ICS 168 Snake Game Documentation

Team Members

Kathy Chiang	55935182
Victor Hsiao	82542906
William Yang	83111331

High-level Concept

The classic game of snake re-imagined for multiple players (2-4).

System Requirements

The game’s client uses the Unity 5 engine. The game engine itself does not need to be downloaded to run the game. Simply running the executable will suffice for clients.

The game’s server is programmed in C#. The server files must be run with Visual Studio or an equivalent tool before any clients can connect to the server.

The database used is SQLite and is handled by the server.

Gameplay

- Each player controls one Snake object.
    - Each Snake object consists of one head piece and 0 or more tail pieces.
- When the Snake’s Head consumes a Food object, the player receives one point and 1 more tail piece
- Snakes must avoid Boundary objects (walls, their own tails, or the other Snake) or die.
- The objective is to get the most points and to avoid dying.
    - Dying incapacitates the Snake for a few seconds.
    - This allows the enemy Snake to gain points with no competition.
- The game ends when one Snake reaches the target score (for example, first to 25).

Optional/Possible Future Plans

- Some Food objects may have power-up effects.
    - This will make them more valuable than normal Food and highly contested.
    - Examples: Increase Snake speed, decrease tail length, slow the opponent

- There may be a separate game mode where, instead of a target score, the game lasts for a set amount of seconds (for example, a 30-second death match).

Game Objects

Snakes, Food, Boundaries, and Scenes (game screens) are the main objects of this game.

Snakes
Snakes start with one head piece that gains one tail piece for each Food object consumed by the head. Food objects are consumed when the head collides with the Food object.

When a Snake dies, a Ghost Snake remains. The Ghost Snake will not be considered a Boundary and cannot move (it just marks the placed where the Snake died). After a set amount of seconds (2-3), the Snake respawns and continues moving.

Snakes can be controlled to move North, South, East, and West (WASD or Arrow Keys).

Food
Static Food objects spawn at random locations on the screen. When a Snake eats a Food object, the Food object will disappear and the Snake will gain one tail piece and one point to the player's score.

Boundaries
The game Boundaries (walls) are set by static horizontal and vertical lines which determine the width and height of the available space for Snakes to move. Snakes are also considered Boundary objects. A Snake dies when it runs into any Boundary.

Scenes
- Title Screen
    - The title screen presents each client with the game's logo and allows them to start the game (move to the Login screen)
- Login
    - The login screen requests a username and password
    - If the username exists, it checks if the password is correct
        - If correct, the user logs in and carries on to Game Selection
        - If not, an error is displayed
    - If the user does not exist, the new user is created
        - Then, the user is brought to Game Selection
- Game Selection
    - A player may host or join a game
        - If hosting a game, the game name must not already exist

            - Once a game is successfully hosted, it is added to the list and the player moves into the Lobby
        - When joining, the game name must exist
            - If not, an error is displayed
            - If it exists, the player joins the game (as long as it is not full) and moved into the Lobby
    - Existing games are shown on a list
- Lobby
    - The lobby presents all players present with a chat box
    - Each player is also listed in the lobby with a READY button
        - Once the READY button is pressed, that person cannot press READY again
        - Once all players press READY, the game begins
- Game
    - In the game scene, players can control their own snake
    - The score is displayed at the top of the screen with each player's username
    - Once someone runs into the wall, the game is over
- Game Over
    - The game over screen allows players to return to Game Selection
    - The game is destroyed, and the winner is logged in the leaderboard on the database

Multi-user Features

Data Stored
Each player is stored in the database in a table named tb_users. Each user has an id number (which starts at 1 and increments with each new user registered), a username, and a password (hashed).

Active games are stored in a table named tb_games. Each game is recognized by an id number (which starts at 1 and increments with each new user registered), a game name which identifies the game to players, and 4 slots to store player usernames.

Finished games are then stored into a table named tb_board. This leaderboard keeps track of the winner of each game, the winning score, the game id number, and the game name. After an active game is destroyed, the same game name may be used. Therefore, the game id must be unique to maintain a unique primary key to keep track of all games in existence.

