COMPUTER NETWORK PROGRAMMING FINAL HOMEWORK PROJECT REPORT

Ege Kutay YÜRÜŞEN

180316017

Computer Engineering

Secondary Education

Contents

INTRODUCTION

Symbols:

 *Flag:* Flag symbol that placed by the player at the first phase of the game. An image.

 *Grid:* Area to be put on flag or capture. A button.

 *Capture Mark on Grid:* Capture mark placed by the player.

 *Captured Grid:* The grid that captured by the opponent

 *Captured Flag:* The captured flag by the opponent.

Assumptions:

- I assumed that CinsFlags War Game is a turn based, battleship-like board game.
- Game starts when one player clicked on “Host” button and the other player clicked on “Connect” to start the game. Before clicking on connect button, the player can type which ip to connect if playing on separate computers is the case.
- The host begins first, after placing all five flags and clicked on “I am Ready” button and “End Turn” then the second player has the right to play its turn.
- Grids are spread upon map according to: How large country is in terms of area, larger countries has multiple 30x30 pixel grids to be captured, small countries has only one 20x20 pixel grid to be captured.
- On first phase of the game each player has to place five flags on grids they want and then push on “I am Ready button”.
- Players can place their flags on same grid on their individual maps.
- After flags are set, the player trying to capture the opponent’s flag by clicking on grids.
- The flag that is captured once cannot be captured again by opponent player.
- The captured grid on the player by the opponent is capturable for the player self but it captures the grid of the opponent only.
- Players cannot see each other flag placed grids but they can see which grids their opponents captured.
- The player who captured all the opponent flags first wins the game.

How to Play:

1. For Host: Run application then click on “Host” button.
2. For Client: Run FlagsWarGame application click on “Connect” button. If you are playing the game on separate pc’s type the local ip of host’s pc and then click on Connect button.
3. The host should place its five flags on grids he wants then click on “I am Ready” button and then press on “End Turn”. The “End Turn” button will turn into red, which means it’s the opponent’s turn.
4. When turn switched to the connected player(client), he/she should place five flags on grids he/she wants. When all flags are place click on “I am Ready” button and end turn.
5. When the turn is switched back to the host, the host can mark a grid to capture the opponent’s flag. Once the mark placed on a grid its place cannot be changed. After placing the capture mark host should click on “End Turn” again.
6. The game goes on until one of the players captures the last remaining flag of the opponent.

PROJECT DETAILS

The project contents are covered as:

- Game Logic Class
- GUI
- TCP Server
- TCP Client
- Send Data
- Receive Data
- Appendix

Game Logic Class

```
namespace FlagsWarGame
{

    3 references
    public class Gamelogic
    {

        static int turn_count;
        private  int playerhealthleft;
        private int[] grid = new int[37];
        private int[] gridPoint = new int[37];
        2 references
        public int[] GridPoint { get => gridPoint; set => gridPoint = value; }
        private int flags_count;
        private int points;
        1 reference
        public Gamelogic()
        {
            flags_count = 5;
            points = 0;
            playerhealthleft = 5;
            for (int i=0; i < grid.Length; i++)
            {
                Grid[i] = 0;
                gridPoint[i] = 0;
            }
        }
```

Figure 1

In this class I covered game-logic side of the game which has fields that used on UI side

- grid[] : This array represents the button(grid) objects. Every element inside is initiliazed as 0, if the given index contains the flag it is set as 1.
- gridPoint[]: This array represents the button(grid) objects. Every element inside is initialized as 0, if the given index captured by the opponent's mark it becomes 1.
- flag_count: Integer to keep flag count of player.
- playerhealthleft: Represents the how many chances player to have before he/she lose.

GUI

Figure 2

Fields and Constructor:

Pointdata: used to send grid array index to the remote target

Receievedpointdata: Received grid array index from the remote target

turnCount: Integer to keep turn count.

game: Gamelogic class object.

