**네트워크 프로그래밍 설계과제**

**[멀티플레이 게임 – 색판뒤집기]**

**제출일: 2022년 06월 08일**

**팀명: 1인조**

**팀원: 2019152022 박소윤**

**[목차]**

1. **문제 정의 및 요구사항 분석**

* **문제 정의**
* **설계과제 목표**
* **요구사항 분석**

1. **소프트웨어 설계**

* **소프트웨어 설계도**
* **메시지 정의 및 포맷 설명**
* **클라이언트-서버 구현 방법**

1. **소프트웨어 구현 및 데모**

* **주요 소스코드 설명**
* **데모 시나리오**
* **프로그램 실행결과 캡처 및 설명**
* **현실적 제한 요소의 반영내용 기술**
* **응용프로그램 성능 분석**

1. **팀워크**
2. **결론 및 총평**
3. **부록: 프로그램 소스**

**[문제 정의 및 요구사항 분석]**

1. **문제 정의**

* 실시간으로 다른 사용자들과 같이 플레이할 수 있는 멀티플레이 게임을 만들고자 함

1. **설계과제 목표**

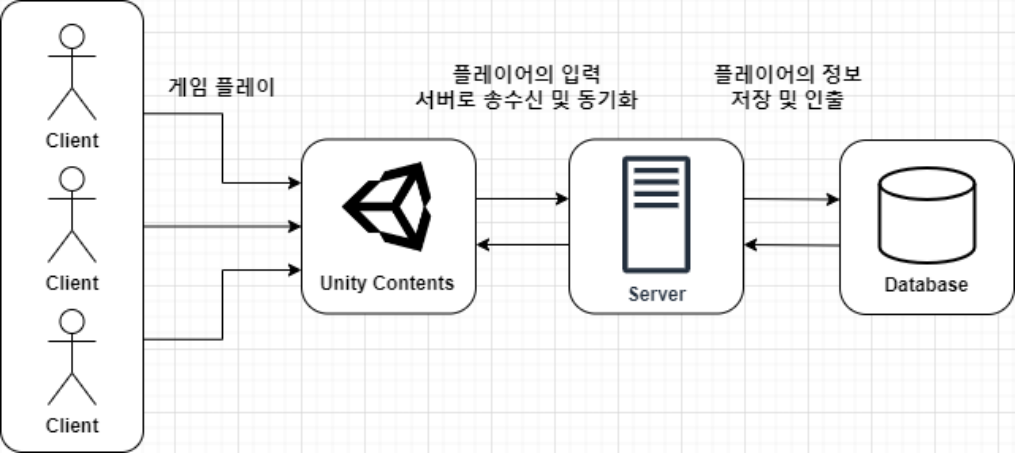
* 색판 뒤집기 게임: 제한시간 내에 최대한 많은 색판을 뒤집는 게임
* Unity를 이용하여 게임 구현 및 C#으로 스크립트 코딩
* DB를 사용하여 로그인, 회원가입 등의 기능 구현
* 병행처리가 가능한 스크립트
* 경쟁상태가 발생하지 않도록 임계영역 설정

1. **요구사항 분석**

* 플레이어는 회원가입 후 로그인을 통해 게임에 접속
* 회원가입 및 로그인 시 에러가 발생하면 디버깅 메시지를 띄움
* 방 안의 플레이어가 모두 준비 상태가 되어야 게임 시작
* 게임이 시작되면 플레이어들에게 특정 색깔이 배정됨
* 사용자가 색판(Default: 회색)을 누르면 자신의 색으로 색판의 색깔이 바뀜
* 제한시간이 끝난 후 가장 많이 색판을 자신의 색으로 바꾼 플레이어가 우승
* 게임이 끝나면 플레이어 순위표 보여줌

**[소프트웨어 설계]**

1. **소프트웨어 설계도**



1. **메시지 정의 및 포맷 설명**

* **Server -> Client**
* welcome: 서버와 클라이언트가 성공적으로 연결되었을 때 전송
* loginOrRegisterSuccess: 클라이언트가 로그인 혹은 회원가입에 성공했을 때 전송
* spawnPlayer: 클라이언트가 방에 입장하였을 때 전송
* playerReady: 클라이언트가 Ready 상태로 전환됐을 때 전송
* gameStart: 모든 클라이언트들이 Ready 상태일 때 전송
* playerPosition, playerRotation, playerFlipped: 클라이언트의 입력에 따른 처리
* playerDisconnected: 클라이언트가 접속 해제했을 때 전송
* **error: 오류 메세지 전송**
* **Client -> Server**
* welcomeReceived: 서버가 보낸 welcome 패킷을 제대로 수신했을 때 전송
* playerEnter: 클라이언트가 로그인에 성공한 후 방에 입장하였을 때 전송
* playerReady: 클라이언트가 Ready 상태로 전환했을 때 전송
* playerMovement, playerFlip: 클라이언트의 입력이 들어왔을 때 전송

1. **클라이언트와 서버의 구현 방법**

* 개발 언어: C#
* 클라이언트, 서버 모두 Unity 2019.4.7f1 에디터를 이용하여 구현
* Unity 에디터는 기본적으로 단일 스레드 방식 사용
* CPU 사용량 감소를 위해 네트워킹 부분은 비동기식 메소드를 사용하여 병행처리
* 여러 클라이언트들의 입력을 컨트롤해야 하므로 경쟁상태로 인한 오류를 방지하기 위해 특정 메소드들을 스케쥴링하는 클래스 정의
* Sqlite를 이용하여 데이터베이스 구축 및 서버와 연결

**[소프트웨어 구현 및 데모]**

1. **주요 소스코드 및 설명 - 서버**

* **Server.cs: Start(), 서버 실행**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* C# System.Net.Sockets 라이브러리 사용
* TCP listener를 열고 클라이언트의 접속을 기다림
* UDP Client를 생성하고 클라이언트가 메시지를 보내기를 기다림
* UDP 프로토콜은 Unreliable하나, 전송 속도가 매우 빠르므로 로컬 플레이어의 위치, 회전 값을 다른 플레이어들과 동기화하기 위해 사용
* 비동기식 메소드 사용, 콜백함수 등록
* DB 파일이 없다면 생성하고, DB와의 연결 상태를 체크함
* **Server.cs: InitializeServerData(), 패킷 핸들러 등록**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 클라이언트 관리를 위한 Dictionary<id, Client 객체> clients 초기화
* 패킷 핸들러 등록(클라이언트로부터 받는 패킷에 대한 액션 핸들러)
* **Server.cs: TCPConnectCallback(), 클라이언트와 TCP 연결 시 콜백**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* **Server.cs: UDPReceiveCallback(), UDP 메시지 수신 시 콜백**

텍스트이(가) 표시된 사진

자동 생성된 설명

* 메시지를 보낸 클라이언트가 서버와 연결되지 않았다면 연결
* 이미 연결된 상태라면 데이터를 주고받음
* **ServerSend.cs: SendTCPData(), SendUDPData(), 해당 프로토콜 패킷 전송**

텍스트이(가) 표시된 사진

자동 생성된 설명

* 패킷 데이터의 길이를 계산하여 패킷의 맨 앞에 추가
* 해당 클라이언트 ID를 가진 클라이언트에게 데이터 전송
* 모든 클라이언트에게 패킷을 전송하는 SendTCPDataToAll, SendUDPDataToAll 메소드도 존재

1. **주요 소스코드 및 설명 - 클라이언트**

* **Client.cs: OnConnectedToServer(), 서버와 클라이언트 연결**

텍스트이(가) 표시된 사진

자동 생성된 설명

* TCP, UDP 객체 초기화 및 TCP 연결
* **Client.cs: InitializeClientData(), 패킷 핸들러 등록**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 패킷 핸들러 등록(서버로부터 받는 패킷에 대한 액션 핸들러)
* **Client.cs: TCP 클래스 – Connect(), ConnetCallback()**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* Socket의 send, recv 버퍼 크기 지정해줌
* Ip, port#을 통해 서버와의 연결 시도
* 연결에 성공한다면 콜백함수 호출, 서버로부터 데이터 읽기 시작
* **Client.cs: TCP 클래스 – SendData(), 서버로 TCP 패킷 전송**

텍스트이(가) 표시된 사진

자동 생성된 설명

* 비동기식 메소드 사용(BeginWrite())
* **Client.cs: UDP 클래스 - Connect(), SendData()**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* UDP는 Connect 과정 없이 서버와 메시지를 주고 받으면 연결 성립
* 비동기식 메소드 사용(BeginSend())

1. **주요 소스코드 및 설명 - 패킷**

* **Packet.cs**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* Enum 형태로 패킷 이름 정의

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

* Read, Write 메소드 존재, 패킷에 데이터를 읽거나 쓸 수 있음
* **ServerSend.cs: Welcome(), Welcome 패킷 전송**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* **ClientHandle.cs: Welcome(), Welcome 패킷 수신**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 패킷 송수신 예시
* 송신 측에서 packet에 Write()한 순서대로 수신 측에서 Read() 해야 함

1. **주요 소스코드 및 설명 - 스케쥴링**

* **ThreadManager.cs: ExcuteOnMainThread()**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 경쟁상태를 일으킬 수 있는 메소드를 스케쥴러에 추가
* **ThreadManager.cs: UpdateMain()**

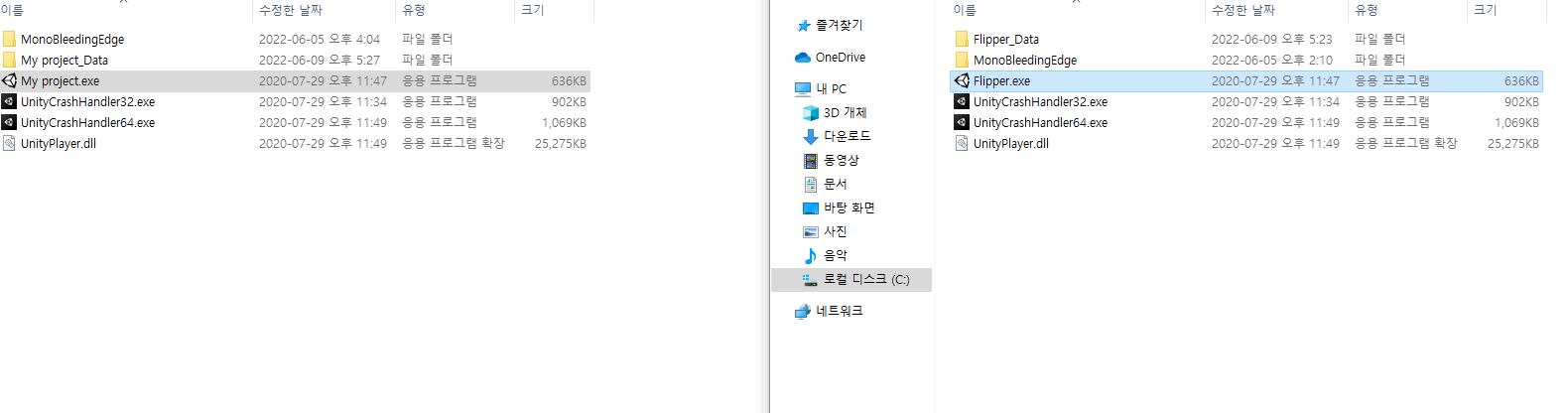
**텍스트이(가) 표시된 사진

자동 생성된 설명**

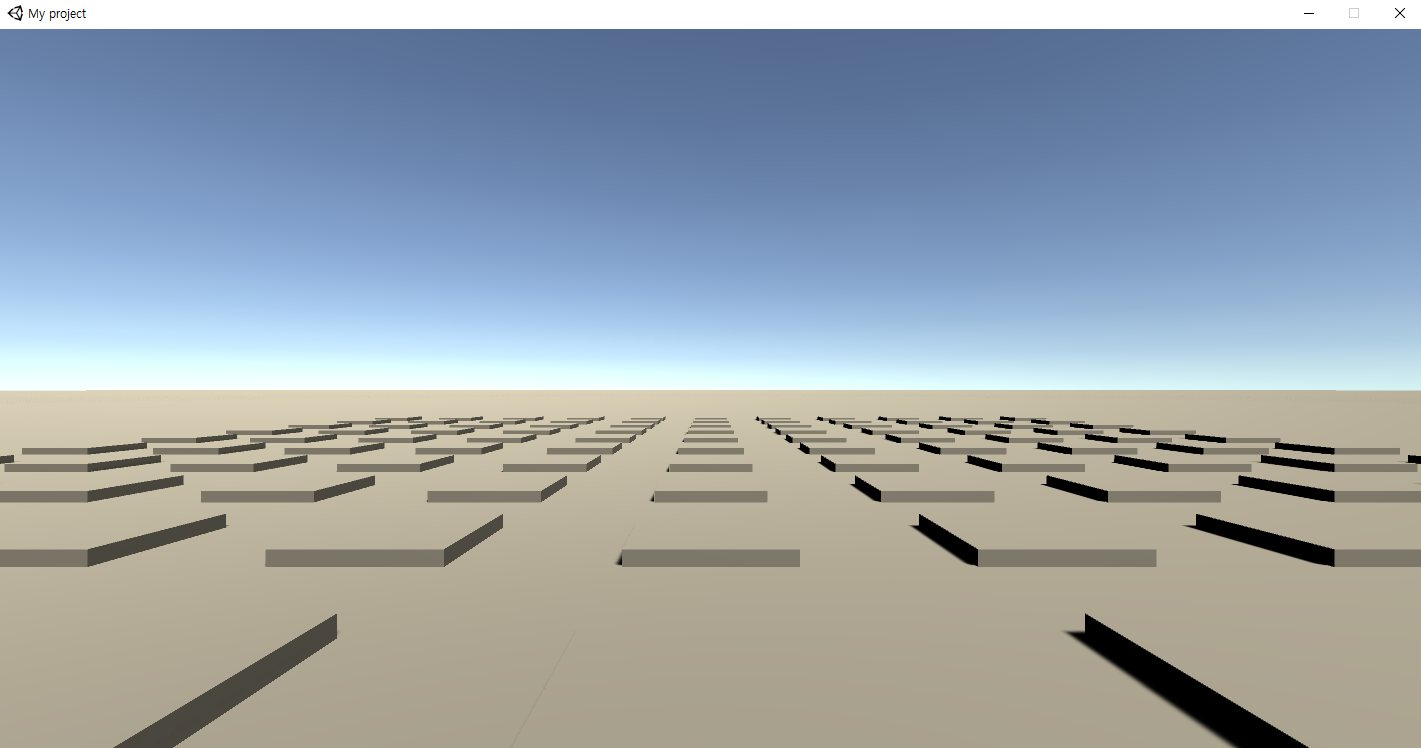
* 프레임마다 실행, 스케쥴러에 있는 메소드를 순차적으로 실행

1. **데모 시나리오**
2. 게임 로그인 진행
3. 게임 플레이
4. 게임 결과화면
5. 게임 종료
6. **프로그램 실행 결과 캡처 및 설명**

