CS 4ZP6: Design Document
Cat and Mouse Game
Revision 1

Group #08, ClawSome Games
Yuan Gao (1330064)
Su Gao (1330065)
James Lee (1318125)
James Zhu (1317457)

Sunday 9th April, 2017

Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview

This document provides a detailed description of the design considerations and key methods and APIs which are used to implement the Cat-and-Mouse Game.

2 Design Principle

2.1 A Component-Based Architecture

The platform upon which we are developing The Cat-and-Mouse Game is the Unity 3D Game Engine, which utilises a component-based architecture.

Within this architecture, every component is encapsulated within the common GameObject class, which allows for

This architecture facilitates fluidity and consistency between the various components of the game (eg. between the Heads Up Display (HUD) and the Physics System), such that, whilst still remaining a **discrete system** in of itself, each component within the system may interact with the other components in a stable and predictable manner through **established APIs**.

Thus, the **separation of concerns** which is enforced by the component based system enables for rapid prototyping and revision of the game, such that various objects and behaviours may be swapped in or out of the system with minimal impact on the operation of the overall system. This aspect

Table 1: **Revision History**

Date	**Version**	**Notes**
11 Jan 2017	0	Design Document 0
9 April 2017	1	Revised document according to feedback provided by Customer.

is particularly useful in the construction of a system (such as a multiplayer game), which typically relies upon the composition of many small, independent systems, each performing a well-defined function.

Furthermore, as this design pattern results in a system which exhibits ***low coupling*** and ***high cohesion***, a highly parallel development life-cycle may be established, such that each component (or a set of related components—such as the HUD and the Physics systems) may be developed in parallel, whilst minimising the risk for cross-contamination and lost development time.

3 Anticipated Changes

3.1 Likely Changes

The following changes may occur during development of the project:

AC1 The amount of features and types of objects will likely be increased during development. For example, currently there are only 5 types of power-ups, but we may increase this amount as we see fit for game balance and to increase user satisfaction.

AC2 Modifications to how the characters interact with the environment are possible. Currently the player can only be blocked by walls but this may change if we decide that wall climbing or destructible terrain improves upon the user experience.

AC3 The game modes system may also be changed, allowing multiple game modes such as a time based mode to be added, rather than only having the current kills based mode.

AC4 Additional skills, abilities and actions that the players can perform will probably be added in the future.

AC5 An increase in sound effects will likely be added.

3.2 Unlikely Changes

The following changes are unlikely to occur during development of the project:

UC1 The engine we use to develop the game, we will most likely stick to using the unity engine

UC2 Currently the way networking is handled is by using the Photon Unity Network package. We will most likely not be changing the way networking is handled in our game

UC3 The programming language used is C# and will most likely stay that way.

4 Module Interface Specification

4.1 Graphical User Interface (GUI)

In this selection, we discuss the Graphical User Interface components, including those belonging to the Menu and Heads-Up Display (HUD) Interfaces.

4.1.1 COMPONENT: MainMenu

The MainMenu class allows the user to connect to the game server, set local game settings, as well as exit the game.

- loadGame(): MainMenu
  Connects to the game server and joins the Game Lobby.
- loadOptionsMenu(): MainMenu
  Launches the Options Menu to allow the user to specific local settings (such as display options).
- ExitGame(): MainMenu
  Terminates the program.

4.1.2 COMPONENT: OptionsMenu

The Options Menu allows the user to specify local game settings.

- setScreenResolution(): Integer<resolutionLevel>
  Sets the current display resolution (width x height), in pixels.
- setFullScreen(): Boolean<isFullScreen>
  Sets whether the game is displayed in Full Screen.
- setGraphicsSetting(): Integer <graphicsLevel>
  Sets the graphical quality in the game. Lower values increase performance whilst decreasing fidelity.
- setAudioEffectsLevel(): Integer<audioEffectsLevel>
  Sets the volume of the sound effects in the game.
- setAudioBackgroundLevel(): Integer<audioBackgroundLevel>
  Sets the volume of the background music in the game.
- setAudioEffectsMute(): Boolean<isAudioEffectsMuted>
  Sets whether the sound effects are muted in the game.
- setAudioBackgroundMute(): Boolean<isAudioBackgroundMuted>
  Sets whether the background music is muted in the game.

