

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД
«ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»


Навчально-науковий інститут математики
та інформаційних технологій


Кафедра інформаційних технологій та систем

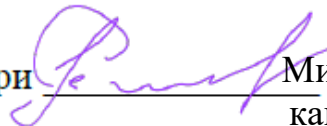
Кузнецов Василь Петрович

**ДОСЛІДЖЕННЯ ВПЛИВУ ПЕРЕДЗАВАНТАЖЕННЯ РЕСУРСІВ НА
КОРИСТУВАЦЬКИЙ ДОСВІД ДОДАТКІВ**

**кваліфікаційна робота
здобувача вищої освіти другого (магістерського) рівня
освітньої програми «Мультимедійні системи»
за спеціальністю 121 „Інженерія програмного забезпечення”**

Особистий підпис  Василь КУЗНЕЦОВ

Науковий керівник  Світлана ПЕРЕЯСЛАВСЬКА,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій та
систем

Завідувач кафедри  Микола СЕМЕНОВ,
кандидат педагогічних наук,
доцент кафедри інформаційних технологій
та систем

Лубни – 2026

АНОТАЦІЯ

Кузнецов Василь Петрович

Тема: Дослідження впливу передзавантаження ресурсів на користувацький досвід додатків.

Спеціальність: 121 «Інженерія програмного забезпечення».

Установа: ДЗ ЛНУ імені Тараса Шевченка, 2026 р.

Магістерська робота містить: 54 с., 4 рис., 2 табл., 1 додат., 26 джерел.

Об'єкт дослідження – процеси управління ресурсами та життєвим циклом активів у мобільних додатках.

Предмет дослідження – методи предиктивного завантаження даних (Predictive Prefetching) для односторінкових додатків (SPA) на основі стохастичних моделей навігації.

Мета роботи – підвищення чуйності інтерфейсу та показників утримання користувачів у мобільних SPA-додатках шляхом розробки методу предиктивного завантаження активів, що базується на аналізі імовірнісних навігаційних патернів.

Методи дослідження. *Загальнонаукові:* аналіз літератури та існуючих підходів до оптимізації (Lazy/Eager loading), синтез вимог до системи, порівняльний аналіз продуктивності. *Спеціальні:* теорія графів (для моделювання навігації), теорія ймовірностей (прогнозування переходів), метод скінченних автоматів (для управління станами), експериментальне моделювання програмного прототипу.

Результати роботи. Розроблено та експериментально перевірено метод предиктивного завантаження ресурсів. Створено спеціалізований програмний емулятор на базі Node.js, що дозволив провести серію зі 93 автоматизованих сесій. Запропонований підхід дозволяє скоротити середній час очікування контенту на **60–80%** порівняно із завантаженням на вимогу (Lazy Loading). При цьому споживання трафіку знижено на **30%** порівняно з методом повного попереднього завантаження (Eager Loading).

Ключові слова: мобільна гра, UX, передзавантаження, ланцюги Маркова, Retention Rate.

ABSTRACT

Kuznetsov Vasyl Petrovych

Topic: Research on the Impact of Resource Preloading on the User Experience of Applications.

Specialty:121 “Software Engineering”.

Institution: Taras Shevchenko Luhansk National University, 2026.

Master’s thesis contains: 54 pages, 4 figures, 2 tables, 1 appendices, 26 references.

Object of research: Resource management processes and asset lifecycle in mobile applications.

Subject of research: Methods of predictive data loading based on stochastic navigation models for user experience optimization.

Purpose of the work: increasing interface responsiveness and user retention rates in mobile SPA applications by developing a predictive asset loading method based on the analysis of probabilistic navigation patterns.

Research methods. General scientific methods include analysis of literature and existing optimization approaches (Lazy/Eager loading), synthesis of system requirements, and comparative performance analysis. Special methods include graph theory (for navigation modeling), probability theory (for transition prediction), finite state machine methods (for state management), and experimental modeling of a software prototype.

Results of the work. A method of predictive resource preloading was developed and experimentally verified. A specialized software emulator based on Node.js was created, which allowed conducting a series of 93 automated sessions. The proposed approach allows for a reduction in average content waiting time by 60–80% compared to on-demand loading (Lazy Loading). At the same time, traffic consumption is reduced by 30% compared to the full preloading method (Eager Loading).

Keywords: mobile game, UX, preloading, Markov chains, Retention Rate.

ЗМІСТ

ВСТУП	8
РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ ПРОДУКТИВНОСТІ ТА УТРИМАННЯ КОРИСТУВАЧІВ У МОБІЛЬНИХ ІГРАХ.....	12
1.1. Економічний контекст та динаміка ринку мобільних розваг 2025 року	12
1.2. Латентність як фактор відтоку: психофізіологічні аспекти.....	13
1.3. Обмеження традиційних стратегій управління ігровими ресурсами	17
Висновки до розділу 1	19
РОЗДІЛ 2. РОЗРОБКА МЕТОДУ ТА АРХІТЕКТУРИ СИСТЕМИ ПРЕДИКТИВНОГО ЗАВАНТАЖЕННЯ	22
2.1. Теоретичні основи моделювання навігації: Графи та Марковські процеси	22
2.2. Формалізація задачі предиктивного завантаження в термінах теорії графів	23
Висновки до розділу 2	26
РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕХАНІЗМУ ПРЕДИКТИВНОГО ЗАВАНТАЖЕННЯ ТА СЕРЕДОВИЩЕ ОЦІНКИ ЕФЕКТИВНОСТІ РІШЕННЯ	27
3.1. Ключові елементи оцінки ефективності рішень передзавантаження активів та мета створення середовища	27
3.2. Архітектура програмного прототипу та обґрунтування технологічного стеку	29
3.3. Реалізація інтерфейсу користувача та візуалізація графа станів	32
3.4. Побудова імовірнісної моделі навігації на основі статистичних даних.	35
3.5. Програмна реалізація механізмів управління ресурсами	36
Висновки до розділу 3	39
РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ.....	40

4.1. Методика проведення експерименту та конфігурація тестового середовища	40
4.2. Аналіз отриманих результатів: порівняння метрик продуктивності.....	41
4.3. Оцінка компромісу між використанням трафіку та швидкістю (Trade-off analysis).....	43
Висновки до розділу 4.....	44
ВИСНОВКИ	46
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	49
ДОДАТКИ	53

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

API	—	Application Programming Interface
CDN	—	Content Delivery Network
CHR	—	Cache Hit Ratio
CPI	—	Cost Per Install
CPU	—	Central Processing Unit
DAG	—	Directed Acyclic Graph
DNS	—	Domain Name System
F2P	—	Free-to-Play
FSM	—	Finite State Machine
HTTP	—	HyperText Transfer Protocol
I/O	—	Input/Output
JSON	—	JavaScript Object Notation
KPI	—	Key Performance Indicator
LCP	—	Largest Contentful Paint
LTE	—	Long-Term Evolution
OPFS	—	Origin Private File System
QUIC	—	Quick UDP Internet Connections
R&D	—	Research and Development
RPG	—	Role –Playing Game
RTT	—	Round Trip Time
SoC	—	System on a Chip
SPA	—	Single Page Application
SSL	—	Secure Sockets Layer

TCP	–	Transmission Control Protocol
TLS	–	Transport Layer Security
TTL	–	Time To Live
UAC	–	User Acquisition Cost
UI	–	User Interface
UX	–	User Experience
XML	–	Extensible Markup Language

ВСТУП

Актуальність теми. Сучасна індустрія мобільних розваг та додатків перебуває на етапі безпрецедентної конкуренції та глибокої технологічної трансформації. Станом на 2025 рік мобільний геймінг залишається домінуючим сектором цифрової економіки з прогнозованим зростанням доходів до 103 мільярдів доларів США [1]. Однак, епоха екстенсивного зростання завершилася: ринок насичений, а боротьба за увагу користувача стає все більш виснажливою та дороговартісною. Розробники стикаються з критичними викликами, головним з яких є не стільки залучення, скільки утримання аудиторії (Retention).

Економічна доцільність утримання користувачів диктується стрімким зростанням витрат на маркетинг. Вартість залучення одного платоспроможного користувача (User Acquisition Cost — UAC) за останні роки зросла на 60%, досягнувши в середньому 29 доларів США [2]. У таких умовах здатність додатку утримувати гравця стає фундаментом бізнес-моделі. Проте статистика є невблаганною: показники утримання першого дня (D1 Retention) для більшості додатків стабілізувалися на рівні 26–28%, а бар'єр довгострокового утримання (D28) долають лише 3% проєктів [3].

Одним із ключових технічних факторів, що провокують відтік користувачів (Churn Rate), є латентність завантаження контенту. [24] Психофізіологічні дослідження сприйняття часу свідчать, що затримки понад 1 секунду порушують когнітивний потік користувача, а паузи тривалістю понад 4 секунди призводять до різкого зростання кількості відмов (abandonment rate) на 25–30% [4]. В умовах мобільних мереж, де швидкість з'єднання є нестабільною, традиційні підходи до управління ресурсами демонструють свою неефективність. Метод повного попереднього завантаження (Eager Loading) створює неприйнятний бар'єр входу

через довгий «холодний старт» та надмірне споживання трафіку, що є критичним для ринків, де ціна за гігабайт залишається високою. Натомість метод завантаження на вимогу (Lazy Loading) переносить затримки безпосередньо в ігровий процес, руйнуючи імерсивність.

Вирішенням цієї дилеми є впровадження інтелектуальних систем предиктивного передзавантаження (Predictive Prefetching). Використання стохастичних моделей, зокрема ланцюгів Маркова, дозволяє системі прогнозувати майбутні дії користувача та завантажувати необхідні активи у моменти пасивної активності («час роздумів»). Розробка та валідація такого методу, який би балансував між миттєвою реакцією інтерфейсу та економією трафіку, зумовлює актуальність даного магістерського дослідження.

Мета і завдання дослідження. Метою роботи є підвищення чуйності інтерфейсу та показників утримання користувачів у мобільних SPA-додатках шляхом розробки методу предиктивного завантаження активів, що базується на аналізі імовірнісних навігаційних патернів.

Для досягнення мети поставлено такі завдання:

1. Проаналізувати сучасний стан ринку мобільних додатків (станом на 2025 рік), дослідити вплив латентності на ключові продуктові метрики (Retention Rate, Churn Rate) та виявити недоліки існуючих патернів завантаження (Lazy/Eager Loading).
2. Розробити математичну модель навігації користувача в односторінкових додатках (SPA) на основі зваженого орієнтованого графа та застосувати апарат ланцюгів Маркова для прогнозування переходів.
3. Спроекувати архітектуру клієнтської системи (модулі GraphManager, AssetsManager, PreloadManager), що підтримує фонове завантаження з урахуванням лімітів конкурентності та пріоритетів.
4. Створити спеціалізований інструментарій для автоматизованого тестування — програмний емулятор на базі Node.js, здатний симулювати

реалістичну поведінку користувача (включно з часом на «роздуми» та стохастичним вибором шляху).

5. Провести експериментальне дослідження (серію зі 100 автоматизованих сесій для кожного режиму), порівняти ефективність запропонованого методу з традиційними підходами за критеріями часу очікування та споживання трафіку.

Об’єкт дослідження: процеси управління ресурсами, життєвим циклом активів та обміну даними у клієнтських мобільних додатках.

Предмет дослідження: методи предиктивного кешування та передзавантаження контенту (Predictive Prefetching) для SPA-додатків на основі стохастичних моделей навігації.

Методи дослідження. У роботі використано комплексний методологічний підхід:

- *загальнонаукові методи:* аналіз та синтез (для вивчення підходів до оптимізації UX), порівняльний аналіз (для оцінки ефективності стратегій завантаження);
- *математичні методи:* теорія графів (для моделювання топовлогії додатку), теорія ймовірностей та випадкових процесів (ланцюги Маркова для прогнозування переходів);
- *інженерні методи:* об’єктно-орієнтоване проєктування (архітектура системи на TypeScript), імітаційне моделювання (створення Node.js емулятора);
- *емпіричні методи:* автоматизований експеримент із збором та статистичною обробкою логів телеметрії.

Наукова новизна одержаних результатів:

1. Удосконалено метод управління завантаженням активів у мобільних додатках шляхом інтеграції динамічної матриці ймовірностей переходів,

що, на відміну від статичних черг, дозволяє адаптувати стратегію кешування під конкретні патерни поведінки гравців.

2. Набула подальшого розвитку модель подання навігаційної структури SPA-додатків у вигляді зваженого орієнтованого графа, доповненого атрибутами ваги ресурсів та ймовірності ребер.
3. Вперше застосовано методологію автоматизованої емуляції UX на базі середовища Node.js із симуляцією «часу роздумів» (Think Time), що дозволяє проводити об'єктивну кількісну оцінку стратегій завантаження без залучення реальних користувачів на етапі розробки.

Практичне значення одержаних результатів. Результати дослідження підтверджують, що запропонований метод дозволяє скоротити середній час очікування контенту на **60–80%** порівняно із завантаженням на вимогу, при цьому заощаджуючи до **30%** трафіку порівняно з повним завантаженням. Розроблений програмний модуль PreloadManager та супровідний емулятор (SessionBatchRunner) можуть бути інтегровані у виробничі процеси студій мобільної розробки (на базі React, React Native, Unity) для профілювання продуктивності та оптимізації показників Retention Rate, що дозволить знизити фінансові ризики втрати аудиторії.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ ПРОДУКТИВНОСТІ ТА УТРИМАННЯ КОРИСТУВАЧІВ У МОБІЛЬНИХ ІГРАХ

1.1. Економічний контекст та динаміка ринку мобільних розваг 2025 року

Станом на 2025 рік індустрія мобільних ігор залишається домінуючим сектором цифрової економіки, проте характер її розвитку зазнав фундаментальних змін. Період екстенсивного зростання, що базувався на органічному трафіку та низькій вартості залучення, завершився. Аналіз ринку свідчить про насичення: хоча прогнозовані доходи сягають 103 мільярдів доларів США, глобальна кількість завантажень у 2024 році продемонструвала спад на 7%, зупинившись на позначці 49 мільярдів [1].