Gridbuttons[]: grid array to store and control buttons on UI by index

Client: A static socket for client or server which depends on which button is clicked on GUI.

gameReady: Boolean for if game is ready to start.

textLog: A listbox for inform the players for events.

canCapture: Boolean for if grid is capturable or not. This variable used for preventing players clicking on same grid over and over.

```
4 references
public partial class Form1 : Form
{
    private static int pointdata;
    private static int receivedpointdata;
    private static byte[] data = new byte[1024];
    private static int turnCount;
    Gamelogic game;
    private static Button[] gridbuttons = new Button[37];
    private static Socket client;
    private Boolean gameReady;
    private static ListBox textLog;
    private Boolean canCapture;
    private static Boolean winCondition;
    private static Boolean flagcaptured;

    1 reference
    public Form1()
    {
        flagcaptured = false;
        canCapture = false;
        turnCount = 0;
        receivedpointdata = -1;
        pointdata = -1;
        gameReady = false;
        game = new Gamelogic();
        winCondition = false;
        InitializeComponent();
        InitializeGrids();
        InitButtonClicks();
        InitTextLog();
```

Figure 3

winCondition: Boolean variable to check game is over for the player or not.

Flagcaptured: Boolean variable to alert the player a flag has captured by the opponent.

InitButtonClicks() Method:

This method initializes buttons clicks a button Tag can represent any object. In this project I used grid button tags to get the index value of the button. On GUI buttons however, I used it to control click operations only. After buttons are initialized their relative button action methods are called within event handler.

```
1 reference
private void InitButtonClicks()
{
    for (int i = 0; i < gridbuttons.Length; i++)
    {
        gridbuttons[i].Click += handlegridButtonclick;
        gridbuttons[i].Tag = i; //button id

    }
    hostButton.Click += new EventHandler(ButtonHostOnClick);
    hostButton.Tag = 0;
    connectButton.Click += new EventHandler(ButtonConnectOnClick);
    connectButton.Tag = 0;
    StartGameButton.Click += new EventHandler(ButtonStartGameOnClick);
    StartGameButton.Tag = 0;
    endTurn.Click += new EventHandler(ButtonEndTurnOnClick);
    buttonDisconnect.Click += new EventHandler(ButtonDisconnectOnClick);
}
```

Figure 4

HandleGridButtonClick() Method:

In this method it controls what happens to the gridButton when a player clicks on them. There is a variable called “clickedButton” which is the button that clicked by the player.In this method I control:

- If it’s not the player’s turn he/she cannot press on buttons. When he/she does it gives an alert.
- If there is already flag on clicked grid and if the game is in first phase, then you can replace the flag by clicking on the grid again.
- After first phase of the game is over, this method checks if a grid is capturable or not to prevent capturing same grid over and over.
- If the grid is clickable after first phase it types “X” on the clicked button. There is pointdata variable which gets the index no of marked button to send the value to its opponent.

```
1 reference
private void HandleGridButtonClick(object sender, EventArgs e)
{
    Button clickedButton = (Button)sender;
    if((int)hostButton.Tag==1 || (int)connectButton.Tag == 1)
    {
        if (turnCount % 2 == 0)
        {
            if (!gameReady)
            {
                if (game.Grid[(int)clickedButton.Tag] == 1)
                {
                    clickedButton.Image = null;
                    game.Grid[(int)clickedButton.Tag] = 0;
                    game.increaseFlagsCount();
                }
                else if (game.Flags_count <= 0)
                {
                    MessageBox.Show("You dont have enough flags!");
                }
                else if (game.Flags_count > 0)
                {
                    Image img = Image.FromFile("redflag.png");
                    clickedButton.Image = img;
                    game.Grid[(int)clickedButton.Tag] = 1;
                    game.decreaseFlagsCount();
                }
                flagCount.Text = $"x{game.Flags_count}";
            }
            if (turnCount > 0)
            {
                if (gameReady)
                {
                    // it should be only clicked once per turn.
                    if (canCapture)
                    {
                        if (game.GridPoint[(int)clickedButton.Tag] != 1 && game.checkhealth())
                        {
                            clickedButton.Text = "X";
                            pointdata = (int)clickedButton.Tag;
                            game.GridPoint[(int)clickedButton.Tag] = 1;
                            canCapture = false;
                        }
                        else
                            MessageBox.Show("You already occupied this place!");
                    }
                }
            }
        }
        else
            MessageBox.Show("This is not your turn! Please wait for the other player finish its turn...");
    }
    else
        MessageBox.Show("Please Click on Host or Connect button to start the game!");
```

