# Technical Design Document:

## Logic & Algorithms:

### Menu System:

- The menu system will be built at the start of the game and will manage creating servers, joining servers, quitting the game and displaying the help information.

**Main menu:**

- We have chosen to use the mouse to control the menu's and select buttons etc. We will need to make a class that contains a sprite called CButton. This class will manage button clicks and highlighting for us. When a user moves their mouse over a button it will highlight (with a coloured border) and then once the user clicks the button, it will do what is required. The main menu will contain four buttons:
    - Join Game:
    - Host Game:
    - Help:
    - Quit:

**Lobby Menu:**

- The lobby menu system will manage holding the clients that are in the current server. The server will be sending the lobby information to each client which they will use to render then names of each player. The lobby menu will have three buttons:
    - Disconnect
    - Ready / Unready
    - Join Game

    The lobby will be handling whether a user is ready or not by having a button which toggles between ready / unready. Once all the players are ready, the game will begin. Once the game starts, the lobby will copy the information about the clients over to a new std::vector which will hold other information such as positions, colour, turning points etc.

**Host Game Menu:**

- The host game menu will contain three text fields for input:
    - Server name: (string, characters only)
    - Player name: (string, characters only)
    - Port number: (restricted to numbers between 60100 – 60200)

The player will have to fill out this information before a server can be created. The menu will have two buttons:

- Host Game

- An arrow for back (previous menu)

Once the player has chosen Host game, the game will create a client and server and join the client to the server. The game will then switch to the lobby menu and will be ready for more players to join.

**Join Game Menu:**

- The join game menu will be used to connect to already hosted servers. The menu will contain two buttons:
  - Join Game
  - An arrow for back (previous menu)

When the join game menu is opened (from the main menu) it will create a client instance which is used to ping to all ports on the IP. If a server receives this ping it will send back a pong which tells the client information about the server. For each server that it finds, it adds to a text field in the centre of the menu. Servers in this list can be selected by the user and the client will use this information to connect to the server if they select Join game.

## Game System:

- The main game system (CPlay, CSnake etc.) will only be created once the game has started. At this point the menu system and all the classes used will be released to save memory.

**Snake Processing:**

- The snake class will handle all of the client's snake processing. This will include movement, rendering and the initial snake building / creation.
- Movement will be altered using GetAsyncKeyState(). Snake has a rather unique movement system in that you cannot stop moving, and hence input (Arrow keys) will be used to change the movement direction (an enum) which is then used to add to the movement (depending on the direction).
- Another complication with snake movement is changing directions. In order to effectively make this work we need to create turning points. Each time a user pushes a new direction and they are able to turn (aligned with the grid) a turning point is created. This turning point is used in processing by checking if any of the pieces of the snake have hit the turning point, and if they have, change the movement direction to that saved in the turning point. Having the snake calculate its positions on client side (only for itself) and then sending off the information saves in calculations when rendering the other snakes. The downside to this is we need to use much more data when networking to communicate all of the positions of each snake.
- Movement is also restricted so that users cannot turn back on themselves (so reversing is impossible).

**Snake Rendering:**

- The snake will rendered by simply calling the Translate Absolute (set position) function on the sprite which corresponds to the colour of the snake being drawn, and then calling render on it. This will be called once for each snake that the server sends out.
- The client will also need to do the same render function for its own snake (In the snake class)
- Lastly, all of the turning points will be rendered as static sprites to cover up the holes or gaps made when the pieces of the snake change directions at the corners.

**Snake colours** (to cover 16 players):

| Colour Name: | Red: | Green: | Blue: | Example: |
|---|---|---|---|---|
| White | 255 | 255 | 255 | |
| Red | 255 | 0 | 0 | |
| Blue | 72 | 118 | 255 | |
| Brown | 255 | 153 | 51 | |
| Yellow | 255 | 255 | 0 | |
| Lime | 142 | 255 | 0 | |
| Aqua | 0 | 245 | 255 | |
| Teal | 0 | 255 | 127 | |
| Purple | 155 | 48 | 255 | |
| Pink | 238 | 18 | 137 | |
| Orange | 255 | 165 | 0 | |
| Green | 0 | 204 | 0 | |
| Silver | 192 | 192 | 192 | |
| Salmon | 250 | 128 | 114 | |
| Royal Blue | 125 | 158 | 192 | |
| Khaki | 240 | 230 | 140 | |

## Networking Packet Design:

We have decided to use the "Raknet" library [2] to assist us with our networking. From the previous assignment (making a UDP chat system) we feel like we have a good understanding of the underlying systems and how to setup networking through Windows. We felt like the best way to utilise our time is to use Raknet for the base networking system which saves us time. With this it allows us to spend more time thinking about what we will send through the network as well as gameplay and other mechanisms we need to create.

## Lobby:

In the lobby menu, there will be 3 things we need to send / receive:

- Name (Raknet string)
- Colour (COLORREF / RGB value)

- Ready status (bool)

The name and colour will be sent when the user enters or leaves the lobby. It will initially be sent to the server (on join) which the server will then broadcast to all of its clients to tell them to update their lobby. The ready status will be a separate packet that will be sent when a user changes whether they are ready or not.

## Game:

### Game Information packet:

The game will have a few different things to handle to keep track of the game status:

- Colour (COLORREF / RGB value)
- Length of their snake (int)
- Each position of their snake (struct containing: int iX, int iY, enum eSnakeDirection).
- The number of turning point
- Each turning point position (struct containing: int iX, int iY)

During the game, each player will be sending packets to the server containing their colour (to determine which player they are, as well as what sprite to render for that player), the length of their snake (to know how many positions to read in), each position of each piece of the snake, the number of turning points (to know how many turning point positions to read in) and each turning point. The server will store all the updated information and it will also broadcast out this information to its clients. Each client will use the received information to render the other snakes onto their screen.

### Death Notice Packet:

We will need a packet which will send if a player has died. This packet will contain nothing but the message ID for "death notice" which will tell the server they have died. The server will broadcast this information to each of its clients when this happens.

### Food Update Packet:

A packet will be made that sends information from the server to the clients containing information about the food items. This will be sent off when food objects are created or when they are eaten by clients. This same ID will also be used for sending packets to the server when they have had a collision with a food item (to tell the server to remove that food item).

- Position (int iX, int iY)

### Score Update Packet:

A packet will be sent from clients to the server containing their current score. This will be stored by the server and used to find the highest score. This highest score will be sent out to each client using the same packet ID which is used by the client's to display the current leader's score.

- Score (int)

**Tab Information Packet:**

In order to allow players to display each player's score and if they are dead or not (to create the tab display) we will need a empty packet that the client will send to the server which notifies the server that they are requesting the player's information. This will tell the server to send the required information back to that particular client.

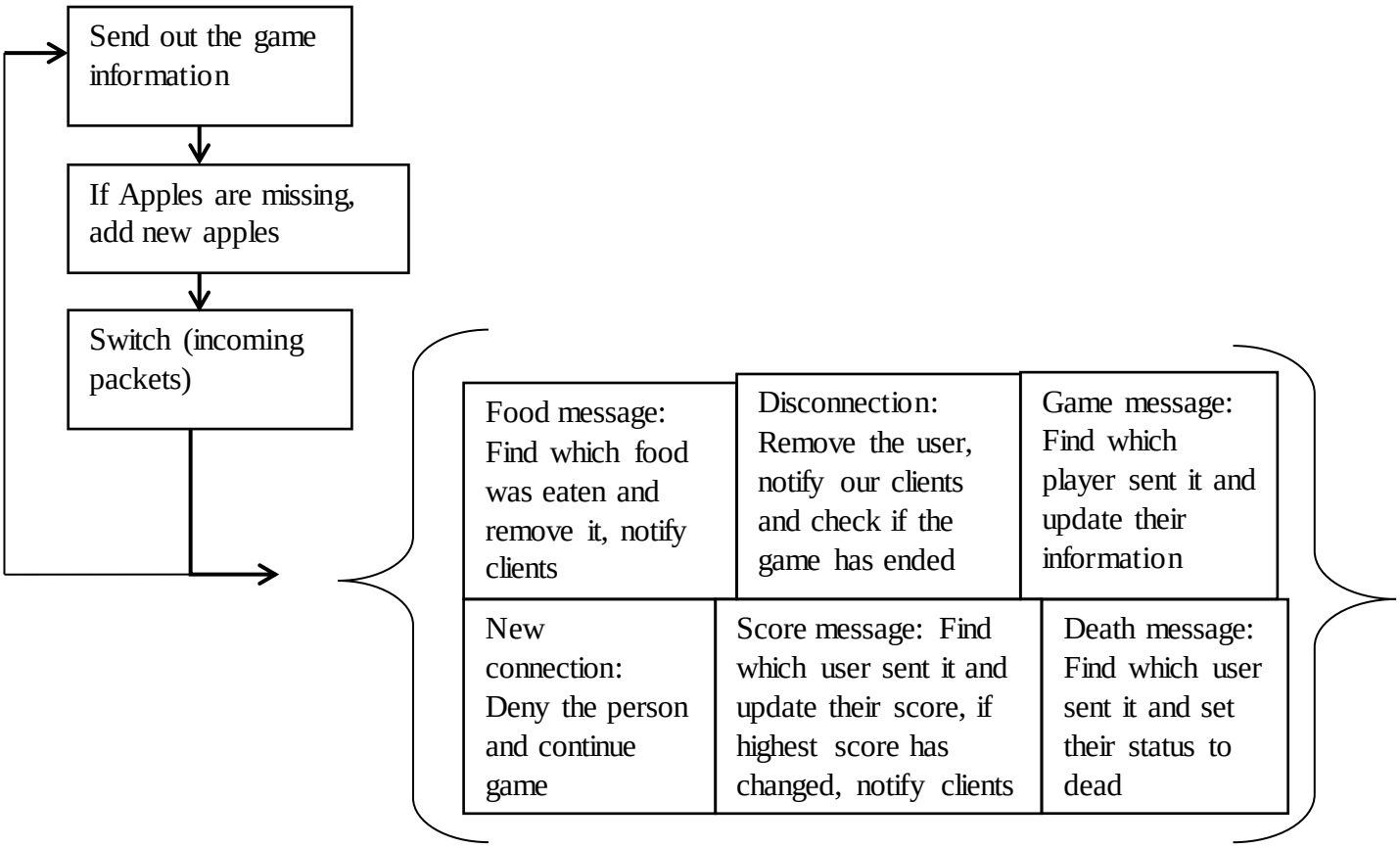
- Name (Raknet string)
- Score (int)
- Colour (COLORREF / RGB value)
- Death status (bool)