Ключовим викликом для розробників стає різке зростання вартості маркетингу. Витрати на залучення одного платоспроможного користувача (User Acquisition Cost — UAC) за останні п'ять років зросли на 60%, досягнувши в середньому 29 доларів США [2]. У висококонкурентних нішах, таких як стратегії або RPG, ціна за одну інсталяцію (CPI) на платформі iOS може перевищувати 6 доларів [3]. У такій економічній парадигмі здатність додатку утримувати аудиторію (Retention) стає не просто показником якості, а умовою рентабельності бізнесу.

Статистика утримання користувачів є критичною. Показники Retention Rate демонструють тенденцію до зниження:

- Day 1 (D1): Середній показник для iOS становить 35.7%, для Android — 27.5%. Лише найкращі проєкти (Топ 25%) утримують понад 28% аудиторії [5].

- Day 7 (D7): Показники різко падають до 7–8%, а для аутсайдерів ринку — до 1.5%.
- Day 28 (D28): Довгострокове утримання є найбільючішою точкою: 75% ігор не можуть подолати бар'єр у 3% [3].

Ці дані підкреслюють, що "вікно можливостей" для зацікавлення користувача вимірюється першими хвилинами, а іноді й секундами сесії. Втрата гравця на етапі першого знайомства через технічні недоліки фактично означає прямі фінансові збитки, які неможливо компенсувати подальшою монетизацією.

1.2. Латентність як фактор відтоку: психофізіологічні аспекти

Технічна продуктивність додатку, зокрема швидкість завантаження контенту (Load Time) та чуйність інтерфейсу (Responsiveness), має пряму кореляцію з показниками відтоку (Churn Rate). Дослідження компанії Zigpoll виявило сильну негативну кореляцію ($r = -0.72$) між часом очікування та утриманням першого дня [4].

Психологія сприйняття часу користувачем у цифровому середовищі базується на жорстких порогових значеннях [6]:

1. < 0.1 с (Миттєвість): Сприймається як безпосередня реакція системи. Користувач відчуває, що фізично маніпулює об'єктами на екрані.
2. $0.1 - 1.0$ с (Потік): Затримка помітна, але потік думок (cognitive flow) не переривається. [20]
3. > 1.0 с (Втрата контролю): Користувач відчуває розрив між своєю дією та реакцією системи. Починається втрата імерсивності.
4. > 10 с (Втрата уваги): Ймовірність перемикання на інший додаток стає критичною.

У сучасних мобільних іграх ця проблема загострюється через зростання "ваги" контенту. [14] Текстури високої роздільної здатності, складні 3D-моделі та аудіо супровід значно збільшують обсяг даних. [18-19] Виникає конфлікт: розробники прагнуть надати візуально багатий досвід, але затримки при завантаженні цього досвіду призводять до зростання показника відмов (Bounce Rate) на 32% вже при затримці у 3 секунди [5]. Латентність діє як "тихий вбивця": користувачі рідко скаржаться на повільне завантаження, вони просто видаляють додаток.

1.3 Основні фактори які роблять внесок у час завантаження

Забезпечення миттєвої чуйності інтерфейсу (Responsiveness) та мінімізація часу очікування є багатофакторною інженерною задачею, вирішення якої вимагає чіткої ідентифікації «вузьких місць» (bottlenecks) у пайплайні доставки контенту. У контексті архітектури сучасних мобільних ігор та SPA-додатків, сумарний час затримки () можна представити як сукупність витрат часу на мережеву взаємодію, обчислювальні процеси та трансфер даних. Детальний аналіз дозволяє виділити три ключові компоненти, що роблять визначальний внесок у формування затримок: накладні витрати на виконання HTTP-запитів, процесинг програмної логіки та завантаження «важких» медіа-ресурсів. [13]

1. Мережева взаємодія та службовий трафік (Network Overhead) Першою складовою є латентність, пов'язана з ініціалізацією з'єднань та обміном службовими даними. Сюди відносяться витрати часу на DNS-резолвінг, встановлення TCP-з'єднання (Handshake) та узгодження шифрування TLS/SSL. Хоча обсяг даних, що передаються під час API-запитів (JSON, XML), є відносно незначним, ключовим обмежуючим фактором виступає показник RTT (Round Trip Time). В умовах мобільних мереж, де стабільність сигналу є змінною величиною, навіть запити з мінімальним пейлоадам можуть створювати відчутні

паузи через фізичні обмеження розповсюдження сигналу та архітектуру стільникових мереж.

2. Обчислювальна складність та обробка логіки (CPU Processing) Другий фактор охоплює час, необхідний центральному процесору (CPU) пристрою для парсингу отриманих даних, виконання скриптів (JavaScript/C#) та опрацювання ігрової логіки. Варто зазначити, що завдяки дії закону Мура та стрімкому розвитку мобільних SoC (System-on-Chip), продуктивність сучасних смартфонів зростає швидше, ніж складність базової логіки клієнтських додатків. Відповідно, за умови дотримання стандартів чистого коду та відсутності блокуючих операцій у головному потоці (Main Thread), процесинг рідко стає основною причиною критичних затримок завантаження сцени.

3. Трансфер та ініціалізація активів (Resource Loading) Найбільш вагомим фактором, що критично впливає на UX, є завантаження медіа-контенту: текстур високої роздільної здатності, 3D-моделей, шейдерів та аудіофайлів. На відміну від програмного коду, обсяг активів зростає експоненціально у гонитві за візуальною якістю (Retina-ready зображення, 4K текстури), тоді як пропускна здатність мобільних каналів зв'язку часто не встигає за цим зростанням. Саме цей процес характеризується передачею значних обсягів даних, що у комбінації з блокуючим характером завантаження створює ефект «заморожування» інтерфейсу або тривалих екранів завантаження.

Для візуалізації впливу зазначених факторів на продуктивність системи розроблено порівняльну характеристику (Таблиця 1.1).

Таблиця 1.1 — Порівняльний аналіз факторів, що впливають на час завантаження мобільних додатків

Фактор затримки	Природа обмеження	Залежність від апаратного забезпечення	Вплив на обсяг даних	Динаміка впливу на UX (Trend)
HTTP Запити (Network Overhead)	Латентність мережі (RTT), кількість запитів	Низька (залежить від якості мережі)	Мінімальний (KB)	Стабільна. Вплив залишається помітним через фізичні обмеження мереж, але нівелюється протоколами HTTP/2 та QUIC.
Обробка логіки (Code Processing)	Продуктивність CPU, оптимізація коду	Висока (CPU Speed)	Середній (MB - код, скрипти)	Спадна. Зростання потужності процесорів випереджає ускладнення ігрової логіки.
Завантаження ресурсів (Asset Loading)	Пропускна здатність каналу (Bandwidth), I/O диска	Середня (швидкість пам'яті/диска)	Критичний (десятки/сотні MB)	Зростаюча. Збільшення деталізації контенту випереджає ріст швидкості мобільного інтернету.

Аналіз наведених факторів дозволяє стверджувати, що в сучасних реаліях мобільного геймінгу домінуючу роль у формуванні негативного користувацького досвіду відіграє саме підсистема завантаження ресурсів. Оскільки мобільні пристрої функціонують у середовищі з нестабільним інтернет-з'єднанням, а очікування користувачів щодо візуальної якості постійно зростають, традиційні методи завантаження стають неефективними. Отже, оптимізація саме механізмів доставки активів, а не мікро-оптимізація коду чи скорочення кількості службових запитів, має найвищий потенціал для покращення метрик Responsiveness та, як наслідок, підвищення показників Retention Rate у довгостроковій перспективі.

1.4. Обмеження традиційних стратегій управління ігровими ресурсами

Для вирішення проблеми доставки контенту індустрія традиційно використовує два полярних підходи, кожен з яких в умовах 2025 року демонструє суттєві вади.

1. Eager Loading (Повне попереднє завантаження)

Передбачає завантаження всіх ресурсів під час "холодного старту" (Cold Start) додатку.

- *Переваги:* Забезпечує ідеальний ігровий процес без пауз після завантаження.
- *Недоліки:* Створює масивний бар'єр на вході. Користувач змушений чекати хвилини перед тим, як побачити меню. Це призводить до марнотратства трафіку: якщо гравець завантажив 500 МБ, але покинув гру на першому рівні, 90% трафіку було витрачено даремно. Це критично для ринків, що розвиваються, де вартість мобільних даних є чутливою [7].

2. Lazy Loading (Ліниве завантаження / Завантаження на вимогу)

Ресурси ініціалізуються лише в момент звернення до них (Runtime).

- *Переваги:* Мінімальний час старту та економія пам'яті.
- *Недоліки:* Переносить затримку безпосередньо в ігровий процес. Поява спінерів або екранів завантаження між рівнями руйнує ефект присутності. Дослідження показують, що переривання ігрового процесу є більш дратівливим фактором, ніж довгий перший запуск [8].

Існуючі "розумні" рішення, такі як AppStreamer або веб-орієнтовані алгоритми Cloudflare (Speed Kit), намагаються використовувати евристики для передбачення [9]. Однак, вони часто не враховують нелінійну природу ігрової навігації [10], де вибір користувача залежить від ігрового контексту, а не лише від структури посилань.

Ключовим компромісом будь-якої стратегії є компроміс «швидкість/трафік»: зменшення часу очікування часто досягається за рахунок додаткових завантажень (over-fetching). Тому для оцінки застосовуються метрики Cache Hit Ratio (CHR), середній час очікування, а також сумарний трафік і його «надлишок».

Висновки до розділу 1

Узагальнюючи результати аналізу стану індустрії мобільних розваг та технічних аспектів розробки додатків станом на 2025 рік, можна зробити наступні висновки:

1. **Економічна необхідність технічної оптимізації.** Сучасний ринок мобільних додатків перейшов від стадії екстенсивного зростання до стадії жорсткої боротьби за утримання (Retention). В умовах, коли вартість залучення користувача (UAC) досягла в середньому 29 доларів США, а показники довгострокового утримання (D28) для більшості проєктів не перевищують 3%, технічна якість продукту стає фундаментальним фактором економічної виживаності. Встановлено, що існує пряма кореляція між продуктивністю додатку та ключовими бізнес-метриками: латентність є «тихим вбивцею» конверсії.
2. **Критичність часових порогів.** Аналіз психофізіологічних аспектів сприйняття часу показав, що користувачі мають жорсткі вимоги до чуйності інтерфейсу. Затримки, що перевищують 1 секунду, порушують когнітивний потік та руйнують імерсивність ігрового процесу. Систематичне порушення цього порогу призводить до різкого зростання показників відтоку (Churn Rate), нівелюючи інвестиції в маркетинг та контент.
3. **Домінування фактора завантаження ресурсів.** Порівняльний аналіз технічних обмежень (CPU, Network, I/O) дозволяє стверджувати, що в сучасних реаліях мобільного геймінгу «вузьким місцем» є підсистема завантаження активів. Зростання візуальної якості контенту випереджає можливості мобільних мереж, що робить традиційні методи завантаження неефективними.

4. **Пріоритет оптимізації.** Доведено, що саме оптимізація механізмів доставки активів, а не мікро-оптимізація програмного коду чи скорочення службових запитів, має найвищий потенціал для покращення метрик чуйності (Responsiveness) та підвищення Retention Rate у довгостроковій перспективі.
5. **Неефективність традиційних патернів завантаження.** Існуючі полярні підходи до управління активами демонструють суттєві обмеження в сучасних умовах:
- **Lazy Loading (Завантаження на вимогу)** вирішує проблему швидкого старту, але переносить технічні затримки безпосередньо в процес активної взаємодії, що негативно впливає на UX. Це призводить до появи дратівливих візуальних індикаторів очікування або проміжних екранів завантаження при кожній зміні ігрового контексту. Така фрагментація сесії руйнує ефект присутності та перериває стан когнітивного «потoku», який є критично важливим для глибокого занурення гравця. Навіть незначні мікро-затримки, накопичуючись протягом тривалої сесії, формують негативний сумарний досвід, що суттєво знижує загальну задоволеність продуктом.
 - **Eager Loading (Повне завантаження)** гарантує плавність процесу, але створює неприйнятний бар'єр входу («холодний старт») та призводить до нераціонального витрачання трафіку, що є критичним для глобального ринку. Визначено потребу в пошуку гібридного методу, який би поєднував швидкість старту Lazy Loading із плавністю Eager Loading.
6. **Обґрунтування вибору математичного апарату.** Для реалізації такого гібридного підходу (Predictive Prefetching) необхідна формалізація навігаційної структури та поведінки користувача.

- Представлення SPA-додатку у вигляді **зваженого орієнтованого графа [11]** дозволяє чітко структурувати залежності між екранами та оцінити «вагу» необхідних ресурсів.
- Використання **ланцюгів Маркова першого порядку** визначено як оптимальний інструмент для стохастичного моделювання переходів. Оскільки дії гравця не є хаотичними, а підпорядковані певним патернам, матриця перехідних ймовірностей дозволяє системі з високою точністю прогнозувати майбутній стан і використовувати час простою (idle time) для фонового завантаження контенту.

Таким чином, у першому розділі сформовано теоретичний та економічний фундамент дослідження. Обґрунтовано доцільність розробки адаптивної системи предиктивного завантаження, яка базується на аналізі імовірнісних сценаріїв навігації, що стане предметом подальшої розробки у наступних розділах роботи.

