Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Master's Thesis

Scalable Mobile Game Architecture: Design and Development

Volodymyr Pukha

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Bc. Volodymyr Pukha

Informatics

Thesis title

Scalable Mobile Game Architecture: Design and Development

Objectives of thesis

The main objective of the thesis is to design and develop a scalable architecture for a mobile game that effectively accommodates the dynamic requirements of modern mobile gaming.
The secondary objective is optimization and performance tuning: Apply optimization techniques to fine-tune the performance of the architectural prototype. Measure and compare the performance metrics between the initial prototype and the optimized version to assess the effectiveness of the applied strategies.

Methodology

The theoretical part of this thesis begins with an extensive literature review to gather information on scalable mobile game architecture design and development. Professional and scientific sources, including research papers, academic journals, and industry publications, will be studied to gain insights into various approaches, design patterns, and best practices related to mobile game architecture.

The practical part of this thesis focuses on designing and developing a scalable architecture for a mobile game. The prototype aims to provide a robust foundation capable of accommodating the dynamic requirements of modern mobile gaming. Selected design patterns, based on the findings from the theoretical analysis, are integrated into the prototype to address common mobile game development challenges.

The proposed extent of the thesis

60-80 pages

Keywords

Architecture, Design patterns, Game development, Mobile, Unity, Programming, Optimization

Recommended information sources

ALLS, Jason. Clean Code in C#: Refactor your legacy C# code base and improve application performance by applying best practices. Birmingham: Packt Publishing, 2020. ISBN 978-1838982973

BARON, David. Game Development Patterns with Unity 2021: Explore practical game development using software design patterns and best practices in Unity and C#. Birmingham: Packt Publishing, 2021. ISBN 978-1800200814

FOWLER, Martin. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley Professional, 2018. ISBN 978-0134757599

MARTIN, Robert. Clean Architecture: A Craftsman's Guide to Software Structure and Design. London: Pearson, 2017. ISBN 978-0134494166

NYSTROM, Robert. Game Programming Patterns. Genever Benning, 2014. ISBN 978-0990582908

Expected date of thesis defence

2023/24 SS – PEF

The Diploma Thesis Supervisor

Ing. Jiří Brožek, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 4. 9. 2023	Electronic approval: 3. 11. 2023
Ing. Martin Pelikán, Ph.D.	**doc. Ing. Tomáš Šubrt, Ph.D.**
Head of department	Dean

Prague on 28. 03. 2024

Declaration

I declare that I have worked on my master's thesis titled "Scalable Mobile Game Architecture: Design and Development" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the master's thesis, I declare that the thesis does not break any copyrights.

In Prague on 28.03.2024 ____________________________

Acknowledgement

I would like to thank Ing. Jiří Brožek, Ph.D. for his advice and support during my work on this thesis.

Scalable Mobile Game Architecture: Design and Development

Abstract

The thesis provides a comprehensive study on the design and development a scalable architecture for mobile games, focusing on performance optimization and adaptability to address the changing needs of the gaming sector. The research process combines a thorough literature review with the practical creation and execution of a prototype to discover and include the most efficient design patterns and optimization strategies. The results show substantial enhancements in performance indicators, demonstrating the efficiency of the suggested architecture in accommodating the dynamic and resource-intensive characteristics of contemporary mobile games. This work enhances the scholarly and practical comprehension of mobile game creation, providing useful insights for future research and implementation in the sector.

Keywords: Architecture, Design patterns, Game development, Mobile, Unity, Programming, Optimization

Škálovatelná architektura mobilních her: Návrh a vývoj

Abstrakt

Disertační práce představuje komplexní studii o návrhu a vývoji škálovatelné architektury pro mobilní hry, zaměřující se na optimalizaci výkonu a adaptabilitu k řešení měnících se potřeb herního sektoru. Výzkumný proces kombinuje důkladný přehled literatury s praktickým vytvořením a provedením prototypu, aby objevil a zahrnul nejefektivnější vzory návrhu a strategie optimalizace. Výsledky ukazují podstatné zlepšení výkonnostních ukazatelů, což dokazuje efektivitu navrhované architektury ve smyslu akomodace dynamických a náročných charakteristik současných mobilních her. Tato práce posiluje akademické a praktické porozumění tvorbě mobilních her, poskytující užitečné vhledy pro budoucí výzkum a implementaci v sektoru.

Klíčová slova: Architektura, Návrhové vzory, Vývoj her, Mobilní zařízení, Unity, Programování, Optimalizace

Table of content

List of pictures and tables

List of pictures

List of tables

1 Introduction

In the rapidly evolving field of mobile gaming, the search for scalable and efficient game architecture is more important than ever. Developers must balance maximizing efficiency and maintaining scalability to accommodate a broad and expanding user base as mobile games become more complex and engaging. The rise of sophisticated mobile technology and the growing expectations of contemporary gamers require a reassessment of conventional game development approaches, highlighting the importance of studying the design and creation of adaptable mobile game structures.

This thesis focuses on creating a scalable architecture for a mobile game to meet the evolving needs of contemporary mobile gaming. Due to the inherent lower power of mobile devices compared to PCs, optimizing games for these platforms is a major challenge. The difference in computational capacity impacts performance and user experience, requiring a careful focus on architecture design and development that emphasizes efficiency and scalability.

This project aims to provide a scalable mobile game architecture that addresses current industry expectations and anticipates future trends and obstacles. The thesis explores optimization and performance tuning by implementing different strategies to enhance the performance of the architectural prototype. This study aims to confirm the effectiveness of the suggested approaches by systematically comparing performance measures before and post-optimization. It provides useful insights into the optimization procedures crucial for mobile gaming.

The thesis starts with a systematic process, doing extensive literature research using a wide range of professional and scientific sources to provide a strong theoretical basis. This research encompasses scalable architectures, design patterns, and optimization strategies, offering a thorough review of current knowledge and practices in the topic. The thesis's practical component shifts from theory to application, explaining the design and development process of a scalable mobile game architecture. The prototype incorporates chosen design patterns from the literature analysis to tackle typical development obstacles, with a focus on enhancing efficiency and scalability.

This research is important for both the academic community and the mobile game industry due to its practical consequences. This thesis attempts to provide a blueprint for scalable mobile game architecture that bridges theoretical frameworks with real-world applications. It empowers developers to construct technologically advanced games that are universally accessible and fun.

This thesis aims to provide a foundation for future research and development in the mobile gaming sector, with the goal of promoting creativity and excellence in designing and optimizing mobile game infrastructures.

2 Objectives and Methodology

2.1 Objectives

The main objective of the thesis is to design and develop a scalable architecture for a mobile game that effectively accommodates the dynamic requirements of modern mobile gaming.

The secondary objective is optimization and performance tuning: Apply optimization techniques to fine-tune the performance of the architectural prototype. Measure and compare the performance metrics between the initial prototype and the optimized version to assess the effectiveness of the applied strategies.

2.2 Methodology

The theoretical part of this thesis begins with an extensive literature review to gather information on scalable mobile game architecture design and development. Professional and scientific sources, including research papers, academic journals, and industry publications, will be studied to gain insights into various approaches, design patterns, and best practices related to mobile game architecture.

The practical part of this thesis focuses on designing and developing scalable architecture for a mobile game. The prototype aims to provide a robust foundation capable of accommodating the dynamic requirements of modern mobile gaming. Selected design patterns, based on the findings from the theoretical analysis, are integrated into the prototype to address common mobile game development challenges.

3 Literature Review

3.1 Fundamentals of Mobile Game Architecture

A mobile game's architecture is the basic framework that determines its structure. It incorporates software techniques and patterns to handle the constraints of game creation and execution on mobile platforms. The architecture of a mobile game describes the structure of the game. The architecture needs to be properly planned in order to improve performance and create a smooth user experience. To build a system with a design and an architecture that minimizes effort and maximize productivity, you need to know which attributes of system architecture lead to that end (Martin, 2017, p. 12). This is because mobile devices have a number of limits and unique characteristics, such as restricted processing power, memory, and screen size. Additionally, there is a requirement for optimal battery management.

3.1.1 Overview of mobile game architecture components

Mobile game architecture comprises several primary components, each fulfilling a distinct function within the overall system. The components are named as follows:

Game Engine

The game engine is the core component of mobile game development. It is responsible for supplying the essential functions needed for game dynamics, including visual rendering, physics computation, audio management, and more. Unity and Unreal Engine are popular due to their comprehensive feature sets and cross-platform compatibility. Unity has been a popular alternative for mobile game makers because of its user-friendly UI, wide asset store, and helpful community. Its versatility on many mobile platforms and capability to sustain excellent performance on a wide array of devices enhance its popularity. Unity dominates the market for mobile gaming, powering the bulk of current games. It powers over 69% of the top mobile games (Unity, 2024). The engine prioritizes mobile development with frequent updates and features tailored to improve mobile gaming experiences.

Client-Server Architecture

Several mobile games utilize a client-server design, especially those that rely on online multiplayer features. This particularly applies to games created for mobile devices. The game is divided into client-side logic, administered by the player's mobile device, and server-side processes, which handle game state management and multiplayer matchmaking due to this structure. This architecture guarantees the game's integrity and facilitates real-time communication among players.

Data Storage and Management

Efficient data storage and management are essential for monitoring game progress, player profiles, in-game assets, and other dynamic information. Efficient data storage and management solutions that combine speed, reliability, and scalability need to be incorporated into the design. Securing and managing data in mobile game development poses distinctive difficulties and possibilities. Developers can choose from different choices for data storage and security, such as local storage on the mobile device, custom-built solutions, or ready-made backend solutions provided as Software as a Service (SaaS).

Local storage solutions provide fast access and offline gaming options, but they might create worries about data security and device storage constraints. Custom solutions provide personalized security and data management capabilities to meet individual gaming needs, but usually demand more development and maintenance resources. SaaS backend solutions offer scalable, secure, and reliable data management platforms that reduce developers' workload by utilizing cloud-based infrastructures optimized for high availability and performance. These services typically include complex features like user authentication, leaderboards, social integration, and more, enhancing the game experience and making it more convenient for developers.

When selecting a data management method, variables to consider include the game's scale, data sensitivity, user expectations, and the development team's ability to deploy and maintain the chosen solution.

Networking

Networking components are crucial in modern mobile gaming, expanding beyond online multiplayer gameplay to include various internet-based functions. Efficient network management is essential for ensuring smooth online interactions, including social features, and enabling frequent content upgrades to improve the overall player experience.

Developers use advanced algorithms and networking protocols to optimize data paths and eliminate lag in competitive online multiplayer games, ensuring minimal latency that can impact the outcome of interactions. Data compression methods are used to reduce bandwidth use, allowing for quicker data transfer speeds even on restricted network connections. It is crucial for both gaming and the effective distribution of updates and new material to users, minimizing loading times and data expenses.

Ensuring secure data transfer is a crucial element of networking in mobile gaming. Players must use strong encryption technologies and secure communication protocols to protect their personal information and digital assets. This guarantees that player data, including as progress, in-game purchases, and personal information, is safeguarded against unauthorized access and any security breaches.

User Interface and User Experience (UI/UX)

Furthermore, incorporating flexible UI/UX design methods ensures a uniform gaming experience on various platforms, such as smartphones and tablets, to accommodate the varied tastes and gaming styles of the mobile gaming community. This requires adjusting to various screen sizes and considering the ergonomic characteristics of touch-based interactions to ensure that game controls are easily accessible and that UI elements do not block important gameplay visuals. The objective is to make a game that is visually attractive, functionally efficient, and pleasant on all device platforms, emphasizing the significance of UI/UX in the effective design and creation of mobile games.

3.1.2 Challenges in mobile game development

Designing mobile game architecture requires careful consideration of several factors to ensure scalability, maintainability, and performance:

Scalability

The architecture should be scalable to accommodate fluctuating loads, ranging from a few users to millions of concurrent gamers, while maintaining speed. This includes expandable server infrastructure, effective resource allocation, and dynamic content distribution systems. The architecture of a system is defined by the boundaries drawn within that system, and by the dependencies that close those boundaries (Martin, 2017, p. 247).

Performance Optimization

Optimizing performance is crucial in mobile game development due to the hardware limitations often found in mobile devices. To optimize game performance, it is necessary to minimize memory usage, improve graphics and animations for mobile GPUs, and provide smooth gameplay at various frame rates. This is a thorough strategy that includes effective asset management, code optimization, and implementing performance-focused design concepts.

Effective asset management involves optimizing texture sizes, using asset compression, and applying level of detail (LOD) approaches to guarantee gaming assets are scaled and rendered properly without straining the device's memory and processing power. Implementing code optimization techniques including reducing redundant calculations, utilizing data caching, and implementing multi-threading can greatly decrease CPU usage and improve game responsiveness.

Cross-Platform Support

Having Cross-Platform Support is crucial for adapting to different mobile operating systems and hardware configurations, allowing games to be launched on several devices and platforms with few modifications. This method expands the possible game audience by allowing access from various devices and simplifies the development and upkeep procedures for creators. Developers can utilize cross-platform development tools like Unity or Unreal Engine to write code once and release it on several platforms like as iOS, Android, desktop, and web with minimal platform-specific adjustments.

Implementing a cross-platform strategy can greatly decrease development time and expenses by removing the necessity of managing distinct codebases for each platform. It also guarantees a uniform game experience on all platforms, which is essential for sustaining user interest and happiness. Cross-platform support simplifies the process of updating and fixing bugs by allowing modifications to be implemented once and then distributed across all platforms, guaranteeing that all players can access the most recent version of the game.

Security and Privacy

Emphasizing security and privacy is crucial in mobile game development to protect user data and prevent cheating or hacking. The design must incorporate security measures like as encryption, secure APIs, and cheat detection tools. Implementing these procedures guarantees the authenticity of the game environment and safeguards sensitive player information, including personal details and payment information.