4.1.3 COMPONENT: HUD

The Heads-Up Display (HUD) allows the user to view informationr regarding and trigger events which may change the state of the environment and/or other players.

- activateSkill(): Integer<skillID>
  Activates the effects of a skill.
- updateDisplayHP(): Integer<currentHP>
  Updates the currently displayed Health Points of the player.
- updateDisplayMP(): Integer<currentMP>
  Updates the currently displayed Mana Points of the player.
- showPlayerStats(): HUD
  Displays the current statistics (including Health, attributes and skills) of the player.

4.1.4 COMPONENT: Skill

The skill module includes all the information about the abilities of the cat and mouse, such as skill name, cooldown, type, tier and so on. It includes many functions that allow the other classes to access the information, such as if the ability is on cooldown. It also updates the HUD in the Update() function when the user applies their skill points, so the skill will show up in the user's available skills. This skill consists of a few small classes, such as the SkillSlot class, which handles a skill's images and indicators on the HUD, the CharSkill class, which handles the skill descriptions as well as it's attributes, and the main class Skill which includes the Start() function, Update() function, and various other helper functions.

4.1.5 COMPONENT: Vitality

This component handles the character type, level, health points and experience points. It also displays the appropriate images for these elements, all in the bottom left corner of the HUD. The Update() function in this component constantly sets the indicators to the user's current statistics, so if the user takes damage the health bar will display the change immediately. The other helper functions such as setCurrentExperiencePoints() and getExp() are used by other classes to indicate changes towards the vitality system.

4.2 Characters

In this section we discuss about the Character components, this includes the player controlled Mouse and Cat characters.

4.2.1 COMPONENT: Cat

The Cat component handles everything related to the cat characters. These are user controlled characters that are in the team opposite of mouse characters. Cat characters have rigidbody components that are from Unity's scripting API, which allow them to be affected by Unity's physics engine. They also have colliders which handles collision detection.

Scripts related to the Cat component include the CatCamMovement and CatMovement scripts. CatCamMovement handles camera controls for a cat, it also sets the position of the camera that follows the cat. CatMovement

handles movement of the cat, it also handles functions such as ability use, jumping, and attacking.

void Update() in CatCamMovement is where everything occurs, including acquiring inputs and transformation/rotation of the camera. The function void FixedUpdate() in CatMovement handles updating the GUI when health is lost, using abilites, movement, attacking, ability pickups, and jumping.

4.2.2 COMPONENT: Mouse

The Mouse component is similar to the Cat component except it uses a different character model and colliders.

Scripts relating to the mouse are MouseCamMovement for camera controls and positioning and also MouseMovement for movement controls and actions.

void Update() in MouseCamMovement is where everything occurs, including acquiring inputs and transformation/rotation of the camera. The FixedUpdate() function does the same actions as the one in CatMovement.

4.2.3 COMPONENT: Abilities

In this section we discuss the abilities component. Abilities, which also include powerups, are picked up on the map and are spawned by the map generation functions described in the Maze section below. Scripts and functions relating to how specific abilities work are under this component. These functions include void raiseHealth(), raiseMana(), raiseSpeed(), raiseJump(), isVisible(), raiseDamage(), doubleJump().

The function raiseHealth() will increase a player's Health points by a fixed amount along with healing it. This will increase the value of the maxHealth variable. raiseMana() will increase a player's Mana points by a fixed amount, affecting the maxMana variable. raiseSpeed() will increase a character's movement speed, affecting the speed variable. raiseJump() will increase jumping height for a player and affect the jumpForce variable. raiseDamage() will increase a player's damage dealt when they attack and affect the attackPower variable. isVisible() allows users to become less visible for a set

period of time. doubleJump() will allow users of the ability to jump twice before landing on the ground.

