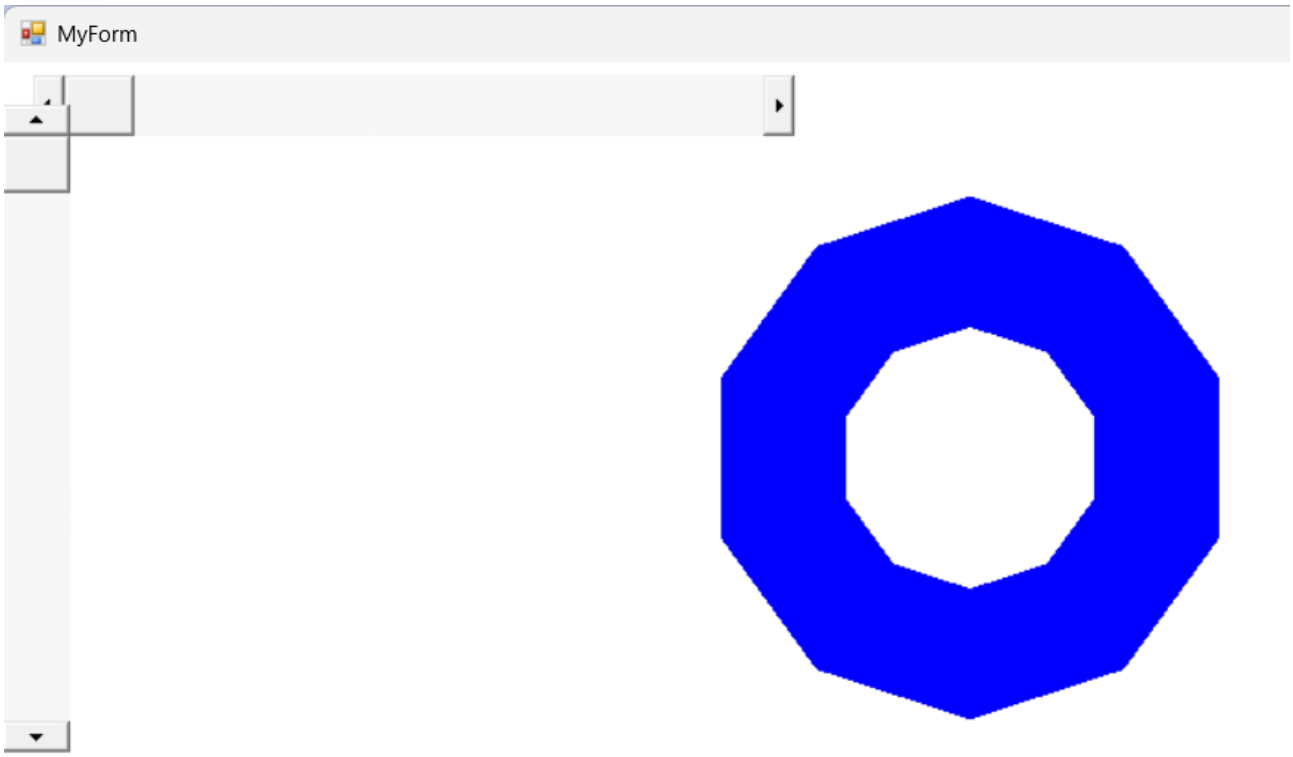


Рисунок 8.4 – Результат виконання першої OpenGL-програми

Розроблена у цьому підрозділі програма може стати основою для подальших розробок. Надалі програми конструюватимуться, в основному, додаванням до цієї програми-основи, модифікуючи функцію OnOpenGL().

8.6 Основні команди відображення графічних примітивів

8.6.1 Синтаксис команд та основні типи OpenGL

Функції та процедури OpenGL, зазвичай, називають командами [5]. Команди OpenGL легко впізнати, навіть, якщо вони використовуються іншою системою або навіть мовою програмування. Вони мають спільні особливості.

Усі команди OpenGL мають префікс *gl*, а кожна назва функції починається з великої літери, наприклад *glColor()*, *glPushMatrix()*. Узагальнено команду OpenGL можна записати у такий спосіб:

rtype CommandName [1 2 3 4] [b s i f d us ui] [v] (atype arg)

у такому записі окремі частини означають таке:

– *CommandName* – ім’я OpenGL-команди, наприклад *glVertex*;
– [1 2 3 4] – кількість аргументів команди;
– [b s i f d us ui] – тип аргументів команди (значення наведені у Таблиці 8.3);
– [v] – літера, що вказує наявність покажчика на масив у аргументі команди.

Параметр *rtype* визначає тип значення, що повертається і для кожної команди вказується у явний вигляд. Параметр *atype* та аргумент *arg* визначаються типом та кількістю аргументів, відповідно.

Таблиця 8.3 – Позначення типів OpenGL та відповідні типи C++

Символ	Тип OpenGL	Тип C (C++)
b	GLbyte	char
s	GLshort	short
i	GLint	int
f	GLfloat	float
d	GLdouble	double
ub	GLbyte	byte
us	GLshort	short
ui	GLuint	Uint

Таким чином команди OpenGL є складеними записами, зовнішній вигляд яких дозволяє дізнатися про кількість та тип застосовуваних параметрів. Кожна

OpenGL-команда може визначати декілька перевантажених варіантів, наприклад, команда визначення вершин glVertex() має такі варіанти:

glVertex2d, glVertex2f, glVertex2i, glVertex2s, glVertex3d, glVertex3f, glVertex3i, glVertex3s, glVertex4d, glVertex4f, glVertex4i, glVertex4s, glVertex2dv, glVertex2fv, glVertex2iv, glVertex2sv, glVertex3dv, glVertex3fv, glVertex3iv, glVertex3sv,
glVertex4dv, glVertex4fv, glVertex4iv, glVertex4sv

Нижче наведено детальні прототипи двох прикладів зі списку:

```
void glVertex2f(float arg1, float arg2);
void glVertex3s(short arg1, short arg2, short arg3);
```

Перший приклад містить два аргументи плаваючого (float) типу, другий – три короткого цілого (int) типу.

Назви констант OpenGL записуються виключно великими літерами, наприклад *GL_COLOR_BUFFER_BIT*. Константи OpenGL-команд будуть надалі описуватися у необхідних випадках.

8.6.2 Команди відображення графічних примітивів OpenGL

Як вже зазначувалося вище, вершини графічних об’єктів OpenGL визначаються за допомогою команди *glVertex*()*. У формальний спосіб її прототипи описуються у такий спосіб:

```
void glVertex [2 3 4] [s i f d](type coord);
void glVertex [2 3 4] [s i f d] v (type coord);
```

Виклик будь-якої команди типу *glVertex*()* визначається чотирма координатами: *x*, *y*, *z* та *w*. При цьому якщо, наприклад використовується команда *glVertex2()* задаються лише координати *x* та *y*, а значення *z* та *w* є фіксованими: *z =0, w=1*. Виклик *glVertex3()* потребуватиме завдання *x*, *y*, *z* і фіксує *w=1*. Виклик *glVertex4()* задаватиме усі чотири координати. Додатково зазначимо, що використання параметра *w* забезпечує врахування впливу перспективи.

Примітивом OpenGL вважається простий графічний об’єкт (точка, лінія, багатокутник, растрове зображення), яким можна маніпулювати як єдиною дискретною сутністю [5]. Для примітивів може встановлюватися ряд параметрів, серед яких – колір.

Поточний колір графічного об’єкта разом з умовами освітлення визначає його поточний сумарний колір. Колір задається двома командами:

```
void glColor [3 4] [b s i f d][v]  (GLtype components);      // встановлення RGBA-кольору
void glIndex [3 4] [s i f d] v (GLtype index);               // індексний режим кольору
```

Зазвичай використається RGBA-режим встановлення кольору. Цей режим використовує червоне, зелене та синє значення кольорів (відповідно R, G, B) для визначення кольору пікселя на екрані. Значення інтенсивності кольорів встановлюються у діапазоні [0.0, 1.0]. Наприклад:

```
glColor3f (0.0, 0.0, 0.0);          // чорний колір
glColor3f (1.0, 0.0, 0.0);          // червоний
glColor3f (0.0, 1.0, 0.0);          // зелений
glColor3f (0.0, 0.0, 1.0);          // синій
glColor3f (1.0, 1.0, 0.0);          // жовтий
glColor3f (1.0, 1.0, 1.0);          // білий
```

Значення альфа-параметра (A) не приймає участі формуванні результуючого кольору, а визначає ступінь прозорості: 0.0 – повна прозорість, 1.0 – повна непрозорість.

Після того, як певний колір встановлено як поточний, цей колір поширюється на всі пізніше створювані примітиви – до встановлення нового.

За допомогою наборів вершин задаються примітиви або групи однотипних примітивів: точки, лінії, з'єднані лінії, замкнені лінії, трикутники різного типу – із пов'язаними вершинами або ребрами, позв'язані та непозв'язані чотири- та інші багатокутники. Для формування набору послідовність точок розміщується всередині комбінації команд *glBegin() / glEnd()*:

```
void glBegin (GLenum mode);     // початок послідовності
void glEnd ( );                 // кінець послідовності
```

Параметр *mode* встановлює графічний об'єкт, що є результатом послідовності вказаних вершин. Значення *mode* наведено у таблиці 8.4.

Таблиця 8.4 – Значення параметра *mode* команди *glBegin()*

Значення *mode*	Результат застосування
GL_POINTS	кожна вершина розглядається як окрема незалежна точка
GL_LINES	пара вершин розглядається як незалежний відрізок (перша пара – перший відрізок, друга – другий і т.д.). Остання непарна вершина ігнорується.
GL_LINE_STRIP	послідовність з одного або декількох зв'язаних відрізків. Перша вершина – початок першого відрізка; друга – кінець, одночасно – початок другого відрізка і т.д. (N-1) відрізків.

Продовження таблиці 8.4

GL_LINE_LOOP	аналогічний режиму GL_LINE_STRIP. Додатково: остання вершина замикається відрізком на першу. N відрізків.
GL_TRIANGLES	кожна трійка вершин формує незалежний трикутник. Якщо кількість вершин не є кратною 3, решта вершин (1 або 2) ігноруються.
GL_TRIANGLE_STRIP	група зв'язаних трикутників, що мають спільні грані: (1, 2, 3) – перший трикутник; (2, 3, 4) – другий трикутник і т.д. (N-2) трикутників.
GL_TRIANGLE_FAN	група зв'язаних трикутників, що формують веєр із спільною вершиною 1: (1, 2, 3) – перший трикутник; (1, 3, 4)– другий трикутник і т.д. (N-2) трикутників.
GL_QUADS	кожна четвірка вершин формує незалежний чотирикутник: (1, 2, 3, 4) – перший, (5, 6, 7, 8) – другий і т.д. Якщо кількість вершин не є кратною 4, решта вершин (1, 2, 3) ігноруються. N/4 чотирикутників.
GL_QUAD_STRIP	група зв'язаних чотирикутників, що мають спільні грані: (1, 2, 4, 3) – перший, (3, 4, 6, 5) – другий і т.д. Якщо N – непарне число, остання вершина ігнорується.
GL_POLYGON	багатокутник, що будується на послідовності точок (> 3). Багатокутник має не перетинати сам себе і має бути опуклим.

Всередині комбінації команд *glBegin() / glEnd()* окрім самих вершин встановлюються додаткові параметри: колір, вектор нормалі, координати текстури, властивості матеріалу, також виконувати виклик списків. Використання інших команд не допускається.

За замовчуванням точка відображується на екрані як окремий піксель, а відрізок лінії також має ширину 1 піксель. Ситуацію можна змінити функціями *glPointSize()* та *glLineWidth()*:

```
void glPointSize   (GLfloat size);      // size – ширина точки у пікселях
void glLineWidth (GLfloat width);   // width – ширина лінії у пікселях
```

Розглянемо приклади побудови примітивів OpenGL. Для цього будемо змінювати і доповнювати функцію OnOpenGL(), визначивши в ній для зручності можливість обертати сцену смугами проктутки та визначивши осі координат:

```
void OnOpenGL(void)
{       glRotated(360.0 * hspos / 100, 0, 1, 0); // обертання сцени навколо осі X
        glRotated(360.0 * vspos / 100, 1, 0, 0); // обертання сцени навколо осі Y
        glTranslatef(0.0, -4.0, 0.0); // зміщення сцени вниз вздовж осі Y (за потреби)
        glLineWidth(2.0);
        glBegin(GL_LINES);
        glColor3f(1.0, 0.0, 0.0);
        glVertex3f(0.0f, 0.0f, 0.0f); glVertex3f(10.0f, 0.0f, 0.0f); // вісь X
        glColor3f(0.0, 1.0, 0.0);
        glVertex3f(0.0f, 0.0f, 0.0f); glVertex3f(0.0f, 10.0f, 0.0f); // вісь Y
        glColor3f(0.0, 0.0, 1.0);
        glVertex3f(0.0f, 0.0f, 0.0f); glVertex3f(0.0f, 0.0f, 10.0f); // вісь Z
        glEnd();
        // далі будуть розміщуватись приклади примітивів
}
```

Приклад 8.1 – Визначення функції OnOpenGL() з обертанням сцени та осями

– незалежні лінії GL_LINES:

для відображення незалежних ліній в функцію OnOpenGL() пропонується додати код Прикладу 8.2, результат виконання якого показано на Рис. 8.5:

```
glLineWidth(5.0);
glBegin(GL_LINES);
glColor3f(0.0, 0.0, 1.0);
glVertex3f(5.0f, 0.0f, 0.0f); glVertex3f(0.0f, 5.0f, 0.0f);
glColor3f(1.0, 1.0, 0.0);
glVertex3f(0.0f, 0.0f, 0.0f); glVertex3f(5.0f, 5.0f, 0.0f);
        glEnd();
```

Приклад 8.2 – Код з незалежними лініями GL_LINES

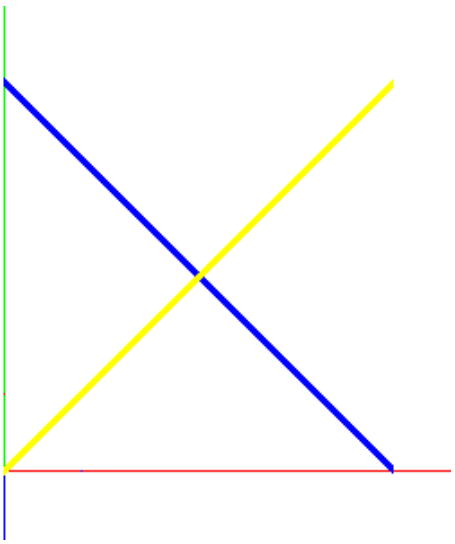


Рисунок 8.5 – Приклад з незалежними лініями GL_LINES

– зв’язані лінії GL_LINES:

для відображення зв’язаних ліній в функцію OnOpenGL() пропонується додати код прикладу 8.3, результат виконання якого показано на рис. 8.6:

```
glLineWidth(10.0);
glBegin(GL_LINE_STRIP);
glColor3f(1.0, 0.0, 1.0);
glVertex2i(0, 5); glVertex2i(5, 5);
glVertex2i(5, 0); glVertex2i(10, 0);
glEnd();
```

Приклад 8.3 – Код для зв’язаних ліній GL_LINE_STRIP

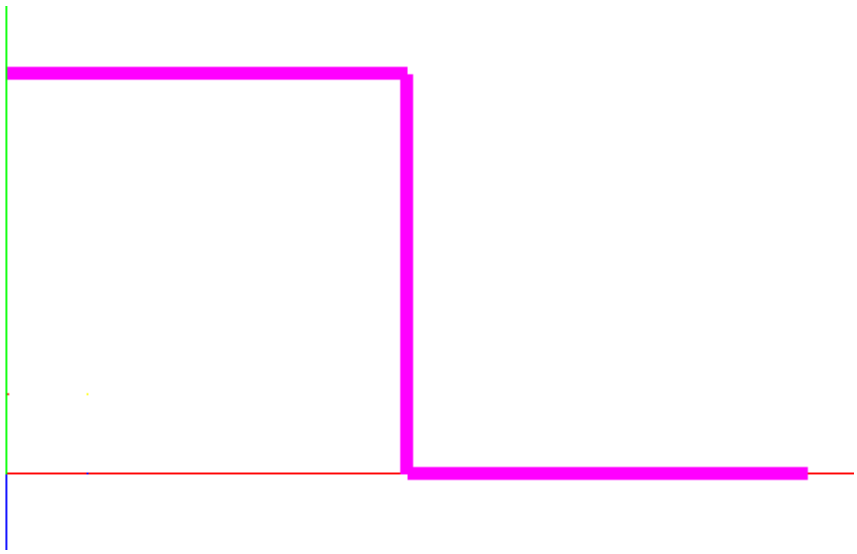


Рисунок 8.6 – Приклад для зв’язаних ліній GL_LINE_STRIP

– замкнена лінія GL_LINE_LOOP:

для демонстрації відображення замкненої лінії пропонується використати полярну систему координат, яка дозволить побудувати п’ятикутник. В функцію OnOpenGL() пропонується додати код Прикладу 8.4, результат виконання якого показано на Рис. 8.7:

```
glLineWidth(15.0);
glBegin(GL_LINE_LOOP);
glColor3f(0.0, 1.0, 0.0);
for (int i = 0; i < 5; i++)
        glVertex2f(2.5 * System::Math::Cos(0.4 * System::Math::PI * i),
                2.5 * System::Math::Sin(0.4 * System::Math::PI * i));
glEnd();
```

