MCAST

ASSESSMENT AND INTERNAL VERIFICATION FRONT SHEET (Individual Criteria)

(Note : This version is to be used for an assignment brief issued to students via Classter)

Course Title	**BSc. Multimedia Software Development**		**Lecturer Name & Surname**	**David Deguara**	
Unit Number & Title	**ITMSD-606-1609 - Connected Gaming**				
Assignment Number, Title / Type	Home Assignment – Developing a Fully Integrated Connected Game				
Date Set	10th March 2025	Deadline Date	7th April 2025		
Student Name		ID Number		Class / Group	

Assessment Criteria	Maximum Mark
KU 1.1 - Choose different game engines for a connected game.	5
KU 1.2 - Define different connected game infrastructures.	5
KU 1.3 - Show how different game infrastructures solve connected game problems.	5
AA 1.4 - Demonstrate how data can be mined from a connected game and show how it may be implemented.	7
KU 2.1 - Choose the correct server infrastructure for different game scenarios.	5
KU 2.2 - Show how downloadable content can be split such that information can be added to a deployed game.	5
AA 2.3 - Demonstrate how downloadable content works in a real connected game.	7
AA 2.4 - Represent statistical game data in a meaningful way.	7
KU 3.1 - Identify ping and show how data transfer speeds can be calculated.	5
AA 3.2 - Establish a DLC infrastructure for a connected game.	7
SE 3.3 - Integrate a library to implement downloadable content.	10
SE 3.4 - Evaluate different methods of collecting real-world player data.	10
KU 4.1 - Illustrate how an implemented networked multiplayer game functions.	5
AA 4.2 - Research how existing libraries may be used to implement the required networked communications.	7
SE 4.3 - Evaluate the networked game by functional metrics.	10
Total Mark	100

Notes to Students:

- This assignment brief has been approved and released by the Internal Verifier through Classter.
- Assessment marks and feedback by the lecturer will be available online via Classter (Http://mcast.classter.com) following release by the Internal Verifier
- Students submitting their assignment on Moodle/Unicheck will be requested to confirm online the following statements:

Student's declaration prior to handing-in of assignment

- I certify that the work submitted for this assignment is my own and that I have read and understood the respective Plagiarism Policy

Student's declaration on assessment special arrangements

- I certify that adequate support was given to me during the assignment through the Institute and/or the Inclusive Education Unit.
- I declare that I refused the special support offered by the Institute.

MCAST

Home Assignment Guidelines

1.1.1 Overall Description

In this assignment, you are required to convert a game of chess which is uploaded on VLE into multiplayer, while also adding a DLC store and data-mining features to it. You are required to use git from the commencement of the assignment and commit regularly. The .git file showing all your commits is mandatory and failure to submit it will result in a potential mark reduction from the final grade of this assignment. Syncing your local git changes to the cloud (such as GitHub or Bitbucket) is not mandatory but is recommended. Make sure when creating your git repository you use a *.gitignore* in the appropriate location, ignoring any SDKs and Unity caches. You will be asked to explain and justify your code in a brief post-submission interview. Comments explaining crucial functionality in code are highly encouraged.

1.1.2 Pre-Submission

- Download and open the 'Chess Game' from VLE.
- Make sure you have Unity 2022.3 LTS and Firebase SDK downloaded.
- A git repository should be created before commencing the project.
- Before submitting, test your application ensure it works well.

1.1.3 Submission

Your submission should be posted in the appropriate VLE section as a compressed .zip file in the form of Name_Surname_Class.zip. Inside the zip file, you should only include:

- Unity Assets folder excluding external libraries such as Firebase SDK
- Unity Project Settings folder
- Unity User Settings folder
- A link to your public git repo.

1.1.4 Post-Submission

Following your submission, in case your git submission does not prove that you did the work yourself, you might be asked for a short interview to demonstrate and explain your project.

MCAST

Multiplayer Chess

Task 1 (42 marks)

Implementation of Turn-Based Multiplayer Using Unity Netcode

KU1.1, KU1.2, KU1.3, KU2.1, KU4.1, AA4.2, SE4.3

Overview

For this task, you are required to convert the existing single-player chess project into a turn-based multiplayer experience using Unity Netcode for GameObjects. This involves understanding various connected game infrastructures, selecting an appropriate server setup and demonstrating a functioning networked game with basic performance evaluation.

Expected Deliverables

- A fully operational turn-based multiplayer chess game using Unity Netcode with clear documentation.
- A short video or live demo showcasing two or more clients connecting, playing a chess match and handling typical scenarios (disconnects, rejoining, etc.).