РОЗДІЛ 2. РОЗРОБКА МЕТОДУ ТА АРХІТЕКТУРИ СИСТЕМИ ПРЕДИКТИВНОГО ЗАВАНТАЖЕННЯ

2.1. Теоретичні основи моделювання навігації: Графи та Марковські процеси

Для побудови системи, яка б поєднувала переваги швидкого старту (Lazy) та миттєвого доступу (Eager), необхідно формалізувати поведінку користувача математично.

Графове моделювання SPA. Архітектуру односторінкового додатку (SPA) або гри можна представити як зважений орієнтований граф

$$G = (V, E)$$

- V (вершини) — множина станів інтерфейсу (Main Menu, Shop, Level 1).
- E (ребра) — множина можливих переходів між ними.
- Кожна вершина v в V має вагу $w(v)$, що відповідає розміру ресурсу (у байтах), необхідних для відображення цього стану.

Стохастичне прогнозування (Ланцюги Маркова) [12]. На відміну від хаотичного руху, поведінка гравця підпорядкована патернам. Для прогнозування наступного кроку доцільно використовувати ланцюги Маркова першого порядку. Основна властивість (марковська властивість) стверджує, що ймовірність переходу до майбутнього стану X_{n+1} залежить виключно від поточного стану X_n :
$$P(X_{n+1} = x \mid X_n = x_n, \dots, X_0 = x_0) = P(X_{n+1} = x \mid X_n = x_n)$$

Система описується матрицею перехідних ймовірностей P , де елемент P_{ij} визначає ймовірність переходу користувача з екрану i на екран j :

$$P = \begin{pmatrix} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \dots & p_{nn} \end{pmatrix}$$

Така модель дозволяє системі "випереджати" дії користувача. Наприклад, якщо статистика показує, що з екрану "Level Complete" 70% гравців переходять до "Next Level", а 30% — до "Main Menu", система може використати час, поки гравець переглядає результати (Idle Time), для фонового завантаження активів наступного рівня. Це дозволяє нівелювати затримки без надмірного споживання трафіку, завантажуючи лише найбільш вірогідні гілки графа.

2.2. Формалізація задачі предиктивного завантаження в термінах теорії графів

Задачу оптимізації завантаження можна сформулювати як задачу пошуку підмножини активів $A_{\text{subset}} \subseteq A_{\text{total}}$, завантаження яких у момент часу t максимізує ймовірність "потрапляння в кеш" (Cache Hit) [15] у момент $t + 1$ при обмеженні на обсяг завантажуваних даних.

Нехай $G = (V, E)$ — граф навігації додатку. Кожній вершині $v \in V$ відповідає набір активів $\text{Assets}(v)$ обсягом $\text{Size}(v)$.

Для поточного стану u (currentScreen) існує множина суміжних вершин $N(u) = \{v \in V \mid (u, v) \in E\}$. Кожному переходу (u, v) приписана ймовірність $P(v|u)$, отримана з матриці переходів.

Алгоритм предиктивного завантаження повинен:

1. Визначити множину кандидатів $C = N(u)$.
2. Відсортувати C за спаданням $P(v|u)$.
3. Обрати підмножину $L \subseteq C$ таку, що $\forall v \in L: P(v|u) > T_{\text{threshold}}$, де $T_{\text{threshold}}$ — мінімальний поріг ймовірності (наприклад, 0.15 або 15%).

4. Ініціювати завантаження $\text{Assets}(v)$ для всіх $v \in L$, які ще не знаходяться в локальному кеші.

Такий підхід дозволяє відсіяти малоймовірні переходи, економлячи трафік, і зосередитись на найбільш вірогідних сценаріях.⁴

На основі розробленої математичної моделі (п. 2.2) пропонується евристичний алгоритм прийняття рішень щодо попереднього завантаження. Його головна мета — трансформувати ймовірнісні характеристики графа навігації у бінарні команди для менеджера ресурсів (завантажувати / не завантажувати).

Ключовою особливістю алгоритму є введення функції корисності, яка зважає ймовірність переходу проти «вартості» завантаження активу (витрати трафіку та часу). У спрощеному вигляді, де пріоритетом є лише мінімізація часу очікування при обмеженому каналі зв'язку, алгоритм фокусується на найбільш вірогідних гілках ланцюга Маркова. [16-17]

Формальний опис процедури вибору активів для предиктивного кешування представлено у вигляді псевдокоду (Алгоритм 2.1).

Алгоритм 2.1. Вибір активів для предиктивного завантаження

Вхідні дані:

- — поточний вузол (Current State);
- — граф навігації з матрицею ймовірностей;
- — множина ідентифікаторів вже завантажених активів;
- — поріг ймовірності (наприклад, 0.15);
- — ліміт одночасних потоків завантаження (наприклад, 3).

Вихідні дані:

- — пріоритетна черга активів на завантаження.

Function $\text{GetPredictiveQueue}(u, G, \text{Cache}, T_threshold, K_limit)$:

1. Ініціалізувати порожній список кандидатів $\text{Candidates} = []$
2. Отримати множину суміжних вершин $N(u)$ з графа G

3. Для кожного суміжного вузла $v \in N(u)$:
 - a. Отримати ймовірність переходу $p = P(v|u)$
 - b. Якщо $p \geq T_threshold$:
 - i. Отримати список активів вузла $A_v = Assets(v)$
 - ii. Для кожного активу $a \in A_v$:
 - Якщо a НЕ належить Cache:
Додати кортеж (a, p) до Candidates
4. Якщо Candidates порожній:
Повернути порожній список (нічого завантажувати)
5. Відсортувати Candidates за спаданням ймовірності p
6. Вибрати перші K_limit елементів зі списку $\rightarrow Q_load$
7. Повернути Q_load

End Function

Аналіз ефективності алгоритму. Запропонований підхід дозволяє суттєво оптимізувати використання мережевих ресурсів порівняно з традиційними методами:

1. На відміну від Eager Loading (повного завантаження): Алгоритм відсікає гілки графа з низькою ймовірністю відвідування (), що запобігає завантаженню надлишкового контенту, який користувач може ніколи не побачити.
2. На відміну від Lazy Loading (завантаження на вимогу): Алгоритм використовує час перебування користувача на поточному екрані () для фонового завантаження найбільш вірогідних наступних активів. При коректному налаштуванні порогу , ймовірність Cache Hit прямує до максимуму.

Таким чином, використання ланцюгів Маркова дозволяє перейти від реактивної моделі завантаження до проактивної, де система «випереджає» дії користувача.

[21]

Висновки до розділу 2

У даному розділі було проведено формалізацію задачі мінімізації латентності в SPA-додатках. Обґрунтовано доцільність використання зважених орієнтованих графів для моделювання навігації та апарату ланцюгів Маркова для прогнозування поведінки користувача. Розроблений узагальнений алгоритм предиктивного вибору активів є теоретичним підґрунтям для створення програмної системи. Практична реалізація даного алгоритму, вибір технологічного стеку, архітектура програмних модулів та результати експериментального дослідження ефективності будуть детально розглянуті у наступному розділі.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕХАНІЗМУ ПРЕДИКТИВНОГО ЗАВАНТАЖЕННЯ ТА СЕРЕДОВИЩЕ ОЦІНКИ ЕФЕКТИВНОСТІ РІШЕННЯ

3.1. Ключові елементи оцінки ефективності рішень передзавантаження активів та мета створення середовища

Для верифікації теоретичних моделей, запропонованих у другому розділі, та отримання емпіричних даних виникає необхідність у створенні спеціалізованого програмного середовища. Головною метою розробки є створення контрольованого експериментального полігону, що дозволяє ізолювати та виміряти вплив різних стратегій управління ресурсами на користувацький досвід (UX) та споживання трафіку в умовах, наближених до реальних.

Для забезпечення об'єктивності дослідження, розроблене середовище повинно задовольняти наступні функціональні вимоги:

1. Емуляція ігрового процесу та топології. Система має відтворювати поведінку реального SPA-додатку, де перехід між екранами (станами) супроводжується завантаженням «важких» активів. Граф навігації (nodes.json) визначає допустимі переходи, а атрибути вузлів — обсяг даних, необхідних для відображення сцени.
2. Підтримка порівняльних режимів роботи. Для коректного А/В тестування середовище повинно підтримувати три ізольовані режими завантаження:
 - On-Demand (Реактивний): Базовий сценарій, де завантаження відбувається виключно після ініціації переходу користувачем (блокуюче очікування).
 - Predictive (Проактивний): Експериментальний режим, що реалізує запропонований алгоритм на основі ланцюгів Маркова [22-23],

виконуючи фонове завантаження найбільш вірогідних наступних сцен.

- Predownload All (Екстенсивний): Сценарій повного завантаження всіх активів одразу після запуску додатку (стрес-тест для каналу зв'язку).

3. Комплексна система телеметрії. Ключовим аспектом є збір метрик (KPI) у реальному часі з можливістю подальшого експорту для статистичного аналізу:

- Average Load Time (Latency): Середній час очікування користувача при переході між екранами.
- Total Traffic (Bandwidth Usage): Сумарний обсяг завантажених даних, що дозволяє оцінити економічну ефективність методу.
- Cache Hit / Cache Miss Ratio: Відсоткове співвідношення переходів, для яких контент вже був підготовлений заздалегідь.
- Total Simulation Time: Загальний час проходження сесії (використовується для логування під час запуску багатьох симуляцій).

4. Масштабованість експерименту (Batch Emulation). Можливість автоматизованого запуску тисяч симуляцій поведінки різних типів користувачів (bot-run) без візуального відображення, що дозволяє нівелювати статистичні похибки та отримати репрезентативну вибірку даних.

3.2. Архітектура програмного прототипу та обґрунтування технологічного стеку

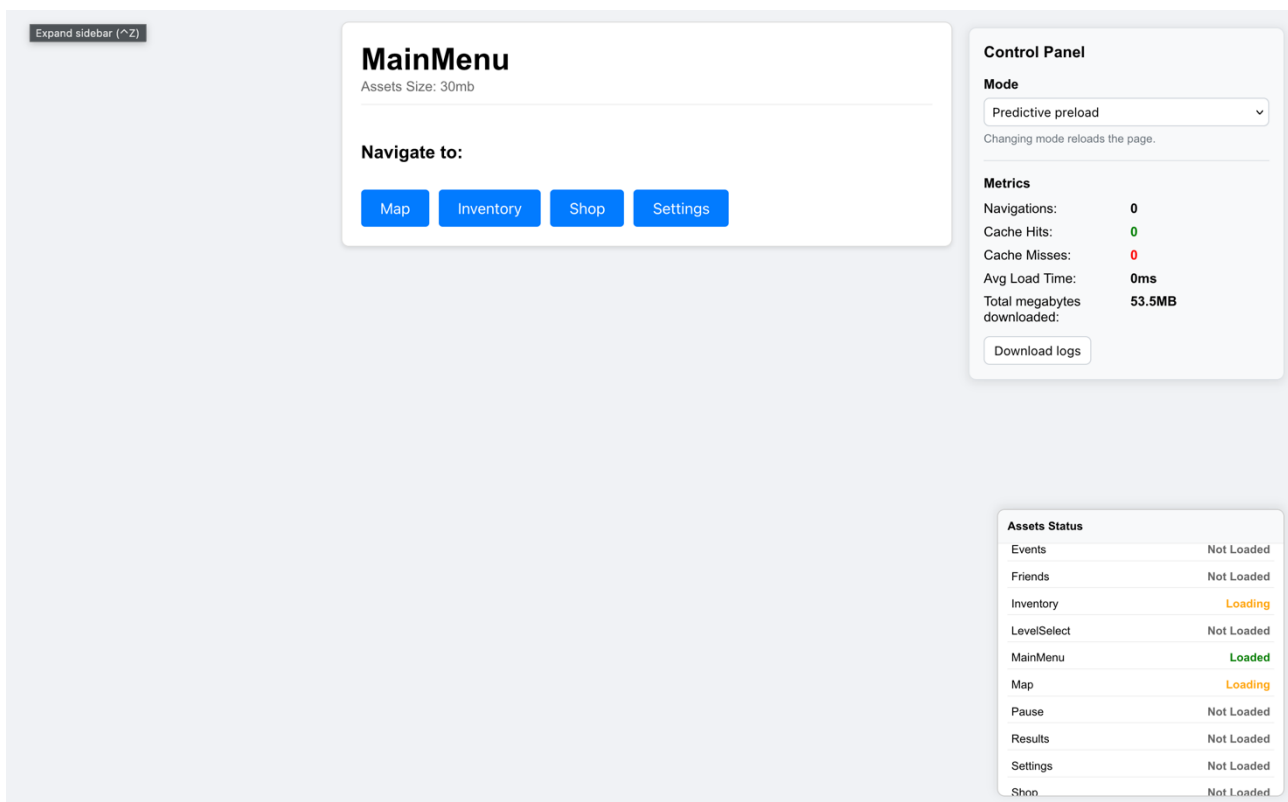


Рис 3.1 — Огляд візуальної частини програмного прототипу

Для реалізації поставлених завдань було розроблено прототип Single Page Application (SPA). Архітектурні рішення та вибір технологічного стеку обумовлені необхідністю поєднання гнучкості розробки візуального інтерфейсу з можливістю виконання того ж коду в середовищі серверної емуляції (Node.js).

3.2.1. Обґрунтування технологічного стеку

Вибір інструментарію базується на наступних критеріях:

- **React:** Використано для побудови компонентної моделі інтерфейсу, де кожен ігровий екран логічно відповідає вузлу графа навігації. Це дозволяє абстрагувати логіку відображення від логіки даних.

- TypeScript: Застосування суворої типізації є критичним для роботи зі складними структурами даних (графи переходів, конфігураційні файли ймовірностей), мінімізуючи помилки під час реалізації алгоритмів маршрутизації.
- Zustand: Легковаговий менеджер станів (State Management). На відміну від Redux, він дозволяє створювати децентралізовані сховища даних (stores) без надлишкового шаблонного коду (boilerplate). Це ідеально підходить для управління глобальними змінними, такими як поточний екран (currentScreen), метрики сесії (metrics) та прогрес завантаження (loadingProgress).