Приклад 8.4 – Код для замкненої лінії GL_LINE_STRIP (п’ятикутник)

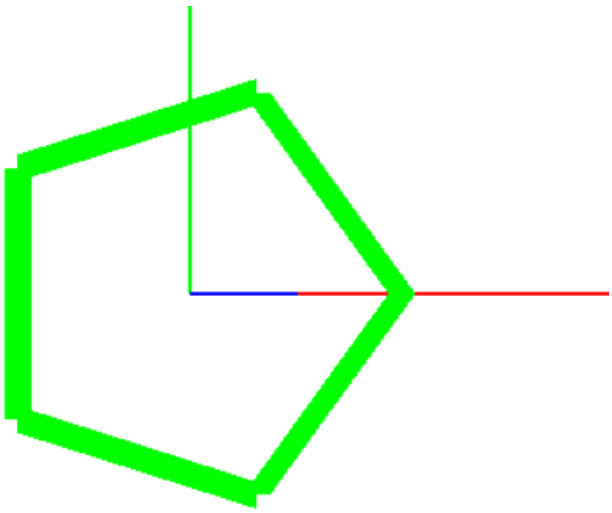


Рисунок 8.6 – Приклад для замкненої лінії GL_LINE_LOOP

– незалежні трикутники GL_TRIANGLES:

для відображення незалежних трикутників в функцію OnOpenGL() пропонується додати код Прикладу 8.5, результат виконання якого показано на Рис. 8.7:

```
glBegin(GL_TRIANGLES);
glColor3f(1.0, 1.0, 0.0);
glVertex2f(0.2, 0.0);  glVertex2f(5.0, 0.0); glVertex2f(2.5, 5.0);      // 1
glColor3f(1.0, 0.0, 1.0);
glVertex2f(-0.2, 0.0);  glVertex2f(-5.0, 0.0); glVertex2f(-2.5, -5.0);  // 2
glEnd();
```

Приклад 8.5 – Код для незалежних трикутників GL_TRIANGLES

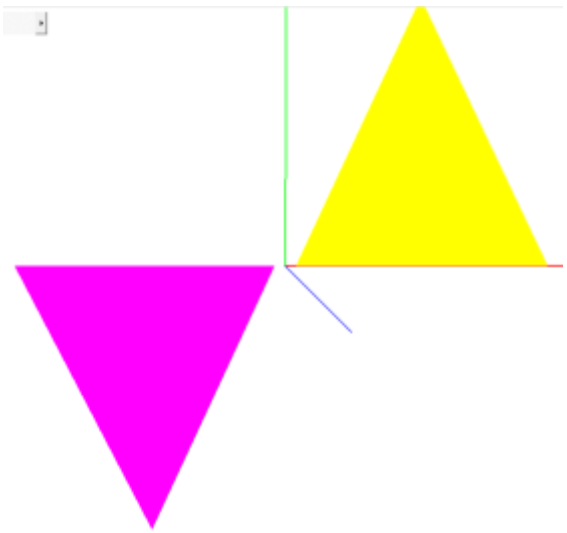


Рисунок 8.7 – Приклад для незалежних трикутників GL_TRIANGLES

– зв’язані смугою трикутники GL_TRIANGLE_STRIP:

для відображення зв’язані смугою трикутників в функцію OnOpenGL() пропонується додати код прикладу 8.6, результат виконання якого показано на рис. 8.8:

```
glBegin(GL_TRIANGLE_STRIP);
glColor3f(1.0, 0.5, 0.0);
glVertex2f(0.0, 0.0);  glVertex2f(-2.5, -5.0);
glColor3f(1.0, 0.0, 1.0);
glVertex2f(-5.0, 0.0); glVertex2f(-7.5, -5.0);
glColor3f(0.0, 1.0, 0.0);
glVertex2f(-10.0, 0.0);
glEnd();
```

Приклад 8.6 – Код для зв’язаних трикутників GL_TRIANGLE_STRIP

Рисунок 8.8 – Приклад для зв’язаних трикутників GL_TRIANGLE_STRIP

– веєр зв’язаних трикутників GL_TRIANGLE_FAN:

для відображення веєра зв’язаних смугою трикутників в функцію OnOpenGL() пропонується додати код Прикладу 8.7, результат виконання якого показано на Рис. 8.9:

```
glBegin(GL_TRIANGLE_FAN);
glColor3f(0.5, 0.5, 0.0);
glVertex2i(0, 0);
glVertex2i(-2, 3);  glVertex2i(2, 3);
glColor3f(0.0, 0.0, 1.0);
glVertex2i(4, 0);
glColor3f(1.0, 1.0, 0.0);
glVertex2i(2, -3);
glEnd();
```

Приклад 8.6 – Код для веєра зв’язаних трикутників GL_TRIANGLE_FAN

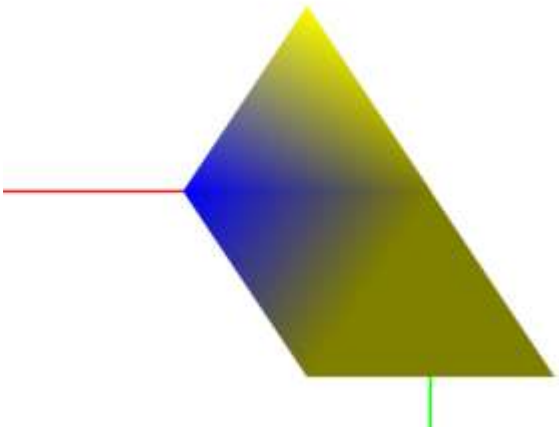


Рисунок 8.9 – Приклад для веєра зв’язаних трикутників GL_TRIANGLE_FAN

– незалежні чоторикутники GL_QUADS:

для відображення незалежних чоторикутників в функцію OnOpenGL() пропонується додати код Прикладу 8.7, результат виконання якого показано на рис. 8.10:

```
glBegin(GL_QUADS);
glColor3f(1.0, 0.0, 0.0);
        glVertex2i(0, 0); glVertex2i(0, 4);
        glVertex2i(4, 4); glVertex2i(4, 0);
glColor3f(0.0, 1.0, 0.0);
        glVertex3i(0, 0, 0); glVertex3i(0, 4, 0);
        glVertex3i(0, 4, 4); glVertex3i(0, 0, 4);
glEnd();
```

Приклад 8.7 – Код для незалежних чотирикутніків GL_QUADS

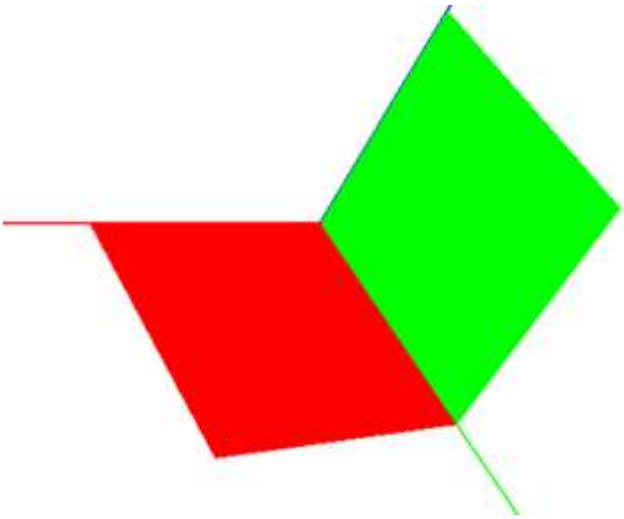


Рисунок 8.10 – Приклад для незалежних чотирикутніків GL_QUADS

– зв’язані чоторикутники GL_QUAD_STRIP:

для відображення зв’язаних чоторикутників в функцію OnOpenGL() пропонується додати код Прикладу 8.8, результат виконання якого показано на Рис. 8.11:

```
glBegin(GL_QUAD_STRIP);
glColor3f(1.0, 0.0, 0.0);
glVertex2i(0, 0); glVertex2i(-2, 0);   glVertex2i(0, 2);  glVertex2i(-2, 4);
glColor3f(0.0, 1.0, 0.0);
glVertex2i(3, 2); glVertex2i(3, 4);
glEnd();
```

Приклад 8.8 – Код для зв’язаних чоторикутників GL_QUAD_STRIP

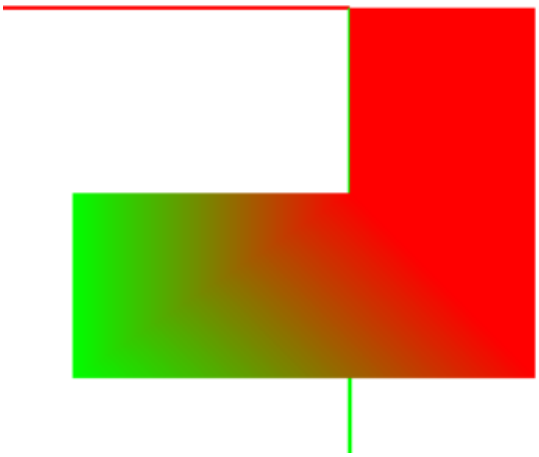


Рисунок 8.11 – Приклад для для зв’язаних чоторикутників GL_QUAD_STRIP

– полігони GL_POLYGON

для відображення полігонів в функцію OnOpenGL() пропонується додати код Прикладів 8.9 (правильний п’ятикутник) та Прикладу 8.10, що демонструє відвутність перетинання полігоном самого себе і наявність перетинання, що показано на Рис. 8.12 і 8.13. Різниця між погігонами – в одній вершині:

```
glBegin(GL_POLYGON);
glColor3f(0.0, 0.5, 1.0);
for (int i = 0; i < 5; i++)
glVertex2f(2.5 * System::Math::Cos(0.4 * System::Math::PI * i),
                        2.5 * System::Math::Sin(0.4 * System::Math::PI * i));
glEnd();
```

Приклад 8.9 – Код для правильного п’ятикутника GL_POLYGON

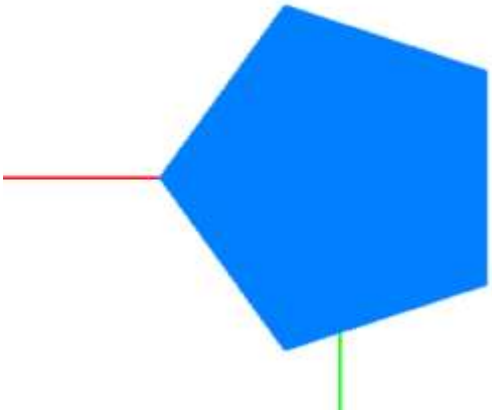


Рисунок 8.12 – Приклад для правильного п’ятикутника GL_POLYGON

```
glBegin(GL_POLYGON);
                    glColor3f(0.0, 0.5, 1.0);
                    glVertex2i(0, 0); glVertex2i(3, -3);
                    glVertex2i(-3, -3); glVertex2i(-5, 0);   // glVertex(5,0); // самоперетин
                    glVertex2i(-3, 3); glVertex2i(3, 3);
glEnd();
```

Приклад 8.10 – Код для полігона правильного типу і з самоперетином

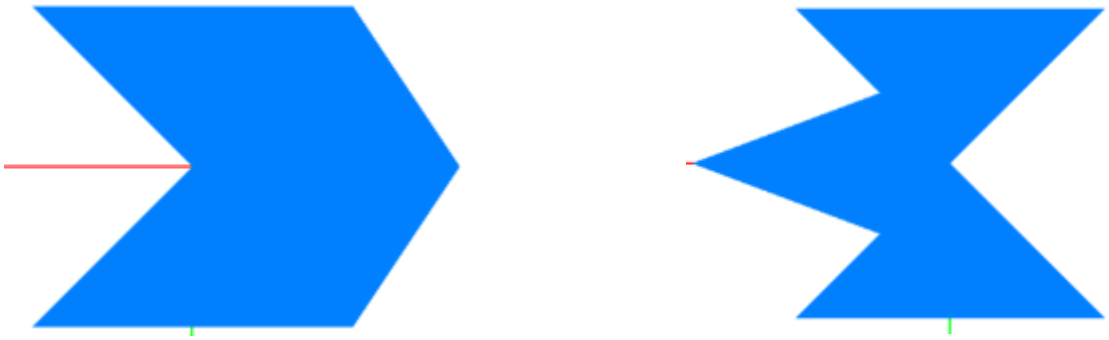


Рисунок 8.13 – Приклад для полігона правильного типу і з самоперетином

– незалежні точки GL_POINTS:

для відображення незалежних точок в функцію OnOpenGL() пропонується додати код Прикладу 8.11, результат виконання якого показано на Рис. 8.14:

```
glPointSize(20.0);
glBegin(GL_POINTS);
glColor3f(1.0, 0.0, 0.0); glVertex3i(0, 3, 0);
glColor3f(1.0, 1.0, 0.0); glVertex3i(0, 0, 3);
glColor3f(0.0, 0.0, 1.0); glVertex3i(3, 0, 0);
glEnd();
```

Приклад 8.11 – Код для незалежних точок GL_POINTS

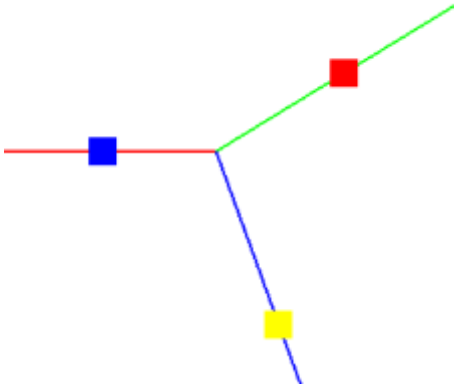


Рисунок 8.14 – Приклад для незалежних точок GL_POINTS

– коло на основі полігона:

оскільки читач помітив відвсутність можливості побудувати примітивами такий простий об’єкт, як коло, пропонується зробити це за допомогою зв’язаних ліній GL_LINE_LOOP і вже знайомого використання полярної системи координат, як це показано у Прикладі 8.12, результат виконання якого показано на Рис. 8.15:

```
glColor3f(0.0, 1.0, 0.0);      //  зелений колір
glLineWidth(5.0);                                  //  ширина лінії – 5.0
glBegin(GL_LINE_LOOP);                        //  відрізки кола
for (int i = 0; i < 100; i++)
{          double angle = 2 * System::Math::PI * i / 100;
           glVertex2f(3.5 * System::Math::Cos(angle), 3.5 * System::Math::Sin(angle));
}
glEnd();
```

Приклад 8.12 – Код побудова кола за допомогою GL_LINE_LOOP

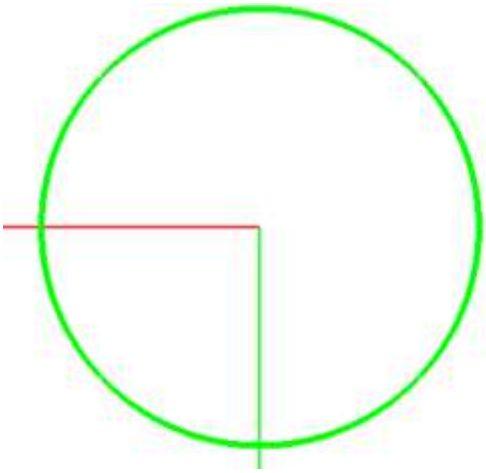


Рисунок 8.15 – Приклад побудови кола за допомогою GL_LINE_LOOP

8.6.2 Команди побудови прямокутників та штрихових ліній

Побудову прямокутників, крім використання примітивів GL_QUADS можна здійснити і більш простим способом, використавши спеціальну команду *glRect*()*:

void glRect[d f i s] (type X1, type Y1, type X2, type Y2);

Команда *glRect*()* для роботи потребує вказання двох діагональних вершин прямокутника – (X1, Y1) та (X2, Y2). По суті записи *glRecti(x1, y1, x2, y2)* та

```
glBegin(GL_POLYGON);
        glVertex2i(x1, y1);
        glVertex2i(x2, y1);
        glVertex2i(x2, y2);
        glVertex2i(x1, y2);
glEnd( );
```

є еквівалентними.

