ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Підвищення продуктивності динамічних інтерфейсів за допомогою архітектури однопрохідного IMGUI з адаптивним компонуванням»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Артем ПЕРЕПЕЛИЦЯ
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-63
Артем ПЕРЕПЕЛИЦЯ

Керівник: В’ячеслав ТРЕЙТЯК
канд. техн. наук.

Рецензент: _____
науковий ступінь, вчене звання *Ім’я, ПРІЗВИЩЕ*

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Перепелиці Артему Олександровичу

1. Тема кваліфікаційної роботи: «Підвищення продуктивності динамічних інтерфейсів за допомогою архітектури однопрохідного IMGUI з адаптивним компонуванням»

керівник кваліфікаційної роботи В’ячеслав ТРЕЙТЯК, канд. техн. наук,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» жовтня 2025 р. № 467.

2. Строк подання кваліфікаційної роботи «19» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, методи створення конвеєру за кадр, методи ідентифікування елементів.

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

1. Порівняльний аналіз можливостей існуючих UI бібліотек в Unity.
2. Аналіз алгоритмів хешування з якісною рівномірністю для коротких ключів.
3. Розробка та валідація алгоритму хеш-карти з підтримкою модифікацій.
4. Розробка та валідація архітектури однопрохідного IMGUI з адаптивним компонуванням.

5. Перелік ілюстративного матеріалу: *презентація*

1. Актуальність роботи — існуючі UI бібліотеки в Unity.
2. Актуальність роботи — Immediate-mode GUI.
3. Практичний результат.
4. Новий алгоритм UI конвеєра з однією фазою.
5. Алгоритм щокадрового адаптивного компонування.
6. Порівняльний аналіз кількості коду.
7. Порівняльний аналіз стабільності кадрів.
8. Порівняльний аналіз CVaR 1% кількості циклів, тис.

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	31.10 – 01.11.2025	
2	Порівняльний аналіз існуючих UI-бібліотек у середовищі Unity	02.11 – 03.11.2025	
3	Вивчення матеріалів для аналізу імплементацій хеш-карт для зберігання інформації про інтерфейс	04.11 – 05.11.2025	
4	Дослідження алгоритмів хешування з високою рівномірністю розподілу для коротких ключів	06.11 – 12.11.2025	
5	Експериментальний аналіз продуктивності кандидатних хеш-таблиць у репрезентативних для IMGUI операціях	13.11 – 19.11.2025	
6	Розроблення та обґрунтування архітектури бібліотеки DiVy	20.11 – 04.12.2025	
7	A/B-оцінювання продуктивності розробленої архітектури	05.12 – 12.12.2025	
8	Оформлення роботи: вступ, висновки, реферат	13.12 – 16.12.2025	
9	Розробка демонстраційних матеріалів	17.12 – 19.12.2025	

Здобувач вищої освіти _____ Артем ПЕРЕПЕЛИЦЯ
(підпис)

Керівник
кваліфікаційної роботи _____ В’ячеслав ТРЕЙТЯК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 95 стор., 5 табл., 40 рис., 36 джерел.

Мета роботи – підвищити стабільність продуктивності динамічних інтерфейсів кінцевих застосунків у Unity через архітектуру Immediate-mode GUI, що підтримує сучасну стилізацію, адаптивне компонування та працює в типовому ігровому циклі без проміжних шарів.
Об’єкт дослідження – процес побудови й відтворення графічних інтерфейсів користувача в Unity для динамічних інтерактивних застосунків.
Предмет дослідження – архітектурні принципи, алгоритмічні та інженерні засоби реалізації однопрохідної IMGUI-бібліотеки з адаптивним компонуванням і розширеною стилізацією.

Короткий зміст роботи: теоретично обґрунтовано засади IMGUI для динамічних застосунків у Unity та сформульовано вимоги до архітектури; запропоновано однопрохідну схему кадру й створено придатну до безпосереднього використання бібліотеку. Проведено експерименти (заміри швидкодії, моделювання системних параметрів, A/B-порівняння з UI Toolkit у репрезентативних сценаріях), які підтвердили інженерну придатність рішення та показали нижчу варіативність часу кадру і менші пікові затримки у низці складних сценаріїв.

Практичне значення полягає у впровадженні однопрохідної IMGUI-архітектури для динамічних UI в Unity, що забезпечує стабільні часові характеристики на мобільних і VR-платформах. Результати дають змогу інтегрувати рішення в інтерфейси кінцевих застосунків і слугують основою для подальшої паралелізації етапів розкладки/рендерингу та розширення візуального функціоналу.

КЛЮЧОВІ СЛОВА: ІНТЕРФЕЙС КОРИСТУВАЧА, IMGUI, UNITY, ХЕШ-КАРТА, АВТОМАТИЧНЕ КОМПОНУВАННЯ, GUI БІБЛІОТЕКА.

ABSTRACT

Text part of the master's qualification work: 95 pages, 40 pictures, 5 tables, 36 sources.

Aim. To improve the performance stability of dynamic end-user interfaces in Unity by means of an Immediate-mode GUI architecture that supports modern styling, adaptive layout, and operates within the standard game update loop without intermediate layers.

Object of study. The process of constructing and rendering graphical user interfaces in Unity for dynamic interactive applications. Subject of study. Architectural principles and algorithmic/engineering techniques for implementing a single-pass IMGUI library with adaptive layout and extended styling.

Summary. The work theoretically substantiates the foundations of IMGUI for dynamic Unity applications and formulates architectural requirements; it proposes a single-pass frame scheme and delivers a library ready for direct use. Experiments were conducted—performance measurements, system-parameter modeling, and A/B comparisons with UI Toolkit under representative scenarios—which confirmed the engineering viability of the solution and showed lower frame-time variability and smaller tail latencies in several complex scenarios.

Practical significance. The implementation of a single-pass IMGUI architecture for dynamic UI in Unity ensures stable timing characteristics on mobile and VR platforms. The results enable integration into end-user application interfaces and provide a basis for further parallelization of layout/render stages and for extending visual functionality.

KEYWORDS: USER INTERFACE, IMGUI, UNITY, HASH MAP, AUTOMATIC LAYOUT, GUI LIBRARY.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

A/B — двоваріантне порівняння/тест (A/B comparison).

API — Application Programming Interface, програмний інтерфейс застосунку.

AR — Augmented Reality, доповнена реальність.

CPU — Central Processing Unit, центральний процесор.

CV — коефіцієнт варіації (метрика стабільності покадрових витрат).

CVaR — Conditional Value-at-Risk, умовне очікуване перевищення; у роботі — середнє значення для верхнього 1% найгірших кадрів.

ECS — Entity Component System, архітектурний підхід до організації ігрових систем.

FPS — Frames Per Second, кадри за секунду.

GC — Garbage Collector, збирач сміття (керування пам’яттю в .NET).

GPU — Graphics Processing Unit, графічний процесор.

GUI — Graphical User Interface, графічний інтерфейс користувача.

HUD — Heads-Up Display, екранна інфопанель/накладка.

ID — Identifier, ідентифікатор елемента/контроля.

IMGUI — Immediate-mode GUI, парадигма «негайного» відтворення інтерфейсу на кожному кадрі.

RMGUI — Retained-mode GUI, парадигма зі збереженням стану між кадрами (дерево об’єктів).

SDF — Signed Distance Function, функція відстані зі знаком.

UGUI — компонентна UI-система Unity на базі Canvas.

UI — User Interface, інтерфейс користувача.

UX — User eXperience, користувацький досвід.

UXML — Unity XML, формат розмітки UI Toolkit.

USS — Unity Style Sheets, аналог CSS для UI Toolkit.

VR — Virtual Reality, віртуальна реальність.

ВСТУП

Актуальність теми дослідження. Сучасні інтерактивні застосунки та ігрові продукти висувають підвищені вимоги до інтерфейсів користувача: вони мають бути візуально насиченими, анімованими, реактивними до дій користувача та стабільними за продуктивністю на різних програмно-апаратних платформах. За цих умов зростає практична цінність підходів до побудови UI, здатних масштабуватися в динамічних сценах, витримувати інтенсивні анімаційні навантаження та забезпечувати передбачувану вартість кадру.

Unity посідає провідне місце серед інструментів розроблення ігор і симуляцій, однак на рівні готових UI-рішень розробники стикаються з низкою обмежень, коли мова йде про високодинамічні інтерфейси кінцевих продуктів, а не внутрішньо-рушійні панелі чи інтерфейси відладки. Типові труднощі охоплюють:

– високу чутливість до перебудов ієрархії елементів,

– накладні витрати на автоматичне компонування при значній варіативності станів,

– складність стабільного досягнення «плаского» однотактного проходу за кадр.

У підсумку постає потреба в архітектурі, що поєднує переваги підходу Immediate-mode з сучасними вимогами до стилізації, темізації та адаптивного компонування, зберігаючи при цьому лінійно передбачувану продуктивність.

Аналіз останніх досліджень і публікацій. У наукових і професійних джерелах описано спектр парадигм побудови UI, зокрема підходи на основі безпосереднього конструювання стану інтерфейсу щоразу під час кадру (Immediate-mode) та підходи з утриманням стану від кадру до кадру (Retained-mode). Показано, що вибір парадигми впливає на модель життєвого циклу елементів, вартість оновлення ієрархій та дисципліну керування станом. Для Unity систематизовано сильні та слабкі сторони наявних UI-систем; висвітлено їх придатність до сценаріїв з високою динамікою, наявністю складних анімацій і

потребою в точному контролі порядку рендерингу. Водночас у літературі бракує завершених архітектур IMGUI-бібліотек для Unity із підтримкою повного сучасного набору функцій інтерфейсу кінцевого продукту та з однопрохідним виконанням за кадр.

Мета роботи — підвищення стабільності продуктивності динамічних інтерфейсів за допомогою архітектури Immediate-mode GUI-бібліотеки для Unity, орієнтованої на інтерфейси кінцевих застосунків і ігор, підтримує сучасні вимоги до стилізації та адаптивного компонування і може використовуватися в типовому ігровому циклі оновлення без додаткових проміжних шарів. Надалі в тексті нова бібліотека позначається як DiVy.

Для досягнення мети передбачено *розв'язання таких завдань*:

а) спроєктувати модель побудови дерева інтерфейсу на кожному кадрі з передбачуваною вартістю операцій незалежно від кількості локальних змін;

б) реалізувати високопродуктивне адаптивне компонування елементів інтерфейсу в режимі виконання кадру;

в) забезпечити точне й маловитратне виявлення інтеракцій користувача з елементами (hit-testing, фокус, стан наведення/натискання) для негайної обробки у застосунку;

г) додати підтримку вибіркової стилізації, темізації, скруглених кутів, градієнтних заливок та інших сучасних візуальних ефектів без втрати однопрохідності;

ґ) передбачити можливість побудови окремих піддерев і обчислювально коштовних операцій у кількох потоках із безпечною синхронізацією в ігровому циклі.

Об'єктом дослідження є процес побудови та відтворення графічних інтерфейсів користувача в середовищі Unity для динамічних інтерактивних застосунків.

Предметом дослідження є архітектурні принципи, алгоритмічні та інженерні засоби реалізації бібліотеки Immediate-mode GUI з однопрохідним виконанням за кадр, адаптивним компонуванням і розширеною стилізацією.

Наукова новизна полягає у:

– формулюванні та обґрунтуванні архітектурної моделі IMGUI для Unity, що поєднує динамічне відтворення дерева елементів на кожному кадрі з підтримкою сучасної стилізації та адаптивного компонування;

– застосуванні однопрохідного підходу до оброблення подій, компонування і рендерингу інтерфейсу з фіксованою послідовністю стадій у межах одного кадру;

– пропозиції підходів мінімізації виділень пам’яті та перебудов, які дозволяють утримувати стабільну вартість кадру за умов значної варіативності станів UI.

Практична значущість полягає в отриманні інженерно придатної до використання в промислових проєктах бібліотеки, орієнтованої на динамічні, анімовані інтерфейси, з можливістю інтеграції у стандартний ігровий цикл Unity і подальшої адаптації під потреби конкретних продуктів.

Основні положення роботи були апробовані у вигляді доповідей на наукових заходах і публікацій у фахових виданнях; окремі результати інтегровані в експериментальні прототипи користувацьких інтерфейсів у середовищі Unity.

Теоретична цінність полягає у формалізації вимог до IMGUI-архітектур для динамічних UI та в узагальненні критеріїв оцінювання їхньої продуктивності й інженерної придатності. Методична цінність полягає у впорядкуванні підходів до побудови одноетапних за кадр конвеєрів оброблення подій, компонування і рендерингу. Прикладна цінність виражається у створенні бібліотеки, яку можна безпосередньо застосувати для розроблення інтерфейсів для кінцевих продуктів у Unity.

Структура роботи. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. У першому розділі подано огляд літератури та концептуальне порівняння парадигм побудови UI, виконано аналіз наявних рішень в екосистемі Unity та визначено методичні передумови оцінювання.

У другому розділі наведено порівняльне дослідження рішень для підвищення ефективності бібліотеки, обґрунтовано методи та метрики оцінювання.

У третьому розділі представлено розробку та реалізацію авторської бібліотеки DiVy та результати експериментального дослідження, включно з порівняльним аналізом відносно конкурентної бібліотеки; узагальнено отримані результати й сформульовано практичні рекомендації щодо застосування та подальшого розвитку.

1 ТЕОРЕТИЧНІ ТА МЕТОДИЧНІ ЗАСАДИ ПОБУДОВИ IMMEDIATE-MODE GUI В UNITY

1.1. Огляд літератури

Цей підрозділ узагальнює наукові та професійні джерела з проєктування графічних інтерфейсів користувача у контексті динамічних інтерактивних застосунків, окреслює ключові парадигми побудови UI, типові архітектурні рішення та відомі інженерні компроміси.

1.1.1. Парадигми побудови UI: Immediate-mode та Retained-mode

У літературі сформувалося дві базові парадигми побудови інтерфейсів:

– Immediate-mode (IMGUI): інтерфейс конструюється щоразу під час кадру шляхом прямого виклику функцій малювання і оброблення подій. Структура інтерфейсу є похідною від поточного стану застосунку; дерево елементів не зберігається між кадрами, а відновлюється заново. Такий підхід забезпечує лінійний контроль порядку відтворення і спрощує синхронізацію зі станом гри, але висуває підвищені вимоги до дисципліни керування ефемерним станом віджета (фокус, наведення курсором, перехоплення подій) [25].

– Retained-mode (RMGUI): інтерфейс описується у вигляді дерева об’єктів, яке зберігає стан між кадрами. Механізми оновлення торкаються лише змінених піддерев. Переваги — вбудовані служби компонування, стилізації, анімацій; недоліки — вартість синхронізації стану з моделлю, перебудови ієрархій, а також непрозора вартість кадру за значної варіативності станів [24].

У дослідженнях відзначено, що вимір «декларативність vs імперативність» є ортогональним до «immediate vs retained»: декларативний опис може реалізовуватися як поверх retained-mode, так і в режимі immediate з відповідним шаром інтерпретації.

1.1.2. Свідчення життєздатності IMGUI у наукових і прикладних роботах

Джерела засвідчують широке застосування IMGUI в інструментальних і наукових проєктах: від швидкого створення спеціалізованих панелей для оброблення даних у реальному часі до графічних інтерфейсів експериментальних систем [36, 28, 29, 31]. Окремі публікації демонструють прототипи, де IMGUI використовується як базова парадигма інтерфейсу для повноцінних застосунків (напр., NBUI як мінімалістична реалізація) [27]. У професійній практиці IMGUI тривалий час переважав для внутрішньо-рушійних інструментів, що пояснює дефіцит публічних звітів про інтерфейси для кінцевих застосунків: комерційні проєкти рідко розкривають такі деталі [36, 30]. Узагальнення цих робіт підкреслює: IMGUI доцільний там, де потрібні передбачувана вартість кадру, лінійний контроль порядку відтворення, тісний зв’язок з ігровим циклом і мінімізація прихованих перерахунків.

1.1.3. Життєвий цикл елементів, керування станом і модель подій

Огляди підкреслюють центральність життєвого циклу елементів для вартості оновлення [26, 24]:

– У RMGUI стан зберігається на вузлах дерева; події розповсюджуються за правилами тунелювання або підняття вгору; оновлення часто ініціюється зміною властивостей і механізмами сповіщення. Перевага — локалізація змін; ризик — дублікація стану та прихована вартість перерахувань дерева.

– У IMGUI стан елементів є ефемерним і виводиться з поточної логіки; події зчитуються з системи вводу у момент виклику функції. Це полегшує синхронізацію зі станом гри та уникає складної маршрутизації подій, однак потребує явного проєктування механізмів фокусу, повторного введення та керування змінними між кадрами.

1.1.4. Компонування та геометрія: моделі та обчислювальна вартість

Література описує декілька класів алгоритмів компонування:

– блочні моделі (stack/flow),

– гнучкі моделі (flexbox-подібні, з розв’язанням обмежень за осями) [32],

– табличні підходи,

– constraint-based системи (рішення систем нерівностей) [33].

Ключові практики: інкрементальність розрахунків, обмеження зворотних залежностей, диференційоване оновлення для стабілізації вартості кадру. Для високодинамічних інтерфейсів підкреслюють важливість обмеження глибини перерахунків і прогнозованості складності.

1.1.5. Рендеринг і конвеєр відтворення

Джерела систематизують два підходи до побудови конвеєра [34]:

– Retained-composition: збирання сцени у проміжне подання (граф сцени, дерево шарів) з подальшим злиттям шарів, формуванням партій і мінімізацією перемикань станів GPU. Перевага — агрегація; недолік — вартість підтримки структури і складність контролю порядку.

– Immediate-draw: послідовне формування команд рендерингу в одному проході, з явним керуванням порядком і формуванням партій. Перевага — передбачуваність; виклик — потреба в суворій дисципліні виділення пам’яті і уніфікації примітивів для ефективного формування партій.

В обох випадках ключовими є: мінімізація викликів малювання, контроль overdraw, використання атласів і попередньої підготовки геометрії.

1.1.6. Продуктивність і пам’ять: відомі компроміси

Огляди зводять міркування продуктивності до кількох інваріантів:

– передбачуваність вартості кадру важливіша за пікову пропускну здатність;

– локальність даних і мінімізація виділень пам’яті прямо впливають на стабільність кадру;

– детермінований порядок рендерингу спрощує відладку і покращує локальність кешу.

Для RMGUI головний ризик — непередбачувані сплески при перебудовах дерева, стилів або компонування. Для IMGUI — ризик втрати однопрохідності за недотримання дисципліни конвеєра.

1.1.7. Емпіричні свідчення щодо продуктивності розробника

Окремі автори повідомляють про значне зниження церемоніальності та прискорення ітерацій у разі застосування IMGUI:

– Скорочення обсягу коду. За свідченням Міші Меттке (автор Nuklear), експериментальна реалізація одного й того самого вікна в Nuklear (IMGUI) і Qt (RMGUI) показала приблизно утричі менший обсяг коду та кількість файлів у варіанті на IMGUI (за словами автора; результати не публіковано) [2].

– Суттєве скорочення часу розробки. Мікулаш Флорек (Lumix Engine) повідомляє, що після переходу з Qt на Dear ImGui час створення інтерфейсів зменшився «у сто разів», що він пов’язує з меншим «ритуальним» навантаженням та прямою відповідністю API робочому циклу рушія [4].

Попри анекдотичність, ці свідчення узгоджуються з гіпотезою про меншу церемоніальність IMGUI і, відповідно, вищу швидкість ітерацій у інструментальних і високодинамічних сценаріях.

З погляду інженерної продуктивності важливою є вартість коду як інтегральний індикатор церемоніальності API. Менший обсяг вихідного коду (SLOC) для реалізації функціонально еквівалентних віджетів або цілих панелей, за інших рівних умов, корелює зі скороченням часу імплементації, зниженням когнітивного навантаження під час читання та огляду, та зменшенням імовірності дефектів, зумовлених структурною складністю. Для UI-бібліотек доцільно розрізняти локальний показник (SLOC окремого віджета) та агрегований показник (сукупний SLOC цільової панелі/екрана).

1.1.8. Енергоспоживання та стабільність кадру

Поширеним є припущення, що IMGUI в середньому «дорожчий» за енергоспоживанням. Наявні вимірювання свідчать про порівнянні середні значення для IMGUI та RMGUI, причому профіль IMGUI є стабільнішим у часі: за

даними Фореста Сміта, середнє споживання енергії становило близько 31 Вт для IMGUI проти 27,5 Вт для RMGUI, але IMGUI демонстрував істотно меншу варіативність споживання [3]. Стабільність профілю є важливою для AR/VR-сценаріїв. На рисунку 1.1 в лівій частині два результати без запущених програм, всередині — програми, побудовані у парадигмі IMGUI, справа — у RMGUI.

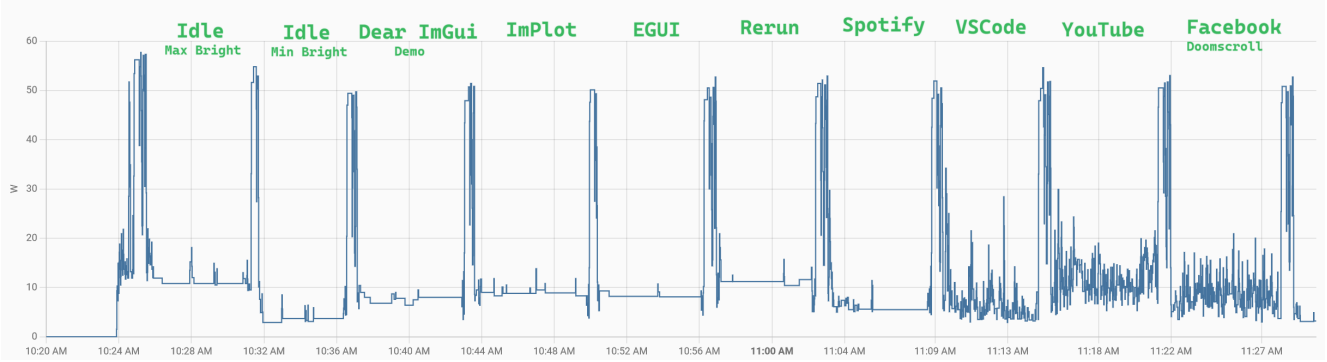


Рис. 1.1 Рівень споживання енергії при використанні різних програм.

1.1.9. AR/VR як особливий контекст для UI-архітектур

Розвиток віртуальної та доповненої реальностей зумовлює специфічні вимоги до інтерфейсів. На ринку помітна поява споживчих AR/VR-пристроїв, зокрема систем, орієнтованих на повсякденні користувацькі застосунки, що супроводжується переходом від ПК-залежних рішень до мобільних шоломів із обмеженими ресурсами CPU та GPU. Для таких систем характерна потреба у стабільній високій частоті оновлення (зазвичай ≥90 кадрів на секунду), а короткочасні затримки відчутно погіршують досвід і можуть спричиняти вестибулярний дискомфорт.

Мікрорухи голови та особливості відображення (кривизна, зміни дистанції до очей, перспектива) призводять до постійної перебудови геометрії UI. За цих умов пріоритетами стають стабільність часу кадру та передбачувана обчислювальна складність навіть за частих змін. Практика показує, що моделі, які спираються на статичні дерева з рідкісними інвалідаціями, гірше узгоджуються з такими вимогами; натомість стабільна швидкодія за умов постійних змін дерева елементів набуває вирішального значення. У цьому сенсі AR/VR виступає «стрес-тестом» для будь-якої UI-архітектури, підсвічуючи її сильні та слабкі сторони.

1.1.10. Декларативні API як індикатор запиту на «IMGUI-подібну» взаємодію

Поширення декларативних бібліотек у широкій розробницькій практиці свідчить про перевагу розробників до API, що наближають досвід до IMGUI: опис бажаного стану UI, швидка реакція на зміну стану, мінімум явного керування довгоживучим деревом. У цьому підрозділі такі приклади використовуються винятково як індикатор користувацьких очікувань від API у середовищах RMGUI.

1.1.11. Узагальнення і виявлені прогалини

а) Теоретична база для IMGUI та RMGUI усталена: окреслено моделі стану, подій, компонування та рендерингу, а також типові компроміси продуктивності.

б) Емпіричні свідчення вказують на меншу церемоніальність IMGUI (менший обсяг коду й час реалізації) та порівнянне середнє енергоспоживання зі стабільнішим профілем у IMGUI.

в) Декларативні API корисні як свідчення запиту на простіші, ближчі до IMGUI, інтерфейси взаємодії для розробника; робити з цього висновки про ефективність їхніх внутрішніх гібридних механізмів некоректно без окремих експериментів.

г) Бракує завершених описів архітектур IMGUI-бібліотек, орієнтованих саме на інтерфейси кінцевих застосунків (розширена стилізація, темізація, анімації, адаптивне компонування) із гарантованою однопрохідністю за кадр і з відтворюваними інженерними метриками.

1.2. Опис наявних бібліотек Unity

Цей підрозділ подає систематизований огляд трьох поколінь UI-рішень в екосистемі Unity: історичної Unity IMGUI, подальшої UGUI та найновішої UI Toolkit. Розгляд фокусується на архітектурних засадах, можливостях компонування й стилізації, характерних обмеженнях продуктивності та інженерній придатності для динамічних інтерфейсів.

1.2.1. Unity IMGUI

Unity IMGUI є першою UI бібліотекою у рушії і реалізує підхід Immediate-mode. Її раннє походження зумовило низку архітектурних рішень, які нині обмежують розширюваність і продуктивність системи, унаслідок чого бібліотеку було фактично законсервовано, а розвиток UI у Unity продовжено в альтернативній парадигмі.

Функціональні можливості. Підтримується базовий набір інтерактивних віджетів (наприклад, Button, RepeatButton, TextField, Toggle). Конструювання складних поведінкових патернів, на кшталт різних дій для короткого та довгого натискання, є ускладненим або недосяжним без суттєвих обхідних рішень.

Компонування. Автоматичне компонування спирається на примітиви BeginVertical/BeginHorizontal, ScrollView, а також обмеження MinWidth/MaxWidth/ExpandWidth. Цього достатньо для простих макетів, однак відсутні важливі механізми адаптивності — процентні розміри від батька, підтримка співвідношення сторін, тощо. Ключове обмеження — неможливість автоматично визначити висоту контейнера з текстом із перенесенням слів; розробник має обчислювати її самостійно, часто з ручним розміщенням елементів.

Продуктивність і виділення пам'яті. Під час кадру бібліотека генерує значні обсяги короткоживучих об'єктів, покладаючи їх прибирання на збирач сміття. Для автоматичного компонування UI-виклики виконуються двічі: спочатку для розкладки, згодом для оброблення взаємодій, що збільшує накладні витрати.

Ідентифікація елементів. Унікальність контролів визначається числовими ідентифікаторами, які повинен вручну підтримувати розробник на рівні вікна. Це створює крихкі місця (збій відповідності «ID ↔ елемент») та збільшує кількість помилок типу «Mismatched LayoutGroup» і «Getting control 0's position…» за некоректних пар Begin*/End*.

Стилізація та теми. Стилі (GUIStyle) і теми (GUISkin) передаються параметрами у виклики створення віджетів, що ускладнює дрібні варіації та повторне використання між темами.

Текст і рендеринг. Зафіксовано проблеми якості відтворення тексту (розмитість) і підвищену вартість зберігання метрик гліфів.

Інтеграція з ігровим циклом. Через двофазність викликів за кадр (layout → interaction) формування UI необхідно виконувати в методі OnGUI, тоді як більшість ігрової логіки розташовується в методі Update. Це вимагає зберігати проміжний стан між викликами в межах одного кадру, підвищуючи складність і споживання пам’яті.

1.2.2. UGUI

UGUI було запроваджено як відповідь на обмеження IMGUI, із глибокою інтеграцією в GameObject-Component архітектуру Unity. Елементи інтерфейсу організовано як ієрархію об’єктів на сцені, в вершині яких стоїть об’єкт Canvas. Кожен об’єкт має компонент RectTransform (позиція/розмір) та функціональні компоненти (Button, Slider, Toggle тощо).

Компонування. Підхід до компонування став простішим і водночас жорсткішим: дочірні елементи задають зміщення/розміри відносно батька, тоді як розмір батьківського контейнера фіксований і не виводиться з розмірів дітей. Це знижує динамічність і можливості підлаштування складних макетів під різні екрани.

*Продуктивність***.** Завдяки дереву елементів, що зберігається між кадрами, і спрощеній розкладці зменшено витрати на перерахунки в динаміці, що поліпшило типову продуктивність порівняно з IMGUI у сценаріях зі значним обсягом статичних піддерев.

Візуальні можливості та стилізація. Порівняно з IMGUI було втрачено низку вбудованих візуальних ефектів (зокрема, округлення кутів і контури), а вбудована підтримка стилів і тем відсутня. Потреба в сучасних ефектах і адаптивності спонукала розробників до індивідуальних реалізацій, які нерідко виявлялися ресурсомісткими через невідповідність базової архітектури вимогам до складної розкладки.

Інженерна придатність. Єдність із загальною GameObject-Component моделлю спрощує навчання та базову інтеграцію, однак обмежена виразність розкладки і брак системної підтримки тем ускладнюють побудову сучасних динамічних інтерфейсів.

1.2.3. UI Toolkit

UI Toolkit репрезентує третє покоління UI в Unity і концептуально спирається на веб-технології: структура описується в UXML (аналог HTML), стилі — у USS (похідна від CSS з меншою кількістю функціоналу).

Цільова мотивація. Декларувалося перенесення частини робіт зі створення інтерфейсів на фахівців із веб-компетенціями та розвантаження ігрових програмістів. На практиці масових випадків залучення суто фронтенд-розробників для UI Unity не зафіксовано; формування ігрового UX залишилося прив’язаним до специфіки рушія та ігрового циклу.

Прийняття спільнотою. Сприйняття розробниками виявилося стриманим: принципи веб-верстки відрізняються від архітектури ігрової логіки, а історичні особливості HTML/CSS не завжди сумісні з вимогами до інтерактивних сцен із жорсткими обмеженнями реального часу. Значна частина команд продовжує використовувати UGUI.

Виразність і обмеження. Частину візуальних можливостей повернуто (округлення кутів, контури, базова автоматична розкладка), однак низка веб-функцій відсутня (наприклад, співвідношення сторін, проміжки між дітьми). Таким чином, бібліотека залишає розрив між очікуваною виразністю сучасних UI і доступним інструментарієм.

Продуктивність і надійність. У практиці експлуатації описано проблеми продуктивності та стабільності: довготривалі витоки пам’яті, пов’язані з кешами текстових метрик; чутливість продуктивності до вбудованих стилів у UXML і кількості файлів USS. Рекомендації зведення стилів до єдиного великого файлу USS ускладнюють масштабування і супровід.

Зв'язування даних. Механізми прив'язки залежать від узгодження імен між C#-класами та UXML-ідентифікаторами; перейменування у розмітці руйнує прив'язки, що створює крихкі точки відмови у великих проєктах.

Ширший контекст. У самій веб-екосистемі спостерігається перехід до декларативних бібліотек (React, Vue тощо), які пропонують API, ближчий до досвіду в IMGUI — опис цільового стану за кадр. Враховуючи, що Unity не працює поверх веб-рушія, перенесення парадигм HTML та CSS без адаптації до вимог реального часу піддається обґрунтованій критиці як таке, що не завжди відповідає специфіці ігрових UI.

1.3. Методична рамка подальшої оцінки

Цей підрозділ формалізує вимоги до нової архітектури IMGUI та визначає критерії, сценарії та методи її подальшої оцінки. Рамка узгоджена з цілями наступних розділів, в яких буде проведено експериментальне дослідження конвеєра кадру, ідентифікації елементів, впливу інженерних практик на продуктивність, а також порівняння з UI Toolkit.

1.3.1. Цільові функціональні вимоги

Візуальна підсистема: закруглені кути, контури, градієнти. Мета цієї вимоги — зберегти та розширити візуальний функціонал попередніх бібліотек.

Адаптивна композиція:

– переніс рядків у тексті з коректним автоматичним обчисленням висоти контейнера;

– відсоткові розміри від батьківського контейнера;

– підтримка фіксованого співвідношення сторін;

– проміжки між дочірніми елементами.

Стилізація та підтримка тем: можливість створювати елементи з однаковим функціоналом і різними стилями; механізми тем, перевикористання токенів стилю та змінних без дублювання параметрів у викликах віджетів.

1.3.2. Архітектурні вимоги, специфічні для IMGUI

а) Однопрохідність і продуктивність. Мінімізація короткоживучих виділень пам’яті за кадр; стабільна вартість кадру під динамічними змінами.

б) Інтеграція з ігровим циклом. Можливість виклику з Update (або еквівалентної стадії) без спеціальних додаткових проходів.

в) Унікальна ідентифікація елементів. Замість ручних числових ID — стабільні ключі на основі структурованого шляху або користувацького ключа з хешуванням. Інваріантність щодо перестановок елементів у списках.

г) Кеші та структури даних. Ефективні кеші для інтерактивних станів (натиснення, перетягування, фокус), а також для метрик тексту з переносами. Оцінка альтернатив хеш-карт для C# поза стандартним Dictionary з точки зору часу доступу та локальності.

г) Сумісність з архітектурою ECS. Відсутність необхідності у статичному дереві та зворотних викликах головного потоку.

1.3.3. Дослідницькі питання та гіпотези

– Конвеєр кадру IMGUI. Які комбінації стадій (input → layout → hit-test → draw) забезпечують однопрохідність без регресу виразності? Гіпотеза: суворе відокремлення стадій і відкладена емісія шарів дозволяють уникнути повторних проходів.

– Ідентифікація елементів. Яка схема стабільних ключів мінімізує помилки прив’язки стану при перестановках? Гіпотеза: хеш шляху + користувацький ключ > індекс у списку.

– Інженерні практики та продуктивність. Як впливають локальність даних, компонування пам’яті, уникнення розгалужень, ефективне використання предиктора та префетчера? Гіпотеза: плоскі структури й послідовний обхід зменшують хвости латентності та GC. Бінарні операції на прапорах мінімізують розгалудження.