Sub-Tasks

1. **Netcode Project Setup (5 marks)**
    - Install and configure Unity Netcode for GameObjects in the project.
    - Write code to initialise network sessions through local game hosting.
2. **Connection and Session Management (5 marks)**
    - Implement code to manage player connections, including joining, leaving and rejoining sessions.
    - Handle basic error cases (e.g., invalid session codes, connection failures).
3. **Turn-Based Logic Enforcement (10 marks)**
    - Write the server-side and client-side code that ensures only one player can move at a time.
    - Enforce turn transitions (e.g., White moves first, then Black, etc.) and notify clients accordingly.
4. **Board State Synchronisation (10 marks)**
    - Implement code to synchronise piece positions across all connected clients.
    - Ensure the server is authoritative for validating moves and updating the game state.
5. **Game-End Conditions (7 marks)**
    - Code to detect checkmate, stalemate, or resignation on the server.
    - Inform all clients of the outcome, ensuring the game session ends gracefully.
6. **Performance Logging and Latency Checks (5 marks)**
    - Write code to measure ping or latency between server and clients (e.g., round-trip times).
    - Log or display these metrics within the Unity console or an in-game debug panel.

MCAST

Task 2 (29 marks)

Development and Integration of a DLC Store for Custom 2D Avatar Skins

KU2.2, AA2.3, AA3.2, SE3.3

Overview

You will create a DLC (Downloadable Content) system that enables the purchase of alternate 2D skins for avatars (chess pieces). The DLC must be retrievable and integrated into the deployed game without requiring a full re-install. Other connected players should be able to see these skins, reinforcing the networked aspect.

Expected Deliverables

- A user-friendly UI that lists available 2D chess piece skins, allows purchases and displays relevant preview images.
- Evidence that newly purchased skins can be retrieved on-demand from a server or cloud location and integrated into the game at runtime.

Sub-Tasks

1. **DLC Store UI Implementation (6 marks)**
    - Write code to create a dedicated DLC store interface.
    - Display available skins (name, preview image, price) and handle user navigation.
2. **Dynamic Loading of Skins (8 marks)**
    - Implement code that fetches skin assets from Firebase Storage.
    - Ensure the game can apply these assets without a full redeployment. The assets cannot be already in the game and need to be permanently downloaded on purchase.
3. **Purchase Transaction Handling (8 marks)**
    - Write code to manage the purchase flow (e.g., checking currency/credits, confirming the transaction).
    - Persist ownership data so players retain their purchased skins across sessions.
4. **Multiplayer Visibility of Purchased Items (7 marks)**
    - Implement code that notifies all connected clients when a player uses a new skin.
    - Ensure other clients update their local displays to show the newly purchased skins.

MCAST

Task 3 (29 marks)

Integration of Firebase and Unity Analytics for Data Mining

AA1.4, AA2.4, SE3.4, KU4.1

Overview

You will record key game events (e.g., DLC purchases, match outcomes) to Unity Analytics or Firebase and also demonstrate how to save and restore game states in Firebase. You will then evaluate the methods used for collecting and presenting these data.

Expected Deliverables

- Integration of Firebase (or a similar service) to store and retrieve important data, such as user progress or match records.
- A basic analytics dashboard or in-game display showing meaningful statistics (top-purchased DLC, popular opening moves, etc.).

Sub-Tasks

1. **Library Integration (7 marks)**
    - Import Firebase SDK into the Unity project and setup the necessary configuration in Firebase.
    - Import Unity Analytics and apply any required configurations.
2. **Event Logging and Tracking (10 marks)**
    - Implement code to log relevant events, namely match start/end and DLC purchase data to Unity Analytics. Include unique identifiers or timestamps for later data analysis.
3. **Saving of Game States (6 marks)**
    - Demonstrate restoring a saved state into an active session or new session, ensuring correct synchronisation with clients.
4. **Restoring of Game States (6 marks)**
    - Demonstrate restoring a saved state into an active session or new session, ensuring correct synchronisation with clients.

MCAST

Appendix A: Documentation

UnityChess (Chess Game)

UnityChess is a 3D Chess game built using Unity/C# and adapted for the assignment from: https://github.com/ErkrodC/UnityChess/tree/development.

Debug: Should you wish to view a 2D representation of the game, you can enable the DebugView GameObject. Otherwise, it is not required and not considered part of the game.