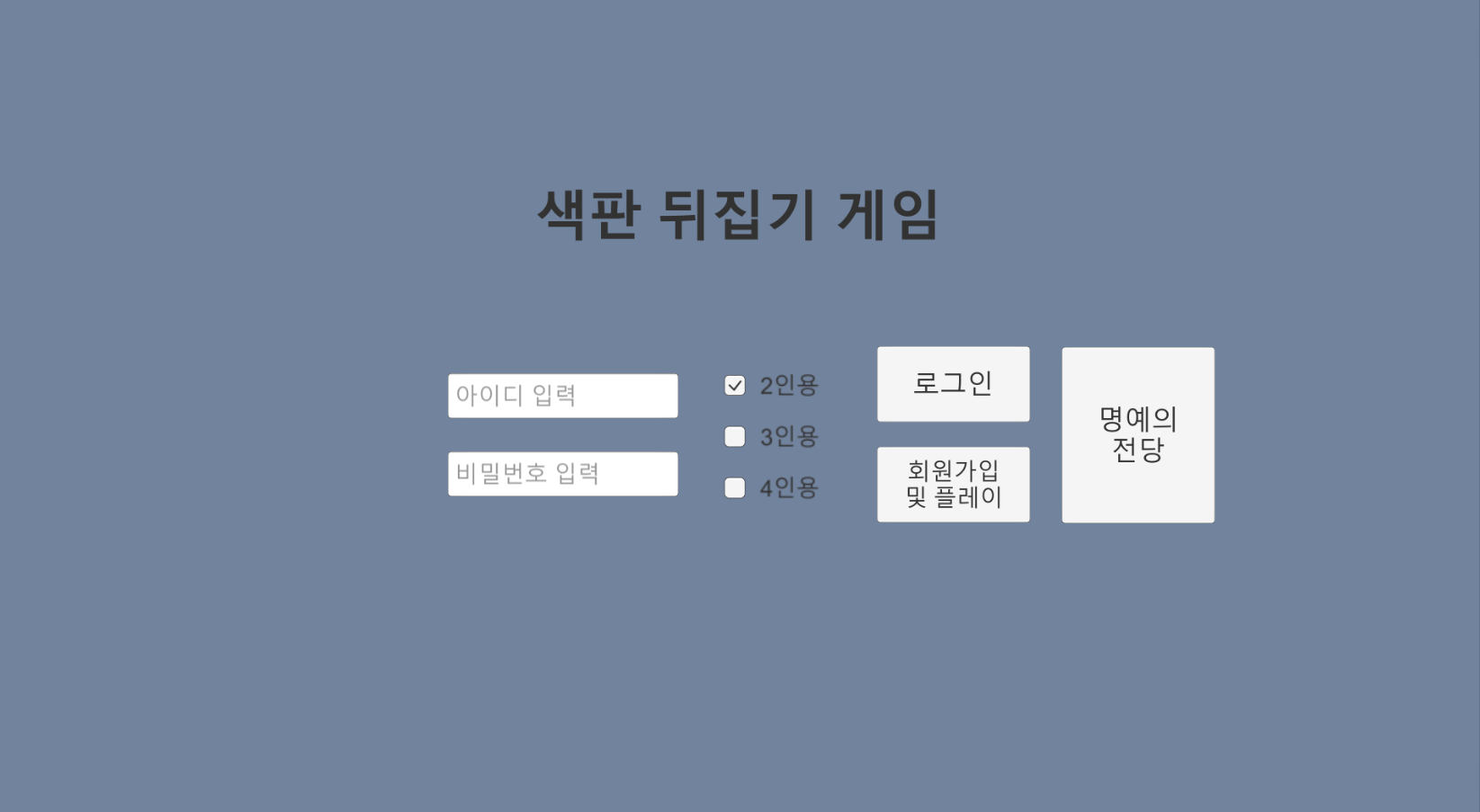
* **서버구동 및 클라이언트 게임 실행**

****

* **서버 및 클라이언트 실행파일**

****

* **서버 구동 모습**
* **메인화면**

****

* 아이디와 비밀번호를 입력한 후 로그인 버튼을 누르면 데이터베이스에 접근해 해당 아이디와 비밀번호가 매칭이 되는지 판단, 유효할 경우 게임 Scene으로 연결
* 회원가입 버튼을 누를 경우, 이미 존재하는 아이디인지를 판단, 존재하지 않을 경우 데이터베이스에 추가 및 게임 Scene으로 연결
* 명예의 전당 버튼은 기능을 구현하지 못함(기존 기능은 순위표 역할)
* **게임화면: 대기**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

* 클라이언트가 R키를 누르면 준비상태로 전환
* 모든 클라이언트가 준비상태라면 게임이 시작됨
* **게임화면: 플레이**

**텍스트, 전자기기, 컴퓨터이(가) 표시된 사진

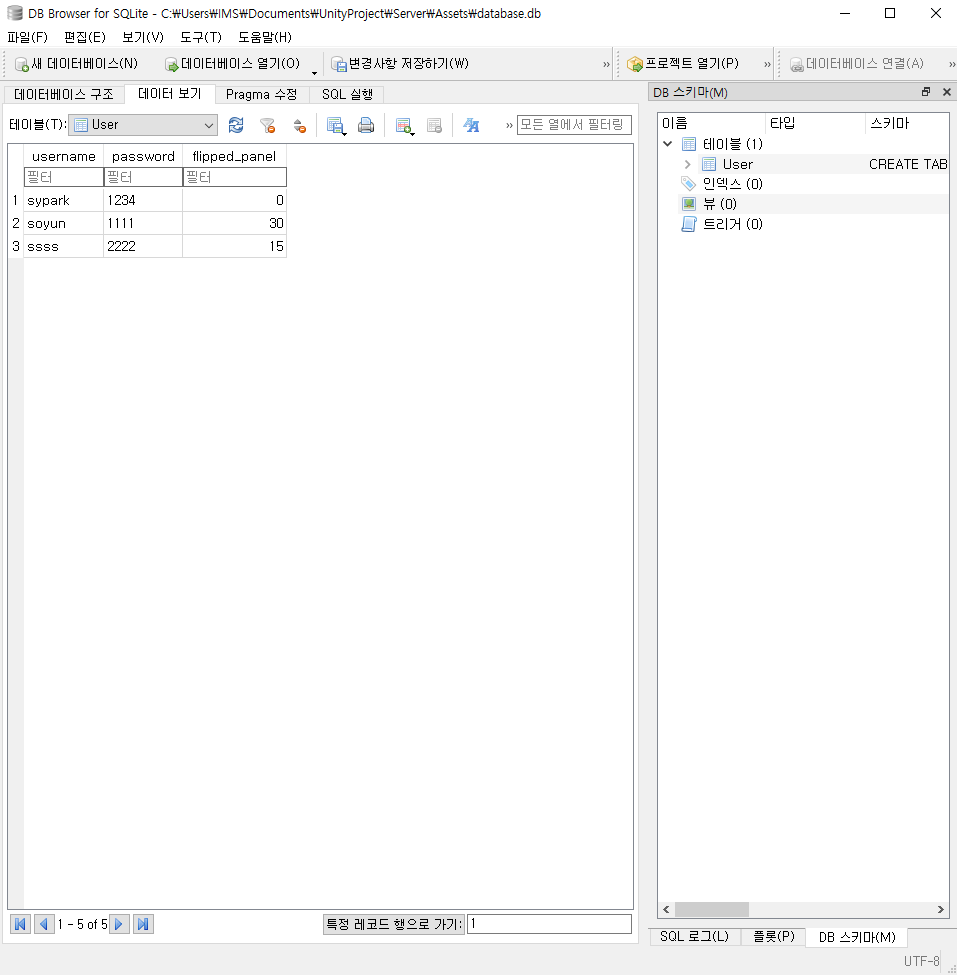
자동 생성된 설명**

* 클라이언트가 조준선에 타일을 맞춘 후 F를 누르면 플레이어에게 할당된 색으로 색판의 색이 바뀜
* 제한시간 안에 최대한 많은 색판을 뒤집는 게임
* **결과화면**

**텍스트, 명함, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

* 색판을 많이 뒤집은 순으로 정렬한 결과표를 보여줌
* 게임종료 버튼을 누르면 게임을 종료할 수 있음
* **데이터베이스 변화**

****

[전]

테이블이(가) 표시된 사진

자동 생성된 설명

[후]

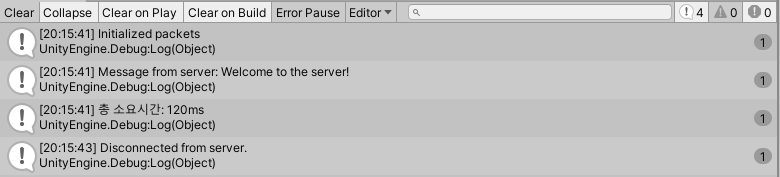
* username: 사용자 아이디
* password: 사용자 비밀번호
* flipped\_panel(사용하지 않음): 현재까지 뒤집은 색판 수 합계

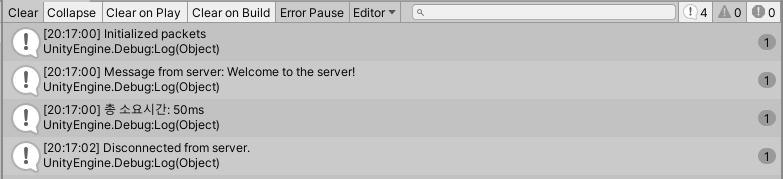
1. **현실적 제한 요소의 반영 내용 기술**

* 3명 이상의 클라이언트가 동시에 접속하여 게임 진행 가능(최대 4명)
* 병행처리를 위해 비동기 메소드를 사용하여 경쟁상태가 발생할 수 있음
* ThreadManager 클래스 구현, 경쟁상태를 야기할 수 있는 스크립트 부분을 ThreadManager의 스케쥴링 List에 넣어서 순차적으로 실행되도록 함
* 외부 ip를 가진 클라이언트도 게임을 플레이 할 수 있게 하고 싶었지만, 공용 공유기를 사용하고 있어 포트 포워딩이 불가하여 구현하지 못함(서버가 사용하고 있는 공유기에서 파생된 사설 ip를 사용하는 클라이언트만 접속 가능)

1. **응용 프로그램 성능 분석**

* **동기 메소드 vs 비동기 메소드**
  + 패킷에 데이터를 읽고 쓰는 함수를 동기, 비동기로 각각 구현하여 시간 측정
  + 로그인 버튼을 누른 직후부터 서버로부터 로그인 성공 패킷을 받을 때까지의 시간을 측정함
  + 결과





* + - 첫번째 사진은 동기 메소드를 사용, 120ms 소요
    - 두번째 사진은 비동기 메소드를 사용, 50ms 소요
    - 여러 번의 시도 결과, 평균적으로 비동기 메소드가 2배 이상 빨랐음

**[팀워크]**

1. 팀 작업 내용

* C# 네트워크 프로그래밍 관련 자료조사
* 유니티를 이용하여 메인화면 및 게임 Scene 구현
* 서버 구동 로직 작성 (참고자료의 영상 바탕으로 작성)
* 클라이언트 구동 로직 작성 (참고자료의 영상 바탕으로 작성)
* 패킷 메시지 정의 및 핸들러 구현
* 유니티와 데이터베이스 연동
* 게임 로직 구현

1. 팀원의 역할 및 작업 내용

* **박소윤**
* C# 네트워크 프로그래밍 관련 자료조사
* 유니티를 이용하여 메인화면 및 게임 Scene 구현
* 서버 구동 로직 작성 (참고자료의 영상 바탕으로 작성)
* 클라이언트 구동 로직 작성 (참고자료의 영상 바탕으로 작성)
* 패킷 메시지 정의 및 핸들러 구현
* 유니티와 데이터베이스 연동
* 게임 로직 구현

