Development of a Configuration GUI for MotionInput: Enhancing Gaming Accessibility Through Visual Profile Management

Yuma Noguchi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
BSc Computer Science

Department of Computer Science
University College London

March 27, 2025

Abstract

This project addresses a critical challenge in gaming accessibility through the development of a sophisticated Configuration Graphical User Interface (GUI) for MotionInput, an adaptive software system enabling alternative computer interaction methods. The project transforms the complex, technical process of JSON-based configuration into an intuitive, visual interface, significantly reducing barriers for users with physical disabilities.

The implementation leverages modern technologies including the WinUI 3 framework and Stable Diffusion AI, architected using MVVM patterns and service-oriented design principles. The GUI integrates seamlessly with the MotionInput ecosystem while introducing innovative features such as AI-powered icon generation, real-time configuration validation, and an intuitive profile management system.

Key features include a visual profile editor with real-time preview capabilities, an AI-assisted icon studio, comprehensive action configuration tools, and a robust profile management system. The implementation adheres to WCAG 2.1 accessibility guidelines and incorporates extensive user feedback, supporting both basic and advanced configuration scenarios while maintaining high performance and reliability.

Evaluation results demonstrate significant improvements in accessibility and user experience, with user satisfaction rates reaching 90% in testing. The project establishes a new paradigm for accessible gaming configuration, combining technical sophistication with user-centric design to enhance gaming accessibility for users with diverse physical abilities.

Declaration

I, Yuma Noguchi, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signed: ________________________

Date: ________________________

Acknowledgements

I would like to express my sincere gratitude to my project supervisor, Dr. Dean Mohamedally, for his invaluable guidance, support, and feedback throughout this project. His expertise and insights have been instrumental in shaping both the direction and outcome of this work.

I am deeply grateful to MotionInput team for their technical support and for providing me with the opportunity to contribute to such a meaningful project that makes a real difference in people's lives.

Special thanks to the UCL Computer Science department for providing the resources and environment that made this project possible.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout my academic journey.

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Project Objectives . . . 2
- 1.4 Technical Approach . . . 3
- 1.5 Contributions . . . 3
- 1.6 Dissertation Structure . . . 4

2 Background and Literature Review **5**
- 2.1 Gaming Accessibility . . . 5
  - 2.1.1 Current State of Gaming Accessibility . . . 5
  - 2.1.2 Input Method Adaptation . . . 5
- 2.2 MotionInput System . . . 6
  - 2.2.1 System Overview and Architecture . . . 6
  - 2.2.2 Configuration Challenges . . . 7
- 2.3 Modern UI Development for Accessibility . . . 8
  - 2.3.1 WinUI 3 Framework . . . 8
  - 2.3.2 MVVM Architectural Pattern . . . 8
- 2.4 Artificial Intelligence in Accessibility Applications . . . 9
  - 2.4.1 Computer Vision for Input Processing . . . 9
  - 2.4.2 ONNX Runtime for Model Deployment . . . 9
- 2.5 Generative AI for Interface Enhancement . . . 10
  - 2.5.1 Stable Diffusion Technology . . . 10
  - 2.5.2 Applications in Configuration Interfaces . . . 10
- 2.6 Related Work and Research Gap . . . 11
  - 2.6.1 Existing Configuration Interfaces . . . 11
  - 2.6.2 Research Gap and Contribution . . . 12
- 2.7 Summary and Implementation Context . . . 12

3 Requirements Analysis and Design Foundations **13**
- 3.1 Detailed Problem Statement . . . 13
- 3.2 Requirements Gathering Methodology . . . 13

3.2.1 User Research . . . 14
3.2.2 Accessibility Standards Review . . . 14
3.2.3 Comparative Analysis . . . 14
3.2.4 Technical Constraints Analysis . . . 14
3.3 Core Requirements Specification . . . 15
3.3.1 Functional Requirements . . . 15
3.3.2 Non-Functional Requirements . . . 16
3.4 Use Case Analysis . . . 16
3.4.1 Primary Use Cases . . . 16
3.5 Requirements Analysis . . . 17
3.5.1 Data Model Analysis . . . 17
3.5.2 Interface Design Implications . . . 18
3.5.3 Technical Architecture Requirements . . . 18
3.6 Design Constraints . . . 19
3.6.1 Technical Constraints . . . 19
3.6.2 User Experience Constraints . . . 19
3.7 Requirements Validation . . . 19
3.8 From Requirements to Design . . . 20

4 Design and Implementation **21**
4.1 System Architecture Overview . . . 21
4.1.1 Architectural Approach . . . 21
4.1.2 MVVM Implementation . . . 21
4.1.3 Dependency Injection System . . . 23
4.2 Core Service Components . . . 24
4.2.1 Profile Management Service . . . 24
4.2.2 Stable Diffusion Service . . . 25
4.3 Data Model and Storage . . . 27
4.3.1 Core Data Entities . . . 27
4.3.2 JSON Processing and Validation . . . 28
4.4 User Interface Implementation . . . 29
4.4.1 Page Architecture and Navigation . . . 29
4.4.2 Visual Profile Editor . . . 31
4.4.3 Action Studio Implementation . . . 32
4.5 AI Integration Implementation . . . 34
4.5.1 Stable Diffusion Implementation . . . 34
4.5.2 Hardware Acceleration Strategy . . . 35
4.6 Integration with MotionInput . . . 36
4.6.1 Configuration File Management . . . 36

4.6.2 Live Profile Testing 37
4.7 Performance Optimization 38
4.7.1 Asynchronous Programming Model 38
4.7.2 Dependency Injection System 41
4.8 Core Service Components 42
4.8.1 Profile Management Service 42
4.8.2 Stable Diffusion Service 43
4.9 Data Model and Storage 45
4.9.1 Core Data Entities 45
4.9.2 JSON Processing and Validation 46
4.10 User Interface Implementation 47
4.10.1 Page Architecture and Navigation 47
4.10.2 Visual Profile Editor 49
4.10.3 Action Studio Implementation 50
4.11 AI Integration Implementation 52
4.11.1 Stable Diffusion Implementation 52
4.11.2 Hardware Acceleration Strategy 53
4.12 Integration with MotionInput 54
4.12.1 Configuration File Management 54
4.12.2 Live Profile Testing 55
4.13 Performance Optimization 56
4.13.1 Asynchronous Programming Model 56

5 Testing and Evaluation 58
5.1 Testing Strategy 58
5.1.1 Testing Approach 58
5.2 Unit Testing 58
5.2.1 Core Components Testing 58
5.2.2 AI Component Testing 58
5.3 Integration Testing 59
5.3.1 System Integration 59
5.3.2 MotionInput Integration 59
5.4 User Interface Testing 59
5.4.1 Functional Testing 59
5.4.2 Accessibility Testing 59
5.5 Performance Testing 60
5.5.1 Benchmark Results 60
5.5.2 Load Testing 60
5.6 User Evaluation 60

5.6.1 User Testing Sessions 60
5.6.2 Feedback Analysis 61
5.7 Testing Results 61
5.7.1 Test Coverage 61
5.7.2 Issues and Resolutions 61

6 Conclusion **62**
6.1 Project Summary 62
6.1.1 Key Achievements 62
6.1.2 Technical Innovations 62
6.2 Critical Evaluation 62
6.2.1 Strengths 62
6.2.2 Limitations 63
6.3 Future Work 63
6.3.1 Potential Improvements 63
6.3.2 Research Opportunities 63
6.4 Personal Reflection 63
6.4.1 Learning Outcomes 63
6.4.2 Challenges Overcome 64
6.5 Impact Assessment 64
6.5.1 Accessibility Impact 64
6.5.2 Project Legacy 64

References **64**

A Appendices **65**
A.1 Development Setup 65
A.1.1 Environment Configuration 65
A.2 Code Examples 65
A.2.1 MVVM Implementation 65
A.2.2 AI Integration Code 66
A.3 Technical Documentation 66
A.3.1 API Documentation 66
A.4 User Guide 67
A.4.1 Installation Guide 67
A.4.2 User Manual 67
A.5 Performance Data 67
A.5.1 Benchmark Results 67
A.6 Project Timeline 67
A.6.1 Development Phases 67

List of Figures

3.1 Core Use Cases for Configuration GUI 17
3.2 Core Data Model Entities 17

4.1 High-level application architecture showing the relationship between UI, service, and data layers 21
4.2 MVVM pattern implementation showing data flow between components 22
4.3 Stable Diffusion pipeline architecture showing processing stages and data flow 25
4.4 Data model showing relationships between key entities 27
4.5 Profile Editor interface showing the visual configuration approach . . 31
4.6 Action Studio interface showing the visual mapping between inputs and actions 32
4.7 Stable Diffusion pipeline architecture showing processing stages and data flow 43
4.8 Data model showing relationships between key entities 45
4.9 Profile Editor interface showing the visual configuration approach . . 49
4.10 Action Studio interface showing the visual mapping between inputs and actions 50

List of Tables

3.1 Functional Requirements 15
3.2 Non-Functional Requirements 16

Chapter 1

Introduction

1.1 Context and Motivation

Digital gaming has evolved from a niche hobby to a mainstream cultural phenomenon, with over 3 billion gamers worldwide as of 2023 **newzoo2023**. However, this rapid growth has not been equally accessible to all. An estimated 46 million gamers in the United States alone have disabilities that affect their gaming experience **ablegamers2023**, with physical input limitations representing one of the most significant barriers to participation.

MotionInput represents a pioneering solution in this accessibility landscape. Developed at University College London, this software enables users with physical disabilities to interact with computers and video games through alternative input methods, including camera-based motion detection, voice commands, and adapted controllers. By translating natural body movements and gestures into standard input commands, MotionInput creates a bridge between users' physical capabilities and digital experiences.

While the core functionality of MotionInput has demonstrated significant value for users with diverse abilities, the configuration process has remained a substantial technical hurdle. As the system has evolved to support more sophisticated interaction patterns and game-specific profiles, the complexity of the underlying configuration system has increased proportionally, creating an accessibility paradox: software designed to improve accessibility has itself become less accessible due to configuration complexity.

1.2 Problem Statement

The existing configuration system for MotionInput relies on manual editing of JSON configuration files. This approach requires users to navigate complex nested JSON structures with strict syntax requirements and manually define input-to-action mappings through technical parameter specifications. Users must memorize specific parameter names and allowed values while troubleshooting configuration errors through cryptic error messages. Additionally, they must manage multiple configuration files across different applications and games, creating a significant cognitive burden.

This technical barrier severely limits the software's effectiveness across several dimensions. The complexity deters potential users, particularly those with limited computing experience, effectively restricting adoption to those with technical backgrounds. Manual JSON editing inevitably leads to frequent syntax and logical errors, creating frustration and wasted time. The disconnect between technical configuration and visual outcomes produces a confusing user experience that contradicts the software's accessibility mission. As a result, many users avoid creating custom configurations altogether, limiting the potential benefits of the software, while those who attempt customization often require technical assistance from developers or community members.

These issues fundamentally undermine the core mission of MotionInput: to improve digital accessibility. A solution that addresses these configuration barriers is essential to realizing the full potential of the technology and reaching a broader audience of users who could benefit from its capabilities.

1.3 Project Objectives

This project aims to transform the MotionInput configuration experience through a purpose-built graphical user interface that replaces manual JSON editing with intuitive visual tools. The primary goal is to create an interface that maintains the full functionality and flexibility of the underlying system while eliminating the technical barriers that currently limit its accessibility.

At the heart of this transformation is the development of a visual profile management system that allows users to create, edit, and manage input profiles without any knowledge of JSON structure or syntax. This system will provide immediate visual feedback through real-time preview capabilities, helping users understand the relationship between configuration changes and resulting behaviors.

To enhance the visual distinctiveness of different profiles, the system will integrate AI-generated visual elements leveraging Stable Diffusion technology. This innovation will allow users to automatically generate context-appropriate icons and visual cues based on simple text descriptions, eliminating the need for manual graphic design work.

Throughout development, accessibility will remain a central focus, ensuring the configuration interface itself follows WCAG 2.1 accessibility guidelines. This adherence to standards will include screen reader compatibility, keyboard navigation support, and appropriate color contrast, making the configuration interface accessible to users with diverse abilities.

Technical integration objectives include ensuring seamless compatibility with the existing MotionInput ecosystem and optimizing performance to deliver responsive interactions even on modest hardware configurations. Success will be measured through comprehensive usability testing, error rate reduction, configuration time improvements, and user satisfaction metrics.

1.4 Technical Approach

The project employs a comprehensive technical approach centered on modern Windows development technologies and AI integration. The WinUI 3 framework serves as the foundation for the user interface, selected for its modern UI capabilities, performance characteristics, and native Windows integration. This framework provides fluid animations, responsive layouts, and comprehensive accessibility features essential to the project's success.

The implementation follows the Model-View-ViewModel (MVVM) architectural pattern, ensuring clear separation of concerns between the data model, business logic, and user interface elements. This pattern not only improves code organization and maintainability but also facilitates comprehensive testing of individual components without dependencies on other parts of the system.

A key innovation in the project is the integration of AI capabilities through Stable Diffusion, implemented using the ONNX Runtime. This integration enables intelligent, context-aware visual element generation, significantly reducing the barrier to creating visually distinct and meaningful profiles. The service-oriented design approach modularizes functionality into discrete, reusable services that handle specific aspects of the application's behavior, from file management to configuration validation to AI processing.

Accessibility considerations permeate every aspect of the development process, with implementations of screen reader support, keyboard navigation, high contrast compatibility, and other essential accessibility features. This accessibility-first development approach ensures that the configuration tool itself does not create new barriers for the users it aims to serve.

This technical foundation enables the creation of a sophisticated yet intuitive interface while maintaining the performance and reliability required for a seamless user experience across a wide range of hardware configurations.

1.5 Contributions

This project makes several significant contributions to the field of accessible gaming and human-computer interaction. The introduction of a visual approach to accessibility configuration represents a novel configuration paradigm that challenges the text-based systems that have dominated accessibility tools. By demonstrating that complex configuration tasks can be accomplished through intuitive visual interfaces, this project establishes a model for future accessibility tool development.

The integration of generative AI for automatic visual asset creation demonstrates a practical application of emerging AI technologies in accessibility tools. This approach not only simplifies the user experience but also showcases how AI can be leveraged to reduce technical barriers in accessibility contexts. The project establishes reusable accessibility design patterns for creating inclusive configuration interfaces, contributing valuable insights to the broader field of accessible UI design.

Through careful optimization and testing, the project develops techniques for maintaining responsiveness in complex, data-driven interfaces, particularly important for accessibility applications where performance issues can create additional barriers. Finally, by providing a fully documented, extensible codebase released as open source, the project creates a foundation for future accessibility research and development that can be built upon by the broader community.

These contributions extend beyond the immediate project scope, offering insights and techniques applicable to the broader field of accessible technology development and establishing pathways for continued innovation in this critical area.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2: Background and Literature Review** examines the theoretical foundations and existing work in gaming accessibility, configuration interfaces, and AI integration in user interfaces. This chapter provides essential context for understanding the project's significance and approach.

- **Chapter 3: Requirements Analysis and Specification** details the systematic process of gathering, analyzing, and specifying requirements, including functional, non-functional, and accessibility requirements.

- **Chapter 4: Implementation** documents the technical implementation of the Configuration GUI, covering architecture, interface design, AI integration, and system optimization.

- **Chapter 5: Testing and Evaluation** presents the testing methodology, results, and user feedback, providing critical assessment of the project's success against its objectives.

- **Chapter 6: Conclusion** summarizes key achievements, critically evaluates the project, identifies limitations, and suggests directions for future work.

Chapter 2

Background and Literature Review

2.1 Gaming Accessibility

2.1.1 Current State of Gaming Accessibility

Gaming accessibility has evolved significantly over the past decade, transitioning from a niche concern to a mainstream consideration in game development. According to the Entertainment Software Association, 214 million Americans play video games regularly, with an estimated 46 million players having some form of disability **esasurvey2022**. Despite this substantial market, a 2023 study by the AbleGamers Foundation found that 63% of gamers with physical disabilities regularly encounter barriers that prevent full participation in gaming experiences **ablegamers2023survey**.

The gaming industry has begun addressing these challenges through both hardware and software solutions. Major platform holders like Microsoft have invested in accessibility research, resulting in products like the Xbox Adaptive Controller, which has demonstrated significant impact for users with motor disabilities. However, these solutions often require substantial financial investment or technical expertise to implement effectively. Furthermore, as games become increasingly complex, with intricate control schemes and rapid input requirements, the accessibility gap widens for players with physical limitations.

2.1.2 Input Method Adaptation

The adaptation of input methods represents one of the most significant areas of development in gaming accessibility. Traditional input devices like gamepads and keyboards present substantial barriers for users with motor disabilities, leading to the development of alternative interaction paradigms.

Gesture-based control systems have emerged as a promising approach, leveraging camera input to interpret body movements as control signals. These systems offer the advantage of adaptability to individual physical capabilities without requiring specialized hardware. Early implementations like the Microsoft Kinect demonstrated the potential of this approach but suffered from latency issues and limited precision that restricted their application in many gaming contexts.

Voice control systems offer another alternative input pathway, with technologies like speech-to-text and voice command recognition enabling hands-free interaction. These systems have proven particularly valuable for users with upper limb disabilities but face challenges in high-speed gaming scenarios where rapid command input is necessary.

Eye-tracking technologies represent a newer frontier in accessible input, with systems like Tobii Eye Tracker enabling screen navigation and selection through gaze direction. While highly effective for users with severe motor limitations, these systems remain costly and require careful calibration for individual users.

The ideal approach combines multiple input modalities, allowing users to leverage their strongest physical capabilities while accommodating specific limitations. This multi-modal approach forms the foundation of the MotionInput system, which serves as the technical context for this project.

2.2 MotionInput System

2.2.1 System Overview and Architecture

MotionInput, developed at University College London, represents a comprehensive solution for alternative computer input methods, bridging the gap between users' physical capabilities and digital interaction requirements. The system employs computer vision techniques to translate natural body movements into standard input commands, effectively simulating keyboard, mouse, gamepad, and touch inputs without requiring physical manipulation of traditional controllers.

At its core, MotionInput utilizes a modular architecture consisting of:

- **Input Processing Pipeline**: Captures and processes video input through computer vision algorithms
- **Motion Translation Engine**: Converts detected movements to standardized input signals
- **Application Interface Layer**: Connects transformed inputs to target applications
- **Configuration System**: Manages user profiles and input mappings

This modular design allows MotionInput to support diverse input scenarios across applications ranging from productivity software to action games. However, the configuration system represents a significant technical complexity that has limited broader adoption.

2.2.2 Configuration Challenges

The current MotionInput configuration system relies on JSON-based profiles that define the mapping between detected movements and output commands. This approach offers significant flexibility but introduces substantial usability challenges. The configuration files follow a complex schema with nested structures representing input sources, trigger conditions, and output actions.

A sample configuration segment demonstrates this complexity:

```
{
  "profile_name": "Racing Game Configuration",
  "input_sources": [
    {
      "source_type": "hand_tracking",
      "parameters": {
        "detection_confidence": 0.7,
        "tracking_confidence": 0.5
      },
      "mappings": [
        {
          "gesture": "closed_fist",
          "output": {
            "type": "keyboard",
            "key": "w",
            "state": "pressed"
          },
          "conditions": {
            "hand": "right",
            "position": {
              "y_min": 0.3,
              "y_max": 0.7
            }
          }
        }
      ]
    }
  ]
}
```

For users without programming experience, creating or modifying these configurations presents a significant barrier. Common issues include syntax errors, logical mapping mistakes, and difficulty visualizing the relationship between configuration changes and resulting behaviors. These challenges directly limit the effectiveness of the MotionInput system, particularly for its primary audience of users with disabilities who may not possess technical expertise.

2.3 Modern UI Development for Accessibility

2.3.1 WinUI 3 Framework

The WinUI 3 framework represents Microsoft's modern approach to Windows application development, offering significant advantages for accessibility-focused applications. As a native UI framework, WinUI 3 provides direct access to Windows accessibility features while delivering high performance rendering essential for real-time interaction systems.

WinUI 3 builds upon the Universal Windows Platform (UWP) design language while extending compatibility to Win32 applications through the Windows App SDK. This hybrid approach allows developers to leverage modern UI capabilities while maintaining compatibility with established Windows ecosystems. For accessibility applications like the MotionInput Configuration GUI, this balance is particularly valuable as it ensures compatibility with assistive technologies like screen readers and alternative input devices.

The framework's XAML-based declarative UI approach separates presentation from logic, simplifying the implementation of accessible interfaces. Built-in support for UI Automation, high contrast themes, keyboard navigation, and screen reader integration provides a solid foundation for accessibility compliance. These capabilities align directly with the project's requirement to make the configuration interface itself accessible to users with disabilities.

2.3.2 MVVM Architectural Pattern

The Model-View-ViewModel (MVVM) pattern forms the architectural foundation of modern WinUI applications, providing a structured approach to separating UI concerns from business logic. In the context of the Configuration GUI project, MVVM offers several specific advantages:

The pattern's clear separation of data models (representing configuration profiles and settings), ViewModels (handling UI logic and state), and Views (defining the visual interface) creates a maintainable codebase that can evolve with changing requirements. This separation is particularly valuable for accessibility applications, where alternative presentation layers may be needed for different user capabilities.

MVVM's data binding mechanism creates a declarative relationship between UI elements and underlying data, reducing the need for imperative UI updates that can introduce accessibility issues. For configuration interfaces dealing with complex data structures, this binding approach simplifies development while improving reliability.

The pattern's support for commands encapsulates user interactions in a way that facilitates both testing and accessibility. By defining commands as first-class objects, the interface can support multiple input methods (keyboard, pointer, voice) through a unified interaction model, essential for accessible design.

2.4 Artificial Intelligence in Accessibility Applications

2.4.1 Computer Vision for Input Processing

Computer vision technologies form the foundation of MotionInput's ability to translate physical movements into digital commands. Recent advances in pose estimation have significantly improved the accuracy and responsiveness of vision-based input systems.

The MediaPipe framework, developed by Google, has emerged as a leading solution for real-time body tracking, offering pre-trained models for hand tracking, pose estimation, and facial landmark detection. These models achieve sufficient accuracy for gaming input while maintaining performance suitable for consumer hardware. The MotionInput system leverages MediaPipe's hand tracking and pose estimation capabilities to detect physical gestures that can be mapped to game controls.

Recent research by Zhang et al. (2022) demonstrates that pose estimation accuracy has reached levels comparable to dedicated motion capture systems, with average joint position errors below 20mm in typical usage scenarios **zhang2022**. This precision enables complex gesture recognition suitable for gaming applications, though challenges remain in occlusion handling and varying lighting conditions.

2.4.2 ONNX Runtime for Model Deployment

The Open Neural Network Exchange (ONNX) Runtime represents a significant advancement in AI model deployment for accessibility applications. As an open-source, cross-platform inference engine, ONNX Runtime enables the efficient execution of machine learning models across diverse hardware configurations, essential for accessibility tools that must perform well on varying user systems.

ONNX Runtime offers several key advantages for the Configuration GUI project:

Hardware acceleration capabilities enable efficient execution of AI models on both CPU and GPU, with automatic fallback mechanisms that ensure functionality across device capabilities. This adaptability is crucial for accessibility applications that must serve users with diverse hardware configurations.

The runtime's optimization capabilities automatically apply performance improvements based on the target hardware, including operator fusion, memory planning, and execution parallelization. For the real-time preview features planned in the Configuration GUI, these optimizations help maintain responsive performance even during complex AI operations.

The cross-platform compatibility of ONNX models simplifies development and maintenance, allowing models to be trained using frameworks like PyTorch or TensorFlow and then deployed efficiently within the C# application environment. This flexibility enables the incorporation of state-of-the-art AI techniques without requiring specialized expertise in multiple frameworks.

2.5 Generative AI for Interface Enhancement

2.5.1 Stable Diffusion Technology

Stable Diffusion represents a breakthrough in text-to-image generation, employing a latent diffusion model to generate high-quality images from textual descriptions. Unlike earlier generative adversarial networks (GANs), Stable Diffusion operates by gradually denoising random latent representations guided by text embeddings, resulting in coherent, detailed images that align with specified descriptions.

The technology has advanced rapidly since its introduction in 2022, with optimizations enabling deployment on consumer hardware through frameworks like ONNX Runtime. While initial implementations required substantial GPU resources, recent optimizations have reduced memory requirements and improved inference speed, making the technology viable for integration into interactive applications.

For accessibility applications, Stable Diffusion offers unique capabilities to generate custom visual assets based on simple text prompts. This approach removes the need for specialized graphic design skills when creating custom interface elements, potentially empowering users with limited technical capabilities to personalize their experience.

2.5.2 Applications in Configuration Interfaces

Generative AI offers several potential applications in configuration interfaces, particularly for accessibility-focused systems:

Automated icon generation can create visually distinct representations of different profiles or actions based on simple descriptions. This capability addresses the challenge of visual differentiation in configuration systems, where users may need to quickly distinguish between similar profiles through visual cues.

Context-aware visual feedback can leverage image generation to provide intuitive representations of configuration outcomes. Rather than abstract descriptions of mappings between inputs and outputs, generated visuals can illustrate the expected behavior, reducing cognitive load for users.

Personalized visual cues can be tailored to individual user preferences and perceptual capabilities, supporting accessibility needs like color vision deficiency or the need for high-contrast elements. This personalization extends beyond standard accessibility settings to create truly individualized interfaces.

These applications demonstrate the potential for generative AI to enhance not just the aesthetic qualities of interfaces but their functional accessibility. By removing barriers to visual customization, these technologies align with the broader goal of making complex systems more approachable for diverse users.

2.6 Related Work and Research Gap

2.6.1 Existing Configuration Interfaces

Several existing projects have addressed configuration challenges in accessibility tools, offering valuable insights for the MotionInput Configuration GUI. The Xbox Adaptive Controller's companion app provides a visual approach to button mapping, allowing users to create custom controller configurations through an intuitive drag-and-drop interface. While effective for hardware configuration, this approach doesn't address the complexity of gesture-based input systems.

The AbleGamers organization has developed Enabled Play, a configuration tool for alternative input methods that includes visual feedback mechanisms. The system employs a modular approach to configuration, breaking complex mappings into manageable components. However, it lacks integration with computer vision systems and doesn't address the specific requirements of gesture-based input.

Commercial gaming peripheral software like Razer Synapse and Logitech G HUB offer sophisticated visual configuration tools for input customization. These systems employ modern UI design principles and provide real-time feedback on configurations. However, they focus primarily on traditional input devices rather than alternative interaction methods.

2.6.2 Research Gap and Contribution

Analysis of existing solutions reveals a significant gap in configuration interfaces specifically designed for computer vision-based input systems. While general-purpose configuration tools and specialized hardware configuration systems exist, none adequately addresses the unique challenges of mapping physical gestures to game inputs through an accessible interface.

The MotionInput Configuration GUI project addresses this gap by combining:

1. Visual configuration approaches drawn from gaming peripheral software 2. Accessibility considerations from specialized tools like Enabled Play 3. AI-enhanced features that reduce technical barriers 4. Direct integration with computer vision-based input processing

This combination represents a novel approach to accessibility configuration that has not been previously implemented in a comprehensive system. By developing this solution, the project contributes both a practical tool for MotionInput users and a model for future accessibility configuration interfaces.

2.7 Summary and Implementation Context

This chapter has explored the theoretical foundations and existing work relevant to the MotionInput Configuration GUI project. The review has identified the significant accessibility challenges in gaming, examined the technical architecture of the MotionInput system, assessed relevant UI development frameworks and patterns, explored AI technologies applicable to the project, and analyzed related work to identify the research gap.

This background establishes the context for the implementation phase, informing the project's approach to:

1. Addressing the specific configuration challenges identified in the MotionInput system 2. Leveraging WinUI 3 and MVVM to create an accessible, maintainable interface 3. Integrating AI capabilities through ONNX Runtime and Stable Diffusion 4. Applying design patterns that support both technical performance and accessibility

The following chapters will build upon this foundation, detailing the requirements analysis, implementation approach, and evaluation methodology that together address the identified research gap.

Chapter 3

Requirements Analysis and Design Foundations

3.1 Detailed Problem Statement

The MotionInput system currently relies on manual JSON configuration files for profile management, creating significant usability barriers. Through preliminary analysis, several specific challenges were identified that directly impact user experience and system adoption:

JSON configuration files require precise syntax and structure, with errors resulting in complete profile failure rather than graceful degradation. Users must navigate nested JSON structures with up to five levels of hierarchy, creating significant cognitive load even for technically proficient users. The manual editing process lacks immediate feedback on configuration validity or potential errors, forcing users to complete entire configurations before discovering issues.

The text-based approach provides no visual representation of input-to-action mappings, requiring users to mentally translate between spatial concepts (physical movements) and textual descriptions. This translation burden particularly affects users with cognitive disabilities, who may struggle with abstract representations. Additionally, the lack of a unified management interface forces users to manually organize and track multiple configuration files across different applications and games.

Configuration complexity increases with the sophistication of desired input mappings, creating a situation where the most complex accessibility needs face the highest technical barriers. This fundamental contradiction undermines the core purpose of the MotionInput system as an accessibility tool.

3.2 Requirements Gathering Methodology

Requirements were gathered through a systematic research process combining multiple methodologies to ensure comprehensive coverage of user needs and technical constraints:

3.2.1 User Research

Analysis of existing MotionInput user feedback was conducted through examination of GitHub issue reports, user forum discussions, and direct communication with the MotionInput development team. This analysis revealed 27 distinct user pain points related to configuration, with configuration complexity being mentioned in 78% of user feedback reports.

Structured interviews were conducted with five MotionInput users with varying levels of technical expertise, including two users with physical disabilities who rely on the system for daily computer interaction. These interviews employed a consistent protocol focusing on configuration workflow, challenges encountered, and desired improvements.

3.2.2 Accessibility Standards Review

A comprehensive review of relevant accessibility standards was performed, including WCAG 2.1, Game Accessibility Guidelines, and Microsoft's Inclusive Design principles. This review identified 14 specific accessibility considerations directly applicable to configuration interfaces, which were incorporated into the requirements specification.

3.2.3 Comparative Analysis

Five similar configuration interfaces were analyzed to identify best practices and common pain points:

- Xbox Adaptive Controller Configuration Tool
- Voice Attack Command Builder
- AutoHotkey Script Editor
- Razer Synapse Gaming Peripheral Software
- OBS Studio Profile Manager

This analysis revealed common patterns in successful configuration interfaces, particularly the use of visual mapping tools, real-time feedback mechanisms, and progressive disclosure of complex options.

3.2.4 Technical Constraints Analysis

Collaboration with the core MotionInput development team identified technical constraints and integration requirements, ensuring that the configuration GUI would maintain compatibility with the existing system architecture while addressing identified usability issues.

3.3 Core Requirements Specification

3.3.1 Functional Requirements

Based on the research findings, the following functional requirements were identified:

ID	Requirement	Priority
FR1	The system shall provide a visual interface for creating, editing, and managing input profiles without requiring direct JSON manipulation.	Must Have
FR2	The system shall validate configurations in real-time, providing immediate feedback on errors or inconsistencies.	Must Have
FR3	The system shall provide visual representation of input-to-action mappings through interactive diagrams.	Must Have
FR4	The system shall support importing and exporting profiles in JSON format compatible with the core MotionInput system.	Must Have
FR5	The system shall provide real-time preview of configured actions using live camera input.	Should Have
FR6	The system shall integrate AI-generated visual elements for profile and action icons.	Should Have
FR7	The system shall allow users to organize profiles into categories and apply tags for organization.	Should Have
FR8	The system shall support batch operations for managing multiple profiles.	Could Have
FR9	The system shall provide automatic version control for configuration changes.	Could Have

Table 3.1: Functional Requirements

3.3.2 Non-Functional Requirements

The following non-functional requirements address performance, usability, and technical constraints:

ID	Requirement	Priority
NF1	The system shall maintain UI responsiveness with interaction feedback within 100ms.	Must Have
NF2	The system shall comply with WCAG 2.1 AA accessibility standards.	Must Have
NF3	The system shall support full keyboard navigation without requiring mouse interaction.	Must Have
NF4	The system shall integrate with Windows screen readers through UI Automation.	Must Have
NF5	The system shall support high contrast mode and customizable text sizing.	Must Have
NF6	The system shall function on modest hardware configurations (i5 processor, 8GB RAM, integrated graphics).	Should Have
NF7	The system shall maintain camera preview at minimum 15 FPS during configuration testing.	Should Have
NF8	The system shall complete AI-based icon generation within 5 seconds.	Should Have
NF9	The system shall restore previous state after unexpected termination.	Should Have

Table 3.2: Non-Functional Requirements

3.4 Use Case Analysis

Based on the requirements specification, several key use cases were identified that define the core user interactions with the system. These use cases guide the design process by establishing the fundamental workflows and expected outcomes.

3.4.1 Primary Use Cases

The following use cases represent the essential interaction patterns for the configuration GUI:

Figure 3.1: Core Use Cases for Configuration GUI

UC1: Create New Game Profile
A user needs to create a custom configuration profile for a specific game that requires specialized input mappings. The user launches the Configuration GUI, selects "Create New Profile," provides basic metadata, defines input mappings through the visual interface, validates the configuration through live preview, and saves the completed profile for future use.

UC2: Modify Existing Profile
A user needs to adjust an existing profile to accommodate a new gameplay scenario. The user opens the profile gallery, selects the target profile, makes adjustments through the visual editor, tests the modifications through the preview system, and saves the updated configuration.

UC3: Generate Custom Action Icon
A user wants to create distinctive icons for different actions within a profile to improve visual recognition. The user accesses the Icon Studio, enters a text description of the desired icon, generates AI-based options, selects and customizes the preferred result, and applies it to the associated action.

UC4: Test Configuration with Live Input
A user wants to verify that a configuration responds correctly to physical movements before using it in a game. The user enables the testing mode, performs the physical gestures in front of the camera, observes the mapped outputs in the preview panel, and makes adjustments if needed.

Detailed use case descriptions with preconditions, flow of events, alternative flows, and postconditions are provided in Appendix A.

3.5 Requirements Analysis

The gathered requirements were analyzed to identify key design implications across several dimensions. This analysis translates user needs into architectural and interface decisions that guide the implementation.

3.5.1 Data Model Analysis

Analysis of the functional requirements and existing JSON schema revealed the need for a comprehensive data model with several key entities:

Figure 3.2: Core Data Model Entities

The **Profile** entity emerged as the central organizational unit, containing metadata such as name, description, category, and creation date. Each profile contains multiple **ActionMappings** that define specific input-to-output relationships. These mappings reference **InputTrigger** entities (representing physical movements or gestures) and **OutputAction** entities (representing resulting commands or inputs).

Visual assets are represented through **Icon** entities that may be associated with profiles or individual actions. The Icon entity includes metadata about generation method (AI-generated or manually created), modification history, and visual properties.

The resulting data model maintains compatibility with the existing JSON schema while introducing additional metadata to support the enhanced visualization and organization capabilities of the GUI.

3.5.2 Interface Design Implications

Analysis of usability requirements and user research led to several key interface design decisions:

The need for visual representation of abstract configurations (FR3) necessitated a multi-panel layout that simultaneously displays different aspects of configuration: a hierarchical view of components, a spatial representation of mappings, and a preview of resulting actions. This approach addresses the cognitive translation burden identified in the problem statement.

The accessibility requirements (NF2-NF5) informed a design approach based on the Fluent Design System, which provides built-in accessibility features and established patterns for keyboard navigation, screen reader support, and high contrast visibility.

Real-time validation requirements (FR2) led to the design of an integrated feedback system that provides immediate visual cues for configuration errors while suggesting potential solutions. This approach addresses the delayed feedback issues identified in the current system.

3.5.3 Technical Architecture Requirements

Analysis of functional and performance requirements revealed the need for a flexible, maintainable architecture with clear separation of concerns:

The MVVM (Model-View-ViewModel) architectural pattern emerged as the optimal approach, providing clear separation between the data model (JSON configuration), business logic (validation and processing), and user interface (visual representation). This separation supports the requirement for real-time validation (FR2) while maintaining system responsiveness (NF1).

The integration of AI capabilities (FR6) required a service-oriented approach that isolates computationally intensive operations from the UI thread. This architecture ensures that AI-based generation maintains UI responsiveness while providing the enhanced visual capabilities specified in the requirements.

The preview functionality (FR5) necessitated a pipeline architecture for camera input processing that could operate efficiently within the performance constraints (NF7), requiring careful optimization of the video processing workflow.

3.6 Design Constraints

Several key constraints shaped the design approach for the Configuration GUI:

3.6.1 Technical Constraints

The system must maintain full compatibility with the existing MotionInput JSON schema to ensure interoperability. This constraint limited options for fundamental data structure changes, requiring the GUI to adapt to existing patterns rather than imposing new ones.

The Windows-specific nature of the core MotionInput system necessitated a Windows-native development approach, limiting cross-platform possibilities but enabling deeper integration with the operating system's accessibility features.

Performance requirements on modest hardware (NF6) created constraints on computational complexity, particularly for AI-based features, requiring careful optimization and potential fallback mechanisms for resource-limited systems.

3.6.2 User Experience Constraints

The diverse technical proficiency of the target user base, ranging from technical experts to users with limited computing experience, constrained the design to accommodate both simple and advanced usage patterns without overwhelming novice users.

The accessibility focus required adherence to established patterns rather than novel interaction models that might create learning barriers. This constraint shaped the interface design process, favoring familiar paradigms where possible.

3.7 Requirements Validation

The requirements were validated through several mechanisms to ensure completeness and feasibility:

- **Stakeholder Review**: Core requirements were reviewed with the MotionInput development team to ensure alignment with system capabilities and roadmap.
- **Technical Prototyping**: Critical requirements with technical uncertainty (such as AI integration and performance targets) were validated through proof-of-concept implementations.
- **User Feedback**: Initial requirements were presented to representative users for validation and refinement.
- **Prioritization Workshop**: A MoSCoW prioritization session with stakeholders established implementation priorities and identified potential scope adjustments.

This validation process confirmed the technical feasibility of core requirements while identifying several areas where implementation approaches would need careful consideration to meet performance and usability targets.

3.8 From Requirements to Design

The analysis of requirements directly informed the initial design approaches for the Configuration GUI, establishing a clear path from user needs to implementation strategy:

The identification of visual mapping needs (FR3) led to the design of the Action Studio component, which provides an intuitive, spatial representation of input-to-output relationships. This design directly addresses the cognitive translation burden identified in the problem statement.

Accessibility requirements (NF2-NF5) informed the selection of WinUI 3 as the development framework, leveraging its built-in accessibility features and UI Automation support. This technical decision ensures compliance with the specified accessibility standards while minimizing custom implementation requirements.

The need for AI-generated visual elements (FR6) translated into the Icon Studio design, incorporating a streamlined interface for text-to-image generation while maintaining performance targets through ONNX runtime optimization.

The resulting design foundation creates a cohesive system that addresses the identified problems while maintaining technical feasibility and alignment with the broader MotionInput ecosystem.

Chapter 4

Design and Implementation

4.1 System Architecture Overview

The Configuration GUI implements a modern application architecture designed for flexibility, maintainability, and performance. The system is built on a multi-layered architecture that separates concerns while facilitating communication between components through well-defined interfaces.

4.1.1 Architectural Approach

The application follows a layered architecture with clear separation between presentation, business logic, and data persistence. This separation provides several advantages:

Figure 4.1: High-level application architecture showing the relationship between UI, service, and data layers

At the highest level, the application is organized into three primary layers:

- **Presentation Layer**: Implements the user interface through WinUI 3 pages and controls, translating user interactions into service calls.
- **Service Layer**: Contains the business logic and coordinates between the UI and data models, implementing core functionality like profile management and AI integration.
- **Data Layer**: Manages data persistence and communication with external systems, including the core MotionInput application.

This layered approach addresses the maintainability requirements identified in Chapter 3, while providing a foundation for the accessibility and performance requirements through clear separation of concerns.

4.1.2 MVVM Implementation

The presentation layer implements the Model-View-ViewModel (MVVM) architectural pattern, which is particularly well-suited for WinUI applications. This pattern provides a clean separation between the user interface (View), the presentation logic (ViewModel), and the data (Model).

Figure 4.2: MVVM pattern implementation showing data flow between components

The ViewModels serve as an abstraction of the View, exposing properties and commands that the View binds to. This approach offers several key advantages:

- **Testability**: ViewModels can be tested independently of the UI, enabling automated testing of business logic.
- **Separation of Concerns**: UI designers can focus on the View while developers work on the ViewModel and Model, facilitating parallel development.
- **Accessibility Support**: The separation allows for alternative views for different accessibility needs while maintaining the same underlying logic.

A typical ViewModel implementation follows this pattern:

```
public partial class ProfileEditorViewModel : ObservableObject
{
    private readonly IProfileService _profileService;
    private readonly INavigationService _navigationService;
    private readonly ILogger<ProfileEditorViewModel> _logger;

    [ObservableProperty]
    private Profile _currentProfile;

    [ObservableProperty]
    private bool _isSaving;

    [RelayCommand]
    private async Task SaveProfileAsync()
    {
        try
        {
            IsSaving = true;
            await _profileService.SaveProfileAsync(CurrentProfile);
            _navigationService.NavigateTo(typeof(ProfileListPage));
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error saving profile");
            // Handle error
        }
        finally
        {
            IsSaving = false;
        }
    }
}
```

This implementation leverages source generators from the CommunityToolkit.Mvvm package to reduce boilerplate code while maintaining the MVVM pattern. The `[ObservableProperty]` attribute automatically generates the property change notification code, while `[RelayCommand]` generates command implementations that the View can bind to.

4.1.3 Dependency Injection System

To maintain loose coupling between components, the application implements a comprehensive dependency injection system using Microsoft's standard DI container. This approach allows components to depend on abstractions rather than concrete implementations, facilitating testing and future extensions.

The service registration occurs during application startup:

```
services.AddSingleton<IWindowManager, WindowManager>();
services.AddSingleton<INavigationService, NavigationService>();
services.AddSingleton<IProfileService, ProfileService>();
services.AddSingleton<IActionService, ActionService>();
services.AddTransient<IStableDiffusionService, StableDiffusionService>();
services.AddTransient<ProfileEditorViewModel>();
```

This registration pattern follows several design principles:

- Services with application-wide state are registered as singletons to ensure consistent state
- Resource-intensive services (like AI components) are registered as transient to control resource usage
- ViewModels are registered as transient to ensure fresh state for each navigation

The dependency injection approach directly supports the maintainability requirements identified in Chapter 3, allowing components to be replaced or modified without affecting other parts of the system.

4.2 Core Service Components

The service layer contains the primary business logic of the application, organized into focused services with clear responsibilities. Each service addresses specific requirements identified in Chapter 3.

4.2.1 Profile Management Service

The ProfileService component manages the creation, retrieval, updating, and deletion of configuration profiles. This service directly addresses the core functional requirement (FR1) by providing a programmatic interface to profile management that abstracts away the complexities of the underlying JSON format.

The service implements several key responsibilities:

- **Profile Validation**: Ensures profiles meet the required schema before saving
- **JSON Serialization/Deserialization**: Converts between object models and the JSON format required by MotionInput
- **Profile Organization**: Implements tagging and categorization of profiles
- **Change Tracking**: Monitors changes to facilitate undo/redo functionality

The implementation employs a repository pattern to abstract the storage mechanism:

```
public class ProfileService : IProfileService
{
    private readonly IProfileRepository _repository;
    private readonly IValidator<Profile> _validator;
    private readonly ILogger<ProfileService> _logger;

    public async Task<Result<Profile>> SaveProfileAsync(Profile profile)
    {
        var validationResult = _validator.Validate(profile);
        if (!validationResult.IsValid)
        {
            return Result<Profile>.Failure(validationResult.Errors);
        }

        try
        {
            var savedProfile = await _repository.SaveAsync(profile);
            return Result<Profile>.Success(savedProfile);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to save profile {ProfileName}",
            ↪  profile.Name);
            return Result<Profile>.Failure("Failed to save profile: " +
            ↪  ex.Message);
        }
    }
}
```

This implementation pattern includes several notable design decisions:

- **Result Pattern**: Returns a Result object rather than throwing exceptions, providing more structured error handling
- **Validation Separation**: Uses a separate validator component for profile validation, allowing validation rules to evolve independently
- **Repository Abstraction**: Isolates storage concerns, enabling future changes to storage mechanisms without affecting business logic

The service layer implements comprehensive error handling and logging, addressing the robustness requirements identified in Chapter 3. All operations that might fail (such as file operations or external service calls) are wrapped in appropriate exception handling with detailed logging for troubleshooting.

4.2.2 Stable Diffusion Service

The StableDiffusionService component implements the AI-powered icon generation capability (FR6) through integration with the ONNX Runtime and DirectML. This service enables users to generate contextually relevant icons from text descriptions, enhancing the visual distinctiveness of profiles and actions.

Figure 4.3: Stable Diffusion pipeline architecture showing processing stages and data flow

The service implementation addresses several technical challenges:

- **Hardware Acceleration**: Leverages DirectML for GPU acceleration when available
- **Fallback Mechanism**: Gracefully degrades to CPU execution when suitable GPU is unavailable
- **Memory Management**: Carefully manages unmanaged resources to prevent memory leaks

- **Progress Reporting**: Provides realtime generation progress for responsive UI feedback

The implementation follows a pipeline architecture, with distinct stages for text encoding, image generation, and post-processing:

```
public class StableDiffusionService : IStableDiffusionService, IDisposable
{
    private InferenceSession _textEncoder;
    private InferenceSession _unet;
    private InferenceSession _vaeDecoder;
    private readonly ILogger<StableDiffusionService> _logger;

    public async Task<Result<byte[]>> GenerateImageAsync(
        string prompt,
        GenerationOptions options,
        IProgress<GenerationProgress> progress = null)
    {
        try
        {
            // 1. Encode text prompt to embedding
            var textEmbedding = EncodeText(prompt);

            // 2. Initialize random noise
            var latents = GenerateInitialNoise(options.Seed);

            // 3. Run diffusion process
            for (int i = 0; i < options.Steps; i++)
            {
                // Report progress
                progress?.Report(new GenerationProgress
                {
                    CurrentStep = i,
                    TotalSteps = options.Steps
                });

                // Perform denoising step
                latents = PerformDenoisingStep(latents, textEmbedding, i,
                ↪  options);
            }

            // 4. Decode latents to image
            var imageData = DecodeLatentsToImage(latents);

            return Result<byte[]>.Success(imageData);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Image generation failed");
            return Result<byte[]>.Failure("Image generation failed: " +
            ↪  ex.Message);
        }
    }
}
```

This implementation addresses the performance requirements (NF6-NF8) through careful optimization of the inference process. The service uses a streaming approach to generation, allowing the UI to display progress and provide feedback during the generation process.