Here is some information regarding the classes:

Board Management
The *BoardManager* class is responsible for initialising the chessboard and managing the placement of pieces. In its *Awake()* method, it creates a grid of 64 squares, each represented as a GameObject, and stores them in a dictionary for quick lookup. This class also handles actions such as casting the rook during castling moves and clearing the board when a new game starts or the game state is reset. Furthermore, it provides utility functions to determine the square closest to a moved piece and to update visual elements on the board.

Game Logic and State Management
The *GameManager* class orchestrates the game logic. It maintains the current board state, handles move execution, and manages special moves such as castling, en passant, and pawn promotion. The game state is tracked using timelines for half-moves, allowing the game to be reset to previous states. Moreover, the class supports serialization of the game state (using FEN or PGN formats), which is useful for saving and loading games. Events are raised during key game transitions, ensuring that other components, such as the UI, can respond accordingly.

User Interface Handling
The *UIManager* class governs the visual feedback provided to the player. It updates turn indicators, move history, and game result texts. The move history is dynamically managed by creating UI elements that represent each full move, and the interface supports resetting the game to any half-move state. In addition, the class manages the display of the promotion user interface, ensuring that the user is able to select the desired piece when a pawn reaches the opposite end of the board.

Piece Interaction and Visual Representation
The *VisualPiece* class manages the interactive aspect of chess pieces. It enables the pieces to be dragged and dropped across the board. On release, the class determines the nearest square based on proximity calculations and then triggers an event to notify the game manager of the attempted move. This design ensures that the visual movement of pieces remains in sync with the underlying game logic.

Singleton and Debug Utilities
A generic singleton base class, *MonoBehaviourSingleton*, is used to ensure that key classes (such as *BoardManager*, *GameManager*, and *UIManager*) have a single instance throughout the game. Additionally, the *UnityChessDebug* class provides a debug view of the chessboard by updating text and colour information for each square. This facilitates easier troubleshooting and visual verification of the board state during development.

MCAST

UnityChessLib (Chess Engine)

Important: These scripts are powering the chess engine and are of less importance for the task at hand. Please do not modify these scripts unless absolutely necessary.

AI Components

In the *UnityChess.AI* namespace, two classes are defined:

- **AssessedMove**: This class encapsulates a chess move alongside an evaluation value. It serves as a data container that pairs a potential move with its corresponding score, which is likely used in decision-making processes within the chess AI.
- **TreeNode**: This class represents a node within a search tree. It holds a board state and the current depth in the search. This structure is fundamental for implementing algorithms such as minimax, as it enables the exploration of future game states in a structured manner.

Core Board Implementation (Base Folder)

The *Board* class is central to the chess logic. It utilises an 8×8 matrix (implemented as a two-dimensional array) to represent the chessboard. Key features include:

- **Indexers and Constructors**: The class provides indexers that allow access to pieces using either a custom *Square* structure or explicit file and rank values. It offers both a constructor that initialises the board with specified square–piece pairs and a deep copy constructor to create an independent duplicate of an existing board.
- **Starting Position**: A static array, *StartingPositionPieces*, defines the standard initial placement of pieces for both sides. This facilitates the setup of a new game.
- **Move Execution and Special Moves**: The *MovePiece* method is responsible for updating the board when a move is executed. It also takes into account special moves (via inheritance from a special move class) by invoking any associated behaviour, such as castling or en passant.
- **Textual Representation**: The *ToTextArt* method generates a textual visualisation of the board, which may be used for debugging or logging purposes.

Rules and Move Legality

The static *Rules* class provides methods to verify the legality of moves and board positions:

- **Checkmate, Stalemate, and Check**: Methods such as *IsPlayerCheckmated*, *IsPlayerStalemated*, and *IsPlayerInCheck* assess the game state based on the number of legal moves available and whether the king is under threat.
- **Move Validation**: The method *MoveObeysRules* verifies that a proposed move does not violate the game rules. This is accomplished by simulating the move on a duplicate board and ensuring that the moving side does not end up in check.
- **Square Attack Determination**: The *IsSquareAttacked* method iterates over potential attack vectors. By considering various offsets, it evaluates whether a given square is threatened by an opposing piece, accounting for the distinct movement patterns of queens, bishops, rooks, kings, pawns, and knights.

The Square Structure

The *Square* structure provides a representation of a single board square. Its main attributes and functionalities include:

- **File and Rank Storage**: Each square is identified by its file (column) and rank (row), and the structure includes methods to check whether these values are within the valid range (1 to 8).
- **Operator Overloading**: Operators have been overloaded to allow for the intuitive addition of square positions and for comparing squares, which simplifies many board-related calculations.
- **String Representation**: A custom *ToString* method ensures that each square can be easily converted into a standard chess notation format.

