2025 게임서버프로그래밍

텀프로젝트 게임 설명서

2020184003 김경환

내용

프로토콜 3

자료구조 4

게임 흐름 6

알고리즘 11

프로토콜

프로토콜은 교수님께서 제공하신 프로토콜을 기본으로 사용한다.

일부 변경/추가한 프로토콜만 기재하도록 하겠다.

- sc_packet_move

```
struct sc_packet_move {
    unsigned char size;
    char type;
    long long id;
    short x, y;
    unsigned int move_time;
};
```

스트레스 테스트에서의 딜레이를 확인하기 위해 move_time을 추가

- cs_packet_warp

```
struct cs_packet_warp {
    unsigned char size;
    char type;
    char zone;
};
```

Warp 패킷을 추가. 해당 패킷은 zone을 이용해 플레이어를 특정 위치(마을, 사냥터)로 빠르게 이동시킨다.

자료구조

유저와 npc의 정보는 SESSION class로 관리

SESSION은 SESSION.h에서 확인 가능하며 중요한 변수만 설명하겠다.

```
short                    _sector_coord[2] = { 0, 0 };
```

현재 자신이 속한 Sector의 index

```
std::chrono::high_resolution_clock::time_point _attack_term;
std::chrono::high_resolution_clock::time_point _move_term;
std::chrono::high_resolution_clock::time_point _revive_term;
std::chrono::high_resolution_clock::time_point _path_trace_term;
```

각 기능별로 쿨타임(이동: 0.5초, 공격: 1초 등)을 저장하는 변수
위에서부터 공격, 이동, 부활, 길찾기를 의미한다.

```
std::stack<NODE>                                    _path;
short                                               _no_near{};
```

npc의 경우 타겟을 쫓아가는 경로를 _path에 저장해 사용한다.

_no_near는 npc가 특정 행동을 했을 때 주변에 어떠한 플레이어가 없으면 증가하며 wakeup한 npc를 다시 재울 때 사용한다.

SESSION에서 사용하는 멤버 함수들은 알고리즘에서 설명하겠다.

SESSION은 concurrent_unordered_map 컨테이너에 담겨 g_users란 이름으로 사용된다.

다음은 타이머 큐에 사용되는 자료구조이다.

```
enum EVENT_TYPE { PL_HEAL, EV_NPC_AI };

struct event_type {
    long long obj_id;
    std::chrono::high_resolution_clock::time_point wakeup_time;
    EVENT_TYPE event_id;
    long long target_id;

    constexpr bool operator < (const event_type& _Left) const
    {
        return (wakeup_time > _Left.wakeup_time);
    }
};
```

타입은 5초마다 플레이어를 회복시키는 PL_HEAL과 npc의 ai를 작동시키는 EV_NPC_AI가 있다.

마지막으로 A* 길찾기를 할 때 사용되는 NODE 구조체이다.

```
struct NODE {
    short _x;
    short _y;

    int _h;     // 휴리스틱 값
    int _g;

    NODE* _parent;
    NODE(short x, short y, int h, int g, NODE* parent)
        : _x(x), _y(y), _h(h), _g(g), _parent(parent)
    {
    }
    constexpr bool operator < (const NODE* _Left) const
    {
        return ( _h + _g > _Left->_h + _Left->_g);
    }
};
```

이 자료구조들은 모두 SESSION.h에서 찾을 수 있다.

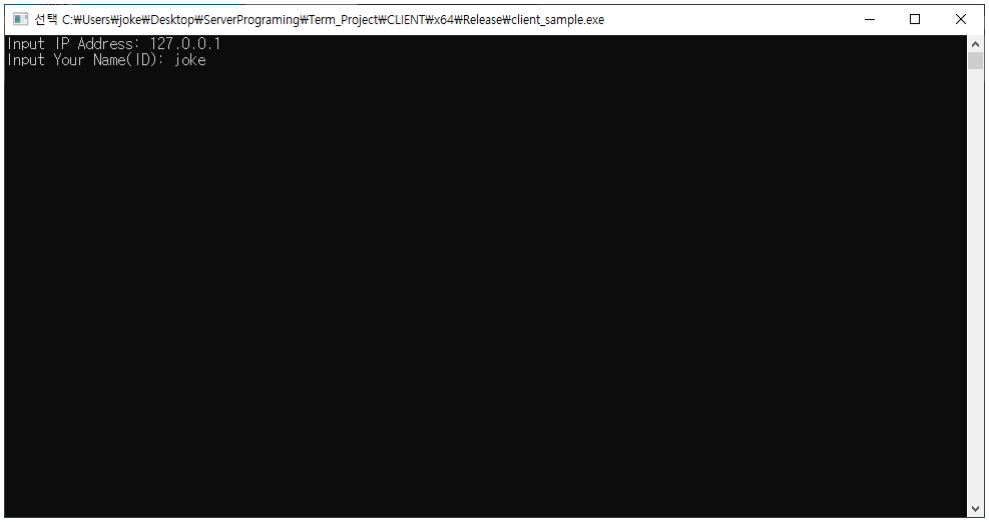
게임 흐름

조작방법: 이동(방향키), 공격(A), 빠른 이동(숫자 1, 2, 3, 4, 9), 종료(ESC)

