**네트워크 게임프로그래밍 Term프로젝트 최종 보고서**

2017182012 김호진

2017182021 양현석

2017182026 윤장혁

**목차**

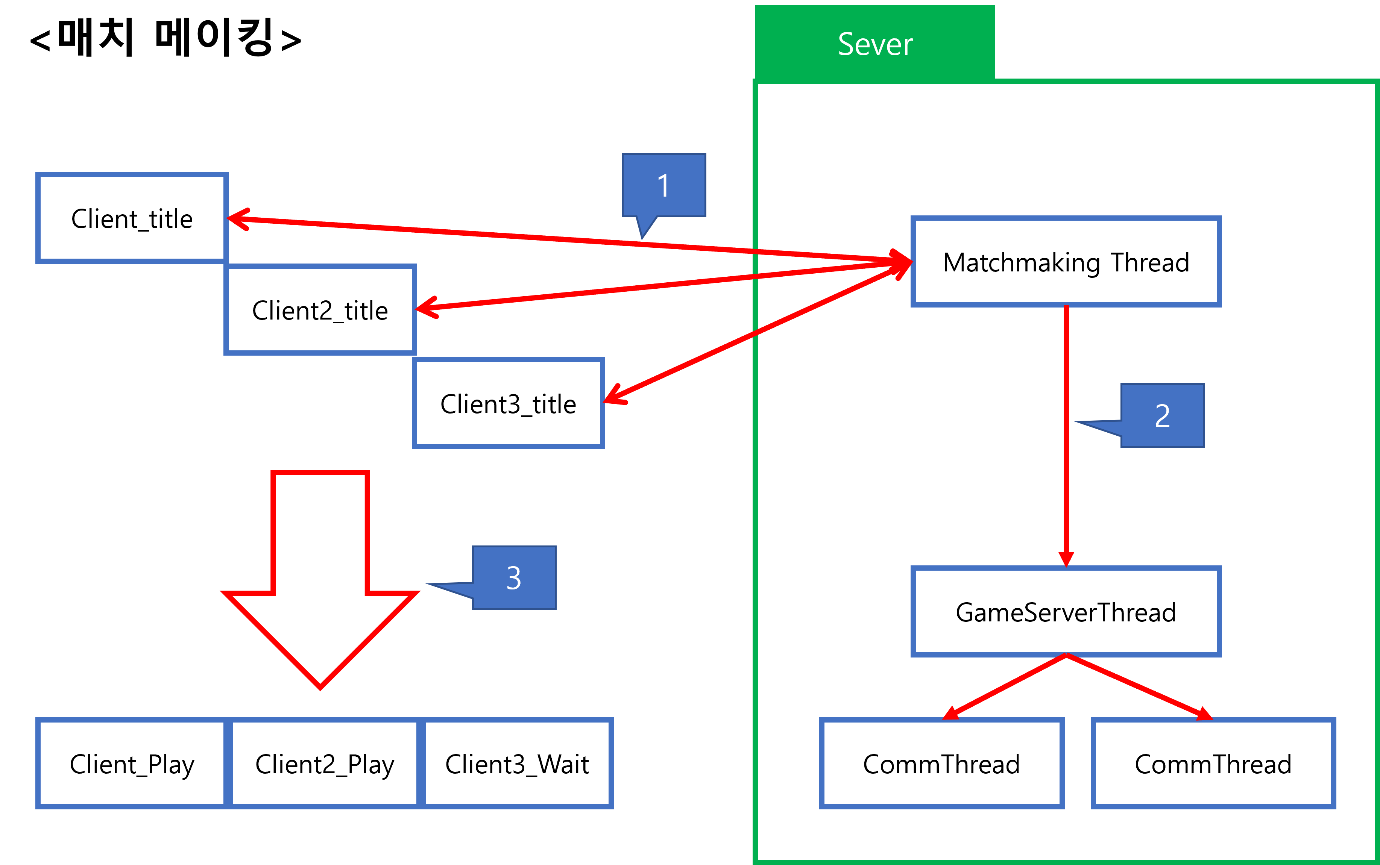
1. 애플리케이션 기획--------------------------------------------------------- p.3
2. 개발 환경------------------------------------------------------------------- p.3
3. HIGH LEVEL디자인--------------------------------------------------------- p.4
4. LOW LEVEL디자인---------------------------------------------------------- p.7
5. 팀원들의 역할 분담-------------------------------------------------------- p.18
6. 개발 일정------------------------------------------------------------------- p.19
7. 일일 보고서---------------------------------------------------------------- p.20
8. **애플리케이션 기획**