4.2.4 COMPONENT: MonsterAI

This component handles the "brain" of the monsters in the maze. All the monsters use this same script, but they have different behaviours based on a variable `monsterType`. These monsters have 3 modes, sleeping (where they do nothing until they are attacked, upon which they switch to attack mode), patrol (where they walk aimlessly around the maze), and attack (where they follow and attack the closest player). This component handles all of the monster calculations, including attack chance, ability usage, switching of modes, and what to do when the monster is defeated. One major detail to note is that this component does not handle the movement of the monsters, it only sets the destination, as the NavMeshAgent (a built in component in Unity) will handle the movement of the monsters through the maze. The function setMonsterType() sets the monster type, playSound() plays the appropriate sound based on a type code and duration, and dealDamage() casts a sphere in front of the monster and deals damage to game objects hit. FixedUpdate() handles the animations of the monster, and Update() deals with the chances that the monster wlll attack or use abilities, as well as the timers until the monsters will change modes or until their abilities are off cooldown.

4.3 Maze

This section will be on Maze components, which creates the room and generates a maze.

4.3.1 COMPONENT: Maze

In this section, we discuss the Maze component.

The CreateRoom() function creates a new room. Function StartMazeCreation() generates the maze, calling other functions in the process. These functions include the functions which generate the arches, doors, walls, and passages.

4.4 Network

In this section we will be discussing the components that handle networking aspects of the game.

4.4.1 COMPONENT: NetworkManager

The NetworkManager class is used to manage networking in this project. In it we can connect to servers, handle network events, and how scenes are created for clients. It syncs actions when multiple clients are connected so that events happening on the screen are the same, and the maze generated is also the same between players.

void OnJoinedRoom() is that function that handles what events occur after joining a room.

void SpawnCat(), void SpawnMouse(), and void SpawnMonster() are functions that can be called in OnJoinedRoom() in order to spawn players (cat or mouse) and monsters.

5 Error Handling

5.1 Missing Files

As the game is not distributed as contained within a single executable file, there will be many folders that will contain assets and libraries that the game will require to run. Should the game download was not complete, the user delete some files, or the downloaded files are incomplete or corrupt, then the game may not run properly (eg. result in missing textures, etc). Whilst having certain missing files will not stop the execution of the game, the absence of major components (such as files defining basic game behaviour) will cause the game to behave in unexpected ways and/or terminate.

5.2 Library Initialization Failures

If the user does not have the required dependencies/libraries installed on their computer, then the game will terminate following an error message being displayed.

5.3 Synchronization Errors

As a major component of the Cat-and-Mouse game requires a consistent Internet connection in order to implement the multiplayer capabilities, there will be many instances wherein many users will be present and interacting within a shared game session.

This has the potential to give rise to synchronization errors, wherein there exists a discrepancy in the state of each user's game within the same session (eg. which may present itself as a variation in the positions of objects and players.)

5.4 Unexpected Errors

Unexpected errors occurring is still a possibility, due to the potential for the game to enter states which are unaccounted for, as well as external factors within the environment in which the system will run, which is out of the control of the development team (such as hardware and/or operating system faults).

6 User Interface Elements

6.1 User Interface

The Cat-and-Mouse game will consist of three main types of interfaces: **Menu Interfaces**, **Heads-Up Display (HUD) Interfaces**, and **Multiplayer Interaction Interfaces**.

6.1.1 Menu Interfaces

Menu interfaces allow for the user to start and specify custom settings for the operation of the game on the client-side. This includes:

- Starting and exiting a new instance of the game, termed a "client". Figure 1 shows an interface capable of this.
- Customising audio, graphical, and other aspects of the user interface, on the client. Figure 2 shows a menu interface with customization options for the game.