**[결론 및 총평]**

1. 네트워크 프로그래밍을 할 때 코드에 오류가 있다면 발견하기가 매우 어려움, 그러므로 효율적으로 디버깅을 할 줄 알아야 하고 코드를 꼼꼼하게 짜야 함
2. 네트워크 프로그래밍은 여러 클라이언트가 하나의 서버에 접속하여 데이터를 주고받으므로 경쟁상태를 잘 제어하는 것이 중요함
3. 서버와 클라이언트가 주고받는 패킷을 미리 정의해 두어야 하고 패킷을 수신했을 때 꼼꼼하게 핸들링 할 수 있어야 함
4. 소켓을 연결하는 것도 물론 중요하지만, 연결 해제할 때 관련된 객체나 변수들을 제대로 초기화해주어야 함
5. 네트워크 관련 메소드는 비동기 메소드를 쓰는 것이 더 효율적임

**[참고자료]**

1. C# Networking Tutorial 시리즈

<https://www.youtube.com/watch?v=uh8XaC0Y5MA&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=1>

<https://www.youtube.com/watch?v=4uHTSknGJaY&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=2>

<https://www.youtube.com/watch?v=QajkUJeypy4&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=3>

<https://www.youtube.com/watch?v=_h6Ta-vxAzQ&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=4>

<https://www.youtube.com/watch?v=Q3G_BBpbCek&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=5>

<https://www.youtube.com/watch?v=qkjr_rv4AIQ&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=6>

<https://www.youtube.com/watch?v=fnc2WKGV2eA&list=PLXkn83W0QkfnqsK8I0RAz5AbUxfg3bOQ5&index=>7

1. 유니티와 sqlite 연동하기 c#

<https://mungto.tistory.com/253>

1. tom-weiland/tcp-udp-networking

<https://github.com/tom-weiland/tcp-udp-networking/tree/master/GameClient/Assets/Scripts>

1. [Unity] 네트워크 프로그래밍

<https://euncero.tistory.com/653\>

**[부록: 프로그램 소스]**

**1. 서버 프로그램**

**- Server.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** System.Net;  **using** System.Net.Sockets;  **using** UnityEngine;    **public** **class** Server  {  **public** **static** **int** MaxPlayers { **get**; **set**; }  **public** **static** **int** Port { **get**; **private** **set**; }  **public** **static** Dictionary<**int**, Client> clients = new Dictionary<**int**, Client>(); *// 클라이언트 저장*  **public** **delegate** **void** PacketHandler(**int** fromClient, Packet packet);  **public** **static** Dictionary<**int**, PacketHandler> packetHandlers;    **private** **static** TcpListener tcpListener; *// TCP Port Listening 기능*  **private** **static** UdpClient udpListener; *// UDP Port Listening 기능*    **public** **static** **void** Start(**int** maxPlayers, **int** port)  {  MaxPlayers = maxPlayers; *// 서버 최대 인원*  Port = port; *// 포트 번호*    Debug.Log("Starting server...");  InitializeServerData(); *// 최대인원 수 만큼 클라이언트 관리 자료구조에 Client 객체 할당 및 핸들러 연결*    tcpListener = new TcpListener(IPAddress.Parse("192.168.1.18"), Port);  tcpListener.Start();  *// 클라이언트 접속 대기 -> 성공 시 콜백함수 호출(비동기)*  tcpListener.BeginAcceptTcpClient(new AsyncCallback(TCPConnectCallback), **null**);    udpListener = new UdpClient(Port);  *// UDP -> 메세지 대기(비동기)*  udpListener.BeginReceive(UDPReceiveCallback, **null**);    Debug.Log($"Server started on {Port}.");    *// DB 관련 메소드*  DBManager.DBCreate(); *// DB 생성*  DBManager.DBConnectionCheck(); *// DB 연결 상태 체크*  }    **public** **static** **void** Stop()  {  tcpListener.Stop();  udpListener.Close();  }    *// 클라이언트가 접속되었을 때 호출되는 콜백함수(from BeginAccepTcpClient)*  **private** **static** **void** TCPConnectCallback(IAsyncResult result)  {  TcpClient client = tcpListener.EndAcceptTcpClient(result);  tcpListener.BeginAcceptTcpClient(new AsyncCallback(TCPConnectCallback), **null**);  Debug.Log($"Incoming connection from {client.Client.RemoteEndPoint}...");  **for** (**int** i = 1; i <= MaxPlayers; i++)  {  *// client의 tcp가 연결되지 않았다면*  **if** (clients[i].tcp.socket == **null**)  {  clients[i].tcp.Connect(client); *// 연결해준다.*  **return**;  }    }  *// for loop이 완료될 때까지 계속 실행된다면, 서버가 꽉 찼다는 의미, 오류를 출력해줌*  Debug.Log($"{client.Client.RemoteEndPoint} failed to connect: Server full!");  }    */// <summary>송신받은 UDP 패킷을 읽는다</summary>*  **private** **static** **void** UDPReceiveCallback(IAsyncResult result)  {  **try**  {  IPEndPoint clientEndPoint = new IPEndPoint(IPAddress.Any, 0);  **byte**[] data = udpListener.EndReceive(result, **ref** clientEndPoint);  udpListener.BeginReceive(UDPReceiveCallback, **null**);    **if** (data.Length < 4)  {  **return**;  }    **using** (Packet packet = new Packet(data))  {  **int** clientId = packet.ReadInt();    **if** (clientId == 0)  {  **return**;  }    **if** (clients[clientId].udp.endPoint == **null**)  {  *// 새로운 연결이라면*  clients[clientId].udp.Connect(clientEndPoint); *// 서버-클라이언트 연결 시도*  **return**;  }    **if** (clients[clientId].udp.endPoint.ToString() == clientEndPoint.ToString())  {  *// 이미 연결된 상태라면*  clients[clientId].udp.HandleData(packet); *// 데이터를 주고 받음*  }  }  }  **catch** (Exception e)  {  Debug.Log($"Error receiving UDP data: {e}");  }  }    */// <summary>Sends a packet to the specified endpoint via UDP.</summary>*  */// <param name="clientEndPoint">The endpoint to send the packet to.</param>*  */// <param name="packet">The packet to send.</param>*  **public** **static** **void** SendUDPData(IPEndPoint clientEndPoint, Packet packet)  {  **try**  {  **if** (clientEndPoint != **null**)  {  udpListener.BeginSend(packet.ToArray(), packet.Length(), clientEndPoint, **null**, **null**);  }  }  **catch** (Exception e)  {  Debug.Log($"Error sending data to {clientEndPoint} via UDP: {e}");  }  }    **private** **static** **void** InitializeServerData()  {  *// clients 딕셔너리 채움, id는 1부터 순차적으로 부여*  **for** (**int** i = 1; i <= MaxPlayers; i++)  {  clients.**Add**(i, new Client(i));  }    packetHandlers = new Dictionary<**int**, PacketHandler>()  {  { (**int**)ClientPackets.welcomeReceived, ServerHandle.WelcomeReceived },  { (**int**)ClientPackets.playerEnter, ServerHandle.PlayerEnteredRoom},  { (**int**)ClientPackets.playerReady, ServerHandle.PlayerReady },  { (**int**)ClientPackets.playerMovement, ServerHandle.PlayerMovement },  { (**int**)ClientPackets.playerFlip, ServerHandle.PlayerFilp },  };  Debug.Log("Initialized packets.");  }  } |

**- Client.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** System.Net;  **using** System.Net.Sockets;  **using** UnityEngine;    **public** **class** Client  {  **public** **static** **int** dataBufferSize = 4096;  **public** Player player;  **public** **int** id;  **public** TCP tcp;  **public** UDP udp;    **public** Client(**int** clientID)  {  id = clientID;  tcp = new TCP(id);  udp = new UDP(id);  }    */// <summary>새로운 플레이어와 다른 플레이어 간의 연결/summary>*  */// <param name="playerName"></param>*  **public** **void** SendIntoGame(**string** playerName, **string** password)  {  player = NetworkManager.instance.InstantiatePlayer();  player.Initialize(id, playerName, password);    *// 플레이어 id에 따른 생성 위치*  **switch** (player.id)  {  **case** 1:  player.transform.position = new Vector3(-15f, 0.5f, 0);  player.transform.rotation = Quaternion.EulerAngles(0f, 90f, 0f);  **break**;  **case** 2:  player.transform.position = new Vector3(15f, 0.5f, 0);  player.transform.rotation = Quaternion.EulerAngles(0f, -90f, 0f);  **break**;  **case** 3:  player.transform.position = new Vector3(-15f, 0.5f, 15f);  player.transform.rotation = Quaternion.EulerAngles(0f, 180f, 0f);  **break**;  **case** 4:  player.transform.position = new Vector3(15f, 0.5f, -15f);  player.transform.rotation = Quaternion.EulerAngles(0f, -180f, 0f);  **break**;    }    **foreach** (Client client **in** Server.clients.Values)  {  **if** (!(client.player == **null**))  {  Debug.Log(client.player.username);  }  }    *// 새로운 플레이어와 다른 플레이어 연결*  **foreach** (Client client **in** Server.clients.Values)  {  **if** (client.player != **null**)  {  **if** (client.id != id)  {  Debug.Log("다른 플레이어 생성!");  Debug.Log($"{client.id} / {client.player.username}");  ServerSend.SpawnPlayer(id, client.player);  }  }  }    *// 다른 플레이어와 새로운 플레이어 연결*  **foreach** (Client client **in** Server.clients.Values)  {  **if** (client.player != **null**)  {  Debug.Log("내 플레이어 생성!");  ServerSend.SpawnPlayer(client.id, player);  }  }  }      *// 클라이언트가 연결 해제*  **private** **void** Disconnect()  {  Debug.Log($"{tcp.socket.Client.RemoteEndPoint} has disconnected");    *// 플레이어가 방에 들어와있는 상태에서 접속을 끊었을 때*  **if** (player != **null**)  {  ThreadManager.ExecuteOnMainThread(() =>  {  UnityEngine.**Object**.Destroy(player.gameObject);  player = **null**;  });    NetworkManager.instance.connectedPlayers.**Remove**(player.username);  NetworkManager.instance.playerCount--; *// 접속 중인 플레이어 수 감소*    *// 플레이어가 Ready 상태였다면*  **if** (player.isReady)  {  NetworkManager.instance.readyCount--;  }    *// 서버에 아무 플레이어도 남아있지 않다면 -> maxPlayer값 초기화*  **if** (NetworkManager.instance.playerCount == 0)  {  NetworkManager.instance.maxPlayer = 1;  NetworkManager.instance.isGameStarted = **false**;  NetworkManager.instance.connectedPlayers.Clear();    **foreach** (ColorPanel panel **in** ColorPanelSpawner.instance.colorPanels)  {  panel.clientID = 0;  }  }  }    tcp.Disconnect();  udp.Disconnect();    ServerSend.PlayerDisconnected(id);  }    */// <summary>클라이언트 UDP</summary>*  **public** **class** TCP  {  **public** TcpClient socket;  **private** **readonly** **int** id;  **private** NetworkStream stream;  **private** Packet receivedData;  **private** **byte**[] receiveBuffer;    **public** TCP(**int** \_id)  {  id = \_id;  }    **public** **void** Connect(TcpClient \_socket)  {  socket = \_socket;  *// 읽어오는 버퍼와 받아오는 버퍼의 크기를 지정해줌(4096바이트)*    socket.ReceiveBufferSize = dataBufferSize;  socket.SendBufferSize = dataBufferSize;    *// GetStream(): TCP 네트워크 스트림 리턴, 네트워크 스트림을 이용해 데이터 송수신*  stream = socket.GetStream();    receivedData = new Packet(); *// receivedData에는 패킷을 저장*  *// 패킷을 데이터를 받아오는 버퍼에 byte 배열을 해당 크기만큼 할당해줌*  receiveBuffer = new **byte**[dataBufferSize];    *// 읽기 시작*  *// params: 읽어 온 것을 저장할 버퍼, 오프셋, 버퍼 사이즈, 콜백함수, 상태*  stream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, **null**);  *//stream.Read(receiveBuffer, 0, dataBufferSize);*    ServerSend.Welcome(id, "Welcome to the server!");  }    **public** **void** SendData(Packet packet)  {  **try**  {  **if** (socket != **null**)  {  *//stream.Write(packet.ToArray(), 0, packet.Length());*  stream.BeginWrite(packet.ToArray(), 0, packet.Length(), **null**, **null**);  }  }  **catch** (Exception e)  {  Debug.Log($"플레이어 {id}에게 데이터를 보내는 도중 오류가 발생했습니다. : {e}");  }  }    **private** **void** ReceiveCallback(IAsyncResult result)  {  **try**  {  **int** byteLength = stream.EndRead(result);  **if** (byteLength <= 0)  {  Server.clients[id].Disconnect();  **return**;  }    **byte**[] data = new **byte**[byteLength];  Array.Copy(receiveBuffer, data, byteLength); *// receiveBuffer의 값을 data에 복사*    receivedData.Reset(HandleData(data));    stream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, **null**); *// 다시 데이터 들어오기를 기다림*  }  **catch** (Exception e)  {  Console.Write($"Error receiving TCP data: {e}");  Server.clients[id].Disconnect();  }  }    */// <summary>연속적인 데이터 여부 판별</summary>*  */// <param name="data"></param>*  */// <returns>해당 패킷이 데이터의 마지막인지 여부</returns>*  **private** **bool** HandleData(**byte**[] data)  {  **int** packetLength = 0;    receivedData.SetBytes(data);    *// 패킷의 처음 데이터는 패킷의 데이터 길이(int형 4바이트)*  *// int형을 읽었을 때 packetLength가 0이하가 된다면, 마지막 데이터임 -> true*  **if** (receivedData.UnreadLength() >= 4)  {  packetLength = receivedData.ReadInt();  **if** (packetLength <= 0)  {  **return** **true**;  }  }    **while** (packetLength > 0 && packetLength <= receivedData.UnreadLength())  {  **byte**[] packetBytes = receivedData.ReadBytes(packetLength);  ThreadManager.ExecuteOnMainThread(() =>  {  **using** (Packet packet = new Packet(packetBytes))  {  **int** packetID = packet.ReadInt();  Server.packetHandlers[packetID](id, packet);  }  });    packetLength = 0;    **if** (receivedData.UnreadLength() >= 4)  {  packetLength = receivedData.ReadInt();  **if** (packetLength <= 0)  {  **return** **true**;  }  }  }    **if** (packetLength <= 1)  {  **return** **true**;  }  **return** **false**;  }    **public** **void** Disconnect()  {  socket.Close();  stream = **null**;  receivedData = **null**;  receiveBuffer = **null**;  socket = **null**;  }  }    **public** **class** UDP  {  **public** IPEndPoint endPoint;    **private** **int** id;    **public** UDP(**int** \_id)  {  id = \_id;  }    */// <summary>서버 UDP</summary>*  */// <param name="\_endPoint">서버의 IPEndPoint -> 클라이언트와 연결</param>*  **public** **void** Connect(IPEndPoint \_endPoint)  {  endPoint = \_endPoint;  }    */// <summary>UDP를 이용한 데이터 전송</summary>*  */// <param name="packet">The packet to send.</param>*  **public** **void** SendData(Packet packet)  {  Server.SendUDPData(endPoint, packet);  }    */// <summary>Prepares received data to be used by the appropriate packet handler methods.</summary>*  */// <param name="packetData">The packet containing the recieved data.</param>*  **public** **void** HandleData(Packet packetData)  {  **int** packetLength = packetData.ReadInt();  **byte**[] packetBytes = packetData.ReadBytes(packetLength);    ThreadManager.ExecuteOnMainThread(() =>  {  **using** (Packet packet = new Packet(packetBytes))  {  **int** packetId = packet.ReadInt();  Server.packetHandlers[packetId](id, packet); *// Call appropriate method to handle the packet*  }  });  }    */// <summary>Cleans up the UDP connection.</summary>*  **public** **void** Disconnect()  {  endPoint = **null**;  }  }  } |