The colour will be used to tell the client which player the information is for. The name will be displayed in the tab menu with their score and death status.

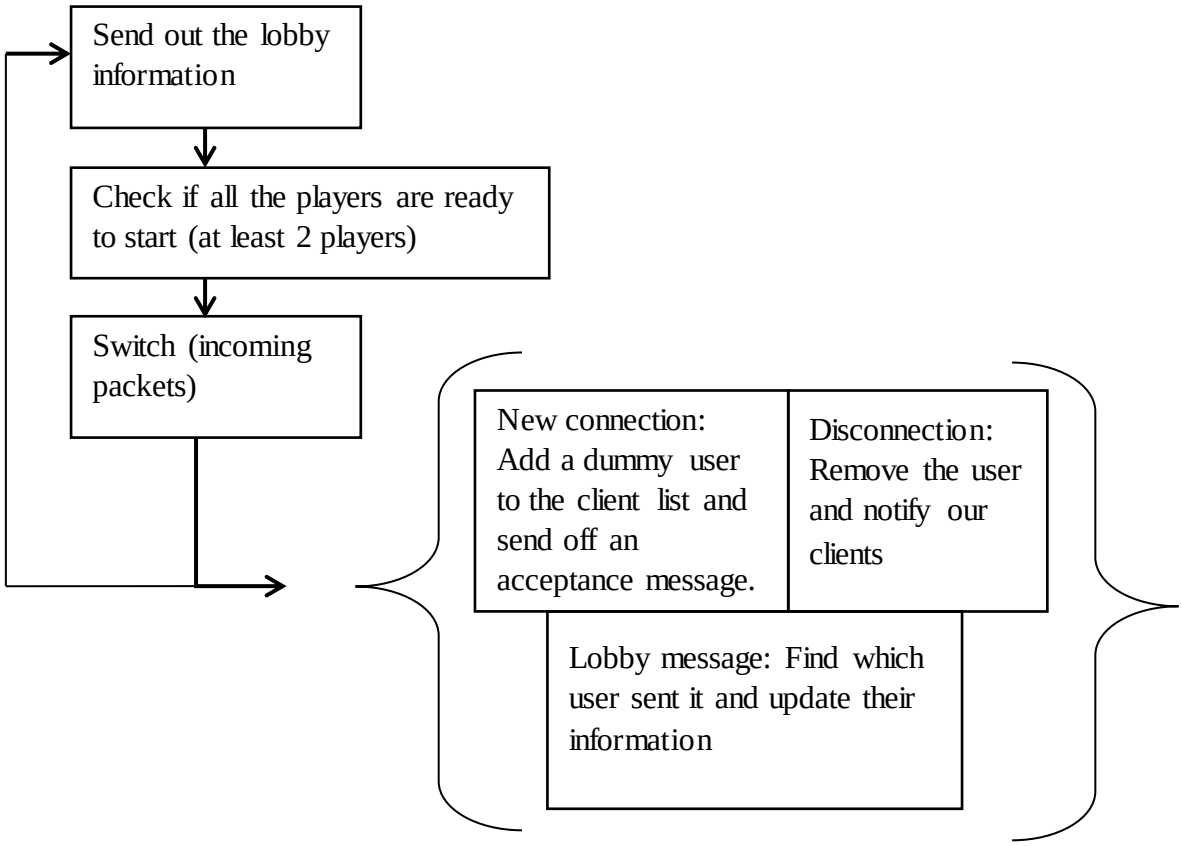
## Important Systems:

## CServer:

**In Game:**

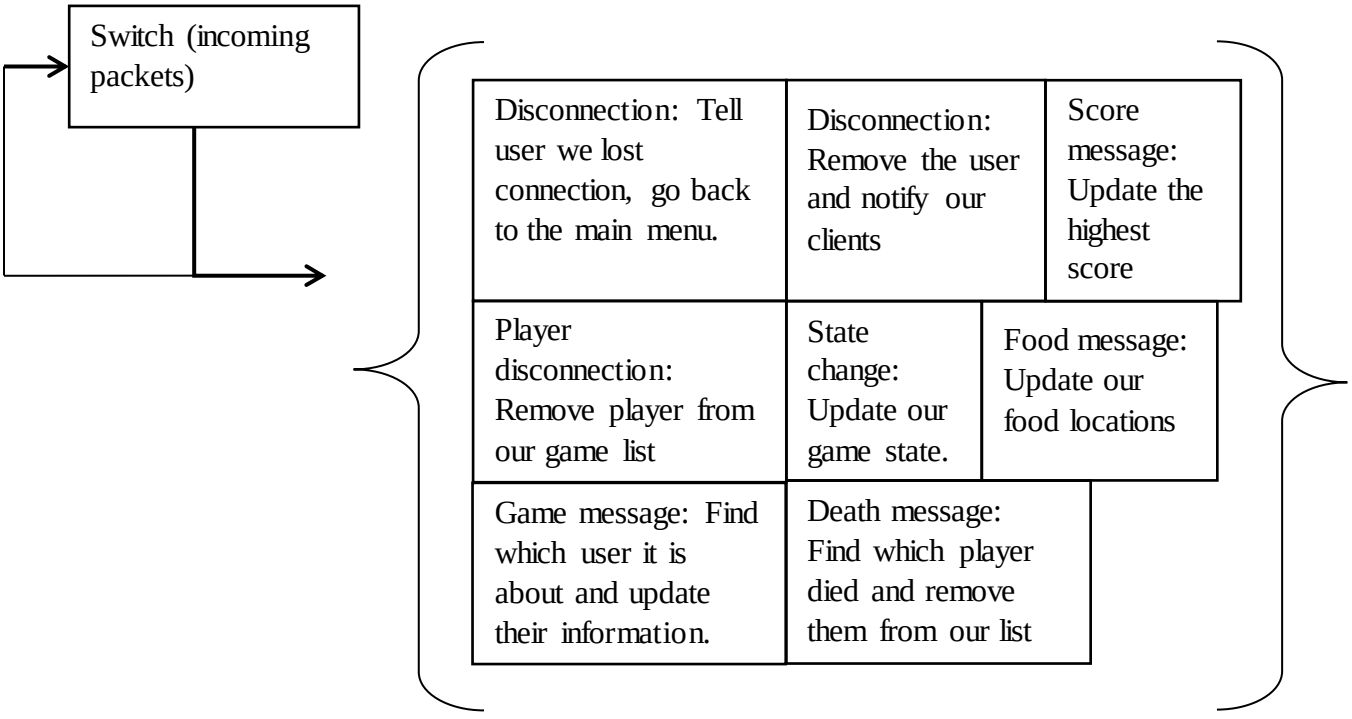


**In Menu:**

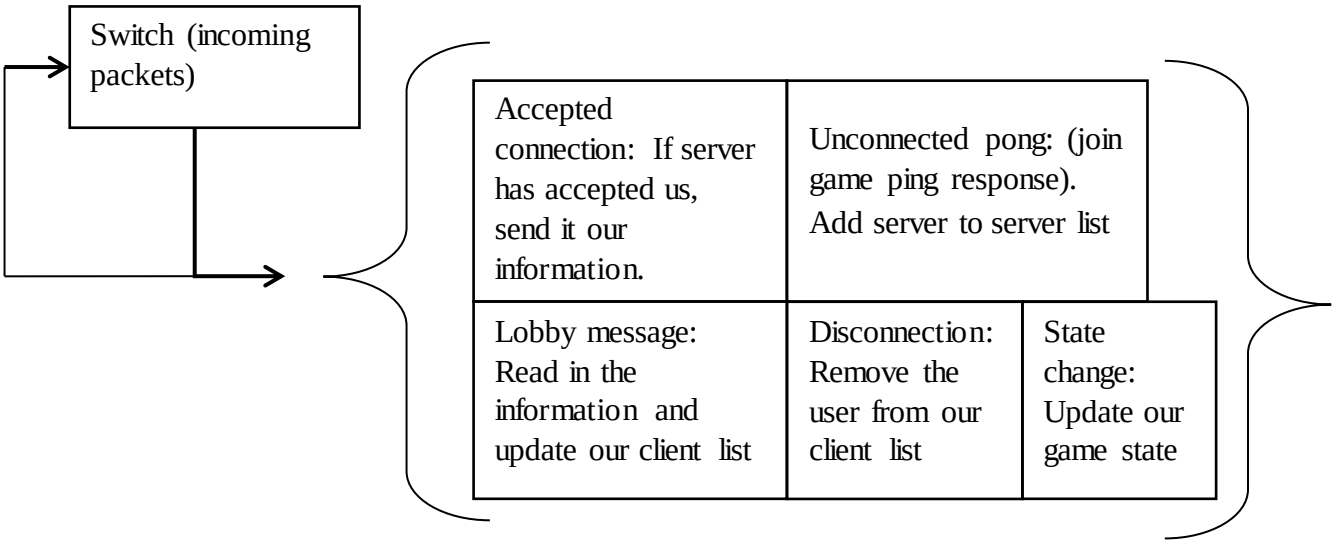



## CClient:

**In Game:**



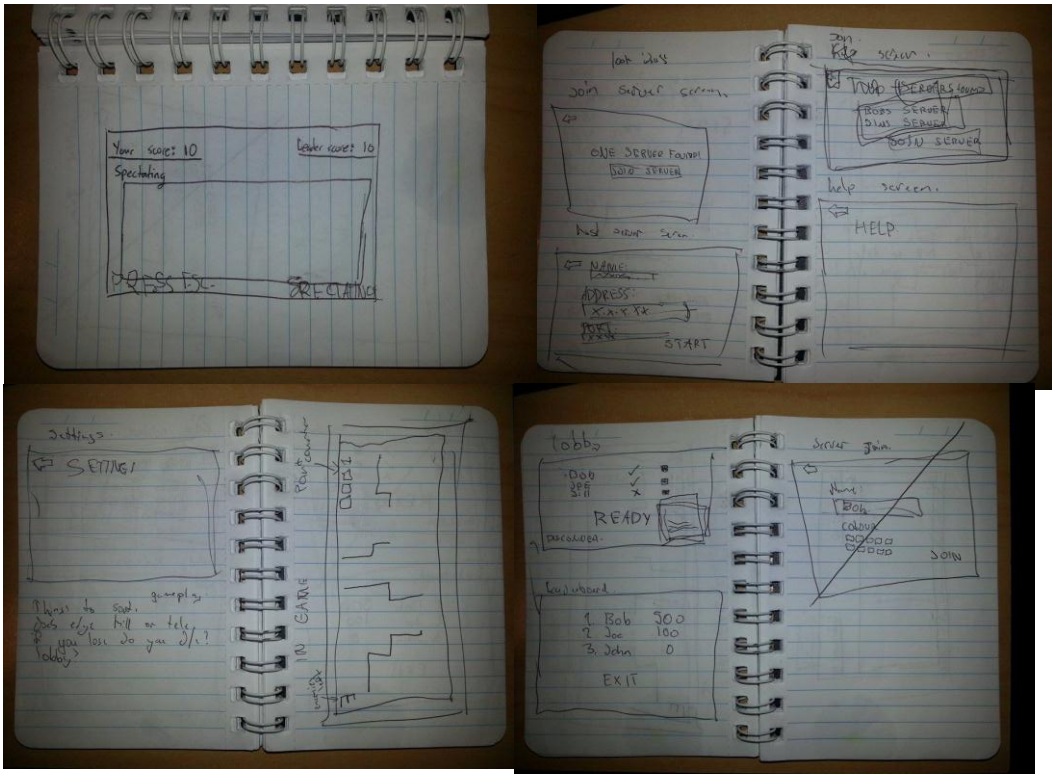
**In Menu:**




## UML Class Diagram:

See "UML Class Diagram.pdf".

## Concept Art:




## References:

[1] http://en.wikipedia.org/wiki/Snake_(video_game)

[2] http://www.jenkinssoftware.com/