|  |
| --- |
| 네트워크 게임 프로그래밍  팀프로젝트 3팀  2016180014 김지성  2017184010 류성호  2019182038조소현 |

**<목차>**

**1. 애플리케이션 기획**

**1) 게임 컨셉**

**2) 게임 소개**

**2. High-Level 디자인**

**1) 클라이언트와 서버 통신 구조**

**2) 로비 구조**

**3) 인 게임 구조**

**3. Low-Level 디자인**

**1) 네트워크 라이브러리**

**2) 클라이언트**

**3) 서버**

**4. 개발 환경**

**5. 팀원 별 역할분담**

**6. 개발 일정**

**1. 애플리케이션 기획**

**1) 게임 컨셉**

탑 뷰형 미로 탈출 게임으로 1명의 태거(Tagger)가 2명의 러너(Runner)를 잡거나 러너가 키를 모아서 최종적으로 미로를 탈출하는 게임입니다.

**2) 게임 소개**

텍스트, 전자기기, 스크린샷, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

플레이어는 미로의 랜덤한 지점에서 시작하고, 시작하는 순간 자신의 역할(태거, 러너)이 정해집니다.

**• 로그인 & 준비**

텍스트, 모니터, 스크린샷, 전자기기이(가) 표시된 사진

자동 생성된 설명

게임을 시작하게 되면 플레이어는 로그인을 할 로그인 화면을 만나게 되며, 엔터를 치면 서버에 자동으로 아이디가 보내져 게임에 접속하게 됩니다.

로그인 화면에서 엔터를 통해 대기 장소로 이동하며, 게임에 접속한 인원을 확인할 수 있습니다. 게임은 4명이 모두 접속할 경우 시작하게 되며, 이 때 역할(러너, 태거)가 정해집니다.

3명이 접속하면 바로 게임이 시작하게 되며, 게임 시작 데이터가 보내진다.

**• 플레이어의 역할**

플레이어의 역할 종류: **태거**, **러너**

플레이어는 좌우 방향키로 캐릭터가 회전하며, 앞에 블록이 없을 경우 자동으로 움직입니다.

**태거**: 3명중 1명이 태거로 결정되며, 태거는 러너 2명을 게임이 끝나기 전에 모두 잡아야만 승리합니다. 태거는 러너에게 접근하여 공격하며, 접촉하면 25만큼의 데미지를 주고, 러너의 HP가 0이 되면 러너를 잡을 수 있습니다.

**러너**: 3명중 2명이 러너로 결정되며, 러너는 맵 속에 숨겨져 있는 키 3개를 찾아서 최종 문을 열고 나가면 승리합니다. 러너는 태거에게 접촉하게 될 경우 도망치기 위하여 일시적으로 이동 속도가 증가하게 됩니다.

**• 맵(Map)**

맵은 최초 설계된 미로를 서버에서 받아와 플레이하게 되며, 형태는 지정되어 있습니다.

맵에는 비드(Bead), 키(Key), 벽(Wall), 탈출구(Door)가 존재하며, 크기 30x30의 미로입니다.

**• 비드(Bead)**

미로의 길에 존재하며, 러너만 먹으며, 먹힌 비드는 사라지게 되어 왔던 길을 표시하기도 하고, 점수의 역할을 하게 됩니다.

태거는 비드의 유무를 통해 러너가 왔다 갔음을 확인할 수 있습니다.

**• 키(Key)**

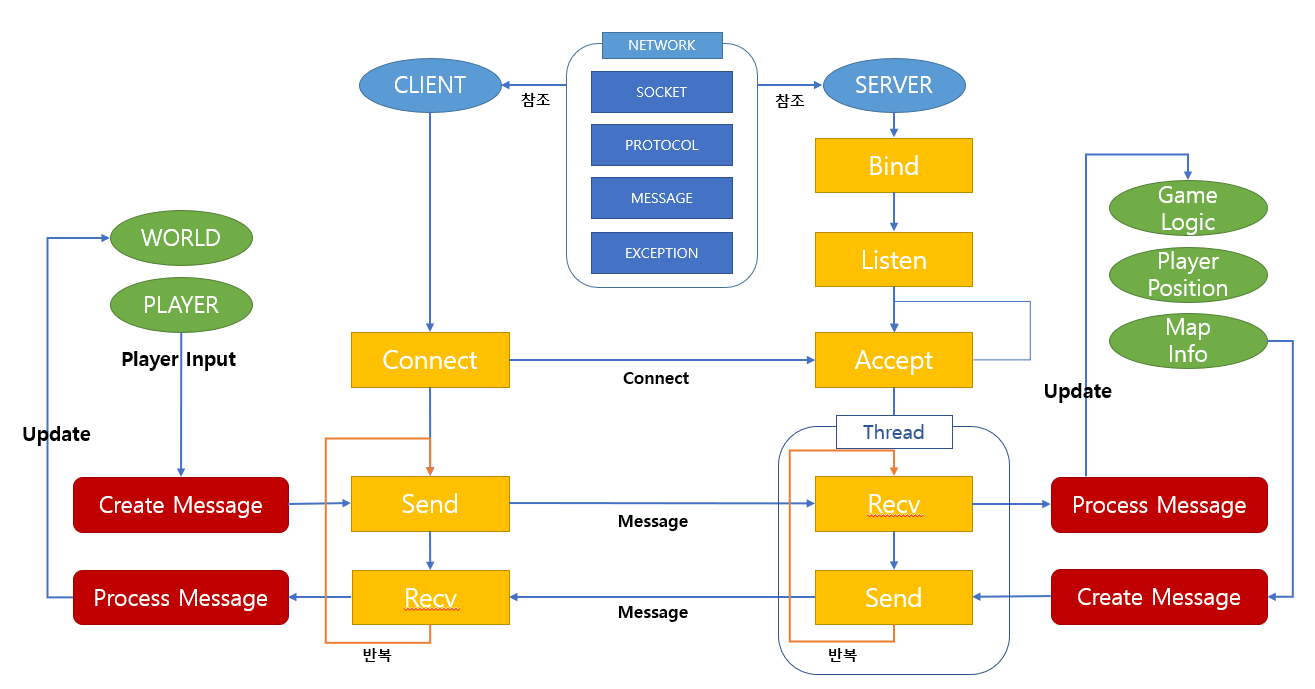
키는 3개가 존재하며, 미로 내 랜덤 한 위치에 등장하게 됩니다.

각 키는 최종 문을 여는 열쇠가 되어 먹는 순간 해당 자물쇠가 해금됩니다.

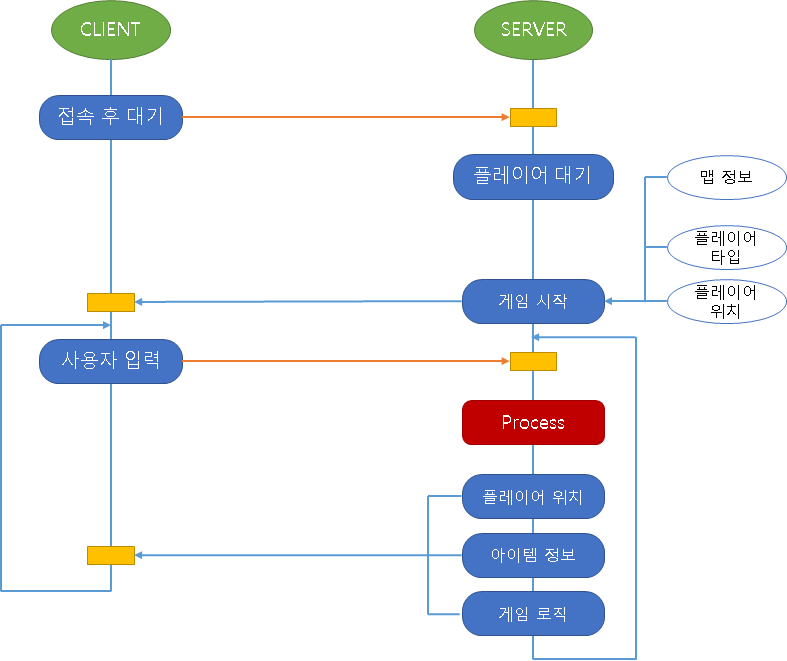
**• 문(Door)**

최종 탈출구로써 맵의 가장자리에 설치되어 있으며, 3개의 키가 모여야만 열립니다.

**2. High Level Design**

 **1) Client, Server 통신 구조**

**2) 프로그램 구조**



**3) 인 게임 구조**

텍스트이(가) 표시된 사진

자동 생성된 설명

**3. Low Level**

TCP와 멀티쓰레드를 활용한 모델

**1) Network Library : 서버와 클라이언트가 모두 공유할 수 있는 라이브러리**

**• Class Socket:**

**: 기본적인 소켓 작업을 담당하는 클래스**

**변수**

- SOCKET m\_socket : 소켓 핸들

- char m\_recvBuffer[MaxBufferSize] : Recv에서 데이터를 받아와 저장하는 버퍼

- Message m\_recvMsg : 링버퍼로 m\_recvBuffer의 데이터를 차례로 저장

**함수**

- void Bind(short ServerPort)

: 포트 번호로 소켓을 바인딩한다.(서버)

- void Listen()

: 접근하는 호스트를 받을 준비를 한다.(서버)

- void Connect(const char\* ServerAddress, short ServerPort)

: 서버 주소로 연결을 시도한다.(클라이언트)

- Socket Accept()

: 클라이언트를 연결하고 소켓 정보를 반환한다.

- void Send(Message& msg)

: 메시지를 상대 호스트에게 전송한다.

- void Recv()

: 상대 호스트가 보낸 메시지를 recvMessage에 저장한다.

**- void Init()**

**: m\_socket을 IPv4로 선언하고, 네이글 알고리즘을 해제한다.**

**• Protocol.h**

**: 어플리케이션 프로토콜과 서버 주소 정보를 담고 있다.**

- const char\* const SERVER\_IP : 서버 IP주소

- const short SERVER\_PORT : 서버 포트 번호

**- const int MaxBufferSize = 1024;** : 버퍼의 최대 크기

**- const int MaxClients = 3;** : 접속할 수 있는 클라이언트 수

**- const int MapSize = 30;** : 맵의 사이즈

- enum class MsgType : char : 메시지를 식별할 수 있는 메시지 형식

{

  LOGIN\_REQUEST, : 클라에서 서버로 로그인 요청

    LOGIN\_OK, : 서버에서 클라로 로그인 승인

    START\_GAME, : 게임 시작 시 필요한 데이터 전송

    PLAYTER\_INPUT, : 클라에서 입력 받은 키 값 전송

    UPDATE\_PLAYER\_INFO, : 플레이어의 위치정보를 모든 클라에 전송

UPDATE\_PLAYER\_POS, : UPDATE\_PLAYER\_INFO로 통합

PLAYER\_JOIN, : LOGIN\_OK에서 같이 해결

    UPDATE\_BEAD, : UPDATE\_STATUS로 통합

    UPDATE\_KEY, : UPDATE\_STATUS로 통합

    DOOR\_OPEN, : UPDATE\_STATUS로 통합

**UPDATE\_STATUS : BEAD, KEY, OPEN의 정보를 보냄**

}

**- enum class PlayerType : char : 플레이어 타입을 지정해주는 공용체**

**{**

**TAGGER, : 태거**

**RUNNER : 러너**

**};**

**enum class WinStatus : char : 승리 상태를 보내주는 공용체**

**{**

**NONE, : 게임 진행**

**RUNNER\_WIN, : 러너 승리**

**TAGGER\_WIN : 태거 승리**

**};**

**- enum class ObjectType : char :오브젝트 타입을 보내주는 공용체**

**{**

**BEAD, : 비드**

**KEY, : 키**

**DOOR, : 탈출구**

**WALL, : 벽**

**NONE, : 아무것도 없음**

**PLAYER, : 플레이어**

**TEXTURE, : 매핑**

**POWERBEAD, : 키**

**BOTTOM, : 바닥**

**ROAD, : 길**

**};**

- **enum class Direction : char : 방향을 보내주는 공용체**

**{**

**UP, : 상**

**DOWN, : 하**

**LEFT, : 좌**

**RIGHT, : 우**

**NONE, : 이동하지 않음**

**};**

**- struct Vector4 : 바운딩 박스를 표현하기 위한 구조체**

**{**

**float MaxX; : 바운딩 박스 좌표의 최대 x 위치**

**float MinX; : 바운딩 박스 좌표의 최소 x 위치**

**float MaxZ; : 바운딩 박스 좌표의 최대 z 위치**

**float MinZ; : 바운딩 박스 좌표의 최소 z 위치**

**};**

- struct login\_request { short size, MsgType type }

: 서버에 로그인 요청을 보내면 서버는 로그인을 허용할 지를 판단한다.

- struct login\_ok { short size, MsgType type, char id, char mapinfo[50][50] }

: 클라이언트에 로그인 수락과, 할당한 id, 맵 정보를 보내면 클라이언트는 이를 통해 필요한 정보를 초기화한다.

- struct player\_join { short size, MsgType type, char totalPlayers }

: 플레이어가 추가되면 총 플레이어의 수를 보내어 대기 화면을 업데이트한다.

- struct start\_game : 게임을 시작할 때 모든 정보를 담기 위한 구조체

{

short size; : 구조체 사이즈

MsgType type; : 해당 메시지 타입 : START\_GAME

char id[3]; : 클라이언트 아이디

PlayerType playertype[3]; : 플레이어의 타입(태거, 러너)

float x[3]; : 플레이어들의 최초 x 위치

float z[3]; : 플레이어들의 최초 z 위치

**char my\_id;** : 자신의 아이디

**char mapinfo[30][30];** : 맵 정보

}

- struct player\_input : 클라에서 입력받은 값을 저장하는 구조체

{

short size; : 구조체 크기

MsgType type; : 해당 메시지 타입 PLAYER\_INPUT

char input; : 입력받은 정보 전달 Left : 0 , Right : 1

float x; : 현재 x 위치

float z; : 현재 z 위치

}

- struct update\_player\_pos { short size, MsgType type, char id[4],

float x[4], float z[4] }

: 서버는 지속적으로 클라이언트들의 위치를 갱신하여 보낸다.

- struct update\_player\_info : 변한 플레이어의 정보를 저장하는 구조체

{

short size; : 구조체 사이즈

MsgType type; : 해당 메시지 타입 UPDATE\_PLAYER\_INFO

char id[3]; : 모든 플레이어의 아이디

char hp; : HP 제거

short score; : 점수 제거(비드 개수를 점수로 보기 때문에)

**float x[3];** : 변화된 플레이어들의 x좌표

**float z[3];** : 변화된 플레이어들의 z좌표

}

- struct update\_object { short size, MsgType type, ObjectType type, char id, bool visible }

: 맵에 존재하는 오브젝트들의 변화되어 사라졌는지, 그 상태를 보낸다.

**- struct object\_status** : 오브젝트들의 변화된 상태를 저장하는 구조체

**{**

**ObjectType objType;** : 무슨 오브젝트인지 파악하기 위한 ObjectType 공용체

**char row, col;** : 오브젝트의 인덱스 위치

**bool active;** : 오브젝트 활성화 여부

**};**

**struct update\_status** : 승리 확인 및 플레이어 생사 여부를 판단하는 구조체

**{**

**short size;** : 구조체의 사이즈

**MsgType type;** : 해당 메시지 타입 UPDATE\_STATUS

**WinStatus win;** : 누가 이겼는지 알려주는 공용체

**bool player\_active[3];** : 플레이어의 생사 여부

**};**

**• CommonInfo.h // Protocol.h 로 통일**

**- struct ObjectInfo** : 오브젝트의 정보들을 저장하는 구조체

**{**

**ObjectType type;** : 오브젝트타입을 알려주는 공용체

**char row, col;** : 오브젝트들의 위치 인덱스를 저장

**float x, z;** : 오브젝트들의 변화되어 있는 위치를 저장

**bool active;** : 오브젝트의 활성화 여부

**float boundingOffset;** : 오브젝트의 바운딩 오프셋

**Vector4 GetBoundingBox()** : 오브젝트 바운딩박스 상자 정보 반환

**{**

**Vector4 box =**

**{**

**x + boundingOffset,**

**x - boundingOffset,**

**z + boundingOffset,**

**z - boundingOffset**

**};**

**return box;**

**}**

**};**

- **struct MapInfo** : 맵에 존재하는 오브젝트들의 정보를 담는 구조체

**{**

**vector<ObjectInfo> beads;** : 비드 정보를 담는 오브젝트 벡터

**vector<ObjectInfo> keys;** : 키 정보를 담는 오브젝트 벡터

**vector<ObjectInfo> walls;** : 벽 정보를 담는 오브젝트 벡터

**ObjectInfo door;** : 최종 문을 담는 오브젝트

**};**

**• Class Message**

**: 여러 메시지를 하나의 버퍼로 구성하고, 여러 메시지로 다시 분리하는 클래스.**

**변수**

**-** const int MaxBufferSize : 최대 버퍼의 크기

**-** char msgBuffer[MaxBufferSize] : 메시지 버퍼

**-** int beginOffset : 버퍼에서 메시지가 시작하는 곳의 인덱스

**-** int endOffset : 버퍼에서 메시지가 끝나는 곳의 인덱스

Private:

- **int m\_readIndex;** : m\_buffer에서 읽을 위치를 나타내는 인덱스

**- int m\_writeIndex;** : m\_buffer에서 쓸 위치를 나타내는 인덱스

**- int m\_remainSize;** : 버퍼에 남은 사이즈

Public:

- char m\_buffer[MaxBufferSize]; : 메시지의 저장 버퍼

**함수**

- void CopyData(void\* msg, int size)

: 메시지를 size만큼 버퍼로 복사하는 함수. endOffset이 증가함.

- void ReleaseData(void\* msg)

: 버퍼의 있는 데이터를 하나의 메시지에 복사하는 함수. beginOffset이 증가함.

**- void Clear();**

**: 메시지 버퍼를 비우는 함수**

**- void Push(char\* msg, int size);**

**: m\_buffer에 size만큼 m\_buffer에 저장**

**- void Pop(char\* msg, int size);**

**: m\_buffer에서 size만큼 msg에 copy**

**- bool IsEmpty();**

**: 비어있는지 확인하는 함수**

**- bool IsFull();**

**: 버퍼가 가득 찼는지 확인하는 함수**

**- MsgType GetMsgType();**

**: 메시지 타입을 알려주는 함수**

**- short GetTotalMsgSize();**

**: 메시지 크기를 알려주는 함수**

**- short PeekNextPacketSize();**

**: 다음 패킷의 사이즈를 알려주는 함수**

**• Class Exception ( std::exception 상속 )**

**: 네트워크 함수에서 에러가 발생할 시 오류를 던질 수 있는 클래스.**

**변수**

- const char\* ErrorString : 오류 정보를 담는 문자열

**함수**

- Exception(const char\* error\_code)

: 에러코드를 받아 에러 정보로 변환해주는 생성자

- virtual const char\* what() const override;

: exception 클래스 오버라이드 함수로서 오류 메시지를 반환한다.

**• Class Timer**

**: 컴퓨터 내에서 시간이 흘러가는 시간을 알려주는 클래스**

**변수 :**

**- clock::time\_point m\_start; : 시간의 시작을 알리는 포인트 지점**

**- clock::time\_point m\_prev; : 프로그램의 시작 전을 알리는 포인트 지점**

**- clock::time\_point m\_curr; : 프로그램의 끝을 알리는 포인트 지점**

**- float m\_elapsed; : 프로그램이 한번 돌아가는데 걸리는 시간 저장**

**함수 :**

**- void Start();**

**: Timer를 시작시키는 함수**

**- void Tick();**

**: m\_elapsed를 구하는 함수**

**- float GetElapsedTime() const;**

**: m\_elapsed를 반환하는 함수**

**- float GetTotalTime() const;**

**: 흘러간 총 시간을 반환해주는 함수**

**2) Client**

**• Struct PlayerInfo**

**: 플레이어마다 필요한 정보를 저장한다.**

**변수**

- char id : 플레이어의 id

- float x : 플레이어 x 좌표

- float z : 플레이어 z 좌표

- PlayerType type : 플레이어의 타입(TAGGER, RUNNER)

- Direction dir : 플레이어의 현재 입력

**• Class Client (Socket 상속) // Network Manager를 참조**

**: 클라이언트가 서버와의 상호작용을 위해 필요한 작업을 하는 클래스이며, 소켓을 상속한다.**

**변수**

- WSAData wsaData : 윈도우 소켓 API용 구조체

- array<ClientInfo, 4> others : 모든 플레이어들의 정보

- std::thread netThread : 네트워크 작업을 위한 쓰레드.

- int hp : 나의 현재 체력

- int score : 나의 현재 점수

**함수**

- void Initialize()

: 소켓 정보를 초기화하고 서버에 연결한다.

- void SendLoginPacket()

: 서버에 로그인 패킷을 보낸다.

- void SendPlayerInput()

: 서버에 플레이어의 입력을 보낸다.

- void Update()

: 서버에 패킷을 보내고 서버로부터 패킷을 받아 처리하는 일을

반복하는 쓰레드 함수.

- void ProcessMessage(char\* packets)

: 각각의 패킷을 전달받아 메시지 형식에 따라 적절한 작업을 수행한다.

**• Class NetworkManager**

**: 클라이언트에서 네트워크 일을 담당하는 클래스**

**변수 :**

**- static NetworkManager\* instance; : 싱글톤**

**- Socket s\_socket; : 클라이언트 소켓**

**- Message m\_sendMsg; : Send 메시지**

**- bool isConnected = false; : 연결되었는지 확인하는 변수**

**- MsgType msgtype; : 받아온 메시지 타입 변수**

**- char myID; : 자신의 ID**

**- char last\_input = -1;**

**: 최근 입력받은 방향(처음엔 입력받지 않아 -1로 초기화)**

**- PlayerInfo players[MaxClients]; : 플레이어들 정보**

**함수:**

**- void Network();**

**: 네트워크 연결, 맵 정보 / 플레이어 정보 초기화 / 업데이트 반복 호출**

**- void Update();**

**: 서버에서 보낸 데이터를 받아서 클라이언트에서 처리해주는 함수**

**- bool GetIsConnected();**

**: 연결 되었는지 확인하는 함수**

**- int GetMyID();**

**: 자신의 ID를 반환해주는 함수**

**- void SendPlayerInput();**

**: 입력받은 키에 따라서 player\_input정보를 서버에 보내주는 함수**

**- PlayerInfo GetPlayerInfo(int id);**

**: 플레이어 정보를 받아오는 함수**

**- void SetLastInput(char input);**

**: 최근 입력받은 키 저장하는 함수**

**3) Server**

**• Class ClientInfo (Socket 상속)**

**: 서버에서 관리하는 클라이언트의 정보들을 담는다.**

**변수**

- char m\_id : 플레이어 ID

- PlayerType m\_type : 플레이어의 타입

- int m\_hp : 플레이어의 체력

- float m\_pos\_x : 플레이어 x좌표

- float m\_pos\_z : 플레이어 z좌표

- float boundingOffset : 바운딩 박스를 만들기 위한 offset

- Message m\_sendMsg : 클라이언트에 보내는 메시지

- Direction m\_direction : 플레이어의 현재 입력

- int score : 플레이어의 현재 점수

**- bool m\_active : 오브젝트의 활성화 상태**

**- mutex m\_sendLock : 메시지 동기화 LOCK**

**- mutex m\_directionLock : 방향 동기화 LOCK**

**- Direction m\_direction : 플레이어 방향**

**함수**

- Vector4 GetBoundingBox()

: boundingOffset으로 플레이어 바운딩 박스의 정보를 반환한다.

- bool IsCollided(int x, int y, start\_game& s\_game)

: 플레이어와 벽이 충돌했는지 판정해주는 함수

- void ProcessPacket(char\* packet)

: 클라이언트가 전달한 메시지를 처리하는 함수

- void CheckObjectStatus()

: 오브젝트와의 충돌을 판별하고 해당 오브젝트의 타입에 따라 오브젝트 정보를 갱신하는 함수

- pair<float, float> GetNewPosition()

: 플레이어의 새로운 위치를 계산하여 반환한다.

- void CreateLoginOkAndMapInfoMsg()

: 로그인 승인과 맵 정보를 메시지를 생성하는 함수

**- void Init(SOCKET sck, int id);**

**: id와 socket 초기화**

**- void SendMsg();**

**: m\_sendMsg 를 클라이언트로 보내는 함수**

**- void ProcessMessage();**

**: 클라이언트에서 입력 받아 보낸 값을 받아 처리하는 함수**

**- void ChangeDirection(player\_input& p\_input);**

**: 플레이어의 방향을 바꿔주는 함수**

**- void SetNewPosition(start\_game& s\_game, float elapsedTime);**

**: 플레이어의 새로운 위치를 설정하는 함수**

**• Class Server**

**변수**

**Private:**

- WSAData m\_wsaData : 윈도우 소켓 API 구조체

- SOCKET m\_listenSocket : 리슨 소켓

- static int m\_countOfKeyAccquired : 클라이언트가 얻은 키의 총 개수

- array<Client, 4> clients : 4개의 클라이언트 정보를 담는 배열

- array<std::thread, 4> threads : 4개의 클라이언트 작업용 쓰레드

- static mutex mapInfoLock : 맵 정보를 동기화하기 위한 뮤텍스 객체

- static mutex countOfKeyLock : key 개수를 동기화하기 위한 뮤텍스 객체

**- vector<thread> m\_threads; : 쓰레드**

**- update\_player\_info m\_player\_info; : 플레이어 정보**

**- update\_status m\_update\_info; : 업데이트 정보**

**- vector<object\_status> m\_object\_info; : 오브젝트 정보**

**- start\_game m\_startGameData; : 게임 시작 초기화 데이터**

**- int m\_taggerIndex = 0; : 태거**

**Public:**

- static MapInfo g\_map; : 현재 맵 정보로서 지속적으로 갱신됨.

**- static array<ClientInfo, MaxClients> g\_clients; : 클라이언트 정보 저장**

**- static mutex g\_timerLock; : timer 이벤트 변수에 필요한 LOCK**

**- static condition\_variable g\_timerCv; : 타이머 틱 동기화하는 이벤트 변수**

**- static Timer g\_timer; : 서버 타이머**

**- static float g\_accum\_time; : 서버에서 누적 시간**

**- static mutex g\_loopLock; : g\_loopCv 이벤트 변수에 필요한 LOCK**

**- static condition\_variable g\_loopCv; : 게임 시작 동기화하는 이벤트 변수**

**- static bool g\_loop; : 게임 루프**

**함수**

- void LoadMap(const char\* filename)

: 텍스트 파일로부터 맵 정보를 로드한다.

- void Update()

: 메인 쓰레드의 작업: Accept처리, 쓰레드 할당, 맵 정보 업데이트 등

- void AcceptNewPlayer()

: 새로운 클라이언트를 연결하고 쓰레드를 할당한다.

- static void RecvAndSend(int id)

: 클라이언트(id)와의 통신을 위한 쓰레드 함수

- void GameStart()

: 게임이 시작되었을 떄, 게임 시작 데이터 보내주는 함수

- void CreatePlayerJoinMsg()

: 새로운 클라이언트가 들어왔을 때 접속해 있는 클라이언트들에게

총 플레이어의 수를 전달할 메시지를 생성하는 함수

**- void CreateUpdateStatusMsg ()**

**: 변화된 게임 정보를 클라이언트에게 보내주는 함수**

**- void InitializeStartGameInfo();**

**: m\_startGameData 초기화, 플레이어 정보 초기화 해주는 함수**

**- void CreatePlayerInfoMsg(float elapsedTime);**

**: 플레이어의 위치를 갱신하여 저장하는 함수**

**- void CopySendMsgToAllClients();**

**: 모든 클라이언트에게 변화된 정보를 m\_SendMsg에 복사하는 함수**

**- vector<object\_status> UpdateObjectStatus(int id);**

**: 비드랑 키가 플레이어와 충돌해서 변화되었는지 확인하여 반환하는 함수**

**- bool CheckWinStatus(int id);**

**: 태거와 러너 승리상태 체크해주는 함수**

**- bool IsCollided(const Vector4& a, const Vector4& b);**

**: 바운딩박스 충돌 되었는지 반환해주는 함수**

**4. 개발환경**

**1) Visual Studio 2019**

**2) GitHub**

**3) OpenGL**

**5. 역할 분담**

|  |  |  |
| --- | --- | --- |
| **조소현** | **류성호** | **김지성** |
| **클라이언트 통신 메시지 처리, 렌더링 작업** | **네트워크 라이브러리, 충돌체크 및 이동 처리, 쓰레드 동기화** | **서버 통신 메시지 및 게임 로직 처리, 쓰레드 동기화** |

**6. 일정 계획표**



- 수정 된 일정 계획표