– Хеш-карти в C#. Які реалізації (та їх налаштування) переважають за латентністю та виділенням пам’яті у профілі «багато коротких звернень за кадр»?

– Алгоритми хешування тексту. Які хеші забезпечать потрібну рівномірність і швидкість у контексті частого кадру? Чи доцільне компіляційне попереднє хешування відомих ключів для нульової вартості у виконанні?

1.3.4. Методи та дизайн експериментів

а) Аналіз пайплайна кадру в IMGUI-бібліотеках. Опис і порівняння дво-, три- та одно-прохідних варіантів; верифікація інваріантів однопрохідності (відсутність «повернень» до вузлів).

б) Схеми ідентифікації. Порівняння числових ID, стабільних ключів (хеш шляху) та складених ID, а також ID, які базуються на місці виклику функції.

в) Структури даних. Порівняння реалізацій хеш-карт у C# у профілях «інтенсивне читання» та «запис великих об’єктів з маленькими ключами», облік виділень пам’яті та швидкодії.

г) Алгоритми хешування тексту. Порівняння швидкості та рівномірності для ключів, характерних для UI, оцінка колізій, тест компіляційного попереднього хешування для відомих шаблонів ключів.

ґ) A/B-порівняння з UI Toolkit. Реалізація ідентичного інтерфейсу в DiVy та в UI Toolkit; заміри середнього часу кадру, перцентилів, виділень пам’яті, чутливості до сценаріїв:

– зміна структури дерева (додавання/видалення піддерев)

– анімації інтеракції.

1) Метрики й оброблення даних. Середнє часу кадру та кількості циклів процесора; коефіцієнт варіації, умовної вартості під ризиком, виділень пам’яті за кадр. Статистичне порівняння, візуалізація розподілів.

2) Порівняння кількості коду: рядків коду за виключенням порожніх та коментарів, кількості літер. Порівняння відбувається у двох категоріях. Перша — віджети. Для окремих віджетів буде побудовано функціонально еквівалентні реалізації. Друга — кінцева панель. Для цільової фінальної панелі (узгоджений макет, набір контролів, навігація фокусу, стани наведення та натиснення) вимірюються:

– сумарний SLOC для C#, розмітки (UXML), стилів (USS) у випадку UI Toolkit;
– для IMGUI — суто C# і шаблони стилізації;

1.4. Висновки до розділу 1: вимоги і очікувані результати

На основі огляду літератури та аналізу наявних бібліотек Unity сформульовано комплекс вимог до нової архітектури IMGUI: сучасна виразність (округлення, контури, градієнти), повна адаптивність (переноси, відсоткові розміри, aspect-ratio, проміжки), системна стилізація та підтримка тем, однопрохідність і низькі виділення пам’яті, стабільні ідентифікатори без ручних числових індексів, сумісність з Update та ECS.

Сформульовано дослідницькі питання та гіпотези щодо конвеєра кадру, ідентифікації елементів, інженерних практик, вибору структур даних і хешування.

Запропоновано методики та дизайн експериментів для наступних розділів: мікробенчмарки, моделювання параметрів системи, A/B-порівняння з UI Toolkit за набором репрезентативних сценаріїв, статистичне узагальнення результатів.

Результатом розділу 2 мають стати верифіковані емпіричні висновки щодо доцільності однопрохідної IMGUI-архітектури, а також конкретні інженерні рішення (схеми ідентифікації, політики кешування, вибір хеш-карт і алгоритмів хешування), які забезпечують стабільність кадру та продуктивність у динамічних сценаріях та на платформах AR/VR.

2 АНАЛІЗ АСПЕКТІВ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ТА ОБҐРУНТУВАННЯ МЕТОДІВ ЕКСПЕРИМЕНТАЛЬНОГО ОЦІНЮВАННЯ

2.1. Аналіз алгоритмів проходів за кадр

Одним із завдань цієї роботи є забезпечення однопрохідного виклику функцій IMGUI-бібліотеки в межах кадру. Такий режим уможливлює (i) співрозміщення побудови інтерфейсу з бізнес-логікою в одній фазі ігрового циклу, (ii) розміщення відладних інтерфейсів безпосередньо в точках виклику цільових функцій, (iii) зниження вартості відтворення UI та (iv) сумісність із архітектурою ECS, де оброблення відбувається в кількох потоках. Для обґрунтування вибору підходу проаналізовано поширені схеми: однопрохідний алгоритм із ручним компонуванням, багатопрохідні варіанти з повною автоматичною розкладкою та однопрохідний алгоритм з відкладеним введенням на один кадр.

2.1.1 Однопрохідний варіант із ручним компонуванням

Найпростішою конфігурацією IMGUI є ручне компонування, за якого геометричні параметри віджетів (позиція, розмір) відомі до моменту їх створення. Такий підхід практично усуває приховані перерахунки, забезпечує детермінований порядок рендерингу й мінімізує витрати CPU, проте обмежує виразність (бракує адаптивної розкладки, залежної від вмісту елементів) і підвищує трудомісткість розробки, оскільки розміри та позиції слід обчислювати вручну для кожного вікна.

Приклад ручного компонування:

```
DoBox(x: 0, y: 0, width: 100, height: 50);
for (int i = 0; i < 4; ++i){
    DoButton($"Button {i+1}", x: i * 25, y: 0, width: 25, height: 50);
}
```

За такого підходу достатньо одного проходу: під час виклику DoButton усі необхідні дані (геометрія, стан вводу, фокус) уже відомі; можна негайно виконати хіт-тест і згенерувати команди рендерингу.

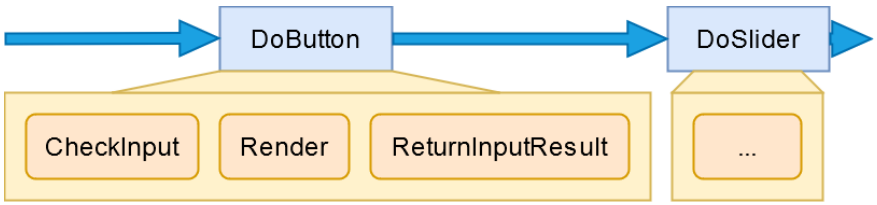


Рис 2.1 Схема роботи IMGUI з ручним компонуванням

У практиці часто застосовують допоміжні засоби напів-автоматичного компонування, що виконують елементарні вирівнювання до викликів віджетів; обмеженням залишається потреба заздалегідь визначити ключові габарити. Приклад коду з використанням допоміжних функцій для напів-автоматичного компонування:

```
if (ImGui.Button("Open")) { /*...*/ }
ImGui::SameLine(); // розмістити наступний елемент на тому ж рядку
if (ImGui.Button("Save")) { /*...*/ }
ImGui.NewLine(); // розмістити наступний елемент на новому рядку
ImGui.TextUnformatted("Status: Ready");
```

Перевагою є низька вартість кадру, однак повної автоматизації компонування (з урахуванням вмісту, перенесення рядків, відсоткових розмірів тощо) досягти неможливо без додаткових проходів.

2.1.2 Багатопрохідні схеми

Щоби забезпечити повністю автоматичне компонування, IMGUI часто використовують у двопрохідному режимі.

– Перший прохід – макетний: функції віджетів викликаються без остаточного врахування дій користувача; система збирає повну інформацію про дерево елементів і «побажання» щодо розміщення. На цій основі виконується розв’язання задачі компонування (власні алгоритми або відомі підходи на кшталт constraint-систем, Flexbox, Cassowary тощо).

– Другий прохід – інтерактивний: за відомих розмірів і позицій відбувається інтерпретація вводу, повернення значень із віджетів і відтворення.

У низці рішень додають третю фазу (окремий прохід для рендеру), відокремлюючи оброблення вводу від емісії команд відтворення.

Приклад використання адаптивного компонування при створенні елементів з Unity IMGUI:

```
GUILayout.BeginArea(Rect(0, 0, 100, 50));
GUILayout.BeginHorizontal();
for (int i = 0; i < 4; ++i) {
        GUILayout.Button($"Button {i+1}");
}
GUILayout.EndHorizontal();
GUILayout.EndArea();
```

Наслідки багатопрохідності:

а) Зниження ефективності через повторне виконання користувацького коду й бібліотечних викликів.

б) Ускладнення поєднання з бізнес-логікою. Код, який має виконувати побічні дії один раз за кадр (наприклад, інтеграція положення об’єкта за Time.deltaTime), при двох викликах спрацює двічі, що призведе до некоректних результатів, якщо спеціально не відокремлювати побічні ефекти від UI-викликів. Приклад такого сценарію наведений нижче.

```
void ProcessSpeed() {
   position += speed * Time.deltaTime;       // має виконатися 1 раз за кадр
    GUILayout.Label($"Speed: {speed}");       // має бути викликаний двічі у 2-прохідній схемі
}
```

Це збільшує когнітивне навантаження на розробника та знижує зручність локального розміщення відладних панелей поруч із логікою.

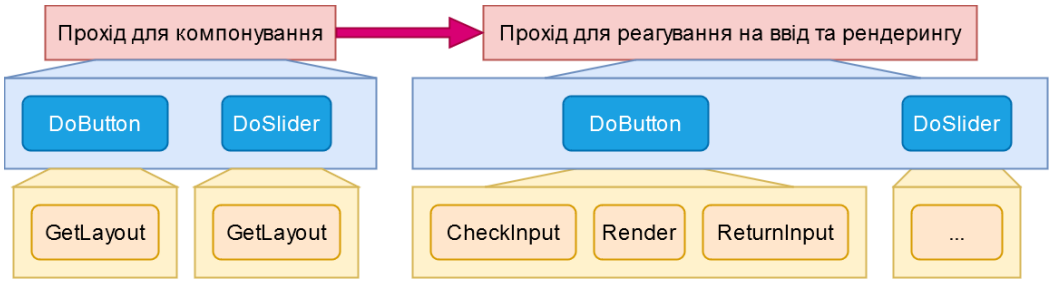


Рис 2.2 Схема роботи IMGUI з двома проходами

2.1.3 Однопрохідний варіант із відкладеним введенням

Альтернативний підхід — один прохід за кадр із відкладеним обробленням введення на один кадр уперед. У кадрі t бібліотека один раз викликає віджети, обчислює геометрію й виконує рендеринг. На початку кадру t+1 доступний фактичний ввід користувача; збережена з t геометрія використовується для хіт-тесту та генерації результатів взаємодії, які передаються відповідним віджетам під час їх виклику в t+1.

Потенційна колізія (натискання на елемент, який зникає до кадру t+1) на практиці малопомітна: за частоти 60 FPS тривалість кадру становить ≈16,7 мс, що істотно менше від типової людської реакції (порядку сотень мілісекунд) і сумарних затримок введення пристрою. Отже, суб'єктивне запізнення взаємодії зберігається на прийнятному рівні навіть за відкладеної інтерпретації. На цій схемі побудовано низку інструментальних застосунків, які демонструють належну керованість взаємодії за збереження однопрохідності [35].

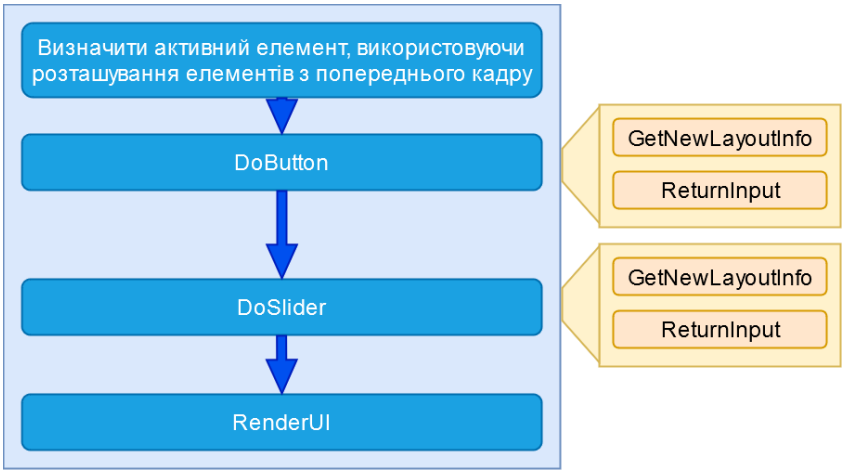


Рис 2.3 Схема роботи IMGUI з адаптивним компонуванням і одним проходом за кадр

2.1.4 Проміжний висновок

Ручне компонування забезпечує мінімальну вартість кадру, але не покриває потреб адаптивних інтерфейсів. Багатопрохідні схеми дають повну автоматичну розкладку ціною підвищеної часу використання процесора та ускладнення інтеграції з бізнес-логікою. Однопрохідний варіант із відкладеним введенням пропонує збалансоване рішення: зберігає однопрохідність і дозволяє автоматичне компонування, зводячи додаткову латентність до рівня, що зазвичай не погіршує користувацьке сприйняття. За результатами аналізу для нової бібліотеки було обрано алгоритм одного проходу за кадр із відкладеним введенням.

2.2 Аналіз способів ідентифікування інтерактивних елементів

Проблематика унікальної ідентифікації віджетів є другою принциповою складовою архітектури IMGUI, розв’язання якої істотно залежить від конкретних інженерних підходів у тій чи іншій бібліотеці. Необхідність ідентифікації зумовлена потребою зберігати коректну відповідність між подіями введення та логічним екземпляром елемента інтерфейсу впродовж кількох кадрів, аби послідовно відстежувати стан взаємодії (натиснуто / перетягується / має фокус тощо).

Специфіка IMGUI полягає в тому, що зовнішній програмний інтерфейс навмисно моделює взаємодію, яка не передбачає управління пам’яттю бібліотеки ззовні. Зокрема, розробник зазвичай не отримує з API адрес елементів у пам’яті. Така постановка задачі делегує управління довгоживучим станом (значення слайдерів, вибрані опції тощо) самому розробникові прикладної логіки.

Водночас зазначене обмеження стосується лише зовнішньої поверхні API. Усередині бібліотеки мінімальне кешування та збереження стану є не лише допустимими, а й необхідними — насамперед для коректного хіт-тесту, маршрутизації подій та підтримки інваріантів взаємодії. Ключова умова — недоступність такого стану для зовнішнього коду і його незмінюваність з боку клієнта бібліотеки.

З практичних міркувань ідентифікувати потрібно лише інтерактивні елементи (кнопки, слайдери, поля вводу тощо). Нестатичні стани на кшталт hot / active / focused мають бути прив’язані до стабільного маркера, який переживає зміни кадрів. Натомість чисто декоративний або статичний вміст (текст, зображення без взаємодії) може вільно перебудовуватися щокадру без прив’язки до ідентифікатора.

2.2.1 Підхід 1: послідовна нумерація під час проходу дерева

У найпростішому варіанті кожному віджету призначається порядковий номер у момент виклику його функції, причому бібліотека автоматично видає наступне вільне значення. Перевага — нульова церемоніальність для користувача: не потрібно передавати додаткові параметри, відповідальність повністю на бібліотеці. Недолік — нестабільність під час структурних змін: якщо елемент у середині ієрархії приховується, ідентифікатори усіх наступних віджетів зсуваються; активний стан може помилково перейти до іншого елемента. Частково проблему пом’якшують «контрольні точки», які фіксують номер у межах піддерева: типово їх ставлять на рівні вікна, а також у місцях очікуваних приховувань секцій. Однак така дисципліна підвищує когнітивне навантаження на розробника і схильна до людської помилки; автоматичне виявлення пропущених контрольних точок бібліотекою нетривіальне. Підхід реалізовано, зокрема, в Unity IMGUI [1].

2.2.2 Підхід 2: хешування користувацьких даних: імен та контекстів

Інший широко використовуваний підхід — формувати ідентифікатор як хеш користувацького маркера, найчастіше — текстової мітки елемента. Якщо у межах вікна назви унікальні, то додаткових параметрів не потрібно: бібліотека хешує текст з назвою і застосовує його як ID. Для уникнення колізій між вікнами вводиться контекст (наприклад, PushContext(string id) та PopContext()), який додається другим аргументом до хеш-функції. Якщо дві кнопки всередині одного вікна мусять мати однаковий напис, застосовується суфікс, що не показується на екрані. Наприклад, у бібліотеці DearImGui у рядках на кшталт "OK##ID1" та

"OK##ID2" частина після ## не відображається, але бере участь у хешуванні [5]. Переваги підходу:

– самодокументованість — зрозумілі для людини назви полегшують діагностику, можна вибудувати деревоподібний інспектор;

– легкість виявлення неунікальних маркерів під час розробки.

Недоліки: не всі елементи мають видимі мітки (кнопки із іконками без тексту), а обов’язок забезпечення унікальності загалом лежить на розробникові, хоча бібліотека допомагає розробникові швидко знаходити проблемні місця.

2.2.3 Підхід 3: ідентифікація за місцем виклику

Третій клас рішень автоматизує формування ID з використанням метаданих місця виклику — назви файлу та номера рядка. У C/C++ відповідні значення надають макроси __FILE__ та __LINE__. Переваги:

– стабільність ідентифікаторів за змін порядку або кількості елементів;

– зняття навантаження з розробника на ручну генерацію ID.

Обмеження:

– коли кілька елементів створюються в одному рядку (наприклад, у циклі), потрібен додатковий розрізнювальний параметр — індекс або ключ, щоби уникнути колізій;

– підхід прив’язаний до можливостей мови. Для деяких мов відсутні або обмежені зручні аналоги передачі файлу та номеру рядка. Зокрема, цей функціонал відсутній у C#, що є особливо критичним у контексті цієї роботи.

Підхід реалізовано, зокрема, у бібліотеці vui [6].

2.2.4 Підхід 4: стан-орієнтована взаємодія без явної ідентифікації

Існують реалізації, що мінімізують саму потребу в ідентифікаторах, зберігаючи лише факт взаємодії, а не прив’язку до конкретного елемента. Якщо курсор виходить за межі віджета або віджет зміщено, активний стан анулюється. Така схема достатня для багатьох сценаріїв, проте втрачає стійкість за складних трансформацій інтерфейсу (динамічне приховування секцій, інтенсивні анімації). Приклад — Nuklear [7].

2.2.5 Підхід 5: фіксація координати взаємодії

Подальшим розвитком попередньої ідеї є збереження історії координат натискання. Тоді кожний кадр можна визначати, чи відбувається взаємодія з елементом, якщо миша була натиснута раніше в тій же точці, де зараз знаходиться елемент. Цей спосіб перестає діяти, якщо елементи рухаються. Це може статись через приховування секції вікна, анімацію елемента, або перетягування блоків у вузловому графі. Прикладом використання цього підходу є Lobster [8].

2.2.6 Проміжний висновок

Вибір стратегії ідентифікації визначає стабільність станів взаємодії, детермінованість поведінки під час структурних змін і церемоніальність API для розробника. Послідовна нумерація мінімізує параметризацію, але чутлива до модифікацій ієрархії. Хешування користувацьких маркерів забезпечує прозору діагностику ціною дисципліни дотримання унікальності. Ідентифікація за місцем виклику максимально автоматизує процес, але накладає обмеження у виборі мови програмування та вимагає додаткових ключів у циклах. Стан-орієнтовані техніки знижують залежність від ID, однак поступаються стабільністю в динамічних сценах.

У цій роботі обрано підхід хешування назв із низкою інженерних удосконалень. Такий підхід є сумісним із C#, не вимагає суворої дисципліни стабілізації ієрархії, як послідовна нумерація, і не має обмежень, притаманних стан-орієнтованій взаємодії, що звужують її застосовність у бібліотеках, орієнтованих на кінцеві застосунки. Вимога явного надання міток не створює надмірного когнітивного навантаження, оскільки у практиці розробки такі мітки все одно використовуються для логування та інструментальної діагностики. Запропоновані модифікації підвищують надійність і ефективність підходу:

а) Деревоподібне просторове іменування. Хеш батьківського вузла використовується як зерно для побудови хешів нащадків. Це локалізує вимогу унікальності до множини елементів усередині одного батьківського контейнера,

усуває колізії між піддеревами та знижує ризик помилок ідентифікації під час структурних змін.

б) Компіляційне попереднє хешування літералів. Генерація коду замінює літеральні мітки їхніми хеш-значеннями на етапі компіляції, що усуває витрати на хешування під час виконання, стабілізує значення ідентифікаторів та не погіршує часові характеристики порівняно з альтернативними схемами формування ID у виконуваній програмі.

2.3 Аналіз імплементацій хеш-карт для зберігання інформації про інтерфейс

Парадигма IMGUI передбачає таку організацію зовнішнього інтерфейсу бібліотеки, за якої створюється ілюзія миттєвої реакції на виклики та відсутності довготривалого стану елементів з погляду користувача API. На відміну від RMGUI, де віджет існує як об’єкт, доступний у просторі імен прикладної програми (напр., через вказівник/посилання на екземпляр) і може прямо модифікуватися, у IMGUI розробник не оперує об’єктними дескрипторами віджетів. Водночас для забезпечення автоматичного компонування та коректної обробки вводу внутрішня частина бібліотеки неминуче підтримує мінімальний стан, інкапсульований від зовнішнього коду. Відтак виникає потреба в механізмі адресації внутрішніх записів за допомогою допоміжної інформації, що генерується під час проходу по дереву інтерфейсу. Як зазначено у попередньому підрозділі, таким механізмом є хешування імен; отже, потрібна високоефективна хеш-карта (hash map), яка зберігає записи, індексовані за хешами цих ідентифікаторів.

Другою критичною зоною застосування хеш-карт є обробка тексту. У найпростішій реалізації, коли у віджет передається великий текстовий блок, IMGUI-процедури повторюють повний аналіз тексту кожного кадру: визначають довжини слів, місця переносів, метрики рядків, а також горизонтальні та вертикальні зміщення. Для значних обсягів тексту додатково необхідно обмежувати видиму область з метою мінімізації команд рендерингу. За відсутності

кешування такі операції можуть домінувати у бюджеті процесора і, за емпіричними оцінками, сягати до 90% часу, витраченого бібліотекою на підготовку кадру. Для бібліотеки, орієнтованої на інтерфейси кінцевих продуктів, є обов’язковим кешувати результати сегментації, розташування переносів і метрик рядків, аби істотно зменшити витрати на повторні обчислення.

Отже, ефективність імплементації хеш-карти безпосередньо визначає витрати процесорного часу на пошук і оновлення двох центральних сутностей IMGUI — записів стану віджетів та кешів текстових метрик. Водночас реалізація має залишатися розширюваною, щоби підтримати специфічні оптимізації, притаманні роботі з інтерфейсами. Нижче наведено приклади інженерних модифікацій до класичної схеми хеш-карти.

а) Уникнення зберігання повного текстового шляху як ключа. Для адресації віджетів природно використовувати хеш шляху вузла в ієрархії (напр., поєднання імен батьків і власної мітки). У класичних реалізаціях пошуку спершу порівнюють хеші, а у випадку збігу — перевіряють повні ключі. Для UI це має три суттєві вади:

– підвищене споживання пам’яті (зберігання довгих рядків-ключів);

– вартісна побудова ключа. Потрібно зчитувати імена батьківських контейнерів і об’єднувати їх; це погіршує локальність і збільшує кількість звернень до пам’яті по «далеких» вузлах дерева;

– висока вартість порівнянь рядків на гарячому шляху пошуку. Практичним розв’язанням є відмова від зберігання тексту ключа та покладання виключно на порівняння хешів, за умови вибору достатньо довгого хешу та якісної хеш-функції з мінімальною імовірністю колізій. Політика обробки колізій у UI-контексті може бути деґрадаційною, але безпечною для зовнішнього вигляду: у разі колізії не критичного елемента створюється неінтерактивний сурогат, а розробник негайно інформується про подію (на етапі розробки або тестування), що дає змогу виправити мітки. Звідси вимога до хеш-карти: підтримка режиму без порівняння повних ключів, коли рішення приймається за хешами.

б) Кероване виселення (eviction) застарілих записів для текстових кешів. Для запобігання витокам пам’яті кеш текстових метрик потребує вчасного видалення неактуальних слотів. Це передбачає фіксацію часу останнього використання та механізм регулярної ревізії. Можливі два підходи:

1) Лінійне сканування кожного кадру: послідовний перегляд елементів хеш-карти й вилучення записів, чий «вік» перевищує поріг. Простий метод, але збільшує навантаження на пам’ять та доступ до кешів процесора.

2) Опортуністичне виселення під час пошуку: процедура локалізації потрібного елемента у відкритому адресуванні або в ланцюжках зіставлень і так торкається кількох сусідніх слотів (перевірка ключів під час зондування). Це вікно доступу можна використати для місцевого очищення: застарілі записи видаляються «по дорозі», синергійно зменшуючи додаткові звернення до пам’яті.

У підсумку хеш-карта має або забезпечувати швидкий послідовний доступ для періодичного сканування, або дозволяти модифікацію алгоритму пошуку так, щоби опортуністично видаляти перевірювані елементи.

Таким чином, з огляду на наведені вимоги, постає завдання порівняльного аналізу наявних імплементацій хеш-карт та вибору такої, що поєднує:

а) високу пропускну здатність пошуку/вставки для профілю «часті звернення з малими ключами та об’єктами помірного розміру»,

б) можливість відмови від порівнянь повних ключів на гарячому шляху з контрольованою політикою колізій,

в) підтримку виселення застарілих записів із мінімальними накладними витратами, та

г) простоту модифікації ядра для інтеграції зі специфікою IMGUI-конвеєра та підсистемою тексту.

Імплементації хеш-карт доцільно поділяти на дві великі групи: вузлові (node-based, із роздільним виділенням вузлів і ланцюжками в бакетах) та пласкі (flat, open-addressed, із збереженням слотів у щільному масиві).

2.3.1 Вузлові хеш-карти

Базова організація включає масив бакетів, кожний з яких зберігає вказівник на голову ланцюжка (часто односпрямованого списку) елементів, що потрапили до цього бакета. За колізії новий елемент приєднується до ланцюжка бакета, типово — вставкою в початок. Характерні властивості:

– Виділення пам’яті та локальність. Кожен вузол (ключ/значення плюс службові поля) виділяється окремо; унаслідок цього доступ відзначається переходами у довільні місця пам’яті з погіршеною локальністю кешу та нижчою ефективністю апаратного попереднього вибіркового завантаження (prefetch).

– Чутливість до колізій. Збільшення коефіцієнта заповнення підвищує середню довжину ланцюжків і погіршує середній час доступу; продуктивність істотно залежить від якості хеш-функції та політики реорганізації бакетів.

– Стабільність посилань. Важлива перевага — стабільність адрес та ітераторів елементів при вставках (до моменту глобального перехешування), що корисно для структур, де збереження посилань є необхідністю.

– Прогнозованість перетинів кешу. Нерівномірне зондування пам’яті ускладнює ефективне використання кеш-ієрархії під час інтенсивних пошуків.

2.3.2 Пласкі хеш-карти

У пласких структурах ключі та значення зберігаються в щільному масиві слотів; колізії розв’язуються зондуванням позицій відповідно до заданої схеми (лінійної, квадратичної, Robin-Hood тощо). Сучасні реалізації (клас «SwissTable») додатково тримають масив байтів керування для групового звіряння кандидатів через SIMD. Характерні властивості:

– Локальність і можливість використання SIMD. Доступ до послідовних слотів формує кластери лінійних читань, що добре узгоджується з кеш-ієрархією та дозволяє векторизувати первинну фільтрацію кандидатів.

– Коефіцієнт заповнення. Продуктивність визначається довжиною кластерів; щоб обмежувати середню кількість зондів, підтримують фактор завантаження нижче порогового значення (орієнтовно 0.7–0.9 залежно від схеми).

– Переміщення під час вставки. У схемах на кшталт Robin-Hood можливі локальні ротації/зсуви елементів задля зменшення дисперсії відстані від «домашнього» бакета; у лінійному/квадратичному зондуванні такі переміщення мінімальні. Сучасні реалізації прагнуть зменшувати обсяг релокацій.

– Паралельні сценарії. Пласке розміщення спрощує пошук декількох значень одночасно із розпаралелюванням на рівні набору запитів, а також внутрішню векторизацію. Водночас конкурентний доступ (паралельні вставки/видалення в межах однієї структури) все одно потребує явних механізмів синхронізації і не є «безкоштовною» властивістю open-addressed підходу.

2.3.3 Порівняльні наслідки для IMGUI

Для конвеєрів користувацького інтерфейсу, де характерні часті читання з невеликими ключами/ідентифікаторами та жорсткі вимоги до локальності, пласкі реалізації зазвичай демонструють меншу середню латентність доступу за рахунок лінійного зондування і можливостей SIMD-фільтрації. Вузлові карти зберігають перевагу стабільності посилань і можуть бути доцільними там, де ця властивість є принциповою. Вибір конкретної схеми має базуватися на цільовому профілі (частка читань/вставок/видалень, допустимий фактор завантаження, вимоги до стабільності ітераторів) та властивостях хеш-функції, що визначають поведінку колізій.

2.3.4 Порівняння розмірів хеш-карт: прості числа проти ступенів двійки

Узагальнена операція відображення хешу у індекс бакета має вигляд:

index = hash % capacity;

де capacity — поточна місткість таблиці (кількість бакетів/слотів). Пряме ділення за модулем є відносно дорогим, тому практичні реалізації намагаються або спростити арифметику, або зменшити чутливість до якості молодших бітів хешу.

Місткість як ступені двійки. Для capacity = 2^k індексація спрощується до:

index = hash & (capacity – 1);

Наприклад, capacity = 1024 = 2^10 = 0b1'0000000000, тоді mask = 1023 = 0b1111111111. Такий підхід зменшує накладні витрати до однієї побітової операції з дуже низькою латентністю.

Недолік полягає в тому, що використовуються лише молодші k бітів. За слабкого розподілу цих бітів (наприклад, коли ключі — послідовні індекси або коли хеш-функція має «мертві» молодші біти) зростає кластеризація і довжина зондування.

Дві поширені інженерні протидії:

– Попереднє перемішування (multiplicative mix):

const uint phi = 0x9E3779B9u; // ≈ 2^32 / φ

index = (hash * phi) & (capacity - 1);

Це покращує розподіл, але все ще відбирає молодші біти добутку.

– Хешування Фібоначчі зі зсувом старших бітів — краща практика для 2^k:

const uint phi = 0x9E3779B9u;

uint shift = 32u - log2(capacity);

index = (hash * phi) >> shift; // беремо старші біти добутку

Відбір старших бітів добутку різко знижує залежність від низькорозрядних патернів.

Цей клас підходів добре узгоджується з сучасними пласкими реалізаціями, де індекс і контрольні байти утворюються з різних частин змішаного хешу, що дозволяє ефективно використовувати перевірки груп бакетів через SIMD.

Місткість як прості числа. Альтернатива — підтримувати місткість як просте число і використовувати загальний модуль:

index = hash % capacity; // capacity — просте

Перевага — менша чутливість до структури молодших бітів хешу: модуль за простим числом ефективно «змішує» внесок різних бітів, і навіть за невдалих хешів знижується систематична колізійність, особливо на малих місткостях.

Вартісність:

– Для константного capacity сучасні компілятори генерують код на базі мультиплікативного наближення ділення, що дорожче за маску, але суттєво швидше за загальне ділення.

– Для змінної місткості залишаються інструкції ділення за модулем з помітнішою затримкою.

Практичні наслідки: модуль за простим числом може дати кращу стійкість при сумнівній якості хеш-функції або в змішаних профілях ключів, але поступається підходам зі ступенем двійки за швидкодією індексації.

Якщо таблиця використовує якісну хеш-функцію або застосовується мультиплікативне відображення у старші біти, то capacity = 2^k зазвичай забезпечує найнижчу ціну індексації та прості механізми масштабування.

Порівняльні висновки і рекомендації. Якщо профіль ключів важко гарантувати (послідовні індекси, псевдо-ідентичні ключі, неоднорідні домени), а також якщо реалізація не виконує якісного перемішування перед індексацією, то вибір простих місткостей з модулем додає стійкості розподілу ціною вищої вартості обчислення індексу.

У високопродуктивних пласких картах переважає 2^k + мультиплікативне хешування зі зсувом старших бітів, тоді як прості числа доречні для простіших реалізацій або як консервативна опція за невпевненості в якості джерела хешів.

Таким чином, вибір між простими місткостями та ступенями двійки є балансом стійкості розподілу проти мінімізації вартості індексації; у сучасній практиці перевага часто надається схемам зі ступенями двійки у поєднанні з належним попереднім перемішуванням і відбором старших бітів добутку.

2.3.5 Стандартна імплементація C# Dictionary

Мова C# надає в стандартній бібліотеці універсальну хеш-таблицю загального призначення — клас Dictionary<TKey, TValue>. Попри назву «словник», її внутрішня організація є класичною хеш-картою з розподілом елементів по бакетах (buckets) на основі хеш-коду ключа. Використання стандартної реалізації є бажаним у разі відповідності функціональним і

нефункціональним вимогам системи, оскільки мінімізує обсяг власного коду, полегшує читання й супровід бібліотеки та спирається на перевірену часом оптимізовану реалізацію.

З погляду структури даних, Dictionary<TKey, TValue> реалізує окреме ланцюжкування (separate chaining), проте без виділення вузлів у купі (heap): замість списків об’єктів використовується пара масивів фіксованих типів. Перший масив int[] buckets зберігає індекси першого елемента в кожному бакеті (спеціальне значення позначає порожній бакет), другий — масив записів Entry[] entries, де кожен запис має форму:

```
struct Entry {
   int    hashCode; // нормалізований (не-від’ємний) хеш
   int    next;     // індекс наступного елемента у ланцюжку, або -1
   TKey   key;
   TValue value;
}
```

Пошук відбувається у два кроки: (1) обчислення індексу бакета за хеш-кодом ключа та поточною місткістю таблиці; (2) лінійне проходження ланцюжка в масиві entries за полем next з послідовним порівнянням hashCode і ключа. Така організація поєднує переваги спискової схеми (простота обробки колізій) з локальністю пам’яті (дані вузлів зберігаються компактно у масиві, без додаткових об’єктних накладних витрат), що загалом позитивно впливає на поведінку кешу порівняно з класичними вузловими структурами на базі зв’язаних списків.