Наприклад, якщо необхідно побудувати синій прямокутник із діагональними вершинами (3.0, 3.0) та (6.0, 6.0), можна написати такий код:

```
glColor3f(0.0, 0.5, 1.0);
glRecti(0, 0, 5, 3);
glColor3f(1.0, 0.5, 0.0);
glRecti(-1, 0, -5, -3);
```

Результат виконання коду показує рис. 8.16:

Рисунок 8.16 – Приклад побудови прямокутників командою glRecti()

Для отримання пунктирних або штрих-пунктирних ліній використовується команда glLineStipple(), що задає шаблон штрихування:

```
void glLineStipple(GLint factor, GLushort pattern);
```

де *factor* – масштабний коефіцієнт, що дозволяє розтягувати наявний шаблон (значеннями від 1 до 256), а *pattern* – 16-розрядна послідовність нулів та одиниць, яка становить шаблон побудови штрихування лінії. Зокрема, лінія з точок задається значенням 0x0101, пунктирна лінія – 0x00FF, штрих-пунктирна лі- нія – 0x1C47. Для використання штрихування необхідно за допомогою функції *glEnable()* включити відповідний режим роботи OpenGL-програми, що і показано у прикладі 8.13:

```
glEnable(GL_LINE_STIPPLE);
glLineStipple(13.0, 0x1C47);
glLineWidth(13.0);
glBegin(GL_LINES);
glColor3f(0.0, 0.0, 1.0);
glVertex3f(5.0f, 0.0f, 0.0f); glVertex3f(0.0f, 5.0f, 0.0f);
glColor3f(1.0, 0.0, 0.0);
glVertex3f(0.0f, 0.0f, 0.0f); glVertex3f(5.0f, 5.0f, 0.0f);
glEnd();
glDisable(GL_LINE_STIPPLE);
```

Приклад 8.13 – Код побудова штрихових ліній

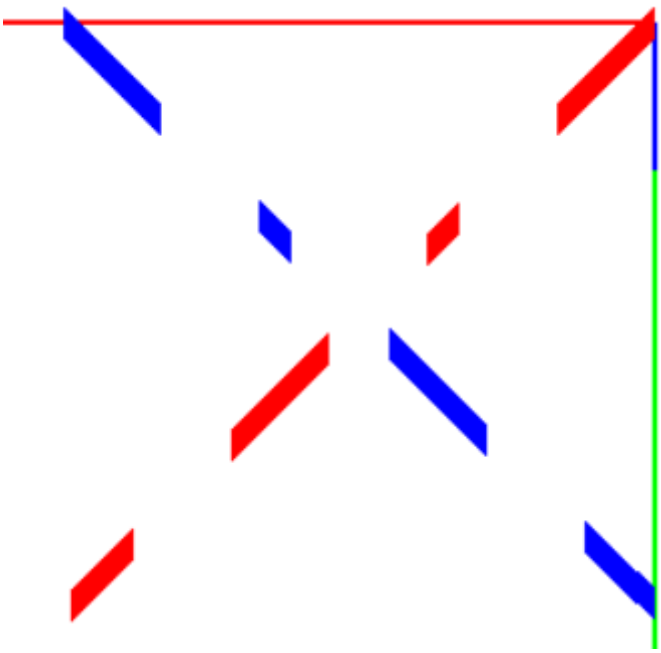


Рисунок 8.16 – Приклад побудови штрихових ліній

8.6.3 OpenGL як кінцевий автомат

Графічна система OpenGL є кінцевим автоматом. Цей автомат може переводитися у різні стани та залишатися у певному стані, доки цей стан не буде змінено новою командою.

За допомогою спеціальних змінних стану забезпечується керування кольором об'єктів, поточною візуалізацією, перетвореннями проекцій, шаблонами штрихування ліній, режимами введення багатокутників, пакування пікселів, контролювати властивості матеріалів, встановлювати розташування та визначати режими роботи джерел освітлення тощо.

За замовчуванням більшість режимів є неактивними. Активізація режимів роботи може призводити до уповільнення швидкості візуалізації, хоча і забезпечуватиме підвищення якості зображень.

Для увімкнення та вимкнення режимів використовуються прості команди:

```
void  glEnable(GLenum cap);
void  glDisable(GLenum cap);
```

Константа *cap* і є тим параметром стану OpenGL, що необхідно увікнути або вимкнути. OpenGL містить більше 40 таких параметрів контролю стану. Серед них такі:

GL_BLEND – керування змішуванням значень кольорів RGBA;

GL_DEPTH_TEST – керування тестом глибини;

GL_FOG – керування туманом;

GL_LINE_STIPPLE – штрихування ліній;

GL_TEXTURE_2D – робота з двовимірними текстурами;

GL_LIGHTING – керування освітленням.

Наприклад, командою *glEnable(GL_LINE_STIPPLE)* можна увімкнути штрихування ліній, а командою *glDisable(GL_LIGHTING)* вимкнути освітлення робочої сцени візуалізації.

За допомогою команди *gllsEnabled()* здійснюється перевірка стану:

```
GLboolean glIsEnabled(GLenum cap); // повертаються значення GL_TRUE та  GL_FALSE
```

Параметром *cap* вказується константа, стан якої контролюється.

8.7 Формування списків зображень та складені об’єкти

8.7.1 Списки зображень

Як видно з попередніх сторінок, програмний код побудови графічних об’єктів, що складаються з великої кількості вершин, є досить об’ємним, особливо, якщо однакові об’єкти доводиться будувати декілька разів. З метою оптимізації виконання таких однакових ділянок коду у OpenGL пропонується використання списків зображень. Список зображень є групою команд OpenGL, що збережена для подальшого виконання.

Використання списків зображень дозволяє поліпшити продуктивність програми, забезпечити кешування команд та їх збереження для подальшого виконання, оптимізувати матричні операції. Крім того, застосування списків зображень дозволяє забезпечити компіляцію растрових зображень, що використовуються у програмі, прискорити відображення освітлених сцен із джерелами освітлення та визначеними властивостями матеріалів і моделями розповсюдження світла, максимізувати ефективність обробки текстур.

Для роботи зі списком слід визначити ім’я списку, обмежити за допомогою спеціальних команд початок та кінець списку.

Кожен список ідентифікується цілочисельним індексом. Неспівпадаючі індекси генеруються командою *glGenLists()*:

```
GLuint glGenLists(GLsizei range);
```

де *range* – кількість суміжних номерів індексів списків зображень.

Команди *glNewList()* та *glEndList()* визначають, відповідно, початок та кінець списку зображень:

```
void glNewList(GLuint list, GLenum mode);
void glEndList(void);
```

де list – ідентифікатор списку, mode – режим формування списку. Режим формування має два значення: GL_COMPILE (список компілюється без виконання) та GL_COMPILE_AND_EXECUTE (список компілюється і одразу виконується). Між викликами команд *glNewList()* та *glEndList()* розміщується, власне, перелік команд, що мають включатися у список. Можна, наприклад, побудувати список побудови чотирьох сторін куба:

```
GLuint Cube;                        // оголошення змінної -  ідентифікатора списку
Cube=glGenLists(1);                 // генерація ідентифікатора списку
glNewList(Cube,GL_COMPILE);  // початок списку
        glBegin(GL_POLYGON);
```

```
                glVertex3f(5.0f,5.0f,5.0f);    glVertex3f(5.0f,-5.0f,5.0f);
                glVertex3f(5.0f,-5.0f,-5.0f); glVertex3f(5.0f,5.0f,-5.0f);
            glEnd();
            glBegin(GL_POLYGON);
                glVertex3f(-5.0f,5.0f,-5.0f); glVertex3f(-5.0f,-5.0f,-5.0f);
                glVertex3f(-5.0f,-5.0f,5.0f); glVertex3f(-5.0f,5.0f,5.0f);
            glEnd();
            glBegin(GL_POLYGON);
                glVertex3f(-5.0f,-5.0f,5.0f); glVertex3f(-5.0f,-5.0f,-5.0f);
                glVertex3f(5.0f,-5.0f,-5.0f); glVertex3f(5.0f,-5.0f,5.0f);
            glEnd();
            glBegin(GL_POLYGON);
                glVertex3f(-5.0f,5.0f,-5.0f); glVertex3f(-5.0f,5.0f,5.0f);
                glVertex3f(5.0f,5.0f,5.0f);    glVertex3f(5.0f,5.0f,-5.0f);
            glEnd();
        glEndList();                    // кінець списку
glCallList(Cube);                // виклик списку
```

Як видно з прикладу, виклик раніше підготовленого списку здійснюється за допомогою функції *glCallList()*:

void glCallList(GLuint list); // list – ідентифікатор списку

Списки графічних зображень можуть складатися у списки більш високого рівня. Таким чином організація списків носитиме ієрархічний характер.

8.7.2 Використання складених об’єктів

Крім простих об’єктів, що формуються прімітивами, існують можливості використання команд, що забезпечують побудову готових складених об’єктів. Їх побудова забезпечується низкою додаткових бібліотек: GLAUX, GLU, GLUT. Використання GLAUX досить довго було надійним і швидким способом отримання доволі складних конструкцій, що складалися з призм, сфер, циліндрів, конусів. Проте, на разі ми маємо неповну сумісність GLAUX і Microsoft Visual Studio, що унеможливлює її використання у більшості випадків.

Доволі надійним способом побудови складених графічних об’єктів є бібліотека GLUT (Graphical Library Utulity Toolkit) [10]. Складені графічні об’єкти GLUT існують у версіях побудови каркасних та твердотільних тривимірних моделей. Каркасні функції тривимірних об’єктів наведено нижче, а твердотільні версії відрізняються лише наявністю складу *Solid* замість *Wire*. Інші операційні системи мають дещо інший склад функцій.

```
// сфера
GLUTAPI void APIENTRY glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
// конус
```

```
GLUTAPI void APIENTRY glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);
// куб
GLUTAPI void APIENTRY glutWireCube(GLdouble size);
// тор
GLUTAPI void APIENTRY glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,
                                    GLint sides, GLint rings);
// додекаедр
GLUTAPI void APIENTRY glutWireDodecahedron(void);
// чайник
GLUTAPI void APIENTRY glutWireTeapot(GLdouble size);
// октаедр
GLUTAPI void APIENTRY glutWireOctahedron(void);
// тетраедр
GLUTAPI void APIENTRY glutWireTetrahedron(void);
// ікосаедр
GLUTAPI void APIENTRY glutWireIcosahedron(void);
```

Для використання бібліотеки GLUT необхідно:

- скачати GLUT за посиланням [10];
- скопіювати GLUT-файли glut.h, glut32.dll, glut32.lib у теку проекту із файлами MyForm.h MyForm.cpp;
- додати директиву підключення бібліотеки glut32.lb:

```
#pragma comment(lib,"glut32.lib")
```

- додати директиву підключення заголовочного файла glut.h:

```
#include "glut.h"
```

- переключити проект у режим платформи x86;
- додати команди GLUT у функцію OnOpenGL().

Для відображення складеного об’єкта бібліотеки GLUT в функцію OnOpenGL() пропонується додати код Прикладу 8.14, а результат його виконання показано на Рис. 8.17:

```
glColor3f(1.0f, 0.0f, 0.0f);
glutWireSphere(0.7, 20, 20);
glColor3f(0.0f, 0.0f, 1.0f);
glutWireTorus(0.7, 2.0, 20, 20);
```

Приклад 8.13 – Код із GLUT-об’єктами (сфера і тор)

Параметрами сфери є радіус та кількість горизонтальних і вертикальних ліній заповнення. Для тора задаються радіус труби, зовнішній радіус, заповнення.

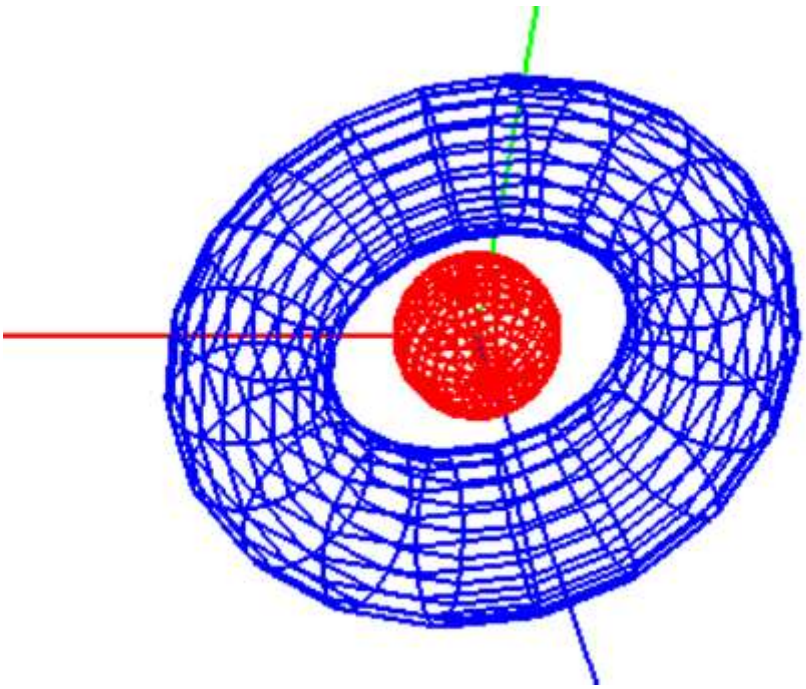


Рисунок 8.17 – Приклад побудови GLUT-об’єктів

8.8 Команди візуалізації OpenGL

8.8.1 Загальні відомості

Основним призначенням комп’ютерної графіки є створення двовимірного зображення тривимірних об’єктів. Це зображення має біти двовимірним з тієї причини, що будь-якому випадку екран комп’ютера є двовимірним.

Перетворення тривимірних координат об’єктів у піксельні точки на екрані включає операції моделювання, візуалізації або вибору кадру (viewing) та операції отримання проекцій. До таких операцій належать обертання (rotation), паралельне перенесення (translation), масштабування (scaling), дзеркальне відображення (reflecting), отримання ортогональних та перспективних проекцій [5]. Зазвичай використовується комбінація декількох перетворень для графічного виведення сцени.

Особливістю виконання команд візуалізації є і те, що оскільки виведення графічної інформації відбувається у прямокутне вікно, об’єкти (або частини об’єктів), що знаходяться за межами цього вікна, мають відсікатися. У тривимірній графіці відсікання виконується за допомогою відкидання об’єктів, що знаходяться по один бік площини відсічення.

Також має встановлюватися відповідність між перетвореними координатами та пікселями екрана. Вказана операція має назву перетворення порту перегляду (viewport).

Для отримання необхідної сцени під час візуалізації виконуються операції перетворення, що є аналогами отримання знімку за допомогою фотокамери. При цьому можна встановити перелік кроків отримання знімку [6]:

1) встановити штатив та направити камеру на сцену для вибору кадру або перетворення вигляду (view transformation);

2) встановити бажану композицію сцени, що фотографується, для перетворення моделі (modeling transformation);

3) оберіть об’єктив камери та визначте масштаб для перетворення проекції (project transformation);

4) визначте наскільки завеликою має бути кінцевий знімок для перетворення порту перегляду (viewport transformation).

Вказані етапи не обов’язково відповідають порядку, у якому виконуються відповідні математичні операції над вершинами об’єкта. На рис. 8.18 наведено порядок здійснення таких операцій.

Рисунок 8.18 – Етапи перетворення вершин

Для встановлення перетворення виду, моделювання та перетворення проекцій створюється матриця *M* розміру *4 x 4*, що потім перемножується на координати кожної вершини *v* на сцені під час виконання перетворення $v'=M \cdot v$. Вершини завжди мають чотири координати (*x*, *y*, *z*, *w*), хоча у більшості випадків координата *w* дорівнює 1, а для двовимірних даних координата *z* дорівнює 0. Слід звернути увагу, що перетворення виду і моделі також автоматично застосовуються до векторів нормалей до поверхні.

Задані перетворення виду та моделі комбінуються для створення матриці видового перетворення (view model matrix), що застосовується до координат об’єктів, які надходять під час їх побудови, для отримання видових координат.

Надалі, якщо задано додаткові площини відсікання, вони будуть використані для вилучення певних об'єктів зі сцени або для отримання перерізу об'єктів.

Після цього OpenGL застосовує матрицю проекцій для отримання координат відсікання (clip coordinates). Це перетворення визначає відображуваний об'єм; об'єкти за його межами відсікаються і, таким чином не відображуються у кінцевій сцені. Наступним кроком виконується перспективний поділ (perspective division), під час якого усі значення координат поділяються на *w* для створення нормалізованих координат пристрою (normalized device coordinates). Нарешті, перетворені координати конвертуються у віконні координати (window coordinates) за допомогою перетворення порту перегляду. Розміри порту перегляду можна змінювати, отримуючи збільшене, стиснуте або розтягнуте зображення.

8.8.2 Особливості систем координат OpenGL

Як вже зазначувалося, у OpenGL вершини завжди задаються чотирма координатами. Це пов'язано [7] з використанням однорідних координат, що надають можливість подання усіх видів координатних перетворень у єдиній формі. В однорідній системі координат положення точки *P(x, y, z)* записується у вигляді *P(W·x, W·y, W·z)* або *P(X, Y, Z, W)* для будь-якого масштабного множника, причому тривимірні координати легко визначаються у вигляді:

$$x = X / W ; y = Y / W; z = Z / W.$$

Таким чином звичайна ортогональна система координат отримується у вигляді проекції однорідної системи на площину *W=1*.

У матричній формі для позначення координат точки у певному тривимірному просторі з використанням однорідних координат застосовується запис [X Y Z W].

Перетворення однорідних координат записуються співвідношеннями:

$$[X\ Y\ Z\ W] = T \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad \text{та} \quad [x'\ y'\ z'\ 1] = \left[\frac{X}{W}\ \frac{Y}{W}\ \frac{Z}{W}\ 1 \right],$$

де T – матриця перетворення.

Зокрема, такі перетворення координат, як $x' = x + l; y' = y + m; z' = z + n$, можна записати у такій матричній формі:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & l \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & n \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+l \\ y+m \\ z+n \end{bmatrix} \quad \text{або} \quad \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & l \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & n \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Якщо ж розглядати тривимірне змінення масштабу – $x'' = x \cdot a$; $y'' = y \cdot e$; $z' = z \cdot j$, можна отримати такі матричні перетворення:

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Перетворення, пов'язані з переміщеннями та масштабуванням, можна об'єднати у єдину матричну форму:

$$\begin{bmatrix} x''' \\ y''' \\ z''' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & l \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & n \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & e & 0 & 0 \\ 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a & 0 & 0 & l \\ 0 & e & 0 & m \\ 0 & 0 & j & n \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Узагальнена матриця перетворень тривимірних координат розміру 4 х 4 має вигляд:

$\begin{bmatrix} a & d & h & l \\ b & e & i & m \\ c & f & j & n \\ p & q & r & s \end{bmatrix}$, при цьому підматриця $\begin{bmatrix} a & d & h \\ b & e & i \\ c & f & j \end{bmatrix}$ здійснює лінійні перетворення – зміну масштабу, зсув та обертання. Вектор $\begin{bmatrix} l \\ m \\ n \end{bmatrix}$ здійснює паралельне перенесення, вектор $[p \quad q \quad r]$ здійснює перспективне перетворення, а останній скалярний елемент s – загальну зміну масштабу.