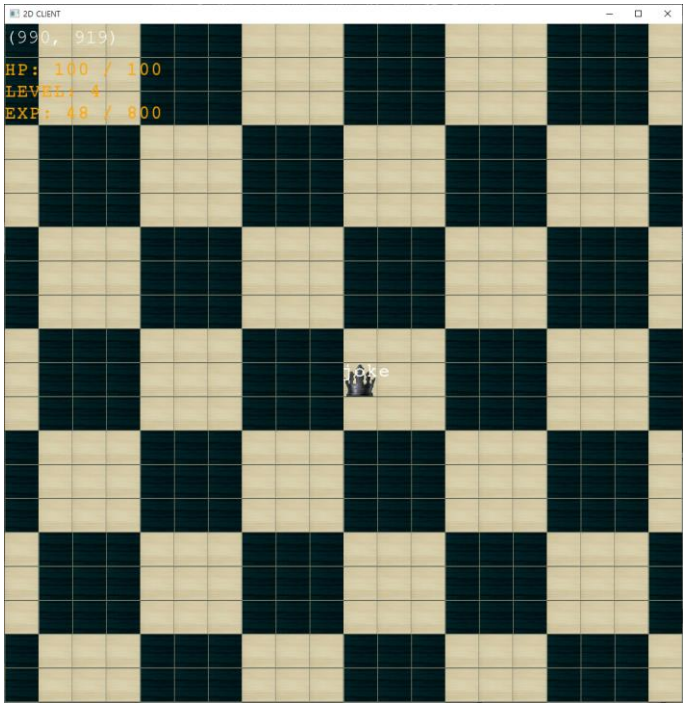
먼저 Term_project 폴더 안에 Term_Project.sln을 이용해 서버를 실행 시킨다.

서버 콘솔에 'NPC_Ready' 가 나타나면 CLIENT 폴더에 있는 클라이언트를 실행시킨다.

서버 주소와 이름을 입력하면 게임이 시작된다.



(요구 정보를 모두 입력하면)



(게임 화면이 나타난다.)

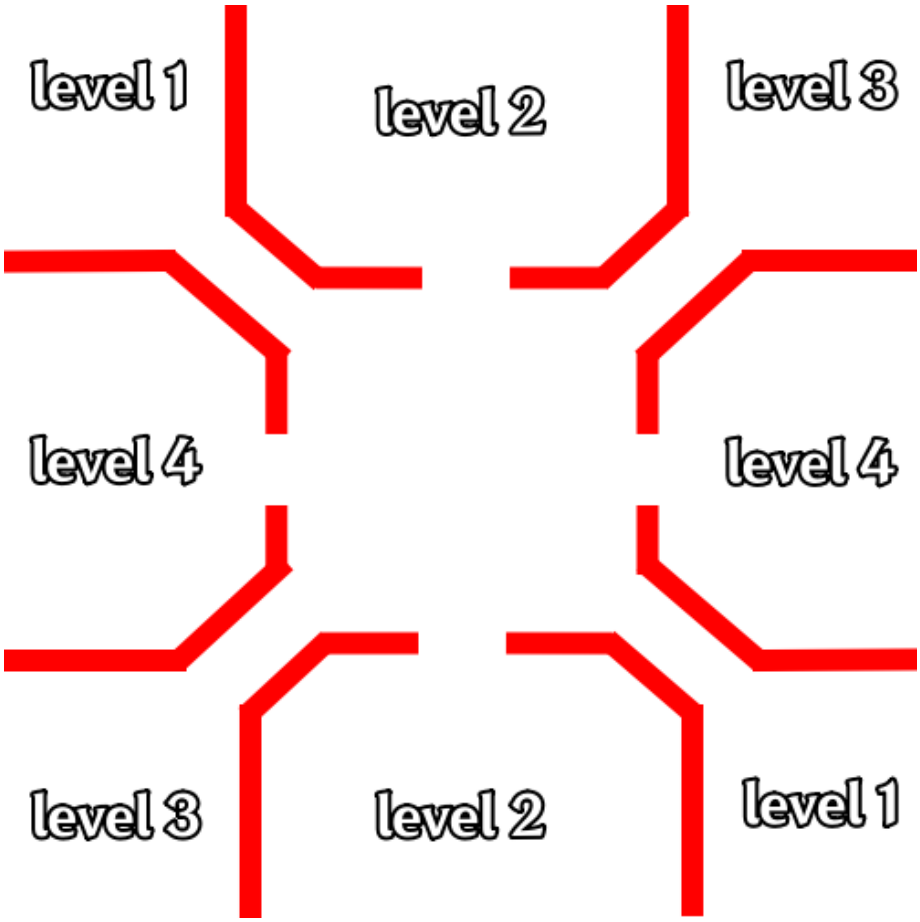
좌측 상단에 스탯 정보가 나오고 enter를 입력하면 채팅 모드를 이용할 수 있다.