Figure 5

ButtonStartGameOnClick() Method

"Im Ready" button method to control events when it's clicked on it. It checks if all five flags are placed and if so, its sets the gameReady as true and changes the button to blue color. It can be only clicked once.

```
1 reference
private void ButtonStartGameOnClick(object sender, EventArgs e)
{

    if (gameReady)
        MessageBox.Show("You already placed all of your flags");
    else if (game.Flags_count == 0)
    {
        gameReady = true;
        Color slateBlue = Color.FromName("SlateBlue");
        StartGameButton.BackColor = slateBlue;
        StartGameButton.Tag = 1;
    }
    else if (game.Flags_count > 0)
        MessageBox.Show("Please place all of your flags");
}
```

Figure 6

TCP Server

ButtonHostOnClick() Method

```
private void ButtonHostOnClick(object sender, EventArgs e)
{
    if ((int)hostButton.Tag == 0&&(int)connectButton.Tag==0)
    {
        try
        {
            textLog.Items.Add("waiting for a client...");
            Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
            IPEndPoint iep = new IPEndPoint(IPAddress.Any, 9050);
            server.Bind(iep);
            server.Listen(5);
            server.BeginAccept(new AsyncCallback(AcceptConn), server);
            MessageBox.Show("Game is hosted successfully... waiting for second player");
            hostButton.Tag = 1;
            Color hostColor = Color.FromArgb(0, 255, 0);
            hostButton.BackColor = hostColor;
            censorPicture.Visible = false;

        }
        catch (System.Net.Sockets.SocketException)
        {
            MessageBox.Show(" ERROR ! THE SOCKET IS IN USE");
        }
    }
    else
        MessageBox.Show("Game is already hosted");
}
```

Figure 7

At the very beginning of the game if one of the players clicks on "Host" button the method at Figure 7 triggers. It creates a Socket called server which is type of ipv4, stream type socket, TCP protocol, then creates an ip end point to bind the socket with the device ip and the port. After binding process is over, server start to listen for the client. The server makes an async callback which means the server accepts the client whenever server is ready to accept.

The "Host" button can be clicked once and if "Connect" button is already clicked, you cannot click on "Host" button in rest of the game. If there is already program running at local ip with 9050, port it gives a "Socket is In Use" error. Which means there is already a program running at that port. An asynchronous client or server socket does not suspend the application while waiting.

AcceptConn() Method

```
void AcceptConn(IAsyncResult iar)
{
    try
    {

        Socket oldserver = (Socket)iar.AsyncState;
        client = oldserver.EndAccept(iar);
        Action<byte[]> add = str =>
        {
            textLog.Items.Add("Connected to: " + client.RemoteEndPoint.ToString());
            textLog.Items.Add("Your turn!");
        };
        BeginInvoke(add, data);
        Thread receiver = new Thread(new ThreadStart(ReceiveData));
        receiver.Start();
    }
    catch (System.InvalidOperationException)
    {
        MessageBox.Show("Warning! Please try on running the app from FlagsWarGame>bin>Debug>FlagsWarGame.exe");
    }
}
```

Figure 8

The method in Figure 8 is invoked after the client is connected to the host (server). A socket object called oldserver is instantized which is belongs to "server" socket in Figure 7. After the process, the client is carried on "client" socket which is instantized in fields of the Form1 class. After this process is done it writes client's ip. A Thread is created for receiving data from client. The receiving data side of the server structure is done in another thread besides GUI. Sending and receiving data side of the server structure is covered in other contents since they are the same method for client as well.