Повне перетворення, отримане шляхом впливу 4х4-матриці на вектор положення та нормалізації отриманого вектора називають білінійним перетворенням [7].

8.8.3 Перетворення координат

У OpenGL передбачається існування 4-х типів матриці – видових, проекційних та текстурних.

Будь-яка матриця може бути встановлена як поточна за допомогою команди *glMatrixMode()*:

void glMatrixMode(GLenum mode);

де параметр *mode* визначає, з яким набором матриць виконуватимуться наступні операції. Значення параметра вказуються у таблиці 8.5.

Таблиця 8.5 – Типові параметри функції *glMatrixMode()*

Значення параметра	Коментар
GL_MODELVIEW	Послідовність операцій над матрицями застосовується до матриці вигляду
GL_PROJECTION	Послідовність операцій над матрицями застосовується до матриці проекції
GL_TEXTURE	Послідовність операцій над матрицями застосовується до матриці текстури

Визначення елементів матриці забезпечує функція *glLoadMatrix*()*:

void glLoadMatrix [f d] (GLtype *m);

у якій параметр *m* – визначає покажчик на 4x4-матрицю. Матриця зберігається як 16 дійсних значень і має тип float або double:

$$\begin{bmatrix} a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \\ a_4 & a_8 & a_{12} & a_{16} \end{bmatrix}$$

Команда замінює поточну матрицю на матрицю, визначену параметром *m*. Поточною є одна з матриць – видова, проекцій або текстури, в залежності від обраного функцією *glMatrixMode()*.

Наступна функція *glLoadIdentify()* також замінює поточну матрицю на одиничну (матрицю ідентичності).

void glLoadIdentify ( );

За змістом вона є еквівалентом *glLoadMatrix*()* із тією різницею, що матриця має головну діагональ, складену з одиниць, а усі інші елементи мають значення 0:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ще одна функція – *glMultMatrix*()* здійснює операцію перемножування матриці *m* (розміром 4x4) на поточну матрицю:

```
void glMultMatrix [f d] (GLtype *m);
```

Результатом *glMultMatrix*()* є встановлення поточної матриці M • T. Зазначимо, що з точки зору обчислень ця операція може бути досить складною.

8.8.4 Перетворення виду

Перетвореннями виду змінюють позицію і орієнтацію точки спостереження за об'єктом. Якщо згадати аналогію камери, можна сказати, що перетворення вигляду (вибір кадру) розміщує штатив камери, орієнтуючи саму камеру на об'єкт. Перетворення виду складаються з паралельних перенесень та поворотів. Для досягнення певної композиції сцени можна або перемістити камеру, або пересунути усі об'єкти у протилежному напрямі. Наприклад, перетворення моделі, що включає обертання об'єкта проти годинникової стрілки є еквівалентним перетворенню виду, що обертає камеру у протилежному напрямі.

В цьому сенсі, команди .NET, описані у розділі 4.9 є 2D-версіями команд OpenGL, хоча і створені набагато пізніше.

Обертання об'єктів здійснюється за допомогою команди *glRotate*()*:

```
void glRotate [f d] (GLtype angle, GLtype x, GLtype y, GLtype z);
```

де обертання на кут *angle* здійснюється навколо вектора, спрямованого з центру системи координат у точку *(x, y, z)*. Виконання команди повертає усі об'єкти.

Припустимо, що у OpenGL-програмі визначаються осі координат, після чого у залежності від значення кута відбувається обертання прямокутника навколо осі *X* (рисунок 8.19). Такі дії програми відображаються кодом Прикладу 8.14:

```
        glLineWidth(2.0);
        glColor3f(0.0, 0.0, 1.0);
        // обертання навколо X з кроком кута 30 градусів
glRotatef(30 * vspos, 1, 0, 0);
```

```
glRectf(3.0, 3.0, 6.0, 6.0);              // відображення прямокутника
```

Приклад 8.14 – Код із використанням команди glRotatef()

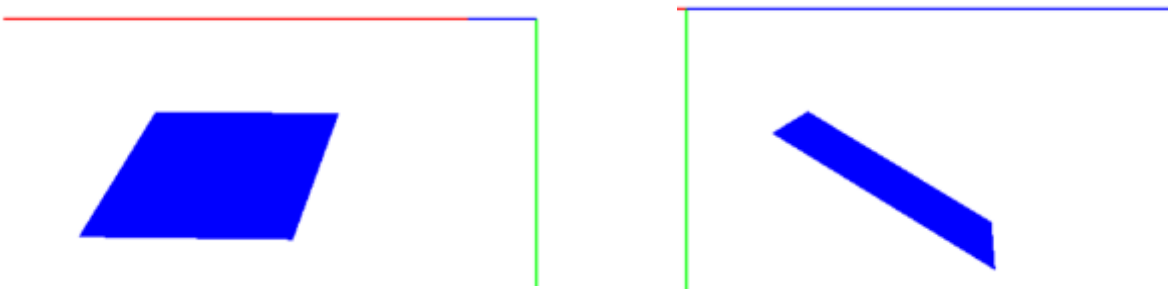


Рисунок 8.19 – Приклад із використанням команди glRotatef()

Паралельне перенесення об’єктів здійснюється за допомогою команди *glTranslate*()*:

```
void glTranslate [f d] (GLtype x, GLtype y, GLtype z);
```

За допомогою цієї команди здійснюється перенесення об’єкта на відстань *x* вздовж осі *X*, *y* – вздовж осі *Y*, *z* – вздовж осі *Z*.

Якщо у OpenGL-програмі у залежності від зміни координати *x* відбувається перенесення об’єкта, дії програми можна відобразити такою частиною програмного коду Прикладу 8.14:

```
glColor3f(0.0, 0.0, 1.0);
glTranslatef(-10 + hspos, 0, 0);        // перенесення вздовж осі X
glRectf(3.0, 3.0, 6.0, 6.0);
```

Приклад 8.14 – Код із використанням команди glTranslatef()

Результат виконання такого програмного коду зображено на рис. 8.20.

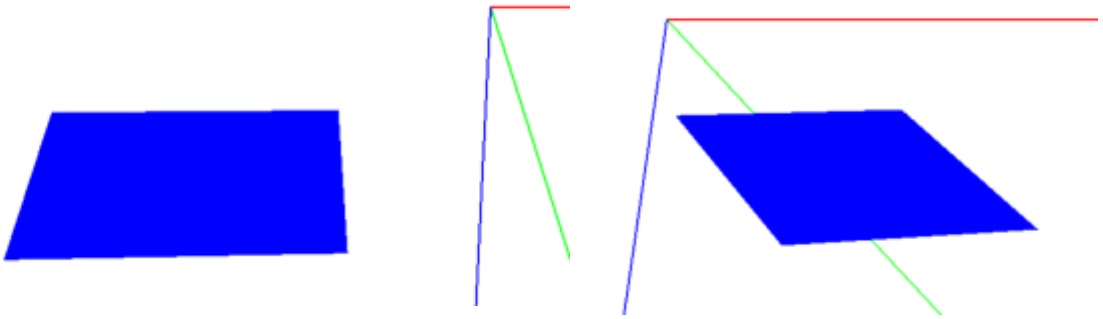


Рисунок 8.20 – Приклад із використанням команди glTranslatef()

Масштабування розмірів об’єктів вздовж визначених координатних осей здійснюється функцією *glScale*()*:

```
void glScale [f d] (GLtype x, GLtype y, GLtype z);
```

Під час виконання перерахованих команд перетворення об’єктів OpenGL здійснює перемноження поточної матриці (*M*) на відповідну матрицю обертання (*R*) або перенесення (*T*) чи масштабування (*S*) і розміщує результат (M · R, M · T або M · S) замість поточної матриці.

Якщо у програми, описаної для обертання та трансляції перед відображенням прямокутника, додати рядок:

```
glScalef(0.25 * vspos, 0.25 * vspos, 1.0);
```

можна отримати вікна програм, зображені на рис. 8.21.

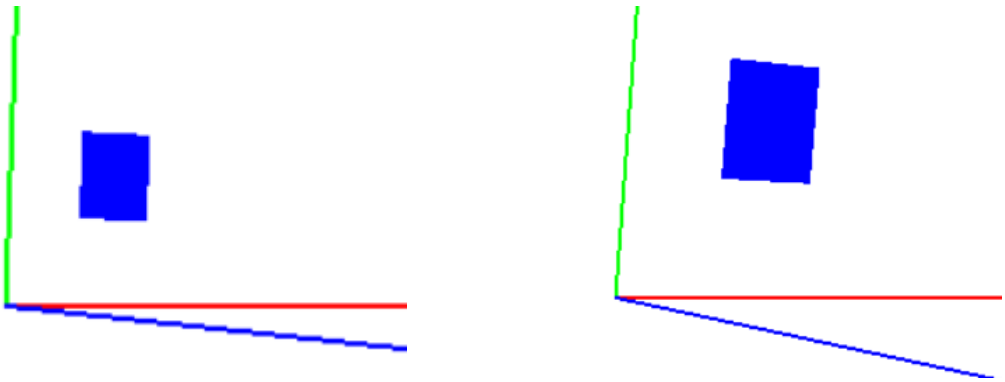


Рисунок 8.21 – Приклад із використанням команди glScalef()

Доволі часто доводиться створювати сцену на початку системи координат або іншому зручному місці, а потім переглядати її з певної точки. Команда *gluLookAt()* призначена саме для цього:

```
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez
                GLdouble centerx, GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz);
```

Така команда приймає три набори аргументів: точку спостереження (*eyex, eyey, eyez*); контрольну точку (*centerx, centery, centerz*), до якої спрямовується камера; точку, що характеризує вектор верхнього напряму (*upx, upy, upz*) – напрям знизу-вгору зони видимості. Для роботи слід обрати точку спостереження, з якої отримується необхідний вид. Контрольна точка зазвичай знаходиться приблизно посередині сцени. Встановлення напряму вгору (Up-vector) інколи може викликати певні складнощі і залежатиме від положення самого об’єкта.

8.9 Керування матричними стеками і приклади складених об’єктів

8.9.1 Команди роботи з матричними стеками

Матриці видового перетворення та проекцій, що створювалися раніше, насправді були лише верхніми елементами у матричних стеках, що використовуються OpenGL. Матричні стеки є дуже корисним способом побудови ієрархічних комп’ютерних моделей. У тих випадках, коли необхідно відображати декілька разів одні й ті самі об’єкти, можна застосувати переходи від одного елемента матричного стека, що характеризує певний стан геометричного перетворення координат, до іншого елемента стека.

Оскільки перетворення координат зберігаються у вигляді матриць, матричний стек є ідеальним механізмом виконання успішних операцій запам’ятовування, паралельного перенесення та відміни. Усі матричні операції, описані до сих пір (*glLoadMatrix(), glMultMatrix(), glLoadIdentify()*), взаємодіють з поточною матрицею, тобто з верхньою матрицею стека. Операції з матричним стеком забезпечуються командами *glPushMatrix()* та *glPopMatrix()*:

```
void glPushMatrix(void);
void glPopMatrix(void);
```

Команда *glPushMatrix()* додає поточну матрицю у вершину стека. При цьому зберігаються усі попередні координатні матриці – як характеристики попередніх станів системи.

Команда *glPopMatrix()* видаляє верхівку стека – координатну матрицю із характеристикою певного стану.

Обидві команди доречно використовувати під час побудови складених об’єктів OpenGL-програм.

Використовуючи команду *glPushMatrix()* можна запам’ятати поточний стан координатних матриць графічної системи, зробити необхідні виклики команд OpenGL для побудови графічних об’єктів, потім здійснити видові перетворення (наприклад перенесення, обертання чи масштабування) і знову-таки побудувати необхідні примітиви OpenGL. Для повернення у попередній (до видових перетворень) стан систем координат OpenGL у програмі буде достатньо використати команду *glPopMatrix()*.

8.9.2 Приклад з об’єктом типу «шасі автомобіля»

Припустимо, що нам необхідно побудувати графічний об’єкт типу «шасі автомобіля», що складатиметься з рами – прямокутника і чотирьох прикріплених до рами коліс.

Для реалізації об’єкта «шасі автомобіля» її код запишемо у функцію OnChassis(), яку викличемо з OnOpenGL(). Прототип функції OnChassis() слід вказати на початку файла MyForm.h після прототипу OnOpenGL(). Аналогічний підхід будемо використати і в подальших прикладах.

При цьому для побудови прямокутника доречно використати команду glRecti(), а для коліс – GLUT-команду glutWireTorus(), що і реалізовано у прикладі 8.15 і показано на рис. 8.22. Послідовність побудови об’єкта пояснюється у коментарях.

```
void OnChassis(void)
{       // рама шасі - прямокутник
        glPushMatrix();  // запам’ятати СК
        glRotatef(90.0, 1.0, 0.0, 0.0);//поворот СК
        glColor3f(0.0, 0.5, 1.0);
        glRecti(0, 0, 6, 6);   // рама шасі
        glPopMatrix();   // відновити СК
        // колеса шасі
        glPushMatrix(); // запам’ятати СК
        glColor3f(1.0, 0.5, 0.0);
        glPushMatrix(); // запам’ятати СК
        glutWireTorus(0.5, 1.0, 10, 10); // wheel1
        glPopMatrix(); // відновити СК
        glTranslatef(6.0, 0.0, 0.0); // перехід
        glPushMatrix(); // запам’ятати СК
        glutWireTorus(0.5, 1.0, 10, 10); // wheel2
        glPopMatrix(); // відновити СК
        glTranslatef(0.0, 0.0, 6.0); // перехід
        glPushMatrix(); // запам’ятати СК
        glutWireTorus(0.5, 1.0, 10, 10); // wheel3
        glPopMatrix(); // відновити СК
        glTranslatef(-6.0, 0.0, 0.0); // перехід
        glPushMatrix(); // запам’ятати СК
        glutWireTorus(0.5, 1.0, 10, 10); // wheel4
        glPopMatrix(); // відновити СК
        glPopMatrix(); // відновити СК
}
```

Приклад 8.15 – Код функції побудови об’єкта «шасі автомобіля»

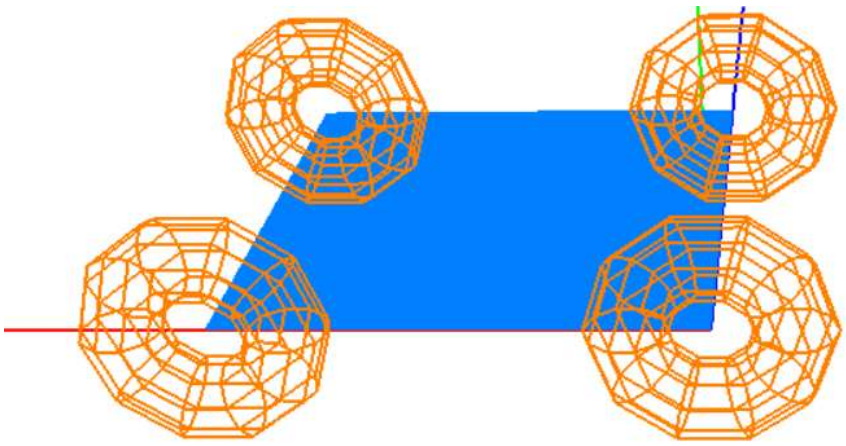


Рисунок 8.22 – Приклад простої моделі «Шасі автомобіля»

8.9.3 Приклад з об’єктом типу «ракета»

Припустимо, що нам необхідно побудувати графічний об’єкт типу ракета, що складатиметься з циліндра – корпуса ракети, конуса – носу ракети та радіально розташованих чотирьох трикутників – стабілізаторів. Для цього реалізуємо і додамо (аналогічно «шасі») функцію OnRocket(). що реалізовано у Прикладі 8.16 і показано на Рис. 8.23. Послідовність побудови об’єкта пояснюється у коментарях.

```
void OnRocket()
{
        glPushMatrix();
        glScalef(0.3, 0.3, 0.3);
        glPushMatrix();// запам'ятати положення основи ракети
        glColor3f(0.0, 0.0, 1.0);
        glRotatef(90.0, 1.0, 0.0, 0.0);// поворот конуса на 90 осі X
        gluCylinder(gluNewQuadric(), 1.5, 1.5, 10.0, 10, 10); // base
        glTranslatef(0.0, 0.0, 10.0); // перехід до верхівки циліндра
        glutSolidCone(1.5, 3.0, 10, 10);
        glPopMatrix();// повертаємося до основи ракети
        glPushMatrix();
        for (int i = 0; i < 4; i++)// цикл із стабілізаторами
        {
                glRotatef(90.0 * i, 0.0, 1.0, 0.0);
                glColor3f(1.0, 0.0, 0.0);
                glBegin(GL_TRIANGLES); // стабілізатор
                glVertex3f(1.5f, 0.0f, 0.0f);
glVertex3f(3.5f, 0.0f, 0.0f);
                glVertex3f(1.5f, -3.0f, 0.0f);
                glEnd();
        }
        glPopMatrix();
        glPopMatrix(); }
```

Приклад 8.16 – Код функції побудови об’єкта «Ракета»

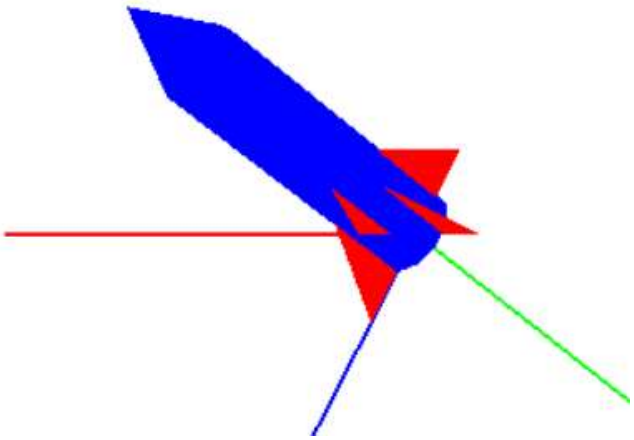


Рисунок 8.23 – Приклад простої моделі «Ракета»

8.9.4 Приклад з об’єктом типу «Маніпулятор робота»

У третьому прикладі побудуємо модель маніпулятора промислового робота. Робот будувати дещо складніше.