Такий підхід дозволив створити універсальне ядро логіки (Logic Core), яке використовується як у браузерній версії для ручного тестування, так і в скриптах масової емуляції (runBatch.ts), забезпечуючи ідентичність алгоритмів у обох середовищах.

3.2.2. Архітектурні шари системи

Система спроектована за модульним принципом (Clean Architecture) і складається з чотирьох основних шарів, кожен з яких має чітко визначену зону відповідальності (див. Рис. 3.1):

1. Шар представлення (Presentation Layer / UI) Відповідає за візуалізацію стану системи та взаємодію з користувачем:

- App.tsx: Точка входу в додаток, що ініціалізує глобальні сервіси.
- Screen.tsx: Основний компонент, що відображає поточний вузол графа та генерує кнопки навігації на основі доступних переходів.
- ControlPanel.tsx: Інструментарій дослідника, що дозволяє перемикати режими (preloadMode), спостерігати за метриками в реальному часі та експортувати логи.
- AssetStatusPanel.tsx: Компонент налагодження, що візуалізує статус кожного активу в системі (стани: *Not Loaded*, *Loading*, *Loaded*).

2. Шар бізнес-логіки (Logic Core) Центральний елемент системи, що реалізує алгоритми управління даними:

- GraphManager: Відповідає за завантаження топології графа з файлу nodes.json та надання інтерфейсу для валідації переходів (isValidTransition).
- PreloadManager: Імплементация розробленого методу предиктивного завантаження. Цей модуль підписується на події навігації, аналізує матрицю ймовірностей та ініціює фонове завантаження кандидатів через AssetsManager.
- AssetsManager: Емулятор мережевого стеку. Він імітує затримки завантаження на основі розміру файлу, керує кешем (структура Set<string>) та, що важливо, виконує дедуплікацію запитів (об'єднання паралельних запитів на один і той самий ресурс).

3. Шар даних та сервісів (Data & Service Layer) Забезпечує роботу з конфігурацією та збереження результатів:

- ConfigService: Завантажує та кешує матрицю ймовірностей (navigation-probabilities.json), яка є основою для прийняття рішень предиктивним алгоритмом.
- LoggerService: Агрегатор телеметрії. Збирає деталізовані логи кожного переходу та завантаження активу, зберігаючи їх у локальному сховищі або файловій системі (OPFS) для подальшого аналізу.

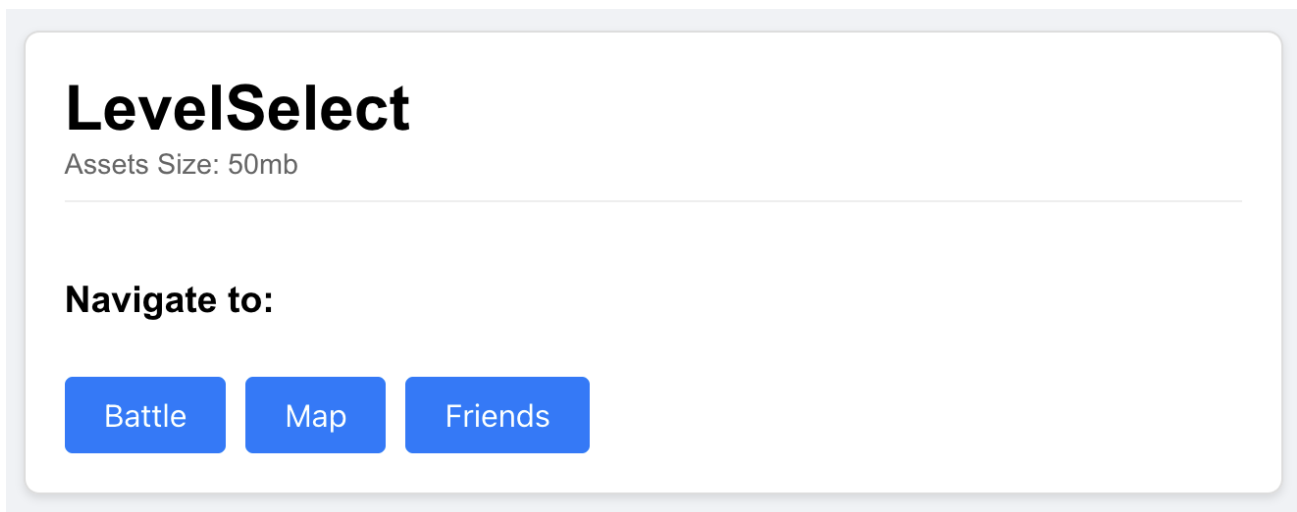
4. Шар емуляції (Emulation Layer) Спеціалізований набір скриптів для проведення масових експериментів без графічного інтерфейсу:

- SessionBatchRunner: Забезпечує запуск тисяч паралельних сесій користувачів.
- Polyfills: Набір адаптерів, що дозволяє виконувати браузерний код (наприклад, fetch, localStorage) у середовищі Node.js, забезпечуючи валідність отриманих даних.

Розроблена архітектура дозволяє провести комплексне дослідження ефективності запропонованого методу, порівнюючи його з альтернативними підходами на великому масиві статистичних даних.

3.3. Реалізація інтерфейсу користувача та візуалізація графа станів

Розроблене програмне середовище підтримує два сценарії експлуатації: інтерактивний режим для ручного тестування (Manual Testing) та автоматизований режим пакетної обробки (Batch Emulation). Для забезпечення можливості візуального контролю коректності роботи алгоритмів було реалізовано веб-інтерфейс, що складається з трьох ключових функціональних



блоків.

1. Блок ігрової сцени (Scene View)

Рис 3.2. Блок ігрової сцени

Центральний елемент інтерфейсу, що емулює екран реального мобільного додатку. Він відображає назву поточної локації (наприклад, "MainMenu") та обсяг ресурсів, необхідних для її відмалювання. Навігація реалізована через динамічну генерацію кнопок переходів на основі топології графа. Структура графа

описується у форматі JSON, де кожен вузол містить перелік суміжних вершин та метадані активів. Приклад опису вузла «Головне меню» наведено у Лістингу 3.1.

Лістинг 3.1 — Фрагмент файлу конфігурації nodes.json

```
"MainMenu": {  
  "assets": { "size": "30mb" },  
  "transitions": ["Map", "Inventory", "Shop", "Settings"]  
},  
...
```

2. Панель управління (Control Panel)

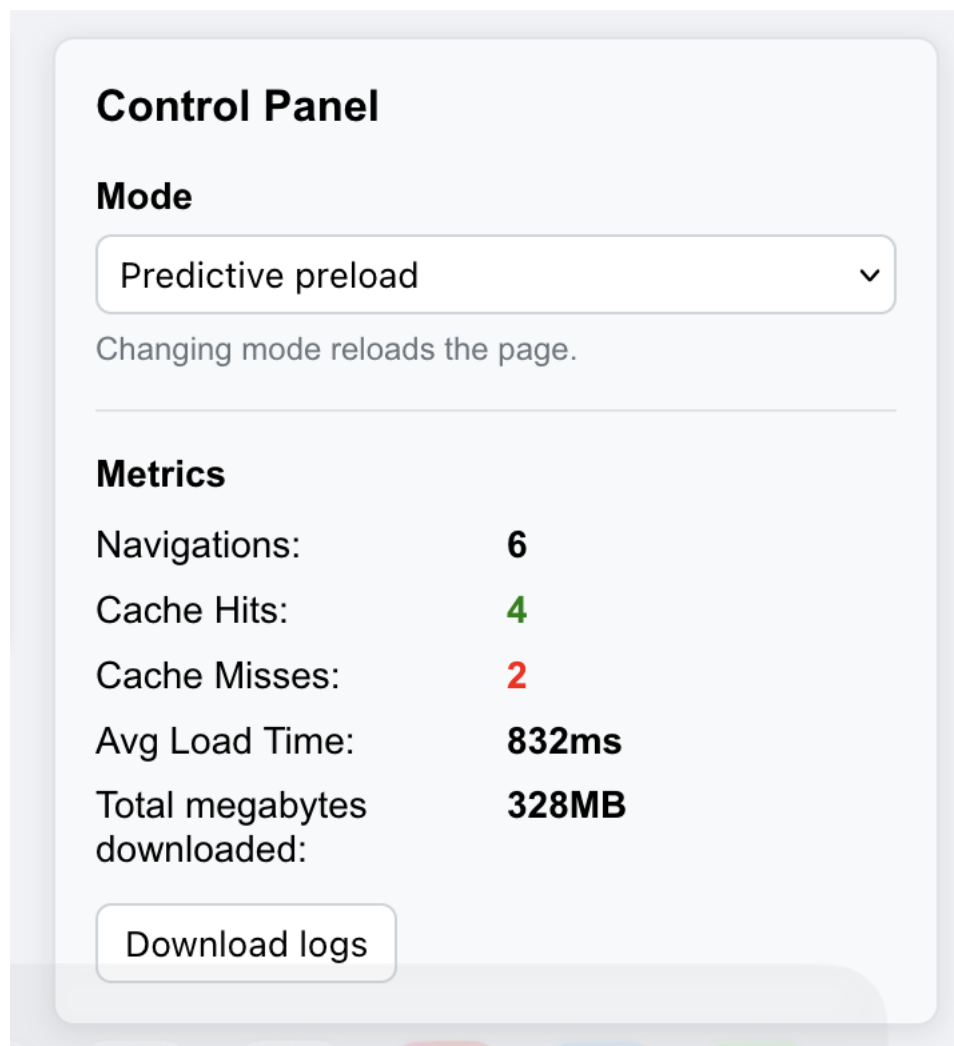


Рис 3.3. Панель управління

Цей компонент виконує роль адміністративного пульта дослідника. Він надає можливості:

- **Вибір стратегії завантаження:** Дозволяє перемикати режими (*On-Demand, Predictive, Predownload All*). При зміні режиму відбувається повне перезавантаження середовища для забезпечення чистоти експерименту.
- **Моніторинг метрик:** Відображає ключові KPI сесії в реальному часі: середній час завантаження (Avg Load Time), загальний обсяг трафіку (Total MB), кількість влучань у кеш (Cache Hits) та промахів (Cache Misses).
- **Експорт даних:** Кнопка «Download logs» ініціює збереження повного логу сесії у текстовий файл для подальшого аналізу.

3. Монітор статусу ресурсів (Assets Status Panel)

Assets Status	
Events	Not Loaded
Friends	Not Loaded
Inventory	Loading
LevelSelect	Not Loaded
MainMenu	Loaded
Map	Loading
Pause	Not Loaded
Results	Not Loaded
Settings	Not Loaded

Рис 3.4. Монітор статусу ресурсів

Інструмент для візуального дебагінгу, що відображає стан кожного активу в системі в одному з трьох станів:

- **Not Loaded:** Ресурс відсутній у пам'яті.
- **Loading:** Відбувається процес емуляції завантаження (Network I/O).
- **Loaded:** Ресурс успішно закешовано та доступний для миттєвого використання.

3.4. Побудова імовірнісної моделі навігації на основі статистичних даних

Ефективність предиктивного алгоритму безпосередньо залежить від точності вхідних даних. У сучасних високонавантажених системах джерелом таких даних виступають системи продуктової аналітики (наприклад, Google Analytics, Firebase, Amplitude), які агрегують поведінкові патерни мільйонів користувачів.

Агрегація індивідуальних сесій дозволяє трансформувати хаотичні переходи у зважений орієнтований граф, де вага ребра відповідає умовній ймовірності переходу. Наприклад, статистичний аналіз може показати, що з головного меню 45% гравців переходять на карту, 20% — до інвентарю, і лише 15% — до налаштувань.

У розробленому прототипі ця модель формалізована у файлі `navigation-probabilities.json`, який слугує базою знань для ланцюгів Маркова (Лістинг 3.2).

Лістинг 3.2 — Фрагмент матриці ймовірностей переходів

```
"MainMenu": {  
  "transitions": [  
    { "id": "Map", "probability": 0.45 },  
    { "id": "Inventory", "probability": 0.20 },  
    { "id": "Shop", "probability": 0.20 },  
    { "id": "Settings", "probability": 0.15 }  
  ]  
},
```

Саме цей розподіл ймовірностей дозволяє алгоритму ранжувати кандидатів на попереднє завантаження, віддаючи пріоритет найбільш вірогідним сценаріям поведінки.

3.5. Програмна реалізація механізмів управління ресурсами

Ядром системи є клас `AssetsManager`, який абстрагує роботу з мережею. Замість реального завантаження файлів, він використовує механізм симуляції затримки (`simulateDownload`), який конвертує розмір активу в мегабайтах у часовий інтервал на основі заданої пропускної здатності мережі.

Система підтримує три алгоритмічні стратегії управління чергою завантажень:

1. Реактивне завантаження (`Download on Demand`) Базовий сценарій, що реалізує патерн *Lazy Loading*. Завантаження ініціюється виключно в момент переходу користувача на нову сцену. Цей метод мінімізує споживання трафіку, але максимізує час очікування.

2. Повне попереднє завантаження (`Predownload All`) Стратегія «грубої сили», що намагається завантажити весь граф активів у фоновому режимі одразу після запуску додатку. Реалізація передбачає ітеративний обхід усіх вузлів графа (Лістинг 3.3).

Лістинг 3.3 — Реалізація стратегії повного завантаження

```
void (async () => {  
  const all = graphManager.getAllNodeIds();  
  for (const nodeId of all) {  
    try {  
      // isBlockingForPlayer: false вказує на фоновий режим  
      await assetsManager.downloadAsset(nodeId, {  
        trigger: 'predownload_all',  
        isBlockingForPlayer: false
```

```

    });
  } catch (e) {
    console.error('[PredownloadAll] Failed for node:', nodeId, e);
  }
}
})();

```

3. **Предиктивне завантаження (Predictive Download)** Розроблений адаптивний метод, що базується на стохастичній моделі. Алгоритм активується подією `onShowComplete` — тобто тільки після того, як поточний екран повністю відображено користувачу, що гарантує відсутність конкуренції за ресурси CPU під час рендерингу.