Network Interactions
Clients control the player's movements and rendering the scene. The first player also maintains the state of Food pieces. The Food is generated client-side, then sent to the server

to sync with all other clients in the same game. The same happens for every client regarding their Snake's positions.

The server stores all of this data and continually syncs it with other clients so that the players can all be rendered with their updated positions and scores.

The server also handles the database so that user logins and the leader board are consistent.

Technical Details

Server
The server code is separated into three classes: Program, Game, and Player.

<u>Program</u>

void ReadCallback(IAsyncResult)

- handles all parsing of messages from the client
    - user - represents the client's username
        - this information is stored
    - pass - represents the client's password (hashed)
        - this information is stored
        - the function *DbLogin* is called to handle the login
    - chat - represents a chat message sent by a player
        - the server appends the client's username to the front of the message
        - then, the messages are sent back to all clients to be displayed in the chat box
    - ackn - represents the client's ability to parse the next message for the client in their message queue (handled by the Player class)
        - the server gets the next message in the queue, if it exists
        - then the message is sent to the client
    - head - represents head piece data for the client
        - the head piece location for the client is sent to all other clients to render
    - tail - represents the tail pieces data for the client
        - the tail piece(s) location(s) for the client is/are sent to all other clients to render
    - list - represents the client's request to list current active games
        - a list of games are appended to a string
        - the string is sent to the client requesting the list of games
    - 1sco - represents the game's player 1 score
        - the score is stored in the correct game for the correct player
    - 2sco - represents the game's player 2 score

        - the score is stored in the correct game for the correct player
    - food - represents the food position from player 1
        - the data is parsed then echoed to all other clients to render
    - read - represents the ready check for a client
        - one ready is added to the correct game
        - if the number of readys matches the number of players in the game, a start message is sent to all clients in the game
    - host - represents a client's request to host a new game with a given game name
        - the server calls the *host* function to deal with creating a new game
    - join - represents a client's request to join a game with a given game name
        - the server puts the client into the correct game, if it exists
    - lobb - represents a client's request to know who is in the game's lobby
        - player names are extracted from the correct game
        - these names are returned to the client
    - quit - represents a disconnect or quitting of the client
        - once someone quits a game, everyone else in the game is also disconnected
        - the game the client belonged to is also removed from the database

int findGameNumWithHandler(Socket)

- a helper function that finds the game a certain client belongs to

string lobbyPlayers(Game, Socket)

- returns the members currently in the lobby to the client to display on the Lobby screen
    - if the username is the host, (host) is appended to the name
    - if the username is the player's username, (you) is appended to the name
    - new lines are added between usernames to create a roughly formatted list

void sendLocation(Game, Socket, String, String)

- called by the commands "head" and "tail"
- handles finding the correct clients to send the position data to

void host(Socket, String)

- checks if a game already exists on the server with the same game name
    - if it does, an error is returned to the client
    - if it is not, the new game is created and added to the database of active games
        - the join command is sent to the client to join their own newly created game

void join(Socket, String)

- checks if the game exists on the server
    - if it does, the join command is sent to the client

        - if it does not, an error is returned to the client

void addMessageToAllPlayers(Game, String)
- a helper function that adds a message to be sent to the client in the player's queue

List<Player> findGameWithHandler(Socket)
- a helper function that returns a list of players from the game the client belongs to

void SendToAllPlayers(List<Player>, String)
- a helper function that sends a string to all players in the list

void SendToOtherPlayers(Socket, List<Player>, String)
- a helper function that sends a string to all players in the list except one that matches the current client

void DbLogin(String, String, Socket)
- handles logging in
- checks if the username and password combination exists
    - if username and password are both correct, the command to log in is sent to the client
    - if the username exists and the password is incorrect, an error is returned to the client
    - if the username does not exist, a new entry is added to the database with the given username and password
        - then, the command to log in is sent to the client

<u>Game</u>

<u>Player</u>

Client
The client is mainly handled by a dbLogin object. This creates and maintains the connection with the server.

<u>ChatSend</u>

<u>dbLogin</u>

<u>LobbyScreen</u>

<u>Manager</u>

<u>Snake</u>

SpawnFood