Кожен робот складається з 4-х рухомих ланок. З метою скорочення програмного коду зробимо ланки робота однотипними і зобразимо їх GLUT-циліндрами.

Також додамо в базову програму:

1. Для забезпечення рухів ланок маніпулятора додамо глобальними змінні кутів ланок маніпулятора:

columnPos – для керування колонною;

shoulderPos – для керування плечем

elbowPos – для керування плечем;

wristPos – для керування схватом руки робота:

```
int columnPos=0, shoulderPos=0, elbowPos=0, wristPos=0;
```

2. Для керування ланками робота додамо в форму MyForm чотири горизонтальні трекбари і передамо їх значення змінним ланок:

```
private: System::Void hScrollBar2_Scroll(System::Object^ sender, System::Windows::Forms::ScrollEventArgs^ e)
{
        columnPos = hScrollBar2->Value;
        Invalidate(); // виклик оновлення форми
}

private: System::Void hScrollBar3_Scroll(System::Object^ sender, System::Windows::Forms::ScrollEventArgs^ e)
{       shoulderPos = hScrollBar3->Value;
        Invalidate(); // виклик оновлення форми }
```

```
private: System::Void hScrollBar4_Scroll(System::Object^ sender, System::Windows::Forms::ScrollEventArgs^ e)
{
        elbowPos = hScrollBar4->Value;
        Invalidate(); // виклик оновлення форми  }

private: System::Void hScrollBar5_Scroll(System::Object^ sender, System::Windows::Forms::ScrollEventArgs^ e)
{
        wristPos = hScrollBar5->Value;
        Invalidate(); // виклик оновлення форми  }
```

3. Для відображення моделі маніпулятора робота визначимо функцію OnRobot(), реалізовану кодом Прикладу 8.17 і показану на Рис. 8.24.

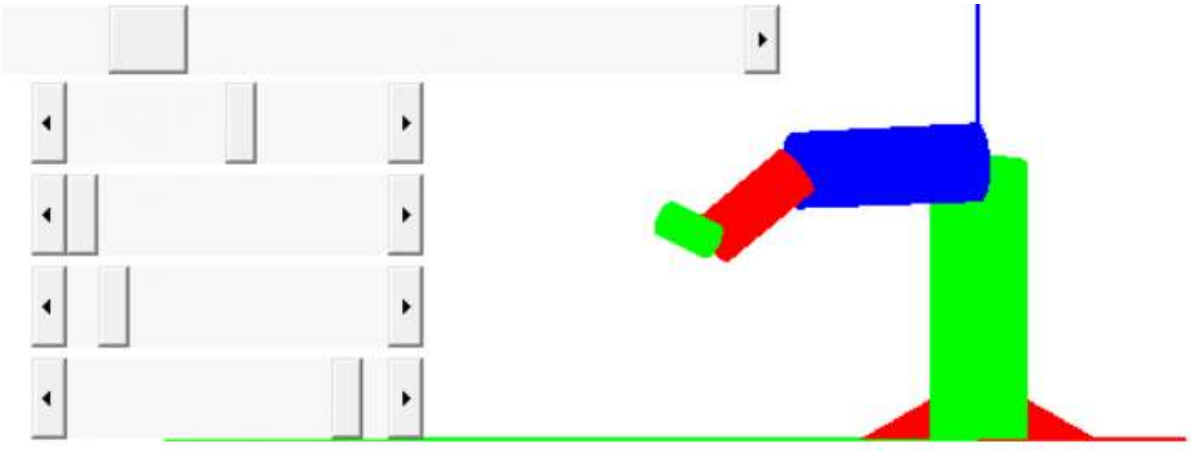


Рисунок 8.24 – Приклад простої моделі «Маніпулятор робота»

В заключенні додамо, що OpenGL є класичним прикладом реалізації математичних моделей у комп'ютерній графіці, що і обумовлює його тримале використання у різних програмних продуктах.

8.10 Засоби використання текстур OpenGL

8.10.1 Очищення вікна

Якщо згадувати виведення зображень у контекст пристрою вікна, очищення здійснюється простим зарисовуванням фоновим кольором необхідної частини вікна. З точки зору вирішення тривимірних завдань OpenGL ця проблема не є надто простою, бо доводиться встановлювати систему координат, положення точки спостереження, напрямок спостереження. Тому найбільш ефективним засобом є використання спеціальних команд.

Кольори пікселів, що обробляються апаратними засобами комп'ютерної графіки, відомі під назвою бітових площин. Вже згадувалося, що існує два методи зберігання такої інформації. Один метод передбачає збереження чотирьох компонентів кольору пікселів – червоного, зеленого, синього та альфа (режим RGBA) – безпосередньо у бітових площинах, другий забезпечує зберігання єдиного значення індексу, що посилається на таблицю відповідності кольорів. Частіше використовується режим RGBA.

Команда *glClearColor()* встановлює поточний колір очищення буфера кольору:

```
void glClearColor( GLclampf red,  GLclampf green,  GLclampf blue, GLclampf alpha );
```

Для роботи з функцією необхідно вказати колір очищення у форматі RGBA.

Безпосереднє очищення буфера кольору здійснюється командою *glClear()*:

```
void glClear(GLbitfield mask);
```

де параметром *mask* вказується необхідний для очищення буфер (або комбінація буферів – за допомогою символу |). Буферами можуть бути: буфер кольору GL_COLOR_BUFFER_BIT, глибини GL_DEPTH_BUFFER_BIT, буфер-накопичувач GL_ACCUM_BUFFER_BIT та трафарету GL_STENCIL_BUFFER_BIT.

Як правило колір очищення задається на початку програми, потім буфери очищуються за необхідністю. Операції очищення є нескладними за записом. Для очищення буферів кольору та глибини можна використати таку послідовність команд:

```
glClearColor(0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT);
```

8.10.2 Загальна інформація про застосування текстур

У розглянутих раніше програмах геометричні примітиви відображувалися або незаповненими, або заповнювалися кольором, встановленим для ліній, що складали примітив. Однак, якщо мати на меті відображення, наприклад, цегляного муру без накладення текстур, доведеться відображати кожну цеглину окремим багатокутником. Без застосування текстур великий плоский мур, що насправді є великим багатокутником, потребує побудови тисяч окремих цеглин. Але навіть і тоді цегла здаватиметься надто штучною і регулярною, не виглядатиме реалістично.

Накладення текстур дозволяє вклеїти зображення цегляного муру (яке можливо отримано з фотографії справжнього муру) у багатокутник і зобразити цілий мур простим багатокутником. Використання текстур надає і додаткові можливості. Якщо розглядати мур у перспективі, цеглини мають здалеку здаватися меншими, ніж з близької відстані. Текстури можна використати для зображення рослинності на поверхні землі або рельєфу місцевості під час моделювання польоту літака, для надання елементам оформлення програми реалістичних властивостей, подібних мармуру, деревині або тканинам. Хоча, зазвичай, текстури використовуються для заповнення багатокутників, їх можна застосувати до усіх примітивів – точок, прямих ліній, бітових зображень [5].

З програмної точки зору текстури є прямокутними масивами даних: даних про колір, яскравість або альфа-компоненти. Індивідуальні значення текстурного масиву часто називають елементами текстури або текселями (texels). Прямокутна текстура може накладатися на непрямокутні області і це може ускладнювати та уповільнювати процес відображення графічних примітивів.

Про текстури можна казати досить багато [5, 6]. У нашому випадку залишається навести загальний порядок їх застосування та конкретні працездатні приклади.

8.10.3 Команди створення і використання простих текстур

Для застосування текстур можна визначити формальну послідовність:

1) створити текстурний об’єкт та визначити його текстури;
2) вказати у який спосіб має застосовуватися текстура;
3) дозволити накладення текстур;
4) вивести сцену із одночасним зазначенням текстурних та геометричних координат або у інший спосіб.

Створення текстурних об’єктів доречно у тих випадках, коли у програмі визначається декілька різних типів структур. Процедура створення текстурних об’єктів дозволяє, створивши текстуру лише один раз, багаторазово використати її для накладення на різні графічні об’єкти.

Команда *glGenTextures()* забезпечує генерацію числового ідентифікатора текстури:

```
void glGenTextures( GLsizei n, GLuint *textures );
```

де *n* – кількість генерованих текстурних об’єктів, *textures* – покажчик на перший елемент масиву, у якому зберігаються імена текстур.

Команда *glBindTexture()* під час першого використання створює новий текстурний об’єкт, якому присвоює ім’я, задане аргументом *texture*:

```
void glBindTexture( GLenum target, GLuint texture );
```

а параметр *target* визначає тип текстури GL_TEXTURE_1D – одновимірна, GL_TEXTURE_2D – двовимірна, GL_TEXTURE_3D – тривимірна. Якщо текстурний об’єкт створено раніше, виклик *glBindTexture()* забезпечує активізацію текстурного об’єкта.

Найчастіше використовуються двовимірні текстури. Для їх визначення використовується OpenGL-команда *glTexImage2D()*:

```
void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);
```

Ця команда визначає фізичні параметри текстури: двовимірність, масштаб, властивості кольору, розміри зображення, формат кольору, тип піксельних даних. Останнім параметром команди є, власне, масив даних, що представляють опис текстури. Параметри команди та пояснення до них подані у таблиці 8.6.

Таблиця 8.6 – Параметри команди *glTexImage2D()*

Ім’я параметра	Коментар
1	2
target	цільова текстура, має дорівнювати GL_TEXTURE_2D
Level	рівень деталізації, 0 – базове значення, n – масштаб зменшення
internalformat	кількість компонентів кольору у текстурі; може приймати значення 1, 2, 3, 4 або значення символьних констант: GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_R3_G3_B2, GL_RGB, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, або GL_RGBA16

Продовження таблиці 8.6

Width	ширина зображення текстури (має бути числом $2^n + 2$(межа) при цілому n)
height	висота зображення текстури (має бути числом $2^m + 2$(межа) при цілому m)
border	ширина рамки (0 або 1)
format	формат піксельних даних; приймає 9 символьних значень: GL_RGBA, GL_RGB, GL_RED, GL_BLUE, GL_ALPHA,GL_GREEN, GL_COLOR_INDEX, GL_LUMINANCE, GL_LUMINANCE_ALPHA,
type	тип даних піксельних даних, значення GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, and GL_FLOAT
pixels	покажчик на масив даних із зображенням

Визначення парамерів текстури може забезпечуватися командами сімейства glTexParameter*():

```
void glTexParameteri(GLenum target, GLenum pname, GLint param);
void glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params);
```

Команда *glTexParameter*()* визначає параметри способу відображення текстури під час її застосування. Особливо це важливо в умовах неспівпадання розмірів текстури з розмірами об'єктів. При цьому може здійснюватися зменшення елементів текстури до розмірів пікселя (параметр GL_TEXTURE_MIN_FILTER), збільшення (GL_TEXTURE_MAG_FILTER), згортання координати *x* (GL_TEXTURE_WRAP_S) або *y* (GL_TEXTURE_WRAP_T).

Параметри команди та пояснення до них подані у таблиці 8.7.

Таблиця 8.7 – Параметри команди glTexParameter**()**

Ім'я параметра	Коментар
Target	цільова одно- (GL_TEXTURE_1D) або двовимірна (GL_TEXTURE_2D) текстура
Pname	символічне ім'я параметра текстури: GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T

Продовження таблиці 8.7

Param	Значення параметра *pname*
params	Покажчик на масив значень параметра *pname*, що задає функцію, яка використовується для застосування текстури до пікселя і може приймати такі значення: GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_LINEAR

Можливі значення параметрів *pname* і *param* зазначені у таблиці 8.8

Таблиця 8.8 – Значення параметрів *pname* і *param* команди glTexParameter***()**

Параметр	Коментар та значення
1	2
GL_TEXTURE_WRAP_S	Параметр згортання *s*-координати текстури у значення GL_CLAMP або GL_REPEAT
GL_TEXTURE_WRAP_T	Параметр згортання *t*-координати текстури у значення GL_CLAMP або GL_REPEAT
GL_TEXTURE_MAG_FILTER	Функція збільшення текстури Значення: GL_NEAREST або GL_LINEAR
GL_TEXTURE_MIN_FILTER	Функція мінімізації текстури. Значення: GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_ MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_ MIPMAP_LINEAR
GL_TEXTURE_BORDER_COLOR	Колір рамки. Будь-які чотири значення у діапазоні [0.0,1.0]
GL_TEXTURE_PRIORITY	Рівень пріоритету. Діапазон значень [0.0, 1.0]

Координати текстури можуть містити від однієї до чотирьох координат. Зазвичай вони позначаються як *s*-, *t*-, *r*-, *q*-координати (таке позначення виріз-

няє їх від звичайних геометричних). Координати текстури визначаються командами *glTexCoord*()*:

```
void glTexCoord [1 2 3 4] [s i f d] (TYPE coords);
void glTexCoord [1 2 3 4] [s i f d] v (TYPE *coords);
```

Дані команди використовуються для встановлення поточних координат текстури як частина даних, асоційованих з вершинами багатокутника. *glTexCoord*()* використовується разом із наступним викликом команди *glVertex*()*. Координати задаються або у явному вигляді, або у вигляді вектора. Зазвичай порядок побудови координат текстури багатокутника вказується у тій самій послідовності, що і координати вершин:

```
glBegin(GL_POLYGON);
glTexCoord2i(0,0);  glVertex3i(4,-2,0);      // координати вказані у порядку,
glTexCoord2i(0,1);  glVertex3i(4,4,0);       // що відповідає обходженню
glTexCoord2i(1,1);  glVertex3i(0,4,0);       // контуру за напрямом
glTexCoord2i(1,0);  glVertex3i(0,-2,0);      // годинникової стрілки
glEnd();
```

Інколи під час визначення текстур вказують параметри взаємодії текстури з фрагментом об’єкта. З цією метою OpenGL реалізує команди *glTexEnv*()*:

```
void glTexEnv [i f] (GLenum target, GLenum pname, GLtype param );
void glTexEnv [i f] v (GLenum target, GLenum pname, GLtype *params );
```

де *target* визначає конфігурацію текстури і завжди дорівнює GL_TEXTURE_ENV, *pname* – символічне ім’я параметра конфігурації текстури із значенням GL_TEXTURE_EN_MODE, *param* – символічна константа із значеннями GL_MODULATE, GL_DECAL, GL_REPLACE та GL_BLEND, *params* – покажчик на масив параметрів типу *param*.