Regular security evaluations and upgrades are essential to address emerging threats and vulnerabilities, in addition to core security policies. Utilizing proactive security measures such as penetration testing, code reviews, and advanced threat detection systems can help identify and address potential security vulnerabilities before they are exploited.

Monetization Strategies

Monetization strategies are essential in mobile game development, requiring infrastructure that can accommodate different approaches like in-app purchases, adverts, and subscription models. It is crucial that these techniques are seamlessly integrated into the gaming experience without diminishing user satisfaction. Attaining this equilibrium necessitates a strategic incorporation of monetization components that seamlessly blend in and enhance the player's experience.

In-app purchases can be created as improvements or accelerators that provide more value, like unique products, characters, or levels. Advertisements can be a profitable source of income when used carefully, particularly if opt-in benefits are offered to give users control over their watching experience. Subscription models must consistently provide value to justify the continual investment from users for premium content or services offered on a recurring basis.

3.1.3 Development Scalability

When talking about the scalability of mobile game architecture, it's crucial to address two main dimensions: performance scalability and development scalability. The former assesses the system's capacity to manage growing workloads without sacrificing efficiency, while the latter emphasizes the architecture's adaptability for growth, facilitating the integration of new features, content, and enhancements over time.

An essential element of a well-structured mobile game architecture is its capacity to adapt as the game progresses through its lifecycle. This implies that the architecture is capable of supporting the existing game elements and is also resilient and adaptable to incorporate future extensions. The measure of a design is how easily it accommodates changes (Nystrom, 2014, p. 19). Scalability in development is essential for the following reasons:

Feature Integration

Straightforward Features Integration in a scalable architecture greatly improves the game development process by enabling the smooth addition of new levels, characters, game modes, or completely new gaming systems. Adaptability is essential for maintaining the game's novelty and interest for players in the long run. Developers may efficiently construct and incorporate new features into the game by using a modular and adaptable architecture, allowing for the game to change and grow in response to user feedback and market trends. This method allows for the ongoing expansion of the game and provides developers with the opportunity to explore new ideas and gameplay mechanisms, enhancing the player experience without requiring significant changes to the current game framework. The ease and efficiency of adding features demonstrate the foresight in architectural design, highlighting the significance of scalability and modularity in the always changing field of game production.

Rapid Prototyping and Iteration

Mobile games frequently go through rapid prototyping and iterations in response to user input and market trends. An adaptable development framework allows for rapid prototyping and testing of new concepts, which enhances the agility of the development process. The faster you can try out ideas and see how they feel, the more you can try and the more likely you are to find something great (Nystrom, 2014, p. 24).

Maintainability

As games become more sophisticated, it becomes more difficult to maintain code quality and manageability. Scalability-focused architecture facilitates the organization of code and assets for improved manageability, boosting the efficiency of development and issue repairs.

It is crucial to include development scalability in the mobile game architecture from the beginning. The game is designed to endure over time by maintaining user interest, handling server traffic, and easily incorporating new material and features to stay dynamic and lively. This strategy supports a sustainable development model by continuously updating the game with new and interesting material to keep attracting players, thereby increasing its longevity and profitability.

Architects and developers may secure the success of your mobile game in the competitive gaming scene by focusing on scalability in performance and development to adapt to changing technology, player expectations, and market dynamics.

3.2 Scalable Architectures for Mobile Games

Focusing on development scalability, designs that facilitate the continuous growth and advancement of mobile games by enabling easy addition of features and expansion of the codebase are emphasized. This viewpoint is essential for achieving long-term success and flexibility in the rapidly changing mobile gaming sector.

3.2.1 Principles of scalable software architecture

Software architecture development scalability focuses on building systems that are flexible, sustainable, and able to grow with ease. Key principles consist of:

Modularity

Game architecture modularity entails organizing the game into distinct, loosely linked modules that may be built, tested, and adjusted independently. Small modules are easy to test, are more readily reused, and are easier to extend and maintain (Alls, 2020, p. 33). This design idea greatly improves the development process by enabling teams to work on many game aspects simultaneously without any hindrance. It allows for easy incorporation or modification of functionality, as alterations in one module rarely impact the entire system. Game creators can enhance flexibility and efficiency by adopting modularity, allowing for faster iterations and changes to the game while preserving a strong and stable core architecture.

Extensibility

Extensibility refers to how easy it is to extend an application by adding new features to it (Alls, 2020, p. 359). Extensibility, facilitated via interfaces, plugins, or extension points in the architecture, allows the game to expand and develop over time. Developers can incorporate new features or material, including more levels, characters, or gameplay mechanisms, into the current codebase with minimal disruption using this method.

Extensibility ensures the game architecture remains relevant in the future and promotes creativity by offering a versatile structure for expanding the game's features.

Reusability

Reusability involves utilizing pre-existing components or services in different sections of a game or across many projects. This method accelerates the creation of new features by eliminating repetition and enhances the uniformity and dependability of the game's code. Promoting reusability guarantees that thoroughly tested and established components improve the game's overall quality, decreasing the chances of faults and errors. Furthermore, the capacity to reuse components can greatly diminish development expenses and duration, rendering it a crucial concept in game design.

Configurability

Configurability in game architecture enables creators to customize game behavior or appearance using external configuration files or parameters, rather than fixed defaults. This flexibility simplifies the process of updating game features, modifying gameplay parameters, or adding new content without changing the game's fundamental code. Configurability improves the game's ability to adjust to various user preferences and market demands, promoting continued engagement and pleasure.

Simplicity

Ensuring simplicity in game architecture is crucial for making the system easy to comprehend, create, and upkeep. Simplifying the design decreases complexity, facilitating the addition of new features by developers or the quick integration of new team members. Emphasizing simplicity prevents excessive complexity and ensures attention is directed towards providing value to the end-user by creating a highly functional game. Streamlining the architecture enhances development cycles and results in a more resilient and organized codebase.

Automation

Automation in game development involves utilizing tools and procedures for automated testing, constructing, and releasing. Practicing this is essential for upholding code quality and game stability by detecting and resolving regressions caused by new code or additions. Automation facilitates seamless integration and delivery, enabling teams to distribute updates and new content quickly and dependably. Automation decreases manual involvement, hence lowering the possibility of human error and guaranteeing that every release adheres to the game's quality criteria.

3.2.2 Importance of scalability in mobile gaming

Within the realm of mobile gaming, the importance of development scalability is heightened by various industry-specific aspects.

Rapid Evolution

The mobile gaming market undergoes quick changes due to developments in technology, changing client preferences, and competitive forces. Scalable designs are essential in a rapidly changing environment, allowing developers to efficiently make changes to their games by adding new features, improvements, and optimizations based on market trends and user input. This mobility enables games to stay current and competitive, ensuring they meet or are beyond player expectations. Adapting quickly to new technical advancements and changes in user behavior is crucial for maintaining growth and establishing a strong market position.

User Engagement

Sustaining user engagement necessitates regularly introducing new content, novel gameplay mechanics, and improved social features to encourage community involvement. An adaptable and versatile development framework is crucial for facilitating the prompt execution of updates and expansions. This versatility guarantees that games may develop alongside their audience, offering players fresh challenges, rewards, and opportunities for interaction. This continuous revitalization of the gaming environment helps maintain current players and draw in new ones, enhancing the overall success and durability of the game.

Global Market Adaptation

Scalability in development is important for adjusting mobile games for international markets. Various regions may exhibit different preferences for gaming genres, revenue mechanisms, and social interaction elements. Utilizing a scalable development method enables the adaptation of content, language, and cultural allusions to cater to different audiences, thus improving the game's attractiveness and accessibility on a global scale. Having a global viewpoint broadens the possible user base and fosters a more inclusive gaming culture, allowing gamers from various backgrounds to connect with material that speaks to them.

Feature Diversity and Innovation

Developing scalability in mobile gaming is crucial for incorporating various features and unique gameplay components. Players are looking for distinctive and rewarding experiences. Scalable architectures enable developers to explore and include various elements, such as augmented reality (AR) interactions and intricate multiplayer systems. This flexibility promotes creative freedom and innovation, allowing games to distinguish themselves in a competitive market. Scalable development procedures facilitate rapid implementation of innovative features, decreasing the time from concept to deployment and allowing developers to take advantage of changing trends and technology.

Technical and Platform Evolution

As mobile devices continue to evolve, with new hardware capabilities and operating system updates, scalable development practices ensure that games can leverage these advancements. This includes optimizing for higher screen resolutions, better processors, and new input methods, as well as ensuring compatibility with future device generations. Scalability in development means games can continuously improve in terms of graphics, performance, and user experience, maintaining relevance and providing players with the best possible experience on the latest devices.

3.3 Design Patterns in Game Development

Design patterns are standardized solutions to typical issues in software engineering that have been modified to tackle the specific obstacles encountered in game development. Implementing these principles can greatly improve the scalability and maintainability of game systems, allowing developers to design more intricate, effective, and adaptable games.

3.3.1 Design Patterns examples

David Baron's book "Game Development Patterns with Unity 2021" discusses important design patterns that are specifically useful in game development. These include, but are not restricted to:

Command Pattern

The Command Pattern encapsulates actions or inputs as objects to provide flexible command queueing, undo operations, and mapping of user inputs to in-game activities.

Command pattern permits us to decouple the object that invokes the operation from the one that knows how to execute it (Baron, 2021, p. 79).

Component Pattern

The Component Pattern, sometimes referred to as Entity-Component-System or ECS, separates game entities' behavior and state by utilizing components that can be linked to entities. This approach simplifies the handling of intricate game state relationships and increases the adaptability of game object behavior. Components are basically plug-and-play for objects. They let us build complex entities with rich behavior by plugging different reusable component objects into sockets on the entity (Nystrom, 2014, p. 294).

Singleton Pattern

The Singleton Pattern guarantees that a class has only one instance. This mechanism can be helpful when you have a class that manages a system that needs to be globally accessible from a singular and consistent entry point (Baron, 2021, p. 37). It is frequently utilized for controlling game state, setups, and accessing resources across a system.

State Pattern

The State Pattern allows us to implement an entity's stateful behaviors as a collection of components that can be assigned dynamically to an object when it changes states (Baron, 2021, p. 59). In game development it is crucial for handling intricate game entities that might be in several states, each with unique characteristics.

Observer Pattern

The core purpose of the Observer Pattern is to establish a one-to-many relationship between objects in which one acts as the subject while the others take the role of observers (Baron, 2021, p. 107). It is beneficial for managing events and separating gaming systems.

Factory Pattern

The Factory Pattern defines an interface for creating objects and defers the responsibility of determining which class to instantiate to subclasses, enabling flexibility in object creation. This pattern is particularly advantageous in game development since it allows for the dynamic generation of game entities based on runtime conditions, hence enhancing the flexibility and scalability of the game structure. The Factory Pattern simplifies updates and improvements to the game by encapsulating the creation logic, allowing new classes to be added without changing the main creation process.

Object Pool

The Object Pool pattern is a design pattern that minimizes the overhead of creating and destroying objects in applications that need numerous instances of objects, particularly those that are costly to construct. The core concept of this pattern is simple—a pool in the form of a container holds a collection of initialized objects in memory (Baron, 2021, p. 95).

Service Locator

The Service Locator pattern is a design pattern that separates the interface from its implementation, enabling objects to access references to other objects, known as services, without requiring knowledge of how these objects are created or handled. The core idea of this pattern is straightforward: it revolves around having a central registry of initialized dependencies (Baron, 2021, p. 203). In game development, the Service Locator pattern is particularly useful for managing and accessing core game services such as audio, input, or state management systems.

3.3.2 Application of design patterns

Implementing these design principles in mobile game development can greatly enhance the scalability and maintainability of the game architecture.

Improving Scalability

Enhancing scalability in game development is essential to handle a growing number of entities, interactions, and intricate game mechanisms while maintaining performance. Implementing design patterns like the Component Pattern enables games to efficiently manage this expansion. Developers can integrate or modify game elements without affecting the fundamental game dynamics by isolating game logic from game state. This division allows for both horizontal and vertical growth of game elements, allowing developers to incorporate new content, improve current features, or introduce completely new gameplay mechanisms with minimal impact on the core system.

Enhancing Maintainability

Utilizing patterns like the Command and State patterns can help create a more structured and organized codebase that follows the single responsibility concept. This organization simplifies the codebase, making it easier to comprehend, troubleshoot, and expand, ultimately decreasing the expenses and labor involved in implementing new features or resolving issues. Maintainability refers to how easy it is to fix bugs and add new functionality (Alls, 2020, p. 359).

Fostering Reusability

To promote reusability, developers might abstract basic game functionality into patterns such as Factory or Singleton. This allows for code reuse in several projects or different sections of the same game, leading to faster development and maintaining consistency in game aspects. For most applications, maintainability is more important than reusability (Martin, 2017, p. 106).

Enhancing adaptability

Implementing design patterns in game architectures enables them to be more flexible in accommodating changes, such as new user needs, platform advancements, or the incorporation of new technology. Adaptability is essential for sustained game development due to the constant changes and upgrades.

Implementing these design patterns simplifies the development process and guarantees the game's ability to adapt and expand over time, addressing new difficulties and possibilities in the mobile gaming industry.

3.4 Clean Architecture and Code Quality

The sustainability of a mobile game, especially in terms of scalability and performance, is heavily determined by its underlying architecture and the quality of its code. This section delves into the crucial importance of clean architecture and code quality in creating scalable and high-performing mobile games.

3.4.1 Clean Architecture principles

Robert C. Martin's "Clean Architecture" offers a detailed manual on organizing software to prioritize maintainability, scalability, and efficiency. Key lessons from the book that are especially pertinent to mobile game development are:

Separation of Concerns

Separation of Concerns is crucial for developing a strong and sustainable game architecture. The idea of a layered architecture is built on the idea of programming to interfaces (Martin, 2017, p. 271). Developers assure a modular structure by separating the game's fundamental functionality from elements like the user interface, database operations, and external integrations. This allows components to be built, maintained, and updated separately. This architectural approach simplifies the development process by enabling teams to concentrate on certain regions without disruption and also improves the system's ability to adapt to changes. For example, making changes to the user interface or updating the database structure can be done with little danger to the game's core logic or overall performance. Moreover, this division streamlines the integration of third-party services and customization of the game for various platforms, enhancing the scalability and adaptability of the game production process.

Dependency Rule

Code dependencies should only be directed towards the main logic. Lower-level modules, including UI and data access layers, should not control the behavior of higher-level modules to maintain the game's fundamental functionality as separate and simple to test. Source code dependencies must point only inward, toward higher-level policies (Martin, 2017, p. 203).

Use of Interactors

Utilizing Interactors, also known as use cases, in game development highlights a methodical way of incorporating business rules and logic into the design. Interactors operate as intermediates between the user interface and the data model, encapsulating the essential functionalities and decisions of the game. This separation helps to disconnect the game's operating logic from its presentation and storage layers. This design improves the flexibility and portability of the game, facilitating modifications to game mechanics and adjustments for different platforms and devices. A modular design simplifies updating game features and mechanics without requiring extensive modifications to the user interface or underlying data structure, making the game development process more sustainable and scalable.

Principle of Least Knowledge

Following the Principle of Least Knowledge, sometimes called the Law of Demeter, significantly influences a system's architecture to enhance efficiency and reliability. By minimizing the interaction between components and ensuring they only communicate through well-defined interfaces, this principle greatly decreases the complexity and interdependence in the system. This design concept simplifies development by localizing changes to specified regions and also makes debugging and testing easier by limiting the breadth of impact. Reducing the connectivity between components significantly improves the modularity of the system, making maintenance easier and facilitating the integration of new features or technologies. This idea is crucial in game development for designing adaptable and strong game structures that can evolve without needing major changes, therefore speeding up development and cutting expenses.

3.4.2 Role of clean code in scalability and performance

Jason Alls' "Clean Code in C#" underscores the significance of crafting clean, comprehensible, and effective code, especially in the realm of C# programming. The techniques detailed in the book, although tailored to C#, are widely relevant to mobile game creation in several programming languages. The primary goal of coding standards and principles in C# is for programmers to become better at their craft by programming code that is more performant and easier to maintain (Alls, 2020, p. 7).

Readability

Code readability is crucial in software development, highlighting the importance of code being easily comprehensible by human readers in addition to being executable by machines. Emphasizing clarity in coding helps with both instant comprehension and long-term manageability of the game's codebase. Concise and legible code eases the onboarding of new team members, speeds up debugging, and streamlines the incorporation of new features or technologies. Developers can enhance collaborative efforts and streamline development workflows by adopting coding standards and practices that prioritize readability. This includes using meaningful naming conventions, comprehensive documentation, and maintaining a consistent code structure to ensure the codebase remains accessible and manageable.

Simplicity

The notion of simplicity in code design emphasizes that the most direct solutions are usually the most efficient. Developers can reduce code complexity, improve performance, and boost maintainability and scalability of the game by avoiding over-engineering and following the KISS (Keep It Simple, Stupid) approach. Streamlined code is faster to run, simpler to test, and easier to adjust, which enhances the development process efficiency and decreases the chance of defects or mistakes. Simplicity helps creators focus on innovation and originality by providing a clear knowledge of the game's design and logic, without needless complications.

Refactoring

Refactoring is essential for the sustainable evolution of software, enabling developers to consistently enhance the code structure and uphold its quality as time progresses. Refactoring involves systematically improving the internal structure of the code without altering its external behavior, assuring the game's adaptability and scalability. Consistent refactoring can help manage technical debt, enhance performance, and improve the code's modularity and reusability. This proactive code maintenance technique ensures the game's architecture can expand smoothly, allowing for the easy integration of new game mechanics, features, and upgrades while maintaining a healthy and strong codebase.

Testing

Testing, particularly automated testing, is a fundamental aspect of dependable software development, guaranteeing the stability and performance of the game as it expands and changes. Automated tests provide a safety measure for developers to modify code or add new features with assurance that any unexpected consequences would be quickly detected. Emphasizing testing encourages a culture of high quality and accuracy, minimizing the chances of problems appearing in the final product and establishing a strong basis for ongoing enhancements. By incorporating testing at various stages of the development process, such as unit, integration, system, and acceptance testing, developers may guarantee the game's reliability, performance, and enjoyment for players throughout its extended development and growth.

Emphasizing clean design and code quality is essential for creating scalable and high-performing mobile games, rather than just being theoretical concepts. Developers can create games that are more engaging for players and easier to maintain and extend by following the principles of clean architecture and creating clean code.

3.5 Code Standards and Methodologies in Game Development

The principles of SOLID, KISS, DRY, and YAGNI are fundamental for creating clean and maintainable code in software development.

SOLID is a collection of five design principles that focus on enhancing the clarity, adaptability, and maintainability of software systems. The principles it encompasses include:

1. **Single Responsibility Principle (SRP)** states that each class should have a single responsibility, meaning it should only have one duty or function.
2. **Open/Closed Principle (OCP)** states that software elements such as classes, modules, and functions should allow for extension without requiring modification.
3. **Liskov Substitution Principle (LSP)** states that objects of a superclass should be substitutable with objects of a subclass without impacting the program's correctness.
4. **Interface Segregation Principle (ISP)** states that clients should not be compelled to rely on techniques that they do not utilize. Specialized interfaces are preferable than a general-purpose interface.
5. **Dependency Inversion Principle (DIP**) states that high-level modules should not rely on low-level modules. Both should rely on abstract concepts. Abstractions should be independent of details, whereas details should rely on abstractions.

KISS (Keep It Simple, Stupid) advocates for simplicity in programming, stating that simpler code is easier to maintain, understand, and less prone to problems. The DRY principle stresses the need of eliminating code duplication to ease maintenance and minimize the risk of inconsistencies or problems.

YAGNI, short for "You Aren’t Going to Need It," is a principle advising developers to refrain from incorporating features until they are actually needed. This approach aids in maintaining a concise codebase that is aligned with present requirements.

These principles help developers create code that is efficient, less error-prone, and adaptive to change, in line with the goals of scalable and robust software architecture.

3.6 Improving Performance and Scalability through Refactoring

Improving the appearance of existing code without altering its functionality is a crucial aspect of software development. Optimizing performance and expanding functionality of mobile games is crucial to effectively manage increased user loading and incorporate new features.

3.6.1 Refactoring

Martin Fowler's influential research on refactoring lays strong groundwork for comprehending and applying refactoring techniques in software development. Here are the key concepts:

Code smells

Fowler presents the notion of "code smells" as signs of possible issues in the code that might require refactoring. Common code smells consist of duplications, lengthy methods, oversized classes, and an overabundance of global variables, all of which can impede performance and scalability.

Refactoring Techniques

The book elaborates on several refactoring strategies tailored to resolve specific challenges in the codebase. The strategies vary from basic adjustments like as renaming variables for clarity to more intricate changes like dividing huge classes into smaller, more coherent ones.

Refactoring to Patterns

Fowler supports using design patterns as focal points for reworking endeavors. Developers can enhance the system's design and its scalability by conforming the software to established patterns.

Continuous Refactoring

Emphasizing the importance of incorporating refactoring into the development process as a regular practice, as opposed to treating it as an isolated undertaking. Consistent refactoring facilitates upgrades and performance enhancements by enhancing the codebase's long-term health.

Testing and Refactoring

The importance of automated testing in the refactoring process is emphasized. Tests operate as a safeguard to maintain functionality while making changes to the code, enabling developers to concentrate on enhancing the code's organization and efficiency. The first step when doing refactoring is to ensure there is a solid set of tests for a particular section of code (Fowler, 2018, p. 9).

3.6.2 Techniques for optimizing game performance

Within mobile game development, many refactoring techniques can be especially utilized to enhance performance and scalability.

Performance Profiling

Profiling is the initial stage in performance optimization for identifying performance bottlenecks. Profiling tools can identify specific parts of the code that use a lot of resources, helping to focus refactoring efforts effectively.

Optimizing Data Structures and Algorithms

Optimizing by implementing more efficient data structures and algorithms can greatly enhance game performance. Programming involves writing a lot of code that implements behavior - but the strength of a program is really founded on its data structures (Fowler, 2018, p. 235). Substituting a linear search with a hash table lookup can save processing time, particularly in crucial performance areas of the game.

Reducing Memory Footprint

Memory usage is a crucial issue in mobile gaming. As you decompose the design, you give each component a budget for resources - time and footprint (Fowler, 2017, p. 77). Optimizing by decreasing memory usage, for example, by removing superfluous object allocations or implementing memory pools for regularly created and destroyed objects, can improve speed and decrease latency.

Parallelization and Asynchronous Processing

Utilizing multi-threading and asynchronous processing can enhance the responsiveness and scalability of mobile games. Implementing parallel processing for independent activities or asynchronous I/O operations can improve playability and optimize hardware resource usage.

Minimizing Render Calls

Rendering graphics frequently hinders game performance. Optimizing render calls by refactoring, including batching draw calls or improving shader usage, can greatly enhance rendering efficiency and frame rates.

These refactoring methods, based on Martin Fowler's ideas and customized for the unique obstacles of mobile game development, are crucial for creating scalable and high-performing games. Developers may guarantee that their games are both functionally robust and optimized for the optimal player experience by consistently evaluating and improving the codebase.

3.7 Challenges in Mobile Games Optimization

Primarily as a result of the hardware and environmental limitations of mobile devices, the creation and optimization of mobile games require particular attention. In order to produce games that are not only captivating but also function optimally on a diverse array of devices, it is imperative to confront these obstacles.

3.7.1 Performance Optimization for Mobile Devices

Despite the ongoing advancements in technology, personal computers still outperform mobile devices in critical areas such as memory, battery life, CPU and GPU capabilities. Optimizing for performance is a deep art that touches all aspects of software (Nystrom, 2014, p. 362). To ensure optimal performance, a meticulous approach to the design and development of games is necessary in light of these constraints.

Mobile devices, as a rule, are characterized by memory limitations, processing speed restrictions, and brief battery life. These limitations impact the degree of intricacy in game visuals, the length of gameplay sessions, and the complexity of games that can be created. Furthermore, extended periods of gameplay on mobile devices may lead to reduced performance as a consequence of thermal limiting. This, in turn, can have adverse effects on the game's responsiveness and visual quality.

In order to achieve optimal game performance on mobile devices, it is necessary to adopt a comprehensive strategy that considers various factors such as resource utilization, battery conservation, and ensuring a smooth user experience.

Efficient Resource Utilization

Performance can be maximized by developers through meticulous memory footprint management, GPU and CPU burden minimization, and efficient utilization of available resources. By reducing the quantity of data that must be processed and rendered, techniques such as level of detail (LOD) rendering, texture compression, and asset aggregation can significantly improve performance.

Reducing Battery Consumption

Ensuring battery life optimization is of utmost importance to prevent excessive depletion, which can negatively impact the device's functionality and user experience. It is possible to implement the following strategies: reduce the frequency of updates and background processes, optimize network utilization by aggregating data transfers, and utilize more energy-efficient rendering techniques. A game that runs beautifully but turns players' phones into space heaters before running out of juice thirty minutes later is not a game that makes people happy (Nystrom, 2014, p. 186).

Ensuring a Constant User Experience

To guarantee a seamless user experience on mobile devices, it is imperative to uphold consistent frame rates, minimize input latency, and optimize user interface elements to accommodate touch inputs. Dynamic resolution scaling and frame rate adjustment are techniques that developers may employ in order to ensure optimal performance on devices with diverse capabilities.

By capitalizing on the intrinsic constraints of mobile devices and implementing focused optimization tactics, programmers have the ability to produce mobile games that provide immersive, superior experiences while maintaining performance and usability standards intact. The success of mobile games in a fiercely competitive market, where user experience and efficacy are pivotal in player retention, is contingent upon these optimization efforts.

3.7.2 Adaptation to Diverse Hardware

The mobile gaming industry is distinguished by an extensive variety of device functionalities, spanning from entry-level smartphones to high-end tablets. The presence of such diversity poses considerable obstacles for developers who strive to deliver a uniform gaming experience across all devices. Processing speed, graphical capabilities, memory capacity, and screen resolution exhibit substantial variation among mobile devices. In order for all players to be able to appreciate the game, it is necessary to develop games that perform well across this spectrum, which necessitates careful consideration of these variations.

Methodologies for Graphics, Performance, and Game Feature Scaling

Various approaches are utilized by developers to scale game elements, visuals, and performance in accordance with the vast array of hardware specifications.

Dynamic Asset Scaling

Multiple sets of assets (textures, models, etc.) are implemented using dynamic asset scaling, which selects them in accordance with the capabilities of the device. Thus, memory and computational demands are reduced as devices with inferior specifications utilize assets with a lower resolution.

Adjustable Graphics Settings

Incorporating features that enable users to modify graphics quality settings directly within the game, thereby enabling users to discover a suitable compromise between visual accuracy and device performance. Mobile games are often more focused on the quality of gameplay than they are on maximizing the detail of the graphics. Many of these games will set an upper limit on the frame rate (usually 30 or 60 FPS) (Nystrom, 2014, p. 187).

Performance Profiling and Benchmarking

Implementing default graphics settings and optimizing performance in accordance with the capabilities of the devices by analyzing the performance of the game across a variety of devices.

Feature Scaling

In order to maintain efficacy on lower-end devices, it may be necessary in certain circumstances to disable or scale down particular features. Potential initiatives to streamline physics calculations, reduce particle effects, or restrict the quantity of on-screen entities are a few examples.

By adopting these methodologies, programmers are able to produce video games that are adequately powered by high-end hardware while ensuring that the gameplay experience on lower-end devices remains substantially uncompromised.

3.8 A/B Testing in Game Development

A/B testing, commonly referred to as split testing, is a user experience research technique involving a randomized trial with two variations, A and B. The analysis examines two iterations of a mobile game to ascertain which one achieves superior results in a specific conversion objective, such as user engagement, session duration, or monetization. A/B testing is a crucial tool in mobile game development for making data-driven decisions that improve player pleasure and game performance.

A/B testing in scalable mobile game architecture allows for the gradual improvement of game features, balancing, user interface design, and monetization techniques. Developers can collect empirical data on the effects of specific changes on player behavior and game performance metrics by releasing various versions of a game piece to different segments of the game's audience.

Implementing A/B testing in mobile games usually includes the following steps:

1. **Hypothesis Formation:** The process starts by creating a hypothesis using observations, player input, or analytics data. The hypothesis seeks to enhance a certain game measure, like boosting player retention through modifications to mission difficulty levels.
2. **Variant Design:** Developers develop multiple versions of the game piece based on alternative hypotheses.
3. **Audience Segmentation:** The game's user base is divided into groups based on behavior, demographics, or other pertinent characteristics to ensure comparability. Each part is subsequently subjected to a distinct version of the game.
4. **Data Collection and Analysis:** Data on player interactions with each variant is gathered and scrutinized to identify the version that delivers the desired outcome most efficiently.
5. **Implementation:** The variation that shows a statistically significant improvement compared to the others is chosen and applied throughout the game.

Integrating A/B testing in game creation is a crucial tactic for improving game features to boost player involvement, retention, and revenue. Multiple platforms provide advanced tools for conducting A/B tests, each with distinct features tailored to the specific requirements of game makers. Firebase, GameAnalytics, and Optimizely excel in their extensive capabilities for doing customized A/B testing specifically designed for mobile games.

Firebase

Firebase A/B Testing helps you optimize your app experience by streamlining the way you run, analyze, and scale product and marketing experiments (Google, 2024). Firebase offers a comprehensive set of tools for mobile and online application development, which includes A/B testing features provided by Firebase Remote Config. Developers can alter their app's design and functionality without the need to release an app update using this feature. Developers may utilize A/B testing in conjunction with Firebase Analytics to analyze the effects of various changes on user behavior and app performance, enabling them to make decisions based on facts.

GameAnalytics

GameAnalytics has a specific A/B testing tool that is tailored to improve game mechanics, user interfaces, and monetization methods through testing several versions of the game. Developers may use the platform to distribute different configurations to specific player groups and track how it affects important metrics like retention, playing, and income. Using statistics to find the most effective game adjustments helps make informed decisions and optimize outcomes.

Optimizely

Optimizely is a prominent figure in experimentation and A/B testing, showcasing its expertise in mobile game creation. An intuitive interface is provided for creating and organizing experiments, and analyzing the outcomes with precision. Optimizely allows creators to test various elements of their game, such as gameplay mechanics and in-app purchase offers, to ensure that each modification enhances the overall player experience and game performance.

Incorporating A/B Testing in Game Development

Developers should begin A/B testing in mobile game development by precisely outlining the goals of each test and identifying the precise metrics to be evaluated. Afterward, they should choose a suitable tool that matches their game's technical needs and testing objectives. The next step is to establish the test variations and distribute them to the specific player segments. Continuous monitoring and analysis of acquired data are crucial during the test to determine the performance of the variants. Developers can use the results to make informed decisions about which improvements to permanently integrate in the game.

A/B testing is a crucial tool for game developers, providing a methodical way to enhance game aspects. Developers may enhance player pleasure and drive game success by utilizing technologies such as Firebase, GameAnalytics, and Optimizely to satisfy the dynamic expectations of the gaming community.

3.9 Remote Configuration for Dynamic Content Management

Remote configs are a way to use unique configuration keys that allow to make modification to variables in game without going in the game code (GameAnalytics, 2024). Developers can modify game behavior and look in real-time using specified variables that can be modified on the fly. Implementing Remote Configuration in a scalable game architecture greatly improves the game's ability to adjust and respond to user feedback and analytics insights, which are important characteristics highlighted in the thesis.

Remote Configuration is a crucial feature in scalable mobile game architecture for sustaining and improving the game's relevance and appeal over time. It enables the smooth modification of game settings, stages, characteristics, and user interfaces based on various player preferences and actions. Adaptability is crucial for appealing to a broad audience, resolving performance issues, and adding new material to maintain the game's interest without causing inconvenience through frequent updates.

Pairing Remote Configuration with A/B testing enhances its effectiveness, as detailed in previous section to empirically assess the effect of various configurations on user engagement and retention. Remote Configuration allows developers to apply the most effective versions determined through A/B testing immediately. This mutually beneficial partnership speeds up the optimization process and guarantees that adjustments are based on data-driven evidence, therefore minimizing the dangers linked to alterations in the game's environment.

Imagine a situation in which A/B testing shows that lowering the difficulty of a specific level leads to higher player retention. Developers can promptly modify the difficulty characteristics of the level for all players using Remote Configuration, allowing them to implement the successful test variation into the game instantly. Immediate action is essential for analyzing data and improving the player's experience without waiting for permission from app stores.

Utilizing Remote Configuration for Dynamic Content Management is essential for creating a scalable architecture for mobile games. It enables creators to dynamically manage game material, ensuring the game stays entertaining, sensitive to player needs, and competitive in the ever-changing mobile gaming industry. Remote Configuration, along with A/B testing, is a key component of a flexible, data-driven strategy for game development and optimization. It helps achieve the main goals set forth in the thesis and enhances the game's ability to adjust and grow effectively.

3.10 Case Studies and Industry Practices

Successful mobile games whose architectures have been able to scale to accommodate millions of participants globally are abundant in the industry. Through a critical analysis of these case studies, one can extract invaluable insights and discern optimal strategies that propel mobile games towards success and scalability.

3.10.1 Analysis of successful scalable mobile game architectures

The extensively embraced mobile iteration of Epic Games' Fortnite exemplifies a mobile game architecture that is both successful and scalable. Fortnite has effectively sustained a substantial global player community, surpassing 350 million as of May 2020, by utilizing Amazon Web Services (AWS). As its 100-player battle-royale format necessitates real-time, fast-paced player interactions, the game's architecture, which is fueled by AWS, ensures an optimal, low-latency gaming experience. Fortnite has successfully managed substantial increases in concurrent participants on all platforms, including mobile devices, due to the cost optimization, scalability, flexibility, and dependability that AWS has furnished (Mijuskovic, 2021).

Further noteworthy instances of mobile games that have effectively expanded their architectures are as follows:

Pokémon GO

Pokémon GO, which was created by Niantic, witnessed tremendous expansion after its debut, utilising geolocation technology and cloud solutions to effectively oversee its enormous international player population and in-game interactions in the real world. The game effectively accommodated millions of users across the globe by adjusting to fluctuating loads while preserving performance.

Clash of Clans

The server-client architecture of Supercell's Clash of Clans enables the game to efficiently manage millions of concurrent users. By enabling frequent updates and feature additions with minimal delay, the game's architecture exemplifies effective scalability strategies in the realm of mobile gaming.

Candy Crush Saga

King's Candy Crush Saga showcases the application of a resilient backend infrastructure in order to facilitate the participation of numerous concurrent players, deliver consistent updates, and preserve the state of the game across various devices. This guarantees users a unified and cohesive experience throughout their involvement in this puzzle adventure.

The significance of scalable architectures in accommodating the ever-changing requirements of mobile games is underscored by these instances. Through the utilization of cloud services, microservices architectures, efficient data cache, and content delivery networks (CDNs), developers can guarantee the continued accessibility, engagement, and performance of their games on an extensive array of mobile devices.

3.10.2 Lessons learned and best practices

In the mobile gaming industry, the achievements of Fortnite, Pokémon GO, Clash of Clans, and Candy Crush Saga highlight a number of scalable game development best practices:

Embrace Cloud Services

By adopting cloud services, which enable dynamic resource scaling to accommodate varying levels of player engagement, games can effectively manage periods of high utilization.

Implement Microservices

The implementation of a microservices architecture improves the resilience and scalability of a game by facilitating updates and scaling of game components in a more straightforward manner.

By implementing these strategies, mobile game developers can guarantee sustained success in the fiercely competitive mobile gaming industry by designing scalable infrastructures capable of accommodating sizable player bases.

4 Practical Part

4.1 Environment selection

Choosing Unity as the main development platform for the developed mobile game prototype had a crucial impact on the architectural design and development process. Unity was chosen due to certain crucial characteristics that highlight its appropriateness for developing scalable, high-performance mobile games.

4.1.1 Reasons for Choosing Unity

Architectural Compatibility

Unity's architecture and features are ideal for constructing scalable game structures. The component-based architecture enables adaptable development and seamless integration of scaling principles, essential for achieving the prototype's goals.

Advanced Development Capabilities

Unity provides a wide range of tools for game creation, such as physics, animation, and UI systems, that are crucial for creating a thorough and operational prototype. The capabilities allow for simulating and testing different architectural designs and scaling tactics in the game environment.

Efficiency in Prototyping

Unity excels at enabling quick prototyping. This capacity is crucial for the thesis since it enables rapid iteration over architectural designs and the evaluation of various scaling options. Unity's visual editor and scripting capabilities facilitate the transformation of architectural ideas into reality.

Performance Optimization Tools

Unity's profiling and optimization capabilities are important for a prototype emphasizing scalability and speed. They provide in-depth examination of the game's performance on various devices and under varied conditions, enabling specific improvements crucial for verifying the architectural design.

Availability of Educational Resources

Unity's extensive usage and helpful community ensure a plethora of educational materials, tutorials, and forums are accessible. These resources are especially useful for tackling the distinct issues of creating scalable game designs and can offer guidance and answers during the development process.

Emphasize scalability and maintainability.

Unity was chosen for its backing of scalable and maintainable game development processes. The support for modular design, asset management, and cross-platform development is in line with the thesis's objective of establishing a scalable mobile game architecture that is readily maintainable and expandable.

4.2 Designing the Game Architecture

The game architecture's design phase prioritized developing a strong and adaptable framework to support the game's fundamental dynamics, provide scalability, and uphold excellent performance on various mobile devices. This phase was crucial for transforming theoretical thoughts into a realistic and functional architecture that fulfills the evolving requirements of mobile gaming.

4.2.1 Game Requirements

The project criteria were initially created to focus on essential issues for performance optimization in order to develop a scalable mobile game architecture that simplifies testing. The criteria influenced the game concept to ensure it offers an interesting user experience and serves as a strong platform for testing and showing optimization tactics.

Requirements:

Scalability

The game must be able to scale to accommodate a fluctuating number of objects and interactions while maintaining performance quality. This requirement is essential for evaluating how the architecture handles load increases by modeling real-world settings with fluctuating game activity.

Performance Optimization

The architecture should facilitate thorough performance optimization testing on various devices, emphasizing rendering efficiency, memory management, and CPU utilization. Optimization methods such object pooling and efficient data loading algorithms should be used.

Cross-Platform Support

The architecture should be interoperable with main mobile platforms (iOS and Android) to assess performance across various hardware and operating systems. This condition is crucial for conducting optimization testing to ensure the wide application of the results.

4.2.2 Game Concept Derived from Requirements

Based on the outlined requirements, the game concept developed is a merge-style strategy game that incorporates elements conducive to optimization testing. Players merge different things on a dynamic playing field to create more intricate creations, advancing through levels with escalating complexity and interaction density. This fundamental feature introduces a variety of items and actions, perfect for testing scalability and performance optimization.

The prototype's game concept is created as a runner game, using elements from popular runner games like "Join Clash." This design choice combines runner and merge dynamics to challenge players and provides a broad platform for investigating optimization strategies within the thesis framework.

Merge Mechanics for Scalability Testing

Utilizing merging mechanics in scalability testing allows for testing a wide range of items due to the mechanic's capacity to introduce several objects. The players' actions lead to dynamic changes in the game states, necessitating the architecture to effectively handle object generation, destruction, and transformation.

Level Progression for Dynamic Content

The game has a level-based progression system that introduces new objects and obstacles as the player advances. This framework enables the evaluation of dynamic content updates to assess the seamless integration of new levels and objects without affecting performance.

Cross-Platform Optimization

It involves building the game to work on both iOS and Android devices, allowing for the evaluation of optimization strategies specific to each platform's hardware and operating system.

This strategic game concept, based on the project's optimization and scalability needs, acts as a flexible framework for creating and evaluating a scalable mobile game architecture. The gaming mechanics and features are carefully selected to test and assess the performance of the architecture, guaranteeing that the created prototype is both engaging and a demonstration of effective game design and optimization.

4.2.3 Selection of architectural patterns and technologies

During the practical development phase of the mobile game architecture, particular architectural patterns and technologies were chosen to guarantee the project achieved its key goals of scalability, performance optimization, and maintainability. The selection process was based on a thorough literature review and analysis of existing industry standards in mobile game development.

Architectural patterns selected:

Model-View-Presenter (MVP)

The MVP pattern was selected for its efficacy in constructing a modular codebase, encouraging a clear division between the game's logic and appearance. This pattern improves the ability to test and maintain the system by clearly separating the management of the game's data (Model), its display (View), and the interactions between them (Presenter). MVP is well-suited for Unity projects that prioritize modularity and scalability, making it a popular architectural choice for games.

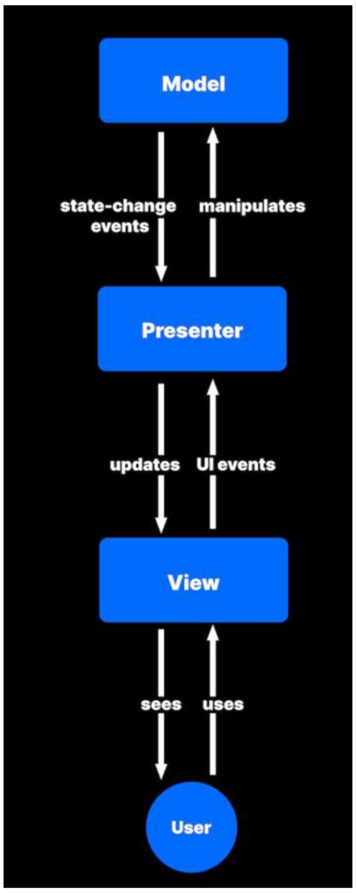


Figure 1 - MVP Architectural Pattern Workflow. Source: Unity, 2022.

Service-Oriented Architecture (SOA)

SOA was picked to ease the integration of numerous independent services, such as in-game purchases, social features, and content updates. SOA allows the game to scale and adapt to changing requirements by organizing it as a network of communicating services. This design enables the seamless integration of new game features and content, facilitating the game's expansion after its release.

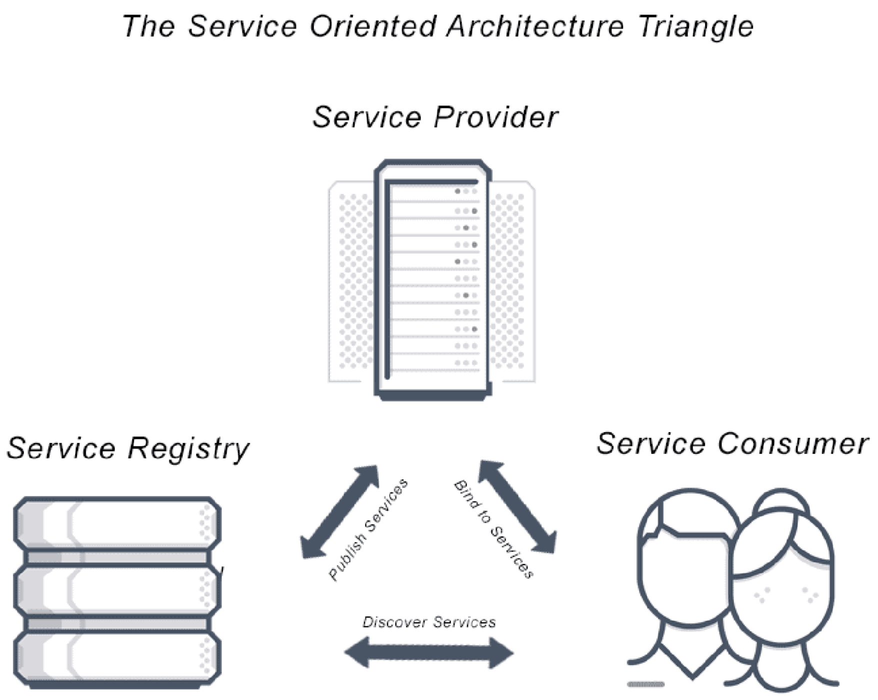


Figure 2 - Components of Service-Oriented Architecture. Source: Avinetworks, 2024.

Service Locator Pattern

This pattern was used to efficiently handle dependencies in the game architecture based on the Service-Oriented Architecture approach. It serves as a central database, Service Provider, where different services, such music, physics, or networking, can be stored and accessed when required. This pattern facilitates the access of common resources and services in the game, encouraging loose coupling and improving the flexibility and extensibility of the game design.

Finite State Machine Pattern

This pattern was chosen to efficiently manage different game states, including menu navigation, gameplay, pausing, and session conclusion. The pattern serves as a central register for these services, allowing for convenient access throughout various sections of the game, which promotes a modular and loosely coupled design. This method simplifies the incorporation and control of external services and APIs while also improving the game's capacity to be maintained and expanded through a versatile structure for service utilization and modifications.

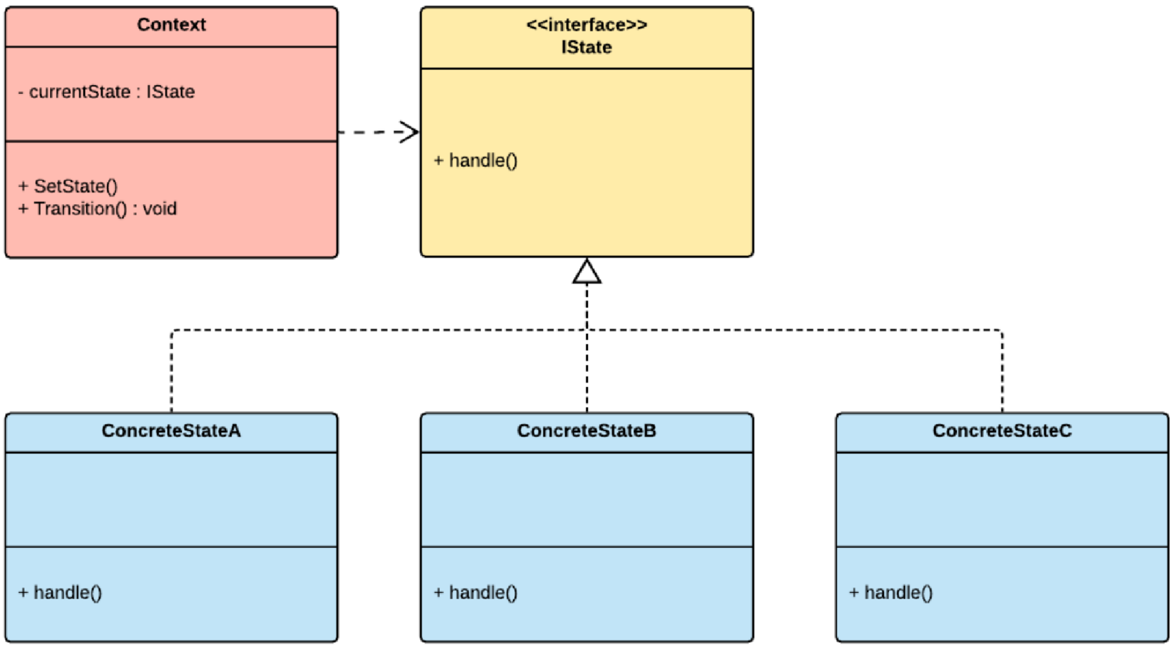


Figure 3 - State Design Pattern Structure. Source: Baron, Game Development Patterns with Unity, 2021.

Publish-Subscribe/Observer Pattern

The publish-subscribe technique was utilized to oversee in-game events and interactions across various components of the game. This paradigm enables a system with strong decoupling, where game elements subscribe to certain events and respond properly when those events occur. It is especially beneficial for developing adaptable and engaging game settings where player inputs prompt a range of game rule reactions.

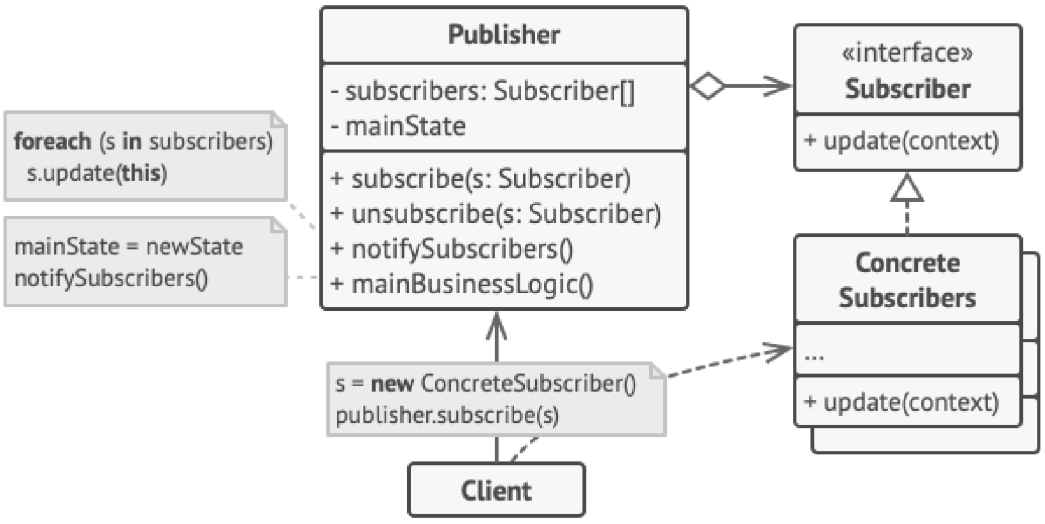


Figure 4 - Observer Design Pattern Diagram. Source: Shvets, Dive Into Design Patterns, 2021.

Proxy Pattern

The proxy pattern was implemented to enhance resource loading and management, which is vital for mobile games that require a delicate balance between performance and resource utilization. This pattern allows the game to delay the loading of large resources until they are required, decreasing initial load times and enhancing the game's performance.

4.3 Implementation of the Prototype

4.3.1 Development environment setup

During the practical implementation part of the thesis, the development environment was established to facilitate the building of the game prototype. The selection of Unity version 2022.3.16f1, the most recent Long-Term Support (LTS) version available, was based on its stability and extensive range of features. JetBrains Rider version 2023.2 was chosen as the coding environment for its extensive capabilities designed specifically for Unity development, providing an effective and robust platform for coding and debugging. The configuration established a strong foundation for creating the scalable mobile game architecture prototype.

4.3.2 High-level overview of the prototype's architecture

The game prototype's architecture was implemented using the Model-View-Presenter (MVP) pattern to separate the game logic from the user interface, improving modularity and maintainability. Implementing the Service Locator pattern enables efficient access to different in-game services, improving the flexibility of the design. Utilizing a Game State Machine pattern effectively controls game states to provide a smooth flow during gameplay. This architectural approach is utilized in the Unity environment, making use of its wide range of development tools to efficiently bring to life the intended game dynamics and interactions.

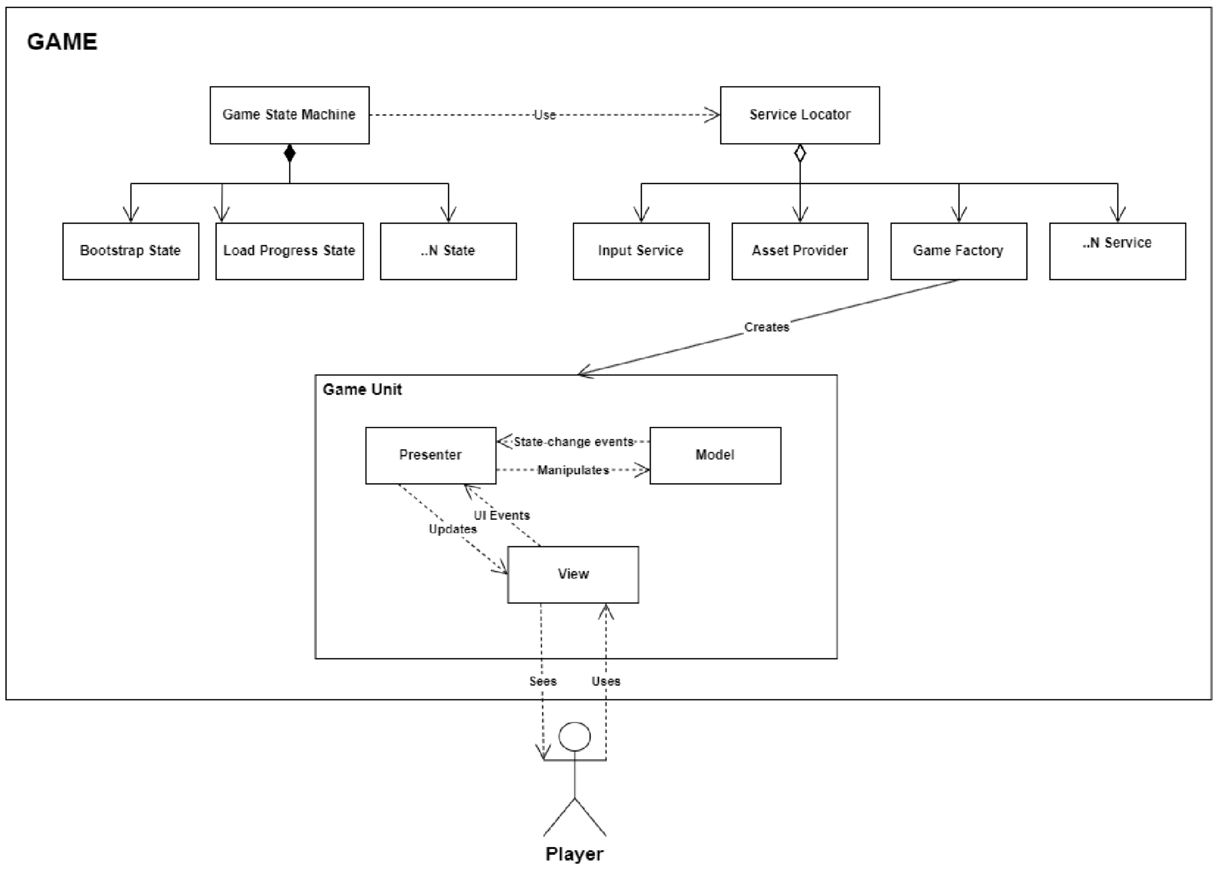


Figure 5 - High-level Game Architecture Overview. Source: Author.

4.3.3 Comprehensive Implementation and Technical Details

The **GameBootstrap** class serves as the entry point, responsible for initializing the fundamental features of the game and transitioning to the initial state. The code for **GameBootstrap** will be presented in detail, showing how it makes use of the Unity **MonoBehaviour** lifecycle and interacts with the game state system.

```
public class GameBootstrap : MonoBehaviour
{
  private Game _game;

  private void Awake()
  {
    _game = new Game();
    _game.StateMachine.Enter<BootstrapState>();

    DontDestroyOnLoad(this);
  }
}
```

Figure 6 – GameBootstrap class code snippet. Source: Author.

The **Game** class is introduced as the primary coordinator for managing the game's state. The GameStateMachine is instantiated with the injected Instance of the ServiceLocator, which serves as a central center for accessing different game services.

```
public class Game
{
      public readonly GameStateMachine StateMachine;

      public Game() =>
      StateMachine = new GameStateMachine(ServiceLocator.Container);
}
```

Figure 8 - Game class code snippet. Source: Author.

The **GameStateMachine** class is responsible for managing the game through various states. The system keeps a dictionary that links each state type to its respective state object, enabling effective state transitions and control. The class is responsible for creating many game states, each designed to oversee specific parts of the game's lifecycle, including startup, progression, and level management.

```
public class GameStateMachine : IGameStateMachine
{
    private readonly Dictionary<Type, IState> _states;
    private IState _activeState;

    public GameStateMachine(ServiceLocator services)
    {
      _states = new Dictionary<Type, IState>
      {
        [typeof(BootstrapState)] = new BootstrapState(this, services),

        [typeof(LoadProgressState)] = new LoadProgressState(this,
        services.Single<ISaveLoadService>(),
        services.Single<IStaticDataService>()),

        [typeof(LoadLevelState)] = new LoadLevelState(this,
        services.Single<IGameFactory>(), services.Single<IUIFactory>()),

        [typeof(GameMenuState)] = new GameMenuState(this,
        services.Single<IGameFactory>(), services.Single<IUIFactory>()),

        [typeof(MergeTableState)] = new
        MergeTableState(services.Single<IUIFactory>()),

        [typeof(LevelState)] = new
        LevelState(services.Single<IGameFactory>()),
    };
  }
}
```

Figure 7 - GameStateMachine class code snippet. Source: Author.

The **GameStateMachine** class in the game architecture uses a dependency injection, where services needed by different states are supplied via an instance of ServiceLocator. Each state can access the necessary services, like save/load system or UI factory, to maintain modularity and testability. The service locator pattern facilitates the centralized registration and retrieval of services, which can be injected into states by the state machine as they are generated and entered.

The Game State Machine includes different states for various game phases such as initialization, resource loading, level management, and gameplay. Various states including BootstrapState, LoadProgressState, LoadLevelState, GameMenuState, MergeTableState, and LevelState are implemented to maintain a structured and efficient game flow.
The flow of the state machine including events of the states is depicted on the diagram below:

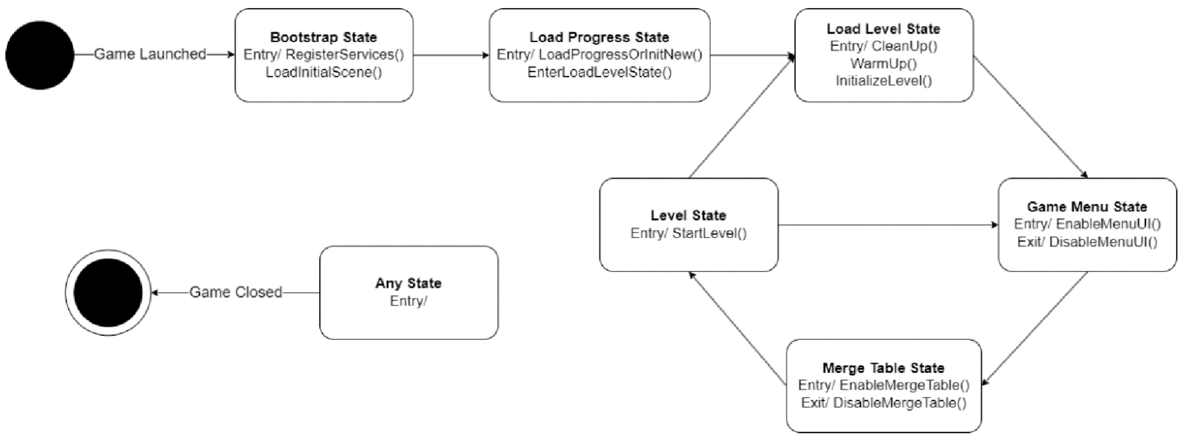


Figure 9 - Game Finite State Machine Diagram. Source: Author.

4.3.4 Integration of design patterns

Service Locator

The Service Locator pattern is represented in the game design via the **ServiceLocator** class, serving as a centralized register for services. The class utilizes a singleton pattern to guarantee the existence of a single instance of the **ServiceLocator** throughout the game. The **RegisterSingle** method is used to register services, while the Single method is used to retrieve them. This method enables the independent and adaptable administration of game services, making it simple to access and alter services without affecting the overall system's architecture.

```
public class ServiceLocator
{
  private static ServiceLocator _instance;
  public static ServiceLocator Container =>
      _instance ??= new ServiceLocator();

  public void RegisterSingle<TService>(TService implementation)
      where TService : IService =>
            Implementation<TService>.ServiceInstance = implementation;

  public TService Single<TService>() where TService : IService =>
    Implementation<TService>.ServiceInstance;

  private class Implementation<TService> where TService : IService
  {
    public static TService ServiceInstance;
  }
}
```

Figure 10 - ServiceLocator class code snippet. Source: Author.

During the Bootstrap state of the game, the **ServiceLocator** and its services are registered. This phase ensures that all essential services are set up and ready before the game progresses to the subsequent states. The design offers a streamlined and orderly approach to service administration by centralizing service registration in the Bootstrap state, creating a stable foundation for the game's operation and future scalability.

Proxy Pattern

The Proxy pattern in the **AssetProvider** class improves asset management by quickly caching assets and managing their asynchronous loading. Assets are cached to avoid repetitive processes, which helps optimize resource utilization and loading speed. This method showcases a useful implementation of the Proxy pattern in Unity-based game development to optimize asset access and management without compromising performance.

```
public class AssetProvider : IAssetProvider
{
  private readonly Dictionary<string, AsyncOperationHandle>
    _assetsCache = new Dictionary<string, AsyncOperationHandle>();

  private readonly Dictionary<string, List<AsyncOperationHandle>>
     _operations = new Dictionary<string, List<AsyncOperationHandle>>();

  public void Initialize() => ...;

  public async Task<T> Load<T>(AssetReference assetReference)
      where T : class {...}

  public async Task<T> Load<T>(string address) where T : class{...}

  public Task<GameObject> Instantiate(string address) => ...;

  public void Cleanup(){...}

  private async Task<T> RunWithCacheCompleted<T>(AsyncOperationHandle<T>
handle, string cacheKey) where T : class {...}

  private void AddHandle<T>(string key, AsyncOperationHandle handle)
      where T : class {...}

}
```

Figure 11 - AssetProvider class code snippet. Source: Author.

MVP Architectural Pattern implementation

The Coins Pickup game unit illustrates the implementation of the MVP pattern in Unity. This class is organized based on the MVP components: Model for handling coin data, such as the amount of the reward, View for displaying it in the game environment, and Presenter for controlling events like coins picked up. This configuration effectively assigns responsibilities to different components, improving the ability to maintain and test code. It also offers a useful structure for implementing MVP in game development situations.

Here is the overview of the class structure:

```
public class CoinsPickup : MonoBehaviour, ICollectable
{
      [Serializable] private MeshRenderer _coinsMesh;
      [Serializable] private IncomeUI _incomeUI;
      [Serializable] private CoinsPickupConfig _config;
      public bool IsCollected { get; set; }

      public static event Action<CoinsPickup> OnCoinsPickupCollected;

      public void OnCollected(){...}
}
```

Figure 12 - CoinsPickup class code snippet. Source: Author.

The **CoinsPickup** class acts as the **Presenter** in this example, managing the game logic for collecting coins. It incorporates **two Views** components: the **MeshRenderer**, which presents the 3D model of the coin in the game environment, and the **IncomeUI**, which manages updating the user interface to show coin pickups. The **CoinsPickupConfig**, acting as the **Model**, contains coin-related information, such as coin amount, and is provided by the StaticDataService to maintain a clear separation between game logic and display following the MVP architecture. This structure enables a distinct allocation of duties within the game's framework.

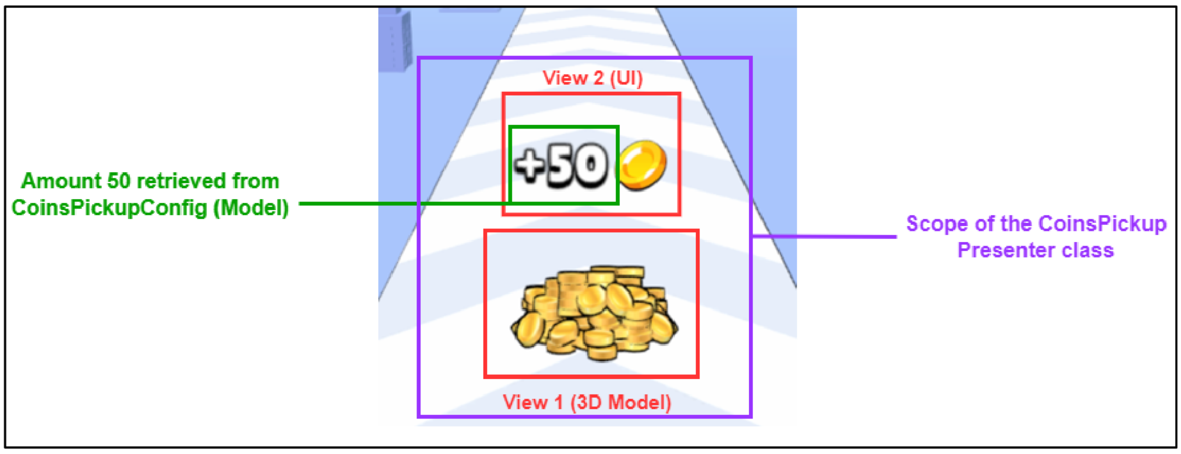

Amount 50 retrieved from CoinsPickupConfig (Model)
View 2 (UI)
+50
View 1 (3D Model)
Scope of the CoinsPickup Presenter class

Figure 13 - MVP CoinsPickup example. Source: Author.

Observer Pattern

The Observer pattern was used into the MVP design to improve the communication between the user interface and game logic. In the example from previous section, The UI View listens to the **OnCoinsCollected** event to update the displayed coin amount whenever coins are collected in the game. This implementation enables a system that is decoupled and responsive, efficiently updating the user interface to reflect changes in the game state. It showcases the pattern's effectiveness in enabling dynamic data updates within the game's architecture.

4.3.5 Static Data Service implementation

The Static Data Service is designed to efficiently maintain and retrieve static game data, including character traits, level information, and gameplay features. Centralizing this data enhances the game's performance by ensuring consistent and quick access, expediting development, and improving the user experience with reduced load times and optimized resource management. This service is intended to complement the upcoming Remote Config Service by enabling dynamic adjustments to static data settings depending on player interactions and feedback, hence improving the game's adaptability and customization. The combination of static data management and dynamic setup highlights the architecture's adaptability and ability to respond to changing gaming requirements.

```
public class StaticDataService : IStaticDataService
{
   public PlayerStaticData PlayerConfig { get; set; }

   private const string PlayerConfigPath = "StaticData/PlayerConfig";

   public void Load()
   {
      PlayerConfig = Resources.Load<ScriptableObject>(PlayerConfigPath)
      as PlayerStaticData;
   }
}
```

Figure 14 - StaticDataService class code snippet. Source: Author.

4.3.6 Implementing Remote Config

Remote Config using GameAnalytics was used in the prototype development for dynamic level and content control. GameAnalytics was selected for its outstanding support for immediate setup adjustments and its compatibility with A/B testing capabilities, enabling smooth upgrades to game content according to player interactions and feedback. The decision was driven by the platform's specialized emphasis on game analytics, providing precise insights and settings that are not as easily accessible or as finely calibrated in other platforms.

A new service called **RemoteConfigService** was developed and added to the **BootstrapState** to incorporate remote configurations into the game prototype. The service was dependency injected into the **LoadProgressState**. The use of **RemoteConfigService** enabled dynamic verification of remote configurations: if remote configurations were present, their values were utilized; otherwise, default values were applied. This configuration allowed the game to adjust its content and settings in response to real-time data and player feedback, improving the overall user experience.

```
public class RemoteConfigService : IRemoteConfigService
{
   public string GetRemoteConfigValue(string key, string defaultValue)
   {
      if (GameAnalytics.IsRemoteConfigsReady())
         return GameAnalytics.GetRemoteConfigsValueAsString(key,
      defaultValue);

      return defaultValue;
   }
}
```

Figure 15 - RemoteConfigService class code snippet. Source: Author.

After integrating the RemoteConfigService into the game prototype, the next important task was configuring the Remote Config settings on GameAnalytics.com. The infrastructure was crucial for managing and deploying dynamic game settings and content updates, allowing the game to adjust in real-time to enhance player experience and engagement using data-driven insights.

4.3.7 Integrating A/B Testing in the Game Prototype

The integration of A/B testing in the game prototype followed a similar strategy to implementing Remote Config, focusing on using GameAnalytics. This strategic decision allowed for the dynamic testing of game variables to enhance user engagement and gameplay using real-time data. The process included implementing various configurations to specific groups of players and evaluating the effects on game performance indicators, enabling data-informed improvements to the game's design and features.