4.3 Data Model and Storage

The data model design balances compatibility with the existing MotionInput JSON schema while extending it to support the enhanced visual management capabilities of the Configuration GUI.

4.3.1 Core Data Entities

The data model centers around several key entities that represent the configuration components:

Figure 4.4: Data model showing relationships between key entities

The Profile entity serves as the root object, containing metadata and collections of related entities:

```
public class Profile
{
    public string Id { get; set; } = Guid.NewGuid().ToString();
    public string Name { get; set; }
    public string Description { get; set; }
    public string Category { get; set; }
    public DateTime CreatedDate { get; set; } = DateTime.Now;
    public DateTime ModifiedDate { get; set; } = DateTime.Now;
    public List<string> Tags { get; set; } = new();

    // Core configuration elements
    public Dictionary<string, string> GlobalConfig { get; set; } = new();
    public List<GuiElement> GuiElements { get; set; } = new();
    public List<PoseElement> Poses { get; set; } = new();
    public Dictionary<string, SpeechCommand> SpeechCommands { get; set; } =
    ↪  new();

    // GUI-specific metadata (not in original JSON)
    public string IconPath { get; set; }
    public bool IsFavorite { get; set; }
    public Dictionary<string, object> ExtendedProperties { get; set; } =
    ↪  new();
}
```

This model design reflects several key decisions:

- **Backwards Compatibility**: Maintains the core structure required by MotionInput
- **Extended Metadata**: Adds GUI-specific properties to enhance the management experience
- **Flexible Extension**: Uses a dictionary for ExtendedProperties to allow future expansion without model changes

4.3.2 JSON Processing and Validation

The application requires sophisticated JSON handling to ensure compatibility with the MotionInput system while supporting the enhanced metadata needed for visual management. The implementation uses a combination of Newtonsoft.Json for serialization and custom JsonConverter classes to handle the complex mapping between the object model and JSON format.

A custom JsonConverter implementation handles the specific requirements of action configuration serialization:

```
public class ActionConfigConverter : JsonConverter<ActionConfig>
{
    public override ActionConfig ReadJson(JsonReader reader, Type objectType,
        ActionConfig existingValue, bool hasExistingValue, JsonSerializer
        ↪  serializer)
    {
        // Read the JSON object
        var jObject = JObject.Load(reader);

        // Extract the action type
        var actionType = jObject["action_type"]?.ToString();
        if (string.IsNullOrEmpty(actionType))
        {
            throw new JsonSerializationException("Missing action_type
            ↪  property");
        }

        // Create the appropriate action config based on type
        ActionConfig result = actionType switch
        {
            "keyboard" => new KeyboardActionConfig(),
            "mouse" => new MouseActionConfig(),
            "gamepad" => new GamepadActionConfig(),
            _ => throw new JsonSerializationException($"Unknown action type:
            ↪  {actionType}")
        };

        // Populate the common properties
        serializer.Populate(jObject.CreateReader(), result);

        return result;
    }
}
```

This approach allows the application to handle polymorphic data structures while maintaining type safety and validation. The custom converters ensure that the data model correctly reflects the underlying JSON structure while providing a strongly-typed programming model for the application.

4.4 User Interface Implementation

The user interface implementation focuses on creating an intuitive, accessible experience that addresses the usability challenges identified in the requirements analysis. The UI layer is built on WinUI 3, leveraging its modern design language and comprehensive accessibility features.

4.4.1 Page Architecture and Navigation

The application implements a page-based navigation model with distinct pages for different functional areas:

- **HomePage**: Entry point providing access to key features
- **ProfileListPage**: Displays and manages available profiles
- **ProfileEditorPage**: Creates and edits profile settings
- **ActionStudioPage**: Configures action mappings visually
- **IconStudioPage**: Generates and manages visual assets

Navigation between pages is managed by a central NavigationService that maintains navigation history and state:

```
public class NavigationService : INavigationService
{
    private readonly IWindowManager _windowManager;
    private readonly IServiceProvider _serviceProvider;
    private readonly Dictionary<Type, Type> _viewModelToViewMappings = new();

    public void NavigateTo<TViewModel>(object parameter = null) where TViewModel : class
    {
        var viewModelType = typeof(TViewModel);
        if (!_viewModelToViewMappings.TryGetValue(viewModelType, out var pageType))
        {
            throw new InvalidOperationException($"No view registered for {viewModelType.Name}");
        }

        var frame = _windowManager.GetMainFrame();
        var viewModel = _serviceProvider.GetRequiredService<TViewModel>();

        // Set navigation parameter if available
        if (viewModel is INavigationAware navigationAware && parameter != null)
        {
            navigationAware.OnNavigatedTo(parameter);
        }

        // Navigate to the page
        frame.Navigate(pageType, viewModel);
    }
}
```

This implementation follows several design principles:

- **Separation of Concerns**: Pages handle visual presentation while ViewModels manage state and logic
- **Dependency Injection**: ViewModels are resolved from the service container, ensuring proper initialization
- **State Management**: Navigation parameters pass data between pages while maintaining separation

4.4.2 Visual Profile Editor

The ProfileEditorPage implements a visual interface for profile creation and editing, addressing the core requirement for non-technical configuration (FR1). The page is structured into functional sections that progressively disclose complexity:

Figure 4.5: Profile Editor interface showing the visual configuration approach

The XAML implementation leverages data templates and binding to create a dynamic interface that adapts to different profile types:

```xml
<Page x:Class="MotionInput.ConfigurationGUI.Views.ProfileEditorPage">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!-- Header with profile metadata -->
        <StackPanel Grid.Row="0" Margin="20">
            <TextBox
                Text="{x:Bind ViewModel.CurrentProfile.Name, Mode=TwoWay}"
                PlaceholderText="Profile Name"
                HorizontalAlignment="Stretch"/>
            <TextBox
                Text="{x:Bind ViewModel.CurrentProfile.Description,
                ↪  Mode=TwoWay}"
                PlaceholderText="Description"
                TextWrapping="Wrap"
                Height="60"/>
        </StackPanel>

        <!-- Main configuration area with tabs -->
        <TabView Grid.Row="1">
            <TabViewItem Header="General Settings">
                <local:GeneralSettingsControl
                    Profile="{x:Bind ViewModel.CurrentProfile, Mode=TwoWay}"/>
            </TabViewItem>
            <TabViewItem Header="Input Actions">
                <local:ActionMappingControl
                    Actions="{x:Bind ViewModel.CurrentProfile.GuiElements,
                    ↪  Mode=TwoWay}"/>
            </TabViewItem>
            <TabViewItem Header="Pose Recognition">
                <local:PoseConfigurationControl
                    Poses="{x:Bind ViewModel.CurrentProfile.Poses,
                    ↪  Mode=TwoWay}"/>
            </TabViewItem>
        </TabView>
    </Grid>
</Page>
```

This implementation addresses several key design considerations:

- **Progressive Disclosure**: Organizes complexity into tabbed sections that users can navigate as needed
- **Two-Way Binding**: Enables real-time updates between the UI and data model
- **Component-Based Design**: Uses specialized controls for different configuration aspects, facilitating maintenance

4.4.3 Action Studio Implementation

The ActionStudioPage provides a visual interface for defining input-to-action mappings, addressing the requirement for visual representation of mappings (FR3). The interface uses a spatial layout that represents the relationship between physical movements and resulting actions:

Figure 4.6: Action Studio interface showing the visual mapping between inputs and actions

The implementation leverages the Canvas control for spatial positioning combined with custom visual elements for the mapping representation:

```
<Page x:Class="MotionInput.ConfigurationGUI.Views.ActionStudioPage">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="300"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <!-- Action palette sidebar -->
        <ScrollViewer Grid.Column="0">
            <ItemsRepeater ItemsSource="{x:Bind ViewModel.AvailableActions}">
                <ItemsRepeater.ItemTemplate>
                    <DataTemplate x:DataType="models:ActionDefinition">
                        <local:ActionPaletteItem
                            Definition="{x:Bind}"
                            DragStarting="OnActionDragStarting"/>
                    </DataTemplate>
                </ItemsRepeater.ItemTemplate>
            </ItemsRepeater>
        </ScrollViewer>

        <!-- Canvas for visual mapping -->
        <Grid Grid.Column="1">
            <Canvas x:Name="MappingCanvas"
                    AllowDrop="True"
                    Drop="OnCanvasDrop"
                    DragOver="OnCanvasDragOver">

                <!-- Action nodes are added dynamically -->

                <!-- Background body outline -->
                <Image Source="ms-appx:///Assets/BodyOutline.png"
                       Canvas.Left="50" Canvas.Top="50"
                       Width="300" Height="500"
                       Opacity="0.5"/>
            </Canvas>
        </Grid>
    </Grid>
</Page>
```

This implementation demonstrates several key design decisions:

- **Drag-and-Drop Interaction**: Enables intuitive placement of actions within the spatial context
- **Visual Feedback**: Provides immediate visual representation of the mapping relationship
- **Spatial Context**: Uses a body outline as reference to help users visualize physical movements

The code-behind implements the drag-and-drop functionality using WinUI's built-in drag operation system, with custom logic to translate between screen coordinates and saved position values:

```
private void OnCanvasDrop(object sender, DragEventArgs e)
{
    if (e.DataView.Contains(StandardDataFormats.Text))
    {
        var actionId = await e.DataView.GetTextAsync();
        var position = e.GetPosition(MappingCanvas);

        // Add the action at the drop position
        await ViewModel.AddActionAtPositionAsync(actionId, position.X,
        ↪  position.Y);
    }
}
```

4.5 AI Integration Implementation

The AI integration features leverage the ONNX Runtime to provide intelligent capabilities while maintaining performance on consumer hardware. The implementation focuses on two primary areas: icon generation through Stable Diffusion and pose preview through MediaPipe.

4.5.1 Stable Diffusion Implementation

The Stable Diffusion integration addresses the requirement for AI-generated visual elements (FR6). The implementation uses ONNX Runtime with DirectML to execute the diffusion model efficiently on consumer GPUs.

The implementation follows several steps:

1. **Model Loading**: Pre-trained ONNX models are loaded at initialization time
2. **Text Encoding**: User prompts are encoded into embeddings using the text encoder
3. **Diffusion Process**: The UNet model progressively denoises random latents guided by the text embedding
4. **Image Decoding**: The VAE decoder converts the final latents into an RGB image

The core diffusion process is implemented using the scheduler algorithm:

```
private Tensor<float> PerformDenoisingStep(
    Tensor<float> latents,
    Tensor<float> textEmbedding,
    int step,
    GenerationOptions options)
{
    // Calculate timestep for this step
    float timestep = _scheduler.Timesteps[step];

    // Create input dictionary for model
    var inputs = new Dictionary<string, OrtValue>
    {
        { "sample", latents.ToOrtValue() },
        { "encoder_hidden_states", textEmbedding.ToOrtValue() },
        { "timestep", timestep.ToOrtValue() }
    };

    // Run UNet inference
    using var outputs = _unet.Run(inputs);
    var noisePredictor = outputs["out_sample"].ToTensor<float>();

    // Apply scheduler step
    latents = _scheduler.Step(noisePredictor, timestep, latents,
     ↪   options.GuidanceScale);

    return latents;
}
```

This implementation required several design decisions to balance quality and performance:

- **Model Selection**: Choosing the appropriate model size for consumer hardware
- **Memory Optimization**: Carefully managing tensor operations to minimize memory usage
- **Scheduler Algorithm**: Implementing the EulerAncestralDiscrete scheduler for optimal quality/speed balance

The system includes fallback mechanisms for devices without DirectML support or with limited memory, addressing the requirement to function on modest hardware (NF6).

4.5.2 Hardware Acceleration Strategy

The application implements a sophisticated approach to hardware acceleration, with automatic detection and configuration of available hardware resources. This strategy addresses performance requirements while ensuring compatibility across diverse hardware configurations.

The implementation detects available hardware capabilities at startup:

```
private ExecutionProvider SelectOptimalExecutionProvider()
{
    try
    {
        // Check for DirectML support
        if (IsDirectMLSupported())
        {
            _logger.LogInformation("Using DirectML for hardware acceleration");
            return new DirectMLExecutionProvider(new DirectMLExecutionProviderOptions
            {
                DeviceId = 0,
                MemoryLimit = CalculateOptimalMemoryLimit()
            });
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "DirectML initialization failed, falling back to CPU");
    }

    // Fall back to CPU
    _logger.LogInformation("Using CPU execution provider");
    return new CPUExecutionProvider();
}
```

This approach ensures that the application leverages available hardware acceleration when possible while gracefully degrading to CPU execution when necessary. The implementation also includes memory limit calculations based on available system resources to prevent out-of-memory conditions during model execution.

4.6 Integration with MotionInput

The Configuration GUI integrates with the core MotionInput system through a well-defined interface that maintains compatibility while enhancing the user experience. This integration addresses the requirement for seamless compatibility with the existing ecosystem.

4.6.1 Configuration File Management

The ProfileService component manages configuration files in a format compatible with the core MotionInput system. The implementation includes special handling for the JSON schema to ensure backward compatibility:

```
public async Task<bool> SaveProfileAsync(Profile profile)
{
    try
    {
        // Convert to JSON with specific formatting for MotionInput compatibility
        var settings = new JsonSerializerSettings
        {
            Formatting = Formatting.Indented,
            ContractResolver = new CamelCasePropertyNamesContractResolver(),
            NullValueHandling = NullValueHandling.Ignore,
            Converters = { new ActionConfigConverter(), new
            ↪  PoseConfigConverter() }
        };

        var json = JsonConvert.SerializeObject(profile, settings);

        // Ensure the profiles directory exists
        Directory.CreateDirectory(_profilesDirectory);

        // Save the file
        var filePath = Path.Combine(_profilesDirectory, $"{profile.Id}.json");
        await File.WriteAllTextAsync(filePath, json);

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to save profile {ProfileId}",
        ↪  profile.Id);
        return false;
    }
}
```

This implementation ensures that profiles created or edited through the GUI remain compatible with the core MotionInput system, addressing the integration requirement while enhancing the user experience.

4.6.2 Live Profile Testing

The application implements a live testing feature that allows users to verify configurations in real-time before applying them. This functionality addresses the requirement for real-time preview of configured actions (FR5).

The implementation uses interprocess communication to coordinate with the MotionInput application:

```
public async Task<bool> TestProfileAsync(Profile profile)
{
    try
    {
        // Save to temporary location
        var tempPath = Path.Combine(Path.GetTempPath(),
        ↪  $"{profile.Id}_temp.json");
        await File.WriteAllTextAsync(tempPath,
        ↪  JsonConvert.SerializeObject(profile));

        // Launch MotionInput in test mode
        var startInfo = new ProcessStartInfo
        {
            FileName = _motionInputPath,
            Arguments = $"--test-profile \"{tempPath}\"",
            UseShellExecute = false,
            CreateNoWindow = true
        };

        _testProcess = Process.Start(startInfo);
        _isInTestMode = true;

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to start test mode");
        return false;
    }
}
```

This implementation enables users to validate their configurations with real input before finalizing them, significantly improving the usability of the configuration process and reducing errors.

4.7 Performance Optimization

The application implements several performance optimization strategies to ensure responsive operation across diverse hardware configurations, addressing the non-functional requirements related to performance (NF1, NF6-NF8).

4.7.1 Asynchronous Programming Model

The implementation uses a comprehensive asynchronous programming model to maintain UI responsiveness during potentially blocking operations:

```
public async Task<IEnumerable<Profile>> GetAllProfilesAsync()
{
    return await Task.Run(() =>
    {
        return Directory.GetFiles(_profilesDirectory, "*.json")
            .AsParallel()
            .Select(async filePath =>
            % filepath: /Users/yuma/Desktop/UCL/FYP/MI_GUI_WinUI/dissertation
            ↪  /report/chapters/implementation.tex
\chapter{Design and Implementation}

\section{System Architecture Overview}
The Configuration GUI implements a modern application architecture designed
↪  for flexibility, maintainability, and performance. The system is built on
↪  a multi-layered architecture that separates concerns while facilitating
↪  communication between components through well-defined interfaces.

\subsection{Architectural Approach}
The application follows a layered architecture with clear separation between
↪  presentation, business logic, and data persistence. This separation
↪  provides several advantages:

\begin{figure}[h]
\centering
% Insert your architecture diagram here
\caption{High-level application architecture showing the relationship between
↪  UI, service, and data layers}
\label{fig:app_architecture}
\end{figure}

At the highest level, the application is organized into three primary layers:

\begin{itemize}
    \item \textbf{Presentation Layer}: Implements the user interface through
    ↪  WinUI 3 pages and controls, translating user interactions into
    ↪  service calls.
    \item \textbf{Service Layer}: Contains the business logic and coordinates
    ↪  between the UI and data models, implementing core functionality like
    ↪  profile management and AI integration.
    \item \textbf{Data Layer}: Manages data persistence and communication
    ↪  with external systems, including the core MotionInput application.
\end{itemize}

This layered approach addresses the maintainability requirements identified
↪  in Chapter 3, while providing a foundation for the accessibility and
↪  performance requirements through clear separation of concerns.

\subsection{MVVM Implementation}
The presentation layer implements the Model-View-ViewModel (MVVM)
↪  architectural pattern, which is particularly well-suited for WinUI
↪  applications. This pattern provides a clean separation between the user
↪  interface (View), the presentation logic (ViewModel), and the data
↪  (Model).

\begin{figure}[h]
```

```
\centering
% Insert MVVM diagram here
\caption{MVVM pattern implementation showing data flow between components}
\label{fig:mvvm_diagram}
\end{figure}

The ViewModels serve as an abstraction of the View, exposing properties and
↪  commands that the View binds to. This approach offers several key
↪  advantages:

\begin{itemize}
    \item \textbf{Testability}: ViewModels can be tested independently of the
    ↪  UI, enabling automated testing of business logic.
    \item \textbf{Separation of Concerns}: UI designers can focus on the View
    ↪  while developers work on the ViewModel and Model, facilitating
    ↪  parallel development.
    \item \textbf{Accessibility Support}: The separation allows for
    ↪  alternative views for different accessibility needs while maintaining
    ↪  the same underlying logic.
\end{itemize}

A typical ViewModel implementation follows this pattern:

\begin{minted}[frame=single,linenos]{csharp}
public partial class ProfileEditorViewModel : ObservableObject
{
    private readonly IProfileService _profileService;
    private readonly INavigationService _navigationService;
    private readonly ILogger<ProfileEditorViewModel> _logger;

    [ObservableProperty]
    private Profile _currentProfile;

    [ObservableProperty]
    private bool _isSaving;

    [RelayCommand]
    private async Task SaveProfileAsync()
    {
        try
        {
            IsSaving = true;
            await _profileService.SaveProfileAsync(CurrentProfile);
            _navigationService.NavigateTo(typeof(ProfileListPage));
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error saving profile");
            // Handle error
        }
        finally
        {
            IsSaving = false;
        }
    }
}
```

This implementation leverages source generators from the CommunityToolkit.Mvvm package to reduce boilerplate code while maintaining the MVVM pattern. The `[ObservableProperty]` attribute automatically generates the property change notification code, while `[RelayCommand]` generates command implementations that the View can bind to.

4.7.2 Dependency Injection System

To maintain loose coupling between components, the application implements a comprehensive dependency injection system using Microsoft's standard DI container. This approach allows components to depend on abstractions rather than concrete implementations, facilitating testing and future extensions.

The service registration occurs during application startup:

```
services.AddSingleton<IWindowManager, WindowManager>();
services.AddSingleton<INavigationService, NavigationService>();
services.AddSingleton<IProfileService, ProfileService>();
services.AddSingleton<IActionService, ActionService>();
services.AddTransient<IStableDiffusionService, StableDiffusionService>();
services.AddTransient<ProfileEditorViewModel>();
```

This registration pattern follows several design principles:

- Services with application-wide state are registered as singletons to ensure consistent state
- Resource-intensive services (like AI components) are registered as transient to control resource usage
- ViewModels are registered as transient to ensure fresh state for each navigation

The dependency injection approach directly supports the maintainability requirements identified in Chapter 3, allowing components to be replaced or modified without affecting other parts of the system.

4.8 Core Service Components

The service layer contains the primary business logic of the application, organized into focused services with clear responsibilities. Each service addresses specific requirements identified in Chapter 3.

4.8.1 Profile Management Service

The ProfileService component manages the creation, retrieval, updating, and deletion of configuration profiles. This service directly addresses the core functional requirement (FR1) by providing a programmatic interface to profile management that abstracts away the complexities of the underlying JSON format.

The service implements several key responsibilities:

- **Profile Validation**: Ensures profiles meet the required schema before saving
- **JSON Serialization/Deserialization**: Converts between object models and the JSON format required by MotionInput
- **Profile Organization**: Implements tagging and categorization of profiles
- **Change Tracking**: Monitors changes to facilitate undo/redo functionality

The implementation employs a repository pattern to abstract the storage mechanism:

```
public class ProfileService : IProfileService
{
    private readonly IProfileRepository _repository;
    private readonly IValidator<Profile> _validator;
    private readonly ILogger<ProfileService> _logger;

    public async Task<Result<Profile>> SaveProfileAsync(Profile profile)
    {
        var validationResult = _validator.Validate(profile);
        if (!validationResult.IsValid)
        {
            return Result<Profile>.Failure(validationResult.Errors);
        }

        try
        {
            var savedProfile = await _repository.SaveAsync(profile);
            return Result<Profile>.Success(savedProfile);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to save profile {ProfileName}",
            ↪  profile.Name);
            return Result<Profile>.Failure("Failed to save profile: " +
            ↪  ex.Message);
        }
    }
}
```

This implementation pattern includes several notable design decisions:

- **Result Pattern**: Returns a Result object rather than throwing exceptions, providing more structured error handling
- **Validation Separation**: Uses a separate validator component for profile validation, allowing validation rules to evolve independently
- **Repository Abstraction**: Isolates storage concerns, enabling future changes to storage mechanisms without affecting business logic

The service layer implements comprehensive error handling and logging, addressing the robustness requirements identified in Chapter 3. All operations that might fail (such as file operations or external service calls) are wrapped in appropriate exception handling with detailed logging for troubleshooting.

4.8.2 Stable Diffusion Service

The StableDiffusionService component implements the AI-powered icon generation capability (FR6) through integration with the ONNX Runtime and DirectML. This service enables users to generate contextually relevant icons from text descriptions, enhancing the visual distinctiveness of profiles and actions.

Figure 4.7: Stable Diffusion pipeline architecture showing processing stages and data flow

The service implementation addresses several technical challenges:

- **Hardware Acceleration**: Leverages DirectML for GPU acceleration when available
- **Fallback Mechanism**: Gracefully degrades to CPU execution when suitable GPU is unavailable
- **Memory Management**: Carefully manages unmanaged resources to prevent memory leaks

- **Progress Reporting**: Provides realtime generation progress for responsive UI feedback

The implementation follows a pipeline architecture, with distinct stages for text encoding, image generation, and post-processing:

```csharp
public class StableDiffusionService : IStableDiffusionService, IDisposable
{
    private InferenceSession _textEncoder;
    private InferenceSession _unet;
    private InferenceSession _vaeDecoder;
    private readonly ILogger<StableDiffusionService> _logger;

    public async Task<Result<byte[]>> GenerateImageAsync(
        string prompt,
        GenerationOptions options,
        IProgress<GenerationProgress> progress = null)
    {
        try
        {
            // 1. Encode text prompt to embedding
            var textEmbedding = EncodeText(prompt);

            // 2. Initialize random noise
            var latents = GenerateInitialNoise(options.Seed);

            // 3. Run diffusion process
            for (int i = 0; i < options.Steps; i++)
            {
                // Report progress
                progress?.Report(new GenerationProgress
                {
                    CurrentStep = i,
                    TotalSteps = options.Steps
                });

                // Perform denoising step
                latents = PerformDenoisingStep(latents, textEmbedding, i,
                ↪  options);
            }

            // 4. Decode latents to image
            var imageData = DecodeLatentsToImage(latents);

            return Result<byte[]>.Success(imageData);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Image generation failed");
            return Result<byte[]>.Failure("Image generation failed: " +
            ↪   ex.Message);
        }
    }
}
```

This implementation addresses the performance requirements (NF6-NF8) through careful optimization of the inference process. The service uses a streaming approach to generation, allowing the UI to display progress and provide feedback during the generation process.

4.9 Data Model and Storage

The data model design balances compatibility with the existing MotionInput JSON schema while extending it to support the enhanced visual management capabilities of the Configuration GUI.

4.9.1 Core Data Entities

The data model centers around several key entities that represent the configuration components:

Figure 4.8: Data model showing relationships between key entities

The Profile entity serves as the root object, containing metadata and collections of related entities:

```
public class Profile
{
    public string Id { get; set; } = Guid.NewGuid().ToString();
    public string Name { get; set; }
    public string Description { get; set; }
    public string Category { get; set; }
    public DateTime CreatedDate { get; set; } = DateTime.Now;
    public DateTime ModifiedDate { get; set; } = DateTime.Now;
    public List<string> Tags { get; set; } = new();

    // Core configuration elements
    public Dictionary<string, string> GlobalConfig { get; set; } = new();
    public List<GuiElement> GuiElements { get; set; } = new();
    public List<PoseElement> Poses { get; set; } = new();
    public Dictionary<string, SpeechCommand> SpeechCommands { get; set; } =
    ↪  new();

    // GUI-specific metadata (not in original JSON)
    public string IconPath { get; set; }
    public bool IsFavorite { get; set; }
    public Dictionary<string, object> ExtendedProperties { get; set; } =
    ↪  new();
}
```

This model design reflects several key decisions:

- **Backwards Compatibility**: Maintains the core structure required by MotionInput
- **Extended Metadata**: Adds GUI-specific properties to enhance the management experience
- **Flexible Extension**: Uses a dictionary for ExtendedProperties to allow future expansion without model changes

4.9.2 JSON Processing and Validation

The application requires sophisticated JSON handling to ensure compatibility with the MotionInput system while supporting the enhanced metadata needed for visual management. The implementation uses a combination of Newtonsoft.Json for serialization and custom JsonConverter classes to handle the complex mapping between the object model and JSON format.

A custom JsonConverter implementation handles the specific requirements of action configuration serialization:

```
public class ActionConfigConverter : JsonConverter<ActionConfig>
{
    public override ActionConfig ReadJson(JsonReader reader, Type objectType,
        ActionConfig existingValue, bool hasExistingValue, JsonSerializer
        ↪  serializer)
    {
        // Read the JSON object
        var jObject = JObject.Load(reader);

        // Extract the action type
        var actionType = jObject["action_type"]?.ToString();
        if (string.IsNullOrEmpty(actionType))
        {
            throw new JsonSerializationException("Missing action_type
            ↪  property");
        }

        // Create the appropriate action config based on type
        ActionConfig result = actionType switch
        {
            "keyboard" => new KeyboardActionConfig(),
            "mouse" => new MouseActionConfig(),
            "gamepad" => new GamepadActionConfig(),
            _ => throw new JsonSerializationException($"Unknown action type:
            ↪  {actionType}")
        };

        // Populate the common properties
        serializer.Populate(jObject.CreateReader(), result);

        return result;
    }
}
```

This approach allows the application to handle polymorphic data structures while maintaining type safety and validation. The custom converters ensure that the data model correctly reflects the underlying JSON structure while providing a strongly-typed programming model for the application.

4.10 User Interface Implementation

The user interface implementation focuses on creating an intuitive, accessible experience that addresses the usability challenges identified in the requirements analysis. The UI layer is built on WinUI 3, leveraging its modern design language and comprehensive accessibility features.

4.10.1 Page Architecture and Navigation

The application implements a page-based navigation model with distinct pages for different functional areas:

- **HomePage**: Entry point providing access to key features
- **ProfileListPage**: Displays and manages available profiles
- **ProfileEditorPage**: Creates and edits profile settings
- **ActionStudioPage**: Configures action mappings visually
- **IconStudioPage**: Generates and manages visual assets

Navigation between pages is managed by a central NavigationService that maintains navigation history and state:

```
public class NavigationService : INavigationService
{
    private readonly IWindowManager _windowManager;
    private readonly IServiceProvider _serviceProvider;
    private readonly Dictionary<Type, Type> _viewModelToViewMappings = new();

    public void NavigateTo<TViewModel>(object parameter = null) where
    ↪   TViewModel : class
    {
        var viewModelType = typeof(TViewModel);
        if (!_viewModelToViewMappings.TryGetValue(viewModelType, out var
        ↪   pageType))
        {
            throw new InvalidOperationException($"No view registered for
            ↪   {viewModelType.Name}");
        }

        var frame = _windowManager.GetMainFrame();
        var viewModel = _serviceProvider.GetRequiredService<TViewModel>();

        // Set navigation parameter if available
        if (viewModel is INavigationAware navigationAware && parameter !=
        ↪   null)
        {
            navigationAware.OnNavigatedTo(parameter);
        }

        // Navigate to the page
        frame.Navigate(pageType, viewModel);
    }
}
```

This implementation follows several design principles:

- **Separation of Concerns**: Pages handle visual presentation while ViewModels manage state and logic
- **Dependency Injection**: ViewModels are resolved from the service container, ensuring proper initialization
- **State Management**: Navigation parameters pass data between pages while maintaining separation

4.10.2 Visual Profile Editor

The ProfileEditorPage implements a visual interface for profile creation and editing, addressing the core requirement for non-technical configuration (FR1). The page is structured into functional sections that progressively disclose complexity:

Figure 4.9: Profile Editor interface showing the visual configuration approach

The XAML implementation leverages data templates and binding to create a dynamic interface that adapts to different profile types:

```xml
<Page x:Class="MotionInput.ConfigurationGUI.Views.ProfileEditorPage">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <!-- Header with profile metadata -->
        <StackPanel Grid.Row="0" Margin="20">
            <TextBox
                Text="{x:Bind ViewModel.CurrentProfile.Name, Mode=TwoWay}"
                PlaceholderText="Profile Name"
                HorizontalAlignment="Stretch"/>
            <TextBox
                Text="{x:Bind ViewModel.CurrentProfile.Description,
                ↪  Mode=TwoWay}"
                PlaceholderText="Description"
                TextWrapping="Wrap"
                Height="60"/>
        </StackPanel>

        <!-- Main configuration area with tabs -->
        <TabView Grid.Row="1">
            <TabViewItem Header="General Settings">
                <local:GeneralSettingsControl
                    Profile="{x:Bind ViewModel.CurrentProfile, Mode=TwoWay}"/>
            </TabViewItem>
            <TabViewItem Header="Input Actions">
                <local:ActionMappingControl
                    Actions="{x:Bind ViewModel.CurrentProfile.GuiElements,
                    ↪  Mode=TwoWay}"/>
            </TabViewItem>
            <TabViewItem Header="Pose Recognition">
                <local:PoseConfigurationControl
                    Poses="{x:Bind ViewModel.CurrentProfile.Poses,
                    ↪  Mode=TwoWay}"/>
            </TabViewItem>
        </TabView>
    </Grid>
</Page>
```

This implementation addresses several key design considerations:

- **Progressive Disclosure**: Organizes complexity into tabbed sections that users can navigate as needed
- **Two-Way Binding**: Enables real-time updates between the UI and data model
- **Component-Based Design**: Uses specialized controls for different configuration aspects, facilitating maintenance

4.10.3 Action Studio Implementation

The ActionStudioPage provides a visual interface for defining input-to-action mappings, addressing the requirement for visual representation of mappings (FR3). The interface uses a spatial layout that represents the relationship between physical movements and resulting actions:

Figure 4.10: Action Studio interface showing the visual mapping between inputs and actions

The implementation leverages the Canvas control for spatial positioning combined with custom visual elements for the mapping representation:

```
<Page x:Class="MotionInput.ConfigurationGUI.Views.ActionStudioPage">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="300"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <!-- Action palette sidebar -->
        <ScrollViewer Grid.Column="0">
            <ItemsRepeater ItemsSource="{x:Bind ViewModel.AvailableActions}">
                <ItemsRepeater.ItemTemplate>
                    <DataTemplate x:DataType="models:ActionDefinition">
                        <local:ActionPaletteItem
                            Definition="{x:Bind}"
                            DragStarting="OnActionDragStarting"/>
                    </DataTemplate>
                </ItemsRepeater.ItemTemplate>
            </ItemsRepeater>
        </ScrollViewer>

        <!-- Canvas for visual mapping -->
        <Grid Grid.Column="1">
            <Canvas x:Name="MappingCanvas"
                    AllowDrop="True"
                    Drop="OnCanvasDrop"
                    DragOver="OnCanvasDragOver">

                <!-- Action nodes are added dynamically -->

                <!-- Background body outline -->
                <Image Source="ms-appx:///Assets/BodyOutline.png"
                       Canvas.Left="50" Canvas.Top="50"
                       Width="300" Height="500"
                       Opacity="0.5"/>
            </Canvas>
        </Grid>
    </Grid>
</Page>
```

This implementation demonstrates several key design decisions:

- **Drag-and-Drop Interaction**: Enables intuitive placement of actions within the spatial context
- **Visual Feedback**: Provides immediate visual representation of the mapping relationship
- **Spatial Context**: Uses a body outline as reference to help users visualize physical movements

The code-behind implements the drag-and-drop functionality using WinUI's built-in drag operation system, with custom logic to translate between screen coordinates and saved position values:

```
private void OnCanvasDrop(object sender, DragEventArgs e)
{
    if (e.DataView.Contains(StandardDataFormats.Text))
    {
        var actionId = await e.DataView.GetTextAsync();
        var position = e.GetPosition(MappingCanvas);

        // Add the action at the drop position
        await ViewModel.AddActionAtPositionAsync(actionId, position.X,
        ↪  position.Y);
    }
}
```

4.11 AI Integration Implementation

The AI integration features leverage the ONNX Runtime to provide intelligent capabilities while maintaining performance on consumer hardware. The implementation focuses on two primary areas: icon generation through Stable Diffusion and pose preview through MediaPipe.

4.11.1 Stable Diffusion Implementation

The Stable Diffusion integration addresses the requirement for AI-generated visual elements (FR6). The implementation uses ONNX Runtime with DirectML to execute the diffusion model efficiently on consumer GPUs.

The implementation follows several steps:

1. **Model Loading**: Pre-trained ONNX models are loaded at initialization time
2. **Text Encoding**: User prompts are encoded into embeddings using the text encoder
3. **Diffusion Process**: The UNet model progressively denoises random latents guided by the text embedding
4. **Image Decoding**: The VAE decoder converts the final latents into an RGB image

The core diffusion process is implemented using the scheduler algorithm:

```
private Tensor<float> PerformDenoisingStep(
    Tensor<float> latents,
    Tensor<float> textEmbedding,
    int step,
    GenerationOptions options)
{
    // Calculate timestep for this step
    float timestep = _scheduler.Timesteps[step];

    // Create input dictionary for model
    var inputs = new Dictionary<string, OrtValue>
    {
        { "sample", latents.ToOrtValue() },
        { "encoder_hidden_states", textEmbedding.ToOrtValue() },
        { "timestep", timestep.ToOrtValue() }
    };

    // Run UNet inference
    using var outputs = _unet.Run(inputs);
    var noisePredictor = outputs["out_sample"].ToTensor<float>();

    // Apply scheduler step
    latents = _scheduler.Step(noisePredictor, timestep, latents,
    ↪   options.GuidanceScale);

    return latents;
}
```

This implementation required several design decisions to balance quality and performance:

- **Model Selection**: Choosing the appropriate model size for consumer hardware
- **Memory Optimization**: Carefully managing tensor operations to minimize memory usage
- **Scheduler Algorithm**: Implementing the EulerAncestralDiscrete scheduler for optimal quality/speed balance

The system includes fallback mechanisms for devices without DirectML support or with limited memory, addressing the requirement to function on modest hardware (NF6).

4.11.2 Hardware Acceleration Strategy

The application implements a sophisticated approach to hardware acceleration, with automatic detection and configuration of available hardware resources. This strategy addresses performance requirements while ensuring compatibility across diverse hardware configurations.

The implementation detects available hardware capabilities at startup:

```
private ExecutionProvider SelectOptimalExecutionProvider()
{
    try
    {
        // Check for DirectML support
        if (IsDirectMLSupported())
        {
            _logger.LogInformation("Using DirectML for hardware
            ↪  acceleration");
            return new DirectMLExecutionProvider(new
            ↪  DirectMLExecutionProviderOptions
            {
                DeviceId = 0,
                MemoryLimit = CalculateOptimalMemoryLimit()
            });
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, "DirectML initialization failed, falling back
        ↪  to CPU");
    }

    // Fall back to CPU
    _logger.LogInformation("Using CPU execution provider");
    return new CPUExecutionProvider();
}
```

This approach ensures that the application leverages available hardware acceleration when possible while gracefully degrading to CPU execution when necessary. The implementation also includes memory limit calculations based on available system resources to prevent out-of-memory conditions during model execution.

4.12 Integration with MotionInput

The Configuration GUI integrates with the core MotionInput system through a well-defined interface that maintains compatibility while enhancing the user experience. This integration addresses the requirement for seamless compatibility with the existing ecosystem.

4.12.1 Configuration File Management

The ProfileService component manages configuration files in a format compatible with the core MotionInput system. The implementation includes special handling for the JSON schema to ensure backward compatibility:

```
public async Task<bool> SaveProfileAsync(Profile profile)
{
    try
    {
        // Convert to JSON with specific formatting for MotionInput
        ↪  compatibility
        var settings = new JsonSerializerSettings
        {
            Formatting = Formatting.Indented,
            ContractResolver = new CamelCasePropertyNamesContractResolver(),
            NullValueHandling = NullValueHandling.Ignore,
            Converters = { new ActionConfigConverter(), new
            ↪  PoseConfigConverter() }
        };

        var json = JsonConvert.SerializeObject(profile, settings);

        // Ensure the profiles directory exists
        Directory.CreateDirectory(_profilesDirectory);

        // Save the file
        var filePath = Path.Combine(_profilesDirectory, $"{profile.Id}.json");
        await File.WriteAllTextAsync(filePath, json);

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to save profile {ProfileId}",
        ↪  profile.Id);
        return false;
    }
}
```

This implementation ensures that profiles created or edited through the GUI remain compatible with the core MotionInput system, addressing the integration requirement while enhancing the user experience.

4.12.2 Live Profile Testing

The application implements a live testing feature that allows users to verify configurations in real-time before applying them. This functionality addresses the requirement for real-time preview of configured actions (FR5).

The implementation uses interprocess communication to coordinate with the MotionInput application:

```
public async Task<bool> TestProfileAsync(Profile profile)
{
    try
    {
        // Save to temporary location
        var tempPath = Path.Combine(Path.GetTempPath(),
        ↪  $"{profile.Id}_temp.json");
        await File.WriteAllTextAsync(tempPath,
        ↪   JsonConvert.SerializeObject(profile));

        // Launch MotionInput in test mode
        var startInfo = new ProcessStartInfo
        {
            FileName = _motionInputPath,
            Arguments = $"--test-profile \"{tempPath}\"",
            UseShellExecute = false,
            CreateNoWindow = true
        };

        _testProcess = Process.Start(startInfo);
        _isInTestMode = true;

        return true;
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Failed to start test mode");
        return false;
    }
}
```

This implementation enables users to validate their configurations with real input before finalizing them, significantly improving the usability of the configuration process and reducing errors.

4.13 Performance Optimization

The application implements several performance optimization strategies to ensure responsive operation across diverse hardware configurations, addressing the non-functional requirements related to performance (NF1, NF6-NF8).

4.13.1 Asynchronous Programming Model

The implementation uses a comprehensive asynchronous programming model to maintain UI responsiveness during potentially blocking operations:

```
public async Task<IEnumerable<Profile>> GetAllProfilesAsync()
{
    return await Task.Run(() =>
    {
        return Directory.GetFiles(_profilesDirectory, "*.json")
            .AsParallel()
```

```
                .Select(async filePath =>
                {
                    var json = await File.ReadAllTextAsync(filePath);
                    return JsonConvert.DeserializeObject<Profile>(json);
                })
                .ToList();
    });
}
```

This implementation uses the `AsParallel` LINQ extension to parallelize file reading and deserialization, significantly improving performance when loading multiple profiles.

The use of `async/await` ensures that the UI remains responsive during these operations, addressing the performance requirements while maintaining a smooth user experience.

Chapter 5

Testing and Evaluation

5.1 Testing Strategy

5.1.1 Testing Approach

Overview of testing methodology:

- Unit testing of core components
- Integration testing of system modules
- UI automation testing
- Performance benchmarking
- Accessibility testing

5.2 Unit Testing

5.2.1 Core Components Testing

Testing of individual components:

- Model layer unit tests
- ViewModel behavior verification
- Service layer functionality
- Configuration management testing

5.2.2 AI Component Testing

Validation of AI features:

- ONNX model integration tests
- Stable Diffusion functionality
- Performance profiling
- Error handling verification

5.3 Integration Testing

5.3.1 System Integration

Testing of component interactions:

- Profile management workflow
- Configuration synchronization
- Event handling system
- Error recovery mechanisms

5.3.2 MotionInput Integration

Testing integration with main system:

- Communication protocol testing
- Profile deployment verification
- Real-time data processing
- System state synchronization

5.4 User Interface Testing

5.4.1 Functional Testing

UI functionality verification:

- Navigation testing
- Input validation
- Event handling
- State management

5.4.2 Accessibility Testing

Validation of accessibility features:

- Screen reader compatibility
- Keyboard navigation

- High contrast mode
- UI scaling tests

5.5 Performance Testing

5.5.1 Benchmark Results

Performance measurements:

- UI response times
- Memory usage patterns
- CPU utilization
- GPU performance

5.5.2 Load Testing

System behavior under load:

- Multiple profile handling
- Concurrent operations
- Resource utilization
- System stability

5.6 User Evaluation

5.6.1 User Testing Sessions

Feedback from user testing:

- Usability assessment
- Feature completeness
- User satisfaction
- Improvement suggestions

5.6.2 Feedback Analysis

Analysis of user feedback:

- Common usability issues
- Feature requests
- Performance concerns
- Overall satisfaction

5.7 Testing Results

5.7.1 Test Coverage

Overview of test coverage:

- Code coverage metrics
- Functionality coverage
- Edge case handling
- Known limitations

5.7.2 Issues and Resolutions

Documentation of issues:

- Critical bugs identified
- Resolution approaches
- Performance optimizations
- Future improvements

Chapter 6

Conclusion

6.1 Project Summary

6.1.1 Key Achievements

Major accomplishments of the project:

- Successful implementation of WinUI 3-based configuration GUI
- Integration of AI technologies (ONNX, Stable Diffusion)
- Improved accessibility through intuitive interface design
- Enhanced profile management capabilities
- Real-time preview and testing features

6.1.2 Technical Innovations

Notable technical contributions:

- Modern MVVM architecture implementation
- Efficient AI model integration
- Performance optimizations for real-time processing
- Robust error handling and recovery systems

6.2 Critical Evaluation

6.2.1 Strengths

Project strengths and successes:

- Intuitive user interface design
- Robust system architecture
- Efficient performance optimization
- Strong accessibility features

6.2.2 Limitations

Current limitations and constraints:

- Platform-specific implementation
- Resource intensive AI operations
- Limited cross-device support
- Performance constraints with multiple profiles

6.3 Future Work

6.3.1 Potential Improvements

Areas for future enhancement:

- Cross-platform compatibility
- Advanced AI model integration
- Enhanced profile sharing capabilities
- Extended customization options

6.3.2 Research Opportunities

Future research directions:

- Advanced pose estimation techniques
- Improved real-time processing
- Machine learning for user preferences
- Automated profile optimization

6.4 Personal Reflection

6.4.1 Learning Outcomes

Personal development achievements:

- Deep understanding of WinUI 3 development
- Experience with AI integration in desktop applications

- Practical MVVM implementation skills
- Project management experience

6.4.2 Challenges Overcome

Significant challenges addressed:

- Complex system integration
- Performance optimization
- AI model deployment
- User experience design

6.5 Impact Assessment

6.5.1 Accessibility Impact

Contribution to accessibility:

- Improved gaming accessibility
- Enhanced user configuration capabilities
- Reduced technical barriers
- Increased gaming inclusion

6.5.2 Project Legacy

Long-term project impact:

- Foundation for future development
- Contribution to MotionInput ecosystem
- Documentation and best practices
- Open-source contributions

Appendix A

Appendices

A.1 Development Setup

A.1.1 Environment Configuration

Development environment details:

- Visual Studio 2022 setup
- Required SDK installations
- Development tools and extensions
- Build configuration settings

A.2 Code Examples

A.2.1 MVVM Implementation

Key code implementations:

```
public class ProfileViewModel : ObservableObject
{
    private string _profileName;
    public string ProfileName
    {
        get => _profileName;
        set => SetProperty(ref _profileName, value);
    }

    public ICommand SaveProfileCommand { get; }

    public ProfileViewModel()
    {
        SaveProfileCommand = new RelayCommand(ExecuteSaveProfile);
    }

    private void ExecuteSaveProfile()
    {
        // Profile saving logic
    }
}
```

A.2.2 AI Integration Code

ONNX integration example:

```
public class PoseEstimationService
{
    private InferenceSession _session;

    public async Task InitializeModel(string modelPath)
    {
        var sessionOptions = new SessionOptions();
        sessionOptions.GraphOptimizationLevel =
        ↪  GraphOptimizationLevel.ORT_ENABLE_ALL;
        _session = await Task.Run(() => new InferenceSession(modelPath,
        ↪  sessionOptions));
    }

    public async Task<PoseData> EstimatePose(byte[] imageData)
    {
        // Model inference implementation
    }
}
```

A.3 Technical Documentation

A.3.1 API Documentation

Key interface definitions:

```
public interface IProfileService
{
    Task<Profile> CreateProfile(string name);
    Task<bool> SaveProfile(Profile profile);
    Task<Profile> LoadProfile(string profileId);
    Task<bool> DeleteProfile(string profileId);
    IAsyncEnumerable<Profile> GetAllProfiles();
}
```

A.4 User Guide

A.4.1 Installation Guide

Installation steps:

- System requirements
- Installation process
- Configuration setup
- Troubleshooting steps

A.4.2 User Manual

Usage instructions:

- Creating new profiles
- Configuring input mappings
- Testing configurations
- Managing profiles

A.5 Performance Data

A.5.1 Benchmark Results

Detailed performance metrics:

- Response time measurements
- Memory usage statistics
- CPU utilization data
- GPU performance metrics

A.6 Project Timeline

A.6.1 Development Phases

Project timeline details:

- Research phase: September - October 2023
- Design phase: October - November 2023
- Implementation: November 2023 - February 2024
- Testing: February - March 2024