Політика масштабування передбачає перерозподіл бакетів зі збільшенням місткості та повторним рознесенням елементів; цільовий коефіцієнт заповнення підтримується нижчим від критичного рівня для обмеження довжини ланцюжків і середнього часу операцій. Конкретна схема зростання місткості визначається версією платформи .NET, у будь-якому випадку застосовується повторне хешування з перерахунком індексів бакетів.

До переваг стандартної реалізації належать:

а) сталі асимптотики очікуваного часу доступу (O(1) у середньому),

б) відсутність об’єктних накладних витрат на вузли завдяки зберіганню Entry у щільному масиві,

в) інженерно відпрацьовані процедури розширення, усунення та повторного використання вільних слотів у entries.

Водночас закритість внутрішньої реалізації для модифікацій обмежує можливості тонкого налаштування під спеціалізовані профілі доступу, зокрема: зміни схеми індексації/перемішування, експериментальні політики контролю колізій, підтримку «м’якого» видалення для кешування текстових метаданих тощо. У випадках, коли потрібні такі розширення, доцільним є порівняння продуктивності стандартного Dictionary з альтернативними пласкими реалізаціями, що допускають гнучкішу модифікацію алгоритмічних вузлів.

2.3.6 Відкрите адресування

Відкрите адресування (open addressing) — це клас пласких хеш-карт, у яких кожен бакет зберігає щонайбільше один запис, а колізії розв’язуються пошуком наступного вільного бакета за наперед визначеною послідовністю зондування. Для ключа з хешем h формується індекс

$$i_0 = index(h),\ \ i_{k+1} = step(i_k, h)\ \ (mod\ \ capacity); \tag{2.1}$$

і перевіряються бакети $i_0, i_1, i_2, \ldots$ до знаходження відповідного ключа або порожньої комірки. Нехай $\alpha = \frac{n}{capacity}$ — коефіцієнт заповнення. За ідеалізованої моделі рівномірного хешування середня вартість операцій становить O(1) до помітного наближення α до 1, однак конкретні схеми зондування по-різному поводяться щодо кластеризації і мають різні робочі діапазони α.

Нижче розглянуто три класичні схеми: лінійне зондування, квадратичне зондування та подвійне хешування.

Лінійне зондування. Схема зондування:

$$i_k = (i_0 + k)\ \ (mod\ \ capacity),\ \ k = 0,1,2,\ldots. \tag{2.2}$$

Тобто у разі колізії перевіряються сусідні бакети.

Властивості:

– Локальність пам’яті. Доступ відбувається лінійно, що добре узгоджується з кеш-ієрархією та передбачуваним попереднім вибірковим завантаженням.

– Первинна кластеризація. Послідовні заповнення формують довгі кластери, до яких приєднується дедалі більше елементів; це зростаюче середнє і гірше перцентильне число зондувань при збільшенні α.

– Очікувані витрати (класичні апроксимації). Для лінійного зондування:

– успішний пошук: $E_{suc} \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$;

– неуспішний пошук: $E_{fail} \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$.

Звідси випливає практичне правило: для стабільної латентності бажано тримати $\alpha \lesssim 0.7–0.8$ [23, С. 513–558].

– Видалення. Потребує «надгробків» (tombstones) або зсувів назад (backward-shift deletion). Надгробки спрощують видалення, але з часом збільшують довжини зондування; періодична перебудова або очищення надгробків стає необхідною.

Переваги/обмеження. Максимальна ефективність використання кешу за помірних α; чутливість до кластеризації на високих αα та до якості індексації (див. §2.3.2).

Квадратичне зондування. Схема зондування:

$$i_k = \left(i_0 + c_1 k + c_2 k^2\right) \pmod{capacity},\ k = 0,1,2,\ldots, \tag{2.3}$$

де коефіцієнти c_1, c_2 — константи; покриття всієї таблиці залежить від вибору capacity та коефіцієнтів.

Властивості:

– Зменшення первинної кластеризації. Кроки зростають, тож елементи з різних початкових позицій рідше зливаються в один великий кластер.

– Вторинна кластеризація. Для однакового початкового індексу різні ключі породжують однакову послідовність зондувань; отже, конфлікти між такими ключами повторюються.

– Покриття і параметри. Для гарантованого охоплення всієї таблиці в межах α<1 потрібні умови на модуль та коефіцієнти. На практиці це звужує простір допустимих конфігурацій і ускладнює адаптивне масштабування.

Переваги/обмеження. Краща стійкість до первинних кластерів порівняно з лінійним зондуванням, але зберігається вторинна кластеризація; поведінка щодо кешу гірша, ніж у лінійного, через нелінійну траєкторію доступів.

Подвійне хешування. Схема зондування:

$$i_k = \left(i_0 + k \cdot \Delta\right) \quad \left(mod \quad capacity\right), \quad \Delta = h_2\left(key\right) \quad \in \quad \left\{1\,, ldots\,, capacity - 1\right\}, \qquad (2.4)$$

де h_2 — друга хеш-функція; зазвичай вимагається $gcd\left(\Delta\,, capacity\right) = 1$ (наприклад, capacity — просте або Δ — непарне при місткості ступеня двійки).

Властивості:

– Придушення кластеризації. Послідовність зондувань суттєво залежить від ключа; це зменшує і первинну, і вторинну кластеризацію, наближаючи поведінку до рівномірного зондування.

– Очікувані витрати (ідеальна модель).

– неуспішний пошук: $E'_{fail} \approx \frac{1}{1-\alpha}$;

– успішний пошук: $E'_{suc} \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$.

Вартість зростає плавно з α, що дозволяє працювати на вищих заповненнях, ніж у лінійному зондуванні.

– Поведінка щодо кешу. Стрибки визначаються Δ і менш передбачувані для попереднього вибіркового завантаження; на малих α різниця з лінійним зондуванням незначна, але на високих α виграш від меншої кластеризації переважає втрати від гіршої локальності.

Переваги/обмеження. Кращий компроміс між стійкістю до кластерів і прийнятною локальністю; потребує другої хеш-функції та коректної взаємної простоти Δ і модуля.

Узагальнення. Лінійне зондування забезпечує найкращу кеш-ефективність і мінімальні накладні витрати на індексацію, але страждає від первинної

кластеризації та різкого зростання довжин зондувань на високих α. Квадратичне зондування зменшує первинні кластери, проте зберігає вторинні та накладає обмеження на параметри. Подвійне хешування надає найстабільніший розподіл зондувань серед розглянутих схем і краще масштабується за α, ціною другої хеш-функції та дещо гіршої передбачуваності доступів для кеш-підсистеми. Вибір конкретної схеми має виходити з цільового профілю доступів, припустимого коефіцієнта заповнення, вимог до латентності і можливості періодичного перехешування.

2.3.7 Хешування «Робін Гуд»

Походження та ідея. Алгоритм Robin Hood Hashing був уперше системно описаний у класичній роботі Педро Целіса [9], де запропоновано принцип «перерозподілу багатства» у відкритій адресації: кожен новий елемент, що прибуває здалеку, має право відсунути елемент, який «живе ближче до дому», аби вирівняти довжини зондових послідовностей. Формально це реалізується через підтримку метрики DIB (distance to initial bucket) — відстань до «домашнього» кошика. Під час вставлення елемент з більшим DIB обмінюється місцями з елементом із меншим DIB, після чого процес триває далі з витісненим записом (його DIB збільшується). Така стратегія зменшує дисперсію довжин пошуку, порівняно з класичним лінійним зондуванням, і забезпечує кращу концентрацію витрат на пошук при високих коефіцієнтах заповнення.

Алгоритмічні властивості.

– Відкрита адресація з «Робін Гуд» вирівнює хвости розподілу довжин зондування: середня вартість пошуку залишається сталою, а варіативність істотно нижча, ніж у звичайного лінійного зондування.

– Інваріант DIB уздовж зондової послідовності дає корисний критерій раннього завершення пошуку: якщо поточний DIB менший за потрібний, подальший перегляд кошиків марний.

– На практиці алгоритм добре працює при високих рівнях завантаження таблиці (α≈0.8–0.9), зберігаючи прийнятні очікувані витрати пошуку; ціна — більше записів/обмінів на вставленні.

Видалення без «надгробків»: зсув назад. Класична відкрита адресація часто застосовує надгробки — позначки видалення, що накопичуються і деградують час пошуку. Для Robin Hood характерною та ефективною є стратегія зсуву назад [10]:

а) після вилучення елемента утворюється порожній слот;

б) послідовно зсуваються ліворуч сусідні записи, поки їхній DIB > 0, кожне переміщення зменшує їхній DIB на одиницю;

в) зсув зупиняється, щойно досягається пустий слот або запис з DIB == 0.

Такий зсув зберігає характеристики алгоритму «Робін Гуд», усуває потребу в надгробках і стабілізує час пошуку в довгих сесіях із активними видаленнями. У профілях, де читання переважає, накладні витрати зсуву назад на рідкісних видаленнях помірні, зате виграш у відсутності накопичених надгробків — значний.

Простота реалізації та модифікованість. Базовий Robin Hood із DIB та зсувом назад реалізується компактно (масив ключів/значень + масив хешів або об’єднана структура запису), без допоміжних метаданих і SIMD-групування. Це суттєво спрощує:

– інтеграцію з існуючими структурами (напр., вимога не зберігати повні ключі, покладатися лише на хеш/локальний контекст),

– додаткові політики очищення «застарілих» записів під час зондування (евікція за часовими мітками чи лічильником доступів),

– експериментування з альтернативними правилами вставлення (наприклад, обмеження максимально допустимого DIB для запобігання довгим ланцюжкам) та зі стратегіями перехешування.

Обмеження та практичні зауваги.

– Вставлення в «щільних» таблицях може спричиняти каскади обмінів; для профілів із інтенсивними вставленнями варто знизити цільову α або застосувати перерозподіл раніше.

– Паралельний доступ потребує зовнішньої синхронізації або ділення на регіони; відкриту адресацію з Robin Hood складно зробити тонко гранульованою та вільною від замків паралельного доступу без суттєвого ускладнення.

– Для уникнення порушень інваріантів критично правильно оновлювати DIB у кожній операції.

Підсумок. Robin Hood Hashing у поєднанні зі зсувом назад пропонує сприятливий баланс: рівномірні довжини пошуку, відсутність деградації від надгробків та відносно проста, прозора реалізація. На відміну від більш важких SIMD-орієнтованих SwissTable-варіантів, базовий алгоритм легко модифікувати під специфічні потреби дослідження та інтегрувати в архітектурні обмеження бібліотеки користувацького інтерфейсу.

2.3.8 Хешування «швейцарська таблиця»

Походження та базова ідея. Під назвою «швейцарська таблиця» (Swiss Table) зазвичай мають на увазі сімейство високопродуктивних хеш-карт із відкритим адресуванням і груповим зондуванням, у яких доступ пришвидшується завдяки компактним службовим байтам і порівнянням цілих груп слотів за один крок процесора. Ключові ідеї: зберігання для кожного слота короткої частини хешу, організація окремого масиву керувальних байтів та перегляд контрольних байтів групами фіксованого розміру із застосуванням векторних інструкцій.

Структура пам'яті. Таблиця містить два паралельні масиви: масив записів (ключ+значення) та масив керувальних байтів. Кожен керувальний байт кодує стан слота (порожній, видалений або зайнятий) і, у випадку зайнятого слота, зберігає кілька старших або молодших бітів хешу як короткий відбиток. Таке розділення дає змогу спочатку фільтрувати кандидатів за легкими порівняннями керувальних байтів, виконуючи доступ до реальних ключів лише для невеликої підмножини позицій.

Групове зондування та векторні порівняння. Керувальні байти організуються групами сталої ширини GGG (типово 16), що відповідає ширині регістру SIMD на цільовій архітектурі. При пошуку обчислюється індекс

початкового кошика $_{i0}$ (звичайно через маску за ступенем двійки) та короткий «відбиток» h_2 з хешу ключа. Далі відбувається циклічний перегляд груп керувальних байтів; кожен крок виконує порівняння всіх GGG байтів групи з h_2 однією векторною операцією, формуючи маску збігів. Перевірці ключів підлягають лише позиції, де «відбиток» збігся. Наявність у групі хоча б одного «порожнього» маркера є сигналом, що далі за поточною послідовністю зондування шуканого ключа немає.

Послідовність зондування. Для зменшення кластеризації використовується квадратичне зондування групами. Типовою є схема зростання відстані між послідовними групами:

$$i_k = \left(i_0 + \frac{k(k+1)}{2} \right) mod \text{ capacity}, \tag{2.5}$$

де k=0,1,2,… — номер кроку зондування групи. Кроки відбуваються цілими групами GGG слотів, що добре узгоджується з апаратною векторизацією.

Вставлення. Для вставлення також переглядаються групи керувальних байтів, шукаючи в них позначку «порожній» або «видалений». Після вибору позиції вносяться ключ і значення, а в керувальний байт записується відбиток. Завдяки груповому перегляду та локальності даних середня кількість довільних звернень до пам’яті невелика навіть за високого завантаження.

Видалення. Типовою є стратегія «м’якого» видалення: слот позначається спеціальним маркером «видалений», що зберігає безперервність послідовності зондування для інших ключів. Щоби не накопичувати маркери видалення, реалізація контролює їх частку та ініціює перебудову таблиці (перехешування або очищення) при перевищенні порогів.

Політика росту та навантаження. Потужність таблиці зазвичай підтримується ступенем двійки, що дає просте перетворення хешу в індекс через побітову маску. Порогове завантаження обирається нижчим за 1 (типово близько 0.8–0.875), аби в кожній «смузі» керувальних байтів із ненульовою ймовірністю траплявся порожній слот, що обмежує довжину послідовностей зондування та стабілізує час пошуку.

Переваги:

– Висока пропускна здатність пошуку завдяки порівнянню цілих груп керувальних байтів однією інструкцією та відкладеному доступу до ключів.

– Хороша ефективність використання кешу: керувальні байти компактні й розміщені щільно, отже на один пропуск читається мало даних.

– Стійкість до вироджених розподілів ключів: короткі відбитки відсікають більшість хибних кандидатів до звернення до повних ключів.

– *Обмеження та особливості впровадження:*

– Складність реалізації: потрібні системні оптимізації навколо розміщення керувальних байтів, обробки «вартових» на межах буфера, коректного переходу між групами та контролю порогів переповнення.

– Необхідність двох шляхів виконання: векторизований (SSE/AVX/NEON) і запасний скалярний для середовищ без відповідних інструкцій.

– Видалення через маркери потребує періодичного очищення, інакше погіршується час пошуку.

Застосовність у C#. У середовищі .NET можлива побудова «швейцарської» таблиці з використанням System.Runtime.Intrinsics для векторних порівнянь керувальних байтів. Але цей функціонал не доступний в Unity і натомість є аналог у вигляді Unity.Burst.Intrinsics. Відмінність бібліотеки Burst у тому, що код, який використовує SIMD, має знаходитись у окремій функції і не може бути вбудований у місце виклику. Це викликає певне занепокоєння щодо ефективності роботи такої імплементації. Потрібно забезпечити:

– ступінь-двійкову місткість і додаткові «хвостові» керувальні байти для безпечних невирівняних завантажень груп,

– обчислення «відбитка» на вставленні та пошуку,

– скалярний резервний шлях і автодетекцію доступних інструкцій,

– політику очищення маркерів видалення разом із контролем порогів завантаження.

Порівняння із класичними підходами. Відносно лінійного/квадратичного зондування без групування «швейцарська таблиця» суттєво скорочує кількість

випадкових доступів до пам’яті та перевірок повних ключів завдяки попередньому фільтруванню за керувальними байтами. У профілях із частими пошуками це забезпечує помітно стабільніший час запитів за високих коефіцієнтів заповнення.

Висновок. «Швейцарська таблиця» поєднує відкриту адресацію, групове зондування та векторні операції над керувальними байтами, що разом забезпечує високу швидкодію пошуку та добру ефективність використання кешу за помірної надбудови складності реалізації. Для завдань кешування станів у IMGUI така конструкція є привабливою, особливо коли профіль навантаження домінують читання та перевірки наявності ключів.

2.3.9 Метод оцінки ефективності хеш-карт для задач IMGUI

Мета та постановка задачі. Метою є обґрунтований вибір імплементації хеш-карти для використання в IMGUI бібліотеці, орієнтованій на інтерфейси кінцевих застосунків. Для цього проводиться порівняльне експериментальне дослідження кількох кандидатів за репрезентативним для IMGUI набором операцій і метрик.

Вибір кандидатів та обґрунтування виключень

До порівняння включено:

– System.Collections.Generic.Dictionary<TKey,TValue> (еталон).

– Варіант із відкритим адресуванням на основі Robin Hood (зі зсувом назад при видаленні).

– Варіант «швейцарська таблиця» з SIMD-оптимізаціями (Burst).

Класичні схеми відкритого адресування (лінійне, квадратичне зондування, подвійне хешування) не розглядалися експериментально, оскільки відомі властивості цих підходів за високих коефіцієнтів заповнення та в сценаріях з інтенсивними пошуками поступаються сучасним модифікаціям (Robin Hood, Swiss Table). Таким чином експеримент зосереджено на методах, що мають кращі передумови для стабільного часу пошуку та ефективного використання кешу.

Набір операцій. Оцінюються операції, які домінують у профілях IMGUI:

– вставлення нового елемента;

– успішний пошук (hit);

– неуспішний пошук (miss);

– «пошук або вставлення» з поверненням посилання на значення (GetOrAddValue):

– випадок наявного ключа (повертається посилання на існуюче значення),

– випадок відсутнього ключа (повертається посилання на попередньо зарезервований вільний слот).

Остання операція є критичною для IMGUI, де часто потрібен один прохід: доступ до існуючого стану віджета або ініціалізація нового стану без подвоєних пошуків. У стандартному Dictionary така семантика відсутня і типово реалізується як пара «TryGetValue → Add», що спричиняє два проходи пошуку. Гіпотеза — цей шаблон створює додаткові витрати порівняно з реалізаціями, що надають GetOrAddValue як примітив.

Межі дослідження. Щоб сфокусуватися на гарячих шляхах, з експерименту свідомо виключено:

– автоматичне розширення ємності: для задач IMGUI таблиці створюються з фіксованою місткістю, підібраною під застосунок. Розширення не належить до гарячих ділянок;

– видалення: у цільових сценаріях відбувається рідко, тож оптимізація під нього не є пріоритетом.

Максимальна досяжна завантаженість як критерій також не оптимізується. Натомість завантаженість контролюється в робочому для UI-навантажень діапазоні (орієнтовно $\alpha \approx 0.75–0.85$), що відповідає стабільному часу кадру. Загальна кількість записів не перевищує 10 тис., типові розміри значення — близько 32 байтів, що узгоджується з обсягами довідкових структур IMGUI. За цих умов сумарні обсяги пам’яті на дві таблиці не перевищують ≈2.5 МіБ, що є незначним для цільових платформ з об’ємом оперативної пам’яті не менше 2 ГіБ.

Дані, ключі та макети зберігання. Ключі моделюються як «малі» (4 байти) — це відповідає використанню коротких числових маркерів або попередньо

обчислених хешів рядкових ідентифікаторів віджетів. Значення мають «помірний» розмір (≈32 байтів), відображаючи типові записи стану/геометрії. Для варіанта Robin Hood додатково порівнюються два макети:

– «масив записів» (ключ, хеш, значення разом) — краща простота та локальність послідовного читання;

– «розділені масиви» (ключ/хеш окремо від масиву значень) — гіпотеза: швидший пошук за рахунок компактнішого індексного шару та меншої кількості промахів по кеш-лініях при первинному зондуванні.

Метрики та представлення результатів. Основна метрика — середній час на операцію.

$$\bar{t}_{op} = \frac{1}{n} \sum_{i=1}^{n} t_i^{(op)}, \tag{2.6}$$

де $\bar{t}_{op}$ — середній час виконання однієї операції заданого типу; n — кількість вимірів; $t_i^{(op)}$ — час i-го виміру для цієї операції.

Нормування: час Dictionary<TKey,TValue> приймається за 1.0; результати інших реалізацій подаються у відносних одиницях (менше — краще). Така нормалізація забезпечує інтерпретованість для практиків .NET і дозволяє прямо зіставляти виграш над усталеною реалізацією.

Процедура вимірювань і відтворюваність:

– Ініціалізація та розігрів JIT/Burst перед збиранням вибірок.

– Фіксація параметрів середовища (версія .NET/Unity, конфігурація GC, частота та планувальник CPU, прив’язка до ядра, вимкнення турборежимів, єдина політика живлення).

– Генерація ключів і шаблонів доступу з фіксованими зернами псевдовипадковості.

– Для кожного сценарію — ≥30 незалежних прогонів; наведення довірчих інтервалів для середніх [21].

$$s_{op} = \sqrt{\frac{1}{(n-1)} \sum_{i=1}^{n} (t_i^{(op)} - \bar{t}_{op})^2}, \tag{2.7}$$

де s_{op} — вибіркове стандартне відхилення; n — кількість вимірів; $t_i^{(op)}$ — час і-го виміру; $\bar{t}_{op}$ — середній час.

$$CI\,95^{(op)} = \left[\bar{t}_{op} - t_{0.975,\,n-1} \frac{s_{op}}{\sqrt{n}},\ \bar{t}_{op} + t_{0.975,\,n-1} \frac{s_{op}}{\sqrt{n}} \right], \tag{2.8}$$

де $CI95^{(op)}$ — 95% довірчий інтервал для середнього часу операції; $t_{0.975,\,n-1}$ — квантиль розподілу Стьюдента з n−1 ступенями вільності для рівня 0.975; решта змінних як вище.

– Перевірка відсутності артефактів між завданнями (вирівнювання кешів, інтервали «охолодження» за потреби).

Фрагменти коду бенчмарку надані у додатку В.

Гіпотези перевірки:

H1. Реалізації з примітивом GetOrAddValue демонструють перевагу над Dictionary<T> у відповідних сценаріях за рахунок уникнення подвійного пошуку.

H2. У макеті Robin Hood з двома масивами первинне зондування працює швидше завдяки компактнішому індексному шару; різниця нівелюється на шляхах із частими записами.

H3. SIMD-оптимізації «швейцарської таблиці» покращують середній час неуспішних пошуків завдяки груповим порівнянням сигнатур.

H4. За фіксованої місткості без розширень і за завантаження α у діапазоні 0.75–0.85 реалізації на основі Robin Hood і «швейцарської таблиці» демонструють менший час виконання операцій, ніж System.Collections.Generic.Dictionary<TKey,TValue>, у більшості з таких сценаріїв: вставлення, успішний пошук, неуспішний пошук, GetOrAddValue (наявний ключ), GetOrAddValue (відсутній ключ).

Критерії прийняття рішень. Імплементація вважається пріоритетною для бібліотеки, якщо вона:

– статистично значуще випереджає еталон за середнім часом у сценаріях пошуку та GetOrAddValue;

– зберігає простоту інтеграції та можливість подальших модифікацій (зміна політики зондування, політики очищення кешів тексту тощо).

Обмеження та загальна валідність. Результати релевантні для навантажень IMGUI із фіксованою місткістю, відсутністю масових видалень і з ключами малого розміру. Екстремальні режими (α→1, часті розширення, агресивні видалення) не є предметом цього дослідження, оскільки вони не відповідають цільовим використанням у запропонованій архітектурі.

2.4 Аналіз алгоритмів хешування

2.4.1 Контекст і критерії відбору

Для IMGUI-бібліотеки потрібна швидка хеш-функція з якісною рівномірністю для коротких ключів (переважно назви віджетів і компактні текстові фрагменти). Орієнтир — рядки до ≈128 байтів; для великих текстових блоків планується повторне використання ідентифікаторів об’єктів замість хешування всього вмісту. Отже, релевантні алгоритми мають забезпечувати низьку латентність на коротких вхідних даних і стабільну рівномірність розподілу хешів.

2.4.2 Опис кандидатів

CRC32. Поліноміальний контрольний підсумок, широко доступний, зокрема із апаратною підтримкою CRC32C. Історично застосовується як швидка контрольна сума. Включений до порівняння, оскільки використовується в окремих IMGUI-рішеннях, наприклад, Dear ImGui.

xxHash3. Сучасний представник сімейства xxHash, проєктований для високої швидкості на малих і середніх обсягах даних. Практично важливо, що існує готова реалізація для .NET/Unity — Unity.Collections.xxHash3. Алгоритм застосовується у прикладних системах на базі IMGUI, зокрема RAD Debugger.

rapidhash (варіант nano). Нещодавній алгоритм, заявленою метою якого є перевищення продуктивності високошвидкісних некриптографічних функцій, з окремим акцентом на архітектури сімейства Apple Silicon (M-серії). Існують варіанти, один із яких — nano — орієнтований на короткі ключі (до ≈48 байтів),

що частково збігається з цілями аналізу. На цей час доступна реалізація мовою C; для використання у .NET/Unity потрібне портування на C#.

2.4.3 Порівняльний аналіз опублікованих бенчмарків

CRC32 проти xxHash-родини. Узагальнені публічні порівняння швидкодії на коротких входах показують, що CRC-алгоритми (навіть за наявності апаратної підтримки) зазвичай поступаються сучасним некриптографічним хеш-функціям у сценаріях «короткий ключ → індексація хеш-таблиці». Для xxHash3 на невеликих буферах фіксується суттєво менша латентність порівняно з CRC32; додатковою перевагою є 64-бітна довжина хешу, що зменшує ймовірність збігів, що критично для режимів, де порівняння ключів на рівні рядків свідомо уникається [11].

Таблиця 2.1

Назва алгоритму	**Розмір, бітів**	**Швидкість на коротких вхідних даних**
xxHash3	64	133.1
CRC3	32	57.9

xxHash3 проти rapidhash nano. Зіставлення на коротких ключах (4–16 байтів) демонструє близькі величини латентності, з невеликою перевагою rapidhash на окремих архітектурах і паритетом або мінімальним відставанням на інших. У середньому різниця знаходиться в межах одиниць відсотків, що свідчить про конкурентність обох функцій у цільовому діапазоні довжин [12].

Таблиця 2.2

Алгоритм	**M1 Pro**	**M3 Pro**	**Neoverse V2**	**AMD Turin**	**Ryzen 9700X**
rapidhash	1.79 нс	1.38 нс	2.05 нс	2.31 нс	1.46 нс
xxh3	1.92 нс	1.50 нс	2.15 нс	2.35 нс	1.45 нс

2.4.4 Якість розподілу та колізії

Порівняння якості за критерієм колізій виконуються відносно еталонного очікування для рівномірного розподілу в просторі 64-бітних значень. Імовірність появи будь-якого конкретного 64-бітного хешу дорівнює $p = \frac{1}{2^{64}}$.

Для n незалежних хешувань очікувана кількість пар, що зіткнуться (тобто матимуть однаковий 64-бітний результат), дорівнює добутку числа неповторних пар на імовірність збігу для однієї пари:

$$E = \frac{n(n-1)}{2} \cdot \frac{1}{2^{64}} = \frac{n(n-1)}{2^{65}}, \tag{2.9}$$

де E – математичне сподівання кількості колізій [22, С. 113-118]. Звідси випливає, чому еталонне значення відрізняється для різних загальних обсягів вхідних даних (наприклад, для 62 ГіБ і 100 ГіБ). Очікувана кількість колізій масштабується квадратично зі змінною n: при меншому n очікуваних збігів менше, і навпаки.

Таблиця 2.3

Алгоритм	**Довжина вхідних даних, байтів**	**Сумарна величина хешів, ГіБ**	**Очікувана кількість колізій**	**Отримана кількість колізій**
XxHash3	32	100	312.5	287
rapidhash	32	62	120.1	131

Окремий зріз для коротших ключів (8 байтів) у тотожних умовах свідчить про відсутність збігів у xxHash3 та істотно більшу за еталон кількість збігів у wyhash, попередника rapidhash, що узгоджується з відомою еволюцією цих сімейств щодо якості рівномірності [13]:

Таблиця 2.4

Алгоритм	**Довжина вхідних даних, байтів**	**Сумарна величина хешів, ГіБ**	**Очікувана кількість колізій**	**Отримана кількість колізій**
xxHash3	8	100	312.5	0
wyhash	8	100	312.5	652

Сумарно, для коротких ключів xxHash3 демонструє близькість до еталонних очікувань або кращі показники, тоді як rapidhash, і особливо попередні варіанти сімейства wyhash, інколи фіксує вищі за еталон значення у співставних налаштуваннях. Це важливо для режимів використання хеш-таблиць, де порівняння повних ключів цілеспрямовано уникається, а стійка рівномірність розподілу мінімізує ризик деградації.

2.4.5 Проміжний висновок і вибір для подальшої роботи

З урахуванням сукупних критеріїв: швидкість на коротких ключах, наявність зрілої імплементації для .NET/Unity, стабільність розподілу на опублікованих тестах — доцільно обрати xxHash3 як базову хеш-функцію. У порівнянні з CRC32 xxHash3 забезпечує меншу латентність на коротких вхідних даних; у порівнянні з rapidhash опубліковані дані свідчать про близьку швидкодію без очевидної переваги rapidhash на широкому спектрі архітектур, за наявності помітної екосистемної переваги xxHash3 у вигляді готових засобів для C# та Unity. Додатково, для задач, де пряме порівняння рядкових ключів обходиться з міркувань продуктивності, якість розподілу xxHash3 на коротких ключах є вирішальною. На цій підставі додаткових експериментів у межах роботи не планується; обраний алгоритм відповідає вимогам сценарію IMGUI.

2.5 Обґрунтування методів A/B-порівняння з UI Toolkit

Було прийнято рішення виконати порівняння DiVy з UI Toolkit. Обґрунтування вибору полягає в тому, що UI Toolkit є найбільш актуальним конкурентом для будь-якого потенційного нового рішення в екосистемі Unity. Хоча історично ближчою за архітектурою є вбудована IMGUI, саме вона розробниками платформи не розглядається як повноцінна основа для інтерфейсів кінцевих застосунків. Її типовою сферою використання залишаються внутрішні або відлагоджувальні панелі, і навіть у цьому сегменті вона поступово витісняється UI Toolkit. Бібліотека UGUI хоч і призначена для кінцевих інтерфейсів, однак не забезпечує повністю автоматичного компонування, що унеможливлює пряме співставлення підсистем компонування. Натомість UI Toolkit має необхідний функціонал, тож дозволяє провести репрезентативне порівняння з DiVy. Додатковим аргументом є офіційна рекомендація Unity застосовувати UI Toolkit для побудови інтерфейсів кінцевих застосунків, що на практиці витісняє UGUI в цій ролі.

Предметом A/B-порівняння будуть дві панелі інтерфейсу з максимально узгодженим дизайном та функціональністю, реалізовані відповідно в UI Toolkit та у DiVy. До складу панелей включаються як стандартні віджети, так і декілька спеціалізованих віджетів, відсутніх у стандартному наборі. Структура панелей проектується для охоплення широкого спектра віджетів і варіантів взаємодії з ними, із наявністю анімаційних ефектів взаємодії, які очікувані в сучасних інтерфейсах кінцевого призначення.

Набір віджетів включає:

- кнопки;
- слайдери (повзунки);
- чекбокси (прапорці);
- регіон з вертикальною прокруткою;
- регіон із двонапрямною прокруткою;
- горизонтальну «випадаючу» секцію;

– вертикальну «випадаючу» секцію.

Набір взаємодій включає:

– наведення курсора на елементи;

– натискання;

– перетягування елементів (зокрема повзунків);

– прокручування за допомогою коліщатка миші.

Панель має містити сценарії взаємодії з поступовою зміною компонування елементів усередині та довкола активного елемента. Висувається гіпотеза, що саме такі сценарії є найбільш витратними для бібліотек користувацьких інтерфейсів у процесорному вимірі.

Методи порівняння двох панелей охоплюють такі групи метрик.

а) Порівняння обсягу програмного коду: кількість рядків коду (без коментарів) та кількість символів для аналогічних імплементацій спеціалізованих віджетів. Для довідки також надається порівняння стандартних віджетів, однак без інтерпретаційних висновків: через ширший функціональний обсяг у зрілої бібліотеки очікується більший кодовий слід. Відповідно, очікувана більша кількість рядків у стандартних віджетах UI Toolkit не є показником меншої інженерної якості, а відображає широту підтримуваної функціональності. Натомість порівняння спеціалізованих віджетів із контрольованим паритетом можливостей є репрезентативним. Гіпотеза: DiVy потребуватиме меншого обсягу коду (рядків і символів) для спеціалізованих віджетів, ніж UI Toolkit.

б) Порівняння обсягу коду користувацьких панелей: кількість рядків (без коментарів) та символів у реалізаціях двох цільових панелей. У розрахунок включаються лише прикладні частини, що визначають структуру панелі, стилі та зв’язування взаємодії з даними; імплементації віджетів не враховуються. Гіпотеза: DiVy забезпечить менший обсяг коду для побудови панелей порівняно з UI Toolkit.

в) Порівняння метрик навантаження за кадр під час взаємодії. Вимірювання виконуються у оптимізованій збірці на апаратній конфігурації Lenovo Legion 5 Pro:

– центральний процесор — AMD Ryzen 7 5800H;

– графічний процесор— Nvidia GeForce RTX 3070 Laptop;

– обсяг оперативної пам’яті — 32 ГіБ;

– операційна система — Windows.

Для нормування метрик проводяться базові вимірювання в «порожній» збірці, яка запускає обов’язкові підсистеми Unity без жодної з порівнюваних панелей. Отримані значення використовуються як еталон для подальших відносних порівнянь, що дозволяє вирахувати вклад саме UI-підсистем у загальний час кадру, виключивши роботу незмінних механізмів ядра рушія.