No modifications to the codebase were required to implement A/B Testing in the game prototype as described in section 4.5. The efficient connection is possible due to GameAnalytics' A/B testing feature, which utilizes the existing **RemoteConfigService**. This solution is convenient because it just requires configuration tweaks on the GameAnalytics platform. It efficiently utilizes existing functionalities to enable A/B testing without the need for extra code.

4.3.8 Implementing Save Load system

The Save Load system is implemented practically using the ES3 asset, a powerful tool for cross-platform serialization and encryption, obtained from the Unity Asset Store. The decision was based on ES3's capacity to effectively manage intricate data types and guarantee the security of stored data, which is crucial for preserving user progress and game state between sessions.

The **SaveLoadService** class was created to handle the game's persistent data, including crucial game state elements such as **LevelManagerData** and **SettingsData**. This service provides a complete solution for storing and retrieving game data, with smooth integration into the overall game structure. The SaveLoadService uses ES3 to securely save and make accessible player progress, settings, and important data, improving the game's user experience by ensuring consistency and dependability in its operation.

4.4 Optimization Techniques Applied

Asset bundling, memory management, and code optimization were crucial methods used during the prototype's development to ensure smooth gaming on various platforms. The Unity Profiler played a key role in pinpointing and resolving performance issues. Object pooling and mesh baking were essential for reducing resource costs by reusing objects and consolidating static objects into single meshes to save draw calls. This optimization improved rendering efficiency and enhanced the overall game experience.

4.4.1 Initial performance metrics and baseline

A dual-platform testing approach will be used for this and Section 4.5, Testing and Validation. It will involve Unity's device simulator on a PC and a Xiaomi Redmi 12 android mobile phone with specific features including 4GB RAM, 128GB storage, a MediaTek Helio G88 processor, Arm Mali-G52 GPU, and a 6.5-inch display with a 90Hz refresh rate. The testing PC features an AMD Ryzen 5 5600X 6-Core Processor running at 3.69 GHz, 32GB of RAM, and an NVIDIA GeForce GTX 1650 Super GPU. This configuration offers a stable base for assessing the game's performance under pressure while utilizing a 64-bit operating system.

This guarantees a thorough assessment of the game's performance on different hardware, ranging from high-end PCs to less powerful mobile devices, supporting the optimization of the game for a wide range of user experiences.

The benchmarking procedure involves measuring parameters such as:

- FPS (Frames Per Second)
- Batches
- Tris (Triangles)
- CPU Usage
- Memory Usage.

The stats will offer a thorough perspective on the game's performance on various devices. Frames per second (FPS) assesses gaming smoothness, whereas Batches and Tris provide information on rendering efficiency. Monitoring CPU and Memory Usage is essential for assessing the game's resource use on the device and directing optimization strategies to maintain a harmonious performance without straining the hardware.

A demo level was developed for the thesis, showcasing a high concentration of dynamically instantiated items. This configuration functions as a stress test for the system, evaluating performance under high demands without utilizing object pooling. This method enables a comprehensive assessment of the game architecture's ability to manage resource-heavy and critical situations, offering useful insights into areas where improvements might greatly enhance performance.

The following table presents the results of the benchmarking tests:

Indicator	**PC**	**Xiaomi Redmi 12**
FPS	30 FPS	15 FPS
Batches	427	427
Tris (Triangles)	394.4k	394.4k
CPU Usage	33 ms	80 ms
Memory Usage	2.71 GB	0.51 GB

Table 1 - Initial Benchmark Performance Metrics Before Optimization. Source: Author.

4.4.2 Optimization strategies

A comprehensive plan was implemented to improve the efficiency and expandability of the mobile game structure. This section explains the deployment of several optimization approaches aimed at resolving typical bottlenecks and enhancing the gaming experience.

Object pooling was used to efficiently handle the creation and deletion of game objects, which helped reduce memory allocation overhead and prevent performance spikes caused by trash collection. This method was crucial in preserving high frame rates and guaranteeing seamless gameplay, particularly in situations with frequent object changes.

Mesh baking was another crucial optimization technique used. By precalculating and merging many meshes into a unified mesh, the amount of draw calls was greatly decreased. This optimized the rendering process and reduced CPU-GPU communication overhead, resulting in enhanced rendering speed.

The reduction of render calls was accomplished by meticulously optimizing the game's rendering pipeline. Methods like batch processing and frustum culling were used to decrease the number of items handled and displayed every frame. This modification significantly reduced the computational workload on the GPU, resulting in improved frame rates and greater visual quality.

Texture compression was used to decrease the size of texture assets while maintaining their quality. This strategy lowered memory consumption and bandwidth needed for loading textures, resulting in faster asset loading times and decreased runtime memory utilization, ultimately improving gaming responsiveness.

The optimization strategies worked together to support the performance enhancement efforts, guaranteeing that the mobile game architecture could provide a smooth and captivating experience on various devices.

4.4.3 Optimization techniques implementation

Object Pool

The object pooling implementation was specifically aimed at streamlining the administration of dynamically created objects, particularly under the "weapons" category. By reusing weapon objects instead of constantly creating and deleting them, we decreased memory allocation overhead and mitigated performance consequences caused by garbage collection.

```
public class ObjectPoolMono<T> where T : MonoBehaviour
{
      private List<T> _pool;
      private readonly T _prefab;
      private readonly Transform _container;
      private bool _autoExpand;

      public ObjectPoolMono(T prefab, int count, Transform container,
            bool autoExpand){...}

      public T GetFreeElement(){...}

      private void CreatePool(int count){...}

      private T CreateObject(bool isActiveByDefault = false){...}

      private bool HasFreeElement(out T element){...}
}
```

Figure 16 - ObjectPoolMono class code snippet. Source: Author.

The code snippet demonstrates an object pooling system for Unity GameObjects using generics to construct a flexible and reusable pool for any form of MonoBehaviour.

The **TableWeaponsPool** class is a specialized implementation of the object pooling pattern in the thesis project, designed to handle **TableWeapon** GameObjects. This object pooling implementation is designed to enhance the creation and control of table weapon objects, which are essential in merge mechanics. The system optimizes game performance and memory efficiency by using object pooling for table weaponry, reducing the overhead from frequent instantiation and destruction.

```
public class TableWeaponsPool : MonoBehaviour
{
      [SerializeField] private TableWeapon _prefab;
      [SerializeField] private int _poolCount = 100;
      [SerializeField] private bool _autoExpand = false;

      private ObjectPoolMono<TableWeapon> _tableWeaponsPool;

      private void Construct() =>
            _tableWeaponsPool = new ObjectPoolMono<TableWeapon>(_prefab,
            _poolCount, _autoExpand, transform);

      private TableWeapon InstantiateTableWeapon() =>
            _tableWeaponsPool.GetFreeElement();
}
```

Figure 17 - TableWeaponsPool class code snippet. Source: Author.

Furthermore, this object pooling technique was expanded to include not only table weapons but also other in-game aspects like adversaries and cash. The extensive use of object pooling in various game objects highlights its importance in creating an efficient and adaptable mobile game structure. The project showcases a practical application of theoretical optimization techniques in game development by exploiting object pooling to enhance game responsiveness and stability.

Draw Calls Optimization

Mesh baking was used on the nearby structures in the game map to combine them into a single mesh and improve draw call efficiency. This method greatly enhanced rendering efficiency by decreasing the quantity of separate draw calls required, therefore boosting the game's performance and guaranteeing a more seamless player experience. The game architecture effectively handled rendering workloads by implementing this optimization, resulting in a visually detailed yet performance-optimized environment.

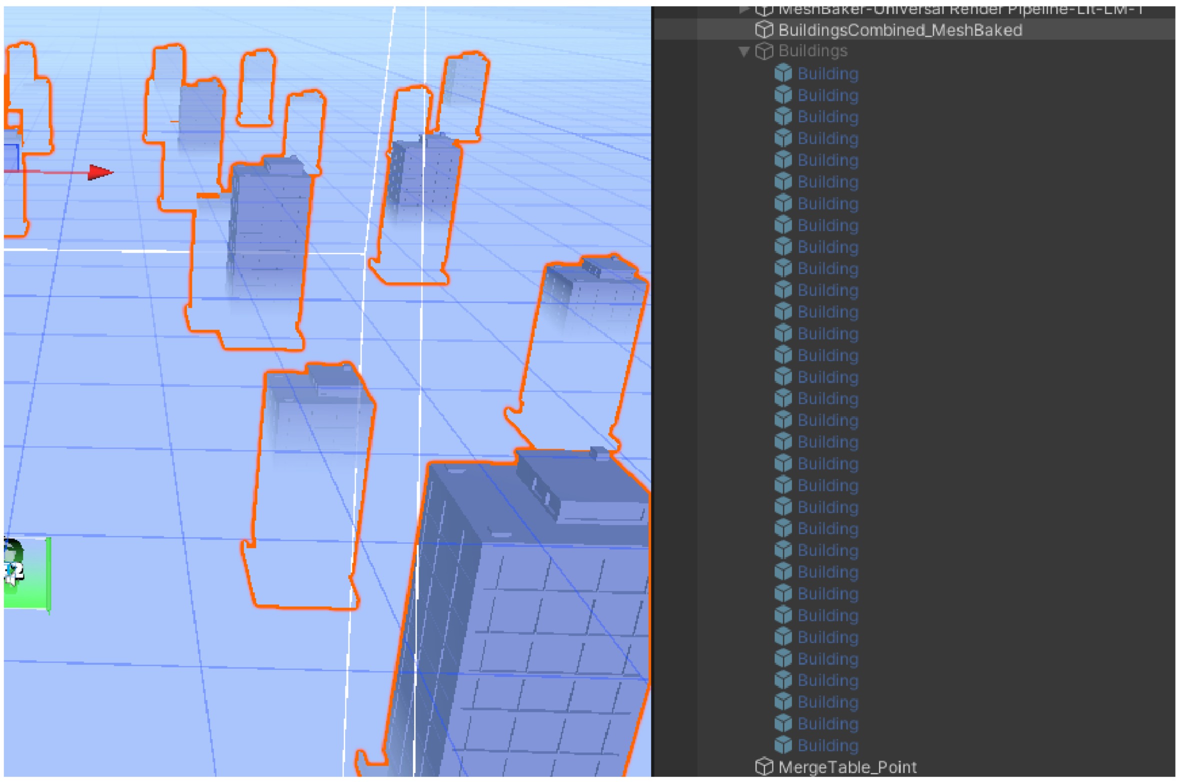
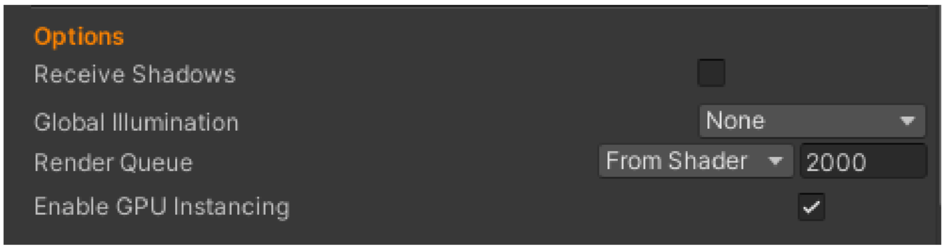


Figure 18 - Mesh Baking application to Building game objects. Source: Author.

Two advanced approaches, GPU Instancing and Draw Call Batching, were used to enhance the rendering pipeline in addition to mesh baking.

GPU instancing was used to significantly decrease the processing burden of rendering numerous instances of objects that have identical shape and material, such foliage and environmental props. Utilizing GPU instancing enables rendering these objects in a single draw call, accommodating multiple instances with little performance repercussions. This strategy is highly effective in densely populated environments with multiple identical objects, such as groupings of trees or clusters of decorations.



Options

Receive Shadows ☐

Global Illumination: None

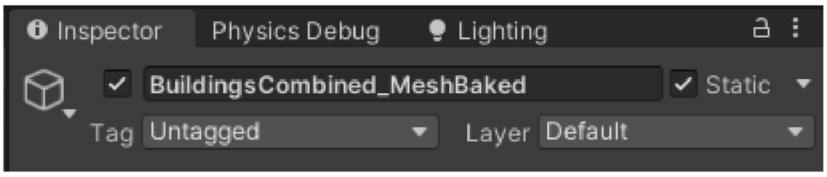
Render Queue: From Shader 2000

Enable GPU Instancing ✓

Figure 19 - GPU Instancing settings. Source: Author.

Unity utilized its built-in **draw call batching** techniques, Static and Dynamic Batching, to reduce the quantity of draw calls. Static Batching optimizes scenes with static geometries by combining rendering data of non-moving game objects into single draw calls.

Dynamic Batching combined the rendering of non-static objects that have few vertices and use the same materials. The batching approaches improved the game's performance by minimizing CPU-GPU connection and allowing it to operate more smoothly on different platforms, in addition to optimizing the mesh baking process.



✓ BuildingsCombined_MeshBaked ✓ Static

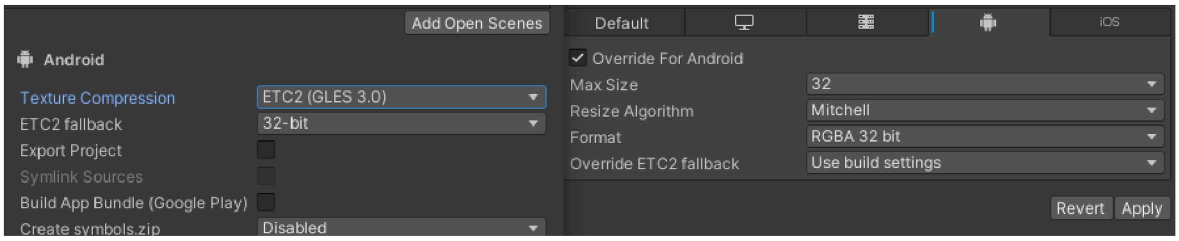
Tag Untagged Layer Default

Figure 20 - Static Batching settings. Source: Author.