**- ServerSend.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ServerSend  {  */// <summary>특정 클라이언트에게 데이터 보내기</summary>*  */// <param name="toClient">패킷을 보낼 클라이언트 ID</param>*  */// <param name="packet">패킷</param>*  **private** **static** **void** SendTCPData(**int** toClient, Packet packet)  {  packet.WriteLength();  Server.clients[toClient].tcp.SendData(packet);  }    */// <summary>Sends a packet to a client via UDP.</summary>*  */// <param name="\_toClient">The client to send the packet the packet to.</param>*  */// <param name="\_packet">The packet to send to the client.</param>*  **private** **static** **void** SendUDPData(**int** toClient, Packet packet)  {  packet.WriteLength();  Server.clients[toClient].udp.SendData(packet);  }      */// <summary>모든 클라이언트에게 데이터 보내기</summary>*  */// <param name="packet">패킷</param>*  **private** **static** **void** SendTCPDataToAll(Packet packet)  {  packet.WriteLength();  **for** (**int** i = 1; i <= Server.MaxPlayers; i++)  {  Server.clients[i].tcp.SendData(packet);  }  }    */// <summary>특정 클라이언트를 제외하고 모두에게 데이터 보내기</summary>*  */// <param name="exceptClient">제외할 클라이언트 ID</param>*  */// <param name="packet">패킷</param>*  **private** **static** **void** SendTCPDataToAll(**int** exceptClient, Packet packet)  {  packet.WriteLength();  **for** (**int** i = 1; i <= Server.MaxPlayers; i++)  {  **if** (i != exceptClient)  {  Server.clients[i].tcp.SendData(packet);  }  }  }    */// <summary>Sends a packet to all clients via UDP.</summary>*  */// <param name="packet">The packet to send.</param>*  **private** **static** **void** SendUDPDataToAll(Packet packet)  {  packet.WriteLength();  **for** (**int** i = 1; i <= Server.MaxPlayers; i++)  {  Server.clients[i].udp.SendData(packet);  }  }  */// <summary>Sends a packet to all clients except one via UDP.</summary>*  */// <param name="exceptClient">The client to NOT send the data to.</param>*  */// <param name="packet">The packet to send.</param>*  **private** **static** **void** SendUDPDataToAll(**int** exceptClient, Packet packet)  {  packet.WriteLength();  **for** (**int** i = 1; i <= Server.MaxPlayers; i++)  {  **if** (i != exceptClient)  {  Server.clients[i].udp.SendData(packet);  }  }  }    #region Packets    */// <summary>서버->클라이언트 Welcome 패킷 생성</summary>*  */// <param name="toClient">패킷을 보낼 클라이언트 ID</param>*  */// <param name="msg">클라이언트에게 보낼 Welcome 메세지</param>*  **public** **static** **void** Welcome(**int** toClient, **string** msg)  {  *// 패킷을 하나 생성*  **using** (Packet packet = new Packet((**int**)ServerPackets.welcome))  {  *// 패킷에 데이터 추가*  packet.Write(msg);  packet.Write(toClient);    SendTCPData(toClient, packet);  }  }    **public** **static** **void** OnLoginOrRegisterSuccess(**int** toClient)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.loginOrRegisterSuccess))  {  packet.Write(toClient);    SendTCPData(toClient, packet);  }  }    **public** **static** **void** PlayerReady() {  **using** (Packet packet = new Packet((**int**)ServerPackets.playerReady))  {  packet.Write(NetworkManager.instance.playerCount);    SendTCPDataToAll(packet);  }  }      **public** **static** **void** SpawnPlayer(**int** toClient, Player player)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.spawnPlayer))  {  packet.Write(player.id);  packet.Write(player.username);  packet.Write(player.transform.position);  packet.Write(player.transform.rotation);    SendTCPData(toClient, packet);  }  }    **public** **static** **void** PlayerDisconnected(**int** playerID)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.playerDisconnected))  {  packet.Write(playerID);  SendTCPDataToAll(packet);  }  }    **public** **static** **void** GameStart()  {  **using** (Packet packet = new Packet((**int**)ServerPackets.gameStart))  {  Debug.Log("GameStart 패킷 보냄");  packet.Write(**true**);  SendTCPDataToAll(packet);  }  }    **public** **static** **void** PlayerPosition(Player player)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.playerPosition))  {  packet.Write(player.id);  packet.Write(player.transform.position);    SendUDPDataToAll(packet);  }  }    **public** **static** **void** PlayerRotation(Player player)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.playerRotation))  {  packet.Write(player.id);  packet.Write(player.transform.rotation);    SendUDPDataToAll(player.id, packet);  }  }    **public** **static** **void** PlayerFlipped(**int** panelID, **int** clientID)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.playerFlipped))  {  packet.Write(panelID);  packet.Write(clientID);    SendTCPDataToAll(packet);  }  }    **public** **static** **void** Error(**int** toClient, **string** msg)  {  **using** (Packet packet = new Packet((**int**)ServerPackets.error))  {  packet.Write(msg);  SendTCPData(toClient, packet);  }  }  #endregion  } |

**- ServerHandle.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ServerHandle  {  **public** **static** **void** WelcomeReceived(**int** fromClient, Packet packet)  {  **int** clientIdCheck = packet.ReadInt();  **string** username = packet.ReadString();  **string** password = packet.ReadString();  **bool** isRegister = packet.ReadBool();  Debug.Log($"username: {username} / password: {password}");    Debug.Log($"{Server.clients[fromClient].tcp.socket.Client.RemoteEndPoint} connected succesfully and is now player {fromClient}");  *// Double Check -> 매개변수로 받은 클라이언트의 ID와 패킷 데이터에 저장된 클라이언트의 ID가 같은 지*  **if** (fromClient != clientIdCheck)  {  Debug.Log($"Player **\"**{username}**\"** (Id: {fromClient}) has assumed the wrong client ID ({clientIdCheck})!");  }    *// 회원가입 시도라면*  **if** (isRegister)  {  *// 회원가입 정보가 유효한지*  *// 중복되는 아이디인지*  **if** (DBManager.DataBaseRead($"SELECT \* FROM user WHERE username='{username}'").Count != 0)  {  ServerSend.Error(fromClient, "Duplicated ID");  **return**;  }    DBManager.DataBaseChange($"INSERT INTO user(username, password, flipped\_panel) VALUES('{username}', '{password}', '{0}')");  }  *// 로그인 시도라면*  **else**  {  *// 로그인 정보가 유효한지 검사*  *// 존재하는 아이디인지*  **if** (DBManager.DataBaseRead($"SELECT \* FROM user WHERE username='{username}'").Count == 0)  {  ServerSend.Error(fromClient, "Wrong ID");  **return**;  }  *// 아이디와 패스워드가 매칭되는지*  **if** (DBManager.DataBaseRead($"SELECT \* FROM user WHERE username='{username}' AND password='{password}'").Count == 0)  {  ServerSend.Error(fromClient, "Wrong Password");  **return**;  }  *// 이미 접속해있는 플레이어인지*  **if** (NetworkManager.instance.connectedPlayers.Contains(username))  {  ServerSend.Error(fromClient, "This player is already in server");  **return**;  }  }    ServerSend.OnLoginOrRegisterSuccess(fromClient); *// 로그인 정보가 유효하다면 로그인 성공 패킷 보냄*  }    **public** **static** **void** PlayerEnteredRoom(**int** fromClient, Packet packet)  {  **string** username = packet.ReadString();  **string** password = packet.ReadString();  **int** maxPlayer = packet.ReadInt();    Server.clients[fromClient].SendIntoGame(username, password); *// 클라이언트(플레이어)를 게임에 연결*  NetworkManager.instance.connectedPlayers.**Add**(username);  NetworkManager.instance.playerCount++;  *// 인원 수가 초기값이라면 maxPlayer값을 받아 넣어 줌*  **if** (NetworkManager.instance.maxPlayer == 1)  {  NetworkManager.instance.maxPlayer = maxPlayer;  }  }    **public** **static** **void** PlayerReady(**int** fromClient, Packet packet)  {  **bool** isReady = packet.ReadBool();  NetworkManager.instance.readyCount++;  Server.clients[fromClient].player.isReady = **true**; *// 플레이어를 레디 상태로 전환*    ServerSend.PlayerReady();  }    **public** **static** **void** PlayerMovement(**int** fromClient, Packet packet)  {  **bool**[] inputs = new **bool**[packet.ReadInt()];  **for** (**int** i = 0; i < inputs.Length; i++)  {  inputs[i] = packet.ReadBool();  }  Quaternion rotation = packet.ReadQuaternion();    Server.clients[fromClient].player.SetInput(inputs, rotation);  }    **public** **static** **void** PlayerFilp(**int** fromClient, Packet packet)  {  **int** panelID = packet.ReadInt();    **foreach** (ColorPanel panel **in** ColorPanelSpawner.instance.colorPanels)  {  **if** (panel.panelID.Equals(panelID))  {  panel.clientID = fromClient;  }  }    ServerSend.PlayerFlipped(panelID, fromClient);  }  } |