У всіх експериментах вертикальна синхронізація вимкнена, а цільова частота кадрів встановлена у 100000 кадрів за секунду (номінальний бюджет 0.1 мс на кадр). Така конфігурація обрана для виключення станів очікування потоком рендерингу: коли продуктивність CPU перевищує частоту оновлення дисплея, очікування вертикального оновлення маскує пікові навантаження, «вирівнюючи» облік часу. Підібране значення цільової частоти забезпечує, що навіть порожня збірка не вкладається в бюджет, а отже виміряний час не забарвлений періодами очікування.

У порівняннях використовуються такі метрики.

– Коефіцієнт варіації кількості циклів процесора у головному потоці за кадр (відносно базового запуску).

$$CV = \frac{\sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(y_i - \bar{y})^2}}{\bar{y}}, \tag{2.10}$$

де CV — коефіцієнт варіації; N — кількість кадрів; y_i — нормалізовані цикли; $\bar{y}$ — середня кількість циклів процесора за кадр. Ціль DiVy — зниження варіативності вартості кадру; отже стабільність є пріоритетнішою за мінімальні чи середні значення. Гіпотеза: у більшості сценаріїв коефіцієнт варіації буде меншим для DiVy.

– Умовна вартість під ризиком 1% (CVaR) кількості циклів процесора у головному потоці за кадр (відносно базового запуску) [20].

$$CVaR_{1\%}=\frac{1}{m}\sum_{k=N-m+1}^{N} y^{(k)}, m=ceil(0.01\,N), \quad (2.11)$$

де $CVaR_{1\%}$ — середнє значення у верхньому 1% «хвоста»; m — кількість кадрів у хвості; $y^{(k)}$ — k-та порядкова статистика; N — кількість кадрів. CVaR характеризує середній рівень навантаження у «хвості» розподілу (найгірші 1% кадрів), що краще відображає досвід користувача від затримок, ніж одиничний максимум. Вибір CVaR також зумовлений варіативністю навіть порожньої збірки, де різні підсистеми рушія виконують нерівномірні обсяги роботи. Значення 1% прийняте як у галузевих джерелах, що аналізують довгі «хвости» часу кадру [14, 15]. Гіпотеза: CVaR буде меншим для DiVy в більшості сценаріїв.

– Середня кількість циклів процесора у головному потоці за кадр (відносно базового запуску).

$$\mu_0=\frac{1}{N_0}\sum_{j=1}^{N_0} b_j, \quad (2.12)$$

де μ_0 — середнє значення циклів процесора за кадр у базовому запуску; N_0 — кількість кадрів у базовому запуску; b_j — кількість циклів процесора в j-му кадрі базового запуску.

$$y_i=x_i-\mu_0, \quad (2.13)$$

де y_i – середнє значення циклів процесора за кадр в і-му сценарії відносно базового запуску, x_i – середнє значення циклів процесора за кадр в і-му сценарії, μ_0 — середнє значення циклів процесора за кадр у базовому запуску. Ця метрика не є визначальною для оцінювання досягнення головної цілі (зменшення затримок у важких сценаріях), оскільки протягом більшої частини часу тестові сцени перебувають у відносно легких режимах. Висувається гіпотеза, що UI Toolkit матиме нижчі середні витрати в багатьох окремих сценаріях, що відповідає характеру підходу RMGUI, у якому за відсутності взаємодії відбувається менше роботи.

– Середня кількість циклів процесора у головному потоці за кадр (відносно базового запуску) у сценарії з активною взаємодією з інтерфейсом. Тут гіпотеза

протилежна: за умов постійної взаємодії (анімації та зміни компонування протягом більшої частини часу) однопрохідне оновлення IMGUI призводить до менших середніх витрат за кадр, тоді як RMGUI, який мінімізує роботу у стані спокою, демонструє підвищені витрати саме під час частих змін структури й компонування. Обстоюється позиція, що вибір архітектури має орієнтуватися на найскладніші та найпоширеніші для цільової програми сценарії, а не на найлегші.

– Сумарний обсяг пам’яті, виділеної та зареєстрованої збирачем сміття за кадр понад середній рівень у стані спокою.

$$\Delta G_{\text{sum}} = \sum_{i=1}^{F} max\left(0, G_i - \overline{G}_{idle}\right), \tag{2.14}$$

де ΔG_{sum} — сумарний обсяг пам’яті, виділеної та зареєстрованої збирачем сміття понад «фоновий» рівень; G_i — обсяг пам’яті у кадрі і для тестованої панелі; $\bar{G}_{idle}$ — середній обсяг пам’яті за кадр у «порожній» збірці; F — кількість кадрів у вибірці. Другорядна ціль — зниження тиску на збирач сміття. Вимірювання у оптимізованих збірках обмежені: у Unity недоступні GC.GetTotalAllocatedBytes(), GC.GetAllocatedBytesForCurrentThread(), GC.CollectionCount(0), тоді як ProfilerRecorder працює лише в налагоджувальному режимі). Абсолютне порівняння обсягів пам’яті є некоректним, натомість оцінюється перевищення над «шумовим» фоном спокійного стану. Припускається, що у налагоджувальному режимі значна частка фонового виділення пам’яті зумовлена метаданими для відладки, тоді як під час інтенсивної взаємодії домінує функціональне виділення пам’яті, пов’язане з розв’язанням задач UI. Такий підхід має обмежену точність, але є індикатором відносних тенденцій за достатнього розриву значень.

У порівняннях навмисно аналізуються процесорні цикли, а не час виконання за кадр. Навіть за відсутності сторонніх вікон у середовищі Windows одночасно працює велика кількість процесів і потоків; планувальник ОС може призупиняти головний потік застосунку, що породжує додаткову дисперсію часових вимірювань без зміни обсягу виконаної роботи. Емпірично це проявляється в тому, що коефіцієнт варіації часу кадру є вищим за коефіцієнт варіації кількості циклів, а окремі кадри мають диспропорційно великий часовий

бюджет без відповідного збільшення циклів. Тому аналіз циклів забезпечує більш стійку основу для порівняння.

Сценарії вимірювань:

– відсутність взаємодії користувача з інтерфейсом;

– наведення курсора на елементи з активацією анімацій наведення;

– переміщення повзунка зі щокадровим округленням значення до цілих (вибрано як навантажувальний приклад із додатковими обчисленнями);

– вертикальна прокрутка панелі зі зміщенням основних елементів;

– двонапрямна прокрутка регіону з великим обсягом тексту;

– розкриття/закриття горизонтальної «випадаючої» панелі (зміщення елементів нижче);

– розкриття/закриття вертикальної «випадаючої» панелі (звуження елементів праворуч);

– зміна теми оформлення;

– сумарний сценарій, у якому користувач послідовно виконує всі вищезазначені дії, моделюючи типову сесію взаємодії.

Для довідки також наводяться:

– середній час кадру (відносно базового запуску);

– умовна вартість під ризиком 1% часу кадру (відносно базового запуску);

– абсолютний максимум часу одного кадру. Оскільки він має велику похибку, він не використовується для порівнянь, але ілюструє верхню межу негативного впливу на сприйняття затримки;

– максимальний обсяг пам’яті, виділеної та зареєстрованої збирачем сміття за кадр.

Окреме обґрунтування застосування інтегральних метрик кадру як противаги суто інструментальним вимірюванням підсистем. Ідеальним варіантом було би маркування всіх релевантних ділянок коду профайлерами та збір окремих метрик для підсистем (наприклад, компонування), але в реальності частина коду UI Toolkit недоступна для інструментування та частково реалізована рідною мовою C++, що унеможливлює такий аналіз. Крім того, у ряді сценаріїв

критичним є внесок графічного процесора. Якщо вузьким місцем стає графічний конвеєр, головний потік очікує завершення робіт, виконуваних на GPU, і тому повні метрики кадру краще відображають загальний вплив сценарію на продуктивність і досвід користувача, ніж ізольовані показники підсистем на головному процесорі.

Витрати на збір метрик були оцінені з метою мінімізації впливу самого процесу вимірювань. Середній час збирання метрик за кадр становив близько 2 мкс, що менше за діапазон подання результатів і менше 1% від середнього часу кадру порожньої збірки; таким чином вплив вимірювань на результати знехтовно малий.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНА ОЦІНКА БІБЛІОТЕКИ IMGUI ДЛЯ UNITY

3.1 Результати аналізу ефективності хеш-карт.

Було виконано порівняння середнього часу операцій для таких імплементацій хеш-карт:

– System.Collections.Generic.Dictionary<TKey,TValue> (еталон);

– варіант із відкритим адресуванням на основі Robin Hood зі зсувом назад під час видалення, з єдиним масивом записів (ключ, хеш, значення разом);

– варіант на основі Robin Hood зі зсувом назад, із розділенням масивів (індексний шар окремо від масиву значень);

– варіант «швейцарська таблиця» з SIMD-оптимізаціями на базі Burst.

Час усіх операцій нормовано відносно Dictionary, який прийнято за 1.0 для кожної категорії. Вимірювання виконувалися в оптимізованій збірці для 10 000 елементів.

3.1.1 Нотатки щодо реалізацій

Для швейцарської таблиці використано апаратно наближені операції порівняння сигнатур за допомогою Burst.Intrinsics з метою групового відсіювання кандидатів під час пошуку. Повна реалізація займає 476 рядків коду.

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
3 usages
private int FindBucketIndex(TKey key, int hash)
{
    var entries = _entries;
    var bucketMask = _bucketMask;
    var controls = _ctrl;
    byte target = H2(hash);

    if (entries.Length == 0) return -1;

    unsafe
    {
        fixed (byte* c0 = &controls[0])
        {
            var ps = new ProbeSeq(H1(hash), bucketMask);
            if (X86.Sse2.IsSse2Supported)
            {
                var needle = Sse2Group.Create(target);
                while (true)
                {
                    var group = Sse2Group.Load(c0 + ps.pos);
                    var match = group.MatchGroup(needle);
                    while (match.AnyBitSet())
                    {
                        int bit = match.LowestSetBitNonZero();
                        match = match.RemoveLowestBit();
                        int idx = (ps.pos + bit) & bucketMask;
                        if (_cmp.Equals(key, entries[idx].Key))
                            return idx;
                    }
                    if (group.MatchEmpty().AnyBitSet())
                        return -1;
                    ps.MoveNext();
                }
            }
            else
            {
                // Fallback scalar path (nuint-width bucket scans)
                while (true)
                {
                    var f = FallbackGroup.Load(c0 + ps.pos);
                    var match = f.MatchByte(target).And(f.MatchFull());
                    while (match.AnyBitSet())
                    {
                        int bit = match.LowestSetBitNonZero();
                        match = match.RemoveLowestBit();
                        int idx = (ps.pos + bit) & bucketMask;
                        if (_cmp.Equals(key, entries[idx].Key))
                            return idx;
                    }
                    if (f.MatchEmpty().AnyBitSet())
                        return -1;
                    ps.MoveNext();
                }
            }
        }
    }
}
```

Рис 3.1 Частина імплементації швейцарської таблиці, що використовує SIMD

У варіанті Robin Hood ключовими є дві процедури: зондування з урахуванням DIB (distance to initial bucket) та зсув назад після видалення. Повна реалізація становить 325 рядків коду.

```
public bool Remove(TKey key)
{
    if (_count == 0) return false;

    int rawHash = key.GetHashCode();
    int hash = GetNonZero(rawHash);
    int init = BucketIndex(rawHash, _indexShift);

    int idxFound = -1;

    for (int i = 0; i < Buckets.Length; ++i)
    {
        int idx = (init + i) & _mask;
        ref readonly Entry e = ref Buckets[idx];

        if (e.Hash == 0) break;
        if (i > e.Dib) break;

        if (e.Hash == hash && e.Key.Equals(key))
        {
            idxFound = idx;
            break;
        }
    }

    if (idxFound < 0) return false;

    // Backshift compaction: shift entries left until a home or empty
    for (int hole = idxFound, next = (hole + 1) & _mask; ; hole = next, next = (next + 1) & _mask)
    {
        ref readonly Entry e = ref Buckets[next];
        if (e.Hash == 0 || e.Dib == 0)
        {
            Buckets[hole] = default;
            --_count;
            return true;
        }

        // Move e back into the hole; DIB decreases by 1
        Entry moved = e;
        moved.Dib -= 1;
        Buckets[hole] = moved;
        Buckets[next] = default;
    }
}
```

Рис 3.2 Частина імплементації алгоритму Robin Hood, що показує використання DIB та зсуву назад

3.1.2 Результати вимірювань

Зведену таблицю з детальними значеннями отриманих результатів, включно з довірчими інтервалами та порівнянням відносно еталону подано у додатку Б.

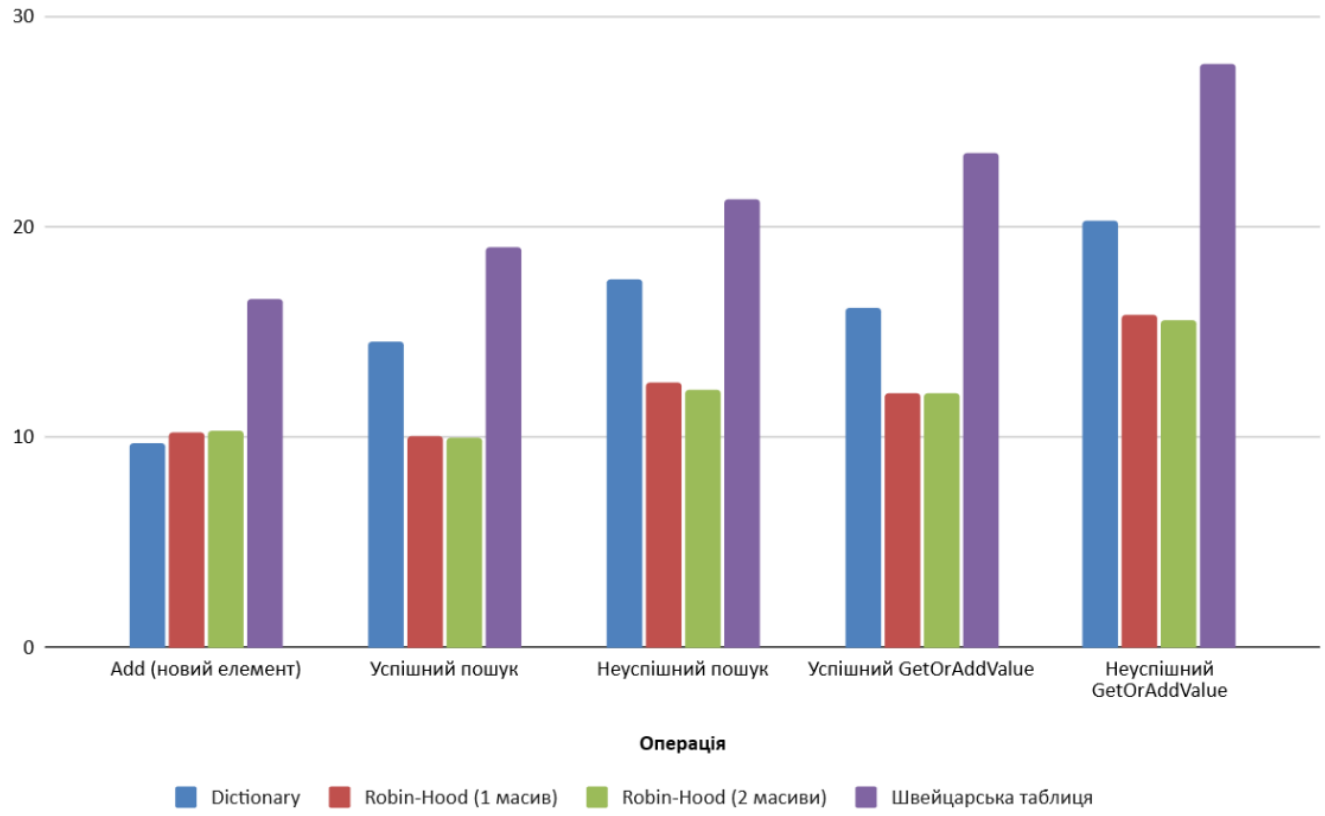


Рис 3.3 Порівняння швидкості виконання операцій між імплементаціями хеш-карт, нс на операцію, менше — краще

Перевірка гіпотез.

H1. Реалізації з примітивом GetOrAddValue мають перевагу над Dictionary<T> у сценаріях, де в останньому потрібні послідовні TryGetValue та Add. Спостереження підтверджують гіпотезу для обох варіантів Robin Hood, але не для швейцарської таблиці. Різниця між одноетапним пошуком та парою TryGetValue → Add у Dictionary виявилася помірною (близько 10–15%). Імовірне пояснення полягає в тому, що після невдалого пошуку відповідні лінії кешу для вставки вже присутні в кеші L1, що зменшує додаткові витрати другої операції. Водночас перевага Robin Hood у GetOrAddValue формується насамперед завдяки швидшому механізму пошуку, а не лише скороченню кількості викликів: відносний виграш у GetOrAddValue (0.75–0.78) менший за виграш у чистому пошуку (0.68–0.72), але стабільно присутній.

H2. У макеті Robin Hood з розділеними масивами первинне зондування очікувалося швидшим завдяки компактнішому індексному шару, тоді як перевага

мала нівелюватися на записах. Гіпотеза частково підтвердилася: фіксується невелика систематична перевага 1–2% у пошуку. Її обмежена величина може пояснюватися тим, що середні довжини послідовностей зондування малі, а пропускна здатність переміщення даних між L1 і регістрами процесора не є домінувальним чинником.

H3. SIMD-оптимізації швейцарської таблиці мали зменшувати середній час неуспішних пошуків завдяки груповому порівнянню сигнатур. За отриманими результатами гіпотеза не підтвердилася для середовища C# та Unity. Можливі причини: відсутність вбудовування викликів із SIMD у місця використання при генерації коду через Burst, а також неідеальна генерація IL або машинного коду для конкретної реалізації на C#. Побічним аргументом виступає і те, що варіант з SIMD зі стандартної бібліотеки C# також не перевищив Dictionary за швидкодією.

https://github.com/ShuiRuTian/SwissTable/issues/1

H4. За фіксованої місткості без розширень і завантаження α у діапазоні 0.75–0.85 реалізації на основі Robin Hood і швейцарської таблиці мали би переважати Dictionary у більшості сценаріїв. Гіпотеза підтвердилася частково: обидва варіанти Robin Hood продемонстрували істотну перевагу у пошуку (успішному та неуспішному), але поступилися у вставці. Швейцарська таблиця в усіх сценаріях була повільнішою за Dictionary.

3.1.3 Проміжні висновки

Для подальшого використання обрано варіант Robin Hood з єдиним масивом записів. За операціями пошуку він продемонстрував прискорення на 22–31% відносно Dictionary, тоді як вставка була повільнішою приблизно на 5%. З огляду на те, що в цільових сценаріях IMGUI пошуки суттєво переважають вставки, загальний час кадру з використанням цього підходу очікувано зменшиться. Швейцарська таблиця в наявній C#-реалізації поступилася як Robin Hood, так і Dictionary; причини фіксованої неефективності для цього середовища можуть стати предметом окремого дослідження. Додатковим міркуванням на користь Robin Hood є менший обсяг коду (325 проти 476 рядків), відсутність залежності

від SIMD та відповідно вища гнучкість для подальших модифікацій базового алгоритму. Попри незначну перевагу варіанта з розділенням масивів у сценаріях пошуку (1–2%), обрано конфігурацію з одним масивом, оскільки зафіксований виграш є незначним, тоді як розділення ускладнює реалізацію та подальшу модифікацію алгоритму.

Приклади модифікацій обраного алгоритму наведені у рисунках 3.4 та 3.5.

```
1 usage
public bool TryGetValue(ulong key, out T value)
{
    int init = BucketIndex(key, _indexShift);

    for (int i = 0; i < Buckets.Length; ++i)
    {
        int idx = (init + i) & _mask;
        ref readonly Entry e = ref Buckets[idx];

        if (e.Value.Equals(default)) break;          // empty bucket ⇒ not found
        if (i > e.Dib) break;                        // we passed where it could be

        if (e.Key == key)
        {
            value = e.Value;
            return true;
        }
    }

    value = default!;
    return false;
}
```

Рис 3.4 Пошук зі встановленням відповідності лише за хешем без порівняння первинних ключів

```
1 usage   SolidAlloy *
public ref CachedTextMeasure GetOrAddValue(string text, bool isStableInstance, int fontID, float fontSize, out bool exists, out int index)
{
    if (NeedsResize(_count + 1)) Resize(Buckets.Length << 1);

    ulong hash = GetHash(text, isStableInstance, fontID, fontSize);
    int init = BucketIndex(hash, _indexShift);

    // We'll insert this if missing; DIB will be set when placed.
    var entryToInsert = new Entry { Dib = 0, Hash = hash, Text = text, IsStableInstance = isStableInstance,
                                    FontID = fontID, FontSize = fontSize, Value = default };
    int placedIndex = -1;

    // No stop condition may sound scary but we made sure we have enough space at the start of the method,
    // so we certainly have enough free slots to place a new element in.
    for (int i = 0, currentDib = 0;
         ;
         ++i, ++currentDib)
    {
        int idx = (init + i) & _mask;
        ref var e = ref Buckets[idx];

        // Do this only while we haven't placed the newcomer yet to avoid invalidating 'placedIndex' by later back-shifts.
        if (placedIndex == -1)
        {
            while (e.Value.MeasuredWordsStartPtr.IsNotNull
                   && _frameCount - e.Value.LastFrameAccessed > 2)
            {
                RemoveAtIndex(idx);
                e = ref Buckets[idx]; // re-examine the slot
            }
        }

        if (e.Value.MeasuredWordsStartPtr.IsNull)
        {
            // place new
            entryToInsert.Dib = currentDib;
            e = entryToInsert;
            ++_count;

            // we displaced the original, remembered its placed index, so returning placedIndex
            // if placedIndex == -1, we didn't move anything
            exists = false;

            if (placedIndex == -1)
            {
                index = idx;
                return ref e.Value;
            }
            else
            {
                index = placedIndex;
                return ref Buckets[index].Value;
            }
        }

        if (e.Hash == hash && e.FontID == fontID && e.FontSize == fontSize && e.IsStableInstance == isStableInstance
            && (isStableInstance ? ReferenceEquals(e.Text, text) : e.Text.Equals(text, StringComparison.Ordinal)))
        {
            exists = true;
            index = idx;
            return ref e.Value;
        }

        // Robin-Hood: newcomer steals if its DIB (i) > resident's DIB
        if (currentDib > e.Dib)
        {
            // swap: newcomer takes slot with its current DIB (i)
            var tmpEntry = e;
            e = entryToInsert;
            e.Dib = currentDib;

            if (placedIndex == -1)
                placedIndex = idx;

            // displaced continues probing with its own DIB (tmp.Dib)
            entryToInsert = tmpEntry;
            currentDib = entryToInsert.Dib; // continue from resident's DIB
        }
    }
}
```

Рис 3.5 Пошук із опортуністичним видаленням застарілих записів під час проходу зондування

3.2 Важливі деталі імплементації бібліотеки IMGUI.

У цьому підрозділі подано опис підсистем, зовнішнього програмного інтерфейсу та структур даних, які суттєво відрізняють DiVy від наявних аналогів і безпосередньо сприяли досягненню цілей щодо зменшення обсягу коду для побудови інтерфейсів і підвищення їх продуктивності.

3.2.1 Зберігання інформації про елементи інтерфейсу

Базовою одиницею подання інтерфейсу в бібліотеці є структура UIBox, що інкапсулює повний набір атрибутів елемента, релевантних для розробника під час опису інтерфейсу користувача.

Єдине контейнерне подання властивостей у формі однієї структури, на відміну від ієрархії класів із наслідуванням, переслідує кілька взаємопов’язаних цілей:

– Спрощення зовнішнього програмного інтерфейсу. Уніфікація точок взаємодії зменшує кількість функцій і типів, з якими оперує розробник, знижуючи когнітивне навантаження на етапі ознайомлення та під час повсякденного використання.

– Прогнозованість використання пам’яті та розміщення даних. Фіксований розмір базового будівельного блока дає можливість наперед оцінювати необхідний обсяг пам’яті для інтерфейсів різної складності. Екземпляри структури (елементи інтерфейсу) розміщуються в єдиному списку, що спрощує міркування про локальність даних і розташування в пам’яті, а також полегшує серіалізацію для збереження й відновлення стану інтерфейсу. Однаковий розмір і компактне зберігання в одному масиві усувають потребу в частих виділеннях керованої пам’яті для кожного елемента, що позитивно позначається на роботі збирача сміття в середовищі C#. Крім того, це дозволяє наперед створювати список із достатньою місткістю на весь час роботи застосунку, уникаючи додаткових виділень під час розширення.

```
[StructLayout(LayoutKind.Auto)]
169 usages  SolidAlloy *  2 extension methods  5 exposing APIs
public partial struct UIBox
{
    public ImageTextUnion ImageTextUnion;
    public string Text; // can't store it in the union because it's a managed object.
    public CornerColors Tint;

    public UIFlags Flags;
    public int LastTouchedFrame;
    public Ptr<UIBox> Parent;
    public Ptr<UIBox> FirstChild;
    public Ptr<UIBox> NextSibling;
    public Ptr<UIBox> Self;
    public ID ID;
    public int TreeDepth; // It's not used right now, but we are going to use it for sorting.

    public float AnimT;
    public float CustomAnimT; // A field you can use for the custom animations. Set it to 0 and 1, and lerp yourself.

    public float2 RelativeOffset;
    public float2 RelativeOffsetTarget;
    public float2 MoveDelta;

    public Vector2<SizeConstraintUnion> SizeConstraints;
    public UIRect ScreenRect;
    internal float2 MinSize;

    public float AspectRatio; // float / height. If 0 → no aspect ratio.
    public Padding Padding;
    public ushort ChildGap;

    1 usage  new *
    public readonly Actions Actions // This property is useful in case you want to check multiple flags at once
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get => Flags.GetActions();
    }

    1 usage  new *
    public readonly bool PressedThisFrame => Flags.IsAnyFlagEnabled(UIFlagsEx.Action_Pressed);
    5 usages  new *
    public readonly bool Clicked          => Flags.IsAnyFlagEnabled(UIFlagsEx.Action_Clicked);
    1 usage  new *
    public readonly bool DoubleClicked    => Flags.IsAnyFlagEnabled(UIFlagsEx.Action_DoubleClicked);
    1 usage  new *
    public readonly bool RightClicked     => Flags.IsAnyFlagEnabled(UIFlagsEx.Action_RightClicked);
    1 usage  new *
    public readonly bool HoldClicked      => Flags.IsAnyFlagEnabled(UIFlagsEx.Action_HoldClicked);

    5 usages  new *
    public readonly bool IsPressed        => Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_Pressed);
    1 usage  new *
    public readonly bool IsHovered        => Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_MouseOver);
    1 usage  new *
    public readonly bool IsMouseOver      => Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_Hovered);

    4 usages  new *
    public readonly bool IsFirstFrame => Flags.IsAnyFlagEnabled(UIFlagsEx.IsFirstFrame);

    8 usages  new *
    internal readonly bool IsPersistent => ID.Hash != 0;

#if DEBUG
    new *
    public override readonly string ToString() => $"{ID.DebugString}({(IsPersistent? "persistent" : "transient")})";
#endif
}
```

Рис 3.6 Структура UIBox

– Підвищення ефективності апаратного передвибіркового завантаження та підтримка паралельних обчислень. Однорідність записів і лінійне розміщення у пам’яті забезпечують прогнозований шаблон доступу під час послідовних ітерацій, що покращує роботу механізмів передвибіркового завантаження. Це також спрощує організацію паралельної обробки: кожна паралельна задача отримує безперервний відрізок однорідних елементів.

Для скорочення використання пам’яті структурою застосовано два взаємодоповнювальні підходи:

а) Дискретне об’єднання для подання даних зображення або тексту. В елементі одночасно активний лише один із двох видів вмісту; відповідні поля зберігаються в об’єднанні ImageTextUnion. Запис для зображення містить посилання на текстуру, параметри заокруглення кутів, ширини обвідки тощо; запис для тексту — посилання на шрифт, розмір, правила перенесення рядків тощо. Така схема уникає дублювання нерелевантних полів.

б) Кодування багатозначних варіантів у єдиному бітовому полі UIFlags. До нього входять, зокрема, ознака «маски» (відсікання відображення дочірніх елементів поза межами контейнера), параметри вирівнювання тексту, а також інформація про інтерактивність: можливість натискання, наведення курсора, перетягування та ін.

```
221 usages  SolidAlloy *  26 extension methods  86 exposing APIs
[Flags] public enum UIFlags : ulong
{
    // bit 0
    LayoutDirection_Horizontal = 0 << 0,
    LayoutDirection_Vertical   = 1 << 0,

    // bit 1
    LayoutOrder_Normal   = 0 << 1,
    LayoutOrder_Reversed = 1 << 1,

    // bis 2-3
    CrossChildAlignment_Left     = 0 << UIFlagsEx.CrossAlignOffset,
    CrossChildAlignment_Center   = 1 << UIFlagsEx.CrossAlignOffset,
    CrossChildAlignment_Right    = 2 << UIFlagsEx.CrossAlignOffset,
    CrossChildAlignment_Top      = CrossChildAlignment_Left,
    CrossChildAlignment_Bottom   = CrossChildAlignment_Right,
    VerticalTextAlignment_Top    = CrossChildAlignment_Top,
    VerticalTextAlignment_Center = CrossChildAlignment_Center,
    VerticalTextAlignment_Bottom = CrossChildAlignment_Bottom,

    // bits 4-5
    MainChildAlignment_Start        = MainChildAlignment.Start << UIFlagsEx.MainAlignmentOffset,
    MainChildAlignment_End          = MainChildAlignment.End << UIFlagsEx.MainAlignmentOffset,
    MainChildAlignment_Center       = MainChildAlignment.Center << UIFlagsEx.MainAlignmentOffset,
    HorizontalTextAlignment_Left    = MainChildAlignment_Start,
    HorizontalTextAlignment_Right   = MainChildAlignment_End,
    HorizontalTextAlignment_Center  = MainChildAlignment_Center,
    // We don't handle Justify for the child alignment, but we will in the future.
    HorizontalTextAlignment_Justify = 3 << UIFlagsEx.MainAlignmentOffset,

    // bits 6-8
    MoveConstraint_FreeMovement = MoveConstraint.FreeMovement, // means we can drag even diagonally
    MoveConstraint_OnlyOnAxes   = MoveConstraint.OnlyOnAxes, // means we can only drag horizontall or vertically, not diagonally.
    MoveConstraint_Horizontal   = MoveConstraint.Horizontal,
    MoveConstraint_Vertical     = MoveConstraint.Vertical,
    MoveConstraint_Fixed        = MoveConstraint.Fixed,

    // bits 9-10
    RelativeOffsetClamp_None    = 0 << UIFlagsEx.RelativeOffsetClampOffset,
    RelativeOffsetClamp_Abrupt  = 1 << UIFlagsEx.RelativeOffsetClampOffset,
    #if HIDE_UNSUPPORTED_FLAGS
    RelativeOffsetClamp_Elastic = 2 << UIFlagsEx.RelativeOffsetClampOffset,
    #endif

    // bits 11-12
    SelfAlign_Inherit = 0 << UIFlagsEx.SelfAlignOffset,
    SelfAlign_Left    = 1 << UIFlagsEx.SelfAlignOffset,
    SelfAlign_Center  = 2 << UIFlagsEx.SelfAlignOffset,
    SelfAlign_Right   = 3 << UIFlagsEx.SelfAlignOffset,
    SelfAlign_Top     = SelfAlign_Left,
    SelfAlign_Bottom  = SelfAlign_Right,

    // NOTE: bits 13-17 are reserved for the actions. They must be inside the 31 range to be converted to int easily.

    // 18-21 reserved for the current state
    State_Disabled = State.Disabled << UIFlagsEx.CurrentStateOffset,

    // bits 22-25 are reserved for the prev frame state.

    // bits 26-29 are reserved for the old state.

    Ability_Clickable = 1 << 30,
    Ability_Hoverable = 1UL << 31,

    // bits 32-33
    Draggable_Delta = Draggable.Delta,
    Draggable_View  = Draggable.View,

    // bits 34-35
    Scrollable_Delta = 1UL << UIFlagsEx.ScrollableOffset,
    Scrollable_View  = 2UL << UIFlagsEx.ScrollableOffset,
```

Рис 3.7 Частина біт-поля UIFlags

3.2.2 Основні функції побудови ієрархії інтерфейсу

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
Frequently called  21 usages  new *
public static ref UIBox DoBox(ID id, SizeConstraint width = default, SizeConstraint height = default, UIFlags flags = default,
                              CornerColors tint = default, float aspectRatio = 0f, Padding padding = default,
                              ushort childGap = default, Sprite sprite = null, Rect customTextureRect = default, float cornerRadius = 0,
                              float edgeSoftness = 1f, float borderThickness = 0f, Color32 borderColor = default)
{
    ref UIBox box = ref StartBox(id, width, height, flags, tint, aspectRatio, padding, childGap, sprite, customTextureRect,
                                 cornerRadius, edgeSoftness, borderThickness, borderColor);

    EndBox();
    return ref box;
}
```

Рис 3.8 Функція зовнішнього інтерфейсу DoBox

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
Frequently called  36 usages  new *
public static Scope BoxScope(ID id, SizeConstraint width = default, SizeConstraint height = default, UIFlags flags = default,
                             CornerColors tint = default, float aspectRatio = 0f, Padding padding = default,
                             ushort childGap = default, Sprite sprite = null, Rect customTextureRect = default, float cornerRadius = 0,
                             float edgeSoftness = 1f, float borderThickness = 0f, Color32 borderColor = default)
{
    ref UIBox box = ref StartBox(id, width, height, flags, tint, aspectRatio, padding, childGap, sprite, customTextureRect,
                                 cornerRadius, edgeSoftness, borderThickness, borderColor);
    return new(box.Self);
}
```

Рис 3.9 Функція зовнішнього інтерфейсу BoxScope

```
Frequently called  13 usages  SolidAlloy *
public static ref UIBox DoText(ID id, string text, bool isStableInstance, SizeConstraint width = default, SizeConstraint height = default,
                               UIFlags flags = default, Color32 tint = default, float aspectRatio = 0f,
                               Padding padding = default, MSDFFontAsset font = default, float fontSize = default)
{
    tint = (tint is {r: 0, g: 0, b: 0, a: 0} && flags.IsAnyFlagDisabled(UIFlags.Tint_OverrideTransparent)) ? COLOR_WHITE : tint;
    ref UIBox newBox = ref GetOrCreateUIBox(id, width, height, flags, CornerColors.Uniform(tint), aspectRatio, padding, 0, false);

    newBox.ImageTextUnion.Text.Font = font == default ? _defaultFont : font;
    newBox.Text = text ?? string.Empty; // just so that we don't have to check for null later.
    newBox.ImageTextUnion.Text.FontSize = fontSize == default ? _defaultFontSize : fontSize;

    ulong isStableInstanceUlong = Unsafe.As<bool, byte>(ref isStableInstance);
    newBox.Flags = newBox.Flags.AddFlag((UIFlags)(isStableInstanceUlong << UIFlagsEx.TextIsStableInstanceOffset));
    newBox.Flags = newBox.Flags.AddFlag(UIFlagsEx.BoxType_Text);

    ExitScopes(newBox.Self, in newBox); // run this immediately for text because it cannot have children.