- Creating a new or joining an existing game session using the Multiplayer Interaction interfaces.

6.1.2 HUD Interfaces

HUD Interfaces allow for the user to provide input using devices such as the keyboard and mouse, in order trigger certain events and thus affect the state of the game session. The HUD Interfaces also allow for the user to perceive output, as a result of these state changes. For instance, players may:

- Activate a skill to affect one or more conditions and/or behaviours within the map
- Be notified on the current Health and Mana status of their character
- Be notified on the current locations and states of the other players within the map
- Send pre-defined, short messages to other users within the map

An example HUD can be seen in Figure 3, where there are 3 buttons in the center indicating skills, a health bar to the left, and a minimap to the right.

6.1.3 Multiplayer Interaction Interfaces

Multiplayer Interaction Interfaces allow for the user to create, connect to, and monitor the status of game sessions using the Photon Unity Networking Interface. Users may:

- Create or join an online multiplayer game session hosted on the Photon Cloud Network. Figure 4 shows an example of what this lobby will look like.
- Select a map, team, and game mode for the multiplayer game session via consensus with other users within the game session. Figure 5 shows an example of what this will look like.
- View the profile information (eg. username) and current status of all users within the game session

Figure 1: Example Start Screen

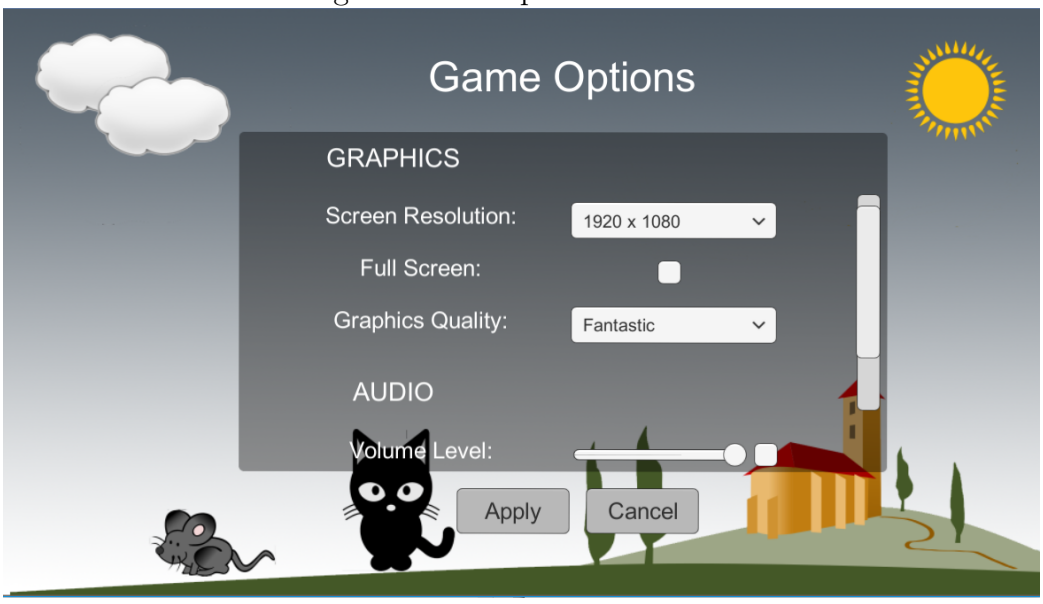

Figure 2: Example Settings Menu

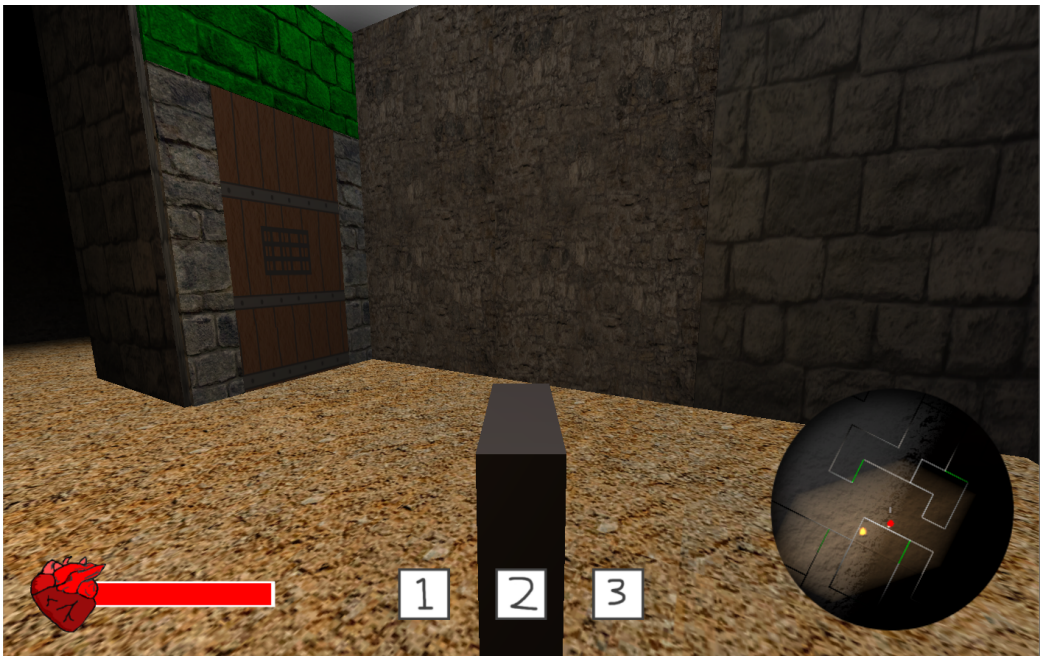

Figure 3: Example HUD

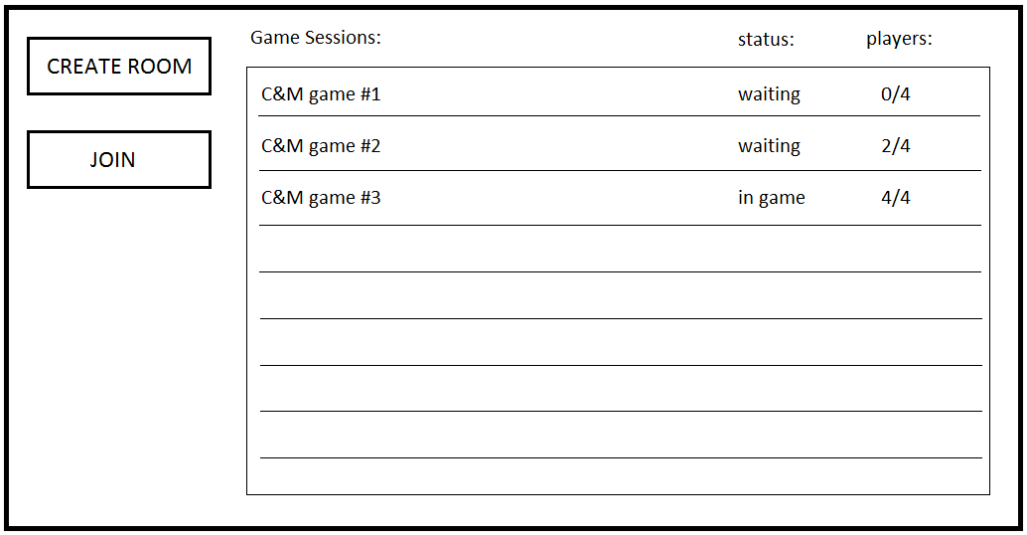

Figure 4: Example Lobby

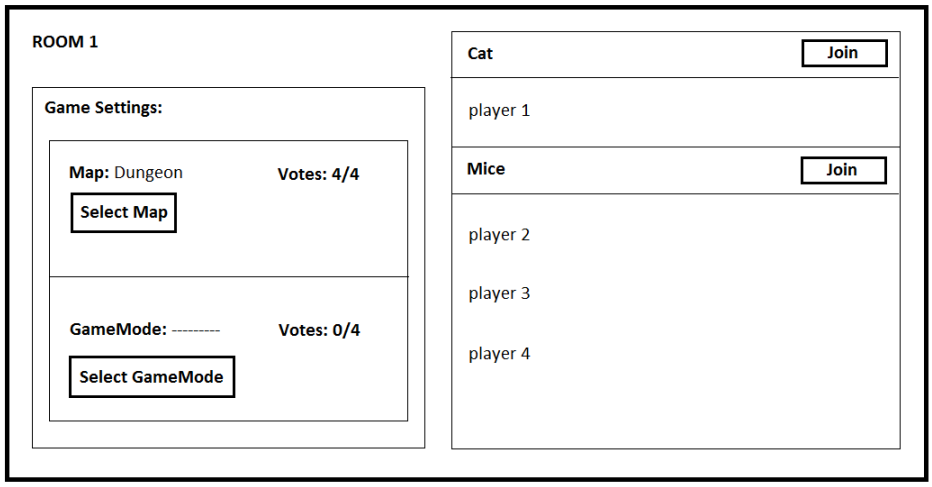

Figure 5: Example Selection Menu

7 Key Algorithms

There are many different algorithms used in the design of this game, ranging from maze generation to how each component interacts with other components and the environment.

7.1 The Maze Generation Algorithm

We wanted a maze that has no closed rooms, meaning that every room is accessible without climbing over or ignoring obstacles such as walls. To do this, we combined the use of many components and their partner scripts, and wrote an algorithm that generated lines of cells in random directions until all the cells in the predetermined space were filled in. Each cell is also created with walls which have directions assigned based on the direction of the cell creation. There were many types of cells, each with their own models such as a wall with a torch on it. After this, rooms were assigned to cells, and walls in between cells of the same room were culled. The next steps were to apply textures to all the cells, then to remove all the duplicate walls and arches that were created. Finally, special rooms were assigned based on how large

each room was, with doors replacing the arches, and these so called "puzzle rooms" are modified to match their puzzle type.

The maze algorithm was also designed in a way that would allow the exact same maze to be generated every time, given a certain seed value. This was done by using functions involving Perlin noise, cell coordinates, and a "seed" value given for generating the maze. This way, during creation of a game lobby during run time, this seed value will be able to be easily passed onto each of the players to generate the same maze.