**- Packet.cs**

|  |
| --- |
| **using** System.Collections;  **using** System;  **using** System.Collections.Generic;  **using** System.Text;  **using** UnityEngine;    */// <summary>서버 -> 클라이언트</summary>*  **public** **enum** ServerPackets  {  welcome = 1,  loginOrRegisterSuccess,  spawnPlayer,  playerReady,  gameStart,  playerPosition,  playerRotation,  playerFlipped,  playerDisconnected,  error,  }    */// <summary>클라이언트 -> 서버</summary>*  **public** **enum** ClientPackets  {  welcomeReceived = 1,  playerEnter,  playerReady,  playerMovement,  playerFlip,  }    *// 패킷 클래스, IDisposable: 열린 파일, 스트림과 같이 GC가 제대로 처리해주지 않는 리소스 정리를 위한 인터페이스, Dispose() 구현 필요*  **public** **class** Packet : IDisposable  {  **private** List<**byte**> buffer;  **private** **byte**[] readableBuffer;  **private** **int** readPos;    */// <summary>비어있는 패킷 생성</summary>*  **public** Packet()  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*  }    */// <summary>ID를 가지고 있는 패킷 생성, 패킷 보낼 때</summary>*  */// <param name="\_id">패킷 ID</param>*  **public** Packet(**int** \_id)  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*    Write(\_id); *// Write packet id to the buffer*  }    */// <summary>데이터를 가지고 있는 패킷 생성, 패킷 받을 때</summary>*  */// <param name="\_data">패킷에 추가할 byte 형식의 데이터</param>*  **public** Packet(**byte**[] \_data)  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*    SetBytes(\_data);  }    #region Functions  */// <summary>패킷에 데이터 추가</summary>*  */// <param name="\_data">패킷에 추가할 byte 형식의 데이터</param>*  **public** **void** SetBytes(**byte**[] \_data)  {  Write(\_data);  readableBuffer = buffer.ToArray();  }    */// <summary>바이트 배열인 패킷 데이터의 길이를 패킷 버퍼의 시작지점에 삽입</summary>*  **public** **void** WriteLength()  {  buffer.InsertRange(0, BitConverter.GetBytes(buffer.Count)); *// Insert the byte length of the packet at the very beginning*  }    */// <summary>주어진 int형의 값을 패킷 버퍼의 시작지점에 삽입</summary>*  */// <param name="\_value">The int to insert.</param>*  **public** **void** InsertInt(**int** \_value)  {  buffer.InsertRange(0, BitConverter.GetBytes(\_value)); *// Insert the int at the start of the buffer*  }    */// <summary>패킷 데이터를 배열형식으로 리턴/summary>*  **public** **byte**[] ToArray()  {  readableBuffer = buffer.ToArray();  **return** readableBuffer;  }    */// <summary>패킷 데이터의 길이 리턴</summary>*  **public** **int** Length()  {  **return** buffer.Count; *// Return the length of buffer*  }    */// <summary>패킷 데이터 중 읽지 않은 부분의 길이 리턴</summary>*  **public** **int** UnreadLength()  {  **return** Length() - readPos; *// Return the remaining length (unread)*  }    */// <summary>패킷을 재사용할 수 있도록 리셋</summary>*  */// <param name="\_shouldReset">패킷 리셋 여부</param>*  **public** **void** Reset(**bool** \_shouldReset = **true**)  {  **if** (\_shouldReset)  {  buffer.Clear(); *// Clear buffer*  readableBuffer = **null**;  readPos = 0; *// Reset readPos*  }  **else**  {  readPos -= 4; *// "Unread" the last read int -> 마지막 4바이트를 안 읽은 걸로*  }  }  #endregion    *// Write할 다양한 데이터 타입을 byte로 변환*  #region Write Data  */// <summary>Adds a byte to the packet.</summary>*  */// <param name="\_value">The byte to add.</param>*  **public** **void** Write(**byte** \_value)  {  buffer.**Add**(\_value);  }  */// <summary>Adds an array of bytes to the packet.</summary>*  */// <param name="\_value">The byte array to add.</param>*  **public** **void** Write(**byte**[] \_value)  {  buffer.AddRange(\_value);  }  */// <summary>Adds a short to the packet.</summary>*  */// <param name="\_value">The short to add.</param>*  **public** **void** Write(**short** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds an int to the packet.</summary>*  */// <param name="\_value">The int to add.</param>*  **public** **void** Write(**int** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a long to the packet.</summary>*  */// <param name="\_value">The long to add.</param>*  **public** **void** Write(**long** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a float to the packet.</summary>*  */// <param name="\_value">The float to add.</param>*  **public** **void** Write(**float** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a bool to the packet.</summary>*  */// <param name="\_value">The bool to add.</param>*  **public** **void** Write(**bool** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a string to the packet.</summary>*  */// <param name="\_value">The string to add.</param>*  **public** **void** Write(**string** \_value)  {  Write(\_value.Length); *// Add the length of the string to the packet*  buffer.AddRange(Encoding.ASCII.GetBytes(\_value)); *// Add the string itself*  }  */// <summary>Adds a Vector3 to the packet.</summary>*  */// <param name="\_value">The Vector3 to add.</param>*  **public** **void** Write(Vector3 \_value)  {  Write(\_value.x);  Write(\_value.y);  Write(\_value.z);  }  */// <summary>Adds a Quaternion to the packet.</summary>*  */// <param name="\_value">The Quaternion to add.</param>*  **public** **void** Write(Quaternion \_value)  {  Write(\_value.x);  Write(\_value.y);  Write(\_value.z);  Write(\_value.w);  }  #endregion    *// Read한 byte를 다양한 데이터 타입으로 변환*  #region Read Data  */// <summary>Reads a byte from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **byte** ReadByte(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **byte** \_value = readableBuffer[readPos]; *// Get the byte at readPos' position*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 1; *// Increase readPos by 1*  }  **return** \_value; *// Return the byte*  }  **else**  {  **throw** new Exception("Could not read value of type 'byte'!");  }  }    */// <summary>Reads an array of bytes from the packet.</summary>*  */// <param name="\_length">The length of the byte array.</param>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **byte**[] ReadBytes(**int** \_length, **bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **byte**[] \_value = buffer.GetRange(readPos, \_length).ToArray(); *// Get the bytes at readPos' position with a range of \_length*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += \_length; *// Increase readPos by \_length*  }  **return** \_value; *// Return the bytes*  }  **else**  {  **throw** new Exception("Could not read value of type 'byte[]'!");  }  }    */// <summary>Reads a short from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **short** ReadShort(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **short** \_value = BitConverter.ToInt16(readableBuffer, readPos); *// Convert the bytes to a short*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true and there are unread bytes*  readPos += 2; *// Increase readPos by 2*  }  **return** \_value; *// Return the short*  }  **else**  {  **throw** new Exception("Could not read value of type 'short'!");  }  }    */// <summary>Reads an int from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **int** ReadInt(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **int** \_value = BitConverter.ToInt32(readableBuffer, readPos); *// Convert the bytes to an int*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 4; *// Increase readPos by 4*  }  **return** \_value; *// Return the int*  }  **else**  {  **throw** new Exception("Could not read value of type 'int'!");  }  }    */// <summary>Reads a long from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **long** ReadLong(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **long** \_value = BitConverter.ToInt64(readableBuffer, readPos); *// Convert the bytes to a long*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 8; *// Increase readPos by 8*  }  **return** \_value; *// Return the long*  }  **else**  {  **throw** new Exception("Could not read value of type 'long'!");  }  }    */// <summary>Reads a float from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **float** ReadFloat(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **float** \_value = BitConverter.ToSingle(readableBuffer, readPos); *// Convert the bytes to a float*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 4; *// Increase readPos by 4*  }  **return** \_value; *// Return the float*  }  **else**  {  **throw** new Exception("Could not read value of type 'float'!");  }  }    */// <summary>Reads a bool from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **bool** ReadBool(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **bool** \_value = BitConverter.ToBoolean(readableBuffer, readPos); *// Convert the bytes to a bool*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 1; *// Increase readPos by 1*  }  **return** \_value; *// Return the bool*  }  **else**  {  **throw** new Exception("Could not read value of type 'bool'!");  }  }    */// <summary>Reads a string from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **string** ReadString(**bool** \_moveReadPos = **true**)  {  **try**  {  **int** \_length = ReadInt(); *// Get the length of the string*  **string** \_value = Encoding.ASCII.GetString(readableBuffer, readPos, \_length); *// Convert the bytes to a string*  **if** (\_moveReadPos && \_value.Length > 0)  {  *// If \_moveReadPos is true string is not empty*  readPos += \_length; *// Increase readPos by the length of the string*  }  **return** \_value; *// Return the string*  }  **catch**  {  **throw** new Exception("Could not read value of type 'string'!");  }  }    */// <summary>Reads a Vector3 from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** Vector3 ReadVector3(**bool** \_moveReadPos = **true**)  {  **return** new Vector3(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));  }    */// <summary>Reads a Quaternion from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** Quaternion ReadQuaternion(**bool** \_moveReadPos = **true**)  {  **return** new Quaternion(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));  }  #endregion    **private** **bool** disposed = **false**;    **protected** **virtual** **void** Dispose(**bool** \_disposing)  {  **if** (!disposed)  {  **if** (\_disposing)  {  buffer = **null**;  readableBuffer = **null**;  readPos = 0;  }    disposed = **true**;  }  }    **public** **void** Dispose()  {  Dispose(**true**);  GC.SuppressFinalize(**this**);  }  } |

**- Player.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** Player : MonoBehaviour  {  **public** **int** id;  **public** **string** username;  **public** **string** password;  **public** **bool** isReady;  **public** CharacterController controller;  **public** **float** gravity = -9.81f;  **public** **float** moveSpeed = 5f;  **public** **float** jumpSpeed = 5f;    **private** **bool**[] inputs; *// 클라이언트가 보낸 입력을 받을 배열*  **private** **float** yVelocity = 0f;    **private** **void** Start()  {  gravity \*= Time.fixedDeltaTime \* Time.fixedDeltaTime;  moveSpeed \*= Time.fixedDeltaTime;  jumpSpeed \*= Time.fixedDeltaTime;  }    **public** **void** Initialize(**int** id, **string** username, **string** password)  {  **this**.id = id;  **this**.username = username;  **this**.password = password;  **this**.isReady = **false**;    inputs = new **bool**[5];  }    **public** **void** FixedUpdate()  {  Vector2 inputDirection = Vector2.zero;  **if** (inputs[0])  {  inputDirection.y += 1;  }  **if** (inputs[1])  {  inputDirection.y -= 1;  }  **if** (inputs[2])  {  inputDirection.x -= 1;  }  **if** (inputs[3])  {  inputDirection.x += 1;  }  Move(inputDirection);  }    *// 클라이언트 움직임 동기화*  **private** **void** Move(Vector2 inputDirection)  {  Vector3 moveDirection = transform.right \* inputDirection.x + transform.forward \* inputDirection.y;  moveDirection \*= moveSpeed;    *// 땅에 닿아있을 때에만 점프 가능*  **if** (controller.isGrounded)  {  yVelocity = 0f;  *// 점프*  **if** (inputs[4])  {  yVelocity = jumpSpeed;  }  }  yVelocity += gravity;  moveDirection.y = yVelocity;    controller.Move(moveDirection);    *// 플레이어의 position, rotation을 담은 패킷을 보냄 -> 모든 클라이언트끼리 동기화*  ServerSend.PlayerPosition(**this**);  ServerSend.PlayerRotation(**this**);  }    **public** **void** SetInput(**bool**[] inputs, Quaternion rotation)  {  **this**.inputs = inputs;  transform.rotation = rotation;  }  } |

**- NetworkManager.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** NetworkManager : MonoBehaviour  {  **public** **static** NetworkManager instance;    **public** GameObject playerPrefab;  **public** List<**string**> connectedPlayers;  **public** **bool** isGameStarted;  **public** **int** maxPlayer;  **public** **int** playerCount;  **public** **int** readyCount;    **private** **void** Awake()  {  **if** (instance == **null**)  {  instance = **this**;  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {  QualitySettings.vSyncCount = 0; *// 동기화를 위한 프레임 드랍 방지*  Application.targetFrameRate = 60; *// Server Logic이 초당 30 ticks 반복되므로 서버의 FrameRate를 제한 -> CPU 사용량 저하 목적*    isGameStarted = **false**;  maxPlayer = 1; *// 초기값*  readyCount = 0;    Server.Start(10, 9330);  }    **private** **void** FixedUpdate()  {  **if** (!isGameStarted && readyCount == maxPlayer)  {  isGameStarted = **true**;  ServerSend.GameStart();  }  }    **private** **void** OnApplicationQuit()  {  Server.Stop();  }    **public** Player InstantiatePlayer()  {  **return** Instantiate(playerPrefab, new Vector3(0f, 0.5f, 0f), Quaternion.identity).GetComponent<Player>();  }  } |

**- DBManager.cs**

|  |
| --- |
| **using** System;  **using** System.Data;  **using** Mono.Data.SqliteClient;  **using** System.IO;  **using** UnityEngine.Networking;  **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** DBManager  {  *// DB 생성*  **public** **static** **void** DBCreate()  {  **string** filePath = **string**.Empty; *// 파일경로*  filePath = Application.dataPath + "/database.db";  **if** (!File.Exists(filePath))  {  File.Copy(Application.streamingAssetsPath + "/database.db", filePath);  }    Debug.Log("DB 생성 완료");  }    *// DB파일 경로 획득*  **public** **static** **string** GetDBFilePath()  {  **string** str = **string**.Empty;  str = "URI=file:" + Application.dataPath + "/database.db";    **return** str;  }    *// DB 연결 체크*  **public** **static** **void** DBConnectionCheck()  {  **try**  {  IDbConnection dbConnection = new SqliteConnection(GetDBFilePath());  dbConnection.Open();    **if** (dbConnection.State == ConnectionState.Open)  {  Debug.Log("DB 연결 성공");  }  **else**  {  Debug.LogError("DB 연결 실패(에러)");  }  }  **catch** (Exception e)  {  Debug.Log($"Error: {e}");  }  }    *// 쿼리문을 통해 데이터 읽어오기*  **public** **static** List<**string**> DataBaseRead(**string** query)  {  IDbConnection dbConnection = new SqliteConnection(GetDBFilePath());  dbConnection.Open();  IDbCommand dbCommand = dbConnection.CreateCommand();  dbCommand.CommandText = query; *// 쿼리 입력*  IDataReader dataReader = dbCommand.ExecuteReader(); *// 쿼리 실행*  List<**String**> data = new List<**String**>();  *// 레코드 읽기*  **while** (dataReader.Read())  {  data.**Add**(dataReader.GetString(0));  data.**Add**(dataReader.GetString(1));  data.**Add**(dataReader.GetString(2));  Debug.Log(dataReader.GetString(0) + ", " + dataReader.GetString(1) + ", " + dataReader.GetString(2));  }  dataReader.Dispose();  dataReader = **null**;  dbCommand.Dispose();  dbCommand = **null**;  dbConnection.Close();  dbConnection = **null**;    **return** data;  }    **public** **static** **void** DataBaseChange(**string** query)  {  IDbConnection dbConnection = new SqliteConnection(GetDBFilePath());  dbConnection.Open();  IDbCommand dbCommand = dbConnection.CreateCommand();    dbCommand.CommandText = query;  dbCommand.ExecuteNonQuery();    dbCommand.Dispose();  dbCommand = **null**;  dbConnection.Close();  dbConnection = **null**;  }  } |