#if DEBUG
    if (!isStableInstance && newBox.Text.Length > 64)
    {
        Debug.LogWarning($"Text box {id.DebugString} with text \"{text}\" was created with unstable instance and has length over 64. "
                         + "This will cause lower hashing performance. "
                         + "Storing it in a member field and passing with isStableInstance=true is strongly advised.");
    }
#endif

    return ref newBox;
}
```

Рис 3.10 Функція зовнішнього інтерфейсу DoText

Зовнішній інтерфейс побудовано навколо трьох базових викликів:

– DoBox — створює елемент ієрархії, що може бути заповнений суцільним кольором, градієнтом або зображенням.

– DoText — створює елемент ієрархії для відображення текстового вмісту.

– BoxScope — відкриває регіон, усередині якого декларуються дочірні елементи.

Відсутність функції на кшталт TextScope означає, що текстові елементи не виступають батьківськими для інших елементів. На основі цих трьох примітивів і опціональних аргументів формується повний набір сучасних інтерфейсних патернів для кінцевих застосунків (приклади наведено далі). Решта функцій виконують допоміжні ролі (напр., обчислення станів анімованих властивостей) і не беруть участі у побудові ієрархії. Обмеження базового API трьома операціями знижує бар'єр входу та спрощує подальшу підтримку коду.

3.2.3 Структура кадру

Життєвий цикл кадру має чітку поетапну організацію:

– Збирання даних про ввід (GatherInputData). Виокремлення цього етапу зумовлене можливістю постачання подій вводу не лише безпосередньо користувачем у поточний момент часу, а й із зовнішніх джерел (наприклад, попередньо записані сценарії для автоматизованого тестування).

– Початок кадру (FrameStart). На цьому етапі інтерпретується отриманий ввід (як з інтерактивних пристроїв, так і з альтернативних джерел), формується стек елементів, потенційно залучених до взаємодії, впорядкований за видимістю на екрані (верхній елемент — останній відмальований). Перевіряються умови інтеракції елементів у стеку та виставляються відповідні ознаки подій. Наприклад, коли елемент доступний для натискання та не перекритий іншими, встановлюється прапорець, що згодом перевіряється розробником через властивість IsPressed.

– Середина кадру. На цьому кроці прикладний код викликає функції побудови ієрархії (зокрема DoBox і DoText), формуючи дерево елементів поточного кадру.

– Кінець кадру (FrameEnd). Задіяний алгоритм автоматичного компонування визначає фінальні розміри та розташування елементів. Після цього

формуються групи відправлення на відтворення та прив’язуються параметри матеріалів (зокрема текстури).

– Відправлення даних на відтворення (RecordRenderGraph). Сформовані групи перетворюються на виклики рендер-ядра рушія, наприклад, через CommandBuffer.DrawProcedural(). По-перше, цей етап виділено окремо відповідно до вимог Unity щодо рознесення логіки та команд відтворення. По-друге, частота основного циклу бізнес-логіки може відрізнятися від частоти надсилання команд на GPU: наприклад, логіка виконується із фіксованою частотою 60 кадрів на секунду, тоді як відображення відбувається настільки часто, наскільки дозволяє поточний графічний процесор.

3.2.4 Алгоритм ідентифікації елементів та хешування назв

Ідентифікація елементів здійснюється за їхнім шляхом у дереві. Шлях включає власну назву елемента та назви його безпосередніх батьківських контейнерів. Ідентифікатор потрібний не всім елементам, а лише тим, стан яких має зберігатися між кадрами. Типові підстави для ідентифікації:

– елемент є інтерактивним (натискання, наведення, перетягування тощо);

– елемент бере участь в анімації;

– необхідно отримати координати відображення елемента з попереднього кадру (рідкісний випадок).

На першому етапі обчислюється хеш назви елемента без урахування батьківських контейнерів. Для цього використовується xxHash3; значення 0 зарезервоване для неінтерактивних елементів і не допускається як валідний хеш.

На другому етапі, за наявності батьківського контейнера, відбувається змішування його хешу з хешем поточного елемента. Зазначений підхід було обрано замість альтернативи, у якій хеш батьківського контейнера виступає зерном для безпосереднього породження хешу дочірнього елемента. Початкове обчислення хешу без батьківського контексту дозволяє попередньо прораховувати значення там, де це можливо, а операція змішування двох уже відомих хешів є

істотно дешевшою, ніж повторне обчислення з урахуванням зерна у кожному кадрі.

Використана властивість покладена в основу етапу модифікації проміжної мови (IL) після компіляції коду C#. Оскільки цей крок є відносно витратним за часом, він виконується лише під час збирання оптимізованих конфігурацій. Замінюючи літеральні назви елементів на їхні попередньо обчислені хеші на етапі компіляції, зменшується час, необхідний на хешування в кожному кадрі, що безпосередньо покращує продуктивність IMGUI-бібліотеки.

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
Frequently called  3 usages
public static ulong HashString(string str, ulong seed)
{
    uint2 hash2;

    unsafe
    {
        fixed (char* p = str)
        {
            hash2 = xxHash3.Hash64(p, str.Length * sizeof(char), seed);
        }
    }

    return Uint2ToUlong(hash2);
}

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    Frequently called  2 usages  SolidAlloy *
    public static ID Persistent(string id)
    {
        ulong hash = HashString(id, 0);
        if (hash == 0) // force non-zero hash by rehashing.
        {
            hash = HashString(id, 0x9E3779B97F4A7C15UL);

            if (hash == 0) // astronomically unlikely
                hash = 1;
        }

        return new ID
        {
            Hash = hash,
#if DEBUG
            DebugString = id
#endif
        };
    }
```

Рис 3.11 Функції розрахунку хешу з назви елемента

```
if (id.Hash != 0)
{
    // 1. Take parent hash to mix
    ulong parentHash = _hashStack.Count > 0 ? _hashStack.Peek() : 0;

    // 2. Create a child hash using parent hash.
    id.Hash = MixHashes(parentHash, id.Hash);
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
Frequently called  1 usage
public static ulong MixHashes(ulong first, ulong second)
{
    const ulong goldenRatio = 0x9E3779B97F4A7C15UL;
    ulong x = first ^ (second + goldenRatio + (first << 6) + (first >> 2));
    return Avalanche64(x);
}
```

Рис 3.12 Ділянки коду, в яких відбувається змішування хешів елемента та його батьківського контейнера

```
ulong hash = IDHelper.Persistent(literal).Hash;

static void ReplaceWithSequencePoint(ILProcessor processor, MethodDebugInformation debugInfo,
                                     int i, Instruction old, Instruction newIns)
{
    var oldSequencePoint = debugInfo.GetSequencePoint(old);
    int sequencePointIndex = debugInfo.SequencePoints.IndexOf(oldSequencePoint);
    processor.Replace(i, newIns);

    if (oldSequencePoint != null)
    {
        debugInfo.SequencePoints[sequencePointIndex] = new SequencePoint(newIns, oldSequencePoint.Document)
        {
            StartLine = oldSequencePoint.StartLine,
            StartColumn = oldSequencePoint.StartColumn,
            EndLine = oldSequencePoint.EndLine,
            EndColumn = oldSequencePoint.EndColumn
        };
    }

    static void PatchScope(ScopeDebugInformation scope, Instruction old, Instruction newIns)
    {
        if (scope == null)
            return;

        if (!scope.Start.IsEndOfMethod && scope.Start.Offset == old.Offset)
            scope.Start = new InstructionOffset(newIns);

        if (!scope.End.IsEndOfMethod && scope.End.Offset == old.Offset)
            scope.End = new InstructionOffset(newIns);

        if (scope.HasScopes)
        {
            foreach (var childScope in scope.Scopes)
            {
                PatchScope(childScope, old, newIns);
            }
        }
    }

    PatchScope(debugInfo.Scope, old, newIns);
}

ReplaceWithSequencePoint(processor, debugInfo, argInstructionIndex, argInstruction,
                         processor.Create(OpCodes.Ldc_I8, unchecked((long)hash)));
ReplaceWithSequencePoint(processor, debugInfo, i, instruction,
                         processor.Create(OpCodes.Call, constantMethodRef));
```

Рис 3.13 Частина коду модифікувача IL, який підміняє назву елемента на хеш на етапі компіляції

3.3 Опис функціоналу, що забезпечує побудову сучасних інтерфейсів і виокремлює бібліотеку серед аналогів

3.3.1 Автоматичне компонування

Автоматичне компонування елементів інтерфейсу є ключовою підсистемою, що якісно виокремлює бібліотеку серед конкурентів, знижує бар'єр входу для розробників і гарантує коректний вигляд інтерфейсу на широкому спектрі дисплеїв. Метою розробки було сформувати алгоритм автоматичного компонування, який задовольняє такі вимоги:

– автоматичне визначення місць перенесення рядків у текстових блоках без фіксованої ширини (функціональність відсутня в Unity IMGUI та UGUI);

– підтримка відсоткових розмірів відносно батьківського контейнера (відсутня в Unity IMGUI та UI Toolkit);

– підтримка фіксованого співвідношення сторін елемента (відсутня в Unity IMGUI та UI Toolkit);

– підтримка проміжків між дочірніми елементами (відсутня у всіх трьох зазначених конкурентів);

– низькі часові витрати навіть у складних ієрархіях (компонування виконується щокадрово).

Запропонований алгоритм спирається на принципи Flexbox, але не відтворює стандарт повністю. Повна специфікація Flexbox є достатньо комплексною, що підвищує ризик великих часових витрат у щокадрових розрахунках; водночас її концепції добре відомі практикам. Основні принципи описаного алгоритму базуються на роботі, проведені Ніком Баркером [17]. До алгоритму інтегровано такі принципи:

– елементи декларують готовність до стиснення (shrink) або зростання (grow). Якщо сума бажаних розмірів перевищує доступний простір, скорочення відбувається насамперед для елементів, що дозволили shrink; за наявності вільного простору його розподіляють серед елементів, які дозволили grow;

– контейнери задають правила розташування безпосередніх дітей: напрямок (ряд або колонка), вирівнювання вздовж головної осі (на початку, у кінці тощо) і вздовж перпендикулярної осі (наприклад, згори, по центру чи знизу для рядка).

Алгоритм спеціально спроєктовано так, щоб коректно підтримувати перенесення слів у текстових блоках без фіксованої ширини. Розрахунок виконується у шість етапів.

а) Обчислення властивих розмірів. Виконується обхід дерева знизу догори (від листя до кореня). Кожен елемент узгоджує власні мінімальні обмеження з сукупним бажаним внеском дочірніх елементів, формуючи мінімальний (нестиснювальний) та бажаний розміри.

б) Розрахунок по горизонтальній осі. Виконується обхід згори донизу. Для кожного контейнера виокремлюється підмножина дітей із нефіксованими розмірами; визначається доступний простір. Якщо він перевищує суму бажаних розмірів, надлишок розподіляється за правилами grow; якщо є дефіцит — скорочення застосовується до елементів, що дозволили shrink. На виході формується кінцевий горизонтальний розмір елементів.

в) Перенесення рядків у текстових блоках і встановлення їхньої бажаної висоти. Після фіксації ширини всі текстові елементи отримують місця перенесення. Для ефективності їх збирають у списку під час етапу 2, тож на етапі 3 виконується лінійна обробка цього списку. Кожному текстовому елементу призначається бажана висота, пропорційна кількості сформованих рядків.

г) Оновлення властивих висот контейнерів. Корекція бажаної висоти текстових блоків впливає на батьків, тож виконується новий прохід знизу догори для актуалізації їхніх властивих висот.

г) Розрахунок по вертикальній осі. Аналог етапу 2 для перпендикулярної осі: визначаються фінальні вертикальні розміри з урахуванням shrink/grow і оновлених властивих висот.

д) Визначення фінальних координат. Проводиться обхід згори донизу з обчисленням позицій елементів у межах контейнерів, із урахуванням порядку

розміщення, розмірів попередніх елементів та оголошених проміжків. Для зменшення вартості повторних обходів на цьому етапі паралельно формуються та групуються дані для рендерингу.

Відсоткові розміри. Елемент може декларувати відсотковий розмір як альтернативу стисненню/зростанню чи фіксованому розміру. У такому разі shrink/grow для нього не застосовують, а обчислення відсоткової частки відбувається у два моменти:

– Етап 1. Внесок елемента з відсотковим розміром враховується у бажаному розмірі батьківського контейнера. Ілюстративний приклад: якщо бажана ширина елемента 10 px має складати 50% ширини контейнера, а інші діти сумарно потребують 5 px, бажана ширина контейнера має стати 20 px, щоб частка елемента дорівнювала 50%.

– Етапи 2 і 5. Після фіксації фінального розміру контейнера відповідні діти одержують свої фінальні розміри як відсоток від розміру батька вздовж відповідної осі.

Фіксоване співвідношення сторін. Елемент може додатково декларувати співвідношення сторін незалежно від обраної політики розмірів (фіксована, відсоткова, shrink/grow). Зміни застосовуються до меншої осі:

– Етап 1. Після визначення бажаних властивих розмірів менша з осей коригується для виконання заданої пропорції.

– Етапи 2 і 5. Якщо елемент водночас має відсоткову політику та вимогу співвідношення сторін, після встановлення відсоткового розміру коригується перпендикулярна вісь.

– Етап 3.5. Після визначення перенесень у тексті, але до другого підйому властивих висот, перераховуються висоти елементів із фіксованою пропорцією, оскільки на цьому кроці ширини вже фіксовані, а змінюваною величиною є висота. Для уникнення зайвих обходів такі елементи, як і текстові, акумулюються в окремому масиві під час етапу 2.

Проміжок між дочірніми елементами. Проміжок задається числовим аргументом під час створення елемента та впливає майже на всі етапи, окрім

етапу 3. У розрахунку бажаної довжини контейнера вздовж осі компонування додається внесок проміжків між послідовними дітьми. Формально сукупна ширина/висота вздовж осі компонування:

$$S_p = \sum_{i=1}^{N} S_i + \sum_{i=1}^{N-1} g, \tag{3.1}$$

де Sp – розмір батьківського елемента вздовж осі компонування, N – кількість дочірніх елементів, Si – розмір і-го дочірнього елемента вздовж цієї осі, g – величина проміжку між сусідніми елементами.

Зазначений підхід уніфікує облік проміжків з урахуванням кількості інтервалів між N елементами (N−1) і застосовується симетрично для рядкового та колонкового розміщення. Завдяки чіткій поетапній організації розрахунків і цільовому використанню списків для «гарячих» підмножин (текстові та пропорційні елементи) алгоритм зберігає низьку часову вартість у щокадровому виконанні для складних ієрархій.

3.3.2 Візуальний функціонал

Закруглені кути. Закруглені кути реалізуються в шейдері на основі функції відстані зі знаком (Signed Distance Function, SDF) для закругленого прямокутника [16].

$$d = |p| - (h - r), \tag{3.2}$$

$$SDF(p; h, r) = \sqrt{max(d_x, 0)^2 + max(d_y, 0)^2} + min(max(d_x, d_y), 0) - r, \tag{3.3}$$

де SDF - відстань зі знаком до межі фігури; p - локальна позиція точки; h - вектор напіврозмірів (h_x, h_y); r - радіус заокруглення; d - допоміжний вектор; d_x, d_y - його компоненти; max, min - покомпонентні операції максимуму/мінімуму; |p| - покомпонентний модуль.

Для створення плавної межі використовується згладжування на основі функції smoothstep [18].

$$t = clamp(\frac{d}{s}, 0, 1), \qquad mask = 1 - (t^2(3 - 2t)), \tag{3.4}$$

де t - нормалізований параметр; d — дистанція; s — константа м’якості. Отримане значення називається маскою та використовується як мультиплікатор для кольору, роблячи деякі пікселі напівпрозорими.

Контури. Контури також використовують концепт SDF для побудування регіону, який забарвлюється окремим кольором. Зовнішня границя контуру — вже розрахована дистанція для самого елемента. Дистанція для внутрішньої границі знаходиться шляхом додавання товщини контуру.

$$Di = Do + t, \tag{3.5}$$

де Di – дистанція до внутрішньої границі контуру, Do – дистанція до зовнішньої границі, t – товщина контуру. Маска кордону формується як добуток «зовнішньої» та «внутрішньої» масок:

$$Mb = Mi \cdot Mo, \tag{3.6}$$

де Mo та Mi отримуються через згладжені пороги над Do та Di відповідно. Важливо, що кольори фону та границі елемента мають накладатись один на один та бути змішані всередині шейдера. Для цього шейдер має використовувати режим заздалегідь помноженої альфи (premultiplied alpha) [19]. Використовуючи цей режим, задача змішування кольорів перекладається з апаратної імплементації графічного процесора на програмну всередині шейдера. В такому випадку кольори фону та границь змішуються за наступними формулами:

$$F = S \circ C; \qquad f_a = s_a c_a; \qquad f_{rgb} = s_{rgb} \circ c_{rgb}, \tag{3.7}$$

де S – вибірка текстури (sample), C — колір фону, ⊙ - поелементне множення, F – добуток по компонентах.

$$F^* = (f_{rgb}(f_a m_f), f_a m_f); \qquad B^* = (b_{rgb}(b_a m_b), b_a m_b), \tag{3.8}$$

де mf – маска зовнішньої лінії, mb – маска внутрішньої лінії (границі), F* — заливка у заздалегідь помноженій формі, B* — контур у заздалегідь помноженій формі.

$$O^*_{rgb} = B^*_{rgb} + F^*_{rgb}(1 - B^*_a); \qquad O^*_a = B^*_a + F^*_a(1 - B^*_a), \tag{3.9}$$

де O* - вихідний колір.

```
float cornerRadius = i.cornerRadius_edgeSoftness_borderThickness.x;
float2 vectorFromCircleCenter = abs(i.uv_localPos.zw) - (i.halfSize_rectCenter.xy - cornerRadius);
float outerDistance = min(max(vectorFromCircleCenter.x, vectorFromCircleCenter.y), 0.0)
                    + length(max(vectorFromCircleCenter, 0.0)) - cornerRadius;

float softness = i.cornerRadius_edgeSoftness_borderThickness.y;
float outerMask = min(1 - smoothstep(0, softness, outerDistance), maskAlpha);
float borderThickness = i.cornerRadius_edgeSoftness_borderThickness.z;
float innerBorderDistance = outerDistance + borderThickness;
float innerBorderMask = smoothstep(0, softness, innerBorderDistance);
float borderMask = innerBorderMask * outerMask;

float4 fillColor = sample * i.color;
float4 fillPreMultiplied = float4(fillColor.rgb * (fillColor.a * outerMask), fillColor.a * outerMask);
float4 borderPreMultiplied = float4(i.borderColor.rgb * (i.borderColor.a * borderMask), i.borderColor.a * borderMask);

float4 outPreMultiplied;
outPreMultiplied.rgb = borderPreMultiplied.rgb + fillPreMultiplied.rgb * (1.0 - borderPreMultiplied.a);
outPreMultiplied.a   = borderPreMultiplied.a   + fillPreMultiplied.a   * (1.0 - borderPreMultiplied.a);
return outPreMultiplied;
```

Рис 3.14 Фрагмент шейдера, відповідальний за закруглені кути, формування контуру та змішування двох шарів кольору в заздалегідь помноженій формі

Градієнти. Для відтворення градієнтів використовується апаратна інтерполяція кольорів графічним процесором на основі атрибутів вершин трикутників. У кожній вершині задаються чотири компоненти кольору, упаковані у 4 байти, які на етапі растеризації конвертуються у формат з плаваючою крапкою та автоматично інтерполюються.

Такий підхід дає змогу ефективно реалізовувати горизонтальні, вертикальні та діагональні градієнти без додаткових витрат у шейдері. Розширення до багатоточкових або радіальних градієнтів може бути предметом подальших робіт, оскільки потребує окремих параметризацій та, ймовірно, інших інтерполяційних стратегій.

```
float4 UnpackRGBA(uint packedColor)
{
    return float4((packedColor      ) & 255,
                  (packedColor >> 8 ) & 255,
                  (packedColor >> 16) & 255,
                  (packedColor >> 24) & 255)
           * (1.0/255.0);
}

uint packedColor = quad.colors[vertToColorIndex[corner]];
o.color = UnpackRGBA(packedColor);
```

Рис 3.15 Конвертація запакованого кольору вершини та передача у регістр COLOR для апаратної інтерполяції

3.3.3 Підтримка стилів, тем, та анімацій

Зовнішній програмний інтерфейс бібліотеки спроєктовано так, щоби не накладати обмежувальних припущень щодо внутрішнього подання стилів і тем. Це забезпечує розробникові свободу вибору структур даних та форматів зберігання конфігурацій відповідно до вимог конкретного застосунку та існуючих інженерних практик. Нижче наведено репрезентативні підходи до організації стилів.

а) Серіалізований клас. Наслідування від ScriptableObject в середовищі Unity дає змогу автоматично серіалізувати поля класу й відображати їх у зручній для редагування формі інспектора. Схема теми оголошується один раз у коді, після чого для кожної теми створюються окремі серіалізовані файли з конкретними значеннями параметрів. Така стратегія полегшує розділення відповідальностей між розробниками та дизайнерами інтерфейсів, а також спрощує керування конфігураціями на різних середовищах (розробка, тестування, випуск).

```
[CreateAssetMenu(menuName = "SolidImgui/Examples/Theme", fileName = "Examples Theme")]
2 asset usages  12 usages  SolidAlloy *  1 exposing API
public class ExampleTheme : ScriptableObject, ISerializationCallbackReceiver
{
    [Serializable]
    1 usage  SolidAlloy  1 exposing API
    public struct PanelStyle
    {
        public Color32 BackgroundColor;  Serializable
        public Color32 TextColor;  Serializable
        public Color32 AvatarBorderColor;  Serializable
        public Color32 WarningTextColor;  Serializable
    }
    public PanelStyle Panel;  Serializable

    [Serializable]
    7 usages  SolidAlloy  4 exposing APIs
    public struct ButtonStyle
    {
        public float FontSize;  Serializable
        public ushort Padding;  Serializable
        public float CornerRadius;  Serializable
        public float BorderThickness;  Serializable

        [Serializable]
        5 usages  SolidAlloy  3 exposing APIs
        public struct AnimProps
        {
            public CornerColors Background;  Serializable
            public Color32 Border;  Serializable
            public Color32 Text;  Serializable
        }

        public AnimProps Normal;  Serializable
        public AnimProps Hovered;  Serializable
        public AnimProps Pressed;  Serializable
    }
    public ButtonStyle BorderButton;  Serializable
    public ButtonStyle FilledButton;  Serializable
    public ButtonStyle SmallerFilledButton;  Serializable
    public ButtonStyle TextButton;  Serializable
```

Рис 3.16 Фрагмент серіалізованого класу для зберігання стилів

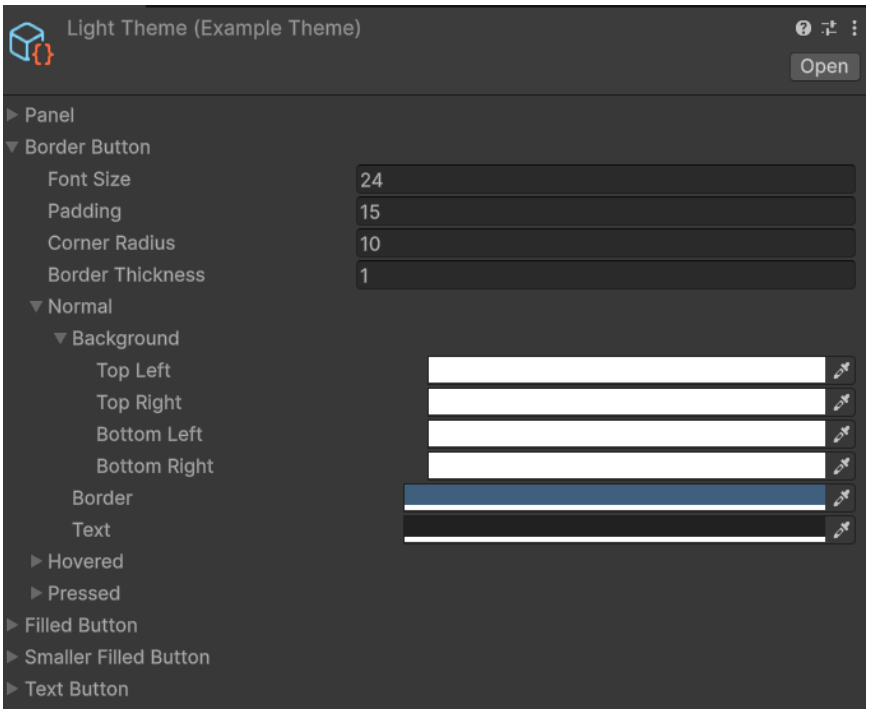


Рис 3.17 Серіалізований файл з відображенням полів у редакторі

б) Структура теми з підтримкою «живої» рекомпіляції. Альтернативно, опис стилів може бути сконструйовано програмно (без серіалізації), що полегшує швидку ітерацію в процесі розробки. Окремі стилі допускають «гарячу» заміну шляхом часткової рекомпіляції та оновлення байткоду без перезапуску застосунку. Це знижує час зворотного зв’язку, полегшує експерименти з варіантами оформлення та мінімізує контекстні перемикання між IDE і середовищем виконання.

```
4 usages  SolidAlloy  3 exposing APIs
public struct ExampleTheme2
{
    // Declare colors, floats, other variables in here
    1 usage  SolidAlloy  1 exposing API
    public struct VariablesStruct
    {
        public Color32 BackgroundColor;
    }
    public VariablesStruct Variables;

    // Declare a struct per widget
    1 usage  SolidAlloy  1 exposing API
    public struct ButtonStyle
    {
        public Color32 BackgroundColor;
        public float CornerRadius;
    }
    public ButtonStyle Button;
}

SolidAlloy
public static class AlternativeStyleExample
{
    // Use InvokeOnHotReloadLocal from HotReload or OnScriptHotReloadNoInstance from FastScriptReload
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterAssembliesLoaded)/*, InvokeOnHotReloadLocal*/]
f UNITY_EDITOR
    [InitializeOnLoadMethod]
ndif
    SolidAlloy
    private static void ConstructThemes()
    {
        // Put variables and unique style fields here.
        // In C# 10, it should become much easier because the fields can be initialized inline in the struct declaration.
        Themes.LightTheme = new()
        {
            Variables =
            {
                BackgroundColor = COLOR_GRAY,
            },
            Button =
            {
                CornerRadius = 10,
            },
        };

        Span<ExampleTheme2> themes = MemoryMarshal.CreateSpan(ref Themes.LightTheme, 2);

        // Put the variables into the fields that use them here.
        for (int i = 0; i < themes.Length; ++i)
        {
            ref var theme = ref themes[i];
            ref var variables = ref theme.Variables;

            ref var button = ref theme.Button;
            button.BackgroundColor = variables.BackgroundColor;
        }
    }
```

Рис 3.18 Приклад використання «живої» рекомпіляції для заміни полів структур стилів

Параметри, що впливають на візуальне оформлення, у всіх публічних функціях бібліотеки є опціональними. Отже, використання додаткових характеристик зовнішнього вигляду не нав’язує розробникові надлишкових витрат коду: для кожного елемента застосовуються лише ті параметри стилю, які явно задані викликом API. Така політика знижує в’язкість коду та сприяє локалізації змін.

```
using Scope buttonScope = BoxScope(id, width, height, cornerRadius: style.CornerRadius,
    borderThickness: style.BorderThickness, padding: Add(style.Padding, (ushort)style.BorderThickness),
    flags: Ability_Clickable | Ability_Hoverable | MainChildAlignment_Center | CrossChildAlignment_Center);
```

Рис 3.19 Використання структури стилю кнопки у виклику функції

Анімації. Підтримку анімацій організовано у двох взаємодоповнювальних категоріях.

а) Стандартні анімації взаємодії: наведення курсором, натискання, вимкнений стан. Прапорці станів взаємодії відстежуються бібліотекою автоматично та зберігаються в її внутрішній області даних. Водночас бібліотека не нав’язує перелік анімованих властивостей: рішення про те, що саме інтерполювати, приймає прикладний код. Для керування переходами між типізованими станами надається функція GetAnimationValues, яка за масивом конфігурацій-кандидатів повертає пару «попередній стан → поточний стан». Після цього прикладний код застосовує інтерполяцію (наприклад, лінійну `lerp`) за нормалізованим параметром прогресу анімації з переходом між відповідними значеннями стилів. Така декомпозиція дозволяє чітко відокремити механіку виявлення станів (відповідальність бібліотеки) від політики відображення (відповідальність розробника), що полегшує адаптацію до різних дизайн-систем.

```
(var prev, var curr) = GetAnimationValues<ExampleTheme.ButtonStyle.AnimProps>(
    stackalloc AnimField<ExampleTheme.ButtonStyle.AnimProps>[]
{
    new(State.None, style.Normal),
    new(State.Hovered, style.Hovered),
    new(State.Hovered | State.Pressed, style.Pressed),
    new(State.Pressed, style.Normal),
}, button.Flags);

button.Tint = CornerColors.Lerp(prev.Background, curr.Background, button.AnimT);
button.Image().BorderColor = Color32.Lerp(prev.Border, curr.Border, button.AnimT);
textBox.Tint = Color32.Lerp(prev.Text, curr.Text, button.AnimT);
```

Рис 3.20 Використання стандартної анімації взаємодії

б) Індивідуальні анімації. До цієї категорії належать сценарії, що не вкладаються у стандартний набір, зокрема анімації контейнерів чи панелей. Показовий приклад — розгортання вертикального «випадаючого» регіону, тривалість і крива інтерполяції якого не збігаються зі стандартними анімаціями

кнопки-перемикача. Для індивідуальних анімацій у структурі UIBox передбачено поле CustomAnimT. На відміну від AnimT, яке керується бібліотекою та слугує суто для читання на прикладному рівні, значення CustomAnimT повністю контролюється та модифікується розробником. Такий підхід забезпечує гнучкість у визначенні часових профілів і синхронізації з подіями доменної логіки.

Водночас виникає ініціалізаційна колізія: бібліотека створює елемент зі значенням CustomAnimT = 0 за замовчуванням, тоді як вихідний візуальний стан інтерфейсу може передбачати, що певний регіон вже «розгорнутий» на момент появи (тобто має відповідати CustomAnimT = 1). Якщо розробник має лише інкрементальне керування прогресом, це породжує артефакти — наприклад, плавне розгортання меню при кожному відкритті вікна, незалежно від бажаного початкового стану. Аналогічну проблему можна спостерігати в UI Toolkit. Для уникнення такого ефекту бібліотека надає механізм виявлення щойно створених анімованих елементів, який дозволяє одноразово встановити CustomAnimT миттєво у відповідне граничне значення (0 або 1) без проміжної інтерполяції, синхронізуючи тим самим графічний стан із початковими даними панелі.

```
if (button.IsFirstFrame)
    button.CustomAnimT = value ? 1f : 0f;

if (button.Clicked)
    value = !value;

button.CustomAnimT += animationRate * ((value ? 1f : 0f) - button.CustomAnimT);
return buttonScope;
```

Рис 3.21 Зміна CustomAnimT у коді

```
if (leftPanelOpenT > 0f)
{
    using (VerticalFoldoutScope(300, leftPanelOpenT))
    {
        using (BoxScope(Trn("profile section"), Grow(), Fit(),
                        LayoutDirection_Vertical | CrossChildAlignment_Center, childGap: 10))
        {
            DoBox(Trn("avatar icon"), Fixed(200), Fixed(200), borderColor: panelStyle.AvatarBorderColor,
                  borderThickness: 2, cornerRadius: 100, tint: COLOR_WHITE, sprite: _avatar);

            DoText(Trn("avatar title"), "UI Showcase", false,
                   tint: panelStyle.AvatarBorderColor, font: Settings.BoldFont, fontSize: 28);
        }
```

Рис 3.22 Використання CustomAnimT для показу та анімації випадаючого регіону (тут перейменований на leftPanelOpenT)

3.4 A/B аналіз проти UI Toolkit

Були розроблені дві функціонально зіставні панелі: одна — у DiVy, інша — в UI Toolkit. Візуальні характеристики та поведінкові сценарії узгоджено настільки, наскільки це дозволяли відмінності в моделях компонування, віджетах та доступних API обох рішень.