Логіка роботи методу (Лістинг 3.4):

1. Отримати список переходів з поточного вузла.
2. Відсортувати їх за спаданням ймовірності.
3. Відфільтрувати переходи з ймовірністю менше порогового значення ().
4. Обмежити чергу завантаження двома найбільш вірогідними кандидатами (Top-2).
5. Ініціювати фонове завантаження для активів, що відсутні в кеші.

Лістинг 3.4 — Реалізація предиктивного алгоритму в PreloadManager

```

async onShowComplete(currentNodeId: string) {
  if (!this.enabled) return;
  const config = configService.getConfig();
  if (!config) return;
  const nodeConfig = config.nodes[currentNodeId];
  if (!nodeConfig) return;
  const candidates = [...nodeConfig.transitions]
    .sort((a, b) => b.probability - a.probability)

```

```

        .filter(t => t.probability > 0.15)
        .slice(0, 2);

    console.log(`[PreloadManager] Preloading candidates for ${currentNodeId}:`,
candidates);

    for (const candidate of candidates) {
        if (!assetsManager.isAssetCached(candidate.id)) {
            assetsManager.downloadAsset(candidate.id, {
                trigger: 'preload',
                isBlockingForPlayer: false
            }).catch(err => console.error('Preload error:', err));
        }
    }
}

```

Такий підхід дозволяє використовувати «час роздумів» (Think Time) користувача на поточній сцені для підготовки ресурсів наступного кроку.

Висновки до розділу 3

У третьому розділі виконано програмну реалізацію методу предиктивного завантаження активів та створено комплексне середовище для оцінки його ефективності. Отримані результати можна узагальнити наступним чином:

1. Розроблено архітектуру емуляційного стенду. Створено SPA-додаток на базі React та TypeScript, який завдяки модульній архітектурі (розділення шарів UI, Logic Core, Data) дозволяє проводити як ручне тестування інтерфейсу, так і автоматизовані навантажувальні тести без участі людини.
2. Реалізовано математичну модель. Інтегровано систему обробки ймовірнісних графів навігації (файл `navigation-probabilities.json`), що дозволяє симулювати реалістичну поведінку користувачів на основі ланцюгів Маркова.
3. Впроваджено три стратегії управління ресурсами. Програмно реалізовано та забезпечено можливість гарячого перемикання між режимами *On-Demand*, *Predownload All* та *Predictive*. Ключовий алгоритм предиктивного завантаження використовує евристику фільтрації за ймовірністю () та обмеженням черги (Тор-2), що теоретично має забезпечити баланс між швидкістю та витратами трафіку. [25]
4. Створено систему телеметрії. Розроблений клас `AssetsManager` та сервіс логування забезпечують збір детальної статистики (час завантаження, обсяг трафіку, `Cache Hit/Miss`), що є необхідною умовою для проведення кількісного порівняльного аналізу в наступному розділі.

Створений програмний комплекс повністю готовий до проведення серії експериментів для підтвердження гіпотези про ефективність запропонованого методу предиктивного кешування.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1. Методика проведення експерименту та конфігурація тестового середовища

Для валідації запропонованого методу предиктивного завантаження було проведено серію автоматизованих експериментів. Метою тестування було отримання кількісних показників продуктивності (час завантаження) та економічності (використання трафіку) для трьох різних стратегій управління ресурсами.

Конфігурація тестового стенду: Експеримент проводився у середовищі емуляції зі стабілізованими параметрами, що дозволило нівелювати вплив зовнішніх факторів (флуктуації реальної мережі, продуктивність пристрою) та забезпечити відтворюваність результатів.

- Параметри емуляції:
 - Мережа: Пропускна здатність обмежена на рівні 10 MB/s, що відповідає стабільному з'єднанню 4G/LTE або Wi-Fi середньої якості.
 - Граф навігації: Використано граф з 10 вузлів, що імітує структуру типової мобільної гри (MainMenu, MissionSelect, Level1, Level2, Shop та ін.) [26].
 - Характеристика активів:
 - Легкі екрани (UI, Меню): 5–10 MB.
 - Важкі екрани (Ігрові рівні, 3D-сцени): 50–100 MB.
 - Тривалість сесії: 60 секунд.
 - Поведінка користувача: Емульовано 31 унікальну сесію для кожної групи. Час перебування на екрані варіювався від 1 до 5 секунд (час

на "роздуми"). Навігація здійснювалася стохастично відповідно до матриці ймовірностей navigation-probabilities.json.

Групи тестування: Для порівняльного аналізу було сформовано три експериментальні групи:

1. Група А (On-Demand / Lazy Loading): Завантаження активів відбувається виключно після ініціації переходу користувачем. Контрольна група.
2. Група Б (Predownload All): Стратегія повного попереднього завантаження всіх ресурсів додатку у фоновому режимі одразу після старту.
3. Група В (Predictive Prefetching): Запропонований метод, що завантажує лише найбільш вірогідні наступні екрани () під час простою користувача.

Запуск симуляції здійснювався за допомогою пакетного сценарію: `nrm run emulate:batch -- --sessionsPerMode 31 --durationSec 60 --minDelaySec 1 --maxDelaySec 5`

4.2. Аналіз отриманих результатів: порівняння метрик продуктивності

Після завершення серії експериментів було оброблено логи телеметрії та розраховано усереднені показники для кожної групи. Узагальнені результати наведено у Таблиці 4.1.

Таблиця 4.1. Порівняльні результати ефективності стратегій завантаження

Метрика	Група А (On-Demand)	Група Б (Predownload All)	Група В (Predictive)
Загальний час завантаження (ALT), сек	23	10	9
Споживання трафіку (Traffic), МБ	226	493	307
Cache Hit Ratio (CHR), %	39.1%	34.7%	43.5%

Інтерпретація результатів:

1. Сумарний час очікування (Latency): Найкращий результат продемонструвала Група В (Predictive) із показником 9 секунд сумарного очікування за сесію. Це більш ніж у 2.5 рази швидше за базовий сценарій (Група А — 23 с). Цікавим є той факт, що предиктивний метод випередив навіть стратегію повного завантаження (Група Б — 10 с). Це пояснюється явищем конкуренції за канал (Bandwidth Contention). У Групі Б система намагається завантажити весь масив даних одразу. Через великий загальний об'єм активів, процес викачування триває значну частину ігрової сесії. У цей період користувач здійснює переходи на екрани, які знаходяться в кінці черги завантаження, що призводить до Cache Miss та очікування. Натомість, Група В пріоритезує лише *необхідне*, забезпечуючи наявність саме тих активів, які потрібні користувачу в наступний момент.
2. Динаміка Cache Hit Ratio (CHR): Показник CHR у Групі В (43.5%) є найвищим. Хоча теоретично Група Б (Predownload All) має прагнути до 100% покриття кешем, на практиці в умовах обмеженої сесії (60 с) та лімітованого каналу цього не відбувається. Гравці "Групи Б" значну частину часу проводять у стані активного завантаження фонових ресурсів,

що парадоксально знижує ймовірність миттєвого доступу до актуального контенту порівняно з розумним прогнозуванням.

3. Витрати трафіку: Група А є найбільш ощадливою (226 МБ), але найповільнішою. Група Б демонструє надмірне споживання ресурсів (493 МБ), завантажуючи контент, до якого користувач може ніколи не дійти. Група В займає проміжну позицію (307 МБ), інвестуючи додатковий трафік виключно у прискорення UX.

4.3. Оцінка компромісу між використанням трафіку та швидкістю (Trade-off analysis)

Результати експерименту дозволяють сформулювати висновки щодо балансу ефективності:

Оптимальний баланс Predictive (Група В): Запропонований метод демонструє найкраще співвідношення "ціна/якість".

- Витрати: Збільшення трафіку на ~36% (порівняно з On-Demand).
- Результат: Скорочення часу очікування на ~60% (з 23 с до 9 с) та перемога над стратегією повного завантаження. Це робить предиктивний підхід "золотою серединою", яка забезпечує преміальний користувацький досвід без необхідності викачувати гігабайти зайвих даних.

Проблема "Холодного старту" у Predownload All (Група Б): Стратегія повного завантаження виявилася менш ефективною в короткостроковій перспективі. Хоча вона гарантує відсутність затримок у дуже довгих сесіях (коли все вже завантажено), на початкових етапах вона створює "затор" у каналі передачі даних. Користувачі змушені конкурувати з фоновим завантажувачем за пропускну здатність, що призводить до промахів кешу (Cache Miss) саме тоді, коли чуйність інтерфейсу є найбільш критичною для першого враження.

Висновки до розділу 4

Проведене експериментальне дослідження забезпечило емпіричну верифікацію запропонованої математичної моделі та дозволило сформулювати наступні узагальнюючі висновки:

1. Доведення переваги стохастичного прогнозування. Результати експерименту однозначно підтверджують, що інтелектуальна пріоритезація завантаження на основі ланцюгів Маркова (Група В) є найбільш ефективною стратегією в умовах обмеженої пропускної здатності мережі. Метод дозволив досягти скорочення сумарного часу очікування у 2,5 рази (до 9 с) порівняно з реактивним підходом, що є критичним показником для забезпечення безперервності ігрового процесу. Це свідчить про те, що використання «часу роздумів» (Think Time) для фоновієї підготовки активів дозволяє фактично нівелювати мережеві затримки для користувача.
2. Вирішення проблеми «холодного старту» та мережевої конкуренції. Важливим відкриттям дослідження стала неефективність стратегії повного передзавантаження (Група Б) на коротких дистанціях. Експеримент продемонстрував, що спроба завантажити все одразу створює ефект «пляшкового горлечка» (Network Congestion), коли корисні дані конкурують у каналі з другорядними. Натомість предиктивний метод забезпечує «розумну чергу», гарантуючи, що пропускна здатність каналу витрачається саме на ті активи, які мають найвищу ймовірність бути запитаними у найближчі секунди. Це дозволило Групі В випередити Групу Б за швидкістю, навіть маючи менший теоретичний обсяг кешованих даних.
3. Оптимізація балансу «Вартість — Результат». Аналіз компромісів (Trade-off analysis) показав високу економічну доцільність впровадження

предиктивного методу. Зростання споживання трафіку на 36% (порівняно з On-Demand) конвертується у непропорційно великий виграш у швидкодії (покращення на ~60%). У сучасних умовах, де вартість передачі даних постійно знижується, а ціна втрати користувача через поганий UX зростає, такий «обмін» трафіку на час є виправданою інвестицією в показники утримання (Retention Rate).

4. Архітектурна значущість. Отримані дані свідчать про те, що відмова від жорстких детермінованих алгоритмів (завантажувати все або нічого) на користь імовірнісних моделей дозволяє системі адаптуватися до поведінки користувача. Це перетворює підсистему завантаження ресурсів з пасивного інструменту на активний компонент управління користувацьким досвідом (Quality of Experience), здатний забезпечувати відчуття миттєвої реакції інтерфейсу навіть у мережах середньої якості (3G/4G).

ВИСНОВКИ

У ході виконання магістерської роботи було вирішено науково-прикладну задачу підвищення чуйності інтерфейсу мобільних SPA-додатків. Шляхом розробки та експериментальної перевірки методу предиктивного завантаження активів отримано наступні результати:

1. Створено ефективну експериментальну платформу. Розроблене програмне середовище, що включає клієнтський додаток та модуль пакетної емуляції, довело свою спроможність як інструмент для моделювання поведінки користувачів та валідації стратегій управління ресурсами. Це дозволило отримати об'єктивні кількісні дані без необхідності залучення реальної аудиторії на етапі R&D.
2. Підтверджено ефективність запропонованого методу. Експериментально доведено перевагу стратегії Predictive Download (на основі ланцюгів Маркова) над традиційними підходами. Метод забезпечив найкращий показник сумарного часу завантаження (9 секунд на сесію), що у 2.5 рази швидше за реактивний підхід (*On-Demand*, 23 с) та навіть ефективніше за повне попереднє завантаження (*Predownload All*, 10 с) в умовах коротких сесій.
3. Досягнуто оптимального балансу ресурсів. Встановлено, що використання ймовірнісного прогнозування дозволяє утримувати високий рівень Cache Hit Ratio (43.5%) при помірному споживанні трафіку (зростання на 36% відносно базового рівня), уникаючи проблеми перевантаження каналу, яка є критичною для методу повного завантаження.

Практичне значення результатів. Отримані результати та розроблені архітектурні рішення мають універсальний характер і можуть бути імплементовані у широкий спектр програмних продуктів:

1. Універсальність застосування: Запропонований підхід є агностичним до типу контенту. Він може бути ефективно застосований у будь-яких системах, де (а) ресурси завантажуються асинхронно (з диска або мережі) і (б) існує можливість побудувати граф станів та зібрати статистику переходів. Це стосується не лише ігор, але й E-commerce додатків, стрімінгових сервісів та складних корпоративних інтерфейсів.
2. Бізнес-цінність (R&D Focus): В умовах високої конкуренції на ринку мобільних додатків показник чуйності (Responsiveness) стає ключовим фактором утримання користувачів. Використання розробленого методу дозволяє компаніям покращити UX без необхідності повної переробки архітектури додатку, лише шляхом інтеграції модуля предиктивного завантаження.
3. Економічна доцільність: Метод дозволяє знизити витрати на CDN порівняно з повним передзавантаженням, фокусуючись лише на тих даних, які з високою ймовірністю будуть використані.