7.2 Monster AI Algorithm

Monsters are placed at specific locations of each map to begin the game with, and have varying modes based on the game state. Each mode corresponds to a different set of actions that the monster can perform. There are three states to the monster AI, attacking, patrolling, and sleeping. =

Attacking: In this state, the monster will attack a player based on two factors, whoever attacked it last, and how far that player is from it. If the player is not in attack range, the monster will chase the player down using a Unity provided pathfinding algorithm, the NavMesh agent. After a certain amount of time, if the monster has not been defeated, it will switch back to the next mode, Patrol.

Patrol: In this mode, the monster will walk back and forth between two points, and if a player moves too close and alerts the monster, it will switch to attack mode. If the monster has not been interrupted after a certain amount of time or certain amount of cycles, then it will switch to the Sleeping mode.

Sleeping: In this state, the monster does not move, and will not perform any actions unless woken up by a player attacking it. However, there will be a chance that the monster will wake up, into the Patrol mode.

7.3 Player Abilities

There will be many types of player abilities in this game, ranging from abilities that interact with other players, the player themselves, or even with

the environment. There will be abilities shared between the cats and the mice, such as jumping or attacking, but there will also be special abilities that only the cat or only the mice will have. Even though every ability is unique, each ability will follow a general algorithm upon execution.

