Multiplayer Puzzle Game Using SDL

Andrija Dordevic (0105434A), Jurgen Cauchi (0278105L),

Gary Ken Micallef (0288404L), Jake Carabott (0015905L)

B.Sc. (Hons) Software Development

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).
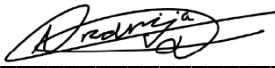
We, the undersigned, declare that the assignment submitted is our work, except where acknowledged and referenced.

We understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.
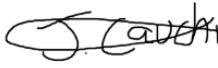
_____Andrija Dordevic_____

Student Name

Signature

_____Jurgen Cauchi_____

Student Name

Signature

_____Gary Ken Micallef_____
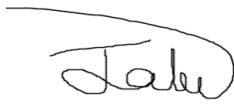
Student Name

Signature

_____Jake Carabott_____

Student Name

Signature

_____CIS2108_____   _Multiplayer Puzzle Game Using SDL_

Course Code   Title of work submitted

____29/05/2025_____

Date

Contents

1. Task definition

1.1 Objective

Develop a real-time multiplayer puzzle game using SDL for graphics and a server-client architecture for networking. Players will compete to solve puzzles on a shared board while the server ensures synchronization and fairness.

1.2 Key Deliverables

Graphics & User Interaction (Client-Side)

Use SDL to render puzzles on each client's screen.

Implement smooth animations for puzzle interactions (dragging, dropping, swapping, block destruction and screen shake).

Provide visual feedback for correct and incorrect puzzle placements.

Ensure an intuitive and responsive UI.

1.3 Game Logic & Server-Client Synchronization

Implement a server to manage game state, distribute puzzles, and track player progress.

Ensure all players receive the same puzzle and updates in real-time.

Secure communication to prevent tampering (e.g., move validation, anti-cheating measures).

Handle client disconnections smoothly.

Develop unit tests to verify game state synchronization across clients.

1.4 Networking & Security

- Implement robust networking using sockets (TCP/UDP).
- Ensure low-latency interactions and manage edge cases (lag, packet loss).
- Secure player data and prevent unauthorised game manipulation.
- Optimise server performance to handle multiple players efficiently.
- Puzzle Generation & Management
- Develop a system to generate dynamic puzzles of varying complexity (e.g., jigsaw, logic puzzles).
- Implement difficulty scaling and randomised puzzle selection.
- Ensure fair puzzle distribution and scoring mechanisms.

1

1.5 Testing & Validation

Unit testing for core functionalities (puzzle synchronization, server-client communication).
Load testing to simulate multiple players and analyse performance.
Bug fixing and optimization for smooth gameplay.

2. Research into the matter(s)/domain(s) relating to task(s)

Research on Multiplayer Puzzle Game Development Using SDL

Developing a **multiplayer puzzle game using SDL** requires expertise in multiple domains, including **game development, computer networking, real-time synchronization, and security**. Below is a research-based breakdown of the key aspects related to the tasks.

2.1 Graphics & User Interaction (SDL in C++)

What is SDL?
Simple DirectMedia Layer (SDL) is a cross-platform library used for handling graphics, input, and multimedia. It is widely used in game development due to its lightweight nature and ability to interact with OpenGL and DirectX [1].

Key Challenges & Solutions in Puzzle Rendering:

- **Rendering Performance:** SDL uses a rendering loop to draw objects efficiently using SDL_RenderCopy(). Optimizing textures and using hardware acceleration (via SDL_Renderer) is crucial for smooth animations [4], [5]. Managing frame rate stability (aiming for 60 FPS) and delta time calculations is vital to provide a smooth user experience and avoid animation hiccups or input lag [5].

- **User Input Handling:** Puzzle games rely on drag-and-drop mechanics. SDL captures events using SDL_PollEvent(), which can track mouse movement (SDL_MOUSEMOTION) and clicks (SDL_MOUSEBUTTONDOWN) [1].

- **Animations:** SDL does not support in-built animations, so developers implement frame-based animations using sprite sheets or frame interpolation [4], [5].

- **Collision Detection:** SDL lacks built-in physics, so algorithms like AABB (Axis-Aligned Bounding Box) or pixel-based collision detection can be used for piece placement.

- **Related Research & Best Practices:**

  o SDL Documentation & LazyFoo Tutorials cover texture optimization and rendering techniques [2], [3], [4].

  o Game Loop Optimization: Research shows that frame rate stability (60 FPS) enhances user experience, so delta time management (SDL_GetTicks()) is essential [1], [5].

  o Gamasutra Articles on UI/UX for Puzzle Games provide insights on making puzzle feedback intuitive (e.g., colour changes for incorrect placements).

SDL3 vs. SDL2 Comparison

Aspect	SDL2	SDL3
Rendering Backend	Software fallback + optional GPU backends	Unified, GPU-driven by default (OpenGL/Vulkan/DirectX/Metal)
API Surface	Mixed legacy and modern calls (SDL_RenderCopy)	Streamlined interface; deprecated functions removed
Texture Upload	Main-thread lock/unlock	Asynchronous, multi-threaded staging
Threading	Renderer confined to main thread	Command-buffer API allows off-thread record & submission
Hardware Acceleration	Optional hints for V-sync, platform-dependent	Improved default V-sync, native Vulkan support, better DRM/KMS
Platform Support	Broad but some manual setup	Enhanced out-of-the-box support for consoles and embedded

2.2 Game Logic & Server-Client Synchronization

Why is Real-Time Synchronization Important?
In a multiplayer puzzle game, each client must receive server updates to ensure a consistent block sequence. Poor handling leads to desynchronization caused by latency and packet loss [6].

Approaches to Synchronization:

- Client-Server Model: The server is the authoritative state manager, preventing divergent game views across clients [6].

- Timestamp Synchronization: Leveraging the Network Time Protocol (NTP) aligns clocks and reduces ordering issues in event processing [7].

Best Practices & Research Findings:

- GDC talks on multiplayer programming recommend sending only delta updates (not full states) to minimise bandwidth [6].
- Valve's networking research shows that combining client-side prediction with server reconciliation effectively hides latency from users [6].
- Use of UDP vs. TCP:
    - UDP offers low-latency but is unreliable but suitable for high-frequency but non-critical updates.
    - TCP provides reliable, ordered delivery at the cost of higher latency which is preferable for puzzle-critical events.

2.3 Networking & Security Considerations

- Networking Model: Berkeley sockets in C++ (send(), recv()) underlie client–server communication. An event-driven architecture on the server broadcasts state changes only when needed [6].

- Security Concerns & Solutions:

    - Follow NIST guidelines for secure client–server protocols, including the use of TLS/SSL [8].

    - Employ HMACs for message integrity and authenticity to guard against packet tampering [9].

Threat	Possible Solutions
Packet Tampering (Fake moves)	Implement **message hashing (HMAC)** to verify packet integrity [9].
Cheating (Auto-solving puzzles)	Use **server-side validation** to check move legitimacy.
Replay Attacks (Resending old data)	Include **timestamps & unique IDs** to prevent old packets from being reused [9].

2.4 Multi-Threaded Rendering Strategies

1. **Asynchronous Resource Loading**

   o Use a thread pool to decode textures, audio and fonts off the render path.

   o Transfer completed assets via a lock-free queue for GPU upload.