* 2D로 제작된 1:1:1로 진행되는 테트리스
* 주어진 공간안에 블록들을 이용해서 하나의 줄을 만들면 해당 줄이 지워진다.
* 한 줄을 완성하면 해당 줄이 지워지면서 상대방을 방해하기 위한 라인을 보낸다.
* 방해라인을 받은 플레이어 최하단부에 한 칸 비워진 라인이 추가된다.
* 만약 플레이어가 블록을 공간내에 넣을 자리가 없다면 패배하게 된다.

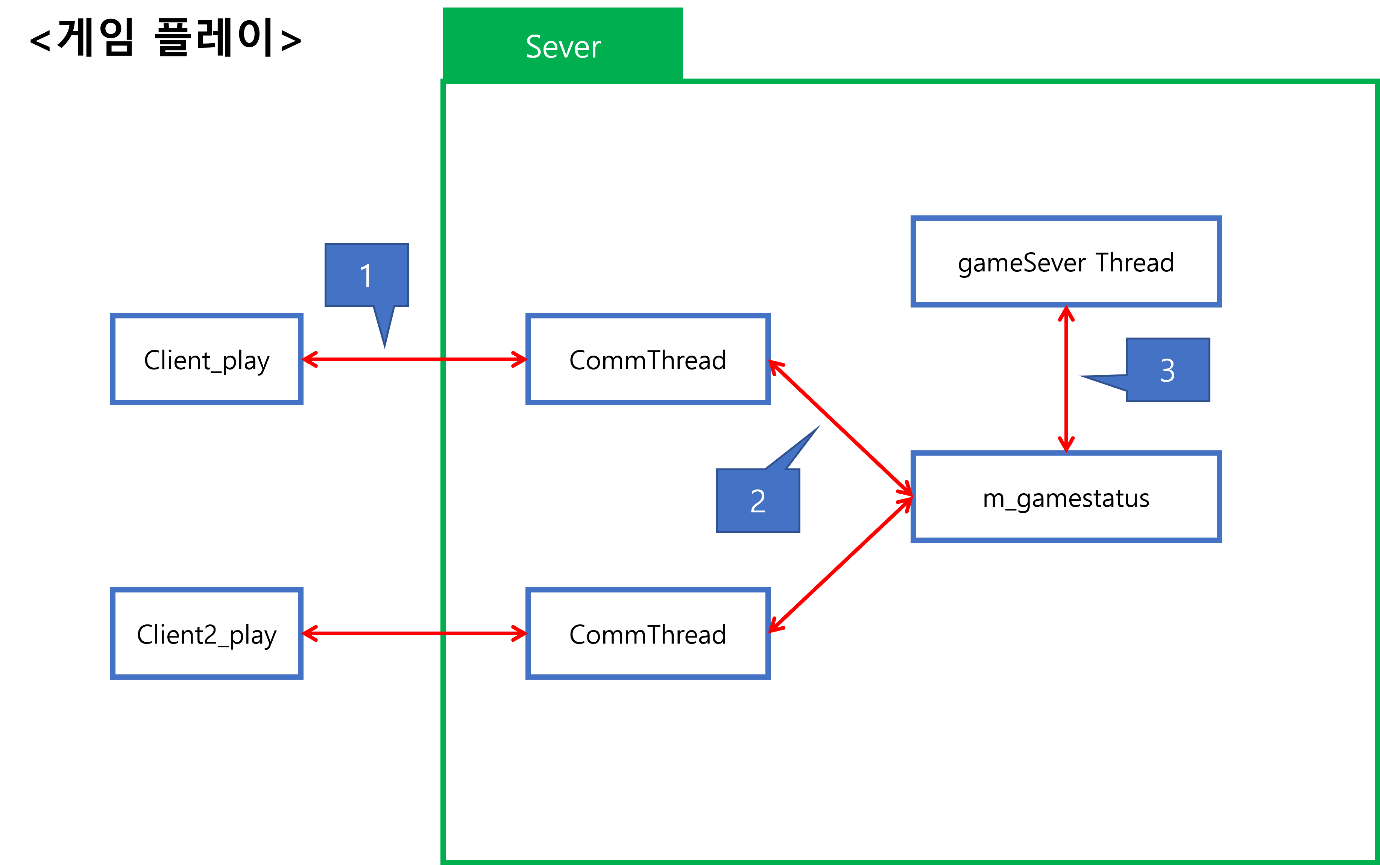
1. **개발환경**

* 사용 운영체제: Windows
* 사용 언어: C, C++, ws2\_32
* 사용 IDE: visual studio 2019
* VCS: git

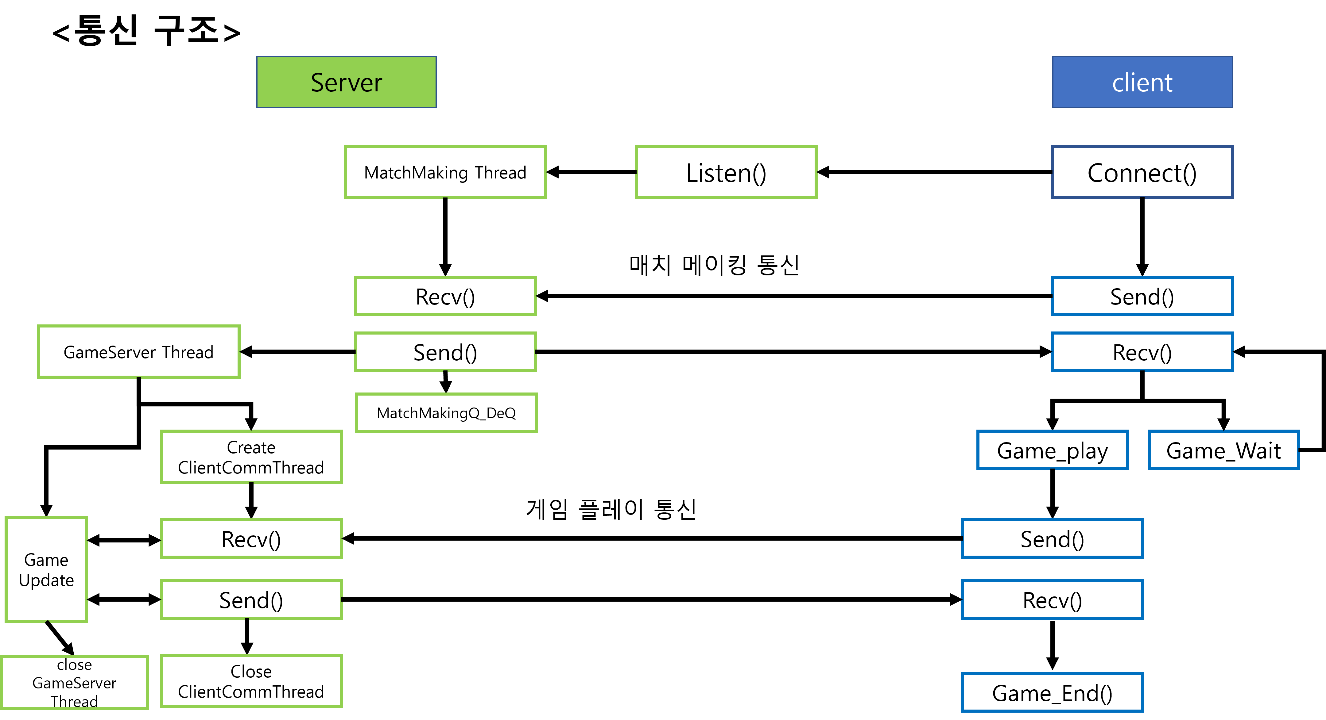
1. **HIGH LEVEL디자인**
   1. **매치 메이킹 서버**



1. 플레이어가 타이틀 화면에서 게임 시작을 눌렀을 경우 서버에 connect을 한다. 그리고 Matchmaking Thread와 통신을 하면서 클라이언트에서 Msg\_Ready를 보내 대기중이라는 상태를 알려준다. 서버에서 클라이언트의 accept을 기다리다가 연결이 되었을 경우 해당 소켓을 MatchMakingQ에 추가를 한다. 남은 클라이언트에게 매칭상대가 올 때 까지Msg\_wait을 보낸다. 대기중인 클라이언트에서 Msg\_ReadCancel이 들어오면, Msg\_ConfirmReadyCancel을 클라이언트에게 보내주고 MatchMakingQ에서 제거한다.
2. MatchMakingQ에 ~~2개의~~ MAX\_PLAYER개의 [[1]](#footnote-1)클라이언트가 들어왔을 경우 MatchMakingQ에 들어있는 클라이언트에게 Msg\_PlayInGame을 보내고 GameSeverThread를 생성하고 클라이언트들과 통신하기 위한 CommThread를 생성한다.
3. Msg\_PlayInGame을 받은 클라이언트들은 Play\_scene으로 넘어간다. Msg\_Wait을 받은 클라이언트는 Wait\_scene으로 넘어간다.
   1. **게임플레이**



1. 클라이언트들은 서버에 있는 CommThread에 자신의 KeyInput을 보낸다.  
   CommThread로부터 상대방의 KeyInput값을 받아 움직임을 클라이언트에서 보이도록 한다.
2. CommThread에 있는 데이터들을 m\_gamestatus에 플레이어별 KeyInput을 갱신하고, 갱신된  
   KeyInput을 읽어간다.
3. KeyInput을 이용하여 플레이어들의 상태를 업데이트한다. 해당 데이터를 m\_gamestatus에 저장한다.
   1. **통신 과정**



1. 서버가 업앤 러닝 상태일 경우 Listen()을 이용해 connect을 받을 준비를 하며 MatchMaking Thread를 생성한다.
2. MathcMaking Thread에서는 Accept을 한 클라이언트의 소켓을 MatchMakingQ에 추가한다.
   1. MatchMakingQ에 추가된 클라이언트는 MatchMaking Thread에 Msg\_Ready를 전송한다.
3. MatchMakingQ에 ~~2개의~~ MAX\_PLAYER개의 [[2]](#footnote-2)클라이언트가 들어 올때까지 MatchMakingQ에 client socket을 추가한다.
   1. ~~2개의~~ MAX\_PLAYER개의2 클라이언트가 들어왔을 경우 GameServerThread를 생성하고 클라이언트들에게 Msg\_PlayInGame을 전송한다. ~~3개의~~ MAX\_PLAYER개이상의2 클라이언트가 들어왔을 경우 ~~2개의~~ MAX\_PLAYER개의2 클라이언트에겐 Msg\_PlayInGame, 전송 남은 클라이언트 에게는 Msg\_Wait을 전송한다.
4. Msg\_PlayInGame을 전송한 클라이언트들은 Play\_Scene으로 넘어가게 된다.
   1. Msg\_Wait을 받은 클라이언트는 Wait\_Scene으로 넘어가 서버로부터 Msg\_PlayInGame이 오는 것을 기다린다.
   2. 서버는 MatchMakingQ를 비우고 Game Server Thread에 Client의 데이터를 받기 위한 CommThread를 생성한다.
5. Client의 키입력 전송
   1. 서버에서는 CommThread에서 받은 키입력 데이터를 업데이트 한 뒤 게임 정보를 client에게 보내준다.
6. Client 게임 종료 후
   1. 서버에서 CommThread와 GameServerThread를 삭제한다
7. **Low Level 디자인**

MatchMaking [서버]

DWORD WINAPI MatchMakingThread(); //매치매이킹 관리 스레드로, 최초 프로세스 시작 시 서버와 함께 실행

queue<SOCKADDR> MatchMakingQ //대기 중인 클라이언트 소켓을 저장하는 전역 변수

enum MSG\_MatchMaking //매치매이킹 시스템에서 사용할 메시지

{

Msg\_Ready, //클라이언트가 서버에게 대기중임을 알림

Msg\_ReadyCancel, //클라이언트가 서버에게 대기 취소를 알림

Msg\_ConfirmCancel, //서버가 클라이언트에게 대기 취소 받았음을 돌려줌

Msg\_PlayInGame //서버가 클라이언트에게 게임이 시작 됐음을 알려줌

Msg\_WaitGame //서버가 클라이언트에게 아직 매칭이 되지 않았음을 알려줌

}

bool isMatchMakingQFull(vector<SOCKET>\*MatchMakingQ)   
//현재 MatchMakingQ에서 대기하는 클라이언트의 개수를 파악해  
MAX\_PLAYER 미만인 경우 MatchMakingQ에 추가되어 있는 클라이언트들에게 Msg\_wait을 보냄 MAX\_PLAYER 이상인 경우 MatchMakingQ에 추가되어 있는 순서대로 MAX\_PLAYER개의 클라이언트에게 Msg\_PlayInGame을 보내주고 CreateGameServerThread와 MatchMakingQ\_DeQ를 실행

void CreateGameServerThread(MatchSockets\* target)

//GameServerThread를 생성하며, MatchMakingQ의 소켓들을 전달

void MatchMakingQ\_CloseSocket(vector<SOCKET>\* MatchMakingQ, SOCKET client);

//MatchMakingQ에서 특정 클라이언트 소켓을 제거

SOCKET MatchMakingQ\_DeQ(vector<SOCKET>\* MatchMakingQ)

//MatchMakingQ에서 제일 앞의 소켓을 제거 후 해당 소켓을 return

int SendMsgtoClient(int Msg, SOCKET client);

// 클라이언트에게 메세지 전송

int RecvMsgfromClient(SOCKET client);

// 클라이언트에게서 메세지 수신

Game Server [서버]

DWORD WINAPI GameServerThread(LPVOID\* arg)

//Msg\_PlayInGame 을 받은 후 실행, GameServerThread 생성

struct Player

{

SOCKET clientSocket;

KeyInput m\_Key[MAX\_PLAYER]; //클라이언트의 키입력을 저장

Gamestatus m\_gamestatus[MAX\_PLAYER]; //게임의 상태를 저장

int m\_GameClientNum; //클라이언트별 번호 메모

CRITICAL\_SECTION cs;

};

struct GameServerThreadData

{

std::vector<SOCKET\*> pClients; //MatchingThread에 넘겨받은 클라이언트 소켓

std::vector<Player\*> pPlayers; //플레이어 개개인의 상태

CGameTimer m\_timer; //업데이트에서 프레임시간 연산 시 사용

CRITICAL\_SECTION cs;[[3]](#footnote-3)

void ~~NewCommThread(void);~~ CreateCommThread(void);[[4]](#footnote-4) //클라이언트와 통신할 스레드 생성

void reset();

void reset\_main();

void new\_block(int ClientNum);

void check\_key();

void KeyUpdate(int clientNum, float fTimeElapsed); // 받은 키에 따라 상태를 업데이트 하는 함수

void TargetShift(int clientNum, float fTimeElapsed); // 쉬프트키를 눌럿을 때 타겟을 변경하는 함수

void drop\_block(int ClientNum, float fTimeElapsed); // 시간이 지남에 따라 블록이 떨어지는 함수

void hard\_drop\_block(int ClientNum); // 스페이스바를 눌럿을 때 한번에 떨어지게 하는 함수

bool check\_crush(int ClientNum, int bx, int by, int b\_rotation); // 충돌을 체크하는 함수

bool check\_crush(int ClientNum, int bx, int by, int b\_rotation, int b\_type); // 충돌을 체크하는 함수

void move\_block(int ClientNum, int dir); // 블록을 상하좌우로 움직이는 함수

void check\_line(int ClientNum); // 한줄이 가득 찻는지 체크하는 함수

void copy\_another\_map(void); // 다른 플레이어에게 내 상태를 복사해주는 함수

void check\_game\_over(int ClientNum); // 게임오버를 체크하는 함수

void check\_win(int ClientNum); // 이겻는지 체크하는 함수

void attack(int ClientNum, int Combo); // 공격을 처리하는 함수

void attacked(int ClientNum); // 공격 받는 것을 처리하는 함수

void check\_level\_up(float fTimeElapsed); // 시간이 지남에 따라 내려오는 속도가 빨라지는 함수

void ActiveItem(int ClientNum, float fTimeElapsed); // 아이템 사용을 처리하는 함수

bool Room\_end(); // 방에 클라이언트가 모두 닫혔을 때 방 스레드를 종료하는 함수

int blocks[7][4][4][4][[5]](#footnote-5) // 블록 모양 정보를 정의한 변수

};

ClientGameData ConvertGameData(Gamestatus m\_gamestate) // 클라이언트 보낼 데이터를 최적화 하는 함수

DWORD WINAPI CommThread(LPVOID\* arg)

[클라이언트]

class GameClient

{

private:

WSADATA wsa;

SOCKET sock;

SOCKADDR\_IN serveraddr;

~~KeyInput m\_keys; //클라이언트의 키입력을 저장~~ // GamePlayScene 안으로 위치 변경

~~Gamestatus m\_gamestatus[MAX\_PLAYER]; //내 클라이언트의 게임 상태~~ // GamePlayScene 안으로 위치 변경

CGameTimer m\_GameTimer;

private: //에러 발생 시 해당 내용 출력

void err\_quit(char\* msg);

void err\_display(char\* msg);

public:

GameClient(); //생성 시 윈속초기화, 소켓 생성, connect를 수행

~GameClient (); //소멸 시 closesocket 과 윈속종료 수행

public:

~~void TitleSceneSend(enum MSG\_MathcingSystem);~~

~~//Msg\_Ready 나 Msg\_ReadyCancel 를 MatchingThread 에 전송~~

~~int TitleSceneRecv();~~

~~//MatchMakingThread 로부터 다음 메시지를 수령~~

~~Msg\_ConfirmReadyCancel: Msg\_ReadyCancel 전송 후, 서버 쪽에서 수신이 확인됐음을~~

~~확인하고 연결 종료~~

~~Msg\_PlayInGame: PlayScene 으로 전환~~

~~Msg\_WaitGame: 현재 Scene에서 Msg\_PlayInGame을 받을때까지 대기~~

~~void PlaySceneSend(); //KeyInput m\_keys 의 값을 Commthread 에 전송~~

~~void PlaySceneRecv(); //GameServer 에서 연산된 값을 CommThread 로부터   
고정-가변 데이터 형태로 수신, 고정은 char형태이며, 가변은GameStatus m\_gamestatus을 수신~~

~~void UpdateData(Gamestatus\* recvGSData);~~

~~//서버에서 받은 데이터로 현재 클라이언트의 데이터를 업데이트~~

~~void SetKeyInput(KeyInput\* keyinput)~~

~~//보내야 할 클라이언트의 키 값을 받아서 저장~~

void ChangeScene(Scene::SceneNum tag); // Scene 체인지

void BuildScene(); // Scene 생성

void Update(); // 업데이트 함수

void Paint(HDC hDC); // 그리기 함수

void KeyDown(unsigned char KEYCODE); // 키 눌렷을 때 메세지 처리 함수

void KeyUp(unsigned char KEYCODE); // 키 땟을 때 메시지 처리 함수

void ScreenRotate(HDC hDC, RECT rt); // 아이템 적용시 화면 돌리는 함수

void SetSOCKET(SOCKET); // 소켓 지정

SOCKET GetSOCKET(); // 소켓 정보 받기

bool InitWSA(); // WSADATA 정의

void ConnetServer(); // 서버 연결

Scene\* m\_pScene; // 현재 Scene

Scene\* m\_arrScene[Scene::SceneNum::Scene\_Count]; // Scene 정보를 담고있는 배열

int m\_ClientNum = -1; // 나의 클라이언트 번호

};

공통 사용 구조체

#define BOARD\_X 11 //게임판 가로크기

#define BOARD\_Y ~~23~~ 25 //게임판 세로크기

#define MAX\_PLAYER ~~2~~ 3[[6]](#footnote-6)// 최대 사용자는 3

~~class GlobalGameData //CommThread와 클라이언트 간의 소켓, 게임 업데이트의 입력 및 출력에 쓰이는 데이터~~

~~{~~

~~private:~~

~~SOCKET m\_sockets[MAX\_PLAYER]; //통신하고 있는 소켓들을 저장~~

~~Gamestatus m\_gamestatus [MAX\_PLAYER]; //캐릭터 상태 저장~~

~~KeyInput m\_keyInput[MAX\_PLAYER]; //각 클라이언트 키 입력 저장~~

~~};~~

struct Gamestatus {

int bx, by; //이동중인 블록의 게임판상의 x,y좌표

int b\_type; //블록 종류

int b\_rotation; //블록 회전값

int b\_type\_next; //다음 블록값

int level; //현재 level

float speed; //블럭이 내려오는 속도 1이면 1초마다 한칸씩 내려옴

int board\_org[BOARD\_Y][BOARD\_X]; //게임판의 정보를 저장하는 배열 모니터에 표시후에 board\_cpy로 복사됨

int board\_cpy[BOARD\_Y][BOARD\_X]; //maincpy는 게임판이 모니터에 표시되기 전의 정보를 가지고 있음 //main의 전체를 계속 모니터에 표시하지 않고 board\_cpy와 배열을 비교해서 값이 달라진 곳만 모니터에 고침

int item; // 아이템

int target; // 공격 대상[[7]](#footnote-7)

KeyFlag m\_KeyFlag; // 키 플레그

GameFlag m\_GameFlag; // 게임 상태 플레그

};

struct MatchSockets {

SOCKET client[MAX\_PLAYER]; //matchmaking에서 gameserver로 보내줌

};

struct ClientGameData {

// 서버에서 업데이트용으로 쓰는것과 별개로 보낼 때 최대한 줄일 수 있는 패킷들을 줄인 구조체

int bx, by; //이동중인 블록의 게임판상의 x,y좌표

int b\_type; //블록 종류

int b\_rotation; //블록 회전값

int b\_type\_next; //다음 블록값

int board\_org[BOARD\_Y][BOARD\_X] = { 0, }; //게임판의 정보를 저장하는 배열 모니터에 표시후에 main\_cpy로 복사됨

int AttackedBlock = 0;

int item = -1; // 0 키 반전

// 1 상대 일시적 스피드 업

// 2 서로간의 블록 판을 바꿈

int target = 1;

bool gameover\_flag = 0; // 게임오버가 됬을 때 알려주는 flag

bool win\_flag = 0; // 이겼을 때 알려주는 flag

bool screen\_rotate\_flag = 0; // 스크린 돌아가는것을 알려주는 flag

bool speedup\_flag = 0; // 스크린 돌아가는것을 알려주는 flag

};

struct KeyInput {

bool left = false; //←

bool right = false; //→

bool up = false; //spin

bool down = false; //빠르게 내리기

bool space = false; //hard drop space(한번에 맨 밑으로 내리기)

bool shift = false; // 타겟 변경

bool ctrl = false; // 아이템 사용

bool enter = false; // 게임 오버시 종료키

};

struct KeyFlag {

bool left\_flag = 0; // 하드드랍할때 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool right\_flag = 0; // 위키 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool down\_flag = 0; // 위키 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool space\_flag = 0; // 하드드랍할때 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool up\_flag = 0; // 위키 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool shift\_flag = 0; // 쉬프트키 꾹 누르고 있어도 한번만 적용되게 해주는 flag

bool ctrl\_flag = 0; // 컨트롤키 꾹 누르고 있어도 한번만 적용되게 해주는 flag

// 위 키 플레그들은 조작감을 위해 서버 업데이트 시 사용

};[[8]](#footnote-8)

Struct GameFlag{

bool gameover\_flag = 0; // 게임오버가 됬을 때 알려주는 flag

bool win\_flag = 0; // 이겼을 때 알려주는 flag

bool screen\_rotate\_flag = 0; // 스크린 돌아가는것을 알려주는 flag

float fScreenRotateTime = 0.0f; // 스크린이 몇 초 돌아갓는지 알려주는 변수

bool speedup\_flag = 0; // 스크린 돌아가는것을 알려주는 flag

float fSpeedUpTime = 0.0f; // 스피드 업이 몇초 됬는지 알려주는 변수

bool new\_block\_on = 0; //새로운 블럭이 필요함을 알리는 flag

bool crush\_on = 0; //현재 이동중인 블록이 충돌상태인지 알려주는 flag

}

1. **팀원들의 역할 분담**

* 김호진: Server main Thread, MatchMaking Thread, 리소스 제작 및 가공
* 양현석: client socket프로그래밍, client Rendering, client scene 제작
* 윤장혁: Game Server Thread, CommThread제작

1. **일정[[9]](#footnote-9)**

테이블이(가) 표시된 사진

자동 생성된 설명

테이블이(가) 표시된 사진

자동 생성된 설명[[10]](#footnote-10)

1. **일일 보고서**

1. 2021년 11월 3일 수정 [↑](#footnote-ref-1)
2. 2021년 11월 3일 수정 [↑](#footnote-ref-2)
3. 2021년 12월 01일 수정(임계영역 생성 및 각 클라이언트 별 이벤트 생성) [↑](#footnote-ref-3)
4. 2021년 11월 3일 수정 [↑](#footnote-ref-4)
5. 2021년 12월 01일 수정(gamestatus 업데이트용 함수 추가) [↑](#footnote-ref-5)
6. 2021년 11월 3일 수정 [↑](#footnote-ref-6)
7. 2021년 11월 3일 수정 [↑](#footnote-ref-7)
8. 2021-11-21 추가 [↑](#footnote-ref-8)
9. 2021년 11월 3일 수정 [↑](#footnote-ref-9)
10. 2021년 12월 01일 일정표 수정 [↑](#footnote-ref-10)