8.10.4 Використання простих текстур на основі масивів даних

Розглянемо приклад програми із побудовою текстур. У програмі будуватиметься призма, до якої застосовується накладення текстури у вигляді шахової дошки. Через те, що у програмі використовується лише одна текстура, текстурний об’єкт окремо не визначається. Для побудови програми необхідно:

1. Створити масив даних текстури:

```
static GLubyte checkImage[64][64][4];
```

2. Ініціалізувати масив текстури:

```
void MakeCheckImage()
{for(int i=0;i<64;i++) {      // текстура має розмір 64 x 64
```

```
 for(int j=0;j<64;j++) { int c=(( ((i&0x8)==0)^((j&0x8))==0))*255;
        checkImage[i][j][0]=(GLubyte)c; checkImage[i][j][1]=(GLubyte)c;
        checkImage[i][j][2]=(GLubyte)c; checkImage[i][j][3]=(GLubyte)255;
}}}
```

3. Реалізувати приклад OnTexture1(), в якому здійснити визначення параметрів текстури і визначення її координат одночасно із побудовою графічних об’єктів. Ця функція наведена у прикладі 12.5 та показана на Рис. 8.25.

```
void OnTexture1()
{       glLineWidth(2.0);
        glColor4f(0.0, 0.0, 1.0, 0.0);
        glEnable(GL_DEPTH_TEST);        // увімкнення тесту глибини
        glEnable(GL_TEXTURE_2D);        // увімкнення використання текстур
        // увімкнення взаємодії текстури і примітиву
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        MakeCheckImage();               // ініціалізація масиву текстур
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        // визначення властивостей текстури
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 64, 64, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, checkImage);
        glBegin(GL_POLYGON);  // побудова сторони призми
                glTexCoord2i(0, 0); glVertex3i(4, -2, 0);
        glTexCoord2i(0, 1); glVertex3i(4, 4, 0);
                glTexCoord2i(1, 1); glVertex3i(0, 4, 0);
        glTexCoord2i(1, 0); glVertex3i(0, -2, 0);
        glEnd();
        glBegin(GL_POLYGON);
                glTexCoord2i(0, 0); glVertex3i(4, 4, 0);
        glTexCoord2i(0, 1); glVertex3i(0, 4, 0);
                glTexCoord2i(1, 1); glVertex3i(0, 4, 6);
        glTexCoord2i(1, 0); glVertex3i(4, 4, 6);
        glEnd();
        glBegin(GL_POLYGON);
                glTexCoord2i(0, 0); glVertex3i(4, -2, 6);
        glTexCoord2i(0, 1); glVertex3i(4, 4, 6);
                glTexCoord2i(1, 1); glVertex3i(0, 4, 6);
        glTexCoord2i(1, 0); glVertex3i(0, -2, 6);
        glEnd();
        glBegin(GL_POLYGON);
                glTexCoord2i(0, 0); glVertex3i(0, -2, 6);
        glTexCoord2i(0, 1); glVertex3i(0, -2, 0);
                glTexCoord2i(1, 1); glVertex3i(4, -2, 0);
        glTexCoord2i(1, 0); glVertex3i(4, -2, 6);
        glEnd();
        glBegin(GL_POLYGON);
                glTexCoord2i(0, 0); glVertex3i(4, -2, 0);
        glTexCoord2i(0, 1); glVertex3i(4, 4, 0);
                glTexCoord2i(1, 1); glVertex3i(4, 4, 6);
        glTexCoord2i(1, 0); glVertex3i(4, -2, 6);
        glEnd();
        glBegin(GL_POLYGON);
```

```
            glTexCoord2i(0, 0); glVertex3i(0, -2, 0);
        glTexCoord2i(0, 1); glVertex3i(0, 4, 0);
            glTexCoord2i(1, 1); glVertex3i(0, 4, 6);
        glTexCoord2i(1, 0); glVertex3i(0, -2, 6);
        glEnd();
        glDisable(GL_TEXTURE_2D);       // вимкнення режиму застосування текстур
}
```

Приклад 8.17 – Код функції OnTexture1() із текстурою, заданою масивом

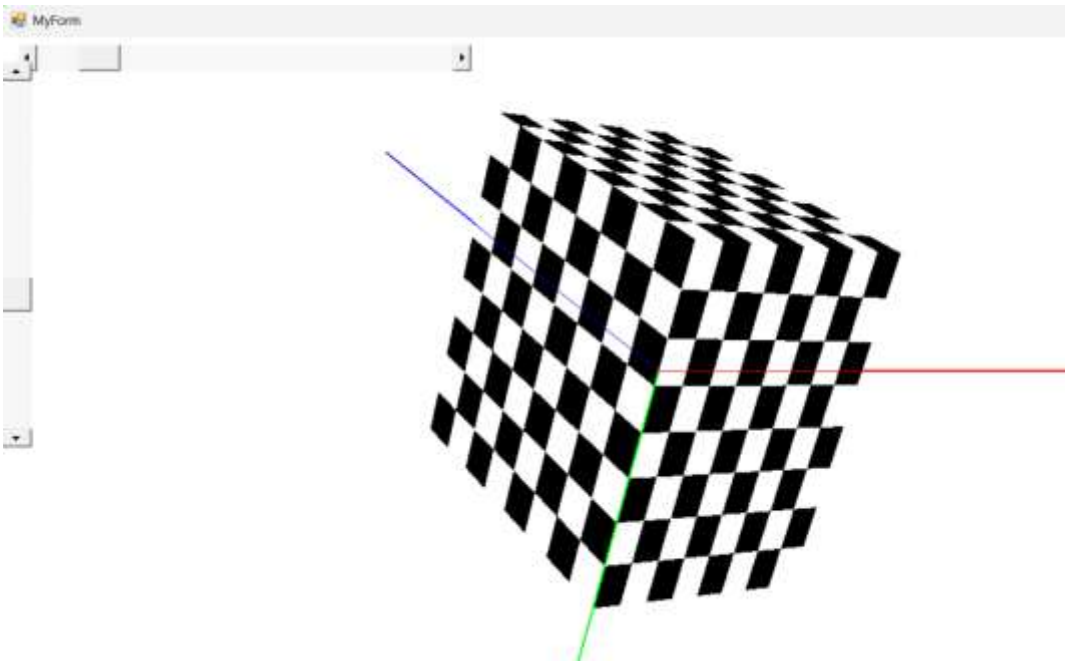


Рисунок 8.25 – Приклад застосування текстури на основі 2D-масива

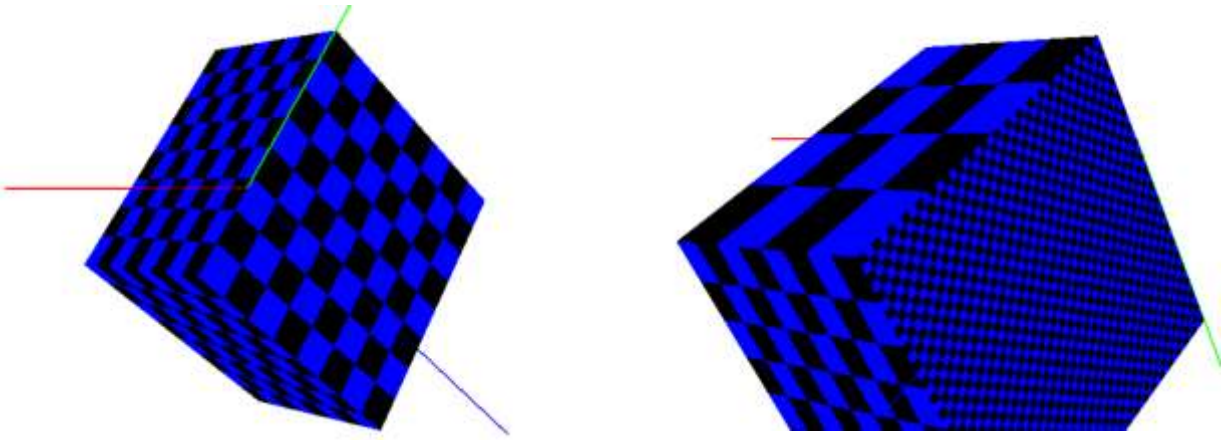
Заміна у функції glTexEnvf() режиму заміщення кольору GL_REPLACE на режим змішування GL_BLEND дає початковий колір об’єкта (Рис. 8.26-а).

Масштаб відображення окремих сторін об’єкта може бути змінено змінами масштабних коефіцієнтів текстурних координат, що показано у Прикладі 8.18 і на Рис. 8.26-б:

```
// визначення різного масштабу текстури
glBegin(GL_POLYGON);// збільшення масштабу у 2 рази
glTexCoord2d(0.0, 0.0); glVertex3i(0, -2, 6);
glTexCoord2d(0.0, 0.5);  glVertex3i(0, -2, 0);
glTexCoord2d(0.5, 0.5); glVertex3i(4, -2, 0);
glTexCoord2d(0.5, 0.0); glVertex3i(4, -2, 6);
glEnd();
glBegin(GL_POLYGON);// зменшення масштабу в 2 рази
glTexCoord2i(0, 0); glVertex3i(4, -2, 0);
glTexCoord2i(0, 1); glVertex3i(4, 4, 0);
glTexCoord2i(1, 1); glVertex3i(4, 4, 6);
glTexCoord2i(1, 0); glVertex3i(4, -2, 6);
```

```
glEnd();
glBegin(GL_POLYGON);// зменшення масштабу в 5 разів
glTexCoord2i(0, 0); glVertex3i(0, -2, 0);
glTexCoord2i(0, 5); glVertex3i(0, 4, 0);
glTexCoord2i(5, 5); glVertex3i(0, 4, 6);
glTexCoord2i(5, 0); glVertex3i(0, -2, 6);
glEnd();
```

Приклад 8.18 – Код функції OnTexture1() для зміни масштабу текстури



а) б)

Рисунок 8.26 – Приклад застосування текстури:
а) в режимі змішування кольору; б) із зміною масштабу текстурних еоординат

8.10.5 Використання текстур на основі растрових зображень

У попередньому пункті розглядалися засоби побудови та використання простих текстур. Однак такий підхід навряд чи задовольнить усіх бажаючих використати тривимірну комп'ютерну графіку. Значно цікавіше окремо створювати растрові зображення і надалі використовувати їх для накладання на графічні об'єкти. З якихось причин у [5] використання растрових зображень для формування текстур не описується зовсім, а у [6] описується у досить заплутаний спосіб. Наше завдання полягає у описі послідовності простих дій програміста із використання текстур із одночасним визначенням необхідних команд OpenGL.

Для завантаження графічних файлів в якості текстур у мережі Internet можна знайти різні версії функцій. В основному, вони зводяться до зчитування файлів формату bmp і формування текстурних масивів. Нижче така функція буде наведена повністю.

Після реалізації функції завантаження текстури з графічного файла використовується функція gluBuild2DMipmaps(), яка отримує зображення і формує на його основі текстуру. gluBuild2DMipmaps() має формат:

```
int gluBuild2DMipmaps(GLenum target, GLint components, GLint width,
                      GLint height, GLenum format, GLenum type, const void *data );
```

Параметрами функції є: *target* – тип цільової текстури (має бути GL_TEXTURE_2D); *components* – кількість компонентів кольору у текстурі (має дорівнювати 1, 2, 3, або 4); *width* та *height* – ширина та висота зображення тестури, відповідно; *format* – формат піксельних даних (має приймати значення: GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, або GL_LUMINANCE_ALPHA); *type* – тип даних текстури (приймає значення GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, або GL_FLOAT); *data* – покажчик на дані зображення, що зберігаються у пам’яті.

Для завантаження текстур з *.bmp-файла у програмі додатково необхідно:

1) оголосити прототипи функцій, що визначатимуться:

```
void OnTexture2();
        unsigned char* LoadTexture(char* fileName, int width, int height);
```

2) реалізувати функцію завантаження текстур з bmp-файлів:

```
unsigned char* LoadTexture(char* fileName, int width, int height)
{
        unsigned char* data;
        unsigned char R, G, B;
        FILE* file;
        file = fopen(fileName, "rb");
        data = (unsigned char*)malloc(width * height * 3);
        fread(data, width * height * 3, 1, file);
        fclose(file);
        int index;
        for (int i = 0; i < width * height; ++i)
        {       index = i * 3;
                B = data[index]; G = data[index + 1]; R = data[index + 2];
                data[index] = R; data[index + 1] = G; data[index + 2] = B;
        }
        return data;
}
```

3) реалізувати функцію застосування текстури, наприклад OnTexture2(), що викликає LoadTexture() для завантаження текстурного масиву, а потім застосовує його для накладання на об’єкт:

```
void OnTexture2()
{	GLuint texture;
	int width = 480, height = 640;
	unsigned char *data = LoadTexture("IMG_3530.bmp",width, height);
	glGenTextures(1, &texture);
	glBindTexture(GL_TEXTURE_2D, texture);
	//texture filtering
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
	glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
	gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, width, height,
	GL_RGB, GL_UNSIGNED_BYTE, data);
	glEnable(GL_TEXTURE_2D); //
	glBegin(GL_POLYGON);   //
		glTexCoord2i(0, 0); glVertex3i(4, -2, 0);
	glTexCoord2i(0, 1); glVertex3i(4, 4, 0);
		glTexCoord2i(1, 1); glVertex3i(0, 4, 0);
	glTexCoord2i(1, 0); glVertex3i(0, -2, 0);
	glEnd();
	glBegin(GL_POLYGON);
		glTexCoord2i(0, 0); glVertex3i(4, 4, 0);
	glTexCoord2i(0, 1); glVertex3i(0, 4, 0);
		glTexCoord2i(1, 1); glVertex3i(0, 4, 6);
	glTexCoord2i(1, 0); glVertex3i(4, 4, 6);
	glEnd();
	glBegin(GL_POLYGON);
		glTexCoord2i(0, 0); glVertex3i(4, -2, 6);
	glTexCoord2i(0, 1); glVertex3i(4, 4, 6);
		glTexCoord2i(1, 1); glVertex3i(0, 4, 6);
	glTexCoord2i(1, 0); glVertex3i(0, -2, 6);
	glEnd();
	glBegin(GL_POLYGON);
		glTexCoord2i(0, 0); glVertex3i(0, -2, 6);
	glTexCoord2i(0, 1); glVertex3i(0, -2, 0);
		glTexCoord2i(1, 1); glVertex3i(4, -2, 0);
	glTexCoord2i(1, 0); glVertex3i(4, -2, 6);
	glEnd();
	glDisable(GL_TEXTURE_2D);	//
	free(data);
}
```

Результат виконання визначеного вище код для завантаженої текстури показаний на Рис. 8.27.

8.10.6 Використання квадратичних об’єктів

Приклади з підрозділів 8.8.4 і 8.8.5 мають досить об’ємний код, тому одразу виникає бажання застосувати текстуру до складеного об’єкта, що будуєть-

ся однією функцією. Але під час побудови складеного об’єкта ми не матимемо змоги вказувати координати текстури (як і координати вершин об’єкта). З цієї причини необхідно шукати інші шляхи.

Рисунок 8.27 – Приклад застосування текстури, завантаженої з bmp-файла

Таким окремим шляхом накладання текстур на складені об’єкти є використання так званих квадратичних об’єктів, що у OpenGL описують поверхні другого порядку. Таким поверхням у OpenGL відповідає ряд команд:

```
// побудова циліндра
void gluCylinder(GLUquadricObj *qobj,GLdouble baseRadius,GLdouble topRadius, GLdouble height, GLint slices,GLint stacks );
```

де *baseRadius* – радіус основи циліндра, *topRadius* – радіус верхньої основи, *height* – висота, *slices* – кількість дискретних складових навколо осі *Z*, *stacks* – кількість дискретних складових вздовж осі *Z*. Параметри сфери є схожими.

```
// побудова сфери
void gluSphere(GLUquadricObj *qobj, GLdouble radius, GLint slices,GLint stacks);
```

Параметри сфери є схожими. *radius* є радіусом сфери.

```
// побудова диска
void gluDisk(GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops );
```

Серед параметрів – внутрішній та зовнішній радіуси (*innerRadius* та *outerRadius* відповідно)

// побудова сегменту диска

void gluPartialDisk(GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);

Характерними параметрами є loops – кількість концентричних кіл – елементів дискретизації диска, початкове та кінцеве значення кутів (*startAngle* та *sweepAngle*, відповідно).

Першим параметром усіх квадратичних функцій виступає покажчик на квадратичний об'єкт, що має тип GLUquadricObj. Квадратичні об'єкти створюються за допомогою функції gluNewQuadric(). Встановлення режиму накладення текстур на квадратичні об'єкти забезпечує функція gluQuadricTexture():

void gluQuadricTexture(GLUquadricObj *quadObject, GLboolean textureCoords);

де перший параметр – покажчик на квадратичний об'єкт, другий – ознака необхідності генерації текстурних координат (приймає значення GL_TRUE або GL_FALSE).

У даному прикладі використовується функція побудови диска, до якого саме і застосовується текстурне зображення, завантажуване з файла. приклад 8.19 містить код, а рис. 8.28 демонструє результат виконання програми.

```
void OnTexture3()
{       int width = 480, height = 640;
        unsigned char* data = LoadTexture("IMG_3530.bmp", width, height);
// оголошення покажчика на квадратичний об'єкт
        GLUquadricObj* m_quadObj;
        m_quadObj = gluNewQuadric();        // створення нового покажчика
// дозвіл на використання текстур
        gluQuadricTexture(gluNewQuadric(), GL_TRUE);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_TEXTURE_2D);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, width, height,
                GL_RGB, GL_UNSIGNED_BYTE, data);
// побудова квадратичного об'єкта
        gluDisk(m_quadObj, 0.0, 5.0, 80, 80);
        glDisable(GL_TEXTURE_2D);
}
```

Приклад 8.19 – Застосування текстури до квадратичного об'єкта (диск)

Рисунок 8.28 – Приклад застосування текстури, завантаженої з bmp-файла для накладання на квадратичний об’єкт (диск)

8.11 Реалізація ефектів освітлення

8.11.1 Світло та освітлення у OpenGL

Для створення реалістичних зображень необхідно визначати як властивості самого об’єкта, так і властивості середовища, у якому він знаходиться. До першої групи властивостей належать способи нанесення текстур, визначення прозорості об’єкта та властивостей матеріалу, з якого зроблено об’єкт. До другої групи належать кількість та властивості джерел освітлення, рівень прозорості середовища, модель джерел освітлення.

Апроксимація світла та освітлення реалізовані у OpenGL у такий спосіб, що світло можна розкласти на червоний, зелений та синій компоненти у відповідності до формату RGB. Колір джерела світла характеризується сумою червоного, зеленого та синього компонентів світла, що випромінюється джерелом, а матеріал поверхні у свою чергу характеризується відсотками червоного, зеленого та синього компонентів, віддзеркалених у різних напрямках [4].

У моделі освітлення OpenGL світло на сцені виходить з декількох джерел, кожен з яких може бути увімкненим та вимкненим. Джерела світла можуть виходити з певного напрямку або положення, інші – бути розсіяними по сцені.

У моделях освітлення OpenGL джерела світла мають впливати на поверхні, що поглинають або відбивають світло. Матеріал може випромінювати власне світло (наприклад, фара автомобіля), може відбивати або розсіювати частину світла в усіх напрямках (поверхня дзеркала).

Модель освітлення OpenGL розглядає освітлення у складі чотирьох незалежних компонентів: розсіяного (або фонового) світла (ambient), дифузної (diffuse), дзеркальної (specular) та емісійної (emission) складових.