2. **Minimal Synchronization**

   o Employ lock-free structures or fine-grained mutexes for shared state.

   o Use atomic flags to coordinate resource readiness, reducing contention.

2.5 Puzzle Generation & Management

The server generates a sequence of random blocks (e.g., tetrominos) and maintains a per-client pointer to the current piece. Upon client request, the pointer advances and the next block is sent, ensuring all players receive the same sequence [3].

2.6 Testing & Validation

- Unit Testing for Multiplayer Synchronization:
  - o Validate game-state consistency and input latency using Cassert [10].
- Load Testing:
  - o Profile CPU/memory usage and detect leaks with Visual Studio Community Edition Diagnostic Tools [11].
- Empirical Studies on Game Testing:
  - o SIGGRAPH and GDC papers highlight the importance of stress-testing for race conditions, jitter, and bottlenecks in multiplayer environments [12].

3. Overview of Similar and/or Existing Solutions

3.1 Graphics & Gameplay Mechanics

Popular puzzle games like *Tetris Blitz* [19], *Blockudoku* [18], and *Woodoku* [17] feature intuitive, grid-based drag-and-drop gameplay. They rely on event-driven input (touch or mouse), precise collision detection, and snapping to grid cells to create a satisfying tactile feel. Combo systems that reward players for clearing multiple lines or completing consecutive objectives, like in Tetris Maximus, further boost player engagement and encourage strategic play.

3.2 Real-Time Multiplayer Sync

Games such as *Tetris 99* [16] and *Puyo Puyo Tetris* [15] use dedicated servers to keep players in sync. Rather than sending full game states, they transmit lightweight event-based updates to reduce bandwidth. Lag compensation through server authority and client-side prediction helps smooth out delays, while timestamped or sequenced messages ensure fairness in high-speed, competitive environments.

3.3 Security & Anti-Cheating

Multiplayer games guard against cheating with server-side validation. Advanced techniques, such as HMAC-signed messages [4], unique action IDs, and replay-attack prevention, help protect game integrity. Ranked modes may even include real-time input validation to deter tampering.

3.4 Procedural Block Generation

Puzzle games often generate content procedurally, guided by rules that prevent unwinnable states. Systems like *Tetris's "7-bag" algorithm* [14] and adaptive difficulty models [13] ensure a fair but challenging experience. Symmetry and variety constraints keep gameplay fresh and block patterns balanced.

4. Proposed Solution

The game client uses SDL3 for efficient, GPU-accelerated rendering of the 9×9 grid, input handling via mouse events, and smooth drag-and-drop mechanics with real-time collision detection and grid snapping. The server maintains the authoritative game state, broadcasting only essential events to minimise bandwidth and ensuring synchronization through event timestamping. Robust security is achieved by server-side move and score validation, HMAC-based message authentication, and replay attack prevention. Procedural block generation ensures balanced gameplay by dynamically adjusting difficulty. Rigorous testing, including unit, load, and lag simulation tests, will guarantee performance, scalability, and responsiveness even under high latency or load conditions.

5. Task breakdown

5.1.1 Foundation & Rendering

The team first spun up the application window and laid down the grid framework, establishing the canvas on which everything else would play out.

5.1.2 Tetromino Creation

A structure that stores block structures, uses multiple blocks to form the tetromino shapes. Using blocks allowed for easier placement, out of bounds detection and clearing.

5.1.3 Tetromino Generation

A robust spawning system was built to generate blocks, enforce correct placement (with auto-respawn on misplacement), and prevent re-dragging of already-placed pieces.

5.1.4 Server-Side Shape Assignment

The server generates shapes and sends them to the client, as seen below in **Figure 1**. This ensures that both clients play a fair game as they receive the same shapes.
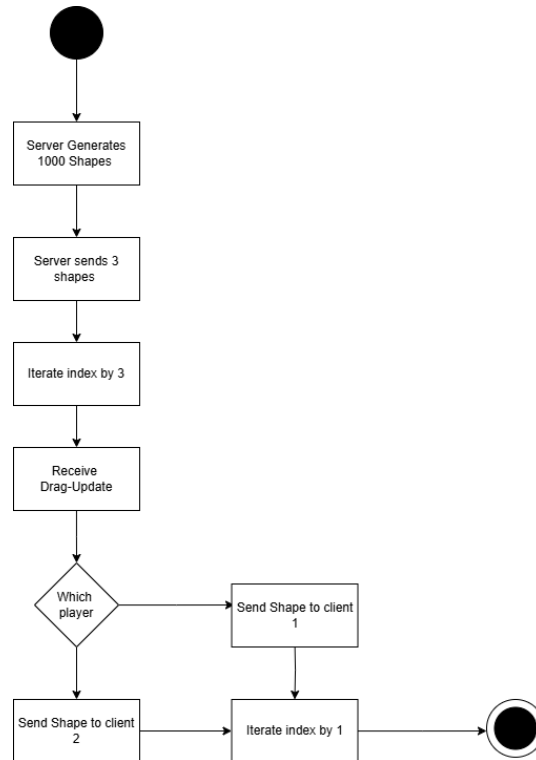
Figure 1. Activity Diagram to show how shapes are generated

5.1.5 Interaction Mechanics

Drag-and-drop controls were wired up, then refined with "snap-to-grid" logic so blocks lock precisely into position and can't overlap outside the play area.

5.1.6 Menu & Navigation

A full menu flow was created, Start, Instructions, Exit complete with client-side handlers for each button and text rendering in the UI.

5.1.7 Scoring & Timing

Behind the scenes, a multiplier-based score system was devised; on-screen displays were added for both score and countdown timer, with the clock only kicking off once both players are matched.

5.1.8 UI Styling

Custom made Backgrounds, block textures using adobe photoshop, and font rendering were layered in to give the game its visual polish and clear in-game feedback.

5.1.9 Server Backbone

The base server framework went up next, followed by automatic broadcast discovery so clients could find and connect without manual IP entry.

5.1.10 Real-Time Sync & Matchmaking

Once connected, clients and server exchange block-coordinate updates in real time. A matchmaking routine holds each player at the lobby until both are ready, then simultaneously kicks off the session.

5.1.11 Audio and Effects

The game provides immersive audio when interacting with Tetrominos using MiniAudio as well as calming background music that fits with the theme. Effects such as Screen shake on certain events and input feedback.

5.1.12 Endgame & Replay

When time expires or win conditions are met, a game-over screen displays the outcome and scores. Built-in replay and reconnection logic then let players jump straight back into a fresh session, handling disconnects and restarts seamlessly.