TCP Client

ButtonConnectOnClick() Method

This method is invoked when the user click on "Connect" button.

The ButtonConnectOnClick method initializes client socket, then IPEndPoint is defined for target server ip. IPEndPoint is changeable by the user if the connection hasn't made yet in order to connect the server from another computer. Default is iep is 127.0.0.1.

```
void ButtonConnectOnClick(object obj, EventArgs ea)
{
    if ((int)hostButton.Tag == 0 && (int)connectButton.Tag == 0)
    {
        client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        IPEndPoint iep = new IPEndPoint(IPAddress.Parse(textBoxIP.Text), 9050);
        client.BeginConnect(iep, new AsyncCallback(Connected), client);
        connectButton.Tag = 1;
        turnCount = 1;
        censorPicture.Visible = false;
        connectButton.BackColor = Color.FromArgb(0, 255, 255);
    }
    else if ((int)connectButton.Tag == 0 && (int)hostButton.Tag == 1)
    {
        MessageBox.Show("Warning! You can't connect to the host because you are the host!");
    }
}
```

Figure 9

Client.BeginConnect line implements the AsyncCallback delegate. It connects to the remote device when the remote device is available and then signals the application thread that the connection is complete.

The reason why turnCount is set to 1 here is to manage turn order between players. This button can be clicked only once and if "Host" button is clicked before, you get an alert.

Connected() Method

After connection to server is completed it creates a thread to receive the data.

```
1 reference
void Connected(IAsyncResult iar)
{
    try
    {
        client.EndConnect(iar);
        Action<byte[]> add = str =>
        {
            textLog.Items.Add("Connected to: " + client.RemoteEndPoint.ToString());
        };
        BeginInvoke(add, data);


        Thread receiver = new Thread(new ThreadStart(ReceiveData));
        MessageBox.Show("Successfully Connected to the host");
        receiver.Start();
        }
    catch (SocketException)
    {
          MessageBox.Show("Error on connecting...Please make sure there is a host to connect");
    }
}
```

Figure 10

Send Data

ButtonEndTurnOnClick() and SendData() methods

This method is invoked when all flags are placed. If player tries to click the button or if its not the player turn it gives alert. When all conditions are met, it increases the turn count by one. If the player's turn count is divisible by two it means, it's the player's turn. So, increasing the number by one means losing control of buttons for the player, when the data is received from the opponent the turn count is increased by one and turn count becomes an even number again. After that it creates a byte array which keeps the byte of the "pointdata" variable.

```
private void ButtonEndTurnOnClick(object sender, EventArgs e)
{
    if (gameReady){
        if (!canCapture)
        {
            if (turnCount % 2 == 0)
            {
                turnCount++;
                byte[] integerData = BitConverter.GetBytes(pointdata);
                try
                {
                    client.BeginSend(integerData, 0, integerData.Length, 0, new AsyncCallback(SendData), client);
                    endTurn.BackColor = Color.FromArgb(255, 0, 0);
                }
                catch (System.NullReferenceException)
                {
                    MessageBox.Show("Error there is noone to send data!");
                    client.Close();
                }

            }
            else
                MessageBox.Show("This is not your turn!");
        }
        else
            MessageBox.Show("Please click on a grid to mark");
    }
    else
        MessageBox.Show("Please after placing all of your flags click on I am Ready button!");
}
```

Figure 11

Pointvariable is keeps the index number of which grids is marked. After converting pointvariable to byte array. The method send data to the receiver and SendData() is invoked by an async callback. In the SendData method, the data is sent whenever the remote target ready to receive the data.

```
private void SendData(IAsyncResult ar)
{

    Socket remote = (Socket)ar.AsyncState;
    remote.EndSend(ar);
    canCapture = true;

}
```