These optimizations, including mesh baking, GPU instancing, and draw call batching, collaborate to lessen the rendering workload on the GPU. Implementing these tactics successfully reduced the frequency of draw calls, leading to enhanced frame rates and a more consistent gaming experience without compromising visual quality.

Texture compression

Texture compression was employed to improve performance on Android devices. The format used was ETC2 (GLES 3.0) for its efficiency in managing alpha channels in 32-bit color textures. The maximum texture size was adjusted to 32 pixels, tailored for specific content categories that do not necessitate high-resolution detail. This method helped save memory while preserving acceptable visual quality. The Unity Editor's settings were methodically modified using the 'Mitchell' resize algorithm to maintain texture integrity while scaling. The modifications helped achieve an ideal equilibrium between visual quality and efficiency, crucial for the limited resources available in mobile gaming.



Android

Texture Compression	ETC2 (GLES 3.0)
ETC2 fallback	32-bit
Export Project	
Symlink Sources	
Build App Bundle (Google Play)	
Create symbols.zip	Disabled

✓ Override For Android

Max Size	32
Resize Algorithm	Mitchell
Format	RGBA 32 bit
Override ETC2 fallback	Use build settings

Figure 21 - Texture Compression settings. Source: Author.

4.5 Testing and Validation

A reassessment of the demonstration level was carried out under the same test settings to gauge the effectiveness of the optimization methods applied in the thesis. The demo level, with a high density of dynamically created items, served as a stress test to push the game's architecture to its limits. The thorough testing was conducted to confirm the performance enhancements resulting from optimization techniques including object pooling, mesh baking, and texture compression.

The following table presents the results of the post-optimization benchmarking tests:

Indicator	**PC**	**Xiaomi Redmi 12**
FPS	80 FPS	45 FPS
Batches	113	113
Tris (Triangles)	394.4k	394.4k
CPU Usage	6.3 ms	30 ms
Memory Usage	1.8 GB	0.38 GB

Table 2 - Benchmark Performance Metrics After Optimization. Source: Author.

5 Results and Discussion

5.1 Performance Metrics Comparison

Evaluating the performance metrics offers a quantitative understanding of the optimization tactics put into practice. This comparison is essential for assessing the optimization approaches' efficacy in improving the game's performance across several platforms.

5.1.1 Before and After Optimization Results

Baseline measurements for the unoptimized demo level were recorded during the initial testing phase for both the PC and the Xiaomi Redmi 12. After implementing optimization approaches, a second round of testing was undertaken under identical settings.

Indicator	**PC**	**Xiaomi Redmi 12**
FPS	30 FPS => **80 FPS**	15 FPS => **45 FPS**
Batches	427 => **113**	427 => **113**
Tris (Triangles)	394.4k	394.4k
CPU Usage	33 ms => **6.3 ms**	80 ms => **30 ms**
Memory Usage	2.71 GB => **1.8 GB**	0.51GB => **0.38GB**

Table 3 - Benchmark Performance Metrics Comparison. Source: Author.

The findings pre and post optimization are as follows:

PC: FPS increased from 30 to 80, showing a substantial improvement in frame rate and suggesting a smoother gaming experience. The number of batches has been decreased from **427** to **113**, suggesting a more efficient rendering process with fewer grouped sets of graphics data to handle. The number of triangles (Tris) remained constant at 394.4k, suggesting that the visual intricacy of the demo level **was preserved** while also improving performance. The CPU usage has decreased from **33 ms** to **6.3 ms**, indicating a more efficient utilization of computational resources. The memory usage has been reduced from **2.71 GB** to **1.8 GB**, indicating improved efficiency in memory utilization.

The **Xiaomi Redmi 12** has increased its FPS from **15** to **45**, more than doubling the frame rate and resulting in a significantly better experience on the mobile platform. The number of batches decreased from **427** to **113**, reflecting the enhanced performance on the PC and indicating more effective draw call processing. Tris (Triangles): Maintaining a consistent level at 394.4k, verifying the **preservation** of the scene's graphical detail. The CPU usage has decreased from **80 ms** to **30 ms**, indicating a significant improvement in processing efficiency. The memory usage decreased from **0.51 GB** to **0.38 GB**, indicating enhanced memory management on the mobile device.

5.1.2 Analysis of the effectiveness of optimization techniques

Comparing performance data before and after implementing optimization strategies shows a significant improvement in game performance. The significant rise in frames per second (FPS) on both systems suggests a smoother gaming experience. The decrease in batches indicates a reduction in the number of render calls due to the implementation of mesh baking and draw call optimization techniques. The stable number of triangles indicates that these improvements were made without compromising visual quality.

The significant reduction in CPU consumption highlights the effectiveness of optimization measures, especially the use of object pooling to minimize the computational burden of creating and removing objects dynamically. Furthermore, the decrease in memory utilization on both platforms indicates that texture compression and other memory management strategies have helped reduce the overall memory footprint, which is especially advantageous for mobile devices with restricted resources.

The results confirm the success of the optimization procedures and emphasize the significance of these techniques in creating games that function efficiently on different hardware specifications. The data confirms the theoretical techniques presented in the thesis and shows how they affect the game's scalability and performance.

5.2 Scalability Assessment

The scalability of the architecture was thoroughly assessed to guarantee it could handle an increase in user base, data volume, and features. The architecture's ability to handle gradual improvements and preserve performance metrics while growing was carefully evaluated.

5.2.1 Evaluation of the Architecture's Scalability

The scalability of the design was assessed by examining its capacity to incorporate new features. This evaluation focused on architectural modularity, component decoupling, and interface simplicity between system components. A change impact study was performed to assess the feasibility of integrating new features into the current framework. This entailed:

- Analyzing the dependency list to identify the components affected by the introduction of new features.
- Estimating the implementation effort needed for the impacted components.

The architecture's modularity was crucial in facilitating the addition of new features, like **game states** and **services**, which is essential for its scalability. The procedure of integrating a new game state into the current framework was simplified to necessitate modifications just within the game class. To create a new game state, register it in the GameStateMachine's constructor and define the transitions to this state. This high level of modularity showcases the architecture's preparedness for expanding and adapting to new gameplay components.

The process of integrating new services was uncomplicated. Introducing a new service required the creation of a service class, registering it in the BootstrapState, and then injecting it into the GameState constructor. The injection of the item might be easily done at any required location using the game factory. This service integration method emphasizes the scalability and flexibility of the architecture, allowing for the seamless addition of new features without causing any disturbance to the current system.

The architecture demonstrated its capacity to incorporate intricate game mechanics and services without requiring substantial reworking, confirming its scalability evaluation. The architecture's capacity to sustain performance while undergoing additions was crucial for its success, ensuring its suitability for future development phases.

5.3 Lessons Learned from the Practical Implementation

Translating theoretical notions into a realistic game development framework provided a valuable learning experience, revealing the intricate relationship between theory and practice.

5.3.1 Applying theoretical concepts to practical development insights

The implementation process highlighted the importance of architectural modularity and component decoupling in creating codebases that are scalable and easy to maintain. An important realization was the crucial need to build systems with extensibility in mind, allowing for the inclusion of new game states or services with little adjustments and without compromising the present system's integrity. This method emphasizes the importance of having a carefully planned architectural design that foresees future needs and modifications.

5.3.2 Design patterns and their impact on scalability

Design patterns were essential for enhancing the scalability and extensibility of the architecture. Design patterns offer a defined strategy for resolving typical design issues, ensuring a coherent and uniform approach throughout the development team. The implementation of these patterns helped create a versatile architecture capable of adapting to new requirements and modifications.

The Singleton design guarantees the creation of only one instance of a service, which minimizes memory usage and ensures uniform access to the service throughout the system. **The Factory** pattern facilitated encapsulation of object generation functionality, streamlining the addition of additional object types to the system**. The Observer** design was crucial in establishing decoupled systems, enabling objects to communicate without direct linkage, thus improving modularity and enhancing the manageability of the codebase.

The patterns had a crucial role in enhancing the architecture's scalability through supporting loose coupling, high cohesion, and encapsulation. They facilitated the system's growth and evolution without requiring significant rewrites or revisions, guaranteeing that the architecture could handle the game's increasing complexity and range of features.

By applying theoretical principles and incorporating design patterns, important lessons were gained regarding the significance of planning, adaptability, and the ability to anticipate change. The discoveries will certainly impact future development efforts, emphasizing the ongoing equilibrium between theoretical underpinnings and their practical applications in game creation.

5.4 Recommendations for Future Work

The game architecture's practical execution is thorough but offers opportunities for further investigation and enhancement. The recommendations are intended to direct future research and development endeavors in order to enhance and advance the game architecture.

5.4.1 Potential areas for further research and development

One important aspect to focus on for improvement is incorporating Dependency Injection (DI) frameworks like **Zenject** and **VContainer** into the game's design. These frameworks provide a structured and adaptable method for managing dependencies, making it easier to separate components and services in the game. Utilizing Dependency Injection frameworks can streamline the testing, upkeep, and expansion of the game's codebase, enhancing its adaptability to modifications and improvements.

Integrating DI frameworks can efficiently facilitate the incorporation of additional game states and services. Simplifying the addition of a new game state can be achieved by making updates directly within the game class, while Dependency Injection (DI) takes care of creating and managing the state's lifespan. This method reduces redundant code and improves the system's modularity. Adding additional services might be simplified by registering them in a centralized area like the BootstrapState and then injecting them where necessary, which would enhance the clear separation of concerns.

Furthermore, investigating the utilization of Dependency Injection frameworks provides research prospects for assessing their influence on the game's performance, scalability, and general design simplicity. Comparative studies can be done to evaluate the advantages and possible drawbacks of utilizing these frameworks in game development, especially in Unity-based projects aimed for mobile platforms.

Future research could investigate more areas beyond the use of DI frameworks:

- Performance Optimization: Refining and optimizing the game's performance by focusing on advanced rendering techniques, asset loading tactics, and memory management approaches.
- Scalability testing involves expanding the evaluation to include a broader variety of devices and network situations to confirm that the game's structure can accommodate an increasing number of players and changing gameplay elements.

- Player Experience: Administering user research to collect input on the game's usability, engagement, and general satisfaction, informing subsequent improvements to the gaming mechanics and user interface design.

The suggestions for future work emphasize the continuous process of game development, stressing the significance of adopting new technologies, processes, and player feedback to consistently improve the game's structure and player experience.

6 Conclusion

This thesis focused on developing a scalable mobile game architecture by combining theoretical knowledge with actual applications. The major goal was to develop a game architecture that fulfills the dynamic needs of contemporary mobile gaming while demonstrating scalability, performance optimization, and maintainability.

Extensive research and development led to the adoption of various design patterns and technologies, including Model-View-Presenter (MVP) for separating concerns, Service-Oriented Architecture (SOA) for integrating independent services, and the use of independent services. The decisions were crucial in creating a strong prototype that showed the practicality and efficiency of the suggested design in dealing with scalability and performance enhancement issues.

The practical aspect of the thesis involved creating a mobile game prototype using Unity to apply the theoretical topics addressed. After implementing optimization techniques, this prototype was extensively tested and showed notable enhancements in performance metrics. The results confirm the effectiveness of the suggested design and optimization techniques, emphasizing their ability to aid in creating scalable and high-performing mobile games.

The thesis suggests exploring Dependency Injection frameworks to improve the manageability of components and services, as well as advanced performance optimization strategies for future work. The recommendations are intended to stimulate additional research and development, facilitating the creation of more advanced and expandable mobile game structures.

This thesis enhances the existing expertise in mobile game development by introducing a prototype of a scalable game architecture. It shows that by using meticulous architectural design, incorporating suitable design patterns, and applying optimization techniques, developers may successfully navigate the challenges of contemporary mobile game production.

7 References

Books

ALLS, Jason. Clean Code in C#: Refactor your legacy C# code base and improve application performance by applying best practices. Birmingham: Packt Publishing, 2020. 487 p. ISBN 978-1838982973

BARON, David. Game Development Patterns with Unity 2021: Explore practical game development using software design patterns and best practices in Unity and C#. Birmingham: Packt Publishing, 2021. 232 p. ISBN 978-1800200814

FOWLER, Martin. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley Professional, 2018. 455 p. ISBN 978-0134757599

MARTIN, Robert. Clean Architecture: A Craftsman's Guide to Software Structure and Design. London: Pearson, 2017. 436 p. ISBN 978-0134494166

NYSTROM, Robert. Game Programming Patterns. 456 p. Genever Benning, 2014. ISBN 978-0990582908

Websites and website posts

Avinetworks. Service-Oriented Architecture [online]. Available at: https://avinetworks.com/glossary/service-oriented-architecture. Accessed 21 March 2024.

GameAnalytics. What are Remote Configs? [online]. Available at: https://docs.gameanalytics.com/features/remote-configs/faq. Accessed 21 March 2024.

Google. Firebase A/B Testing [online]. Available at: https://firebase.google.com/docs/ab-testing. Accessed 21 March 2024.

MIJUSKOVIC, Veselin. AWS Gaming Guide For Game Development Companies: How To Easily Build Scalable Web Architectures [online]. March 16, 2021. Available at: https://superadmins.com/building-scalable-cloud-architecture-aws-gaming. Accessed 21 March, 2024.

SHVETS,Alexander. Dive Into Design Patterns [online]. Pamplona: Refactoring.Guru, 2018. 406 p. Available at: https://refactoring.guru/design-patterns/book. Accessed 21 March, 2024.

Unity. Unity powers over 69% of the top mobile games [online]. Available at: https://unity.com/solutions/mobile. Accessed 21 March 2024.