Обмеження проведеного дослідження. При інтерпретації результатів необхідно враховувати певні обмеження, зумовлені специфікою експериментального середовища:

1. Штучність середовища емуляції: Експеримент проводився за умов стабільної пропускної здатності мережі (10 MB/s), що є ідеалізованим сценарієм. У реальних мобільних мережах можливі флуктуації швидкості, втрата пакетів та зміна типу з'єднання (Wi-Fi/4G), що може вплинути на точність виконання прогнозів.
2. Фактор тривалості сесії: Дослідження було обмежено сесіями тривалістю 60 секунд. Це створило певний біас (упередження) проти стратегії *Predownload All*. Якби середня тривалість сесії була значно довшою (наприклад, 10–15 хвилин), стратегія повного завантаження з часом компенсувала б початкові затримки та забезпечила б 100% Cache Hit на

пізніх етапах гри. Отже, предиктивний метод є найбільш ефективним саме для коротких та середніх сесій, характерних для мобільного геймінгу.

Рекомендації та напрямки подальших досліджень. Для подальшого розвитку тематики та вдосконалення методу пропонуються наступні напрямки:

1. Розширення сфери застосування (API Prefetching): Принцип предиктивного завантаження доцільно поширити не лише на статичні асети (зображення, моделі), але й на «легкі» GET HTTP-запити (JSON дані), що не змінюють стан сервера (ідемпотентні запити). Це дозволить миттєво відображати динамічний контент, наприклад, профіль гравця або список товарів.
2. Кластеризація користувачів (Segmentation): Точність прогнозів можна суттєво підвищити шляхом сегментації аудиторії. Різні категорії користувачів мають різні патерни поведінки (наприклад, «High-level гравці» частіше відвідують магазин або рейди, тоді як «новачки» — навчальні рівні). Побудова окремих матриць ймовірностей для кожного кластера дозволить зробити передбачення більш персоналізованими та ефективними.
3. Динамічна адаптація (Real-time Learning): Перспективним є впровадження механізму оновлення ваги ребер графа в реальному часі безпосередньо на пристрої клієнта, підлаштовуючись під поточну ігрову сесію конкретного користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mobile App Growth Statistics 2025 // SQ Magazine. 2025.
URL: <https://sqmagazine.co.uk/mobile-app-growth-statistics/> (дата звернення 11.01.2026).
2. State of Mobile Gaming 2024 // Sensor Tower. 2024.
URL: <https://sensortower.com/blog/state-of-mobile-gaming-2024> (дата звернення 11.01.2026).
3. Mobile Game Retention Rates (2025) // Business of Apps. 2025.
URL: <https://www.businessofapps.com/data/mobile-game-retention-rates/> (дата звернення 11.01.2026).
4. Predicting Purchase Decisions in Mobile Free-to-Play Games.
URL: <https://cdn.aaai.org/ojs/12788/12788-52-16305-1-2-20201228.pdf> (дата звернення 21.12.2025).
5. Correlation Between App Load Times and User Retention Rates // Zigpoll.
URL: <https://www.zigpoll.com> (дата звернення 11.01.2026).
6. How Game Performance Affects Player Retention // Geniee.
URL: <https://genieee.com/how-game-performance-affects-player-retention/> (дата звернення 11.01.2026).
7. Claypool M. The Effects of Game Loading Time on Quality of Experience // FDG 2024: Proceedings of the 19th International Conference on the Foundations of Digital Games. Worcester, MA, USA, 2024. Article No. 11.
URL: <https://web.cs.wpi.edu/~claypool/papers/game-load-fdg-24/paper.pdf> (дата звернення 11.01.2026).
8. By Making the Site 1 Second Faster We Could Increase Engagement // Medium.
URL: <https://medium.com/design-bootcamp/by-making-the-site-1-second-faster-we-could-increase-engagement-by-5-87cd72eac643> (дата звернення 21.12.2025).

9. Top 8 Web Application Performance Metrics // MetricFire. Medium.
URL: <https://medium.com/@MetricFire/top-8-web-application-performance-metrics-a9afdf4cec611> (дата звернення 21.12.2025).
- 10.Asset Bundles Tips and Pitfalls // Unity.
URL: <https://unity.com/blog/engine-platform/unity-asset-bundles-tips-pitfall> (дата звернення 21.12.2025).
- 11.Introducing Speed Brain: Helping Web Pages Load 45% Faster // Cloudflare Blog.
URL: <https://blog.cloudflare.com/introducing-speed-brain/> (дата звернення 21.12.2025).
- 12.Optimize network access: Prefetch data // Android Developers.
URL: <https://developer.android.com/develop/connectivity/network-ops/network-access-optimization> (дата звернення 11.01.2026).
- 13.Reid L. Art Asset Budgeting Strategies for Mobile // Linden Reid Blog. 2022.
URL: <https://lindenreidblog.com/2022/05/27/optimization-strategies-for-mobile/> (дата звернення 11.01.2026).
- 14.How Fast Load Times Improve User Experience // Addicta.
URL: <https://addictaco.com/how-fast-load-times-improve-user-experience/> (дата звернення 11.01.2026).
- 15.Lee S., Li T., Bhattacharya D. та ін. AppStreamer: Reducing Storage Requirements of Mobile Games through Predictive Streaming // Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks (EWSN '20). 2020. P. 167–178.
URL: https://engineering.purdue.edu/dcs1/publications/papers/2019/appstreamer_ewsn_20_cameraready.pdf (дата звернення 11.01.2026).

16. Predictive Preloading // Speed Kit.
URL: <https://www.speedkit.com/feature/predictive-preloading> (дата звернення 11.01.2026).
17. Mathematics for Computer Science: Directed Graphs // MIT OpenCourseWare.
2010.
URL: https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-fall-2010/e6db7638031b754f5f68012946af4763_MIT6_042JF10_chap06.pdf
(дата звернення 11.01.2026).
18. Markov Chain Based Web User Journey Prediction // ThatWare.
URL: <https://thatware.co/markov-chain-based-web-user-journey-prediction/>
(дата звернення 11.01.2026).
19. What is a Cache Hit Ratio and How Do You Calculate It? // StormIT.
URL: <https://www.stormit.cloud/blog/cache-hit-ratio-what-is-it/> (дата звернення 21.12.2025).
20. Web Navigation Prediction Using Markov-Based Models // Semantic Scholar.
URL: <https://www.semanticscholar.org/paper/Web-navigation-prediction-using-Markov-based-an-Jindal-Sardana/098e25a272375ed82d773936fc811996879f2c76> (дата звернення 21.12.2025).
21. Visitor Navigation Pattern Prediction Using Markov Model, Association Rules and Ambiguous Rules // ResearchGate.
URL:
https://www.researchgate.net/publication/365728941_Visitor_Navigation_Pattern_Prediction_using_Markov_Model_Association_rules_and_Ambiguous_rules
(дата звернення 21.12.2025).
22. Graph Theory's Applications in Real-World // Philippine Statistics Authority.
URL:

<https://www.philstat.org/index.php/MSEA/article/download/2926/2306/5078>

(дата звернення 21.12.2025).

23. An Introduction to Markov Chains (Step by Step) // CalcWorkshop.
URL: <https://calcworkshop.com/vector-spaces/markov-chain-applications/> (дата звернення 21.12.2025).
24. How Markov Chains Predict Outcomes in Games Like Big Bass Splash // OWTC.
URL: <https://owtc.org/how-markov-chains-predict-outcomes-in-games-like-big-bass-splash/> (дата звернення 21.12.2025).
25. Prefetching Location-Based Metaverse Assets over Named Data Networking.
URL: <https://ieeexplore.ieee.org/iel8/11223274/11223287/11223322.pdf> (дата звернення 21.12.2025).
26. Graph-Based Path Planning // Intro to Autonomous Robots Class Notes.
URL: <https://fiveable.me/introduction-autonomous-robots/unit-6/graph-based-path-planning/study-guide/BxSNGYOnyvVBOeM3> (дата звернення 21.12.2025).

ДОДАТКИ

Додаток А. Вихідний код додатку

```
---  
// ===== FILE: src/core/AssetsManager.ts =====  
  
import { parseAssetSize } from '../utils/assetSizeParser';  
import { simulateDownload } from '../utils/downloadSimulator';  
import { graphManager } from './GraphManager';  
import { loggerService } from './services/LoggerService';  
import type { AssetLoadTrigger } from './types';  
  
interface ActiveDownload {  
  controller: AbortController;  
  promise: Promise<void>;  
  listeners: ((progress: number) => void)[];  
  lastProgress: number;  
  sizeBytes: number;  
  downloadedBytes: number;  
}  
  
export interface DownloadAssetOptions {  
  onProgress?: (progress: number) => void;  
  trigger?: AssetLoadTrigger;  
  
  navigationIndex?: number;  
  
  isBlockingForPlayer?: boolean;  
}  
  
type AllocatedBytesListener = (bytes: number) => void;
```

```

type TotalMegabytesDownloadedListener = (mb: number) => void;

class AssetsManager {
  private cache = new Set<string>();
  private cachedAssetBytes = new Map<string, number>();
  private activeDownloads = new Map<string, ActiveDownload>();

  private allocatedBytesListeners = new Set<AllocatedBytesListener>();
  private totalMegabytesDownloadedListeners = new
Set<TotalMegabytesDownloadedListener>();

  private notifyAllocatedBytesChanged() {
    const bytes = this.getCurrentAllocatedBytes();
    this.allocatedBytesListeners.forEach((l) => l(bytes));
  }

  private notifyTotalMegabytesDownloadedChanged() {
    const mb = this.getTotalMegabytesDownloaded();
    this.totalMegabytesDownloadedListeners.forEach((l) => l(mb));
  }

  onAllocatedBytesChanged(listener: AllocatedBytesListener): () => void {
    this.allocatedBytesListeners.add(listener);

    listener(this.getCurrentAllocatedBytes());
    return () => this.allocatedBytesListeners.delete(listener);
  }
}

```

```

onTotalMegabytesDownloadedChanged(listener:
TotalMegabytesDownloadedListener): () => void {
    this.totalMegabytesDownloadedListeners.add(listener);

    listener(this.getTotalMegabytesDownloaded());
    return () => this.totalMegabytesDownloadedListeners.delete(listener);
}

getCurrentAllocatedBytes(): number {
    let bytes = 0;
    for (const b of this.cachedAssetBytes.values()) bytes += b;
    for (const d of this.activeDownloads.values()) bytes += d.downloadedBytes;
    return bytes;
}

getTotalMegabytesDownloaded(): number {
    return this.getCurrentAllocatedBytes() / (1024 * 1024);
}

isAssetCached(nodeId: string): boolean {
    return this.cache.has(nodeId);
}

getCacheStatus(): Set<string> {
    return new Set(this.cache);
}

clearCache(): void {
    this.cache.clear();
    this.cachedAssetBytes.clear();
}

```

```

    this.notifyAllocatedBytesChanged();
    this.notifyTotalMegabytesDownloadedChanged();
}

```

```

private getAssetSizeBytes(nodeId: string): number {
    const node = graphManager.getNode(nodeId);
    if (!node) return 0;
    const sizeMB = parseAssetSize(node.assets.size);
    return Math.max(0, Math.round(sizeMB * 1024 * 1024));
}

```

```

private logAssetLine(params: {
    nodeId: string;
    trigger: AssetLoadTrigger;
    navigationIndex: number;
    wasCacheHit: boolean;
    playerLoadDurationMs: number;
    assetSizeBytes: number;
}) {
    loggerService.logAssetDownload({
        timestamp: Date.now(),
        nodeId: params.nodeId,
        trigger: params.trigger,
        navigationIndex: params.navigationIndex,
        wasCacheHit: params.wasCacheHit,
        playerLoadDurationMs: params.playerLoadDurationMs,
        totalMegabytesDownloaded: this.getTotalMegabytesDownloaded(),
        assetSizeBytes: params.assetSizeBytes
    });
}

```



```
});  
}
```

```
async downloadAsset(nodeId: string, options: DownloadAssetOptions = {}):  
Promise<void> {  
  const {  
    onProgress,  
    trigger = 'unknown',  
    navigationIndex = loggerService.getCurrentNavigationIndex(),  
    isBlockingForPlayer = true  
  } = options;  
  
  const start = Date.now();  
  const wasCachedAtStart = this.isAssetCached(nodeId);  
  const assetSizeBytes = this.getAssetSizeBytes(nodeId);  
  
  if (wasCachedAtStart) {  
    onProgress?.(100);  
    this.logAssetLine({  
      nodeId,  
      trigger,  
      navigationIndex,  
      wasCacheHit: true,  
      playerLoadDurationMs: 0,  
      assetSizeBytes  
    });  
    return;  
  }  
}
```

```

const existing = this.activeDownloads.get(nodeId);
if (existing) {
  console.log(`[AssetsManager] Attaching to existing download for: ${nodeId}`);
  if (onProgress) {
    onProgress(existing.lastProgress);
    existing.listeners.push(onProgress);
  }
  await existing.promise;
  const duration = Date.now() - start;
  this.logAssetLine({
    nodeId,
    trigger,
    navigationIndex,
    wasCacheHit: false,
    playerLoadDurationMs: isBlockingForPlayer ? duration : 0,
    assetSizeBytes: existing.sizeBytes
  });
  return;
}

```

```

const node = graphManager.getNode(nodeId);
if (!node) {
  throw new Error(`Node ${nodeId} not found`);
}
const sizeMB = parseAssetSize(node.assets.size);
const controller = new AbortController();
const listeners: ((p: number) => void)[] = [];