채팅 모드 중엔 키보드를 입력해 채팅을 입력하고 enter를 재입력해 채팅을 보낼 수 있다.

타인의 채팅이나 시스템 메시지가 오면 채팅창이 3초간 켜지며 최대 10개까지 이전 채팅과 함께 나타난다.(채팅 모드일땐 상시로 보여준다.)

또한 타인의 채팅은 내 시야에 있을 때만 채팅창에 나타난다.
(맵 전체의 유저를 보기엔 부하가 너무 크다 판단했다.)

게임의 맵은 다음과 같이 구성되어 있다.



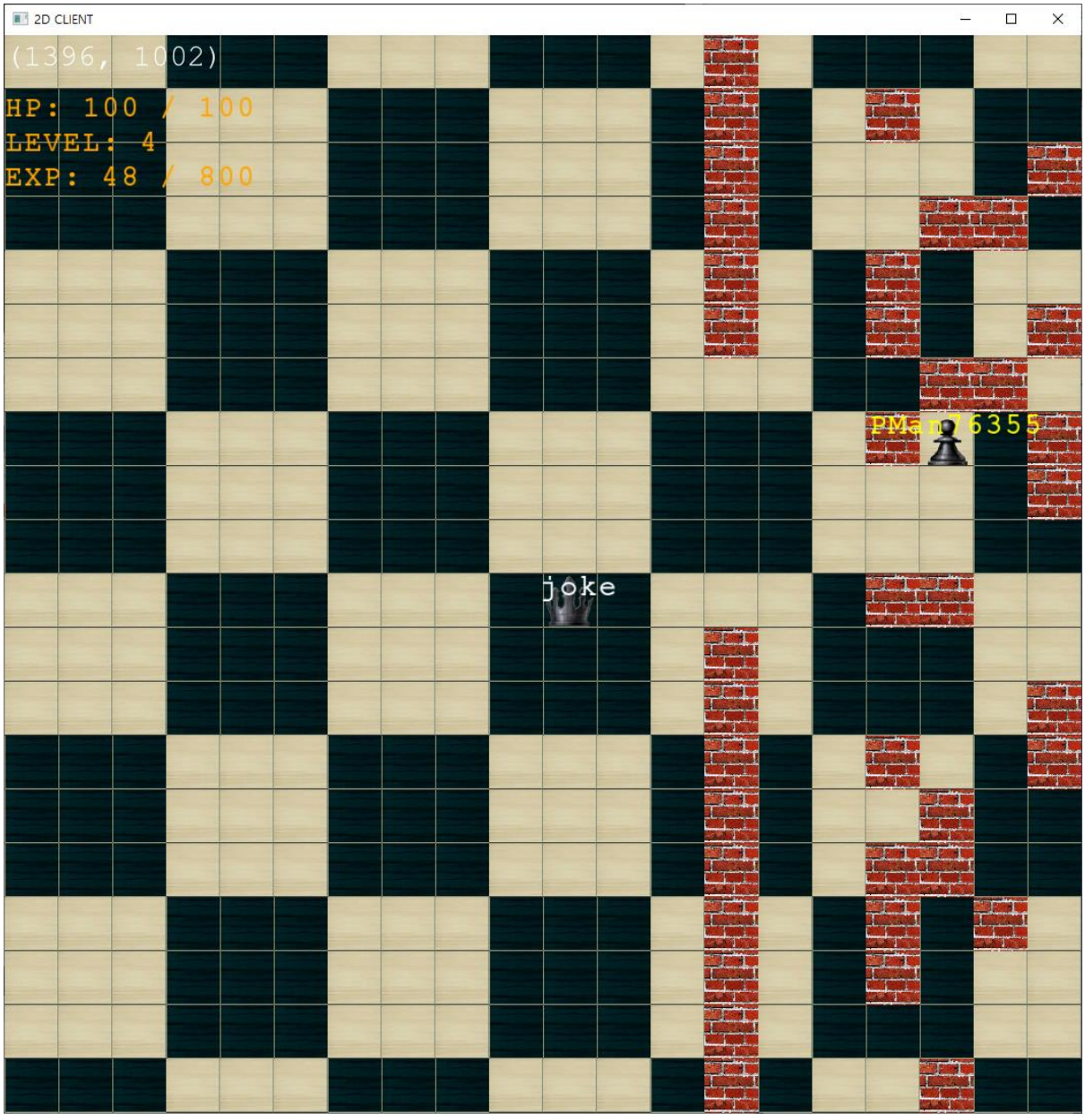
가운데 공간은 광장(마을)이며 마을에선 이동속도에 제한이 없다.
(사냥터에선 0.5초 제한이 있다.)