**- ThreadManager.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ThreadManager : MonoBehaviour  {  **private** **static** **readonly** List<Action> executeOnMainThread = new List<Action>();  **private** **static** **readonly** List<Action> executeCopiedOnMainThread = new List<Action>();  **private** **static** **bool** actionToExecuteOnMainThread = **false**;    **private** **void** FixedUpdate()  {  UpdateMain();  }    */// <summary>Sets an action to be executed on the main thread.</summary>*  */// <param name="\_action">The action to be executed on the main thread.</param>*  **public** **static** **void** ExecuteOnMainThread(Action \_action)  {  **if** (\_action == **null**)  {  Console.WriteLine("No action to execute on main thread!");  **return**;  }    **lock** (executeOnMainThread)  {  executeOnMainThread.**Add**(\_action);  actionToExecuteOnMainThread = **true**;  }  }    */// <summary>Executes all code meant to run on the main thread. NOTE: Call this ONLY from the main thread.</summary>*  **public** **static** **void** UpdateMain()  {  **if** (actionToExecuteOnMainThread)  {  executeCopiedOnMainThread.Clear();  **lock** (executeOnMainThread)  {  executeCopiedOnMainThread.AddRange(executeOnMainThread);  executeOnMainThread.Clear();  actionToExecuteOnMainThread = **false**;  }    **for** (**int** i = 0; i < executeCopiedOnMainThread.Count; i++)  {  executeCopiedOnMainThread[i]();  }  }  }  } |

**- Constants.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** Constants  {  **public** **const** **int** TICKS\_PER\_SEC = 30;  **public** **const** **int** MS\_PER\_TICK = 1000 / TICKS\_PER\_SEC;  } |

**- ColorPanel.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ColorPanel : MonoBehaviour  {  **public** **int** panelID;  **public** **int** clientID;  } |

**- ColorPanelSpawner.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ColorPanelSpawner : MonoBehaviour  {  **public** **static** ColorPanelSpawner instance;  **public** List<ColorPanel> colorPanels;  **public** GameObject colorPanelPrefab;    **private** Vector3 initPosition;  **private** Vector3 nextPosition;  **private** **float** distance;  **private** **int** panelCount;    **public** **void** Awake()  {  *// 싱글톤 객체 생성*  **if** (instance == **null**)  {  instance = **this**;  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {  initPosition = nextPosition = new Vector3(-10f, 0.05f, 10f);  distance = 2f;  panelCount = 1;    **for** (**int** i = 0; i <= 10; i++)  {  **for** (**int** j = 0; j <= 10; j++)  {  GameObject colorPanel = Instantiate(colorPanelPrefab, nextPosition, Quaternion.identity);  colorPanel.GetComponent<ColorPanel>().panelID = panelCount++;  colorPanels.**Add**(colorPanel.GetComponent<ColorPanel>());  colorPanel.transform.parent = GameObject.Find("World").transform;  nextPosition.x += distance;  }  initPosition.z -= distance;  nextPosition = initPosition;  }  }  } |

**- 클라이언트 프로그램**

**- Client.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;  **using** System.Net;  **using** System.Net.Sockets;  **using** System;      **public** **class** Client : MonoBehaviour  {  **public** **static** Client instance; *// 싱글톤 객체*  **public** **static** **int** dataBufferSize = 4096;    **public** **string** ip = "192.168.1.18"; *// 127.0.0.01은 loopback ip로 localhost의 아이피를 의미함*  **public** **int** port = 9330;  **public** **int** myID = 0;  **public** TCP tcp;  **public** UDP udp;    **bool** isConnected = **false**;  **private** **delegate** **void** PacketHandler(Packet packet);  **private** **static** Dictionary<**int**, PacketHandler> packetHandlers;      **public** **void** Awake()  {  *// 싱글톤 객체 생성*  **if** (instance == **null**)  {  instance = **this**;  DontDestroyOnLoad(**this**);  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {    }    **private** **void** OnApplicationQuit()  {  Disconnect();  }    */// <summary>서버와 연결 -> 서버로부터 Welcome 패킷을 받음으로써 확인</summary>*  **public** **void** OnConnectedToServer()  {  tcp = new TCP();  udp = new UDP();    InitializeClientData();    isConnected = **true**;  tcp.Connect();  }    */// <summary>연결 설정 단계, 패킷과 핸들러 매칭</summary>*  **private** **void** InitializeClientData()  {  packetHandlers = new Dictionary<**int**, PacketHandler>()  {  { (**int**)ServerPackets.welcome, ClientHandle.Welcome },  { (**int**)ServerPackets.loginOrRegisterSuccess, ClientHandle.OnLoginOrRegisterSuccess },  { (**int**)ServerPackets.spawnPlayer, ClientHandle.SpawnPlayer },  { (**int**)ServerPackets.playerReady, ClientHandle.PlayerReady },  { (**int**)ServerPackets.gameStart, ClientHandle.GameStart },  { (**int**)ServerPackets.playerPosition, ClientHandle.PlayerPosition },  { (**int**)ServerPackets.playerRotation, ClientHandle.PlayerRotation },  { (**int**)ServerPackets.playerDisconnected, ClientHandle.PlayerDisconnected },  { (**int**)ServerPackets.playerFlipped, ClientHandle.PlayerFlipped },  { (**int**)ServerPackets.error, ClientHandle.Error },  };  Debug.Log("Initialized packets");  }    **public** **void** Disconnect()  {  **if** (isConnected)  {  isConnected = **false**;  tcp.socket.Close();  udp.socket.Close();    Debug.Log("Disconnected from server.");  }  }    */// <summary>클라이언트 TCP</summary>*  **public** **class** TCP  {  **public** TcpClient socket;    **private** NetworkStream stream;  **private** Packet receivedData;  **private** **byte**[] receiveBuffer;    */// <summary>서버와 연결 시도</summary>*  **public** **void** Connect()  {  *// 소켓의 send, recv 버퍼 크기 지정해줌*  socket = new TcpClient  {  ReceiveBufferSize = dataBufferSize,  SendBufferSize = dataBufferSize  };    receiveBuffer = new **byte**[dataBufferSize];  *// 서버ip, 9330, 콜백함수, 소켓 객체 -> 서버와 연결 시도*  socket.BeginConnect(instance.ip, instance.port, ConnectCallback, socket);  }    */// <summary>연결 시도 결과</summary>*  */// <param name="result">비동기 연결 시도 결과</param>*  **private** **void** ConnectCallback(IAsyncResult result)  {  socket.EndConnect(result); *// 비동기 연결시도 끝냄*    *// 연결 안되어있다면 -> return*  **if** (!socket.Connected)  {  **return**;  }    *// 연결되어 있다면 아래 스크립트 실행*  stream = socket.GetStream();    receivedData = new Packet(); *// Init*    stream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, **null**); *// 소켓을 통해 서버로부터의 데이터 읽어옴*  *//stream.Read(receiveBuffer, 0, dataBufferSize);*  }    **public** **void** SendData(Packet packet)  {  **try**  {  **if** (socket != **null**)  {  *//stream.Write(packet.ToArray(), 0, packet.Length());*  stream.BeginWrite(packet.ToArray(), 0, packet.Length(), **null**, **null**);  }  }  **catch** (Exception e)  {  Debug.Log($"TCP를 통해 서버에 데이터를 보내는 도중 오류가 발생했습니다. : {e}");  }  }    */// <summary>서버가 보낸 데이터를 받았을 때의 콜백</summary>*  */// <param name="result"></param>*  **private** **void** ReceiveCallback(IAsyncResult result)  {  **try**  {  **int** byteLength = stream.EndRead(result); *// 비동기 읽기 시도 끝냄*  *// 읽은 게 없다면, return*  **if** (byteLength <= 0)  {  Disconnect();  **return**;  }    **byte**[] data = new **byte**[byteLength];  Array.Copy(receiveBuffer, data, byteLength);    *// 보내고자 하는 데이터가 너무 클 경우 여러번 나누어 보낼 수도 있음*  *// 패킷을 받는 족족 Reset하게 되면 데이터를 취합할 수 없음*  *// HandleData의 리턴값을 이용하여 데이터를 모두 읽어서 Reset해도 되는 패킷인지 아닌지를 판단*  receivedData.Reset(HandleData(data));  stream.BeginRead(receiveBuffer, 0, dataBufferSize, ReceiveCallback, **null**);  }  **catch**  {  Disconnect();  }  }    */// <summary>receivedData를 사용할 수 있도록 handle</summary>*  */// <param name="data">receivedData</param>*  */// <returns>해당 패킷이 데이터의 마지막인지 여부</returns>*  **private** **bool** HandleData(**byte**[] data)  {  **int** packetLength = 0;    receivedData.SetBytes(data);    *// 패킷 데이터의 처음은 패킷의 데이터 길이(int형 4바이트)*  *// int형을 읽었을 때 packetLength가 0이하가 된다면, 패킷에 데이터 없음*  **if** (receivedData.UnreadLength() >= 4)  {  packetLength = receivedData.ReadInt();  *// 패킷에 데이터가 없다면*  **if** (packetLength <= 0)  {  **return** **true**; *// receviedData Reset, 재사용 가능하도록*  }  }    *// 패킷에 데이터를 모두 읽을 때까지*  **while** (packetLength > 0 && packetLength <= receivedData.UnreadLength())  {  **byte**[] packetBytes = receivedData.ReadBytes(packetLength);  *// MainThread에서 실행되어야할 함수*  ThreadManager.ExecuteOnMainThread(() =>  {  **using** (Packet packet = new Packet(packetBytes))  {  **int** packetID = packet.ReadInt(); *// 패킷 ID 받아옴*  packetHandlers[packetID](packet); *// 해당 패킷 핸들러함수 실행*  }  });    packetLength = 0;    **if** (receivedData.UnreadLength() >= 4)  {  packetLength = receivedData.ReadInt();  **if** (packetLength <= 0)  {  **return** **true**;  }  }  }    **if** (packetLength <= 1)  {  **return** **true**;  }  *// 아직도 데이터를 모두 읽어오지 못했다면 -> 패킷 분할해서 보낸 것임*  **return** **false**;  }    **private** **void** Disconnect()  {  instance.Disconnect();    stream = **null**;  receivedData = **null**;  receiveBuffer = **null**;  socket = **null**;  }  }    */// <summary>클라이언트 UDP</summary>*  **public** **class** UDP  {  **public** UdpClient socket; *// UDP 클라이언트 생성*  **public** IPEndPoint endPoint; *// endpoint 지정*    **public** UDP()  {  endPoint = new IPEndPoint(IPAddress.Parse(instance.ip), instance.port);  }    */// <summary>UDP 사용을 위해 서버와 연결</summary>*  */// <param name="localPort">socket의 bind를 위한 port#</param>*  **public** **void** Connect(**int** localPort)  {  socket = new UdpClient(localPort);    socket.Connect(endPoint); *// IPEndPoint*  socket.BeginReceive(ReceiveCallback, **null**); *// 서버로부터의 입력 받음*    *// 서버에게 UDP 패킷 전송*  **using** (Packet packet = new Packet())  {  SendData(packet);  }  }    */// <summary>UDP를 이용한 데이터 전송</summary>*  */// <param name="packet">The packet to send.</param>*  **public** **void** SendData(Packet packet)  {  **try**  {  packet.InsertInt(instance.myID); *// 패킷의 시작부분에 클라이언트의 ID 추가*  **if** (socket != **null**)  {  socket.BeginSend(packet.ToArray(), packet.Length(), **null**, **null**);  }  }  **catch** (Exception e)  {  Debug.Log($"Error sending data to server via UDP: {e}");  }  }    */// <summary>Receives incoming UDP data.</summary>*  **private** **void** ReceiveCallback(IAsyncResult result)  {  **try**  {  **byte**[] data = socket.EndReceive(result, **ref** endPoint);  socket.BeginReceive(ReceiveCallback, **null**);    **if** (data.Length < 4)  {  instance.Disconnect();  **return**;  }    HandleData(data);  }  **catch**  {  Disconnect();  }  }    */// <summary>Prepares received data to be used by the appropriate packet handler methods.</summary>*  */// <param name="data">The recieved data.</param>*  **private** **void** HandleData(**byte**[] data)  {  **using** (Packet packet = new Packet(data))  {  **int** packetLength = packet.ReadInt();  data = packet.ReadBytes(packetLength);  }    ThreadManager.ExecuteOnMainThread(() =>  {  **using** (Packet packet = new Packet(data))  {  **int** packetId = packet.ReadInt();  packetHandlers[packetId](packet); *// Call appropriate method to handle the packet*  }  });  }    */// <summary>Disconnects from the server and cleans up the UDP connection.</summary>*  **private** **void** Disconnect()  {  instance.Disconnect();    endPoint = **null**;  socket = **null**;  }  }    } |