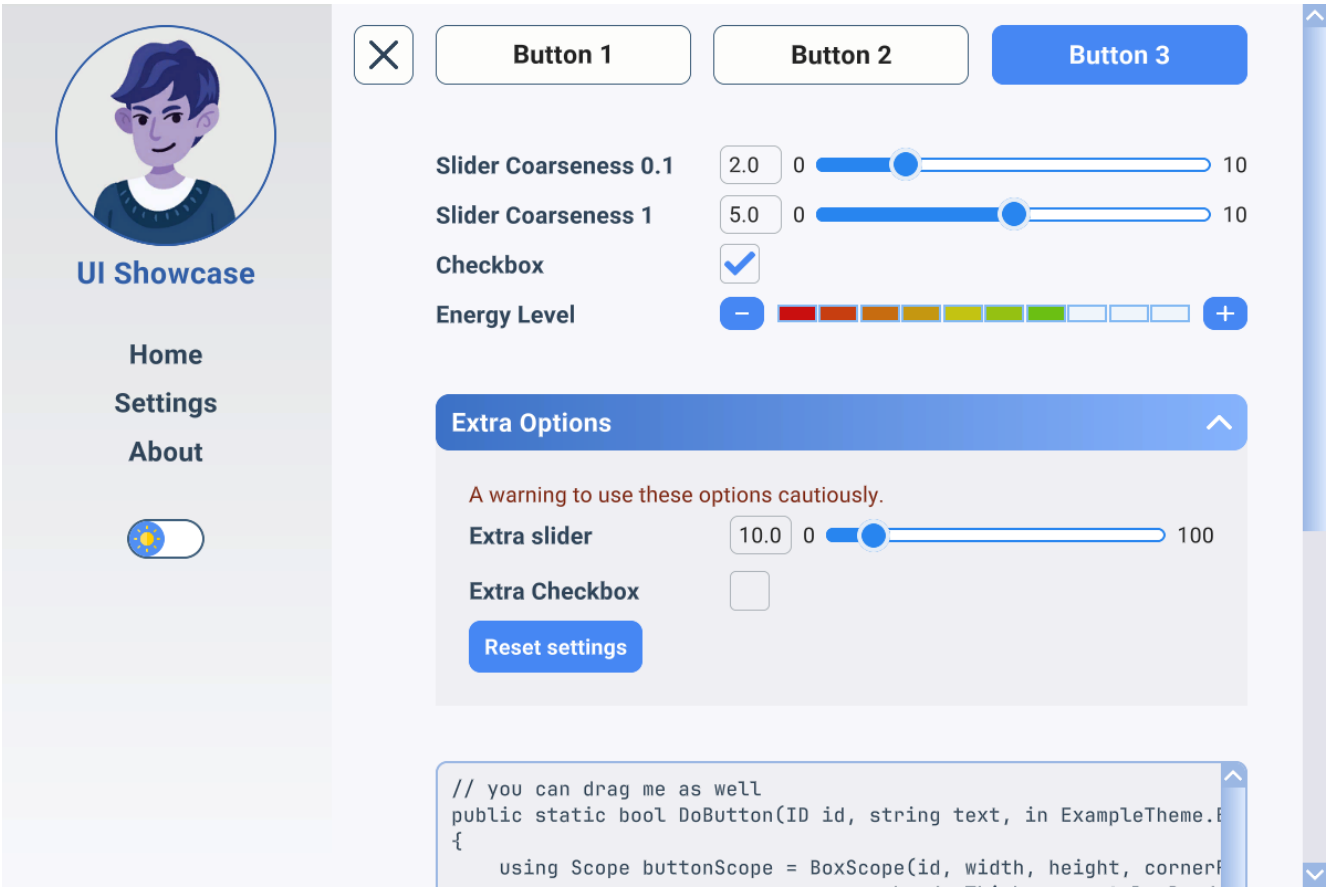


Рис 3.23 Панель, імплементована в DiVy

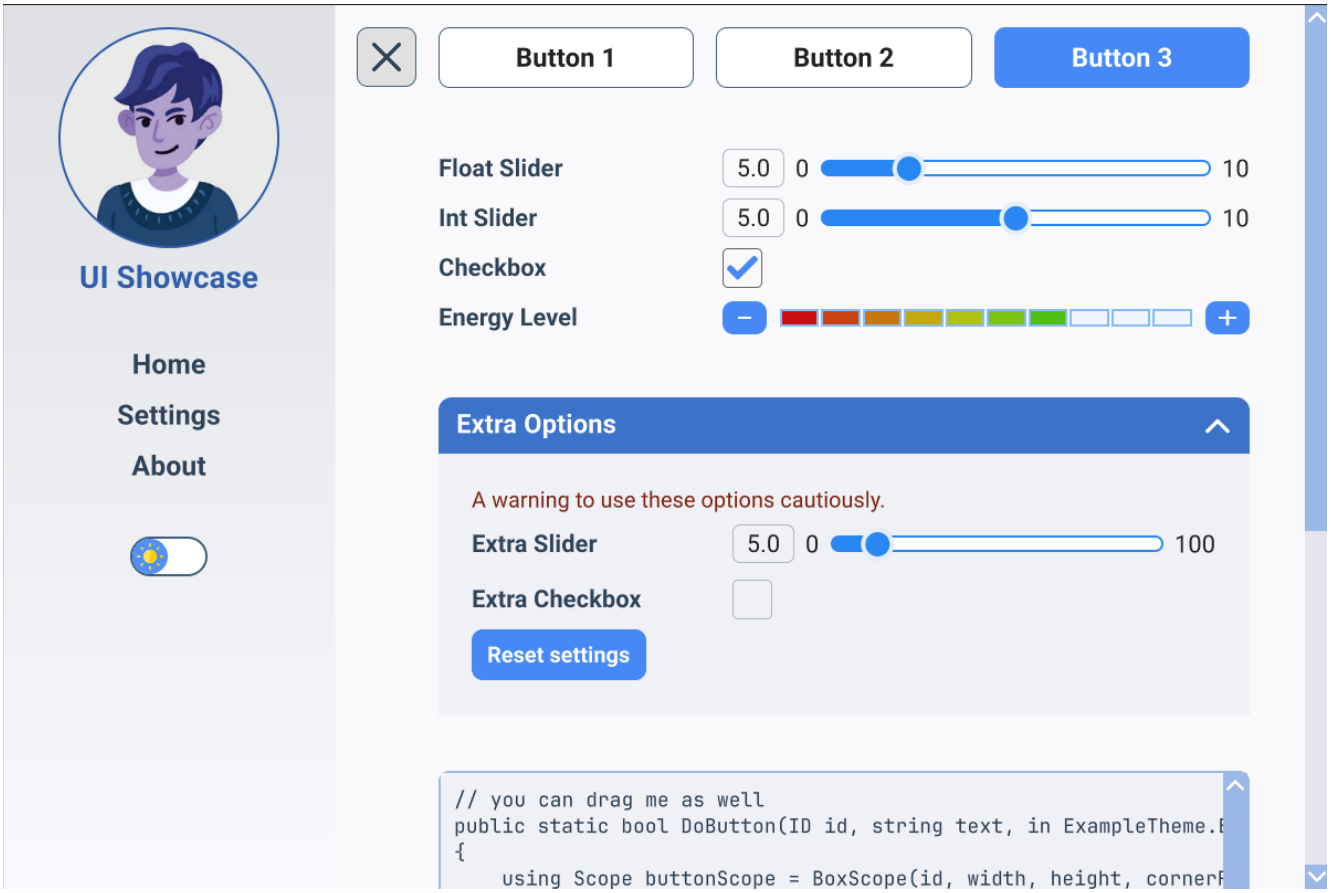


Рис 3.24 Панель, побудована за допомогою UI Toolkit

Незважаючи на прагнення до максимальної еквівалентності, між реалізаціями залишилися відмінності, зумовлені неповнотою підтримки низки концептів у UI Toolkit:

– віджет повзунка не підтримує округлення значення до заданого кроку (наприклад, 0.1);

– окремі елементи (зокрема чекбокс) мають стани наведення та фокусу, але не стан натискання;

– градієнти не застосовуються до інтерактивних елементів (наприклад, кнопок);

– заокруглення кутів для контейнера з прокручуванням працює некоректно (контури на кутах обрізаються);

– під час кожного відкриття горизонтальної «випадаючої» секції повзунок анімується із зміною позиції, незалежно від попереднього стану;

– контейнер із горизонтальною прокруткою не підтримує керування комбінацією Shift + коліщатко миші.

3.4.1 Порівняння обсягу коду

Виконано підрахунок обсягу коду для двох спеціалізованих віджетів (SegmentSlider та SlideToggle), а також для прикладного коду цільової панелі, без урахування реалізацій самих віджетів. Додатково для контексту порівняно сумарний обсяг коду стандартної бібліотеки віджетів, задіяної під час побудови панелі (кнопки, повзунки, контейнери з прокручуванням, «випадаючі» списки тощо).

Такий дизайн порівняння обрано замість попарного зіставлення окремих стандартних віджетів з двох причин. По-перше, в UI Toolkit широко використовується наслідування від базових типів, тож усунення спільних класів із підрахунку спотворило б реальну вартість повторного використання; по-друге, включення цих базових компонентів до підрахунку для кожного віджета призвело б до кратного дублювання внеску. Відтак для стандартної бібліотеки оцінюється інтегральний обсяг. Для спеціалізованих віджетів і прикладної панелі порівняння є прямим, оскільки функціонал контролюється ідентичними вимогами.

Таблиця 3.1

Порівняння кількості коду у різних категоріях та віджетах

Порівняння	Кількість рядків коду			Кількість літер коду		
	UI Toolkit	DiVy	Співвідношення	UI Toolkit	DiVy	Співвідношення
SegmentSlider	212	66	3.21	5383	2285	2.36
SlideToggle	136	46	2.96	3840	1519	2.53
Панель	952	353	2.70	31935	9993	3.20
Стандартна бібліотека віджетів	3921	489	8.02	143969	19516	7.38

За підсумками вимірювань встановлено, що для побудови спеціалізованих віджетів і кінцевих панелей на основі UI Toolkit потрібно у середньому в 2.5–3 рази більше коду, ніж для еквівалентної реалізації у DiVy. Це співвідношення узгоджується з опублікованими оцінками, наведеними Меттке (≈3×) [2]. Відмінність між коефіцієнтами для рядків і для кількості символів пояснюється відмінностями мов і стилів опису інтерфейсів: у UI Toolkit значна частина оформлення подається в CSS, де рядки зазвичай коротші; водночас і в DiVy стилі задаються компактними структурними полями. Для спеціалізованих віджетів у UI Toolkit стилізаційні описи виявилися об’ємнішими за власне програмну логіку, що підвищило співвідношення саме в підрахунку рядків. Натомість для панелей протилежний ефект зумовлений тим, що ієрархії UI Toolkit декларуються у форматі UXML із відносно довгими рядками.

Як було зазначено, інтегральне зіставлення стандартних бібліотек наведено для наочності та не використовується для інтерпретаційних висновків. Значний розрив (≈8× за рядками) очікуваний, оскільки UI Toolkit є зрілою екосистемою з ширшим охопленням функціоналу. Коректним об’єктом порівняння для висновків залишаються спеціалізовані віджети та прикладні панелі, де функціонал жорстко вирівняно.

3.4.2 Порівняння стабільності та продуктивності за кадр

Було здійснено вимірювання низки покадрових метрик у різних сценаріях взаємодії з інтерфейсом. Базовою точкою нормування слугувала «порожня» збірка рушія (без цільових панелей), що дозволяє відокремити внесок UI-підсистем від фіксованих витрат ядра.

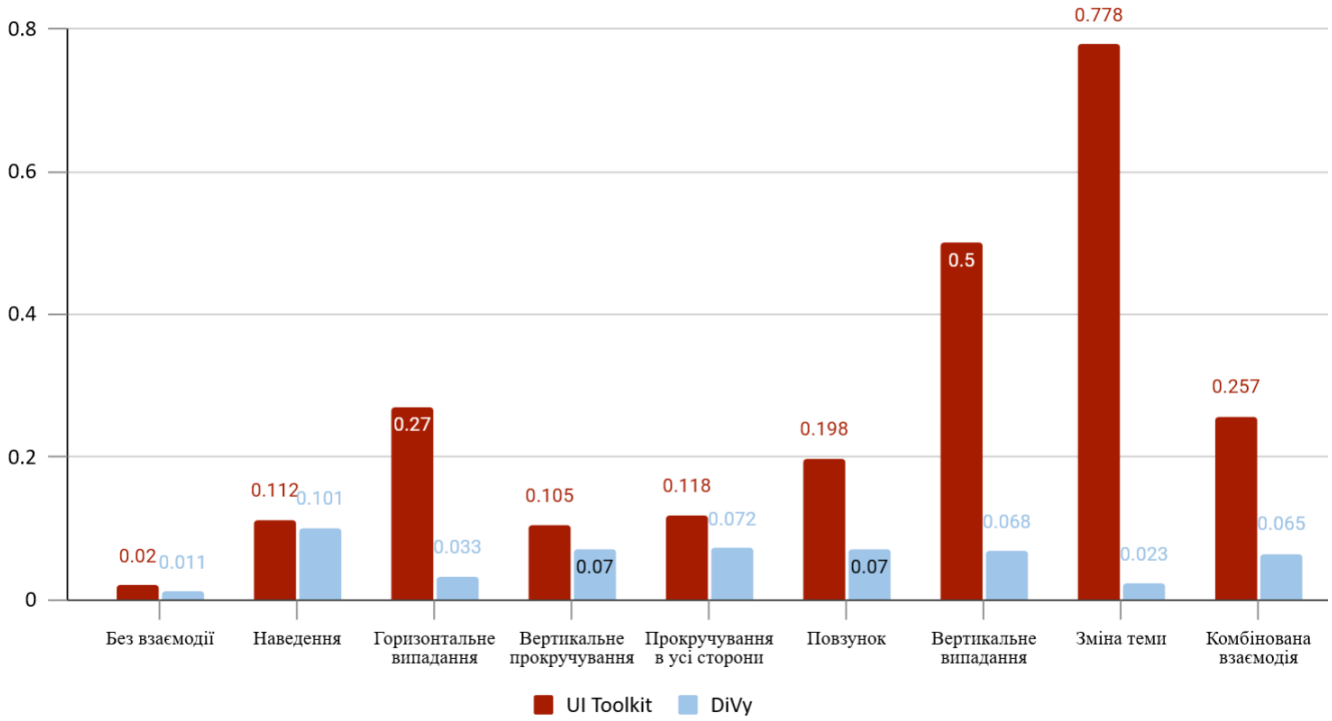


Рис 3.25 Коефіцієнт варіації кадрів у різних сценаріях, менше — краще

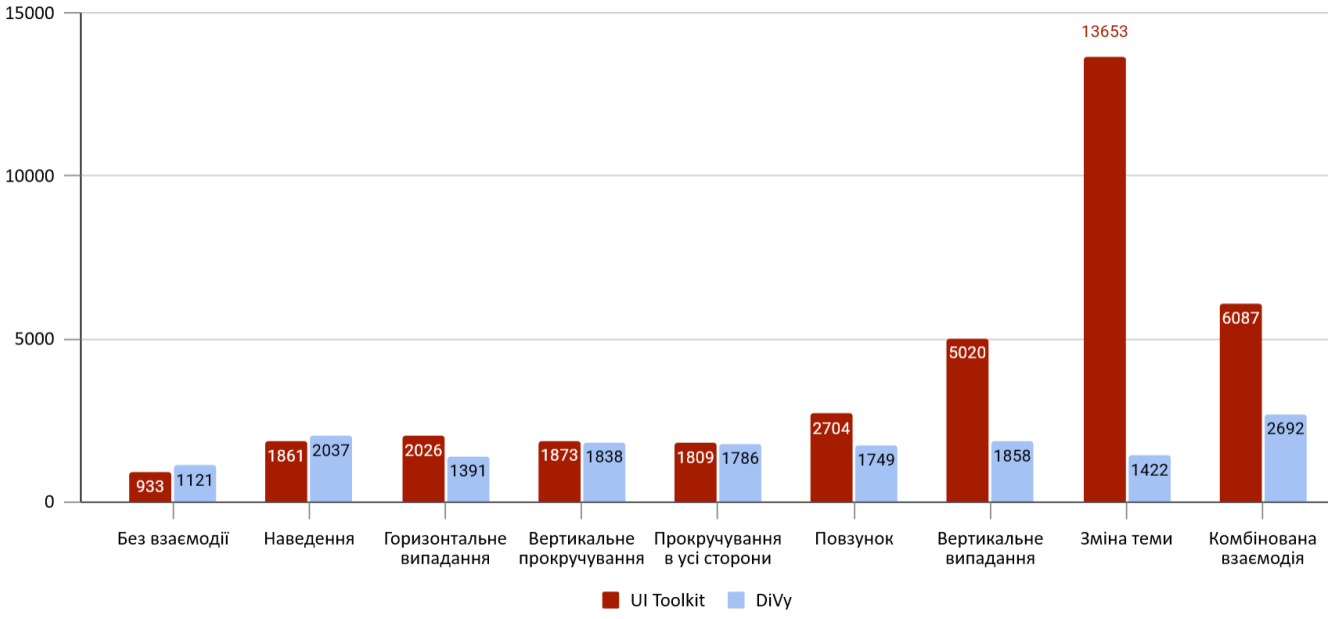


Рис 3.26 CVaR 1% кількості циклів, тис, менше — краще

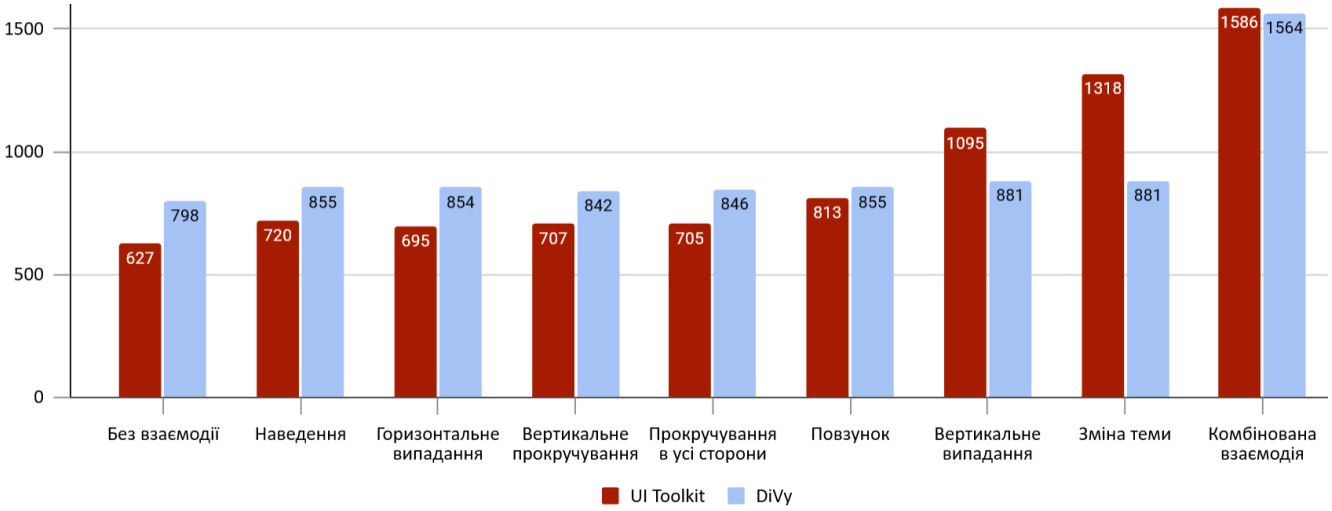


Рис 3.27 Середня кількість циклів процесора, тис, менше — краще

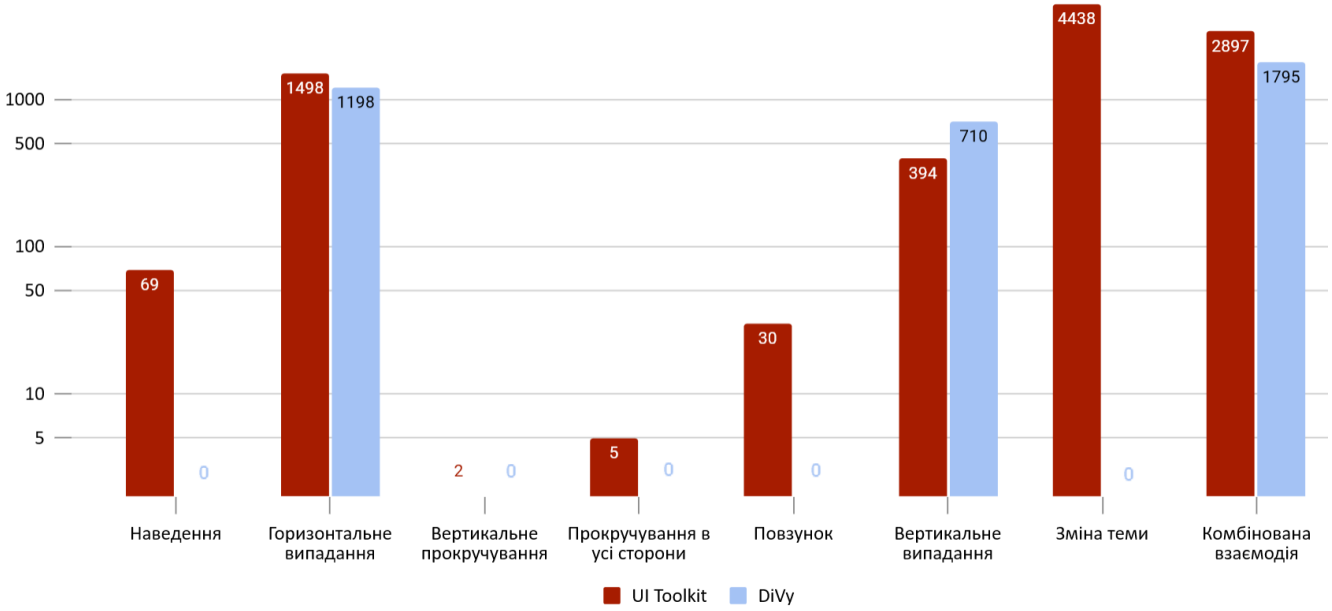


Рис 3.28 Сумарна виділена пам’ять, КБ, менше — краще

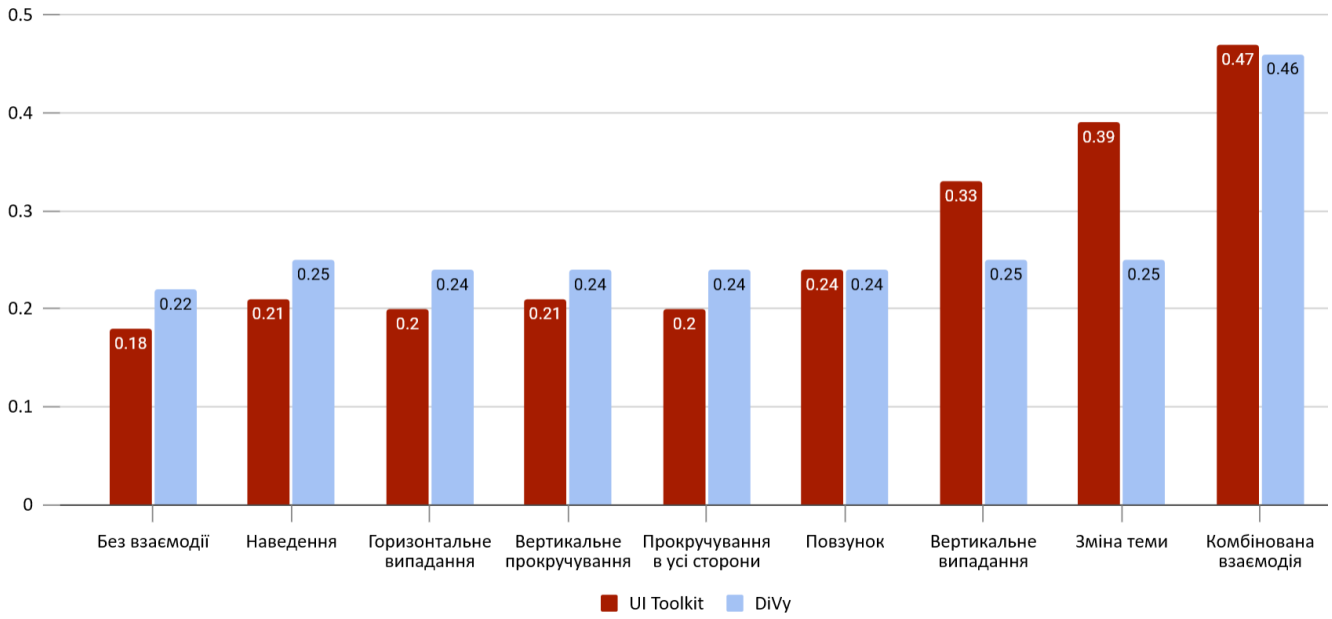


Рис 3.29 Середній час за кадр, мс, менше — краще

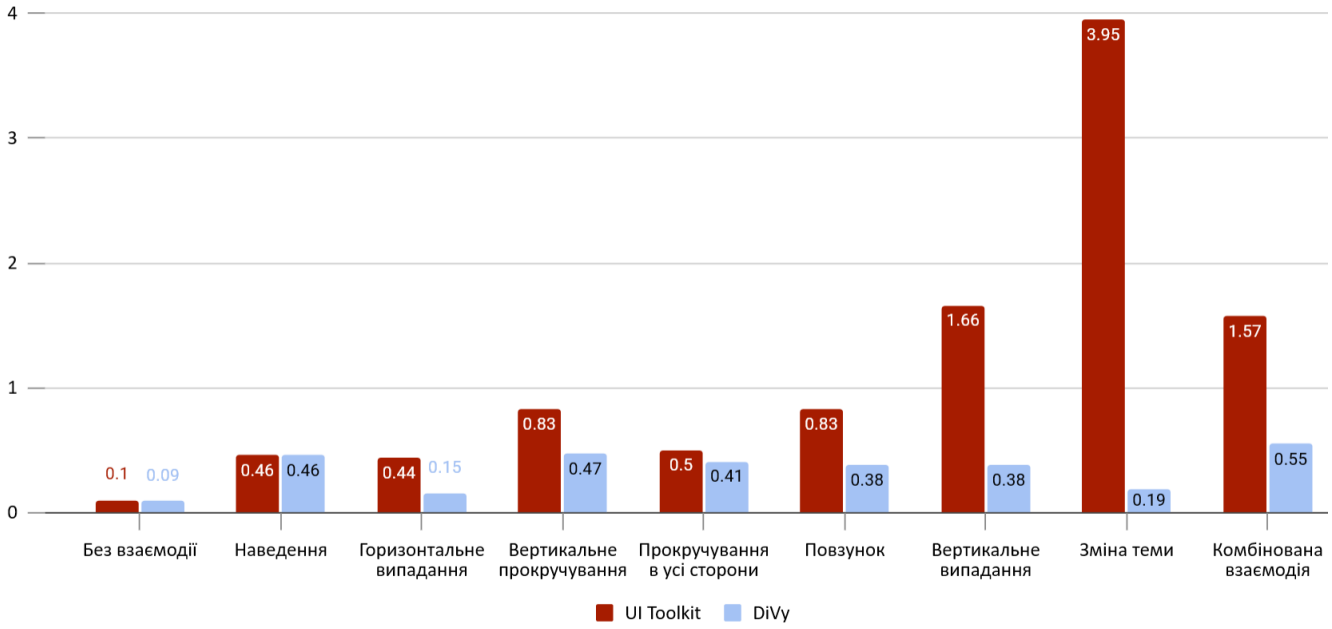


Рис 3.30 CVaR 1% часу за кадр, мс, менше — краще

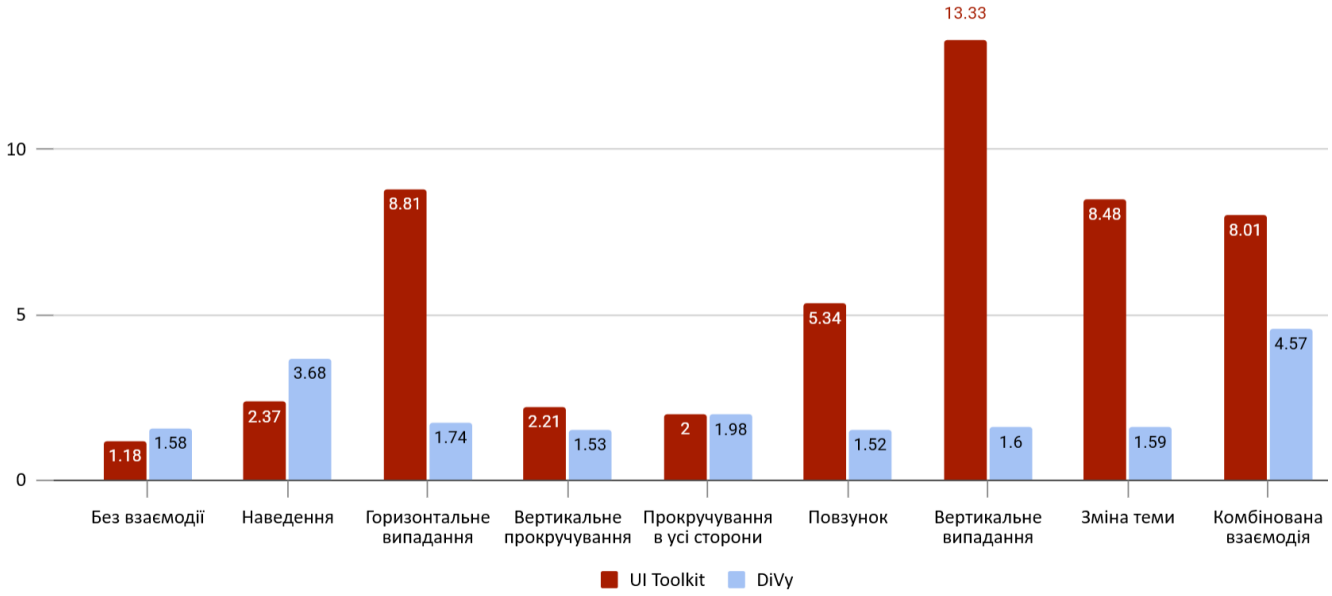


Рис 3.31 Максимальний час за кадр, мс, менше — краще

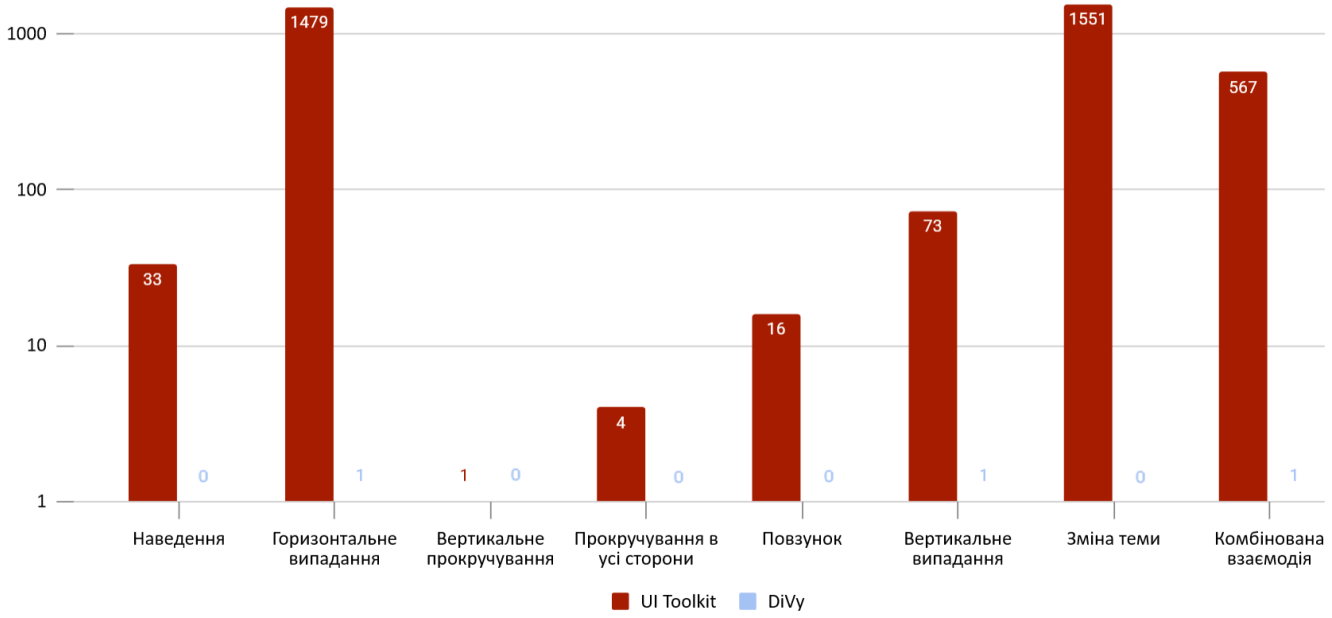


Рис 3.32 Максимальне виділення пам’яті за кадр, КБ, менше — краще

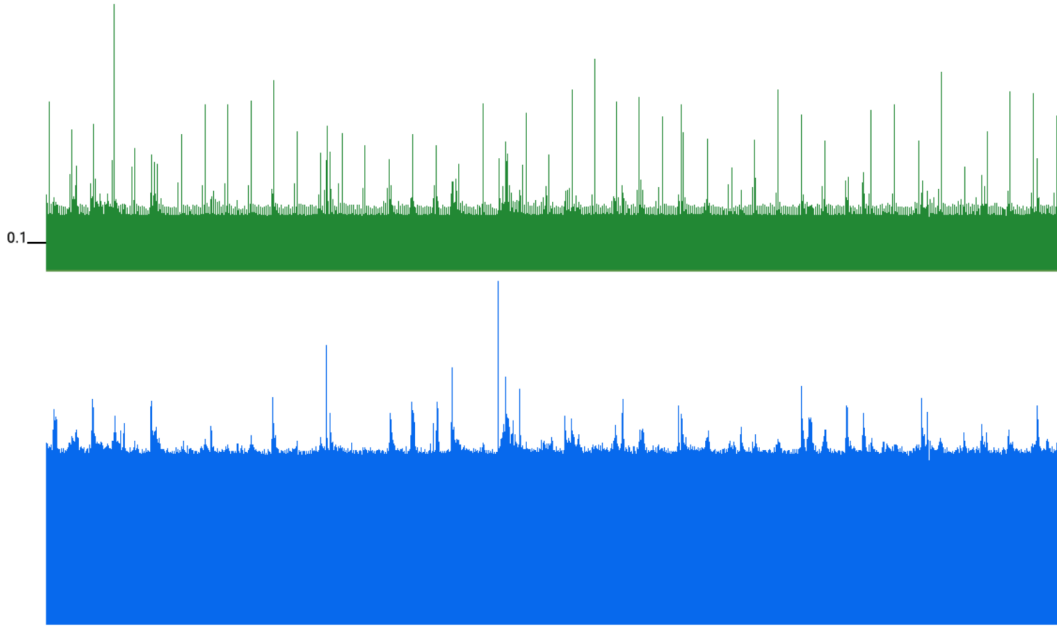


Рис 3.33 Демонстрація варіації між кадрами у пустій збірці; зеленим кольором позначений час, синім — кількість циклів процесора

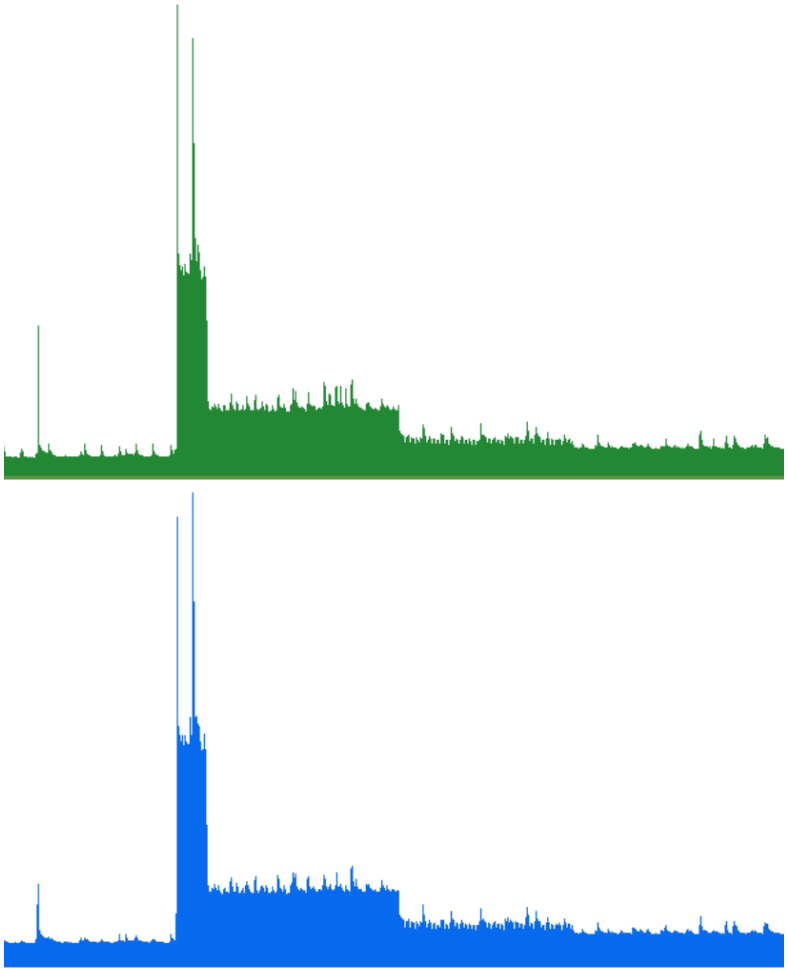


Рис 3.34 Демонстрація затримки кадрів при зміні теми в UI Toolkit, найвищий пік затримки — 8.48 мс

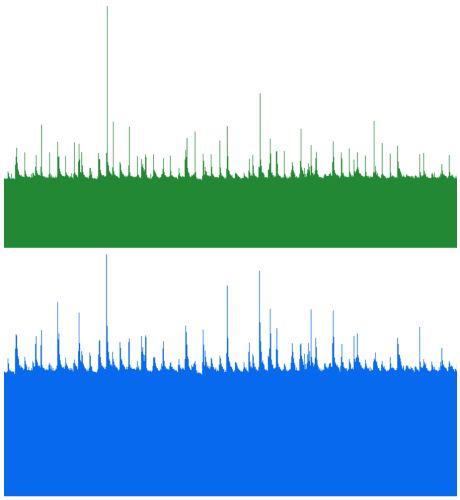


Рис 3.35 Демонстрація затримки кадрів при зміні теми в DiVy. Висота зображення нормалізована відносно найвищого піку затримки — 1.59 мс. Розмір збільшений у ≈5.3 разів порівняно з Рис 3.34

Перевірка гіпотез:

H1. «DiVy потребує меншого обсягу коду для спеціалізованих віджетів та побудови панелей». Гіпотеза підтверджена: на імплементацію функціонально паритетних компонентів витрачається приблизно у 2.5–3 рази менше коду (див. окремий підрозділ про кількість коду).