Розсіяне освітлення формується світлом, напрям якого неможливо визначити. Зокрема освітлення будь-якої кімнати має значну складову розсіяного світла з тієї причини, що значна частина світла, яке досягає ока людини, перед тим була віддзеркалена великою множиною поверхонь.

Дифузна складова є світлом, що походить з одного напрямку, тому ця складова є більш яскравою, коли падає прямо на поверхню, і менш яскравою, коли світло сковзає поверхнею. Як тільки таке світло стикається з поверхнею, воно рівномірно розсіюється в усіх напрямках і зберігає рівномірну яскравість незалежно від положення спостерігача. Будь-яке світло, що походить з певного місця або напрямку, міститиме дифузну складову.

Дзеркальна складова походить з певного напряму і під час зіткнення з поверхнею віддзеркалюється у певному напрямку. У доповнення до розсіяної, дифузної та дзеркальної складових, колір матеріалу може бути емісійним, таким, що імітує світло, яке надходить від об'єкта. У моделі освітлення OpenGL емісійне світло поверхні додає яскравості об'єкта, але на таку емісійну складову не можуть впливати інші джерела світла. Емісійна складова не додає додаткового світла у загальну сцену.

8.11.2 Визначення джерел світла

Джерела освітлення OpenGL-програм мають декілька властивостей: колір, положення та напрямок і встановлюються за допомогою команди *glLight*()*:

```
void glLight[i f] (GLenum light, GLenum pname, GLfloat param);
void glLight[i f] (GLenum light, GLenum pname, GLfloat *params);
```

Параметр *light* визначає ідентифікатор джерела освітлення. Ідентифікатори знаходяться у діапазоні імен GL_LIGHT0 – GL_LIGHT7. Таким чином кількість джерел освітлення зазвичай обмежується вісьмома.

Для роботи з командою *glLight*()* необхідно задати значення параметра, а потім встановити його для необхідного джерела світла.

Параметр *pname* означає вибір параметра, значення якого встановлюється у третьому аргументі – *param* (або *params*). У таблиці 8.10 вказуються та коментуються характерні значення *pname*.

Таблиця 8.10 – Опис значень параметра *pname* команди *glLight*()*

Значення параметра	Коментар	Значення за замовчуванням
GL_AMBIENT	інтенсивність розсіяного (фонового) освітлення у форматі RGBA	(0.0, 0.0, 0.0, 1.0)
GL_DIFFUSE	інтенсивність дифузного освітлення у форматі RGBA	(1.0,1.0,1.0,1.0)- LIGHT0; (0.0,0.0,0.0,1.0) – інші джерела
GL_SPECULAR	інтенсивність дзеркального відбиття у форматі RGBA	(1.0, 1.0, 1.0, 1.0) – для джерела LIGHT0 та (0.0, 0.0, 0.0, 1.0) – для інших джерел
GL_POSITION	положення джерела світла – у координатах (x,y,z,w)	(0.0, 0.0, 1.0, 0.0)
GL_SPOT_DIRECTION	Напрям освітлення (прожектора) – у координатах (x,y,z,w), визначається якщо GL_SPOT_CUTOFF не рівний 180	(0.0, 0.0, -1.0, 1.0)
GL_SPOT_EXPONENT	експоненціальний розподіл інтенсивності у світловому конусі прожектора – ціле або дійсне число у діапазоні [0, 128]	0.0 (розсіяне світло)
GL_SPOT_CUTOFF	половина максимального кута розбіжності світлового потоку - ціле або дійсне значення у діапазоні [0, 90]	180 (максимальний кут – для розсіяного світла)
GL_CONSTANT_ATTENUATION	постійний коефіцієнт загасання	1.0
GL_LINEAR_ATTENUATION	лінійний коефіцієнт загасання	0.0
GL_QUADRATIC_ATTENUATION	квадратичний коефіцієнт загасання	0.0

Під час розробки OpenGL-програм джерела світла звичайно визначаються у функціях, що забезпечують ініціацію параметрів роботи програми. Як тільки відповідні джерела освітлення визначені, вони можуть бути увімкнені і використані. Загальна освітленість сцен OpenGL-програми вмикається командою *glEnable()*:

glEnable(GL_LIGHTNING);

Після увімкнення світла можна окремо активізувати необхідні раніше визначені джерела освітлення, наприклад:

glEnable(GL_LIGHT0);

Визначення параметрів джерела світла показує приклад 8.20:

```
        GLfloat light_ambient[]={1.0,0.0,1.0,1.0};
        GLfloat light_diffuse[]={1.0,0.0,1.0,1.0};
        GLfloat light_specular[]={1.0,1.0,1.0,0.0};
        GLfloat light_position[]={1.0,1.0,1.0,0.0};
        glLightfv (GL_LIGHT0, GL_POSITION, light_ambient);
        glLightfv (GL_LIGHT0, GL_DIFFUSE, light_diffuse);
        glLightfv (GL_LIGHT0, GL_SPECULAR, light_specular);
        glLightfv (GL_LIGHT0, GL_POSITION, light_position);
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
        glEnable (GL_LIGHTING);
glEnable (GL_LIGHT0);
```

Приклад 8.20 – Визначення параметрів світла

Як видно з прикладу, OpenGL дозволяє асоціювати з джерелом освітлення три різних параметри: GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR. Параметр GL_AMBIENT належить до RGBA-інтенсивності розсіяного світла, що додається до сцени певним джерелом. Інший параметр – GL_DIFFUSE найбільше відповідає тому, що називають кольором світла. Він встановлює колір дифузного світла, що додається певним джерелом у сцену. Третій параметр – GL_SPECULAR впливає на колір найбільш яскраво освітленої частини об'єктів (наприклад, відблиск на поверхні скляної пляшки). Для досягнення найбільшої реалістичності слід встановлювати GL_SPECULAR рівним GL_DIFFUSE [4].

Джерело світла може розміщуватися нескінченно далеко або близько відностно сцени. Нескінченно далекі джерела освітлення належать до направлених (directional) джерел, при цьому промені світла можна вважати паралельними. Саме таким джерелом можна вважати Сонце. Другий тип називають позиційним (positional) з тієї причини, що його точне положення на сцені визначає напрям променів світла. Прикладом позиційного джерела є настільна лампа.

Для реальних джерел освітлення інтенсивність світла зменшується зі збільшенням відстані від джерела. Для направленого світла, що знаходиться нескінченно далеко, немає сенсу вмикати загасання. Загасання світла використовується для позиційних джерел. OpenGL забезпечує загасання світла відповідно до такої формули:

$$Коефіцієнт\_загасання = \frac{1}{k_c + k_l d + k_q d^2};$$

де d – відстань між положенням джерела світла та вершиною,

k_c = GL_CONSTANT_ATTENUATION, k_l = GL_LINEAR_ATTENUATION,

k_q = GL_QUADRATIC_ATTENUATION.

Ці параметри можна встановити у різні, відмінні від замовчуваних, значення, наприклад:

```
GLfloat spot_direction[]={-1.0,-1.0,0.0};
glLightfv (GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightfv (GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightfv (GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 1.5);
```

Позиційне джерело світла можна примусити функціонувати як прожектор (spotlight). Це дозволить обмежити конусом випромінюване світло і простерігати лише зону, обмежену конусом. Для створення прожектора необхідно визначити межі конуса світла. За межами конусу світло не розповсюджуватиметься. За замовчуванням контролюючий параметр GL_SPOT_CUTOFF дорівнює 180 градусів. Це як раз означає розповсюдження світла в усіх напрямках. Для визначення прожектора також необхідно задавати напрям світла прожектора – вісь конуса світла. Значення кута і прожектора та його вісь задаються такою послідовністю команд:

```
glLightfv (GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
glLightfv (GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

За замовчуванням напрям прожектора встановлюється у (0.0,0.0,-1.0), що означає випромінення світла вздовж негативного напрямку осі *Z*. Цей фактор необхідно враховувати під час визначення джерел світла.

Інтенсивність розповсюдження світла всередині конуса можна контролювати, встановлюючи коефіцієнт загасання, описаний вище, також, за допомогою параметра GL_SPOT_EXPONENT, для керування концентрацією світла всередині конуса. Інтенсивність світла має найвище значення в центрі конуса і зага-

сає за формулою $cos^{e}\alpha$, де α – кут між віссю та напрямом від джерела світла до освітлюваної вершини.

Як вже зазначувалося під час визначення команди *glLight*()*, у програмі можна визначити до 8-ми джерел світла. Для цього необхідно вказати відповідні ідентифікатори, наприклад:

```
glLightfv (GL_LIGHT1, GL_DIFFUSE, light_diffuse);
glLightfv (GL_LIGHT1, GL_POSITION, light_position);
glEnable  (GL_LIGHT1);
```

Джерело світла можна формувати одночасно з об'єктом сцени або камерою і, таким чином, прив'язати його до об'єкта або камери. З іншого боку, можна сформувати стаціонарне джерело світла, що знаходитиметься на місці поки інші об'єкти переміщуються.

Розглянемо ще один приклад. Припустимо, що у програмі необхідно визначити джерело світла із властивостями прожектора. У точці розташування джерела світла розташуємо сферу, що на сцені позначатиме джерело освітлення, а на певній відстані від джерела освітлення побудуємо диск. Кут конуса світла прожектора регулюватиметься за допомогою горизонтальної смуги прокручування головного вікна програми. Текст коду обробника OnLightSpot() наведено у прикладі 8.21, а її результат дії програми – на рис. 8.29.

Зазначимо, що попередньо встановлено чорний колір фону:

```
        void OnLightSpot()
{ glClearColor(0.0, 0.0, 0.0, 0.0);
          glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
          glScalef(0.5f, 0.5f, 0.5f);
          glTranslatef(3.0, 3.0, 0.0);                          // переміщення сцени
          GLfloat emission[] = { 1.0,0.0,0.0,0.0 };             // червоний колір емісії
          glMaterialfv(GL_FRONT, GL_EMISSION, emission);        // увімкнення емісії
 // розташування джерела світла
          GLfloat light_position[] = { 0.0,0.0,10.0,1.0 };
 // білий колір світла прожектора
          GLfloat light_diffuse[] = { 1.0,1.0,1.0,1.0 };
          GLfloat spot_direction[] = { 0.0,0.0,-1.0,1.0 };      // напрям прожектора
          // встановлення кольору світла
          glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
          // встановлення положення джерела
          glLightfv(GL_LIGHT0, GL_POSITION, light_position);
 // встановлення напряму
          glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
 // встановлення кута прожектора
          glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, hspos - 50.0);
          glEnable(GL_LIGHTING);                                // загальне увімкнення освітлення фону
          glEnable(GL_LIGHT0);                  // увімкнення джерела світла
          GLUquadricObj* m_quadObj;             // оголошення квадратичного об'єкта
```

```
        m_quadObj = gluNewQuadric();            // створення квадратичного об'єкта
        gluDisk(m_quadObj, 0.0, 5.0, 80, 80);   // побудова диска
        glPushMatrix();
            glTranslatef(0.0, 0.0, 10.0);
            glutSolidSphere(0.5,30,30);         // побудова сфери
        glPopMatrix();
}
```

Приклад 8.21 – Визначення параметрів прожектора

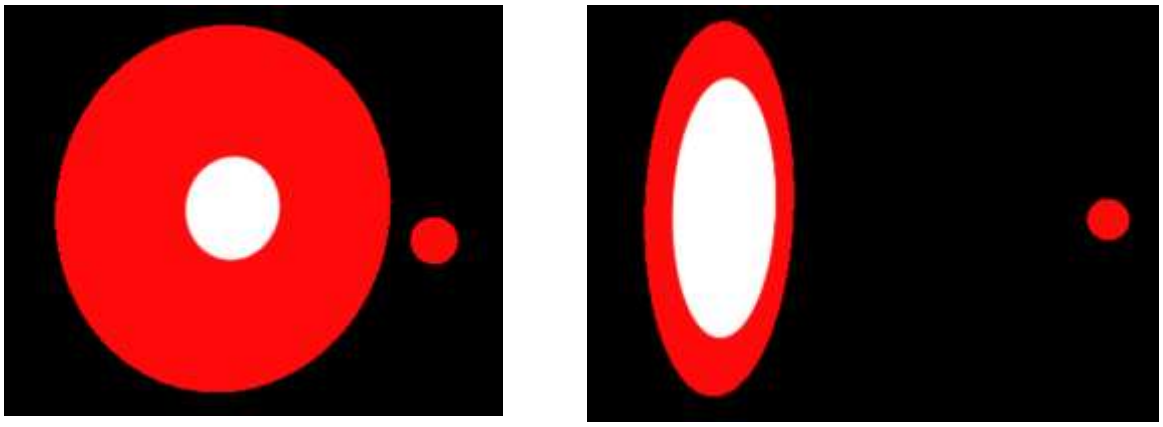


Рисунок 8.29 – Зміна освітлення диска у залежності від кута прожектора

8.11.4 Використання ефекту туману

Не є секретом, що комп'ютерні моделі є занадто чіткими і нереалістичними. Використання ефектів згладжування додає реалістичності з-за згладжування граней об'єктів. Ефект туману *(fog)* забезпечує плавне розмиття віддалених об'єктів. Взагалі, під загальною назвою "туман" у OpenGL імітують різні атмосферні ефекти: марево, імла, дим, забруднення. Ефект туману використовується тоді, коли необхідно імітувати обмежену видимість, зоктема у таких програмах, як авіасимулятори.

Якщо увімкнути режим туману, об'єкти по мірі їх віддалення від точки спостереження починають переходити у колір туману. Щільність туману можна керувати, для чого змінювати швидкість поступового змінення об'єктів по мірі змінення відстані. Туман (fog) реалізується шляхом зміни кольору об'єктів сцени у залежності від їх глибини, тобто відстані до точки спостереження. Зміна кольору відбувається або для вершин примітивів, або для кожного пікселя на етапі растеризації у залежності від реалізації OpenGL.

Туман впливає на перетворення освітлених та текстурованих об'єктів і застосовується до усіх типів графічних примітивів.

Туман має визначатися за допомогою команди *glFog*()* і вмикається за допомогою команди *glEnable()*:

```
glEnable(GL_FOG);
```

За допомогою команди *glFog*()* обирається колір туману і визначаються рівняння щільності. Увімкнений режим туману змішує колір туману з кольором вхідного фрагмента, використовуючи коефіцієнт змішування туману у залежності від глибини об'єктів, тобто відстані до точки спостереження. Зміна кольору відбувається або для вершин примітивів, або для кожного пікселя на етапі растеризації у залежності від реалізації OpenGL.

Метод обчислення інтенсивності туману у вершині можна визначити за допомогою команд

```
void glFog[if] (enum pname, T param);
void glFog[if]v (enum pname, T params);
```

Для визначення параметрів туману необхідно вказати аргумент *pname* і встановити його значення у другий аргумент – параметр papam (або params). Параметр *pname* може приймати значення, вказані у таблиці 8.10.

Якщо у параметр *param* встановлюється значення GL_FOG_MODE, необхідно визначити спосіб обчислення інтенсивності туману у окремій точці сцени. У такому випадку *param* може приймати значення, наведені у таблиці 8.11.

Таблиця 8.10 – Опис основних параметрів туману

Значення параметра	Коментар
GL_FOG_MODE	визначає формулу, за якою обчислюватиметься інтенсивність туману
GL_FOG_DENSITY	визначає щільність туману
GL_FOG_START	початок зони туману (використовується у лінійній характеристиці)
GL_FOG_END	кінець зони туману (використовується у лінійній характеристиці)

Можна навести такий приклад використання туману: нехай на сцені зображено дерево і включено режим туману. Таким чином, із збільшенням відстані до дерева дія туману посилюватиметься. Для реалізації програми необхідно реалізувати такі кроки:

1) визначити розсіяне (ambient) освітлення сцени;
2) визначити параметри туману;
3) побудувати графічні об'єкти, що складатимуть дерево.

Відстань до дерева реалізуємо, як і у попередніх прикладах, за допомогою горизонтальної смуги прокручування.

Таблиця 8.11 – Визначення інтенсивності туману

Значення параметра	Коментар
GL_EXP	Інтенсивність обчислюється за формулою $f = \exp(-(density \cdot Z))$, де *density* – щільність, Z – відстань між точкою спостереження та центром фрагмента у системі координат спостерігача
GL_EXP2	Інтенсивність обчислюється за формулою $f = \exp(-(density \cdot Z)^2)$, де *density* – щільність Z – відстань між точкою спостереження та центром фрагмента у системі координат спостерігача
GL_LINEAR	Інтенсивність обчислюється за формулою $f = \frac{end - Z}{end - start}$, де Z – відстань між точкою спостереження та центром фрагмента у системі координат спостерігача, *end* – визначається параметром GL_FOG_END, а *start* – параметром GL_FOG_START

Текст обробника *OnFog()* наводиться у прикладі 8.22, там також наведені необхідні коментарі до написаного коду. Результат роботи програми наведено на рис. 8.30.

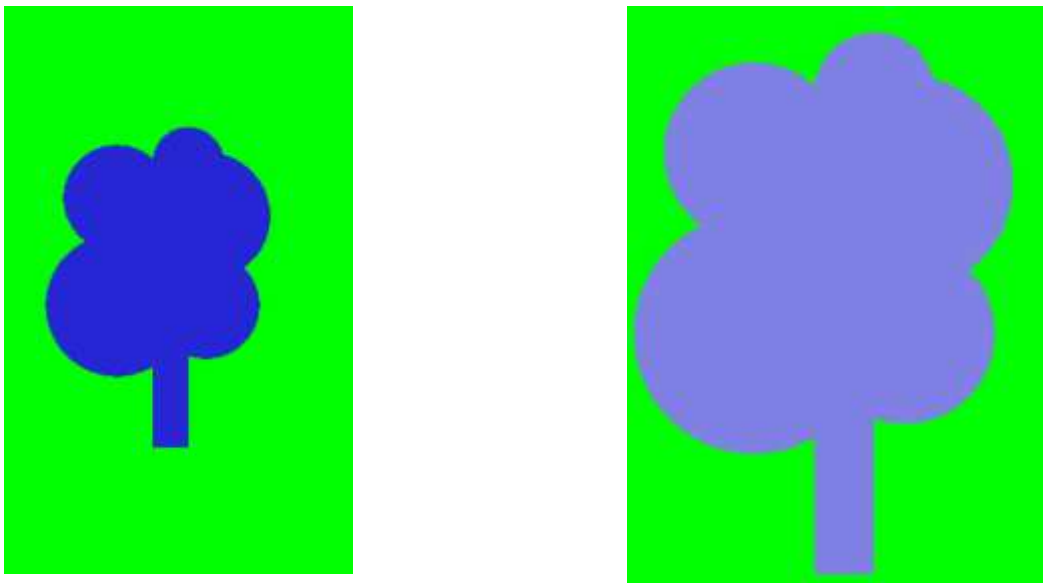
```
void OnFog()
{
        glClearColor(0.0, 1.0, 0.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // зміщення сцени (для кращого відображення)
glTranslatef(0.0, -5.0, 0.0);
// зміна положення об'єктів
        glTranslatef(0.0, 0.0, columnPos - 50.0);
        glEnable(GL_DEPTH_TEST);                    // увімкнено тест глибини
        GLfloat light_ambient[] = { 1.0,1.0,1.0,0.0 }; // колір розсіяного світла
// встановлення розсіяного світла
        glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
        glEnable(GL_LIGHTING);          // увімкнення освітлення
        glEnable(GL_LIGHT0);                        // увімкнення світла