**- ClientSend.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ClientSend : MonoBehaviour  {  */// <summary>Sends a packet to the server via TCP.</summary>*  */// <param name="\_packet">The packet to send to the sever.</param>*  **private** **static** **void** SendTCPData(Packet packet)  {  packet.WriteLength();  Client.instance.tcp.SendData(packet);  }    */// <summary>Sends a packet to the server via UDP.</summary>*  */// <param name="\_packet">The packet to send to the sever.</param>*  **private** **static** **void** SendUDPData(Packet packet)  {  packet.WriteLength();  Client.instance.udp.SendData(packet);  }    #region Packets    */// <summary>서버가 보낸 Welcome 패킷을 받았음을 서버에게 알림</summary>*  **public** **static** **void** WelcomeReceived()  {  **using** (Packet packet = new Packet((**int**)ClientPackets.welcomeReceived))  {  packet.Write(Client.instance.myID);  packet.Write(UIManager.instance.usernameField.text);  packet.Write(UIManager.instance.passwordField.text);  packet.Write(UIManager.instance.isRegister);    SendTCPData(packet);  }  }    **public** **static** **void** PlayerEnteredRoom()  {  **using** (Packet packet = new Packet((**int**)ClientPackets.playerEnter))  {  packet.Write(UIManager.instance.usernameField.text);  packet.Write(UIManager.instance.passwordField.text);  packet.Write(UIManager.instance.maxPlayer);    SendTCPData(packet);  }  }    **public** **static** **void** PlayerReady()  {  **using** (Packet packet = new Packet((**int**)ClientPackets.playerReady))  {  packet.Write(**true**);  SendTCPData(packet);  }  }    **public** **static** **void** PlayerMovement(**bool**[] inputs)  {  **using** (Packet packet = new Packet((**int**)ClientPackets.playerMovement))  {  packet.Write(inputs.Length);  **foreach** (**bool** input **in** inputs)  {  packet.Write(input);  }  packet.Write(GameManager.players[Client.instance.myID].transform.rotation);    SendUDPData(packet);  }  }    **public** **static** **void** PlayerFlip(ColorPanel panel)  {  **using** (Packet packet = new Packet((**int**)ClientPackets.playerFlip))  {  packet.Write(panel.panelID);    SendTCPData(packet);  }  }  #endregion  } |

**- ClientHandle.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** System.Net;  **using** UnityEngine;  **using** UnityEngine.SceneManagement;    **public** **class** ClientHandle : MonoBehaviour  {  */// <summary>서버로부터 Welcome 패킷을 받았을 때의 처리</summary>*  */// <param name="packet">패킷</param>*  **public** **static** **void** Welcome(Packet packet)  {  **string** msg = packet.ReadString();  **int** myID = packet.ReadInt();    Debug.Log($"Message from server: {msg}");    Client.instance.myID = myID;  ClientSend.WelcomeReceived();  *// 플레이어 이동 처리를 위한 UDP 연결*  Client.instance.udp.Connect(((IPEndPoint)Client.instance.tcp.socket.Client.LocalEndPoint).Port);  }    **public** **static** **void** OnLoginOrRegisterSuccess(Packet packet)  {  UIManager.instance.watch.Stop();  Debug.Log($"총 소요시간: {UIManager.instance.watch.ElapsedMilliseconds}ms");  SceneManager.sceneLoaded += LoadedSceneEvent;  SceneManager.LoadScene("Room");    }    **public** **static** **void** LoadedSceneEvent(Scene scene, LoadSceneMode mode)  {  ClientSend.PlayerEnteredRoom();  }      **public** **static** **void** PlayerDisconnected(Packet packet)  {  **int** id = packet.ReadInt();    **if** (GameManager.players.ContainsKey(id))  {  Destroy(GameManager.players[id].gameObject); *// player 오브젝트 Destroy*  GameManager.players.**Remove**(id); *// player Dictionary에서 지움*  }  }    **public** **static** **void** PlayerReady(Packet packet)  {  GameManager.instance.readyCount = packet.ReadInt();  }    **public** **static** **void** GameStart(Packet packet)  {  Debug.Log("GameStart 패킷 받음");  GameManager.instance.GameStart();  Debug.Log("게임시작");  }    */// <summary>서버로부터 SpawnPlayer 패킷을 받았을 때의 처리</summary>*  */// <param name="packet"></param>*  **public** **static** **void** SpawnPlayer(Packet packet)  {  Debug.Log("Spawn 패킷 받음!");  **int** id = packet.ReadInt();  **string** username = packet.ReadString();  Vector3 position = packet.ReadVector3();  Quaternion rotation = packet.ReadQuaternion();    GameManager.instance.SpawnPlayer(id, username, position, rotation);  }    */// <summary>서버로부터 PlayerPosition 패킷을 받았을 때의 처리</summary>*  */// <param name="packet"></param>*  **public** **static** **void** PlayerPosition(Packet packet)  {  **int** id = packet.ReadInt();  Vector3 position = packet.ReadVector3();    **try**  {  GameManager.players[id].transform.position = position;  }  **catch** (Exception e)  {  Debug.Log("아직 생성 안됨");  }  }    */// <summary>서버로부터 PlayerRotation 패킷을 받았을 때의 처리</summary>*  */// <param name="packet"></param>*  **public** **static** **void** PlayerRotation(Packet packet)  {  **int** id = packet.ReadInt();  Quaternion rotation = packet.ReadQuaternion();    **try**  {  GameManager.players[id].transform.rotation = rotation;  }  **catch** (Exception e)  {  Debug.Log("아직 생성 안됨");  }  }    **public** **static** **void** PlayerFlipped(Packet packet)  {  **int** panelID = packet.ReadInt();  **int** clientID = packet.ReadInt();    **foreach** (ColorPanel panel **in** ColorPanelSpawner.instance.colorPanels)  {  **if** (panel.panelID.Equals(panelID))  {  panel.clientID = clientID;  ColorPanelSpawner.instance.ChangePanelColor(panelID, clientID);  }  }  }    **public** **static** **void** Error(Packet packet)  {  **string** msg = packet.ReadString();    Debug.LogError($"서버로부터의 에러 메세지: {msg}");  UIManager.instance.isButtonClicked = **false**;  UIManager.instance.DebugText.text = msg;  Client.instance.Disconnect();  }  } |

**- Packet.cs**

|  |
| --- |
| **using** System;  **using** System.Collections;  **using** System.Collections.Generic;  **using** System.Text;  **using** UnityEngine;    */// <summary>서버->클라이언트 패킷</summary>*  **public** **enum** ServerPackets  {  welcome = 1,  loginOrRegisterSuccess,  spawnPlayer,  playerReady,  gameStart,  playerPosition,  playerRotation,  playerFlipped,  playerDisconnected,  error,  }    */// <summary>클라이언트->서버 패킷</summary>*  **public** **enum** ClientPackets  {  welcomeReceived = 1,  playerEnter,  playerReady,  playerMovement,  playerFlip,  }    **public** **class** Packet : IDisposable  {  **private** List<**byte**> buffer;  **private** **byte**[] readableBuffer;  **private** **int** readPos;    */// <summary>Creates a new empty packet (without an ID).</summary>*  **public** Packet()  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*  }    */// <summary>Creates a new packet with a given ID. Used for sending.</summary>*  */// <param name="\_id">The packet ID.</param>*  **public** Packet(**int** \_id)  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*    Write(\_id); *// Write packet id to the buffer*  }    */// <summary>Creates a packet from which data can be read. Used for receiving.</summary>*  */// <param name="\_data">The bytes to add to the packet.</param>*  **public** Packet(**byte**[] \_data)  {  buffer = new List<**byte**>(); *// Initialize buffer*  readPos = 0; *// Set readPos to 0*    SetBytes(\_data);  }    #region Functions  */// <summary>Sets the packet's content and prepares it to be read.</summary>*  */// <param name="\_data">The bytes to add to the packet.</param>*  **public** **void** SetBytes(**byte**[] \_data)  {  Write(\_data);  readableBuffer = buffer.ToArray();  }    */// <summary>Inserts the length of the packet's content at the start of the buffer.</summary>*  **public** **void** WriteLength()  {  buffer.InsertRange(0, BitConverter.GetBytes(buffer.Count)); *// Insert the byte length of the packet at the very beginning*  }    */// <summary>Inserts the given int at the start of the buffer.</summary>*  */// <param name="\_value">The int to insert.</param>*  **public** **void** InsertInt(**int** \_value)  {  buffer.InsertRange(0, BitConverter.GetBytes(\_value)); *// Insert the int at the start of the buffer*  }    */// <summary>Gets the packet's content in array form.</summary>*  **public** **byte**[] ToArray()  {  readableBuffer = buffer.ToArray();  **return** readableBuffer;  }    */// <summary>Gets the length of the packet's content.</summary>*  **public** **int** Length()  {  **return** buffer.Count; *// Return the length of buffer*  }    */// <summary>Gets the length of the unread data contained in the packet.</summary>*  **public** **int** UnreadLength()  {  **return** Length() - readPos; *// Return the remaining length (unread)*  }    */// <summary>Resets the packet instance to allow it to be reused.</summary>*  */// <param name="\_shouldReset">Whether or not to reset the packet.</param>*  **public** **void** Reset(**bool** \_shouldReset = **true**)  {  **if** (\_shouldReset)  {  buffer.Clear(); *// Clear buffer*  readableBuffer = **null**;  readPos = 0; *// Reset readPos*  }  **else**  {  readPos -= 4; *// "Unread" the last read int*  }  }  #endregion    #region Write Data  */// <summary>byte 패킷</summary>*  **public** **void** Write(**byte** \_value)  {  buffer.**Add**(\_value);  }  */// <summary>byte[] 패킷</summary>*  **public** **void** Write(**byte**[] \_value)  {  buffer.AddRange(\_value);  }  */// <summary>short 패킷</summary>*  **public** **void** Write(**short** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds an int to the packet.</summary>*  */// <param name="\_value">The int to add.</param>*  **public** **void** Write(**int** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a long to the packet.</summary>*  */// <param name="\_value">The long to add.</param>*  **public** **void** Write(**long** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a float to the packet.</summary>*  */// <param name="\_value">The float to add.</param>*  **public** **void** Write(**float** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a bool to the packet.</summary>*  */// <param name="\_value">The bool to add.</param>*  **public** **void** Write(**bool** \_value)  {  buffer.AddRange(BitConverter.GetBytes(\_value));  }  */// <summary>Adds a string to the packet.</summary>*  */// <param name="\_value">The string to add.</param>*  **public** **void** Write(**string** \_value)  {  Write(\_value.Length); *// Add the length of the string to the packet*  buffer.AddRange(Encoding.ASCII.GetBytes(\_value)); *// Add the string itself*  }  */// <summary>Adds a Vector3 to the packet.</summary>*  */// <param name="\_value">The Vector3 to add.</param>*  **public** **void** Write(Vector3 \_value)  {  Write(\_value.x);  Write(\_value.y);  Write(\_value.z);  }  */// <summary>Adds a Quaternion to the packet.</summary>*  */// <param name="\_value">The Quaternion to add.</param>*  **public** **void** Write(Quaternion \_value)  {  Write(\_value.x);  Write(\_value.y);  Write(\_value.z);  Write(\_value.w);  }  #endregion    #region Read Data  */// <summary>Reads a byte from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **byte** ReadByte(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **byte** \_value = readableBuffer[readPos]; *// Get the byte at readPos' position*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 1; *// Increase readPos by 1*  }  **return** \_value; *// Return the byte*  }  **else**  {  **throw** new Exception("Could not read value of type 'byte'!");  }  }    */// <summary>Reads an array of bytes from the packet.</summary>*  */// <param name="\_length">The length of the byte array.</param>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **byte**[] ReadBytes(**int** \_length, **bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **byte**[] \_value = buffer.GetRange(readPos, \_length).ToArray(); *// Get the bytes at readPos' position with a range of \_length*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += \_length; *// Increase readPos by \_length*  }  **return** \_value; *// Return the bytes*  }  **else**  {  **throw** new Exception("Could not read value of type 'byte[]'!");  }  }    */// <summary>Reads a short from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **short** ReadShort(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **short** \_value = BitConverter.ToInt16(readableBuffer, readPos); *// Convert the bytes to a short*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true and there are unread bytes*  readPos += 2; *// Increase readPos by 2*  }  **return** \_value; *// Return the short*  }  **else**  {  **throw** new Exception("Could not read value of type 'short'!");  }  }    */// <summary>Reads an int from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **int** ReadInt(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **int** \_value = BitConverter.ToInt32(readableBuffer, readPos); *// Convert the bytes to an int*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 4; *// Increase readPos by 4*  }  **return** \_value; *// Return the int*  }  **else**  {  **throw** new Exception("Could not read value of type 'int'!");  }  }    */// <summary>Reads a long from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **long** ReadLong(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **long** \_value = BitConverter.ToInt64(readableBuffer, readPos); *// Convert the bytes to a long*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 8; *// Increase readPos by 8*  }  **return** \_value; *// Return the long*  }  **else**  {  **throw** new Exception("Could not read value of type 'long'!");  }  }    */// <summary>Reads a float from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **float** ReadFloat(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **float** \_value = BitConverter.ToSingle(readableBuffer, readPos); *// Convert the bytes to a float*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 4; *// Increase readPos by 4*  }  **return** \_value; *// Return the float*  }  **else**  {  **throw** new Exception("Could not read value of type 'float'!");  }  }    */// <summary>Reads a bool from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **bool** ReadBool(**bool** \_moveReadPos = **true**)  {  **if** (buffer.Count > readPos)  {  *// If there are unread bytes*  **bool** \_value = BitConverter.ToBoolean(readableBuffer, readPos); *// Convert the bytes to a bool*  **if** (\_moveReadPos)  {  *// If \_moveReadPos is true*  readPos += 1; *// Increase readPos by 1*  }  **return** \_value; *// Return the bool*  }  **else**  {  **throw** new Exception("Could not read value of type 'bool'!");  }  }    */// <summary>Reads a string from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** **string** ReadString(**bool** \_moveReadPos = **true**)  {  **try**  {  **int** \_length = ReadInt(); *// Get the length of the string*  **string** \_value = Encoding.ASCII.GetString(readableBuffer, readPos, \_length); *// Convert the bytes to a string*  **if** (\_moveReadPos && \_value.Length > 0)  {  *// If \_moveReadPos is true string is not empty*  readPos += \_length; *// Increase readPos by the length of the string*  }  **return** \_value; *// Return the string*  }  **catch**  {  **throw** new Exception("Could not read value of type 'string'!");  }  }    */// <summary>Reads a Vector3 from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** Vector3 ReadVector3(**bool** \_moveReadPos = **true**)  {  **return** new Vector3(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));  }    */// <summary>Reads a Quaternion from the packet.</summary>*  */// <param name="\_moveReadPos">Whether or not to move the buffer's read position.</param>*  **public** Quaternion ReadQuaternion(**bool** \_moveReadPos = **true**)  {  **return** new Quaternion(ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos), ReadFloat(\_moveReadPos));  }  #endregion    **private** **bool** disposed = **false**;    **protected** **virtual** **void** Dispose(**bool** \_disposing)  {  **if** (!disposed)  {  **if** (\_disposing)  {  buffer = **null**;  readableBuffer = **null**;  readPos = 0;  }    disposed = **true**;  }  }    **public** **void** Dispose()  {  Dispose(**true**);  GC.SuppressFinalize(**this**);  }  } |