H2. «У більшості сценаріїв коефіцієнт варіації та CVaR менші для DiVy». Гіпотеза підтверджена. Чим складніший сценарій (анімоване розкриття/складання, зміна теми), тим більшим є розрив на користь IMGUI. У легких режимах на кшталт наведення миші UI Toolkit інколи має незначну перевагу за CVaR, однак за стабільністю IMGUI випереджає у всіх сценаріях, включно зі спокійним режимом.

H3. «UI Toolkit матиме нижчі середні витрати в багатьох окремих сценаріях, а за умови постійної взаємодії — навпаки». Гіпотеза частково підтверджена. У простих/середніх сценаріях UI Toolkit часто має нижчі середні цикли та час, однак у складних (напр., зміна теми) і при тривалій активній взаємодії середні значення у нього зростають; IMGUI у цих умовах показав кращі результати, ніж очікувалося первісно.

H4. «DiVy має менші виділення пам’яті, ніж UI Toolkit». Гіпотеза загалом підтверджена для навантажених сценаріїв та піків. Для IMGUI у неоптимізованій

збірці спостерігається пропорційне зростання виділень зі зростанням кількості елементів, що ми пов’язуємо з ідентифікаційними даними/назвами, які усуваються під час оптимізованої збірки. У UI Toolkit чітко виражені піки виділення під час взаємодій (імовірно, створення/перебудова структур), тоді як IMGUI оперує попередньо зарезервованими масивами та уникає додаткових виділень пам’яті. На рисунку 3.36 спостерігається стабільно підвищене споживання під час відображення нових елементів і повернення до базового рівня після згортання списку.

Рис 3.36 Графік виділення пам’яті IMGUI у сценарії розгортання горизонтального випадаючого списку

Обговорення пікових затримок та практичні наслідки. У комбінованому сценарії максимальний час кадру в UI Toolkit сягав ≈8 мс, а під час розкриття вертикального меню — ≈13 мс. З огляду на простоту тестової панелі це свідчить про ризики для складніших застосунків. Крім того, тестова конфігурація потужніша за типові мобільні системи та VR-шоломи; на цільових платформах піки ймовірно будуть вищими. Для мобільних це означає деградацію зручності, для VR — критичні проблеми комфорту (симптоми кіберзахитування). Натомість DiVy у всіх тестах продемонструвала максимум близько ≈4.5 мс (≈218 FPS), що забезпечує запас для слабших систем. Подальша робота доцільна у двох напрямах: детальний аналіз піку в комбінованому сценарії (адже окремі підсценарії давали максимум лише до ≈1.6 мс) та дослідження паралелізації внутрішніх процесів бібліотеки для подальшого зниження пікових навантажень і підвищення придатності рішення для VR-платформ.

ВИСНОВКИ

У роботі теоретично обґрунтовано засади побудови бібліотеки на основі парадигми Immediate-mode GUI для динамічних застосунків та впорядковано критерії оцінювання їхньої продуктивності та інженерної придатності; методична новизна полягає у впорядкуванні одноетапного за кадр конвеєра, а прикладний результат — у створенні бібліотеки, придатної до безпосереднього використання в Unity.

Проведено комплексний аналіз методів побудови користувацьких інтерфейсів та існуючих рішень у середовищі Unity. Це дозволило обґрунтувати доцільність використання архітектури Immediate Mode GUI для забезпечення стабільності динамічних застосунків.

Сформульовано комплекс вимог до архітектури сучасної UI-бібліотеки, що включає підтримку розширених візуальних властивостей (округлення, контури, градієнти), повне адаптивне компонування та одноетапний конвеєр кадру.

Запропоновано та обґрунтовано однопрохідну схему побудови кадру, яка сумісна з механізмами адаптивного компонування та забезпечує високу передбачуваність часу виконання.

Доведено перевагу розробленого рішення над альтернативами в середовищі Unity за повнотою функціоналу, необхідного для створення сучасних різноманітних інтерфейсів.

Експериментальний аналіз підтвердив, що розроблене рішення зменшує необхідний обсяг коду в середньому в 3 рази. Розроблена архітектура продемонструвала істотно вищу стабільність: коефіцієнт варіації кадру становить 0.25 проти 0.6 у аналогів, а пікові затримки знижено з 6 млн до 2.8 млн циклів процесора.

Результати дослідження апробовано та опубліковано у наступних статті та тезах:

1. Перепелиця А.О, Герцюк М. М. Порівняння IMGUI та RMGUI парадигм. II Міжнародна науково-практична конференція «Сучасні аспекти

діджиталізації та інформатизації в програмній та комп’ютерній інженерії», 19 грудня 2024 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2024.

2. Перепелиця А.О., Дібрівний О.А. Аналіз підходів до адаптивного компонування та ідентифікування елементів в Immediate Mode GUI для їх використання в Unity// Зв'язок. № 1 (173), 2025. С. 35–42.

3. Перепелиця А.О., Дібрівний О.А. Порівняння кількості проходів за кадр в парадигмі Immediate Mode GUI. VI Всеукраїнська науково-технічна конференція "Застосування програмного забезпечення в інформаційно-комунікаційних технологіях", 24 квітня 2025 р., Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С. 223–228.

ПЕРЕЛІК ПОСИЛАНЬ

4. MonoBehaviour.OnGUI() [Електронний ресурс] // docs.unity3d.com. – Режим доступу: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnGUI.html (дата звернення: 26.11.2024). – Назва з екрана.

5. Micha Mettke [Електронний ресурс] : випуск 3 / ведуч. Р. Фльорі ; гість М. Меттке // The Handmade Network Podcast. – Режим доступу: https://handmade.network/podcast/ep/c1174949-adc4-492d-89b5-ca73dea4ff16 (дата звернення: 26.11.2024). – Назва з екрана.

6. Сміт Ф. Proving Immediate Mode GUIs are Performant. www.forrestthewoods.com. URL: https://www.forrestthewoods.com/blog/proving-immediate-mode-guis-are-performant/ (дата звернення: 11.12.2024).

7. Флорек М. Gallery: Post your screenshots / code here (PART 1). github.com. URL: https://github.com/ocornut/imgui/issues/123#issuecomment-142700775 (date of access: 11.12.2024).

8. Окорню О. FAQ (Frequently Asked Questions) [Електронний ресурс] / Омар Окорню // github.com. – Режим доступу: https://github.com/ocornut/imgui/blob/master/docs/FAQ.md (дата звернення: 26.11.2024). – Назва з екрана.

9. Роуз Г. vui [Електронний ресурс] / Генрі Роуз // github.com. – Режим доступу: https://github.com/heroseh/vui (дата звернення: 26.11.2024). – Назва з екрана.

10. Меттке М. Nuklear [Електронний ресурс] / Міша Меттке // github.com. – Режим доступу: https://github.com/vurtun/nuklear (дата звернення: 26.11.2024). – Назва з екрана.

11. ван Оортмерссен В. lobster [Електронний ресурс] / Воутер ван Оортмерссен // github.com. – Режим доступу: https://github.com/aardappel/lobster (дата звернення: 26.11.2024). – Назва з екрана.

12. Целіс П., Ларсон П.-А., Мунро Д. І. Robin Hood Hashing. *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, м. Portland, штат Орегон, США, 21–23 жовт. 1985 р. 1985. Режим доступу: https://doi.org/10.1109/sfcs.1985.48 (дата звернення: 17.11.2025).

13. Гуссаерт Е. Robin Hood hashing: backward shift deletion. codecapsule.com. URL: https://codecapsule.com/2013/11/17/robin-hood-hashing-backward-shift-deletion/ (дата звернення: 17.11.2025).

14. Коллет Й. Hash Algorithms - Performance comparison. github.com. URL: https://github.com/Cyan4973/xxHash/wiki/Performance-comparison#benchmarks-concentrating-on-small-data-/ (дата звернення: 17.11.2025).

15. Де Карлі Н. rapidhash - Very fast, high quality, platform-independent. github.com. URL: https://github.com/Nicoshev/rapidhash/ (дата звернення: 17.11.2025).

16. Коллет Й. Collision ratio comparison. github.com. URL: https://github.com/Cyan4973/xxHash/wiki/Collision-ratio-comparison#collision-study/ (дата звернення: 17.11.2025).

17. Арнольд А., Маррон М. Catalpa: GC for a Low-Variance Software Stack. *arXiv preprint arXiv:2509.13429*. 2025. URL: https://arxiv.org/abs/2509.13429 (дата звернення: 17.11.2025).

18. Когіас М., Буньйон Е. Tail-tolerance as a Systems Principle not a Metric. *4th Asia-Pacific Workshop on Networking*. 2020. С. 16–22. URL: https://doi.org/10.1145/3411029.3411032 (дата звернення: 17.11.2025).

19. Томчак Л. GPU Ray Marching of Distance Fields : магістерська робота. Люнгбю-Торбек, 2012. URL: https://doi.org/10.1016/j.net.2017.08.008 (дата звернення: 17.11.2025).

20. Баркер Н. Clay, A UI Layout Library. github.com. URL: https://github.com/nicbarker/clay (дата звернення: 17.11.2025).

21. Гонзалез П., Лов Д. The Book of Shaders. 2015. URL: https://thebookofshaders.com/ (дата звернення: 17.11.2025).

22. Портер Т., Дафф Т. Compositing digital images. *ACM SIGGRAPH Computer Graphics*. 1984. Т. 18, № 3. С. 253–259. URL: https://doi.org/10.1145/964965.808606 (дата звернення: 17.11.2025).

23. Рокафеллар Р, Урясев С. Optimization of conditional value-at-risk. *The Journal of Risk*. 2000. Т. 2, № 3. С. 21–41. URL: https://doi.org/10.21314/jor.2000.038 (дата звернення: 17.11.2025).

24. Монтгомері Д., Рунгер Г. Applied Statistics and Probability for Engineers. Wiley & Sons, Incorporated, John, 2013.

25. Мітценмахер М, Упфаль Е. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, 2012.

26. Кнут Д. Art of Computer Programming. Pearson Education, Limited, 2022. 640 с.

27. Майерс Б. Past, Present and Future of User Interface Software Tools. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. 2000. Т. 44, № 2. С. 319. URL: https://doi.org/10.1177/154193120004400206 (дата звернення: 17.11.2025).

28. Мураторі К. Immediate-Mode Graphical User Interfaces - 2005 [Електронний ресурс], 2016 / Кейсі Мураторі // youtube.com. – Режим доступу: https://youtu.be/Z1qyvQsjK5Y (дата звернення: 26.11.2024). – Назва з екрана.

29. Рейенскауг Т. Models-Views-Controllers. *https://jpaulgibson.synology.me/~jpaulgibson/TSP/Teaching/Teaching-ReadingMaterial/Reenskaug79b.pdf*.

30. ван Бернем С. Л. Nbui : магістерська робота. Аахен, 2019.

31. Брендель Ф., Лідтке С. Exploring the Immediate Mode GUI Concept for Graphical User Interfaces in Mixed Reality Applications. GI VR / AR Workshop

2022, м. Гамбург, 14–16 верес. 2022 р. Гамбург, 2022. URL: https://dl.gi.de/server/api/core/bitstreams/b8c94eda-1405-4b23-bc9f-e184cad04602/content (дата звернення: 11.12.2024).

32. Кей А. С. The early history of Smalltalk. ACM SIGPLAN Notices. 1993. Т. 28, № 3. С. 69–95. URL: https://doi.org/10.1145/155360.155364 (дата звернення: 11.12.2024).

33. Ольссон Т., Еріксон М. An exploration and ex- periment tool suite for code to architecture mapping techniques. Proceedings of the 13th European Conference on Software Architecture. 2019. Т. 2. С. 26–29.

34. Покорни П., Шевчик Д. An application for solving truth function. Intelligent Algorithms in Software Engineering. 2020. С. 341–351.

35. CSS Flexible Box Layout Module Level 1. *W3C*. URL: https://www.w3.org/TR/css-flexbox-1/ (дата звернення: 17.11.2025).

36. Бадрос Г., Борнінг А., Стакі П. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction*. 2001. Т. 8, № 4. С. 267–306. URL: https://doi.org/10.1145/504704.504705 (дата звернення: 17.11.2025).

37. Акенін-Мьолер Т. Real-Time Rendering. Fourth edition. | Boca Raton : Taylor & Francis, CRC Press, 2018. : A K Peters/CRC Press, 2018. URL: https://doi.org/10.1201/b22086 (дата звернення: 17.11.2025).

38. Фльорі Р. UI, Part 2: Every Single Frame (IMGUI). www.rfleury.com. URL: https://www.rfleury.com/p/ui-part-2-build-it-every-frame-immediate (дата звернення: 11.12.2024).

39. Фуско Л. Providing a remote DebugGUI to DPL. 2021. URL: https://cds.cern.ch/record/2780603/files/CERN%20Report.pdf (дата звернення: 10.12.2024).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Магістерська робота

«Підвищення продуктивності динамічних інтерфейсів за допомогою архітектури однопрохідного IMGUI з адаптивним компонуванням»

Виконав: студент групи ПДМ-63 Артем ПЕРЕПЕЛИЦЯ

Керівник: канд. техн. наук, завідувач кафедри Інтернет-технологій
В’ячеслав ТРЕЙТЯК

Київ - 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення стабільності продуктивності динамічних інтерфейсів кінцевих застосунків за допомогою архітектури Immediate-mode GUI-бібліотеки для Unity, орієнтованої на інтерфейси кінцевих застосунків, яка забезпечує стабільну продуктивність у динамічних сценах, підтримує сучасні вимоги до стилізації та адаптивного компонування.

Об’єкт дослідження: процес побудови та відтворення графічних інтерфейсів користувача в середовищі Unity для динамічних інтерактивних застосунків.

Предмет дослідження: архітектурні принципи, алгоритмічні та інженерні засоби реалізації бібліотеки Immediate-mode GUI з однопрохідним виконанням за кадр, адаптивним компонуванням і розширеною стилізацією.

АКТУАЛЬНІСТЬ РОБОТИ

Категорія	Unity IMGUI	UGUI	UI Toolkit
Продуктивність в динамічних інтерфейсах	Низька	Середня	Низька
Адаптивне компонування	Обмежене, не підтримується переніс рядків	Відсутнє, розмір дочірніх контейнерів не впливає на батьківський	Майже повне, не підтримуються проміжки, відсоткові розміри, фіксована пропорція сторін
Візуальний функціонал	Розмитий текст, відсутність градієнтів	Не підтримуються закруглені кути, контури	Не підтримуються градієнти

3

АКТУАЛЬНІСТЬ РОБОТИ

Retained-mode GUI — статичне дерево елементів, ефективність у статичних інтерфейсах.

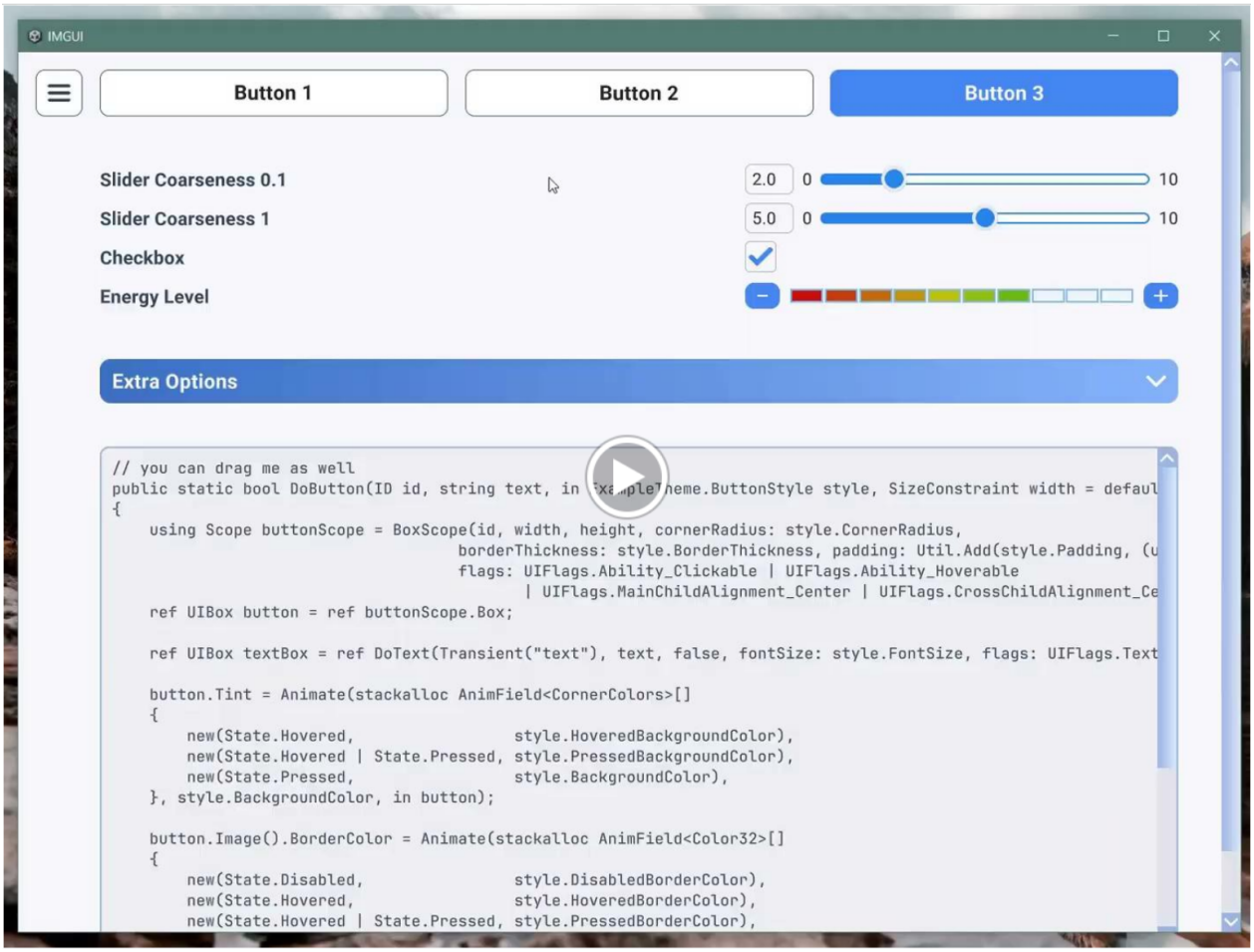
```
var button = new Button();
button.onClick += …
```

Immediate-mode GUI — дерево постійно перебудовується, стабільність у динамічних сценаріях.

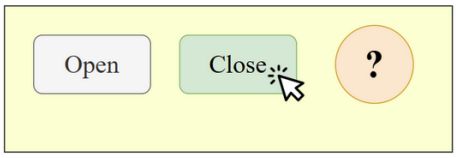
```
if (DoButton(«Open»)) {
    …
}
```

Недоліки існуючих IMGUI бібліотек:
- потреба у подвійному виклику функцій для адаптивного компонування (двопрохідність)
- компроміси при ідентифікації елементів: відсутність ідентифікаторів ускладнює деякі види взаємодій; колізії числових ідентифікаторів складно відслідковувати.

4

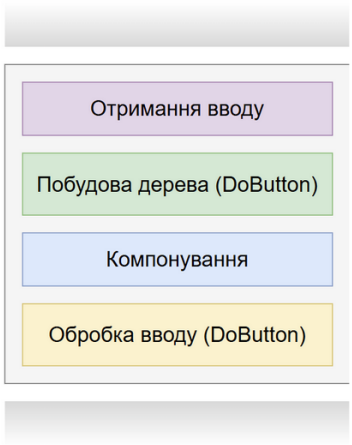

IMGUI
Button 1
Button 2
Button 3
Slider Coarseness 0.1
2.0
0
10
Slider Coarseness 1
5.0
0
10
Checkbox
Energy Level
Extra Options
// you can drag me as well
public static bool DoButton(ID id, string text, in
neme.ButtonStyle style, SizeConstraint width = defaul
{
using Scope buttonScope = BoxScope(id, width, height, cornerRadius: style.CornerRadius,
borderThickness: style.BorderThickness, padding: Util.Add(style.Padding, (u
flags: UIFlags.Ability_Clickable | UIFlags.Ability_Hoverable
| UIFlags.MainChildAlignment_Center | UIFlags.CrossChildAlignment_Ce
ref UIBox button = ref buttonScope.Box;
ref UIBox textBox = ref DoText(Transient("text"), text, false, fontSize: style.FontSize, flags: UIFlags.Text
button.Tint = Animate(stackalloc AnimField<CornerColors>[]
{
new(State.Hovered, style.HoveredBackgroundColor),
new(State.Hovered | State.Pressed, style.PressedBackgroundColor),
new(State.Pressed, style.BackgroundColor),
}, style.BackgroundColor, in button);
button.Image().BorderColor = Animate(stackalloc AnimField<Color32>[]
{
new(State.Disabled, style.DisabledBorderColor),
new(State.Hovered, style.HoveredBorderColor),
new(State.Hovered | State.Pressed, style.PressedBorderColor),

НОВИЙ АЛГОРИТМ UI КОНВЕЄРА З ОДНІЄЮ ФАЗОЮ



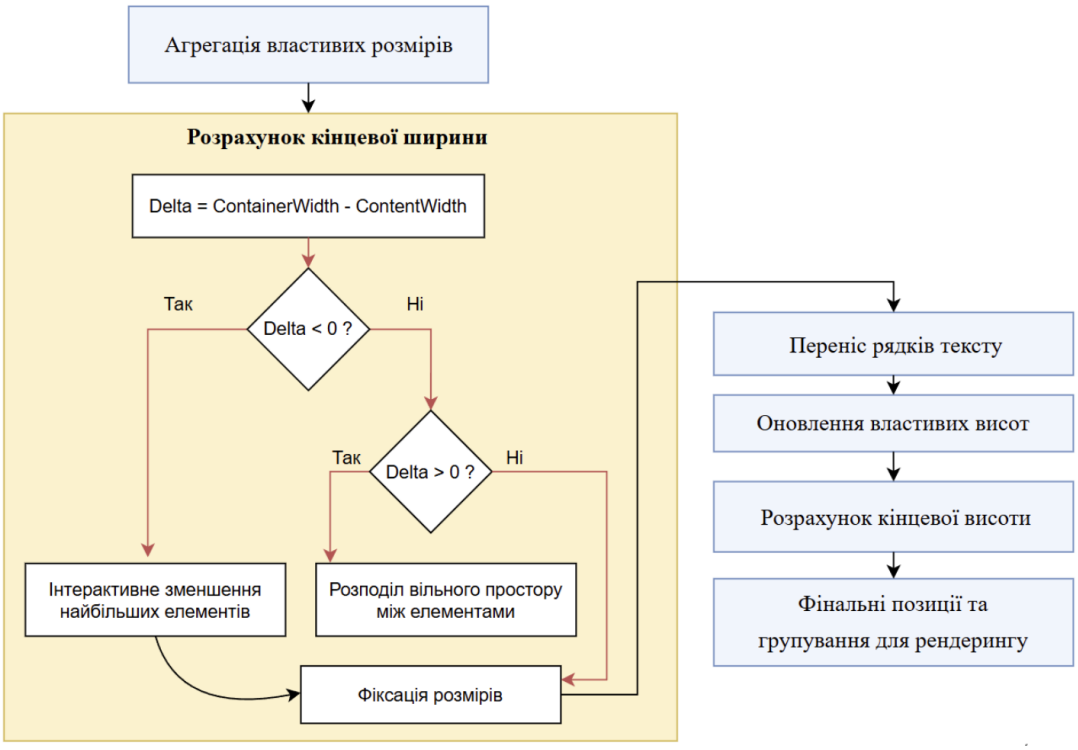

```
if (DoButton(«Close»)) {
    …
}
```

Старий алгоритм з подвійним викликом DoButton

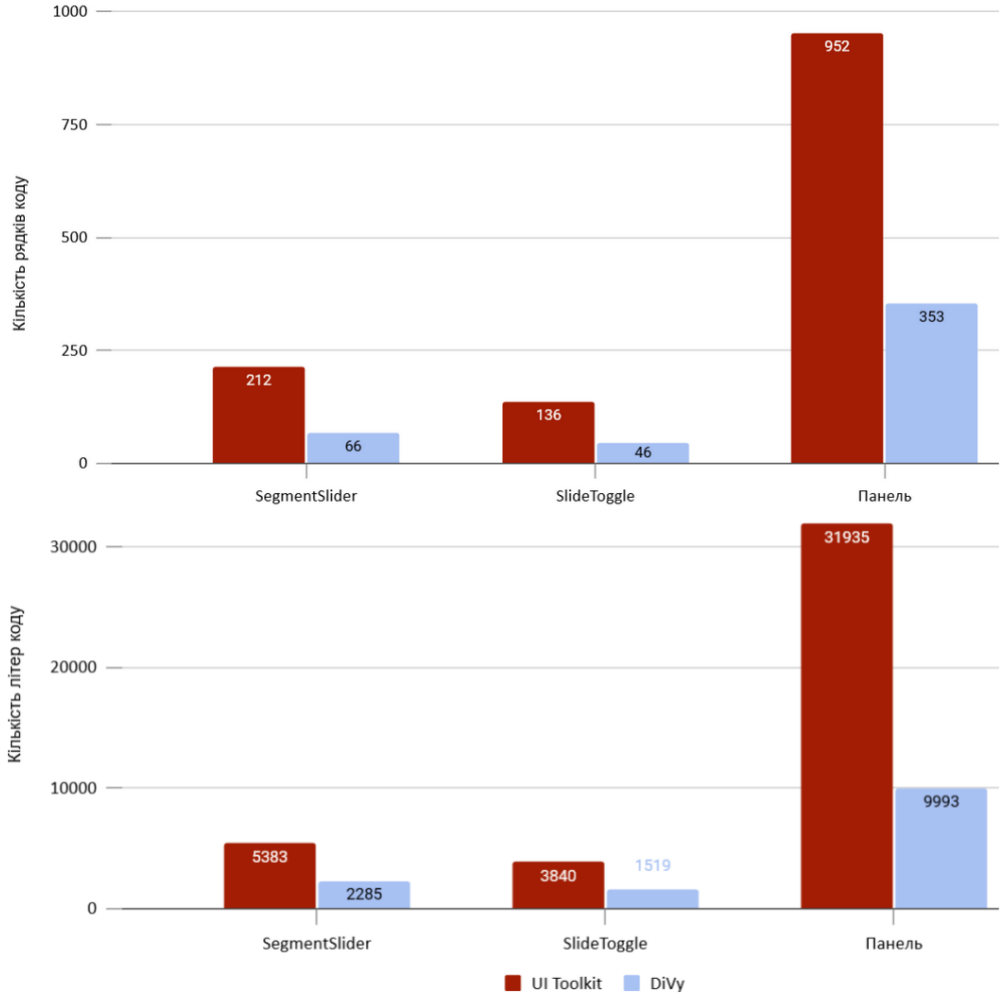


Новий алгоритм

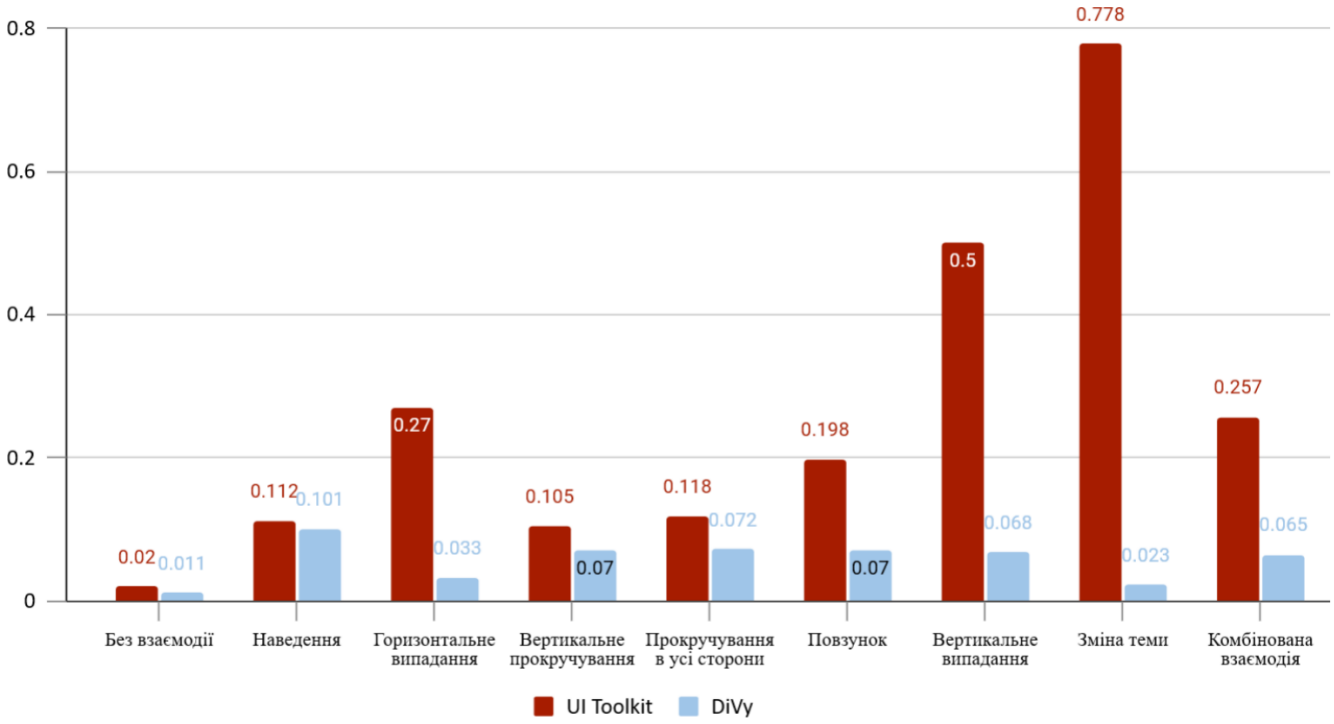
АЛГОРИТМ ЩОКАДРОВОГО АДАПТИВНОГО КОМПОНУВАННЯ



ПОРІВНЯЛЬНИЙ АНАЛІЗ КІЛЬКОСТІ КОДУ

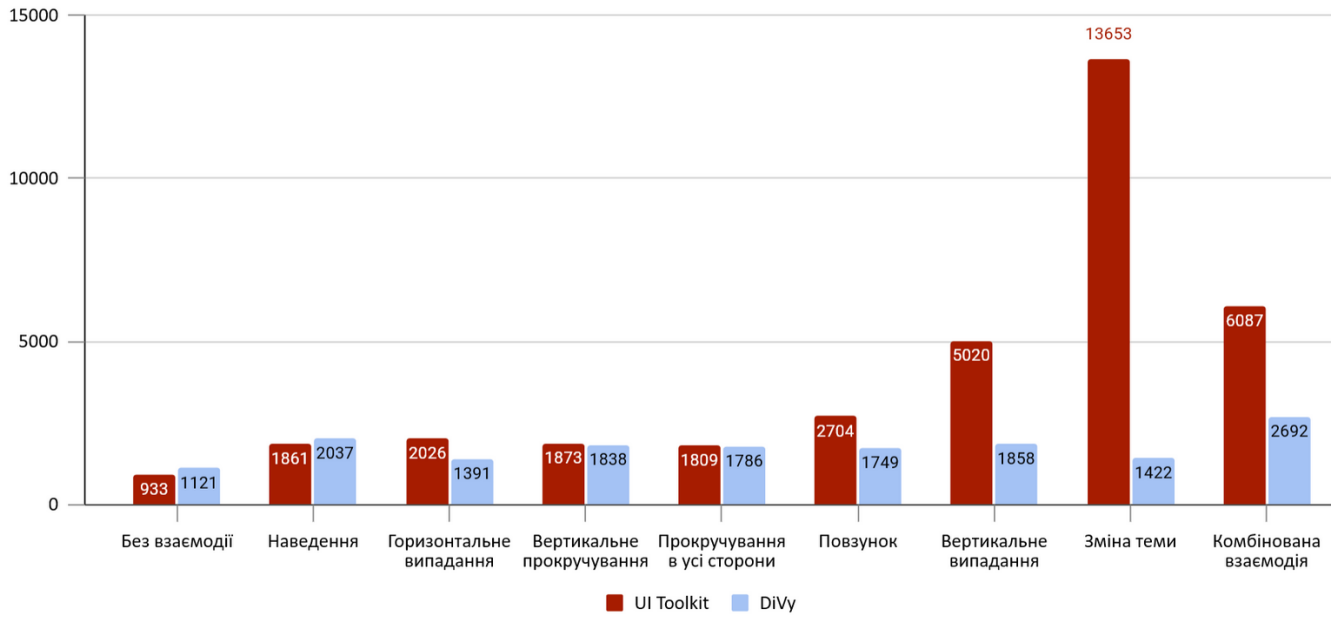


ПОРІВНЯЛЬНИЙ АНАЛІЗ СТАБІЛЬНОСТІ КАДРІВ



9

ПОРІВНЯЛЬНИЙ АНАЛІЗ CVaR 1% КІЛЬКОСТІ ЦИКЛІВ, ТИС



$$CVaR_{1\%} = \frac{1}{m} \sum_{k=N-m+1}^{N} y^{(k)}, m = ceil(0.01\,N)$$

, де $CVaR_{1\%}$ — середнє значення у 1% найгірших кадрів; m — кількість кадрів у хвості; $y^{(k)}$ — k-та порядкова статистика; N — кількість кадрів.

