**Text

Description automatically generated**

**Multiplayer Puzzle Game Using SDL**

Andrija Dordevic (0105434A), Jurgen Cauchi (ID),

Gary Ken Micallef (ID), Jake Carabott (ID)

B.Sc. (Hons) Software Development

Study-unit: **Group Applied Practical Task (GAPT)**

Code: **CIS2108**

Lecturer: **Dr Clyde Meli**

**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).

I / We\*, the undersigned, declare that the assignment submitted is my / our\* work, except where acknowledged and referenced.

I / 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.

\* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

\_\_\_\_\_\_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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date

Contents

[1. Task definition 1](#_Toc193702705)

[2. Research into the matter(s)/domain(s) relating to task(s) 3](#_Toc193702706)

[3. Overview of any similar and/or existing solutions 7](#_Toc193702707)

[4. The proposed solution (considering the preceding the three points) 8](#_Toc193702708)

[5. Task breakdown (including distribution among group members) 10](#_Toc193702709)

[6. Project plan and/or methodology of work 11](#_Toc193702710)

[7. Specification and Design 12](#_Toc193702716)

[a Principal system components and architecture 12](#_Toc193702717)

[b Data model and architecture 12](#_Toc193702718)

[c Infrastructure details (e.g. services, hardware and software used, protocols, external libraries, and reused/reusable components) 12](#_Toc193702719)

[d User interface design (if applicable) 12](#_Toc193702720)

[e Non-functional properties, such as (where applicable), usability, performance and security considerations (i.e., access control, robustness, backup and recovery considerations)). 12](#_Toc193702721)

[8. Evaluation (requirement coverage, testing strategy and results) 13](#_Toc193702722)

[9. Conclusions and future work 14](#_Toc193702723)

[10. Acknowledgements 15](#_Toc193702724)

[11. References 16](#_Toc193702725)

[12. Appendices (incl. meeting logs) 17](#_Toc193702726)

# Task definition

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. **Key Deliverables**
2. **Graphics & User Interaction (Client-Side)**
   * Use SDL to render puzzles on each client’s screen.
   * Implement smooth animations for puzzle interactions (dragging, dropping, swapping).
   * Provide visual feedback for correct and incorrect puzzle placements.
   * Ensure an intuitive and responsive UI.
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 and reconnections smoothly.
   * Develop unit tests to verify game state synchronization across clients.
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 unauthorized game manipulation.
   * Optimize server performance to handle multiple players efficiently.
5. **Puzzle Generation & Management**
   * Develop a system to generate dynamic puzzles of varying complexity (e.g., jigsaw, logic puzzles).
   * Implement difficulty scaling and randomized puzzle selection.
   * Ensure fair puzzle distribution and scoring mechanisms.
6. **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.

# 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.

**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.

**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.
* **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).
* **Animations:** SDL does not support in-built animations, so developers implement frame-based animations using **sprite sheets** or frame interpolation.
* **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**
* **SDL Documentation & LazyFoo Tutorials** (covering texture optimization, rendering techniques)
* **Game Loop Optimization**: Research shows that frame rate stability (60 FPS) enhances user experience, so delta time management (SDL\_GetTicks()) is essential.
* **Gamasutra Articles on UI/UX for Puzzle Games**: Insights on making puzzle feedback intuitive (e.g., colour changes for incorrect placements).

**2. Game Logic & Server-Client Synchronization**

**Why is Real-Time Synchronization Important?**  
In a multiplayer puzzle game, each client must receive updates from the server to ensure **all players see the same block sequence**. If not handled well, inconsistencies (desynchronization) can occur due to **network latency and packet loss**.

**Approaches to Synchronization:**

* **Client-Server Model:** The server acts as the **authoritative source** of truth, ensuring fair gameplay.
* **Timestamp Synchronization:** Using **Network Time Protocol (NTP)** or local timestamps to keep actions in sync.
* **Best Practices & Research Findings**
* **Research from Multiplayer Game Programming (GDC Talks):** Suggests sending **only necessary updates** instead of full game states to reduce network load.
* **Lag Compensation Techniques:** Valve’s networking research highlights **input prediction** and **server reconciliation** to reduce delays in user actions.
* **Use of UDP vs. TCP:**
  + **UDP** is faster but unreliable (used in FPS games).
  + **TCP** ensures reliability but has higher latency (better for puzzle games).
  + Hybrid models (TCP for important events, UDP for real-time updates) are common.

**3. Networking & Security Considerations**

* **Networking Model**
* **Sockets (Berkeley Sockets API in C++)** enable communication between client and server using **send() and recv() functions**.
* **Game State Updates:** The server should maintain a global **game state** and distribute changes using an **event-driven model**.
* **Security Concerns & Solutions**
* **NIST Cybersecurity Guidelines**: Covers secure client-server communication.
* **GDC Talks on Multiplayer Networking**: Emphasizes using encryption (TLS, SSL) for data security.

| **Threat** | **Possible Solutions** |
| --- | --- |
| **Packet Tampering** (Fake moves) | Implement **message hashing (HMAC)** to verify packet integrity. |
| **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. |

**4. Puzzle Generation & Management**

The server generates a sequence of random blocks which are then sent to the connected clients when a session starts upon request of a new block. The server stores a pointer for each client’s current tetromino and updates the pointer upon request from the client when a new block is needed.