Receive Data

```
private void ReceiveData()
{

    int recv;
    int intData;
    while (true)
    {
        recv = client.Receive(data);
        //stringData = Encoding.ASCII.GetString(data, 0, recv);
        intData = BitConverter.ToInt32(data, 0);

        receivedpointdata = intData;
        flagcaptured = false;
        if (receivedpointdata <= gridbuttons.Length)
        {
            turnCount++;
            textLog.Invoke((MethodInvoker)delegate
            {
                textLog.Items.Add("Your turn!");
                endTurn.BackColor = Color.FromArgb(255, 255, 255);
            });
        }

        if (intData > -1&& receivedpointdata <= gridbuttons.Length)
        {
            gridbuttons[receivedpointdata].Invoke((MethodInvoker)delegate
            {
                gridbuttons[receivedpointdata].BackColor = Color.FromArgb(127, 127, 127);

                if (game.Grid[receivedpointdata] == 1)
                {
                    game.Playerhealthleft--;
                    textLog.Items.Add("Your oppenent captured your flag! Flags left: "+game.Playerhealthleft);
                    flagcaptured = true;

                    if (game.Playerhealthleft <= 0)
                    {
                        winCondition = true;
                        textLog.Items.Add("You lost the game ! your opponent captured all of your flags!");
                        MessageBox.Show("You lost the game");
                    }
                }
            });
```

Figure 12

Two local variables are defined which are: recv to keep the size of buffer of the received data and intData to get the index of the button array. Since this function is on a separate thread, it keeps running the codes in the while loop to get the data almost same time when data is sent from the opponent player. Then the receivepointdata gets the index of the grid from intData. Since receivepointdata is a static int variable it is visible for both threads. Then it checks if the player received any data lesser than gridbutton count, the reason why it checks this is because there are special definitions for received or send data such as integer 888 and integer 999 (their condition is defined in Figure 13). To access GUI elements from separate thread, the Invoke method is a must. So, I used invoke method anywhere that contains event about GUI elements. Inside of the invoked method on Figure 11, If the received data contains the index of a flag on a grid the player's health reduced by one.

```
    if (flagcaptured)
    {
        byte[] flagcapturedcode = BitConverter.GetBytes(888);
        client.BeginSend(flagcapturedcode, 0, flagcapturedcode.Length, 0, new AsyncCallback(SendData), client);

    }
    if (receivedpointdata == 888)
    {
        Action<byte[]> add = str =>
        {
            textLog.Items.Add("You captured one of your opponents flag !");
        };
        BeginInvoke(add, data);
    }
    if (receivedpointdata == 999)
    {
        Action<byte[]> add = str =>
        {
            textLog.Items.Add("You won the game!");
        };
        BeginInvoke(add, data);

    }
    if (winCondition)
        break;
}

byte[] integerData = BitConverter.GetBytes(999);
client.BeginSend(integerData, 0, integerData.Length, 0, new AsyncCallback(SendData), client);
return;
```

Figure 13

In the continuation of the code, there is a condition that if the opponent player captures the player's flag it sends the information to the opponent player by integer as 888. The information is texted to the textLog that he/she captured the flag of the opponent. If the player captures the last flag of it's opponent. The wincondition is set to true, so it breaks the while loop. After the loop breaks, the player sends the integer 999 to the opponent which texts information about he/she won the game.

After win condition is met, grid buttons are locked.

Appendix

There is also a disconnect button to disconnect safely, if you are open the application from bin folder, I highly recommend use the of disconnect button. Otherwise, the socket is stuck in the background.

The program is tested on Visual Studio 2019 Professional on separate PCs.

Sources used during developing the project:

- https://stackoverflow.com/questions/661561/how-do-i-update-the-gui-from-another-thread
- Richard Blums-Net Programming C# book
- https://www.youtube.com/watch?v=n5WneLo6vOY guide for learning C# Windows Forms

A picture from the end of the game: