PORTFOLIO

이용석

E-MAIL. emtm95@naver.com
TEL. 010 - 3915 - 6040

INDEX

Be Anyone

게임 소개 – Be Anyone

장르
- MMO RPG

개발 환경
- Direct X 12
- Visual Studio 2019
- Git Hub
- Terragen4

개발 인원
- 클라이언트 2
- 서버 1

기획 기간
- 2020.09 ~ 2020.12

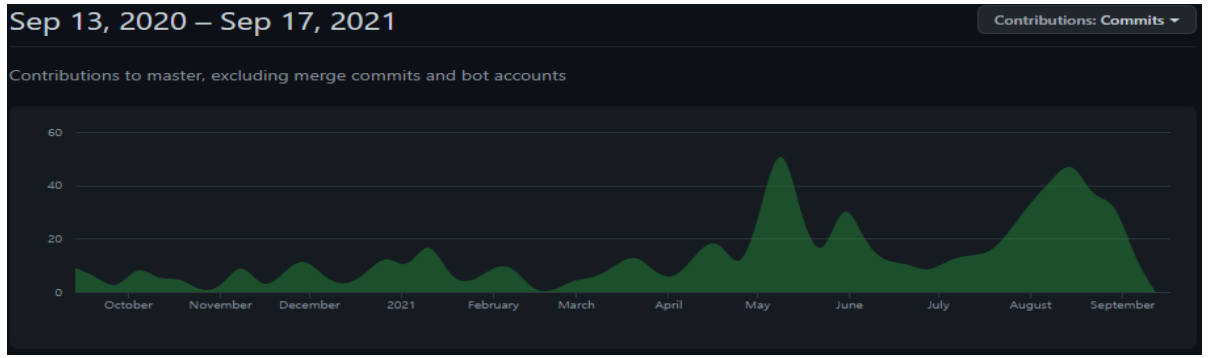
개발 기간
- 2021.01 ~ 2021.09

담당 업무
- IOCP를 사용한 멀티플레이 게임 구현
- 클라이언트 측 네트워킹
- 쿼드트리 공간분할 알고리즘을 사용한 시야처리
- 데드레커닝 (추측항법)
- 음향 기능 담당

협업

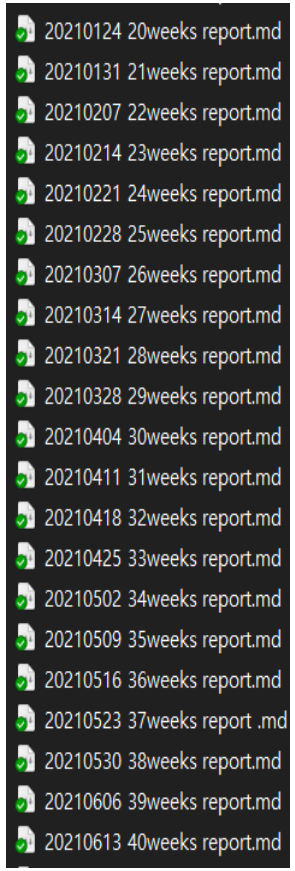
장기 프로젝트 관리

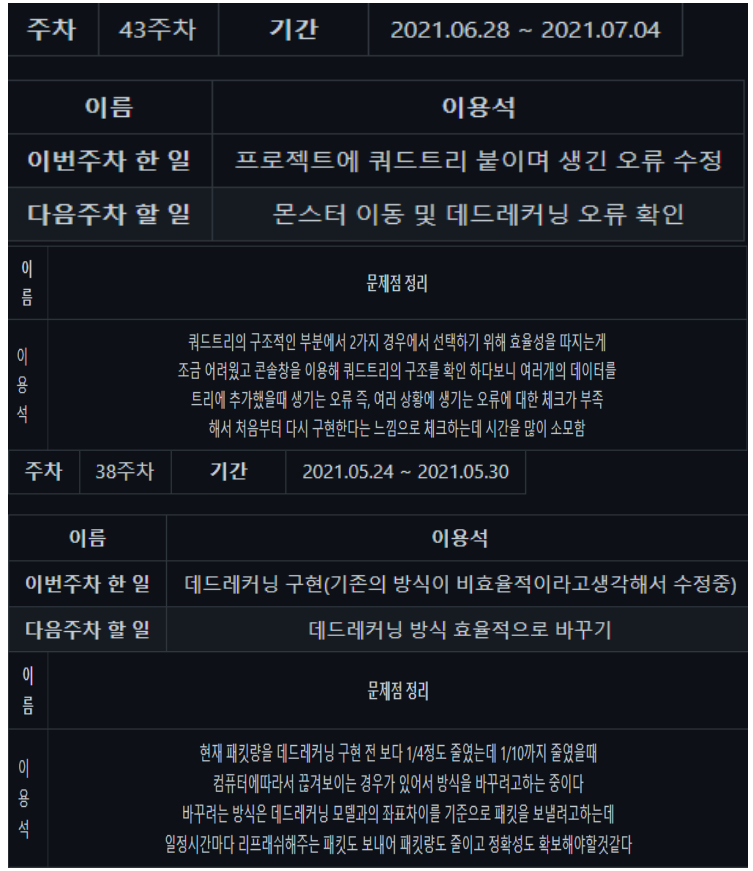
- 팀원들과의 효율적인 협업을 위해 Git Hub를 사용하여 프로젝트를 관리했습니다.
- 매 주 수요일이나 회의가 필요한 상황에 디스코드나 보이스톡을 사용하여 회의를 진행하고 한 주간의 작업, 다음 목표에 대해 공유했습니다.
- 매주 일요일 한주간의 작업내용, 작업 중 문제점, 다음 목표 등을 MarkDown으로 작성하여 공유했습니다.
- 프로젝트 팀장을 맡게 되어 회의록 작성과 팀원들의 작업 피드백, 목표 설정을 담당했습니다.



Git Hub : https://github.com/dydtjr0316/HellGate

Youtube : https://youtu.be/ESPotuZG0UY





객체 관리

```
typedef unordered_map<uShort, CGameObject*> OBJLIST;
typedef unordered_set<uShort> RECKONER;
typedef unordered_set<uShort> MONSTER_RECKONER;
class CMediatorMgr
{
private:
    OBJLIST             m_ObjectList;
    RECKONER            m_ReckonerList;
    MONSTER_RECKONER    m_MonsterReckonerList;
```

```
case CS_LOGIN: {
    cs_packet_login* packet = reinterpret_cast<cs_packet_login*>(buf) ;
    tempLock.lock();
    m_pSendMgr->Send_LoginOK_Packet(user_id);
    m_pMediator->ReckonerAdd(user_id);
    tempLock.unlock();

    Enter_Game(user_id, packet->name);
}
```

```
for (auto& reckoner : m_pMediator->GetReckonerList())
```
플레이어 이동, 시야처리, 그에 따른 패킷 처리를 해주는 코드

```
for (auto& monster : m_pMediator->GetMonsterReckonerList())
```
몬스터(보스 + 일반) 이동, 시야처리, 그에 따른 패킷 처리를 해주는 코드

- 모든 객체를 저장해 놓는 OBJLIST 변수
  - ✓ 미리 만들어 놓은 Player, Monster를 저장해 놓고 관리하는 변수
  - ✓ 순서가 필요 없고 탐색을 많이 해야 하고 ID와 객체를 저장하기 위해 unordered_map을 사용했습니다.
- 이동, 공격등의 Event를 진행중인 Player, Monster를 저장해 놓는 RECKONER, MONSTER_RECKONER변수
  - ✓ 순서가 필요 없고 ID만 저장해서 OBJLIST에서 찾아 사용하면 되므로 unoredered_set을 사용했습니다.
  - ✓ 불필요한(접속하지 않은 플레이어, 이동하지않는 몬스터) 객체에 대해 불필요한 검사를 하지 않기 위해 검사가 필요할 때만 Insert해 주고 사용하고 있습니다.
- Processing_thread 에서 RECKONER, MONSTER_RECKONER에 존재하는 ID를 사용하여 OBJLIST에서 해당 객체를 검색하여 객체에 대한 이동, 시야처리, 그에 따른 패킷 처리를 진행합니다.

IOCP & 멀티 스레드

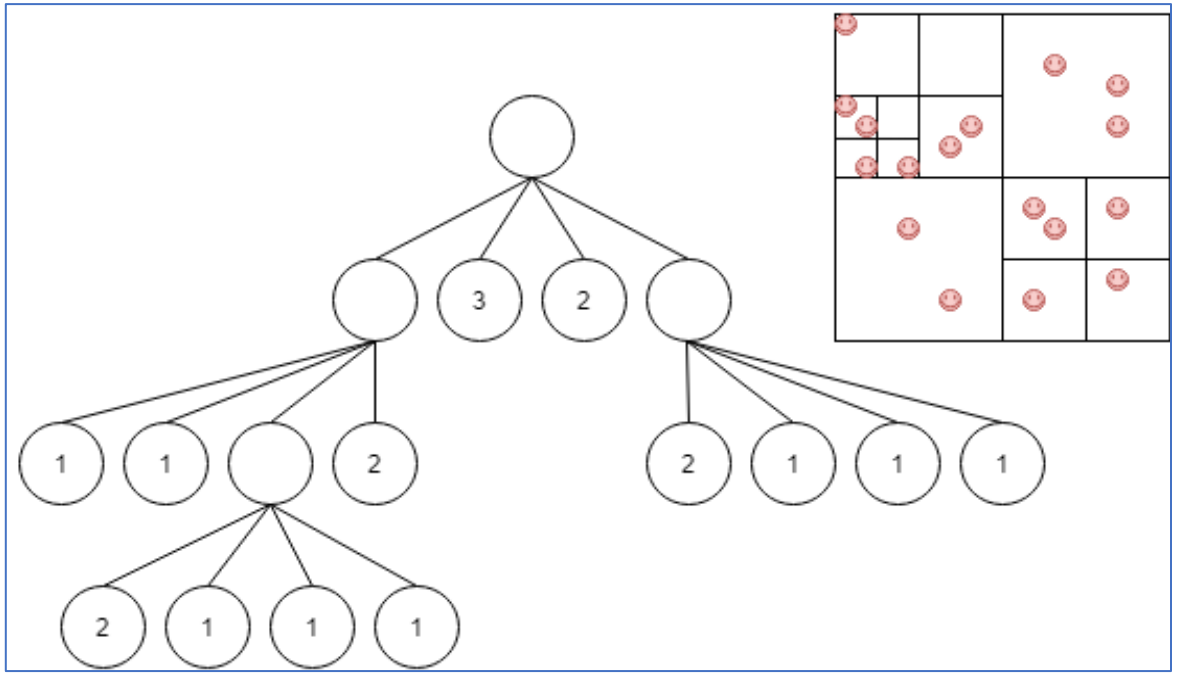
```
g_IocpHandle = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, NULL, 0);
g_listenSocket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
CreateIoCompletionPort(reinterpret_cast<HANDLE>(g_listenSocket), g_IocpHandle, 10000, 0);

SOCKADDR_IN s_address;
memset(&s_address, 0, sizeof(s_address));
s_address.sin_family = AF_INET;
s_address.sin_port = htons(SERVER_PORT);
s_address.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
::bind(g_listenSocket, reinterpret_cast<sockaddr*>(&s_address), sizeof(s_address));
listen(g_listenSocket, SOMAXCONN);

SOCKET c_socket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
EXOVER accept_over;

accept_over.op = ENUMOP::OP_ACCEPT;
accept_over.c_socket = c_socket;
accept_over.wsabuf.len = static_cast<int>(c_socket);
ZeroMemory(&accept_over.over, sizeof(accept_over.over));
AcceptEx(g_listenSocket, c_socket, accept_over.io_buf, 0, 32, 32, NULL, &accept_over.over);
```

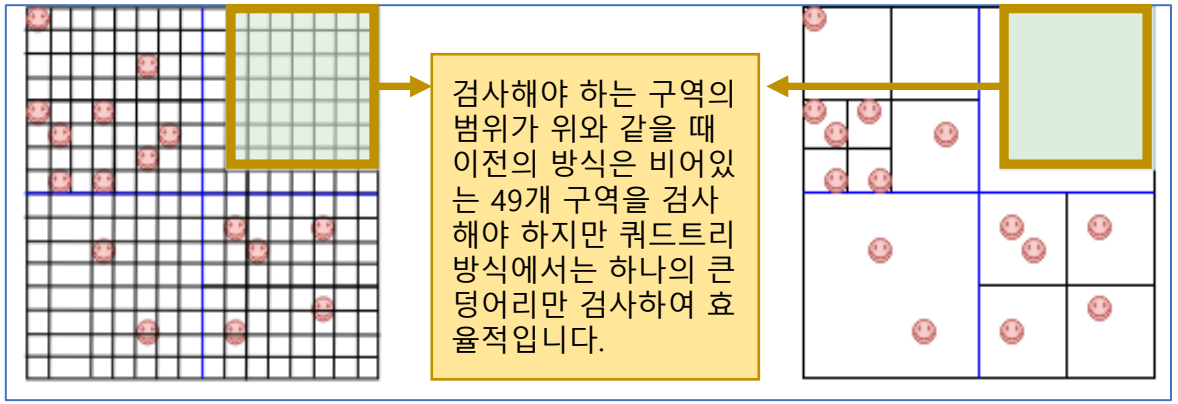
- 많은 동접자 수를 요구하는 MMO RPG 게임 장르 개발을 위한 IOCP 서버 구현

- 멀티코어를 효율적으로 사용하기 위한 멀티스레드 프로그래밍
  - ✓ 네트워킹을 담당하는 worker_threads
  - ✓ 이동, 시야처리, 시야처리에 따른 Enter, Leave패킷 전송을 담당하는 process_threads
  - ✓ 몬스터의 이동에 대한 명령을 처리해주는 timer_thread

```
thread time_thread(&CNetMgr::Timer_Worker, &Netmgr);
for (int i = 0; i < 7; ++i) worker_threads.emplace_back(thread(&CNetMgr::Worker_Thread, &Netmgr));
for (int i = 0; i < 2; ++i) process_threads.emplace_back(thread(&CNetMgr::Processing_Thead, &Netmgr));
for (auto& th : worker_threads) th.join();
for (auto& th : process_threads) th.join();
time_thread.join();
```

시야 처리(QuadTree)



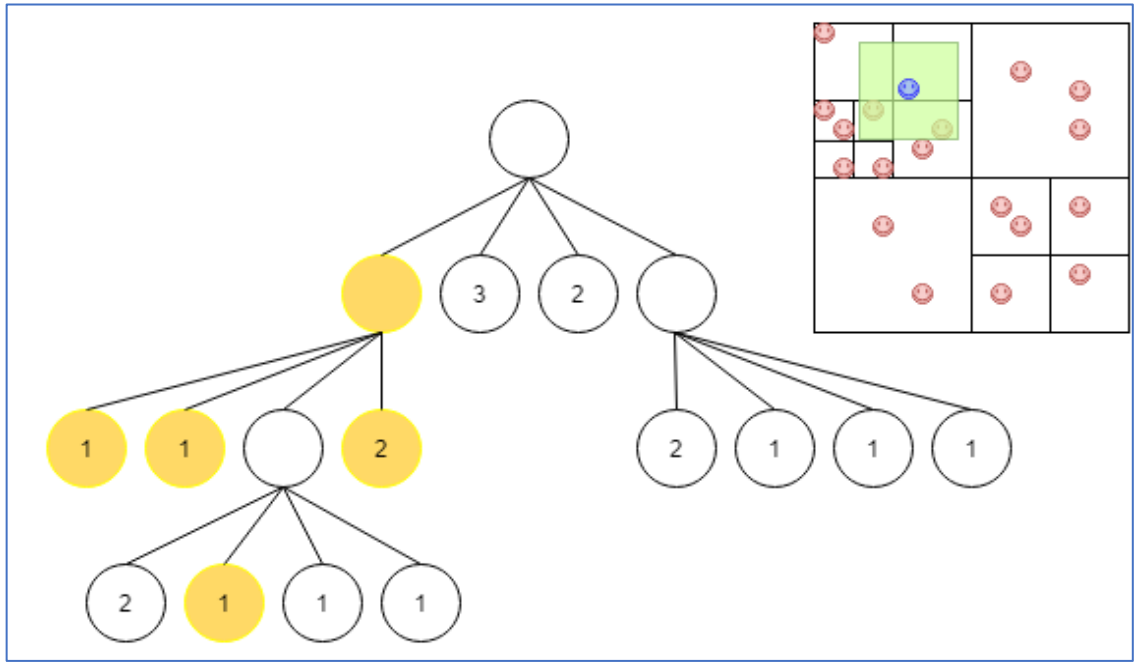


- 시야처리를 하지 않는다면 동접자가 N명일때 플레이어 한 명의 상태를 동기화 해주기위해 N * N의 패킷이 필요하게 되는데 이는 많은 동접자 확보가 중요한 MMO RPG 게임의 특성상 최적화 해야 한다고 생각해서 구현하게 되었습니다.

- 이전 프로젝트에서는 맵 전체를 격자 형태로 나누어 구현 했었는데 이런 방식에서는 유저가 없는 구역에 대해서도 검사를 하고 유저가 없는 구역이 많이 모여 있는 경우 비효율 적이라고 생각했습니다. 그래서 유저가 없는 구역에 대해서 큰 덩어리로 떼어내어 검사를 하지 않을 수 있는 QuadTree 알고리즘을 시야처리에 사용하게 되었습니다.

- QuadTree는 하나의 노드에 일정 수 이상의 플레이어가 존재하면 4개의 자식 노드로 나누고 부모에 있던 플레이어를 자식 노드에 넘겨주는 방식으로 구현했습니다.

시야 처리 (Search 알고리즘)



쿼드트리의 Search 알고리즘

```
switch (m_boundary.CheckIntersects(searchBoundary))
{
case CheckIntersect::NON:
    break;
case CheckIntersect::INTER:// 단순 걸치기만 하는 노드
    if (m_bisDivide) { ... }
    else { ... }
    break;
case CheckIntersect::CONTAIN:// 범위에 완전 포함되는 노드
    if (m_bisDivide) { ... }
    else { ... }
    break;
}
```

Contain인 경우

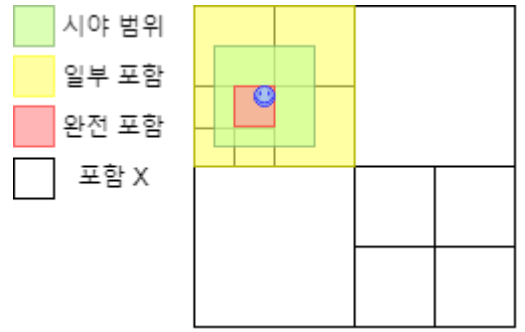
```
if (m_bisDivide)    // 자식이 있는 노드라면
{
    unordered_set<uShort> temp;
    for (auto& child : m_pChild)
    {
        temp = child->search_Contain(searchBoundary);
        found.insert(temp.begin(), temp.end());
    }
}
else
{
    found.insert(m_vpPlayers.begin(), m_vpPlayers.end());
}
```

- 시야처리를 진행하는 플레이어의 시야범위를 기준으로 포함되는 노드를 색출하여 필요한 노드의 플레이어들만 시야범위에 있는지 판단하여 필요 없는 연산을 줄였습니다.
- 왼쪽의 그림을 보시면 Root노드의 오른쪽 3개의 노드는 시야범위에 포함되지 않기때문에 Search시에 Pass하게 되어 해당 노드들의 플레이어에 대해서는 시야에 대한 검사를 할 필요가 없게 됩니다.
- Search 시에 시야범위와 노드의 범위 간 겹치는 정도에 따라 3가지 상황으로 나누어 그에 따른 최적의 검사를 진행하도록 했습니다.

✓ 시야에 완전 포함되는 노드

✓ 시야에 일부 포함되는 노드

✓ 시야에 포함되지 않는 노드



최적화 (시야처리)

```
unordered_set<uShort> view_list;
```

```
obj->GetLock().lock();
unordered_set<uShort>old_viewList = obj->GetViewList();
//시야처리 : Process Thread에서 움직이기 전에 시야에 있던 객체의 정보를 가져옴
if (!obj->GetBoundary().contains(newPos))
{
    g_QuadTree.Delete(obj);
    obj->SetPosV(newPos);
    g_QuadTree.Insert(obj);
}
else
    obj->SetPosV(newPos);

unordered_set<uShort> new_viewList = g_QuadTree.search(CBoundary(m_pMediator->Find(reckoner)));
// 시야처리 : 그러곤 새로운 위치에서의 시야에 포함되는 플레이어를 찾음
obj->GetLock().unlock();
```

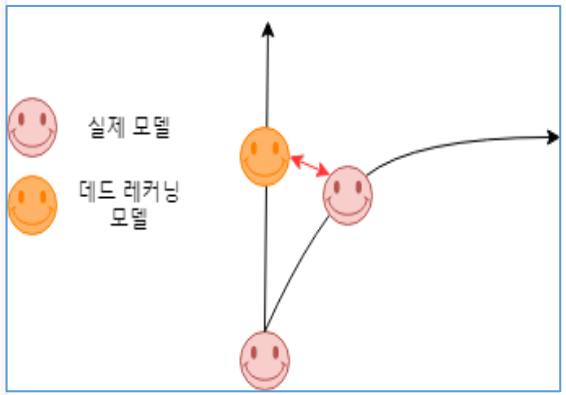
```
for (auto& ob : new_viewList) //시야에 새로 들어온 객체 구분
{
    if (ob == reckoner)continue;
    if (0 == old_viewList.count(ob)) // 새로 들어온 아이디
    {
        m_pSendMgr->Send_Enter_Packet(reckoner, ob);
        if (m_pMediator->IsType(ob, OBJECT_TYPE::CLIENT))
        {
            m_pSendMgr->Send_Enter_Packet(ob, reckoner);

        }
        obj->GetLock().lock();
        obj->GetViewList().insert(ob);
        m_pMediator->Find(ob)->GetViewList().insert(reckoner);
        obj->GetLock().unlock();
    }
    else // new, old 둘다 있을때
    {
        if (m_pMediator->IsType(ob, OBJECT_TYPE::CLIENT))
        {
            if (CAST_CLIENT(obj)->GetIsRefresh())
                m_pSendMgr->Send_Move_Packet(ob, reckoner, CAST_CLIENT(obj)->GetDir());
        }
    }
}
```

- 성능 프로파일러를 통해 가장 큰 부하를 차지하는 알고리즘이 QuadTree의 Search 알고리즘이라는 것을 확인했고 그에 따라 플레이어의 Position 값이 바뀌기 전 후에 search 함수를 실행하여 패킷 처리를 하는 방식에서 플레이어에 view_list 라는 변수를 추가하여 Position이 바뀌기 전 시야에 있던 플레이어를 저장하여 Search 함수 호출 빈도를 줄였습니다.

- View_list는 플레이어의 모든 정보를 저장할 필요가 없고 정렬이 필요 없기 때문에unordered_set을 사용하여 플레이어의 ID만 저장했습니다.

- View_list는 플레이어가 게임에 입장했을 때 search 알고리즘을 사용하여 Setting 해주고 이동 및 시야처리를 담당하는 Processing_thread에서 이동 전, 후의 시야를 비교 후 view_list에 삽입, 삭제를 진행했고 그에 따라 Enter, Leave, Move 패킷을 상황에 맞게 전송하였습니다.

데드 레커닝




```
if (packetMoving)
{
    if (FirstPacket)
    {
        temp = packetLocalPos + packetDirVec * packetspeed * (DT);
        SetBisFirst(false);
    }
    else
    {
        temp = playerTrans->GetLocalPos() + packetDirVec * packetspeed * (DT);
    }
```

```
CDeadReckoner*   m_pDeadReckoner;
```

```
class CDeadReckoner
{
private:
    Vector3 m_v3localPos;
    Vector3 m_v3DirVec;
    float m_fRotateY;
    float m_fSpeed;
```

Player Object는 데드레커닝 모델을 멤버변수로 가지고 있으며 데드레커닝 모델은 이동에 필요한 변수만을 가지고 있습니다.

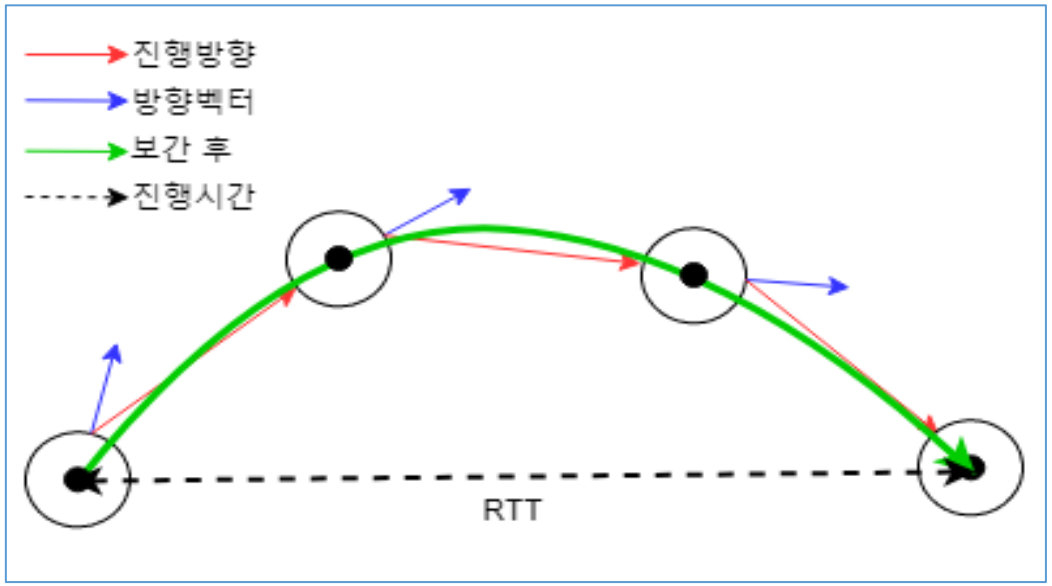
서버로 부터 Move 패킷을 받을 때 방향벡터와 속도, 위치를 갱신해 주고 다음 Move 패킷이 오기 전까지 해당 데이터로 데드레커닝을 진행 했습니다.

```
if (GetReckoner()->isFollowing() && moveKeyInput)// 일정거리이상 실제 모델과 멀어졌을때
{
    ReckonerMove = true;
    g_netMgr.Send_Move_Packet(dir, localPos, worldDir, vRot.y, system_clock::now(), DT, ReckonerMove);

    GetReckoner()->SetDirVec(worldDir);
    GetReckoner()->SetRotateY(vRot.y);
    GetReckoner()->SetLocalPos(GetObj()->Transform()->GetLocalPos());
    CountTime();
}
```

실제모델과 데드레커닝 모델의 위치가 멀어졌을 때 패킷을 전송 후 데드레커닝모델의 데이터를 실제모델의 데이터로 갱신해준다.

- 데드레커닝 모델과의 거리차를 통한 패킷 전송
  - ✓ 움직이는 플레이어의 위치, 방향 벡터 등이 동일한 데드레커닝 모델을 플레이어마다 생성하여 데드레커닝 모델과 실제모델의 좌표차이가 일정크기 이상 커졌을 때 데드레커닝의 정보를 실제모델과 일치하게 갱신해 주고 실제모델의 정보를 서버에 보내주어 서버에서 계산하는 해당 플레이어의 위치와 방향벡터를 업데이트해 주었습니다.
  - ✓ 데드레커닝 모델은 실제 모델의 방향을 따라 직진 하는 형태로 이동하기 때문에 실제모델이 회전 했을 때 데드레커닝 모델과 좌표차이가 발생하게 됩니다.
  - ✓ 일직선으로 이동하여 패킷을 오랫동안 보내지 않으면 서버와의 좌표차이로 인해 동기화에 문제가 생길 수 있기 때문에 정확성을 위해 2초마다 Move Packet을 전송하여 서버와 클라이언트간 좌표를 동기화 해주었습니다.

데드 레커닝



- Cubic Spline 보간
  - ✓ 클라이언트와 서버에서 각각 좌표에 대한 연산을 해주고 있기 때문에 보간을 하지 않는다면 좌표 값이 틀어지는 경우가 생기기 때문에 보간을 구현했습니다.
  - ✓ 클라이언트에서 Move 패킷에 system_clock::now()를 추가하여 전송해 주었고 클라이언트 -> 서버 -> 클라이언트 로 전송되는 과정 동안의 시간을 계산하여 RTT를 도출해내서 보간을 진행했습니다.
  - ✓ RTT를 1/3로 나누고 현재 플레이어의 방향벡터와 회전값을 이용해 RTT만큼의 좌표이동을 예측 후 보간을 진행했습니다.

서버 -> 클라

```
case SC_PACKET_MOVE:
{
    ////cout << "SC_PACKET_MOVE" << endl;
    sc_packet_move* packet = reinterpret_cast<sc_packet_move*>(ptr);
    int other_id = packet->id;

    if (other_id == g_myid) { ... }
    else
    {
        if (0 != g_Object.count(other_id))
        {
            if (CheckObjType(other_id) == OBJECT_TYPE::CLIENT)
            {
                system_clock::time_point end = system_clock::now();
                nanoseconds rtt = duration_cast<nanoseconds>(end - packet->Start);
```

클라 -> 서버

```
system_clock::time_point start = system_clock::now();
g_netMgr.Send_Move_Packet(dir, localPos, worldDir, vRot.y, start, ReckonerMove);
```

몬스터 이동

```
for (auto& user : viewList)
{
    if (m_pMediator->IsType(user, OBJECT_TYPE::MONSTER))
    {
        if (m_pMediator->Find(user)->GetStatus() == OBJSTATUS::ST_SLEEP)
            WakeUp_Monster(user);
    }
}
```

```
void CNetMgr::WakeUp_Monster(const uShort& id)
{
    if(CAST_MONSTER(m_pMediator->Find(id))->GetBisMoving())return;

    if (CAS((int*)(&(m_pMediator->Find(id)->GetStatus())), OBJSTATUS::ST_SLEEP, (int)ST_ACTIVE))
    {
        //CAS(메모리, expected, update)
        //    메모리의 값이 expected면 update로 바꾸고, true 리턴
        //    메모리의 값이 expected가 아니면 false 리턴
        //    WAIT FREE를 유지하는 알고리즘
        Add_Timer(id, ENUMOP::OP_RAMDON_MOVE_MONSTER, system_clock::now());
    }
}
```

```
if (ev.event_id == ENUMOP::OP_RAMDON_MOVE_MONSTER)
{
    EXOVER* over = new EXOVER();
    over->op = ENUMOP::OP_RAMDON_MOVE_MONSTER;
    PostQueuedCompletionStatus(g_hiocpHandle, 1, ev.obj_id, &over->over);
}
```

- Move 패킷을 처리하면서 해당 플레이어 주위에 있는 Sleep 상태의 Monster에 대해 WakeUp_Monster 함수를 호출한다.
- WakeUp_Monster에서는 atomic_compare_exchange_strong 함수를 사용한 CAS 함수 구현을 통해 WAIT FREE를 유지하는 알고리즘을 채택하여 사용
- Add Timer에서는 우선순위 큐에 Event를 Push해주고 Timer_thread에서 PostQueuedCompletionStatus 함수를 사용해 Worker_Thread를 깨워준다. Worker_thread에서는 실제 움직임에 대한 연산을 해주지 않고 Processing_thread에서 연산을 해주기위한 명령을 전달해주는 역할을 한다.

Game Server

게임 소개 – 게임서버 프로그래밍

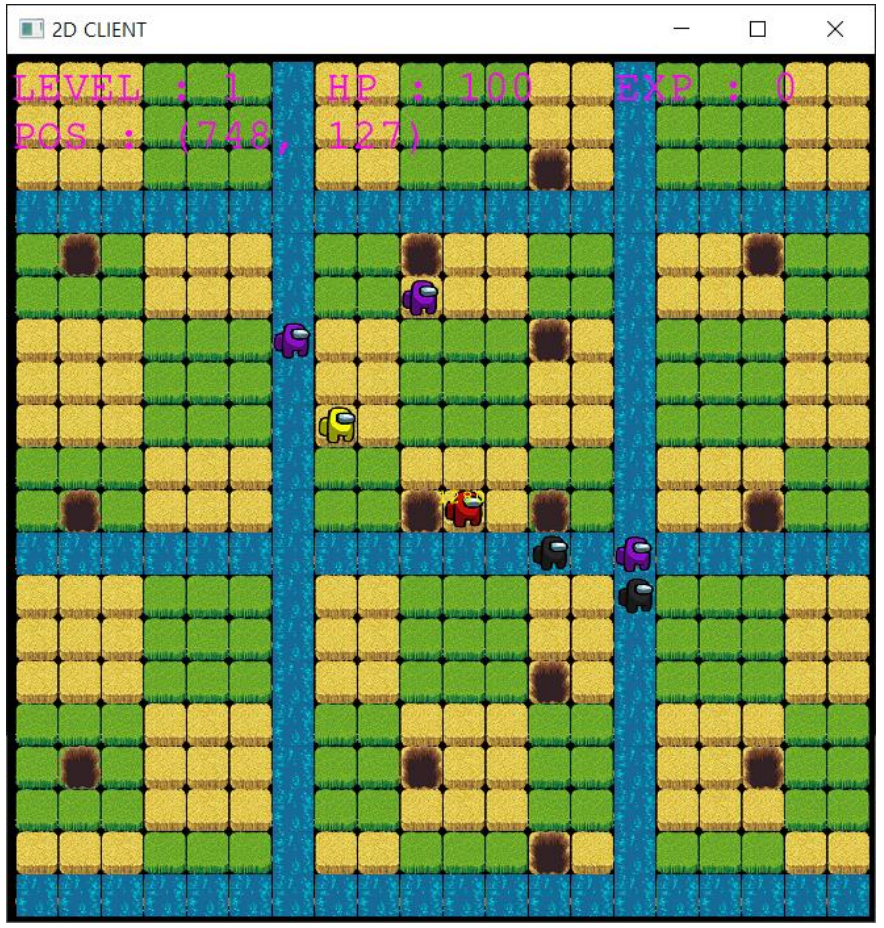
장르
- 2D RPG

개발 환경
- SFML
- Visual Studio 2019
- Git Hub
- MSSQL Management Studio

개발 인원
- 1인 개발

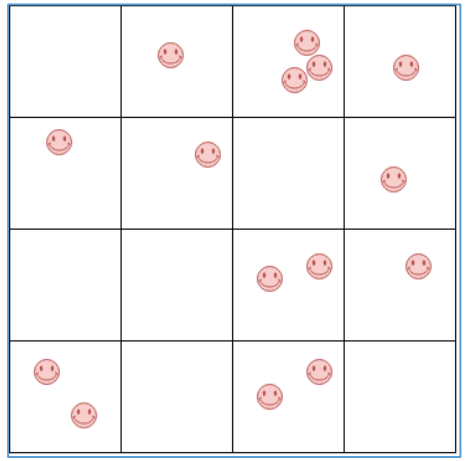
개발기간
- 2020.10 ~ 2020.12

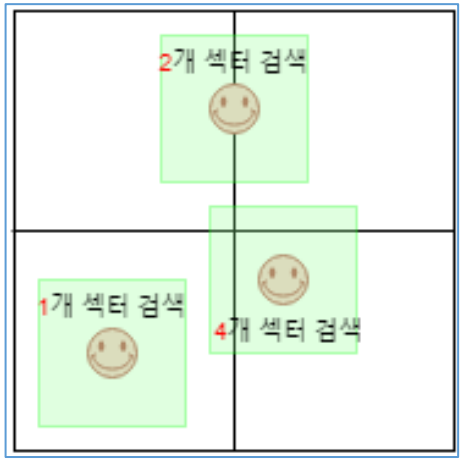
담당 업무
- SFML을 사용한 2D 클라이언트 구현
- IOCP를 사용한 멀티플레이게임 구현
- 데이터베이스 연동
- STRESS TEST 진행
- 시야처리(Sector)



Youtube : https://youtu.be/gSPpXXZsoVs

시야 처리(Sector)






```
void Change_Sector(int user_id)
{
    CURRENT_SECTOR oldSector(g_clients[user_id].m_iCSector.x, g_clients[user_id].m_iCSector.y);

    g_clients[user_id].m_iCSector.x = g_clients[user_id].x / (WORLD_WIDTH / SECTOR_ROW);
    g_clients[user_id].m_iCSector.y = g_clients[user_id].y / (WORLD_WIDTH / SECTOR_COL);


    if (oldSector.x != g_clients[user_id].m_iCSector.x ||
        oldSector.y != g_clients[user_id].m_iCSector.y)
    {
        sector[g_clients[user_id].m_iCSector.x][g_clients[user_id].m_iCSector.y].insert(user_id);
        sector[oldSector.x][oldSector.y].erase(user_id);
    }
}
```

- 전체 맵을 격자형태로 미리 나눠 놓고 움직임에 따라 Sector를 바꿔주며 진행했고 너무 많은 섹터를 검사하지 않기 위해 Sector 하나의 크기는 시야범위의 1.5배로 설정하여 최대 4개의 섹터를 검색하도록 하였습니다.

- 검색 범위에 포함된 섹터의 모든 객체에 대해 거리를 계산하고 시야내에 있다면 패킷을 주고 받는 형태로 구현했습니다.

- 삽입/삭제 연산을 줄이기위해 Move 패킷을 받았을때 이전에 속해있던 섹터와 이동한 좌표가 속한 섹터를 비교하여 섹터 변화가 있는 경우에만 섹터를 옮겨주었습니다.

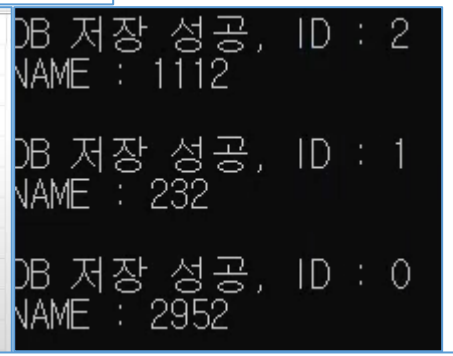
```
unordered_set<int>  sector[SECTOR_ROW][SECTOR_COL];
```

- Sector에서는 ID가 중복되어서는 안되고 순서가 필요 없으며 접속하는 유저가 많아질 수록 삽입/삭제와 검색이 늘어나는데 섹터내의 객체에 대한 검색이 많이 늘어난다고 생각해서 검색 속도가 빠른 unordered_set을 사용하였습니다.

DataBase

	Column Name	Data Type
🔑	ID	int
	NAME	nchar(20)
	USER_LEVEL	int
	X	int
	Y	int
	HP	int
	EXP	int

	ID	NAME	USER_LEVEL	X	Y	HP	EXP
1	0	2952	1	41	67	100	0
2	1	232	1	734	100	100	0
3	2	1112	1	781	480	91	0
4	3	3	1	280	553	100	0
5	4	4	1	569	455	56	0
6	5	5	1	101	141	100	0
7	6	6	1	83	20	100	0
8	7	7	1	362	497	100	0
9	8	8	1	573	742	45	0
10	9	9	1	25	628	100	0

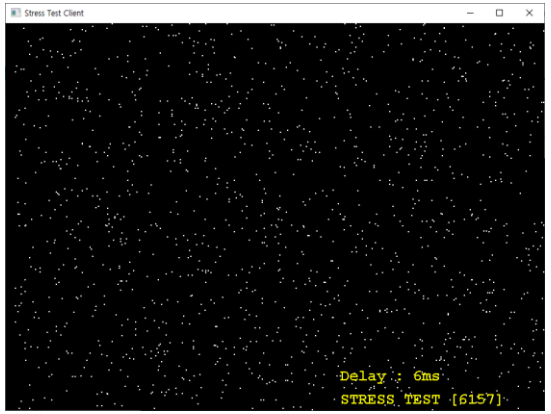


- Microsoft SQL Server Management Studio를 사용해 데이터를 관리 했습니다.
- 게임 플레이에 필요한 ID와 이름, 레벨, 위치, HP, EXP를 데이터 베이스에 저장했으며 네트워크 소요 시간을 줄이기 위해 Store Procedure를 사용하여 구현했습니다.
- 저장시에 해당 ID가 이미 저장되어 있다면 Update 그렇지 않다면 Insert를 하기 위해 조건을 추가하여 Upsert 구문을 구현했습니다.

```
IF EXISTS(SELECT * FROM USER_TABLE WHERE ID = @Param1)
begin
UPDATE USER_TABLE set NAME = @Name, USER_LEVEL = @Param3, X = @Param4, Y = @Param5, HP = @Param6, EXP = @Param7
WHERE ID = @Param1
end

ELSE
begin
INSERT INTO USER_TABLE (ID, NAME, USER_LEVEL, X, Y, HP, EXP)
VALUES(@Param1,@Name, @Param3, @Param4, @Param5, @Param6, @Param7)
end
```

Stress Test

9Thread



8Thread

- 작업환경
  - ✓ AMD Ryzen 7 3700X 8-Core
  - ✓ LAN
  - ✓ 플레이어마다 1초에 한번 Move패킷 송수신
  - ✓ 서버

    8개의 패킷처리(worker) Threads, 1개의 Timer Thread

7Thread

9 Thread						
횟수	1	2	3	4	5	평균
테스트	6305	6157	6101	5980	6233	6155.2

8 Thread						
횟수	1	2	3	4	5	평균
테스트	5979	6115	6013	5994	6043	6028.8

7 Thread						
횟수	1	2	3	4	5	평균
테스트	5476	5805	5481	5777	5646	5637

- 테스트
  - ✓ Timer Thread와 테스트 프로젝트의 스레드 개수는 고정
  - ✓ 서버의 스레드 개수가 7/8/9개 일 때의 동접 평균을 계산하여 최대치인 9Thread 채택

Network Game Programing

게임 소개 – 게임서버 프로그래밍

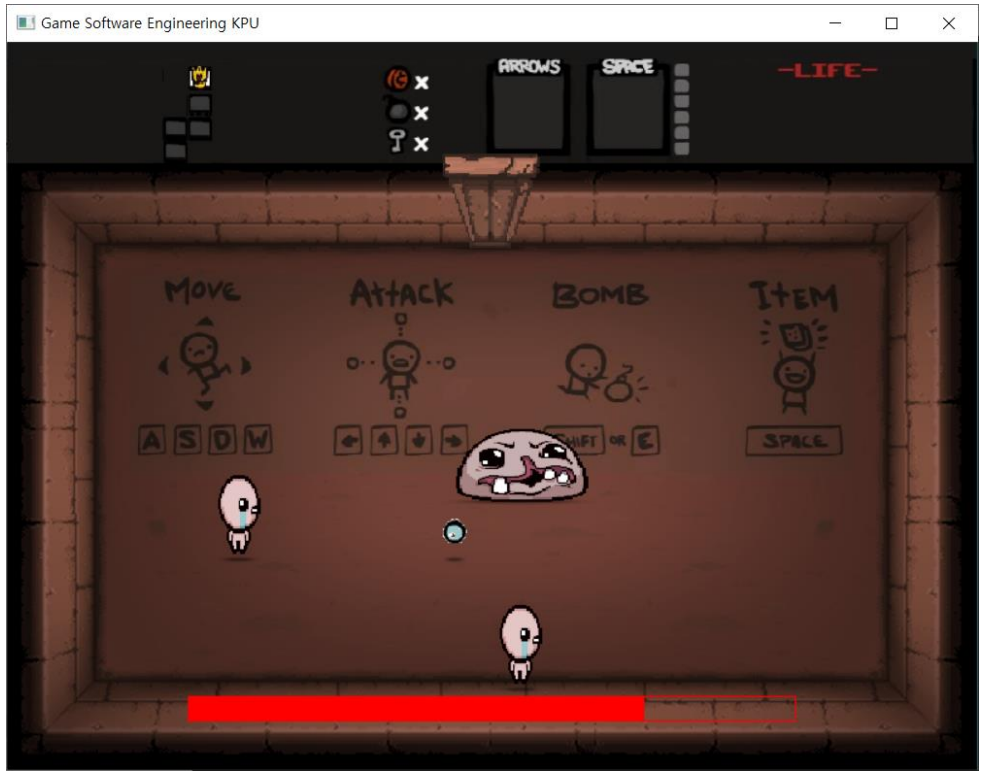
장르
- 2인용 2D 아이작

개발 환경
- Open GL
- Visual Studio 2019
- Git Hub

개발 인원
- 2인 개발

개발기간
- 2020.10 ~ 2020.12

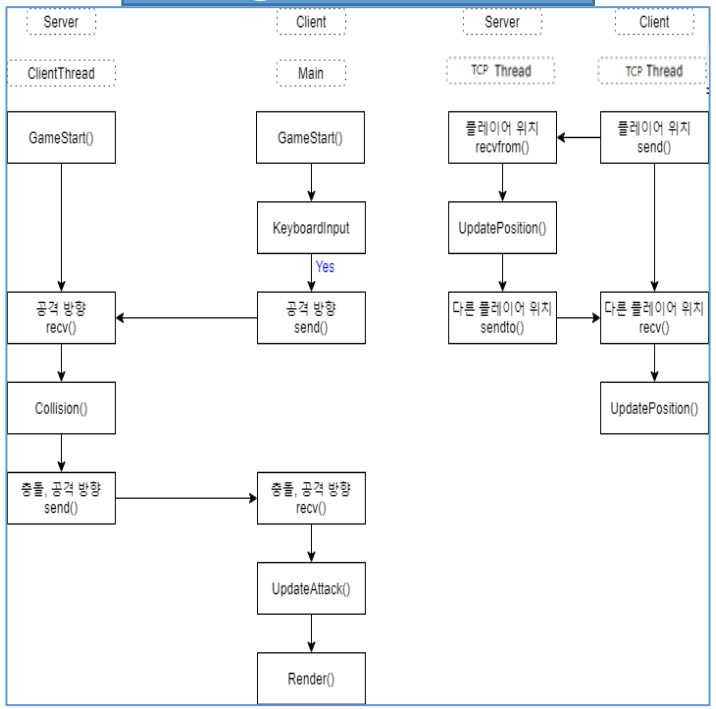
담당 업무
- Open GL을 사용한 2D 클라이언트 구현
- TCP 서버 구현



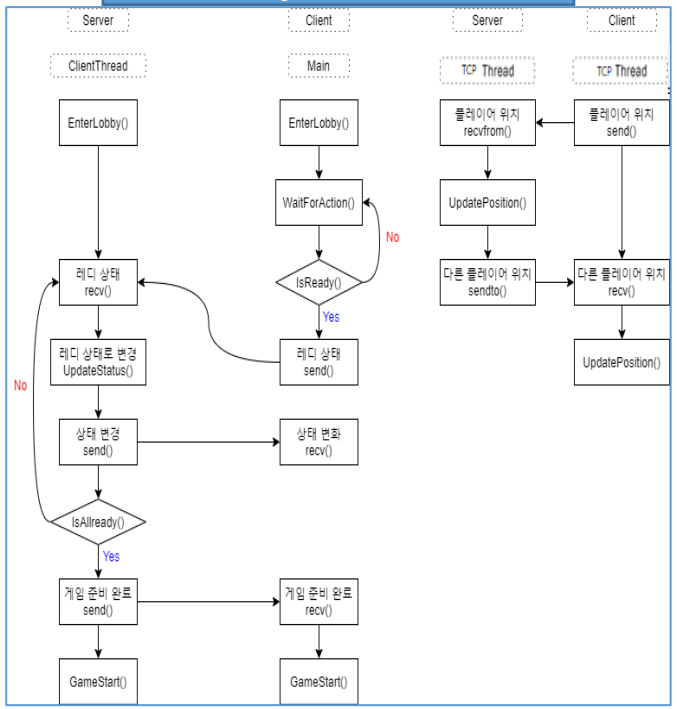
기획 및 협업

High Level Design

LogIn Network



Lobby Network



Low Level Design

```
float m_healthPoint;
float m_posX, m_posY, m_posZ;    // Position
float m_velX, m_velY, m_velZ;    // Velocity

float m_r, m_g, m_b, m_a;        // color
float m_volX, m_volY, m_volZ;    // Volume
float m_fricCoef;                // friction
int m_type;                      // object type
int m_textID;

float m_sx, m_sy, m_sz;          // size
float m_mass;                    // size
```

```
struct CS_Client_Logout_Packet
{
    char size;
    char type;
    char id;
};
```

- 개발 전 High/Low Level Design을 통해 게임의 전체적인 구성에 대해 팀원과 의견을 공유 하며 전체적인 게임의 흐름과 패킷 구조, 함수구현, 분업 관련 내용을 기획하였고 기획에 따라 개발을 진행했습니다.
- 클라이언트 수정 및 동기화, 서버 프레임워크 제작을 담당했습니다.

개발 내용

```
int ServerFrame::recvn(SOCKET s, char* buf, int len, int flags)
{
    int received;
    char* ptr = buf;
    int left = len;

    while (left > 0) {
        received = recv(s, ptr, left, flags);
        if (received == SOCKET_ERROR)
            return SOCKET_ERROR;
        else if (received == 0)
            break;
        left -= received;
        ptr += received;
    }

    return (len - left);
}
```

```
struct CS_Client_Logout_Packet
{
    char size;
    char type;
    char id;
};
```

- TCP통신의 특성상 지정한 크기보다 작은 데이터가 버퍼에 복사 될 수 있고 그에 따라 보낸 데이터의 크기 전부를 받기 까지 대기 시간이 생기므로 이를 방지하기 위해 패킷에 해당 데이터의 사이즈를 넣어서 보내주고 받은 데이터의 크기만큼 recv할 때까지 recv함수를 호출하여 대기 시간을 줄이는 방식으로 패킷 수신을 구현했습니다.

- 패킷에 어떤 역할을 하는 패킷인지를 저장하고있는 type 변수를 추가하여 상황에 맞는 연산을 진행합니다.

```
if (move_packet.type == CS_PACKET_MOVE)
if (loginok_packet.type == NICKNAME_USE)
```