Timeline Utility

The generic *Timeline<T>* class manages a sequential history of game states or moves (for example, half-moves). Its features include:

- **Current State Management**: The class maintains a head index that represents the current state within the timeline. This facilitates operations such as undoing moves or branching out in the case of alternate move histories.
- **Future Elements Pruning**: When a new move is added that deviates from the existing timeline, future elements are pruned, ensuring that the timeline remains consistent with the actual game progress.

Game Serialization

Within the *GameSerialization* folder, the *FENSerializer* class is implemented. Its responsibilities include:

- **Serialisation to FEN**: The *Serialize* method converts the current game state into a FEN (Forsyth–Edwards Notation) string. This involves constructing a board string that accurately represents the positions of all pieces, as well as appending information on the side to move, castling rights, en passant possibilities, the half-move clock, and the turn number.
- **Deserialization from FEN**: Conversely, the *Deserialize* method reconstructs a game from a FEN string. It parses the board representation, recreates the corresponding square–piece pairs, and restores the game conditions.
- **Helper Methods**: Methods such as *CalculateBoardString*, *GetFENPieceSymbol*, and *CalculateCastlingInfoString* assist in ensuring that the FEN representation adheres to the standard format, thereby enabling reliable game saving and loading.

Game Conditions

The *GameConditions* structure encapsulates non-board aspects of the game state, such as:

- **Castling Rights and En Passant**: It records whether each side retains the ability to castle on either side, as well as the current en passant square (if any).
- **Move Counters**: The half-move clock and turn number are maintained, which are essential for determining draw conditions and the progression of the game.
- **Updating Game Conditions**: The *CalculateEndingConditions* method computes new conditions after a half-move has been executed. It considers whether key pieces (such as kings or rooks) have moved, thereby updating castling rights, and adjusts the en passant square and half-move clock accordingly.

Game Pieces

The abstract *Piece* class defines the common properties and methods shared by all chess pieces. It stores the owner (or side) of the piece and mandates the implementation of the *CalculateLegalMoves* method. Moreover, it provides a *ToTextArt* method that utilises Unicode symbols to represent each piece in a human-readable format. The generic subclass *Piece* simplifies the process of deep copying by returning a new instance of the specific piece type with the same owner.

Individual Piece Implementations

- **Bishop**
  The *Bishop* class generates legal moves by iterating over a set of diagonal offsets. For each offset, it continuously evaluates squares in that direction until either an illegal move is detected or the path is blocked by another piece. Moves are only added if they pass the legality check performed by the rules engine.

- **King**
  The *King* class first evaluates all adjacent squares through a helper method that considers the surrounding offsets. Additionally, it handles the special castling move. The castling logic is implemented by checking that the king is not in check and that the castling rights are still available. It further ensures that the squares between the king and the rook are both unoccupied and not under attack. This careful examination reflects the complexity of castling rules in standard chess.

- **Knight**
  The *Knight* class utilises a fixed set of knight-specific offsets. It iterates through these predefined moves and adds each legal move to the result if the move does not contravene any game rules. This approach is relatively straightforward given the unique L-shaped movement pattern of the knight.

- **Pawn**
  The *Pawn* class is more intricate due to the multiple movement rules it must observe. It divides its move calculations into three distinct areas:

  - *Forward Movement*: The pawn first checks the square directly ahead. If unoccupied, a normal move is allowed. If the pawn is in its initial position, the possibility of a two-square move is also examined.
  - *Diagonal Attacks*: The pawn considers diagonal squares for capturing enemy pieces. When such a move results in the pawn reaching the far rank, a promotion move is generated.
  - *En Passant*: The class checks whether an en passant capture is applicable by verifying that the pawn is on the correct rank and that the target square meets the specific conditions for en passant.

- **Queen**
  The *Queen* class effectively combines the movement capabilities of both the rook and the bishop. It iterates over all surrounding offsets—encompassing both cardinal and diagonal directions—and continues in each direction until an obstruction is encountered. This comprehensive approach enables the queen to cover the entire board, subject to standard chess rules.

- **Rook**
  The *Rook* class focuses on moves along the cardinal directions. By iterating over a set of cardinal offsets, it examines each square along a given direction until it reaches the board's edge or an occupied square. The move is only included if it complies with the rules regarding piece movement and capture.