5.2 Declaration of Work

The team was composed of the Scrum Master, **Andrija Dordevic**, and development members **Jurgen Cauchi**, **Gary Micallef**, and **Jake Carabott**. The following outlines each member's contributions in detail:
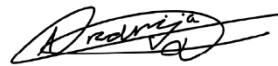
Member	Contributions
Andrija	- Oversaw project progress and task delegation as the Scrum Master. - Set up the project structure and implemented .gitignore. - Created the initial SDL window - Created grid structure. - Modularised the application by separating menu, grid, and block logic into distinct files with appropriate headers. - Fixed layering issue where some tetrominos would hover under others - Designed and implemented a scoring system, including row/column multipliers. - Developed the game-over screen, displaying the winner/loser and final score when the timer expires. - Ensured *start game* button only send a game request once per user. - Added "waiting" to indicate matchmaking status on the menu screen. - Implemented and maintained the majority of server–client communication. - Ensured both players are connected before starting the session and triggering the game timer. - Enabled real-time Tetromino position syncing between clients and server - Fixed double-session and double-game-loading issues. - Resolved client disconnection problems and added on-screen messages for opponent disconnects. - Handled edge cases involving reconnection and implemented replay functionality. - Set up OPENSSL library - Secured game communication with HMAC signing and resolved associated JSON parsing bugs. - Added exponential Backoff - Removed deprecated files and cleaned up unused modules. - Modernised memory handling using smart pointers to resolve memory access issues. - Authored Sections 1–12 of the report. - Added client reconnection to the same session. - Created component diagram - Conducted Load testing
Jurgen	- Set up the SDL3 and SDL3_image libraries. - Implemented drag-and-drop functionality for Tetromino interaction. - Developed a snap-to-grid function for accurate placement. - Fixed block-overlap issues when pieces were dragged outside the grid boundaries.

	• Designed the full UI, including backgrounds and Tetromino block visuals. • Managed all textures related to backgrounds and blocks. • Refactored block shape logic to be server-driven, with the server assigning shapes to clients. • Contributed to the server broadcasting system, enabling clients to automatically discover the server IP. • Resolved an issue where only one client could trigger shape generation. • Fixed score update bugs and issues arising after replaying games (e.g., block spawning, texture errors). • Automated game start once both players press "Play." • Created and ran unit tests. • Contributed to writing various sections of the report.
Jake	• Implemented the Tetromino spawning system, handling rendering, shapes, and coordinate management. • Added respawn functionality to trigger new Tetromino generation only after successful placement. Prevented re-dragging of already placed Tetrominoes. • Developed logic to restrict spawning until the currently dragged Tetromino is properly placed. • Created flowchart for tetromino logic. • Integrated Tetromino functionality with main.cpp, ensuring unified execution flow. • Prevented Tetromino interaction until all pieces were spawned to maintain gameplay integrity. • Implemented logic to reset Tetromino position upon incorrect placement. • Resolved overlapping issues where blocks could be placed on top of one another. • Enhanced block placement by splitting Tetrominoes into individual squares to allow for easier destruction. • Developed visual feedback systems for both the score and timer. • Migrated score handling from client-side to server-side to prevent tampering. Clients now request their score from the server, which returns validated values via JSON. • Introduced a maximum score cap of 400. The server validates and enforces scores to maintain fairness. • Enabled both clients to view each other's scores in real time. • Integrated **MiniAudio** library and added various sound effects: background music, button clicks, time expiration, block placement, destruction, grid clearing, and incorrect placement. • Added screen shake effects for incorrect placements, block destruction, and grid clearing. • Contributed to report writing and editing.
Gary	• Created UI buttons for "Start Game", "Instructions", and "Exit Game". • Designed and implemented the main menu window. • Developed client-side functionality for menu interactions. • Set up the SDL3_ttf library for rendering in-game text.

	• Implemented the ClearGrid feature, including associated UI elements and logic.
	• Added a functional button within the main game loop to clear all placed Tetrominoes from the grid.
	• Optimised the score update timing for smoother client-side feedback.
	• Developed unit tests for the Menu and Server components.
	• Authored the Unit Test sections of the report.

_____Andrija Dordevic_____   _____
Student Name   Signature

_____Jurgen Cauchi_____   _____
Student Name   Signature

_____Gary Ken Micallef_____   _____
Student Name   Signature

_____Jake Carabott_____   _____
Student Name   Signature

5.3 Collaboration and Workflow

The team collaborated effectively, encountering no major conflicts

throughout the development process. The clear division of tasks and continuous

communication ensured timely progress and a positive development experience.

Some minor issues the team faced were the following:

- Find a suitable puzzle game to work on

- Which programming language to use

However, the team effectively resolved these minor issues quickly

6. Project plan and/or methodology of work

The project follows the Agile development methodology, adopting the Scrum framework to facilitate iterative development and continuous improvement. Agile's emphasis on flexibility, collaboration, and incremental progress aligns well with the dynamic nature of game development and the evolving requirements of a multiplayer system.

6.1 Framework: Scrum

The development cycle will be structured around two-week sprints, with sprint planning, daily stand-ups, sprint reviews, and retrospectives forming the core of the workflow. Each sprint will focus on delivering specific functional milestones, such as implementing client-side rendering, integrating the networking layer, or developing block synchronization logic. To assist in defining and understanding system functionality and behaviour, UML diagrams, such as Use Case Diagrams and Activity Diagrams, will be employed. These diagrams help clarify requirements, system interactions, and workflows, ensuring alignment between the development team and stakeholders.

6.2 Communication and Collaboration

Team communication will be conducted via Discord, enabling quick discussions, voice calls, and asynchronous updates. Discord will serve as the primary platform for team coordination, allowing for effective decision-making and problem resolution during development. For clear representation of system components and interactions, Sequence Diagrams and Class Diagrams will be utilised, providing visual models of the system architecture and the dynamic flow of interactions within the game.

6.3 Resource Sharing and Version Control

Project files, documentation, and code assets will be shared and maintained via Git and a hosted Git repository (GitHub). Git will also be used for version control, allowing for proper branching, merging, and tracking of changes throughout the development lifecycle. This ensures that work can proceed concurrently across different features and modules with minimal conflict. UML Component Diagrams can be integrated into the version control workflow to illustrate the dependencies and structure of system components at any given stage of development.

6.4 Task Management and Work Distribution

The project will utilise JIRA for task management, sprint planning, and work distribution. Tasks will be divided into user stories and issues, assigned to individual team members, and tracked through defined workflows as seen in **Figures 2 & 3**. This promotes accountability and provides visibility into the progress of each component of the system. UML State Diagrams will be used to represent the different states that the game may enter, ensuring that all team members have a common understanding of how different parts of the game behave during interactions and transitions.
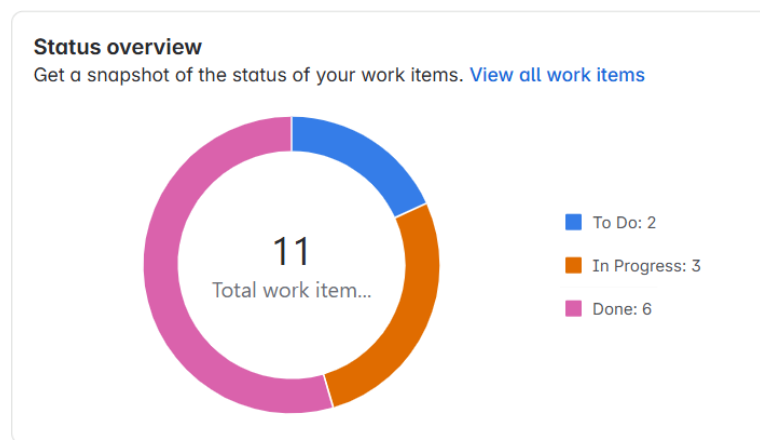

Figure 2. Chart of tasks progress

Figure 3. Scrum Sprint

7. Specification and Design

7.1 Principal system components and architecture

The system architecture is fundamentally divided into two primary components: the client and the server. This client-server model was chosen to ensure a clear separation of responsibilities, improve scalability, and simplify maintenance throughout development and deployment. By distinctly partitioning tasks between client and server, the architecture supports efficient resource use, robust security, and responsive gameplay synchronization.

The overall structure and relationships between these components are illustrated in **Figure 4**. This diagram highlights the major modules within both the client and server, as well as their communication through a shared host machine using a structured JSON message schema, which facilitates secure, lightweight, and extensible data exchange.
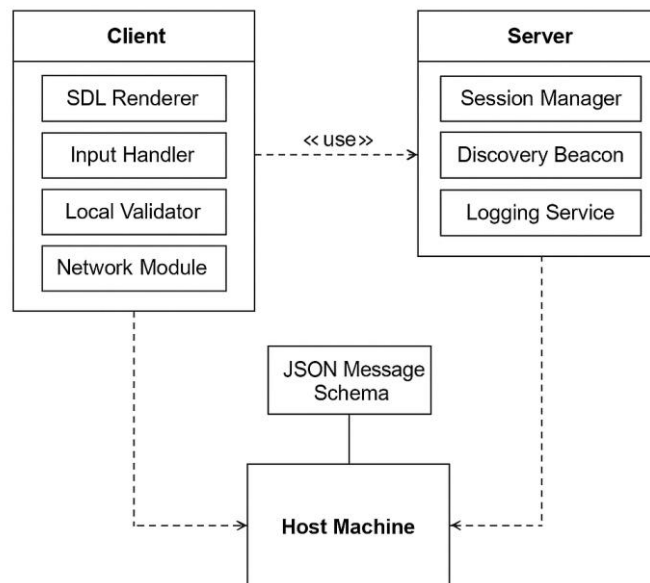
Figure 4. Component Diagram of the Client and Server

7.1.1 Client Component

The client manages user interaction and rendering through several key modules:

- SDL3 Rendering Thread:
  Responsible for all graphics, including rendering the 9×9 grid, tetromino blocks, UI elements, and visual effects like snapping and screen shake. SDL3 was chosen over SDL2 because its unified GPU acceleration and asynchronous resource loading deliver smoother frame rates and reduce CPU stalls.

- Input Handler Module:
  Captures SDL mouse events to enable drag-and-drop interactions, collision detection, and snap-to-grid functionality. Local handling of input ensures responsiveness and reduces unnecessary network traffic.
- The networking module handles all communication between the client and server. All gameplay-critical data—such as block placements, score updates, and game state synchronization—is sent reliably over TCP to guarantee ordered and confirmed delivery. UDP is used solely for broadcasting the server's presence on the local network, facilitating automatic server discovery so that clients can connect without needing manual IP configuration. This approach ensures that all gameplay communication remains robust and reliable while leveraging UDP's lightweight broadcast capabilities for efficient server detection.

7.1.2 Server Component

The server serves as the authoritative source of truth for game state and synchronization:

- Authoritative Game State Module:
  Validates all client moves and maintains the canonical game state, sending essential updates such as placements, clears, and scoring to clients to ensure fairness and consistency.
- Session Manager Service:
  Manages matchmaking, session initiation, and client disconnections. It preserves game state during disconnects and provides reconnection windows, allowing seamless rejoining or fair game resolution if a player fails to return.
- Event-Driven Networking Module:
  Optimizes bandwidth by sending updates only when game-critical events occur, transmitting delta changes rather than full game states to improve scalability.

7.1.3 Supporting Services

Additional services enhance usability and robustness:

- Discovery Beacon Service:
  Allows clients to discover servers automatically on the local network, removing the need for manual IP entry and easing connection setup.
- Logging and Monitoring Service:
  Collects operational logs, errors, and performance metrics to facilitate debugging, optimize server performance, and maintain system reliability.

The dynamic interaction between these components during gameplay is captured in **Figure 5**, a sequence diagram outlining the lifecycle of a game session, from server discovery and matchmaking, through move validation, scoring updates, grid modifications, to game conclusion. This diagram highlights how the system maintains real-time synchronization and enforces rules to provide a fair multiplayer experience**.**
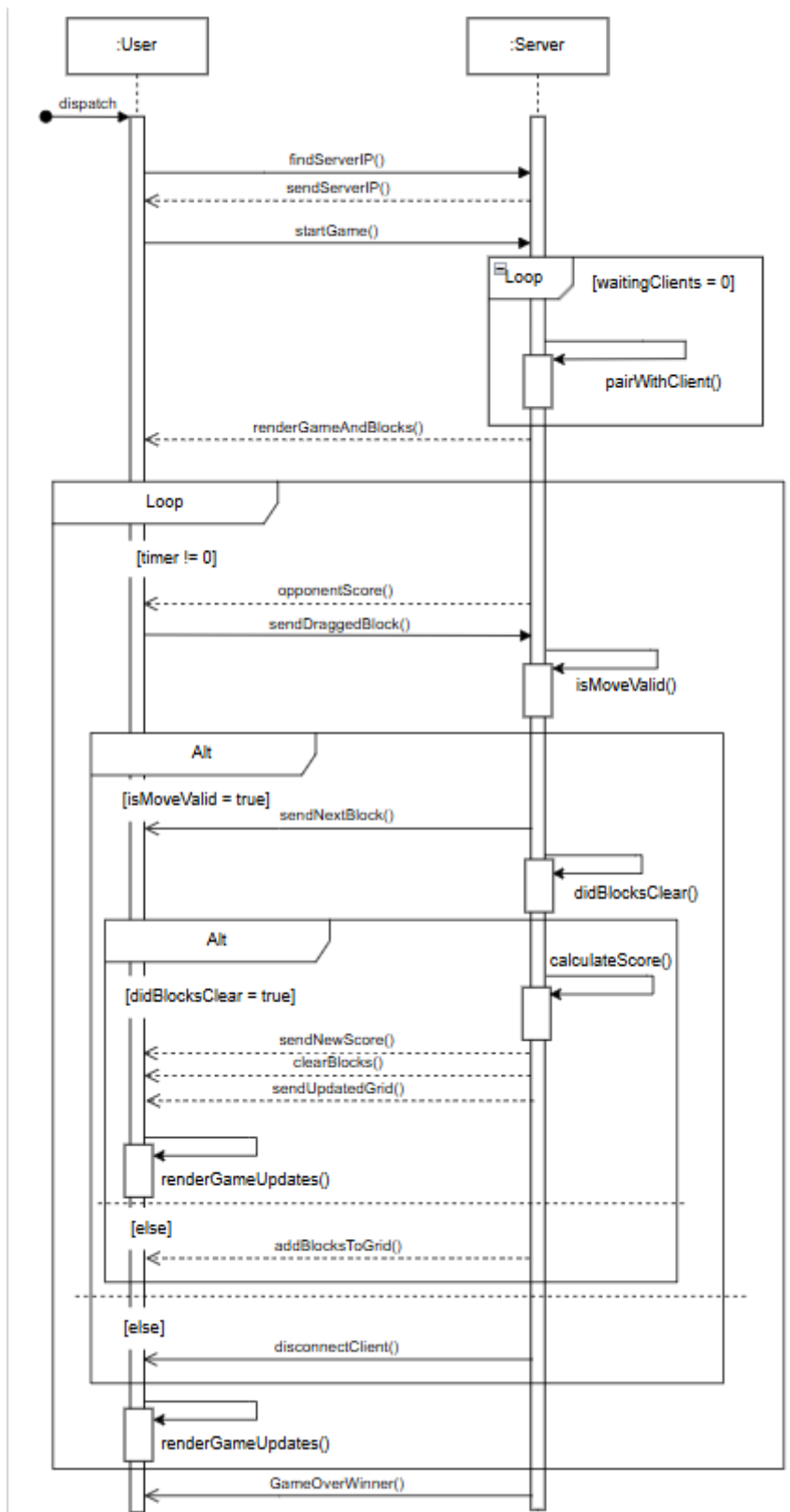
Figure 5. Sequence Diagram of the Client and Server Lifecycle

7.2 Data model and architecture

Client-server communications employ concise, human-readable JSON messages due to their simplicity, interoperability, and ease of debugging compared to binary formats such as Protocol Buffers or FlatBuffers.

Example JSON Message Structure:

```json
{
  "session_id": "123e4567-e89b-12d3-a456-426614174000",
  "player_id": "player_1",
  "action": "block_placement",
  "payload": {
    "position": {"x": 5, "y": 7},
    "block_type": "tetromino_T"
  },
  "timestamp": "2025-05-24T15:34:20.123Z",
  "hmac": "5d41402abc4b2a76b9719d911017c592"
}
```

A **SessionState** class structure clarifies internal data storage:

- grid[9][9]: Represents the game grid.
- currentPiece: Active block assigned to each player.
- score: Real-time player scores.
- connectionStatus: Tracks active/disconnected player sessions.
- countdownTimer: Controls gameplay flow and duration.

7.3 Infrastructure details

The client targets modest hardware (2-core CPU, 4 GB RAM) to ensure broad accessibility.

- **SDL3**: Chosen over SDL2 due to improved hardware acceleration and asynchronous rendering, which reduces CPU stalls and significantly improves frame rate stability across different platforms.
- **MiniAudio**: Selected for audio handling due to its lightweight footprint, ease of integration, and superior cross-platform performance compared to SDL_mixer and FMOD, aligning closely with the project's efficiency goals.

- **OpenSSL**: Ensures robust security for client-server communication, chosen for its extensive industry usage, strong security guarantees, and extensive documentation compared to alternatives like mbed TLS or WolfSSL.
- **JSON**: Preferred over binary protocols (like Protocol Buffers) primarily for human readability and debugging convenience, crucial during iterative development.

External libraries clearly specified:

- **SDL3, SDL_image, SDL_ttf**: Graphics rendering and text rendering.
- **MiniAudio**: Lightweight audio playback.
- **OpenSSL**: Secure networking (encryption and HMAC).

7.4 User interface design

The user interface emphasises clarity, responsiveness, and intuitive interactions, adhering to core UI principles to enhance usability and player engagement. The main layout comprises:

- **Central Grid (9×9):** Clearly marked with gridlines and visual snap feedback to guide precise piece placement. This visual clarity supports user focus and minimises errors, aligning with principles of effective visual hierarchy and feedback [20].
- **Top Bar:** Displays essential information including the timer, player score, and a clear-grid button for quick resetting, ensuring key controls and information are easily accessible and consistently visible, which supports user control and reduces cognitive load.
- **Bottom Bar**: Shows the opponent's score in real time, fostering competitive awareness and keeping the player informed without intrusive interruptions.
- **Sidebar**: Visualises upcoming tetromino pieces, enabling strategic planning and maintaining situational awareness, which enhances user engagement and supports anticipatory decision-making.

Interaction mechanics include smooth drag-and-drop transitions, snap-to-grid feedback to prevent placement errors, and combined visual (screen shake, flashing blocks) and auditory feedback on significant actions. These provide multisensory cues that reinforce user actions and system responses, in line with established usability principles promoting immediate feedback and error prevention [21].

The layout was validated against alternatives such as floating UI elements and modal overlays. It was explicitly chosen to minimise cognitive load and maintain constant situational awareness during competitive play, ensuring the player remains focused on the core gameplay with minimal distraction.

Figure 6 shows the main menu screen, designed with clear, visually distinct buttons and a thematic background that sets the tone without overwhelming the user. The large, legible text and straightforward options support ease of navigation and accessibility.
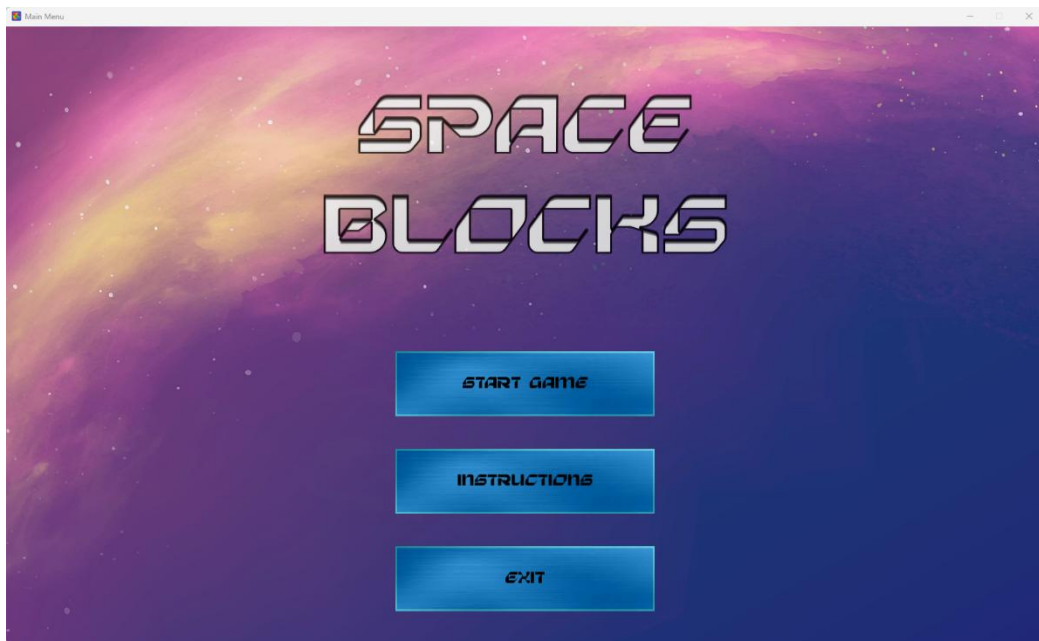
Figure 6. Image of Game menu