**- PlayerController.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** PlayerController : MonoBehaviour  {  **private** **void** FixedUpdate()  {  SendInputToServer();  }    *// 입력된 값을 서버에게 보냄*  **private** **void** SendInputToServer()  {  **bool**[] inputs = new **bool**[]  {  Input.GetKey(KeyCode.W),  Input.GetKey(KeyCode.S),  Input.GetKey(KeyCode.A),  Input.GetKey(KeyCode.D),  Input.GetKey(KeyCode.Space),  };    ClientSend.PlayerMovement(inputs);  }  } |

**- PlayerManager.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** PlayerManager : MonoBehaviour  {  **public** **int** id;  **public** **string** username;  **public** **bool** isReady;  } |

**- GameManager.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** System.Linq;  **using** UnityEngine;  **using** UnityEngine.UI;    **public** **class** GameManager : MonoBehaviour  {  **public** **static** GameManager instance;    **public** **static** Dictionary<**int**, PlayerManager> players = new Dictionary<**int**, PlayerManager>();    **public** GameObject localPlayerPrefab;  **public** GameObject playerPrefab;  **public** GameObject readyPanel;  **public** Text readyText;  **public** Text playtimeText;  **public** GameObject aim;  **public** GameObject resultPanel;  **public** List<GameObject> labels;    **public** **bool** isReady;  **public** **int** readyCount;  **public** **bool** isGameStarted;    [SerializeField]  **private** **float** playtime;    **public** **void** Awake()  {  *// 싱글톤 객체 생성*  **if** (instance == **null**)  {  instance = **this**;  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {  isReady = **false**;  isGameStarted = **false**;  readyPanel.SetActive(**true**);  resultPanel.SetActive(**false**);  playtimeText.gameObject.SetActive(**false**);  aim.SetActive(**false**);  playtime = 10f;  readyCount = 0;  }    **private** **void** Update()  {  *// 제한시간이 다 되었다면*  **if** (isGameStarted && playtime < 0)  {  isGameStarted = **false**;  aim.SetActive(**false**);    Dictionary<**int**, **int**> results = new Dictionary<**int**, **int**>();  **for** (**int** i = 1; i <= readyCount; i++)  {  results.**Add**(i, 0);  }    **foreach** (ColorPanel panel **in** ColorPanelSpawner.instance.colorPanels)  {  **if** (panel.clientID != 0)  results[panel.clientID]++;  }    GetResult(results);  resultPanel.SetActive(**true**);  }  **if** (isGameStarted)  {  playtime -= Time.deltaTime;  playtimeText.text = "남은 시간: " + (**int**)playtime + "초";  }    *// 플레이어가 Ready하지 않았을 때만*  **if** (!isReady)  {  **if** (Input.GetKeyDown(KeyCode.R))  {  PlayerManager player;  players.TryGetValue(Client.instance.myID, **out** player);  player.isReady = **true**;  isReady = **true**;  ClientSend.PlayerReady();  }  }  }    **public** **void** SpawnPlayer(**int** id, **string** username, Vector3 position, Quaternion rotation)  {  GameObject player;  **if** (id == Client.instance.myID)  {  player = Instantiate(localPlayerPrefab, position, rotation);    }  **else**  {  player = Instantiate(playerPrefab, position, rotation);  }    InitPlayer(player.GetComponent<PlayerManager>(), id, username);  player.transform.Find("Canvas").GetChild(0).GetComponent<Text>().text = username;  players.**Add**(id, player.GetComponent<PlayerManager>());  }    **public** **void** GameStart()  {  StartCoroutine(GameStartCoroutine());  }    **private** **void** InitPlayer(PlayerManager player, **int** id, **string** username)  {  player.id = id;  player.username = username;  player.isReady = **false**;  }    **public** IEnumerator GameStartCoroutine()  {  readyText.text = "제한시간 안에 최대한 많은**\n**색판을 뒤집어라!";  **yield** **return** new WaitForSeconds(3f);  **for** (**int** i = 5; i >= 1; i--)  {  readyText.text = i + "";  **yield** **return** new WaitForSeconds(1f);  }  readyText.text = "Start!";  **yield** **return** new WaitForSeconds(1f);  readyPanel.SetActive(**false**);  playtimeText.gameObject.SetActive(**true**);  isGameStarted = **true**;  ColorPanelSpawner.instance.panels.SetActive(**true**);  aim.SetActive(**true**);  }    *// 결과창 순위표*  **private** **void** GetResult(Dictionary<**int**, **int**> results)  {  **var** sortedResults = results.OrderByDescending(x => x.**Value**);  **int** rank = 1;  **int** cnt = 0;    **foreach** (**var** result **in** sortedResults)  {  labels[cnt].SetActive(**true**);  labels[cnt].transform.GetChild(0).GetComponent<Text>().text = $"{rank}위"; *// 순위*  labels[cnt].transform.GetChild(1).GetComponent<Text>().text = players[result.Key].username; *// 플레이어 이름*  labels[cnt].transform.GetChild(2).GetComponent<Text>().text = $"{result.Value}"; *// 뒤집은 색판 개수*  *// 플레이어 색*  **switch** (result.Key)  {  **case** 1:  labels[cnt].transform.GetChild(3).GetComponent<Image>().color = Color.red;  **break**;  **case** 2:  labels[cnt].transform.GetChild(3).GetComponent<Image>().color = Color.blue;  **break**;  **case** 3:  labels[cnt].transform.GetChild(3).GetComponent<Image>().color = Color.yellow;  **break**;  **case** 4:  labels[cnt].transform.GetChild(3).GetComponent<Image>().color = Color.green;  **break**;  }  rank++;  cnt++;  }  }    **public** **void** Quit()  {  Debug.Log("게임종료");  Application.Quit();  }  } |

**- CamerController.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    */// <summary>카메라 움직임 및 Ray 관리</summary>*  **public** **class** CameraController : MonoBehaviour  {  **public** PlayerManager player;  **public** **float** sensitivity = 100f;  **public** **float** clampAngle = 85f;  **public** **float** maxDistance = 5f;    **private** **float** verticalRotation;  **private** **float** horizontalRotation;  **private** **int** layerMask;  **private** Ray ray;  **private** RaycastHit rayHit;  **private** GameObject selectedPanel;      **private** **void** Start()  {  verticalRotation = transform.localEulerAngles.x;  horizontalRotation = player.transform.eulerAngles.y;    layerMask = 1 << LayerMask.NameToLayer("ColorPanel");  }    **private** **void** Update()  {  Look();  *//Debug.DrawRay(transform.position, transform.forward \* 5f, Color.red); // 사용자 시야 Ray 그림*  ray = GetComponent<Camera>().ViewportPointToRay(new Vector3(0.5f, 0.5f, 0f)); *// 화면의 가운데 기준으로 ray 생성*    *// hit한 Object가 ColorPanel일 경우*  **if** (Physics.Raycast(ray, **out** rayHit, maxDistance, layerMask))  {  Debug.DrawLine(ray.origin, rayHit.point, Color.green);  selectedPanel = rayHit.transform.GetChild(0).gameObject;  selectedPanel.SetActive(**true**);  **if** (Input.GetKeyDown(KeyCode.F))  {  Debug.Log("색판 뒤집음");  ClientSend.PlayerFlip(rayHit.collider.gameObject.GetComponent<ColorPanel>());  }  }  **else**  {  **if** (selectedPanel != **null**)  selectedPanel.SetActive(**false**);  Debug.DrawRay(ray.origin, ray.direction \* maxDistance, Color.red);  }  }    **private** **void** Look()  {  **float** mouseVertical = -Input.GetAxis("Mouse Y");  **float** mouseHorizontal = Input.GetAxis("Mouse X");    verticalRotation += mouseVertical \* sensitivity \* Time.deltaTime;  horizontalRotation += mouseHorizontal \* sensitivity \* Time.deltaTime;    verticalRotation = Mathf.Clamp(verticalRotation, -clampAngle, clampAngle); *// vertical 움직임 제한*    transform.localRotation = Quaternion.Euler(verticalRotation, 0f, 0f);  player.transform.rotation = Quaternion.Euler(0f, horizontalRotation, 0f);  }  } |

**- UIManager.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;  **using** UnityEngine.UI;  **using** UnityEngine.SceneManagement;      **public** **class** UIManager : MonoBehaviour  {  **public** **static** UIManager instance; *// 싱글톤 객체*    **public** GameObject camera;  **public** GameObject startMenu;  **public** InputField usernameField;  **public** InputField passwordField;  **public** List<Toggle> maxPlayerToggles;  **public** Text DebugText;    **public** **int** maxPlayer;  **public** **bool** isButtonClicked;  **private** **float** timer;  **public** **bool** isRegister;    **public** System.Diagnostics.Stopwatch watch;    **private** **void** Awake()  {  **if** (instance == **null**)  {  instance = **this**;  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {  maxPlayer = 2;  maxPlayerToggles[0].isOn = **true**;  timer = 0;  isRegister = **false**;    watch = new System.Diagnostics.Stopwatch();  }    **private** **void** Update()  {  **if** (isButtonClicked)  {  timer += Time.deltaTime;  *// 에러 핸들링이 불가능하다면 서버와 연결에 실패함을 의미*  **if** (timer > 3f)  {  Debug.LogError("Error: 서버가 동작하고 있지 않거나 방이 꽉 찼습니다.");  DebugText.text = "Server is not running or Room is full";  timer = 0f;  isButtonClicked = **false**;  }  }  }    **public** **void** OnConnectedToServer(**bool** isRegister)  {  *// 텍스트필드를 다 채웠을 경우에만 로그인/회원가입 진행*  **if** (usernameField.text != "" && passwordField.text != "")  {  watch.Start();  **this**.isRegister = isRegister;  isButtonClicked = **true**;  Client.instance.OnConnectedToServer();  }  **else**  {  DebugText.text = "Fill your TextField Area!";  }  }    **public** **void** OnToggleValueChanges(**int** maxPlayer)  {  **this**.maxPlayer = maxPlayer;  }  } |

**- ColorPanel.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    **public** **class** ColorPanel : MonoBehaviour  {  **public** **int** panelID;  **public** **int** clientID;  } |

**- ColorPanelSpawner.cs**

|  |
| --- |
| **using** System.Collections;  **using** System.Collections.Generic;  **using** UnityEngine;    *// 색판을 생성하는 코드*  **public** **class** ColorPanelSpawner : MonoBehaviour  {  **public** **static** ColorPanelSpawner instance;  **public** GameObject colorPanelPrefab;  **public** GameObject panels;  **public** List<ColorPanel> colorPanels;  **public** List<Material> colors;    **private** Vector3 initPosition;  **private** Vector3 nextPosition;  **private** **float** distance;  **private** **int** panelCount;    **public** **void** Awake()  {  *// 싱글톤 객체 생성*  **if** (instance == **null**)  {  instance = **this**;  }  **else** **if** (instance != **this**)  {  Debug.Log("싱글톤 인스턴스가 존재하므로 오브젝트를 Destroy합니다.");  Destroy(**this**);  }  }    **private** **void** Start()  {  initPosition = nextPosition = new Vector3(-10f, 0.05f, 10f);  distance = 2f;  panelCount = 1;    **for** (**int** i = 0; i <= 10; i++)  {  **for** (**int** j = 0; j <= 10; j++)  {  GameObject colorPanel = Instantiate(colorPanelPrefab, nextPosition, Quaternion.identity);  colorPanel.GetComponent<ColorPanel>().panelID = panelCount++;  colorPanels.**Add**(colorPanel.GetComponent<ColorPanel>());  colorPanel.transform.parent = GameObject.Find("Panels").transform;  nextPosition.x += distance;  }  initPosition.z -= distance;  nextPosition = initPosition;  }  panels.SetActive(**false**);  }    **public** **void** ChangePanelColor(**int** panelID, **int** clientID)  {  colorPanels[panelID - 1].gameObject.GetComponent<MeshRenderer>().material = colors[clientID];  }  } |