10

ВИСНОВКИ

1. Проведено комплексний аналіз методів побудови користувацьких інтерфейсів та існуючих рішень у середовищі Unity. Це дозволило обґрунтувати доцільність використання архітектури Immediate Mode GUI для забезпечення стабільності динамічних застосунків.
2. Сформульовано комплекс вимог до архітектури сучасної UI-бібліотеки, що включає підтримку розширених візуальних властивостей (округлення, контури, градієнти), повне адаптивне компонування та одноетапний конвеєр кадру.
3. Запропоновано та обґрунтовано однопрохідну схему побудови кадру, яка сумісна з механізмами адаптивного компонування та забезпечує високу передбачуваність часу виконання.
4. Доведено перевагу розробленого рішення над альтернативами в середовищі Unity за повнотою функціоналу, необхідного для створення сучасних різноманітних інтерфейсів.
5. Експериментальний аналіз підтвердив, що розроблене рішення зменшує необхідний обсяг коду в середньому в 3 рази. Розроблена архітектура продемонструвала істотно вищу стабільність: коефіцієнт варіації кадру становить 0.25 проти 0.6 у аналогів, а пікові затримки знижено з 6 млн до 2.8 млн циклів процессора.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті:

Перепелиця А.О., Дібрівний О.А. Аналіз підходів до адаптивного компонування та ідентифікування елементів в Immediate Mode GUI для їх використання в Unity. Зв'язок. 2025. № 1 (173). С. 35–42.

Тези доповідей:

1. Перепелиця А.О, Герцюк М. М. Порівняння IMGUI та RMGUI парадигм. II Міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп’ютерній інженерії». Збірник тез. – Київ // ДУІКТ, 2024.

2. Перепелиця А.О., Дібрівний О.А. Порівняння кількості проходів за кадр в парадигмі Immediate Mode GUI. VI Всеукраїнська науково-технічна конференція "Застосування програмного забезпечення в інформаційно- комунікаційних технологіях". 2025. С. 223–228.

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

1. Структура UIBox

```
[StructLayout(LayoutKind.Auto)]
public partial struct UIBox
{
   public ImageTextUnion ImageTextUnion;
   public string Text; // can't store it in the union because it's a
managed object.
   public CornerColors Tint;

   public UIFlags Flags;
   public int LastTouchedFrame;
   public Ptr<UIBox> Parent;
   public Ptr<UIBox> FirstChild;
   public Ptr<UIBox> NextSibling;
   public Ptr<UIBox> Self;
   public ID ID;
   public int TreeDepth; // It's not used right now, but we are going
to use it for sorting.

   public float AnimT;
   public float CustomAnimT; // A field you can use for the
custom animations. Set it to 0 and 1, and lerp yourself.

   public float2 RelativeOffset;
   public float2 RelativeOffsetTarget;
   public float2 MoveDelta;
   public Vector2<SizeConstraintUnion> SizeConstraints;
   public UIRect ScreenRect;
   internal float2 MinSize;

   public float AspectRatio; // float / height. If 0 -> no aspect ratio.
   public Padding Padding;
   public ushort ChildGap;

   public readonly Actions Actions // This property is useful in
case you want to check multiple flags at once
   {
      [MethodImpl(MethodImplOptions.AggressiveInlining)]
      get => Flags.GetActions();
   }

   public readonly bool PressedThisFrame =>
Flags.IsAnyFlagEnabled(UIFlagsEx.Action_Pressed);
   public readonly bool Clicked         =>
Flags.IsAnyFlagEnabled(UIFlagsEx.Action_Clicked);
   public readonly bool DoubleClicked   =>
Flags.IsAnyFlagEnabled(UIFlagsEx.Action_DoubleClicked);
   public readonly bool RightClicked    =>
Flags.IsAnyFlagEnabled(UIFlagsEx.Action_RightClicked);
   public readonly bool HoldClicked     =>
Flags.IsAnyFlagEnabled(UIFlagsEx.Action_HoldClicked);

   public readonly bool IsPressed       =>
Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_Pressed);
   public readonly bool IsHovered       =>
Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_MouseOver);
   public readonly bool IsMouseOver     =>
Flags.IsAnyFlagEnabled(UIFlagsEx.CurrentState_Hovered);

   public readonly bool IsFirstFrame =>
Flags.IsAnyFlagEnabled(UIFlagsEx.IsFirstFrame);

   internal readonly bool IsPersistent => ID.Hash != 0;

#if DEBUG
   public override readonly string ToString() =>
$"{ID.DebugString}({(IsPersistent? "persistent" : "transient")})";
#endif
}
```

2. Публічні функції створення елементів

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static ref UIBox DoBox(ID id, SizeConstraint width =
default, SizeConstraint height = default, UIFlags flags = default,
                     CornerColors tint = default, float aspectRatio =
0f, Padding padding = default,
                     ushort childGap = default, Sprite sprite = null,
Rect customTextureRect = default, float cornerRadius = 0,
                     float edgeSoftness = 1f, float borderThickness
= 0f, Color32 borderColor = default)
{
   ref UIBox box = ref StartBox(id, width, height, flags, tint,
aspectRatio, padding, childGap, sprite, customTextureRect,
                      cornerRadius, edgeSoftness, borderThickness,
borderColor);
   EndBox();
   return ref box;
}

public static ref UIBox StartBox(ID id, SizeConstraint width =
default, SizeConstraint height = default, UIFlags flags = default,
                     CornerColors tint = default, float aspectRatio
= 0f, Padding padding = default,
                     ushort childGap = default, Sprite sprite = null,
Rect customTextureRect = default, float cornerRadius = 0,
                     float edgeSoftness = 1f, float borderThickness
= 0f, Color32 borderColor = default)
{
   ref UIBox newBox = ref GetOrCreateUIBox(id, width, height,
flags, tint, aspectRatio, padding, childGap, true);

   if (sprite is null) // No need to check the lifetime. The user
should either pass null or a live texture.
   {
      newBox.ImageTextUnion.Image.Texture = _whitePixel;
      newBox.ImageTextUnion.Image.TextureRect =
_whitePixelTextureRect;
   }
   else
   {
      newBox.ImageTextUnion.Image.Texture = sprite.texture;
      newBox.ImageTextUnion.Image.TextureRect =
customTextureRect == default ? sprite.textureRect :
customTextureRect;
   }

   newBox.ImageTextUnion.Image.CornerRadius = cornerRadius;
   newBox.ImageTextUnion.Image.EdgeSoftness = edgeSoftness;
   newBox.ImageTextUnion.Image.BorderThickness =
borderThickness;
   newBox.ImageTextUnion.Image.BorderColor = borderColor;

   return ref newBox;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void EndBox() => EndScope();

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static Scope BoxScope(ID id, SizeConstraint width =
default, SizeConstraint height = default, UIFlags flags = default,
```

```
                    CornerColors tint = default, float aspectRatio =
0f, Padding padding = default,
                    ushort childGap = default, Sprite sprite = null,
Rect customTextureRect = default, float cornerRadius = 0,
                    float edgeSoftness = 1f, float borderThickness =
0f, Color32 borderColor = default)
{
   ref UIBox box = ref StartBox(id, width, height, flags, tint,
aspectRatio, padding, childGap, sprite, customTextureRect,
                     cornerRadius, edgeSoftness, borderThickness,
borderColor);
   return new(box.Self);
}

public static ref UIBox DoText(ID id, string text, bool
isStableInstance, SizeConstraint width = default, SizeConstraint
height = default,
                    UIFlags flags = default, Color32 tint = default,
float aspectRatio = 0f,
                    Padding padding = default, MSDFFontAsset
font = default, float fontSize = default)
{
   tint = (tint is {r: 0, g: 0, b: 0, a: 0} &&
flags.IsAnyFlagDisabled(UIFlags.Tint_OverrideTransparent)) ?
COLOR_WHITE : tint;
   ref UIBox newBox = ref GetOrCreateUIBox(id, width, height,
flags, CornerColors.Uniform(tint), aspectRatio, padding, 0, false);

   newBox.ImageTextUnion.Text.Font = font == default ?
_defaultFont : font;
   newBox.Text = text ?? string.Empty; // just so that we don't
have to check for null later.
   newBox.ImageTextUnion.Text.FontSize = fontSize == default ?
_defaultFontSize : fontSize;

   ulong isStableInstanceUlong = Unsafe.As<bool, byte>(ref
isStableInstance);
   newBox.Flags = newBox.Flags.AddFlag((UIFlags)
(isStableInstanceUlong <<
UIFlagsEx.TextIsStableInstanceOffset));
   newBox.Flags =
newBox.Flags.AddFlag(UIFlagsEx.BoxType_Text);

   ExitScopes(newBox.Self, in newBox); // run this immediately
for text because it cannot have children.

   return ref newBox;
}
```

3. Функція SizeContainersAlongAxis – другий етап автоматичного компонування

```
static void SizeContainersAlongAxis(int axis)
{
   for (Ptr<UIBox> ptr = _rootBoxPtr; ptr.IsNotNull; )
   {
      ref UIBox containerBox = ref _boxPool[ptr];

      if (containerBox.FirstChild.IsNotNull)
      {
         int growContainerCount = 0;
         float containerSize = containerBox.ScreenRect.Size[axis];

         float parentPadding =
containerBox.Padding.GetPaddingSumAlongDirection(axis);

         float innerContentSize = 0;
         float totalPaddingAndChildGaps = parentPadding;
         bool sizingAlongAxis =
containerBox.Flags.GetLayoutDirectionInt() == axis;
         _resizableChildrenBuffer.Clear(false);
         _resizableChildrenSecondaryBuffer.Clear(false);

         for (Ptr<UIBox> childPtr = containerBox.FirstChild;
childPtr.IsNotNull; )
         {
            ref UIBox child = ref _boxPool[childPtr];

            float childSize = child.ScreenRect.Size[axis];
            SizeConstraintType constraintType =
child.Flags.GetSizingAlongAxis(axis);

            if (constraintType < SizeConstraintType.Percent
               && (child.Flags.GetBoxType() !=
UIFlagsEx.BoxType_Text || child.Flags.IsTextWrapEnabled()))
            {
               _resizableChildrenBuffer.Add(childPtr);
            }

            if
(child.Flags.IsAnyFlagDisabled(UIFlagsEx.Layout_ExcludeFrom
Layout))
            {
               if (sizingAlongAxis)
               {
                  innerContentSize += constraintType ==
SizeConstraintType.Percent ? 0 : childSize;

                  if (constraintType == SizeConstraintType.Grow)
                     ++growContainerCount;

                  if (childPtr != containerBox.FirstChild)
                  {
                     innerContentSize += containerBox.ChildGap; //
For children after index 0, the childAxisOffset is the gap from the
previous child
                     totalPaddingAndChildGaps +=
containerBox.ChildGap;
                  }
               }
               else
               {
                  innerContentSize = Mathf.Max(childSize,
innerContentSize);
               }
            }

            childPtr = child.NextSibling;
         }

         // expand percentage containers to size
         for (ref UIBox child = ref
_boxPool[containerBox.FirstChild];
              !Unsafe.IsNullRef(ref child);
              child = ref _boxPool[child.NextSibling])
         {
            if (child.Flags.GetSizingAlongAxis(axis) ==
SizeConstraintType.Percent)
            {
               child.ScreenRect.Size[axis] = (containerSize -
totalPaddingAndChildGaps) * child.SizeConstraints[axis].Percent;

               if (sizingAlongAxis)
                  innerContentSize += child.ScreenRect.Size[axis];

               UpdateAspectRatioBox(ref child);
            }
         }

         if (sizingAlongAxis)
         {
```

```
            float sizeToDistribute = containerSize - parentPadding -
innerContentSize;

            // The content is too large, compress the children as
much as possible
            if (sizeToDistribute < 0)
            {
               bool shrinkChildren = true;

               // If the parent clips content in this axis direction,
don't compress children, just leave them alone
               if
(containerBox.Flags.IsAnyFlagEnabled(UIFlags.IsMask))
               {
                  ref UIBox firstChild = ref
_boxPool[containerBox.FirstChild];
                  UIFlags moveConstraint =
firstChild.Flags.GetMoveConstraints();

                  if (!(moveConstraint ==
UIFlags.MoveConstraint_Horizontal && axis == 1
                     || moveConstraint ==
UIFlags.MoveConstraint_Vertical && axis == 0)
                     && moveConstraint !=
UIFlags.MoveConstraint_Fixed)
                  {
                     shrinkChildren = false;
                  }
               }

               static float ShrinkChildren(float sizeToDistribute, int
axis, ref StructList<Ptr<UIBox>> resizableChildren, ref
StructList<Ptr<UIBox>> backupList)
               {
                  while (sizeToDistribute < -EPSILON &&
resizableChildren.Count > 0)
                  {
                     float largest = 0;
                     float secondLargest = 0;
                     float sizeToAdd = sizeToDistribute;

                     for (int i = 0; i < resizableChildren.Count; ++i)
                     {
                        ref UIBox child = ref
_boxPool[resizableChildren[i]];
                        float childSize = child.ScreenRect.Size[axis];

                        if (ApproxEqual(childSize, largest))
                           continue;

                        if (childSize > largest)
                        {
                           secondLargest = largest;
                           largest = childSize;
                        }
                        else
                        {
                           secondLargest = Mathf.Max(secondLargest,
childSize);
                           sizeToAdd = secondLargest - largest;
                        }
                     }

                     sizeToAdd = Mathf.Max(sizeToAdd,
sizeToDistribute / resizableChildren.Count);

                     for (int i = 0; i < resizableChildren.Count; ++i)
                     {
                        Ptr<UIBox> childPtr = resizableChildren[i];
                        ref UIBox child = ref _boxPool[childPtr];
                        if (ApproxEqual(child.ScreenRect.Size[axis],
largest))
                        {
                           float softMinSize = child.MinSize[axis];
                           float previousSize =
child.ScreenRect.Size[axis];
                           child.ScreenRect.Size[axis] += sizeToAdd;
                           if (child.ScreenRect.Size[axis] <=
softMinSize)
                           {
                              child.ScreenRect.Size[axis] =
softMinSize;

resizableChildren.RemoveSwapBack(i--);

                              if (!Unsafe.IsNullRef(ref backupList)
&& softMinSize > child.SizeConstraints[axis].MinMax.Min +
EPSILON)
                              {
                                 // We no longer will need to use the
soft min size during the frame, so we can change it to the hard min
size.
                                 // We will use the backup list to further
compress the children if the container still hasn't shrunk enough.
                                 child.MinSize[axis] =
child.SizeConstraints[axis].MinMax.Min;
                                 backupList.Add(childPtr);
                              }
                           }

                           sizeToDistribute -=
child.ScreenRect.Size[axis] - previousSize;
                        }
                     }
                  }

                  return sizeToDistribute;
               }

               if (shrinkChildren)
               {
                  sizeToDistribute =
ShrinkChildren(sizeToDistribute, axis, ref
_resizableChildrenBuffer, ref
_resizableChildrenSecondaryBuffer);
                  ShrinkChildren(sizeToDistribute, axis, ref
_resizableChildrenSecondaryBuffer, ref
Unsafe.NullRef<StructList<Ptr<UIBox>>>());
               }
            }
            // The content is too small, allow SIZING_GROW
containers to expand
            else if (sizeToDistribute > 0 && growContainerCount >
0)
            {
               for (int i = 0; i < _resizableChildrenBuffer.Count; +
+i)
               {
                  ref UIBox child = ref
_boxPool[_resizableChildrenBuffer[i]];
                  SizeConstraintType constraintType =
child.Flags.GetSizingAlongAxis(axis);
                  if (constraintType != SizeConstraintType.Grow)
                  {

_resizableChildrenBuffer.RemoveSwapBack(i--);
                  }
               }
```

```
            while (sizeToDistribute > EPSILON &&
_resizableChildrenBuffer.Count > 0)
            {
               float smallest = float.MaxValue;
               float secondSmallest = float.MaxValue;
               float sizeToAdd = sizeToDistribute;

               for (int i = 0; i < _resizableChildrenBuffer.Count; +
+i)
               {
                  ref UIBox child = ref
_boxPool[_resizableChildrenBuffer[i]];
                  if (ApproxEqual(child.ScreenRect.Size[axis],
smallest))
                     continue;

                  if (child.ScreenRect.Size[axis] < smallest)
                  {
                     secondSmallest = smallest;
                     smallest = child.ScreenRect.Size[axis];
                  }
                  else
                  {
                     secondSmallest = Mathf.Min(secondSmallest,
child.ScreenRect.Size[axis]);
                     sizeToAdd = secondSmallest - smallest;
                  }
               }

               sizeToAdd = Mathf.Min(sizeToAdd,
sizeToDistribute / _resizableChildrenBuffer.Count);

               for (int i = 0; i < _resizableChildrenBuffer.Count; +
+i)
               {
                  ref UIBox child = ref
_boxPool[_resizableChildrenBuffer[i]];
                  float maxSize =
child.SizeConstraints[axis].MinMax.Max;
                  float previousSize = child.ScreenRect.Size[axis];

                  if (ApproxEqual(child.ScreenRect.Size[axis],
smallest))
                  {
                     child.ScreenRect.Size[axis] += sizeToAdd;

                     if (child.ScreenRect.Size[axis] >= maxSize)
                     {
                        child.ScreenRect.Size[axis] = maxSize;

_resizableChildrenBuffer.RemoveSwapBack(i--);
                     }

                     sizeToDistribute -=
child.ScreenRect.Size[axis] - previousSize;
                  }
               }
            }
         }
         // Sizing along the non layout axis ("off axis")
         else
         {
            for (int m = 0; m < _resizableChildrenBuffer.Count; +
+m)
            {
               ref UIBox child = ref
_boxPool[_resizableChildrenBuffer[m]];
               float minSize =
child.SizeConstraints[axis].MinMax.Min;
               float maxSize = containerSize - parentPadding;

               if
(containerBox.Flags.IsAnyFlagEnabled(UIFlags.IsMask))
               {
                  // With this set of operations we get false if:
                  // MoveConstraint_HorizontalOnly and
VerticalAxis
                  // MoveConstraint_VerticalOnly and
HorizontalAxis
                  int moveConstraintAxis = unchecked((int)
((ulong)child.Flags >> UIFlagsEx.MoveConstraintsOffset));

                  // TODO: I added the fixed flag and broke a single
branch using bit operations. Need to find a new way to branch
only once here.
                  if (((~(moveConstraintAxis ^ axis)) & 0b11) != 0
&& child.Flags.GetMoveConstraints() !=
UIFlags.MoveConstraint_Fixed)
                  {
                     maxSize = Mathf.Max(maxSize,
innerContentSize);
                  }
               }

               SizeConstraintType constraintType =
child.Flags.GetSizingAlongAxis(axis);

               if (constraintType == SizeConstraintType.Grow)
                  child.ScreenRect.Size[axis] = Mathf.Min(maxSize,
child.SizeConstraints[axis].MinMax.Max);

               child.ScreenRect.Size[axis] =
Mathf.Clamp(child.ScreenRect.Size[axis], minSize, maxSize);
            }
         }

         ptr = containerBox.FirstChild;
         continue;
      }

      if (containerBox.NextSibling.IsNotNull)
      {
         ptr = containerBox.NextSibling;
      }
      else
      {
         Ptr<UIBox> parentPtr = containerBox.Parent;

         while (true)
         {
            if (parentPtr.IsNull)
            {
               ptr = default;
               break;
            }

            ref UIBox parent = ref _boxPool[parentPtr];

            if (parent.NextSibling.IsNotNull)
            {
               ptr = parent.NextSibling;
               break;
            }

            parentPtr = parent.Parent;
         }
      }
   }
}
```

ДОДАТОК В. МАТЕРІАЛИ ТА РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ ХЕШ-КАРТ

```
1 usage
private static double MeasureGetOrAddHit_Dictionary(int capacity, int[] keys, Value32[] vals)
{
    // warmup: попередньо заповнюємо всі ключі
    for (int w = 0; w < WarmupIters; w++)
    {
        var m = new Dictionary<int, Value32>(capacity);
        for (int i = 0; i < N; i++) m.Add(keys[i], vals[i]);
        long s = 0;
        for (int i = 0; i < N; i++)
        {
            if (m.TryGetValue(keys[i], out var v)) s += v.C;
        }
        _sink += s;
    }

    double totalNs = 0;
    for (int r = 0; r < MeasureIters; r++)
    {
        ForceGc();
        var m = new Dictionary<int, Value32>(capacity);
        for (int i = 0; i < N; i++) m.Add(keys[i], vals[i]);

        var sw = Stopwatch.StartNew();
        long s = 0;
        for (int i = 0; i < N; i++)
        {
            // емулюємо "GetOrAdd" для існуючого елементу як TryGetValue
            if (m.TryGetValue(keys[i], out var v)) s += v.D;
        }
        sw.Stop();
        _sink += s;
        totalNs += ElapsedNs(sw);
    }
    var ns = totalNs / MeasureIters / N;
    Console.WriteLine($"{nameof(Dictionary<int, Value32>),-28} GetOrAdd.Hit    {ns,10:F2} ns/op");
    return ns;
}
```

Рис В.1 Фрагмент коду бенчмарку з тестом успішної операції GetOrAddValue на Dictionary

```
1 usage
private static Metric MeasureGetOrAddMiss_RHSplit(string name, int capacity, int[] keys, Value32[] vals)
{
    for (int w = 0; w < WarmupIters; w++)
    {
        var m = new RobinHoodSplitHashTable<int, Value32>(capacity);
        long s = 0;
        for (int i = 0; i < N; i++)
        {
            bool exists;
            ref Value32 slot = ref m.GetOrAddValue(keys[i], out exists);
            if (!exists) slot = vals[i];
            s += slot.A;
        }
        _sink += s;
    }

    var trials = new double[MeasureIters];
    for (int rep = 0; rep < MeasureIters; rep++)
    {
        ForceGc();
        var m = new RobinHoodSplitHashTable<int, Value32>(capacity);

        var sw = Stopwatch.StartNew();
        long s = 0;
        for (int i = 0; i < N; i++)
        {
            bool exists;
            ref Value32 slot = ref m.GetOrAddValue(keys[i], out exists);
            if (!exists) slot = vals[i];
            s += slot.B;
        }
        sw.Stop();
        _sink += s;
        trials[rep] = ElapsedNs(sw) / N;
    }
    var (mean, ci) = MeanAndCi95(trials);
    Console.WriteLine($"{name,-28} GetOrAdd.Miss   {mean,10:F2} ± {ci:F2} ns/op");
    return new Metric { MeanNsPerOp = mean, Ci95NsPerOp = ci };
}
```

Рис В.2 Фрагмент коду бенчмарку з тестом неуспішної операції GetOrAddValue в Robin-Hood з двома масивами

В таблиці В.1 заміри часу подано як середнє значення та вибіркове стандартне відхилення; нижче також наведено відносні показники щодо еталону.

Таблиця В.1

Порівняння швидкості виконання операцій між імплементаціями хеш-карт

	Dictionary	**Robin-Hood з одним масивом**	**Robin-Hood з двома масивами**	**Швейцарська таблиця**
Вставлення нового елемента (Add), середнє значення, нс/операцію	9.73 ± 1.46	10.19 ± 1.37	10.31 ± 1.33	16.55 ± 3.29
Вставлення нового елемента (Add), відносно еталону	1.00	1.05	1.06	1.70
Успішний пошук значення, середнє значення, нс/операцію	14.57 ± 3.69	10.04 ± 2.56	9.93 ± 2.34	19.04 ± 4.10
Успішний пошук значення, відносно еталону	1.00	0.69	0.68	1.31
Неуспішний пошук значення, середнє значення, нс/операцію	17.47 ± 4.23	12.61 ± 2.79	12.24 ± 2.85	21.28 ± 5.11
Неуспішний пошук значення, відносно еталону	1.00	0.72	0.70	1.22
Успішний пошук або додавання значення (GetOrAddValue), середнє значення, нс/операцію	16.12 ± 4.08	12.08 ± 2.76	12.07 ± 2.95	23.49 ± 5.88

Продовження таблиці В.1

Порівняння швидкості виконання операцій між імплементаціями хеш-карт

	Dictionary	**Robin-Hood з одним масивом**	**Robin-Hood з двома масивами**	**Швейцарська таблиця**
Успішний пошук або додавання значення (GetOrAddValue), відносно еталону	1.00	0.75	0.75	1.46
Неуспішний пошук або додавання значення (GetOrAddValue), середнє значення, нс/операцію	20.25 ± 8.95	15.82 ± 6.61	15.57 ± 6.98	27.71 ± 11.34
Неуспішний пошук або додавання значення (GetOrAddValue), відносно еталону	1.00	0.78	0.77	1.37

ДОДАТОК Г. РЕЗУЛЬТАТИ ПОРІВНЯЛЬНОГО АНАЛІЗУ ПРОДУКТИВНОСТІ UI-БІБЛІОТЕК.

```
ulong currentThreadCycles = GetThreadCycles();

_frameDatas[_count++] = new()
{
    Delta = Time.deltaTime,
    ThreadCycleDelta = currentThreadCycles - _prevThreadCycles,
    UnityGC = _frameGC.LastValue,
};

_prevThreadCycles = currentThreadCycles;
```

Рис Г.1 Частина профайлера, відповідальна за зберігання метрик кадру

Таблиця Г.1

Режим спокою. Дані для пустої збірки наведені у абсолютний значеннях, для бібліотек — відносно до пустої збірки

	Пуста збірка	UI Toolkit	IMGUI
CV	0.059	0.020	0.011
CVaR 1% кількості циклів, тис	889	933	1121
Середня кількість циклів процесора, тис	670	627	798
Середня виділена пам’ять за цикл, байт	0	39	2698
Середній час за кадр, мс	0.25	0.18	0.22
CVaR 1% часу за кадр, мс	0.70	0.10	0.09
Максимальний час за кадр, мс	1.25	1.18	1.58
Максимальне виділення пам’яті за кадр, КБ	0	39	2698

Таблиця Г.2

Наведення на елементи

	UI Toolkit	IMGUI
CV	0.112	0.101
CVaR 1% кількості циклів, тис	1861	2037
Середня кількість циклів процесора, тис	720	855
Сумарна виділена пам’ять, КБ	69	0
Середній час за кадр, мс	0.21	0.25
CVaR 1% часу за кадр, мс	0.46	0.46
Максимальний час за кадр, мс	2.37	3.68
Максимальне виділення пам’яті за кадр, КБ	33	0

Таблиця Г.3

Розкриття горизонтального випадаючого меню

	UI Toolkit	IMGUI
CV	0.27	0.033
CVaR 1% кількості циклів, тис	2026	1391
Середня кількість циклів процесора, тис	695	854
Сумарна виділена пам’ять, КБ	1498	1198
Середній час за кадр, мс	0.2	0.24
CVaR 1% часу за кадр, мс	0.44	0.15
Максимальний час за кадр, мс	8.81	1.74
Максимальне виділення пам’яті за кадр, КБ	1479	1

Таблиця Г.4

Прокручування вертикального списку

	UI Toolkit	IMGUI
CV	0.105	0.07
CVaR 1% кількості циклів, тис	1873	1838
Середня кількість циклів процесора, тис	707	842
Сумарна виділена пам’ять, КБ	2	0
Середній час за кадр, мс	0.21	0.24
CVaR 1% часу за кадр, мс	0.83	0.47
Максимальний час за кадр, мс	2.21	1.53
Максимальне виділення пам’яті за кадр, КБ	1	0

Таблиця Г.5

Прокручування текстового контейнера у всі сторони

	UI Toolkit	IMGUI
CV	0.118	0.072
CVaR 1% кількості циклів, тис	1809	1786
Середня кількість циклів процесора, тис	705	846
Сумарна виділена пам’ять, КБ	5	0
Середній час за кадр, мс	0.2	0.24
CVaR 1% часу за кадр, мс	0.5	0.41
Максимальний час за кадр, мс	2.0	1.98
Максимальне виділення пам’яті за кадр, КБ	4	0

Таблиця Г.6

Взаємодія з повзунком з округленням

	UI Toolkit	IMGUI
CV	0.198	0.07
CVaR 1% кількості циклів, тис	2704	1749
Середня кількість циклів процесора, тис	813	855
Сумарна виділена пам’ять, КБ	30	0
Середній час за кадр, мс	0.24	0.24
CVaR 1% часу за кадр, мс	0.83	0.38
Максимальний час за кадр, мс	5.34	1.52
Максимальне виділення пам’яті за кадр, КБ	16	0

Таблиця Г.7

Розкриття вертикального випадаючого меню

	UI Toolkit	IMGUI
CV	0.5	0.068
CVaR 1% кількості циклів, тис	5020	1858
Середня кількість циклів процесора, тис	1095	881
Сумарна виділена пам’ять, КБ	394	710
Середній час за кадр, мс	0.33	0.25
CVaR 1% часу за кадр, мс	1.66	0.38
Максимальний час за кадр, мс	13.33	1.6
Максимальне виділення пам’яті за кадр, КБ	73	1

Таблиця Г.8

Зміна теми

	UI Toolkit	IMGUI
CV	0.778	0.023
CVaR 1% кількості циклів, тис	13653	1422
Середня кількість циклів процесора, тис	1318	881
Сумарна виділена пам’ять, КБ	4438	0
Середній час за кадр, мс	0.39	0.25
CVaR 1% часу за кадр, мс	3.95	0.19
Максимальний час за кадр, мс	8.48	1.59
Максимальне виділення пам’яті за кадр, КБ	1551	0

Таблиця Г.9

Комбінована взаємодія з панеллю

	UI Toolkit	IMGUI
CV	0.257	0.065
CVaR 1% кількості циклів, тис	6087	2692
Середня кількість циклів процесора, тис	1586	1564
Сумарна виділена пам’ять, КБ	2897	1795
Середній час за кадр, мс	0.47	0.46
CVaR 1% часу за кадр, мс	1.57	0.55
Максимальний час за кадр, мс	8.01	4.57
Максимальне виділення пам’яті за кадр, КБ	567	1