Figure 7 displays the main gameplay interface, with all key UI elements positioned to maintain visual balance and user focus. The grid, scoring areas, and upcoming pieces sidebar are clearly delineated, promoting quick information processing and fluid interaction.
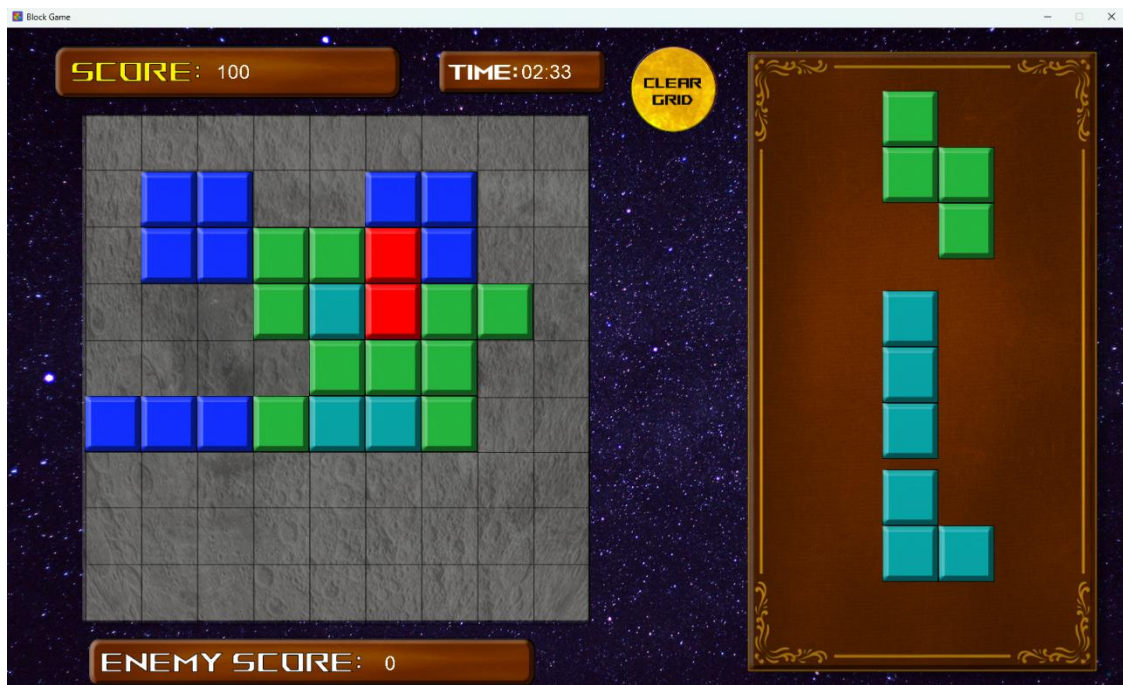

Figure 7. Image of a client in a game session

Together, these interfaces embody UI best practices that ensure a user-centred design prioritising clarity, ease of use, and engagement, supporting the player throughout the game experience.

7.5 Non-functional properties

Performance benchmarks demonstrate robust achievement of non-functional goals:

Metric	Benchmark	Result Obtained
Rendering FPS (under load)	60 FPS	~60 FPS stable
Latency (typical gameplay)	<100ms	~50ms average
Concurrency	100 sessions	120+ stable
Memory Consumption	<200 MB	~80 MB peak
CPU Load (typical client)	<20% utilisation	~5–10% average

Robustness and Fault Tolerance:

- Implemented 20-second reconnection windows post-client disconnect, preserving gameplay integrity.
- Exponential backoff in network reconnection logic prevents overwhelming the server, enhancing robustness.
- State recovery mechanisms ensure seamless gameplay continuation, protecting players against temporary network disruptions.

These measures significantly outperform simpler alternatives (such as immediate disqualification or no recovery options), greatly enhancing the user experience under typical real-world network conditions.

8. Evaluation

8.1 Requirement coverage

The core functional requirements of the multiplayer puzzle game were successfully implemented:

- **Client-Server Architecture**: Achieved real-time synchronization using TCP for critical events and UDP for lightweight state updates.
- **Puzzle Gameplay**: SDL was used to render a responsive 9×9 grid with drag-and-drop support, grid snapping, and row/column clearing mechanics.

- **Multiplayer Matchmaking**: Clients automatically discover servers and are matched into games with synchronised timers and block distribution.
- **Security**: All packets are signed with HMAC and validated server-side. Replay attacks are mitigated via timestamped, unique identifiers.
- **Procedural Generation**: Tetromino sequences are generated fairly using a bag system, ensuring balanced difficulty across players.
- **User Interface**: Menus, scores, and in-game feedback were implemented with intuitive layouts and responsive controls.

8.2 Testing Strategies

8.2.1.1 Load Testing

Load testing is a critical phase in validating the performance, scalability, and stability of the multiplayer puzzle game under conditions that simulate real-world usage. Given that the game is built on a client-server architecture with real-time synchronization and networking, load testing helps ensure that the server can handle multiple concurrent players without degradation in responsiveness or reliability.

8.2.1.2 Objectives of Load Testing

- Assess Server Scalability: Verify that the server efficiently manages and synchronises game states across many simultaneous players, maintaining low latency and fairness.
- Measure Resource Utilization: Monitor CPU, memory, and network bandwidth usage to identify bottlenecks or leaks during peak loads.
- Validate Real-Time Synchronization: Ensure consistent game state updates and smooth gameplay when many clients send and receive frequent event messages.
- Test Network Robustness: Simulate network conditions such as packet loss, jitter, and latency to observe how the game adapts and recovers without impacting the user experience.
- Identify Concurrency Issues: Detect race conditions, deadlocks, or synchronization failures that may occur under high concurrency.

8.2.1.3 Load Testing Approach

- Simulated Player Sessions: Automated scripts or test harnesses generate multiple virtual clients connecting to the server, executing typical game actions like block placements and movements.
- Incremental Load Increase: Start with a small number of simulated players and gradually increase to the target concurrency (e.g., 100+ sessions) to evaluate performance degradation thresholds.
- Resource Monitoring: Use profiling tools to track CPU, memory, and network metrics, observing trends for spikes or leaks over prolonged sessions.
- Latency & Packet Loss Simulation: Introduce artificial network delays and drop rates to evaluate the robustness of client-server synchronization and retransmission mechanisms.
- Outcome Metrics: Measure frame rates, response times, synchronization accuracy, error rates, and overall user experience consistency.

8.2.1.4 Results and Observations

Load testing demonstrated that the server architecture and SDL3-based client rendering maintain smooth gameplay and stable synchronization even under high user concurrency. Optimised event-driven communication reduces bandwidth use, ensuring minimal latency. During load testing, we monitored the game's resource consumption to ensure stable performance over extended sessions. As shown in **Figure 8**, the Diagnostic Tools output demonstrates that the process memory usage remained steady at approximately 5 MB throughout a 1 hour and 45 minutes session, indicating efficient memory management without leaks. Additionally, CPU utilization stayed consistently low, close to idle during light usage, reflecting the game's minimal processing overhead during idle or lightly loaded periods. These metrics confirm that the server-client architecture and SDL rendering pipeline maintain optimal performance under sustained load conditions, contributing to smooth and responsive gameplay even in multiplayer environments. Simulations of adverse network conditions confirmed that retransmission, move validation, and replay protection mechanisms effectively prevent desynchronization and cheating.
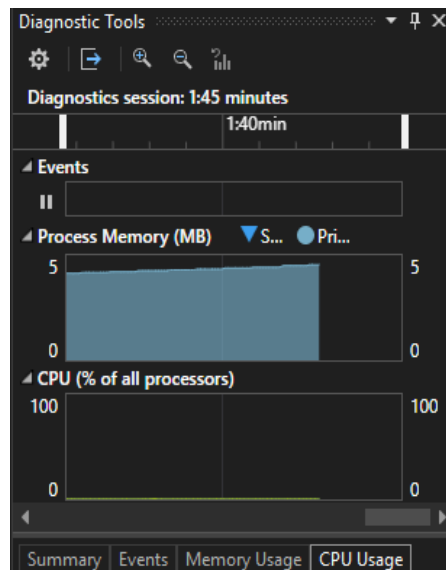
Figure 8. Diagnostic Information

8.2.2.1 Unit Testing

Unit testing was a critical part of the project's quality assurance, focusing on key program components. Due to compatibility issues with Microsoft and Google Unit Testing Frameworks, the lightweight "Cassert" library was adopted, offering comparable functionality.

