**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 puzzle state**. 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.
* **Dead Reckoning & Interpolation:** Used to predict missing data in case of lag. Dead reckoning predicts the player's next move, while interpolation smooths out movements.
* **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. |
| **DDoS Attacks** (Server overload) | Implement **rate limiting and CAPTCHA** for suspicious connections. |
| **Replay Attacks (Resending old data)** | Include **timestamps & unique IDs** to prevent old packets from being reused. |

**4. Puzzle Generation & Management**

* **Procedural Puzzle Generation**

Since the game requires **dynamically generated puzzles**, procedural generation techniques can be used:

* **Jigsaw Puzzles:**
  + Use **image segmentation algorithms** (e.g., watershed transformation) to break images into randomized pieces.
  + Research on **heuristic methods** suggests a balance between **piece complexity** and **player engagement**.
* **Word & Logic Puzzles:**
  + **Markov Chains** or **context-free grammars** can generate random, solvable puzzles.
  + **Pathfinding Puzzles:** Can use *A Algorithm*\* to ensure puzzles have unique solutions.
* **AI-Assisted Puzzle Generation**

Recent studies suggest that **machine learning models** (e.g., Reinforcement Learning) can generate balanced puzzles based on player skill. **Adaptive difficulty scaling** is also a growing trend in puzzle game research.

* **Research References**
* **AI in Games (MIT Press)** – Covers AI-driven puzzle generation.

**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**
* Simulating multiple players using **Apache JMeter** or **Locust** to analyse performance under stress.
* 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 any similar and/or existing solutions

# The proposed solution (considering the preceding the three points)

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

* **Grid-based rendering**: Using SDL to render a **10x10 or similar grid** where players place block pieces.
* **Drag-and-drop mechanics**: Handle user input (SDL\_MOUSEBUTTONDOWN, SDL\_MOUSEMOTION, SDL\_MOUSEBUTTONUP) to allow smooth placement of blocks.
* **Collision detection**: Ensure blocks can only be placed in valid positions using **grid-based snapping**.
* **Clearing rows/columns**: Logic to detect and remove full rows/columns.
* **Combo & Scoring System**: Chain reactions for multiple cleared lines should grant bonus points.

**Real-time Multiplayer Synchronization**

* **Server updates game state**: Players must see the **same grid and opponent progress** in real-time.
* **Fast event transmission**: Sending only **block placements & row/column clears** instead of full board updates reduces bandwidth usage.
* **Handling conflicts**: If two players clear rows simultaneously, the server should **prioritize timestamps** to maintain fairness.
* **Lag compensation**: The server can **predict player moves** based on past behaviour and smooth animations for late-arriving updates.

**Security & Anti-Cheating Measures**

* **Preventing fake moves**: Validate each move server-side to prevent unauthorized block placements.
* **Score validation**: Ensure scoring calculations happen server-side to prevent hacking attempts.
* **Replay protection**: Use unique IDs and timestamps for each move to prevent reusing old packets.

**Procedural Block Generation**

* **Balanced difficulty**: The server should generate block pieces in a way that prevents players from getting stuck too often.
* **Fair distribution**: Ensure both players receive the same blocks to avoid unfair advantages.

**Testing & Performance Optimization**

* **Unit tests**: Verify row-clearing logic, move validation, and server-client synchronization.
* **Load testing**: Ensure the server can handle multiple concurrent games.
* **Lag simulation**: Introduce artificial delays to test how the game reacts under high-latency conditions.

# Task breakdown (including distribution among group members)

# Project plan and/or methodology of work



# Specification and Design

## Principal system components and architecture

## Data model and architecture

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

## User interface design (if applicable)

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

# Evaluation (requirement coverage, testing strategy and results)

# Conclusions and future work

# Acknowledgements

# References

# Appendices (incl. meeting logs)