```

```

if (onProgress) listeners.push(onProgress);
const updateProgress = (p: number) => {
  const download = this.activeDownloads.get(nodeId);
  if (download) {
    download.lastProgress = p;
    download.downloadedBytes = Math.round((download.sizeBytes * p) / 100);
    this.notifyAllocatedBytesChanged();
    this.notifyTotalMegabytesDownloadedChanged();
  }
  listeners.forEach(l => l(p));
};
const promise = (async () => {
  try {
    console.log(`[AssetsManager] Starting download for: ${nodeId}`);
    await simulateDownload(
      sizeMB,
      updateProgress,
      controller.signal
    );
    this.cache.add(nodeId);
    this.cachedAssetBytes.set(nodeId, assetSizeBytes);
    console.log(`[AssetsManager] Download finished for: ${nodeId}`);
  } catch (e: any) {
    if (e.message === 'Download aborted') {
      console.log(`Download aborted for ${nodeId}`);
    } else {
      throw e;
    }
  }
}

```

```

    } finally {
        this.activeDownloads.delete(nodeId);
        this.notifyAllocatedBytesChanged();
        this.notifyTotalMegabytesDownloadedChanged();
    }
    })();
    this.activeDownloads.set(nodeId, {
        controller,
        promise,
        listeners,
        lastProgress: 0,
        sizeBytes: assetSizeBytes,
        downloadedBytes: 0
    });

    await promise;
    const duration = Date.now() - start;
    this.logAssetLine({
        nodeId,
        trigger,
        navigationIndex,
        wasCacheHit: false,
        playerLoadDurationMs: isBlockingForPlayer ? duration : 0,
        assetSizeBytes
    });
    return;
}

isDownloading(nodeId: string): boolean {

```

```

    return this.activeDownloads.has(nodeId);
  }
  cancelDownload(nodeId: string): void {
    const existing = this.activeDownloads.get(nodeId);
    if (existing) {
      existing.controller.abort();
      this.activeDownloads.delete(nodeId);
      this.notifyAllocatedBytesChanged();
      this.notifyTotalMegabytesDownloadedChanged();
    }
  }
}

export const assetsManager = new AssetsManager();

// ===== FILE: src/core/GraphManager.ts =====
import type { GraphConfig, Node } from '../types';
class GraphManager {
  private nodes: Record<string, Node> = {};
  private initialized = false;
  async loadGraph(): Promise<void> {
    if (this.initialized) return;
    try {
      const response = await fetch('/nodes.json');
      if (!response.ok) {
        throw new Error(`Failed to load nodes: ${response.statusText}`);
      }
      const data: GraphConfig = await response.json();
      this.nodes = data.nodes;
    }
  }
}

```

```

    this.initialized = true;
  } catch (error) {
    console.error('Error loading graph:', error);
    throw error;
  }
}

getNode(nodeId: string): Node | undefined {
  return this.nodes[nodeId];
}

getAvailableTransitions(nodeId: string): string[] {
  const node = this.nodes[nodeId];
  return node ? node.transitions : [];
}

isValidTransition(fromId: string, toId: string): boolean {
  const transitions = this.getAvailableTransitions(fromId);
  return transitions.includes(toId);
}

getAllNodeIds(): string[] {
  return Object.keys(this.nodes);
}
}

export const graphManager = new GraphManager();

// ===== FILE: src/core/NavigationManager.ts =====
import { graphManager } from './GraphManager';
class NavigationManager {
  private history: string[] = [];
  initialize(startNode: string) {

```

```

    this.history = [startNode];
  }
  validateTransition(fromNode: string, toNode: string): boolean {
    return graphManager.isValidTransition(fromNode, toNode);
  }
  pushHistory(nodeId: string) {
    this.history.push(nodeId);
  }
  popHistory(): string | undefined {
    if (this.history.length > 1) {
      this.history.pop();
      return this.history[this.history.length - 1];
    }
    return undefined;
  }
  getHistory(): string[] {
    return [...this.history];
  }
  getAvailableTransitions(nodeId: string): string[] {
    return graphManager.getAvailableTransitions(nodeId);
  }
}

export const navigationManager = new NavigationManager();

// ===== FILE: src/core/PreloadManager.ts =====
import { configService } from '../services/ConfigService';
import { assetsManager } from '../AssetsManager';
class PreloadManager {

```

```

private enabled = false;

enable() {
  this.enabled = true;
  configService.fetchPreloadConfig();
}

disable() {
  this.enabled = false;
}

isEnabled() {
  return this.enabled;
}

async onShowComplete(currentNodeId: string) {
  if (!this.enabled) return;
  const config = configService.getConfig();
  if (!config) return;
  const nodeConfig = config.nodes[currentNodeId];
  if (!nodeConfig) return;
  const candidates = [...nodeConfig.transitions]
    .sort((a, b) => b.probability - a.probability)
    .filter(t => t.probability > 0.15)
    .slice(0, 2);
  console.log(`[PreloadManager] Preloading candidates for ${currentNodeId}:`,
candidates);
  for (const candidate of candidates) {
    if (!assetsManager.isAssetCached(candidate.id)) {
      assetsManager.downloadAsset(candidate.id, {
        trigger: 'preload',
        isBlockingForPlayer: false

```



```

    }).catch(err => console.error('Preload error:', err));
  }
}
}
}

export const preloadManager = new PreloadManager();

// ===== FILE: src/index.css =====
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  background-color: #f0f2f5;
}
code {
  font-family: source-code-pro, Menlo, Monaco, Consolas, 'Courier New',
    monospace;
}

// ===== FILE: src/main.tsx =====
import React from 'react'
import ReactDOM from 'react-dom/client'
import { App } from './ui/App'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')!).render(

```

```

<React.StrictMode>
  <App />
</React.StrictMode>,
)

```

```

// ===== FILE: src/services/ConfigService.ts =====
import type { PreloadConfig } from '../types';
class ConfigService {
  private config: PreloadConfig | null = null;
  async fetchPreloadConfig(): Promise<PreloadConfig> {
    if (this.config) {
      return this.config;
    }
    try {
      const response = await fetch('/navigation-probabilities.json');
      if (!response.ok) {
        throw new Error(`Failed to load config: ${response.statusText}`);
      }
      this.config = await response.json();
      return this.config!;
    } catch (error) {
      console.error('Error fetching preload config:', error);
      return { nodes: {} };
    }
  }
  getConfig(): PreloadConfig | null {
    return this.config;
  }
}

```

```

}

export const configService = new ConfigService();

// ===== FILE: src/services/LoggerService.ts =====

import type { AssetDownloadLogLine, NavigationLog } from '../types';

function formatMB(mb: number): string {
  if (!Number.isFinite(mb) || mb <= 0) return '0MB';
  return `${mb.toFixed(mb < 10 ? 2 : mb < 100 ? 1 : 0)}MB`;
}

function bytesToMB(bytes: number): number {
  if (!Number.isFinite(bytes) || bytes <= 0) return 0;
  return bytes / (1024 * 1024);
}

function formatAssetLogLine(
  l: AssetDownloadLogLine,
  totals: { totalWaitTimeMs: number; totalCacheHits: number; totalCacheMisses:
number }
): string {
  const ts = new Date(l.timestamp).toISOString();
  const hitMiss = l.wasCacheHit ? 'HIT' : 'MISS';
  return [
    ts,
    `nav=${l.navigationIndex}`,
    `trigger=${l.trigger}`,
    `asset=${l.nodeId}`,
  ]

```

```

    `cache=${hitMiss}`,
    `playerLoad=${Math.round(l.playerLoadDurationMs)}ms`,
    `totalWait=${Math.round(totals.totalWaitTimeMs)}ms`,
    `allHits=${totals.totalCacheHits}`,
    `allMisses=${totals.totalCacheMisses}`,
    `totalMBDownloaded=${formatMB(l.totalMegabytesDownloaded)}`,
    `assetSize=${formatMB(bytesToMB(l.assetSizeBytes))}`
  ].join(' ');
}

async function appendLineToPersistentLog(line: string): Promise<void> {
  const navAny = navigator as any;
  if (navAny?.storage?.getDirectory) {
    const dir = await navAny.storage.getDirectory();
    const handle = await dir.getFileHandle('asset-metrics.log', { create: true });
    const file = await handle.getFile();
    const writable = await handle.createWritable({ keepExistingData: true });
    await writable.seek(file.size);
    await writable.write(`${line}\n`);
    await writable.close();
    return;
  }

  const key = 'asset-metrics.log';
  const existing = localStorage.getItem(key) ?? '';
  localStorage.setItem(key, `${existing}${line}\n`);
}

```

```

class LoggerService {
  private logs: NavigationLog[] = [];
  private assetLines: string[] = [];
  private writeQueue: Promise<void> = Promise.resolve();
  private currentNavigationIndex = 0;
  private totalWaitTimeMs = 0;
  private totalCacheHits = 0;
  private totalCacheMisses = 0;

  setCurrentNavigationIndex(navigationIndex: number): void {
    this.currentNavigationIndex = navigationIndex;
  }

  getCurrentNavigationIndex(): number {
    return this.currentNavigationIndex;
  }

  logNavigation(from: string, to: string, assetLoadDuration: number, wasAssetCached:
boolean): void {
    const log: NavigationLog = {
      timestamp: Date.now(),
      fromNode: from,
      toNode: to,
      assetLoadDuration,
      wasAssetCached
    };
    this.logs.push(log);
    console.log('Navigation Logged:', log);
  }
}

```

```
}
```

```
logAssetDownload(line: AssetDownloadLogLine): void {
```

```
  this.totalWaitTimeMs += line.playerLoadDurationMs;
```

```
  if (line.wasCacheHit) {
```

```
    this.totalCacheHits += 1;
```

```
  } else {
```

```
    this.totalCacheMisses += 1;
```

```
}
```

```
const formatted = formatAssetLogLine(line, {
```

```
  totalWaitTimeMs: this.totalWaitTimeMs,
```

```
  totalCacheHits: this.totalCacheHits,
```

```
  totalCacheMisses: this.totalCacheMisses
```

```
});
```

```
this.assetLines.push(formatted);
```

```
console.log(formatted);
```

```
this.writeQueue = this.writeQueue
```

```
  .then(() => appendLineToPersistentLog(formatted))
```

```
  .catch((e) => console.warn('[LoggerService] Failed to persist log line:', e));
```

```
}
```

```
exportAssetLogText(): string {
```

```
  return `${this.assetLines.join('\n')}\n`;
```

```
}
```

```
downloadAssetLogFile(fileName = 'asset-metrics.log'): void {
```

```
  const text = this.exportAssetLogText();
```

```

const blob = new Blob([text], { type: 'text/plain' });
const url = URL.createObjectURL(blob);
const a = document.createElement('a');
a.href = url;
a.download = fileName;
a.click();
setTimeout(() => URL.revokeObjectURL(url), 0);
}

exportLogs(): NavigationLog[] {
  return [...this.logs];
}

async sendLogsToBackend(): Promise<void> {
  console.log(`Sending ${this.logs.length} logs to backend...`);
  await new Promise(resolve => setTimeout(resolve, 500));
  console.log('Logs sent successfully (simulated)');
}
}

export const loggerService = new LoggerService();

// ===== FILE: src/store/useAppStore.ts =====
import { create } from 'zustand';
import type { ScreenState, Metrics, PreloadMode } from '../types';
import { graphManager } from '../core/GraphManager';
import { navigationManager } from '../core/NavigationManager';
import { assetsManager } from '../core/AssetsManager';
import { preloadManager } from '../core/PreloadManager';
import { loggerService } from '../services/LoggerService';

```

```

function readPreloadModeFromStorage(): PreloadMode {
  const mode = localStorage.getItem('preload_mode');
  if (mode === 'on_demand' || mode === 'predictive' || mode === 'predownload_all') {
    return mode;
  }

  const legacyEnabled = localStorage.getItem('preload_enabled') === 'true';
  const migrated: PreloadMode = legacyEnabled ? 'predictive' : 'on_demand';
  localStorage.setItem('preload_mode', migrated);
  return migrated;
}

interface AppState {
  currentScreen: string | null;
  screenState: ScreenState;
  loadingProgress: number;
  preloadMode: PreloadMode;
  metrics: Metrics;
  initApp: () => Promise<void>;
  navigateTo: (nodeId: string) => Promise<void>;
  goBack: () => void;
  setPreloadMode: (mode: PreloadMode) => void;
  completeShow: () => void;
}

export const useAppStore = create<AppState>((set, get) => ({
  currentScreen: null,
  screenState: 'IDLE',
  loadingProgress: 0,

```



```

preloadMode: readPreloadModeFromStorage(),
metrics: {
  totalNavigations: 0,
  cacheHits: 0,
  cacheMisses: 0,
  averageLoadTime: 0,
  preloadAccuracy: 0,
  totalMegabytesDownloaded: 0
},
initApp: async () => {

  assetsManager.onTotalMegabytesDownloadedChanged((mb) => {
    set((state) => ({
      metrics: {
        ...state.metrics,
        totalMegabytesDownloaded: mb
      }
    }));
  });

  const { preloadMode } = get();
  if (preloadMode === 'predictive') {
    preloadManager.enable();
  } else {
    preloadManager.disable();
  }
  await graphManager.loadGraph();
  const startNode = 'MainMenu';

```

```

navigationManager.initialize(startNode);
set({ screenState: 'LOADING', currentScreen: startNode, loadingProgress: 0 });
loggerService.setCurrentNavigationIndex(0);
await assetsManager.downloadAsset(startNode, {
  trigger: 'init',
  navigationIndex: 0,
  isBlockingForPlayer: true,
  onProgress: (p) => set({ loadingProgress: p })
});
set({
  screenState: 'SHOW',
  loadingProgress: 100
});
get().completeShow();

```

```

if (preloadMode === 'predownload_all') {
  void (async () => {
    const all = graphManager.getAllNodeIds();
    for (const nodeId of all) {
      try {
        await assetsManager.downloadAsset(nodeId, {
          trigger: 'predownload_all',
          isBlockingForPlayer: false
        });
      } catch (e) {
        console.error('[PredownloadAll] Failed for node:', nodeId, e);
      }
    }
  })
}

```

```

    }
  })();
}
},
navigateTo: async (targetNodeId: string) => {
  const { currentScreen } = get();
  if (!currentScreen) return;
  if (!navigationManager.validateTransition(currentScreen, targetNodeId)) {
    console.error(`Invalid transition: ${currentScreen} -> ${targetNodeId}`);
    return;
  }
  set({ screenState: 'SHOW_COMPLETE' });
  navigationManager.pushHistory(targetNodeId);
  const cached = assetsManager.isAssetCached(targetNodeId);
  const navigationIndex = get().metrics.totalNavigations + 1;
  const startTime = Date.now();
  set({
    currentScreen: targetNodeId,
    screenState: 'LOADING',
    loadingProgress: 0
  });
  await assetsManager.downloadAsset(targetNodeId, {
    trigger: 'navigation',
    navigationIndex,
    isBlockingForPlayer: true,
    onProgress: (p) => set({ loadingProgress: p })
  });
  const duration = Date.now() - startTime;

```

```

set((state) => ({
  metrics: {
    ...state.metrics,
    totalNavigations: state.metrics.totalNavigations + 1,
    cacheHits: state.metrics.cacheHits + (cached ? 1 : 0),
    cacheMisses: state.metrics.cacheMisses + (cached ? 0 : 1),
    averageLoadTime: (state.metrics.averageLoadTime
state.metrics.totalNavigations + duration) / (state.metrics.totalNavigations + 1)
    }
  }));
loggerService.setCurrentNavigationIndex(navigationIndex);
loggerService.logNavigation(currentScreen, targetNodeId, duration, cached);
set({
  screenState: 'SHOW',
  loadingProgress: 100
});
get().completeShow();
},
goBack: () => {
},
setPreloadMode: (mode: PreloadMode) => {
  localStorage.setItem('preload_mode', mode);
  localStorage.setItem('preload_enabled', String(mode === 'predictive'));
  window.location.reload();
},
completeShow: () => {
  const { currentScreen, preloadMode } = get();
  if (currentScreen && preloadMode === 'predictive') {

```

```
    preloadManager.onShowComplete(currentScreen);
  }
}
}});
```

```
// ===== FILE: src/types/index.ts =====
```

```
export interface Node {
  assets: { size: string };
  transitions: string[];
}

export interface GraphConfig {
  nodes: Record<string, Node>;
}

export interface PreloadConfig {
  nodes: Record<string, {
    transitions: Array<{
      id: string;
      probability: number;
    }>;
  }>;
}

export interface NavigationLog {
  timestamp: number;
  fromNode: string;
  toNode: string;
  assetLoadDuration: number;
  wasAssetCached: boolean;
}
```

```

export type PreloadMode = 'on_demand' | 'predictive' | 'predownload_all';

export type AssetLoadTrigger = 'init' | 'navigation' | 'preload' | 'predownload_all' |
'unknown';

export interface AssetDownloadLogLine {
  timestamp: number;
  nodeId: string;
  trigger: AssetLoadTrigger;
  navigationIndex: number;
  wasCacheHit: boolean;
  playerLoadDurationMs: number;
  totalMegabytesDownloaded: number;
  assetSizeBytes: number;
}

export type ScreenState = 'IDLE' | 'LOADING' | 'SHOW' | 'SHOW_COMPLETE';
export interface Metrics {
  totalNavigations: number;
  cacheHits: number;
  cacheMisses: number;
  averageLoadTime: number;
  preloadAccuracy: number;
  totalMegabytesDownloaded: number;
}

// ===== FILE: src/ui/App.tsx =====
import React, { useEffect } from 'react';

```

```

import { useAppStore } from '../store/useAppStore';
import { Screen } from './Screen';
import { ControlPanel } from './ControlPanel';
import { AssetStatusPanel } from './AssetStatusPanel';
export const App: React.FC = () => {
  const { initApp, currentScreen } = useAppStore();
  useEffect(() => {
    initApp();
  }, []);
  if (!currentScreen) {
    return <div style={{ padding: '20px' }}>Initializing Application...</div>;
  }
  return (
    <div style={{ fontFamily: 'Arial, sans-serif', minHeight: '100vh', padding: '40px' }}>
      <Screen />
      <ControlPanel />
      <AssetStatusPanel />
    </div>
  );
};

```

// ===== FILE: src/ui/AssetStatusPanel.tsx =====

```

import React, { useEffect, useState } from 'react';
import { graphManager } from '../core/GraphManager';
import { assetsManager } from '../core/AssetsManager';
import { useAppStore } from '../store/useAppStore';
interface AssetState {
  id: string;

```

```

    status: 'Not Loaded' | 'Loading' | 'Loaded';
  }
export const AssetStatusPanel: React.FC = () => {
  const [assets, setAssets] = useState<AssetState[]>([]);
  const { metrics } = useAppStore();
  useEffect(() => {
    const updateAssets = () => {
      const allNodes = graphManager.getAllNodeIds();
      const cached = assetsManager.getCacheStatus();
      const nextAssets: AssetState[] = allNodes.map(nodeId => {
        let status: AssetState['status'] = 'Not Loaded';
        if (cached.has(nodeId)) {
          status = 'Loaded';
        } else if (assetsManager.isDownloading(nodeId)) {
          status = 'Loading';
        }
        return { id: nodeId, status };
      });
      setAssets(nextAssets);
    };
    updateAssets();
    const interval = setInterval(updateAssets, 500);
    return () => clearInterval(interval);
  }, [metrics]);
  const sortedAssets = [...assets].sort((a, b) => a.id.localeCompare(b.id));
  return (
    <div style={ {
      position: 'fixed',

```



```
bottom: '20px',
right: '20px',
width: '300px',
height: '300px',
backgroundColor: 'white',
border: '1px solid #ccc',
borderRadius: '8px',
display: 'flex',
flexDirection: 'column',
boxShadow: '0 -2px 10px rgba(0,0,0,0.1)',
fontSize: '12px',
zIndex: 1000
}}>
```

```
<div style={{
  padding: '10px',
  borderBottom: '1px solid #eee',
  fontWeight: 'bold',
  backgroundColor: '#f8f9fa',
  borderTopLeftRadius: '8px',
  borderTopRightRadius: '8px'
}}>
```

Assets Status

```
</div>
```

```
<div style={{
  flex: 1,
  overflowY: 'auto',
  padding: '10px'
}}>
```

```

    {sortedAssets.map(asset => {
      let color = '#666';
      if (asset.status === 'Loaded') color = 'green';
      if (asset.status === 'Loading') color = 'orange';
      return (
        <div key={asset.id} style={{
          display: 'flex',
          justifyContent: 'space-between',
          marginBottom: '5px',
          padding: '4px',
          borderBottom: '1px solid #f0f0f0'
        }}>
          <span>{asset.id}</span>
          <span style={{ color, fontWeight: 'bold' }}>{asset.status}</span>
        </div>
      );
    })}
  </div>
</div>
);
};

```

// ===== FILE: src/ui/ControlPanel.tsx =====

```

import React from 'react';
import { useAppStore } from '../store/useAppStore';
import { loggerService } from '../services/LoggerService';

```

```

function formatMB(mb: number): string {

```

```

if (!Number.isFinite(mb) || mb <= 0) return '0MB';
return `${mb.toFixed(mb < 10 ? 2 : mb < 100 ? 1 : 0)}MB`;
}

export const ControlPanel: React.FC = () => {
  const { preloadMode, setPreloadMode, metrics } = useAppStore();
  return (
    <div style={{
      position: 'fixed',
      top: '20px',
      right: '20px',
      width: '300px',
      padding: '15px',
      backgroundColor: '#f8f9fa',
      border: '1px solid #dee2e6',
      borderRadius: '8px',
      fontSize: '14px',
      boxShadow: '0 2px 10px rgba(0,0,0,0.1)'
    }}>
      <h3 style={{ marginTop: 0 }}>Control Panel</h3>
      <div style={{ marginBottom: '15px' }}>
        <div style={{ display: 'grid', gridTemplateColumns: '1fr', gap: '6px' }}>
          <strong>Mode</strong>
          <select
            value={preloadMode}
            onChange={(e) => setPreloadMode(e.target.value as 'on_demand' | 'predictive'
| 'predownload_all')}
            style={{
              padding: '6px 8px',

```

```

        borderRadius: '6px',
        border: '1px solid #ced4da',
        background: 'white'
    }}
>
    <option value="on_demand">Download on demand (default)</option>
    <option value="predictive">Predictive preload</option>
    <option value="predownload_all">Predownload all</option>
</select>
<div style={{ fontSize: '12px', color: '#6c757d' }}>
    Changing mode reloads the page.
</div>
</div>
</div>
<hr style={{ border: 'none', borderTop: '1px solid #dee2e6', margin: '15px 0' }} />
<div>
    <h4 style={{ margin: '0 0 10px 0' }}>Metrics</h4>
    <div style={{ display: 'grid', gridTemplateColumns: '1fr 1fr', gap: '8px' }}>
        <div>Navigations:</div>
        <div style={{ fontWeight: 'bold' }}>{metrics.totalNavigations}</div>
        <div>Cache Hits:</div>
        <div style={{ color: 'green', fontWeight: 'bold' }}>{metrics.cacheHits}</div>
        <div>Cache Misses:</div>
        <div style={{ color: 'red', fontWeight: 'bold' }}>{metrics.cacheMisses}</div>
        <div>Avg Load Time:</div>
        <div
            style={{
                fontWeight:
                    'bold'
            }}>{Math.round(metrics.averageLoadTime)}ms</div>
        <div>Total megabytes downloaded:</div>

```

```

        <div style={{ fontWeight: 'bold'
    }}>{formatMB(metrics.totalMegabytesDownloaded)}</div>
  </div>
  <div style={{ marginTop: '12px', display: 'flex', gap: '8px' }}>
    <button
      type="button"
      onClick={() => loggerService.downloadAssetLogFile()}
      style={{
        padding: '6px 10px',
        borderRadius: '6px',
        border: '1px solid #ced4da',
        background: 'white',
        cursor: 'pointer'
      }}
    >
      Download logs
    </button>
  </div>
</div>
</div>
</div>
);
};

// ===== FILE: src/ui/LoadingIndicator.tsx =====
import React from 'react';
interface Props {
  progress: number;
}

```

```

export const LoadingIndicator: React.FC<Props> = ({ progress }) => {
  return (
    <div style={{ width: '100%', padding: '20px', textAlign: 'center' }}>
      <h3>Loading Assets...</h3>
      <div style={{
        width: '90%',
        height: '20px',
        backgroundColor: '#eee',
        borderRadius: '10px',
        overflow: 'hidden',
        border: '1px solid #ccc'
      }}>
        <div style={{
          width: `${progress}%`,
          height: '100%',
          backgroundColor: '#4CAF50',
          transition: 'width 0.1s ease-in-out'
        }} />
      </div>
      <p>{Math.round(progress)}%</p>
    </div>
  );
};

// ===== FILE: src/ui/Screen.tsx =====
import React from 'react';
import { useAppStore } from '../store/useAppStore';
import { graphManager } from '../core/GraphManager';

```

```

import { LoadingIndicator } from './LoadingIndicator';
export const Screen: React.FC = () => {
  const { currentScreen, screenState, loadingProgress, navigateTo } = useAppStore();
  if (!currentScreen) return <div>Initializing...</div>;
  const node = graphManager.getNode(currentScreen);
  if (!node) return <div>Error: Node {currentScreen} not found</div>;
  const isLoading = screenState === 'LOADING';
  return (
    <div style={{
      padding: '20px',
      border: '1px solid #ddd',
      borderRadius: '8px',
      maxWidth: '600px',
      margin: '0 auto',
      backgroundColor: '#fff',
      boxShadow: '0 2px 4px rgba(0,0,0,0.1)'
    }}>
      <div style={{ borderBottom: '1px solid #eee', marginBottom: '20px',
paddingBottom: '10px' }}>
        <h1 style={{ margin: 0 }}>{currentScreen}</h1>
        <span style={{ color: '#666', fontSize: '0.9em' }}>
          Assets Size: {node.assets.size}
        </span>
      </div>
      {isLoading ? (
        <LoadingIndicator progress={loadingProgress} />
      ) : (
        <div style={{ display: 'flex', flexDirection: 'column', gap: '10px' }}>

```

```

<h3>Navigate to:</h3>

<div style={{ display: 'flex', flexWrap: 'wrap', gap: '10px' }}>
  {node.transitions.map(targetId => (
    <button
      key={targetId}
      onClick={() => navigateTo(targetId)}
      style={{
        padding: '10px 20px',
        fontSize: '16px',
        cursor: 'pointer',
        backgroundColor: '#007bff',
        color: 'white',
        border: 'none',
        borderRadius: '4px'
      }}
    >
      {targetId}
    </button>
  ))}
</div>
</div>
)}
</div>
);
};

```

```

// ===== FILE: src/utils/assetSizeParser.ts =====
export function parseAssetSize(sizeStr: string): number {

```



```

    if (!sizeStr) return 0;
    const match = sizeStr.toLowerCase().match(/(\d+)(\s*mb)?/);
    if (match && match[1]) {
        return parseInt(match[1], 10);
    }
    return 0;
}

// ===== FILE: src/utils/downloadSimulator.ts =====
export async function simulateDownload(
    sizeInMB: number,
    onProgress: (p: number) => void,
    signal?: AbortSignal
): Promise<void> {
    const msPerMB = 100;
    const totalTime = sizeInMB * msPerMB;
    const stepTime = 50;
    const steps = Math.ceil(totalTime / stepTime);
    onProgress(0);
    for (let i = 1; i <= steps; i++) {
        if (signal?.aborted) {
            throw new Error('Download aborted');
        }
        await new Promise(resolve => setTimeout(resolve, stepTime));
        const progress = Math.min(100, (i / steps) * 100);
        onProgress(progress);
    }
    onProgress(100);
}

```

}