Upon a key press, the algorithm will first check to see what type of player pressed the ability key. Next, it will see if the player currently has access to the ability, meaning that if the player has not currently "learned" the ability, it will not be available to them for use. If the ability has been learned, then the algorithm will check to see if the ability is off cooldown. If the ability is still going through it's cooldown, or recharging period, then the ability will not be used. If the ability is ready however, then the player will perform the ability.

If the ability causes interaction between the ability and other objects to occur, then a collision check is performed, looking through each object with a certain flag in the collision radius, and performing a certain action if a collision is detected. If not, then the appropriate action dealing with no collision is then taken.

8 Data Model and Storage

The Cat-and-Mouse Game does not store any permanent state data either locally or on the server regarding any users or existing/completed game sessions, outside of any currently active game sessions. Upon termination of the game session (when all clients within the Room have exited), any temporary data which may have been stored (termed "session data") is deleted.

A limited amount of strictly *local* data may be stored on the user's system to maintain continuity of the user client between game sessions. This only includes settings which are fully localised and do not affect the behaviour of the game session on the server, such as audio or graphical settings.

9 Communication Protocols

The Cat-and-Mouse Game utilises the Photon Unity Networking (PUN) Interface to enable online multiplayer capabilities.

9.1 Photon Unity Networking (PUN) Interface

The PUN interface allows for users on instances of the game (termed "clients") to connect to a common game session, wherein they may be able to interact with each other via a shared game environment.

9.1.1 Server

The PUN handles all interactions between game clients via a **Master Server**. The Master Server facilitates the establishment of connections with and transfer of data to and from all active clients, manages the allocation of different sessions of the game, as well as ensures that the available network resources are being distributed equally amongst all clients.

The first step a client must do when wishing to access a game session is to connect to the Master Server. The master server be neutral to any one particular game client, to prevent the termination of the game for *all game clients* upon disconnection.

The **Photon Cloud**, a global distributed network of servers provided by Exit Games, will be utilised as the Master Server for the Cat-and-Mouse Game.

9.1.2 Game Lobby

The **game lobby** is which all of the clients are placed upon upon successful connection to the Master Server.

It allows for all clients to see all of the available game sessions (termed **Rooms**) which are currently active, as well as select a Room to join.

9.1.3 Rooms

Rooms are each of the instances of the game which is currently running on the Master Server. Multiple clients (up to the maximum on the number of players allowed within a game session) may be connected to any one room.

Upon connection to the **Lobby** section of a Room, clients may specify a map environment, game mode, as well as other properties for the match. All clients within the Room must arrive at a consensus regarding these properties before a Match may be started.

From this point on, the game state of all clients will be synchronised to the Master Server. Should the connection of any individual client be terminated, their game session will be terminated as well.

Upon the confirmation of the properties of a match, all clients will join a **Match**, a common environment (also termed a "map" which has been set accordingly. Players may interact with each other within the context of the Match through the triggering of events (*eg. such as activating a skill*), which then updates the common state of the Room and synchronises with all its participating clients.

10 Implementation Specifics

10.1 Platform

The Cat-and-Mouse Game is implemented on the Unity 3D Game Engine.

The game supports computer systems running:

Operating System: Microsoft Windows, XP SP2 or higher

Graphics Card: Supports DirectX 9 (Shader Model 3.0) ***or*** DirectX 11 DirectX feature level 9.3 support

CPU: Supports the SSE2 instruction set

Input: Keyboard and mouse

Networking: Broadband Internet connection. Always online

10.2 Language

The game is developed using the **C# programming language.**

10.3 Supporting Libraries

The game utilises the the following external packages and libraries (in addition to the core Unity 3D Engine):

Package/Library	**Function**
Photon Unity Networking	Unity 3D Package. Enables multiplayer capabilities through an online server.

Table 2: **External libraries and packages utilised by the Cat-and-Mouse Game**