**5. Testing & Validation**

* **Unit Testing for Multiplayer Synchronization**
* **Google Test / Microsoft Unit Testing Framework** can be used to validate:
  + **Game state synchronization (server vs. client updates)**
  + **Input handling latency (measuring response delay)**
  + **Security mechanisms (e.g., packet validation, encryption)**
* **Load Testing**
* Profiling CPU/memory usage of the server using **Valgrind** (for memory leaks).
* **Empirical Studies on Game Testing**
* **GDC & SIGGRAPH Papers on Multiplayer Testing** suggest focusing on **race conditions, network jitter, and performance bottlenecks**.

# Overview of Similar and/or Existing Solutions

**Graphics & Gameplay Mechanics**  
Popular puzzle games like Tetris Blitz, Blockudoku, and Woodoku 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, boost player engagement and encourage strategic play.

**Real-Time Multiplayer Sync**  
Games such as Tetris 99 and Puyo Puyo Tetris 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.

**Security & Anti-Cheating**  
Multiplayer games guard against cheating with server-side validation, as seen in titles like Among Us and Fortnite. Advanced techniques, such as HMAC-signed messages, unique action IDs, and replay attack prevention, help protect game integrity. Ranked modes may even include real-time input validation to deter tampering.

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

# Proposed Solution

**Graphics and Gameplay Mechanics (SDL in C++)**

The game’s graphical rendering utilizes SDL to display a 9x9 grid where blocks are placed. Drag-and-drop mechanics are implemented using SDL mouse events to facilitate smooth, real-time user input. Collision detection and grid snapping ensure blocks are positioned in valid spaces. The core gameplay includes the detection of full rows/columns, which are cleared, with a combo and scoring system rewarding chain reactions to enhance competitive gameplay.

**Real-time Multiplayer Synchronization**

The server will maintain the game state and broadcast updates to all clients in real time. To reduce bandwidth usage, only critical events (block placements, row/column clears) will be transmitted. Event timestamping resolves conflicts, ensuring that actions, such as simultaneous row clears, are processed in the correct order, maintaining fairness and synchronization across clients.

**Security and Anti-Cheating Measures**

To prevent unauthorized actions, all moves are validated server-side before being executed, ensuring only legitimate actions affect the game state. Score validation will also occur server-side to prevent manipulation. To avoid replay attacks, each move will be assigned a unique ID as well as an HMAC, preventing old packets from being reused.

**Procedural Block Generation**

The server will generate blocks dynamically, ensuring balanced difficulty and fair distribution of blocks between players. The system will prevent players from encountering unreasonably difficult block combinations, ensuring that outcomes are based on skill and strategy rather than chance.

**Testing and Performance Optimisation**

The solution will undergo unit testing for critical components such as game logic, move validation, and synchronization. Load testing will simulate concurrent games to ensure server scalability. Additionally, lag simulation will assess the game’s performance under high-latency conditions, ensuring responsiveness in less-than-ideal network environments.

# Task breakdown

**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.

**Tetromino Lifecycle**

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.

**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.

**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.

**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.

**UI Styling**

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

**Server Backbone**

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

**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.

**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.

Andrija: Project management (SCRUM), core framework, gird, networking, scoring design, security, stability

Jake: Tetromino mechanics, spawning/placement logic, audio & effects, score migration

Jurgen: Drag-drop & grid snapping, texture/UI design, server discovery, stability fixes, unit tests

Gary: Menu UI & controls, text rendering, clear-grid feature, client-score performance, unit tests

# 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.

**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.

**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 utilized, providing visual models of the system architecture and the dynamic flow of interactions within the game.

**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.

**Task Management and Work Distribution**

The project will utilize 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. 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.



# Specification and Design

## Principal system components and architecture

We split the game into a client, responsible for SDL rendering, input handling, local move validation, and TCP/UDP networking, and a server that manages sessions, maintains authoritative game state, and broadcasts minimal updates. A lightweight discovery beacon and logging service round out the design, ensuring clients can auto-find the server and operations are monitored.

## Data model and architecture

All client–server exchanges use concise JSON messages carrying session and player IDs, action or event types, payload, timestamp, and HMAC. In memory, each ‘SessionState’ holds the 9×9 grid, player-specific state (current piece, score, connection flag), and a countdown timer.

## Infrastructure details

We target a small VPS (2 vCPU, 4 GB RAM) with auto-scaling. SDL3 (plus SDL\_image and SDL\_ttf) and MiniAudio drive the client, while native sockets and OpenSSL are used to secure networking.

## User interface design

The core view is a 9×9 grid with snap-to-grid highlights, flanked by a top bar (timer, personal score, clear grid button), a bottom bar (opponent’s score) and a side bar (available tetrominos). Menus, lobby messages, and game-over overlays guide flow, while visual cues (screen shaking) and audio feedback keep interactions crisp at a steady 60 FPS.

## Non-functional properties

We aim for ≤2 clicks to matchmaking, 60 FPS rendering, and support for 100+ concurrent sessions on minimal hardware. Security is enforced via HMAC-signed messages and replay protection.

# Evaluation

# Conclusions and future work

Single player mode, high score. Create custom game

# Acknowledgements

# References

# Appendices (incl. meeting logs)