        GLfloat FogColor[4] = { 0.0,0.0,0.8,0.0 };  // колір туману
        glFogi(GL_FOG_MODE, GL_LINEAR);             // лінійний закон туману
        glFogfv(GL_FOG_COLOR, FogColor);            // встановлення кольору туману
        glFogi(GL_FOG_DENSITY, 1.0);                // встановлення щільності туману
        glFogf(GL_FOG_START, 0.0);                  // початок зони туману
        glFogf(GL_FOG_END, 10.0);                   // кінець зони туману
        glEnable(GL_FOG);                           // увімкнення туману
```

```
GLUquadricObj* m_quadObj;                    // оголошення квадратичного об’єкта
m_quadObj = gluNewQuadric();
glRecti(0, 0, 1, 4);                         // «стовбур дерева»
glPushMatrix();                              // збереження поточної СК
glTranslatef(-1.0, 4.0, 0.0);          // переміщення у точку центра кола
gluDisk(m_quadObj, 0.0, 2.0, 80, 80);        // відображення кола
glPopMatrix();                               // повернення у базову СК
glPushMatrix();
        glTranslatef(1.5, 4.0, 0.0); gluDisk(m_quadObj, 0.0, 1.5, 80, 80);
glPopMatrix();
glPushMatrix();
        glTranslatef(1.5, 6.5, 0.0); gluDisk(m_quadObj, 0.0, 1.8, 80, 80);
glPopMatrix();
glPushMatrix();
        glTranslatef(1.0, 8.0, 0.0); gluDisk(m_quadObj, 0.0, 1.0, 80, 80);
glPopMatrix();
glPushMatrix();
        glTranslatef(-1.0, 7.0, 0.0); gluDisk(m_quadObj, 0.0, 1.5, 80, 80);
glPopMatrix();

}
```

Приклад 8.22 – Код обробника із реалізацію туману



а) б)

Рисунок 8.30 – Реалізація ефекту туману
(а – об’єкт віддалений у зону туману, б – наближений об’єкт)

8.11.5 Прозорість об’єктів та її використання

У реальному світі існує багато прозорих об’єктів – віконне скло, прозорий посуд тощо. З практичної точки зору важливим є вміння створювати такі прозорі об’єкти. Бібліотека OpenGL надає можливість механізму роботи з напівпрозорими об’єктами. Прозорість реалізується за допомогою змішування кольо-

рів (blending). Алгоритм змішування комбінує кольори вхідних пікселів (тобто "кандидатів" на поміщення у буфер кадру) з кольорами відповідних пікселів, що вже зберігаються у буфері. Для змішування використовується четверта компонента кольору – альфа-компонента, тому цей режим інакше називають альфа-змішуванням. Програма може керувати інтенсивністю альфа-компоненти саме так, як інтенсивністю основних кольорів, тобто задавати значення інтенсивності для кожного пікселя або кожної вершини примітиву.

Режим змішування вмикається за допомогою команди *glEnable()*:

```
glEnable(GL_BLEND);
```

Параметри змішування визначаються командою *glBlendFunc()*:

```
void glBlendFunc(enum src,enum dst);
```

Параметр *src* визначає, яким чином отримується коефіцієнт k_1 початкового кольору пікселя, а параметр *dst* задає спосіб отримання коефіцієнта k_2 для кольору у буфері кадру. Сумарний колір розраховується за формулою:

$$res=c_{src}\cdot k_1+c_{dst}\cdot k_2,$$

де c_{src} – колір початкового пікселя, c_{dst} – колір пікселя у буфері кадру. Зазначимо, що усі перераховані параметри (res, k_1, k_1, $c_{src,}$ c_{dst}) є чотирьохкомпонентними RGBA-векторами). Найбільш використовувані значення аргументів *src* и *dst* наведені у таблиці 8.12.

Таблиця 8.12 – Деякі значення констант режимів змішування

Значення параметра	Коментар
GL_SRC_ALPHA	$k=(A_s,A_s,A_s,A_s)$
GL_SRC_ONE_MINUS_ALPHA	$k=(1,1,1,1)-(A_s,A_s,A_s,A_s)$
GL_DST_COLOR	$k=(R_d,G_d,B_d)$
GL_ONE_MINUS_DST_COLOR	$k=(1,1,1,1)- (R_d,G_d,B_d,A_d)$
GL_DST_ALPHA	$k=(A_d,A_d,A_d,A_d)$
GL_DST_ONE_MINUS_ALPHA	$k=(1,1,1,1)-(A_d,A_d,A_d,A_d)$
GL_SRC_COLOR	$k=(R_s,G_s,B_s)$
GL_ONE_MINUS_SRC_COLOR	$k=(1,1,1,1)- (R_s,G_s,B_s,A_s)$

У таблиці 8.12 R_s, G_s, B_s, A_s означають відповідні кольори початкового пікселя (пікселя-джерела (source)), R_s, G_s, B_s, A_s – кольори цільового пікселя (destination).

Припустимо, що необхідно реалізувати виведення прозорих графічних об’єктів. Коефіцієнт прозорості задається альфа-компонентою кольору, для непрозорих об’єктів альфа-компонента дорівнює 1, для абсолютно прозорих, тобто невидимих, дорівнює 1. Для активізації прозорих властивостей необхідно увімкнути режим прозорості та визначити параметри змішування:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Напівпрозорі об’єкти можна задати визначивши альфа-параметр кольору, вищий за 0. У прикладі 8.23 наведено варіант обробника *OnTransparency()* із прозорими об’єктами – двома дисками (на основі квадратичних команд) та прямокутником. Результат використання такого обробника наведено на рис. 8.31.

```
void OnTransparency()
{       glTranslatef(-1.0, -4.0, 0.0);
        glRotatef(360.0 * vspos / 100, 1, 0, 0);                    // обертання навколо осі X
        glTranslatef(0.0, 0.0, hspos - 50.0);
        glEnable(GL_DEPTH_TEST);                                    // увімкнення тесту глибини
        glEnable(GL_BLEND);                                         // увімкнення режиму прозорості
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // режим змішування
        GLUquadricObj* m_quadObj;
        m_quadObj = gluNewQuadric();
        glColor4f(1.0, 0.0, 0.0, 0.9);                  // альфа-параметр встановлено у 0.9
        glRecti(0, 0, 4, 4);                    // прямокутник
        glColor4f(0.0, 1.0, 0.0, 0.9);
        glPushMatrix();
        glTranslatef(-1.0, 4.0, 0.0);
        gluDisk(m_quadObj, 0.0, 2.0, 80, 80);           // диск
        glPopMatrix();
        glColor4f(0.0, 1.0, 1.0, 0.9);
        glPushMatrix();
        glTranslatef(1.5, 4.0, 0.0);
        gluDisk(m_quadObj, 0.0, 1.5, 80, 80);           // диск
        glPopMatrix();
}
```

Приклад 8.23 – Приклад реалізації прозорих об’єктів

Якщо у сцені графічного виведення є декілька прозорих об’єктів, що перекриваються, коректний вивід можна гарантувати тільки за таких умов:

1) усі прозорі об’єкти виводяться після непрозорих;

2) виведення прозорих об’єктів має виконуватися упорядковано за зменшенням глибини, тобто виводитися, починаючи з об’єктів, що є найбільш віддаленими від спостерігача.

У OpenGL команди обробляються у порядку їх надходження, тому для реалізації перелічених вимог достатньо розставити виклики команд *glVertex*()* у відповідному порядку.

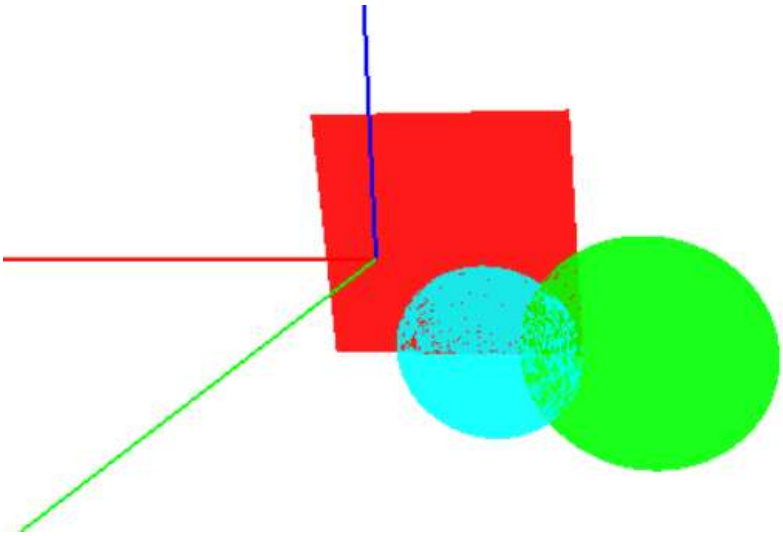


Рисунок 8.31 – Виведення сцени із прозорими об’єктами

У заключенні розділу слід сказати, що, звичайно, дуже складно на 58-ти сторінках розглянути технологію OpenGL і проілюструвати їх прикладами. Тому увагу звернуто на найбільш прості і, водночас, цікаві аспекти технології. Натомість, на наш погляд, вони дають уявлення про графічне програмування і забезпечують підґрунтя для більш ретельного вивчення.

8.12 Контрольні завдання

1. Навести основні характерні властивості графічної бібліотеки OpenGL.
2. Пояснити функціонування конвеєру візуалізації OpenGL.
3. Пояснити організацію графічного виведення у звичайних .NET-програмах.
4. Пояснити особливості організації графічного виведення у .NET -програмах із використанням команд OpenGL.
5. Пояснити організацію конструктора мінімальної OpenGL-програми, побудованої основі .NET -проекту.
6. Пояснити перетворення систем координат OpenGL під час виконання команд візуалізації.
7. Пояснити способи накладення текстур у OpenGL-програмах.
8. Реалізувати мінімальну OpenGL-програму на основі .NET -проекту.
9. Реалізувати обробник OpenGL-програми для побудови об’єктів типу куб, піраміда, конус на основі базових примітивів OpenGL.

10. Реалізувати обробник OpenGL-програми із побудовою об’єкта типу робот, забезпечите керування окремими суглобами роботами за допомогою регуляторів.

11. Реалізувати обробник OpenGL-програми із побудовою об’єкта типу ракета, забезпечити накладення текстур на поверхні ракети.

12. Реалізувати програму із прожектором – джерелом освітлення, забезпечити регулювання його робочих параметрів (розташування, напрям, кут освітлення).

13. Реалізувати програму із ефектом туману, забезпечити регулювання параметрів туману.

ПІСЛЯМОВА

Епоха Просвітництва (кінець XVII – початок XIX століть) характерна появою надзвичайно обдарованих вчених, освічених у багатьох галузях знань. Їх, згодом, було названо вченими-енциклопедистами. Стан суспільства і наук того часу надавав можливостей одній людині засвоювати практично все наукове надбання людства і ставати універсальним вченим майже з усіх наук.

Бурхливий розвиток цивілізації у XIX та, особливо XX столітті призвів до появи і розвитку багатьох напрямків і спеціалізацій наукової діяльності. Нині бажання стати універсальним спеціалістом з усіх галузь знань здається неможливим, хоча протягом життя людина прагнутиме до освоєння все нових обріїв науки і знання.

Галузь програмування виникла у 60-х роках вже минулого двадцятого століття. Раніше програмування було прерогативою вузького кола спеціалістів, що працювали у промисловій, військовій, космічній галузях або займалися статистичними аспектами економіки. Входження комп'ютерів у широкий вжиток призвело до поширення ремесла програміста і розвитку програмування.

Написання книжок з програмування тільки з першого погляду здається легкою справою. Автори багатьох навчальних видань намагаються охопити практично усі аспекти тематики. Їх підручники сягають 800-1000 сторінок і потім поважно займають місце на книжкових полицях, збираючи пил. Щоправда, знання програмних технологій мають погану звичку швидко застарівати і тоді поважний фоліант стає надійним засобом розколювання горіхів. Це, звичайно, жарт, але, перебираючи книжкову полицю, завжди можна знайти багато книжок, що не виправдали сподівань читача і витрачених ним грошей.

У нашому випадку охоплення усіх технологій програмування – майже безнадійна справа. Жоден підручник, навчальний посібник або інше видання не охоплює усі аспекти Visual C++. Найновіша інформація зазвичай з'являється у формі технічних звітів і керівництв та часто не стає предметом книговидання. Більше того, певні напрями програмування непомітно з'являються і безслідно зникають так і не отримавши розвитку. Таким чином, надійними, з точки зору опису, методами програмування є ті, що вже не блищать новизною, але пройшли випробовування у реальному житті.

У ході написання посібника автор намагався систематично викласти основи створення Windows-програм засобами Visual C++, щоразу закладаючи практичне підґрунтя у вигляді, в основному, власноруч розроблених програм. Стислий обсяг книжки, звичайно, дозволив розглянути лише прості і короткі

програми, але на їх основі читач, отримавши паперовий дороговказ, зможе, власноруч розробляти програми відповідно до власних потреб. Автор має надію, що його праця не стане марно витраченим часом і принесе хоч невелику, але конкретну користь у вивченні технологій програмування.

ПЕРЕЛІК ПОСИЛАНЬ

1. Цимбал О.М. Технології програмування: Visual C++. – Харків: ХНУРЕ, 2006. – 336 с.
2. Stephen R.G. Fraser. Pro Visual C++/CLI and the .NET 2.0 Platform, 2006.
3. Marcus Heege. Expert C++/CLI .NET for Visual C++ Programmers, 2007.
4. Tsimbal A.M. System software. Summary of lectures. – Kharkiv, KhNURE, 2003. – 194 p.
5. OpenGL programming guide : the official guide to learning OpenGL, version 4.3 / Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane ; the Khronos OpenGL ARB Working Group. – Eighth edition, 2013. – 938 p.
6. Програмування комп'ютерної графіки та мультимедійні засоби: навчальний посібник / Л.М. Журавчак, О.М. Левченко. – Львів, Видавництво Львівської політехніки, 2019. – 276 с.
7. Gordon Hogenson. Foundations of C++/CLI. The Visual C++ Language for .NET 3.5, 2008.
8. Проценко В.С., Чаленко П.Й., Ставровський А.Б. Техніка програмування мовою Сі. – К.: Либідь, 1993. – 224 с.
9. Комп'ютерна графіка: навчальний посібник / Є.В. Бородавка, О.О. Терентьєв. Київ: КНУБА, 2023. 132 с.
10. https://www.opengl.org/resources/libraries/glut/glut_downloads.php
11. https://learn.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=net-9.0
12. https://learn.microsoft.com/en-us/dotnet/api/system.threading.semaphore?view=net-9.0
13. https://learn.microsoft.com/en-us/dotnet/api/system.threading.eventwaithandle?view=net-9.0
14. https://learn.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads

ПРЕДМЕТНИЙ ПОКАЖЧИК

НАВЧАЛЬНЕ ВИДАННЯ

ЦИМБАЛ Олександр Михайлович
БРОННІКОВ Артем Ігорович

ТЕХНОЛОГІЇ ПРОГРАМУВАННЯ:
C++/CLI в Microsoft Visual Studio

для студентів
спеціальності G7 Автоматизація, комп'ютерно-інтегровані
технології та робототехніка

Формат 60x84 1/16. Папір офсет. Друк цифровий
Умов. друк. арк. ___ Наклад 200 прим. Зам ___

Видавництво та друк
ФОП Іванченко І.С.
пр. Тракторобудівників, 89-а/62, м. Харків, 61135
тел.: +38(050/093) 40-243-50
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготовників та розповсюджувачів видавничої продукції ДК №4388 від 15.08.2012 р.
www.monograf.com.ua