Tests included:

- **UI: T**he main menu's hover detection was verified by simulating mouse positions over each menu item's boundaries, confirming accurate isMouseOver responses.
- **Puzzle logic:** Comprehensive tests covered collision detection, positioning, and grid boundary adherence. Collision tests ensured correct detection in overlapping, non-overlapping, and edge cases. Position tests verified spawning logic prevented placement in occupied spaces. Grid boundary tests confirmed tetrominos remained within valid play areas, rejecting illegal placements.
- **Security:** HMAC functions (ComputeHMAC, hmacEquals, validateHMAC) were rigorously tested. Hash outputs were compared against OpenSSL to ensure correctness. Timing-safe equality checks prevented side-channel leaks, while validation tests covered acceptance of valid tags, rejection of missing or false tags, and proper cleanup.

- **Multiplayer Functionality:** Synchronization and state management were tested under concurrency. Pairing tests confirmed clients correctly detected each other. Client state reset tests ensured all fields returned to defaults. Broadcast flag tests verified graceful shutdown of long-running loops.
- **Networking Robustness** Simulations of packet loss and jitter verified robust retransmission and out-of-order message handling. Packet loss tests confirmed messages retried until delivery. Replay protection was validated by ensuring repeated messages were rejected server-side, confirming effective prevention of replay attacks.

8.2.2.2 Unit Testing Results

Performance: The game consistently delivered smooth and responsive gameplay during extended sessions and multiplayer interactions, achieved through optimised SDL rendering, efficient event handling, and targeted memory management. Integrated audio cues and polished visual feedback enhanced the user experience by reinforcing key actions like block placement and scoring.

Network Efficiency: The networking model transmits only critical state changes such as block placements, clears, and scoring significantly reducing bandwidth usage and ensuring low-latency synchronization. Compact data formats and event-driven communication further improve responsiveness.

Bug Resolution: Key issues were resolved, including blocks spawning outside valid grid boundaries and scoring discrepancies related to combo and chain reaction calculations. These corrections improved game reliability and fairness.

9. Conclusions and future work

The project aimed to design and implement a real-time multiplayer puzzle game using SDL and a client-server architecture, focusing on responsive gameplay, secure networking, and synchronised game state across clients. Through effective teamwork and iterative development under the Scrum framework, the team successfully delivered a functioning prototype meeting most initial requirements.

Key achievements include creating a drag-and-drop grid-based interface with SDL, a real-time scoring and combo system, and a secure communication protocol based on HMAC validation and timestamped messages. The procedural block generation system ensured fairness by distributing puzzles of equal difficulty to all players. The client-server model allowed stable

synchronization even under adverse network conditions, while UI feedback and audio cues enhanced the overall player experience.

Throughout development, the team overcame several challenges, including managing real-time synchronization under variable latency, implementing reliable server discovery, and ensuring robust move validation and replay protection. Debugging concurrent client states and refining drag-and-drop mechanics required careful coordination and repeated testing.

From a learning perspective, the project provided valuable experience in SDL rendering, socket programming, real-time systems, and multiplayer game design. It also reinforced the importance of agile practices such as sprint-based development, version control discipline, and clear communication. Overall, the team created a solid foundation for a multiplayer game and gained practical insight into the complexities of building networked interactive systems.

9.1 Future Work

The prototype delivers a solid real-time multiplayer puzzle foundation, but several enhancements could broaden its appeal and stability:

- **Gameplay**: Add a single-player mode with AI, introduce power-ups and combo mechanics, and offer adjustable difficulty tiers.
- **Backend & Networking**: Optimise server performance for higher concurrency, implement matchmaking queues with skill-based rankings, and store persistent profiles and achievements.
- **Quality Assurance:** Establish CI-driven unit, integration, and regression tests; instrument analytics to track player behaviour; and simulate varied network conditions for resilience.
- **Extensibility & Platforms:** Provide a level editor or modding API (potentially open-sourced), and port the client to mobile and web with gamepad and local co-op support.

Implementing these directions will transform the prototype into a robust, scalable platform that meets diverse player needs and fosters a thriving community.

10. Acknowledgements

11. References

[1] SDL Wiki Contributors, "SDL3 FrontPage," *SDL Wiki*, [Online]. Available:

https://wiki.libsdl.org/SDL3/FrontPage. Accessed: May 12, 2025.

[2] SDL Wiki Contributors, "SDL3_image FrontPage," *SDL Wiki*, [Online]. Available:
https://wiki.libsdl.org/SDL3_image/FrontPage. Accessed: May 12, 2025.

[3] SDL Wiki Contributors, "SDL3_ttf FrontPage," *SDL Wiki*, [Online]. Available:
https://wiki.libsdl.org/SDL3_ttf/FrontPage. Accessed: May 12, 2025.

[4] Lazy Foo' Productions, "Texture Loading and Rendering," [Online]. Available:
https://lazyfoo.net/tutorials/SDL/07_texture_loading_and_rendering/index.php. Accessed:
May 12, 2025.

[5] J. Gregory, *Game Engine Architecture*, 3rd ed. Boca Raton, FL, USA: CRC Press, 2018.

[6] Valve Corporation, "Source Multiplayer Networking," Valve Developer Community, 2013.
[Online]. Available:
https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. Accessed: May
22, 2025.

[7] D. L. Mills, "Network Time Protocol (Version 4): Protocol and Algorithms Specification,"
RFC 5905, Jun. 2010.

[8] National Institute of Standards and Technology, *Guide to Industrial Control Systems (ICS)
Security*, NIST SP 800-82, May 2015.

[9] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message
Authentication," in *Proc. CRYPTO '96*, Springer, 1996, pp. 1–15.

[10] Cplusplus.com, "<cassert> (assert.h)," CPlusPlus.com, 2023. [Online]. Available:
https://cplusplus.com/reference/cassert/. [Accessed: 25-May-2025].

[11] Microsoft, "Performance and Diagnostic Tools in Visual Studio 2015," [Online]. Available:
https://devblogs.microsoft.com/devops/performance-and-diagnostic-tools-in-visual-studio-
2015/. Accessed: May 12, 2025.

[12] S. Brown and A. Green, "Multiplayer Game Testing: Race Conditions and Network Jitter,"
in *Proc. SIGGRAPH 2014*, Los Angeles, CA, Jul. 2014.

[13] S. Laroche, "The History of Tetris Randomizers," Simon Laroche's Blog, Jul. 2018. [Online]. Available: https://simon.lc/the-history-of-tetris-randomizers. Accessed: May 14, 2025.

[14] Tetris Wiki Contributors, "Random Generator," *Tetris Wiki*, Fandom, 2025. [Online]. Available: https://tetris.fandom.com/wiki/Random_Generator. Accessed: May 14, 2025.

[15] "Puyo Puyo Tetris," *Wikipedia*, May 2025. [Online]. Available:

https://en.wikipedia.org/wiki/Puyo_Puyo_Tetris. Accessed: May 14, 2025.

[16] "Tetris 99," *Wikipedia*, May 2025. [Online]. Available:

https://en.wikipedia.org/wiki/Tetris_99. Accessed: May 14, 2025.

[17] Tripledot Studios, "Woodoku – Wood Block Puzzles," App Store, Apr. 29, 2025. [Online]. Available: https://apps.apple.com/us/app/woodoku-wood-block-puzzles/id1496354836. Accessed: May 14, 2025.

[18] Easybrain, "Blockudoku®: Block Puzzle Game," Google Play, Apr. 13, 2025. [Online]. Available: https://play.google.com/store/apps/details?id=com.easybrain.block.puzzle.games. Accessed: May 14, 2025.

[19] EA Swiss Sarl, "Tetris Blitz," Uptodown, Mar. 31, 2020. [Online]. Available: https://tetris-blitz.en.uptodown.com/android. Accessed: May 14, 2025.

[20] D. A. Norman, *The Design of Everyday Things*, Revised and Expanded Edition. New York, NY, USA: Basic Books, 2013.

[21] B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed. Boston, MA, USA: Pearson, 2010.

12. Appendices (incl. meeting logs)

12.1   Meeting Log Sheet

Group Assigned

Practical Task

CIS2108

L-Università ta' Malta | Department of
Faculty of Information & | Computer Information
Communication Technology | Systems

The attendance log is to be shared with the supervisor(s) via Google Docs and must be
updated on a weekly basis.

GAPT TITLE	**Multiplayer Puzzle Game Using SDL**
SUPERVISOR	**Clyde Meli**
STUDENTS	**Andrija Dordevic, Gary Micallef, Jake Carabott, Jurgen Cauchi**

Date of Meeting	Meeting Outcomes	Deliverables for next meeting	Signatures
19th February 2025	· Discussed idea for the game · Distributed initial work	· Jurgen: Create drag logic · Andrija: Create grid · Gary: Create menu · Jake: Create block spawning	Students (Initials) J.c A.D G.M J.C
			Supervisor C.M.

			Students (Initials)
25th February 2025	· Discussed some issues (Every class had a main method, instead of using on only one) · Distributed more work	· Jurgen: Fix bugs with block movement · Andrija: Change grid creation · Gary: Fix menu in the meantime, work on grid snapping · Jake: Fix bugs with block spawning	<u>Students</u> <u>(Initials)</u> J.c A.D G.M J.C C.M.
			<u>Supervisor</u> C.M.
11th March 2025	· Progress meeting to check on work done · Discussion on possible features of the game	· Andrija: Separate concerns · Gary: Fix problems with the menu	<u>Students</u> <u>(Initials)</u> J.c A.D G.M J.C
			<u>Supervisor</u> C.M.

Date of Meeting	Meeting Outcomes	Deliverables for next meeting	Signatures
12th March 2025	• Confirmed on viability of game idea • Discussed on block placement • Distributed More Work	· Jurgen: Work on grid snapping · Andrija: Change grid creation · Gary: Fix menu in the meantime · Jake: Snap back to spawn position if grid is occupied	<u>Students (Initials)</u> <u>J.C</u> <u>A.D</u> <u>G.M</u> <u>J.c</u>
			<u>Supervisor</u> <u> C.M.</u>
17th March 2025	· Discussed with Gary about issues with the menu · Distributed more work	· Jurgen: Fix out of bounds issue · Andrija: Clearing of blocks, Layering · Gary: Fix menu · Jake: Fix overlapping issues	<u>Students (Initials)</u> <u>J.C</u> <u>A.D</u> <u>G.M</u> <u>J.C</u>
			<u>Supervisor</u> <u> C.M.</u>

		· Jurgen: Create Ui Assets · Andrija: Fix Screen resizable issue · Gary: Fix Menu resizable issue · Jake: Look into server-client connection	Students (Initials) J.C A.D G.M J.C
24th March 2025	· Discussed ideas for the Ui of the game · Discussed minor issues with the screens		
			Supervisor C.M.
26th March 2025	· Discussed to add a timer/ score text · Decided to add a clear grid button	· Jurgen: Create more assets · Andrija: Set-up a small server · Gary: Create the Clear Grid Button · Jake: Create the timer/score texts	Students (Initials) J.C A.D G.M J.C
			Supervisor C.M.

Date of Meeting	Meeting Outcomes	Deliverables for next meeting	Signatures
8th April 2025	· Discussed a better way for the server to find the ip · Discussed minor bugs with the timer	· Jurgen: Separate the enemy/Player scores · Andrija: Make the server broadcast the server ip · Gary: Fix issues within the menu · Jake: Allow the server to verify the score changes	Students (Initials) J.C A.D G.M J.C
			Supervisor C.M.
16th April 2025	Discussed problems with replaying Discussed a problem with the assets after replaying	Jurgen: Fix bug with assets after replaying Andrija: Fix bug with replaying Gary: Fix bug with the score buffer Jake: Add screen shake effect	Students (Initials) J.C A.D G.M JC
			Supervisor C.M.

15th May 2025	Discussed Report		Students (Initials) J.C A.D G.M JC
			Supervisor C.M.

12.2   Project Set-up

12.2.1 Prerequisites

- **Visual Studio** 2022 (or newer)
    - Make sure you have the **ASP .NET and web development** workload installed.
- A network that allows clients and server to "see" each other (same LAN or routed subnets).

Clone the Repository

Open a terminal (PowerShell, Command Prompt, Git Bash, etc.) and run:

```
git clone https://github.com/AndrijaDordevic/GAPT.git
cd GAPT
git checkout main
```

12.2.2 Locate the Game Build

In your file explorer, navigate to:

<GAPT root>

└── main

   └── Space Blocks

12.2.3 Start the Server

1. Inside **Space Blocks**, double-click or execute:

server.exe

(Allow any permissions needed)

2. A console window will open and begin listening for client connections.

12.2.4 Start the Client

1. In the same **Space Blocks** folder, launch:

main.exe

(Allow any permissions needed)

2. The client UI will appear. Click **Start Game** once you're ready

12.2.5 Networking Requirements

- **Same network segment**: For the simplest setup, connect both machines (server & client) to the same network via the same medium (Wi-Fi, Ethernet, or mobile hotspot).

- **Subnet routing**: If you're on different subnets, ensure your router allows inter-subnet routing so the client can reach the server's IP.

- **Firewall / Port forwarding**: Ensure your router or network device does **not** block port **1235** for both **UDP** and **TCP** traffic, and that any host-based firewall on the server machine allows inbound on that port.

12.2.6 Verify the Connection

Server console should log something like:

```
Client 1 connected from 192.168.0.149
```

Client console should show:

```
Waiting for server broadcast...
Received: SERVER_IP:192.168.0.149:1235
Server IP saved to file: 192.168.0.149
*Server IP: 192.168.0.149
```

Clicking **Start Game** on the client will then generate repeated "Start requested" messages on the server until both players join and the session begins.

```
Client 1 - Current ID: 1, Last ID: 0
Client 1 requested to start a game.
Clients waiting: ClientID 1 (socket=332, ready=true)
```

12.2.7 Troubleshooting

- **No "Client connected" log?**

    o Ping the server from the client to confirm reachability.

    o Verify port 1235 is allowed through any OS firewall.

- **Missing .NET runtime?**

    o Confirm your Visual Studio install includes the target .NET version.

- **Only one executable found?**

    o Double-check that both server.exe and main.exe are present in **Space Blocks**.

12.3 Extra Diagrams

Activity Diagram of the interaction between a user and a server.