나머지 8개 공간은 사냥터이며 사냥터에는 몬스터와 장애물들이 있다. 몬스터의 레벨은 위 그림과 같이 정해져있다.

하나의 사냥터에는 25000마리의 npc가 존재하며 사냥터의 30%는 장애물로 채워져 있다.

게임의 흐름은 자신에게 맞는 사냥터를 찾아가 몬스터를 잡고 레벨을 올려 강해지는게 흐름이다.

1~4 번을 누르면 각 번호에 맞는 레벨의 사냥터 입구로 빠른 이동이 가능하다. 마을로 돌아가고싶으면 9번을 눌러 언제든지 마을로 빠르게 이동할 수 있다.



4번을 눌러 4level 사냥터 입구로 온 모습이다.

몬스터의 종류는 총 4가지이며 이미지는 다음과 같다.

이중 킹은 플레이어(본인), 룩은 플레이어(타인)을 의미하며

Peace 고정은 폰

Peace Roming은 나이트

Agro 고정은 퀸

Agro Roming은 비숍이 담당한다.

Peace는 몬스터는 플레이어가 공격하면 쫓아오고 Agro 몬스터는 일정 거리 이내에 들어오면 쫓아온다.

쫓아오는 몬스터에게 충돌하면 플레이어의 체력이 감소한다. 체력은 5초마다 최대체력의 10%만큼 차오른다.
(A*가 완벽하지 않아 몬스터가 플레이어 바로 앞까지만 온다. 체력 감소를 보고싶으면 몬스터가 오는 경로에 서있어야 한다.)

몬스터를 처치하면 몬스터의 종류와 레벨에 따라 경험치를 획득하며 일정량의 경험치를 채우면 레벨업을 한다.

레벨을 많이 올려 강해지고 사냥터를 정복해보도록 하자.

알고리즘

- 로그인

로그인 시 클라이언트에서 cs_packet_login을 보내고 해당 패킷 안에 있는 이름을 이용해 부적할한지 중복인지 체크하도록 한다.

```
case C2S_P_LOGIN: {
    cs_packet_login* p = reinterpret_cast<cs_packet_login*>(packet);

    std::wstring sqlcommand{ L"EXEC Get_User_Info " };
    std::string sName{ p->name };

    for (int i = 0; i < MAX_USER; ++i) {
        if (g_users.count(i)) {
            std::shared_ptr<SESSION> cl = g_users.at(i);
            if (cl != nullptr) {
                if (!strcmp(cl->_name, p->name)) {
                    do_send_login_fail(1);
                    return;
                }
            }
        }
    }
```

먼저 패킷의 이름과 g_users를 이용해 이름 중복 여부를 체크하고 중복일 시 login_fail 패킷에 reason을 1로 설정해 보낸다.

```
if (sName.size() >= 20) {
    do_send_login_fail(2);
    return;
}
for (auto& c : sName) {
    if (false == isalpha(c) && false == isdigit(c)) {
        do_send_login_fail(2);
        return;
    }
}
```

이름 길이과 특수 문자가 들어갔는지도 체크한다.

모든 검사를 통과해 적절한 이름으로 판정되면 database에 해당 이름으로 정보를 요청한다.

database에서 Get_User_Info 프로시저를 만들어 놨기에 사용하도록 한다.

Database에 해당 이름으로 된 유저가 없다면 Insert_New_User를 이용해 추가하도록 한다.

```
retcode = SQLExecDirect(hstmt, (SQLWCHAR*)sqlcommand.data(), SQL_NTS);
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
    retcode = SQLBindCol(hstmt, 1, SQL_C_LONG, &db_max_hp, 4, &cb_db_max_hp);
    retcode = SQLBindCol(hstmt, 2, SQL_C_LONG, &db_hp, 4, &cb_db_hp);
    retcode = SQLBindCol(hstmt, 3, SQL_C_LONG, &db_level , 4, &cb_db_level);
    retcode = SQLBindCol(hstmt, 4, SQL_C_LONG, &db_exp, 4, &cb_db_exp);
    retcode = SQLBindCol(hstmt, 5, SQL_C_LONG, &db_x, 4, &cb_db_x);
    retcode = SQLBindCol(hstmt, 6, SQL_C_LONG, &db_y, 4, &cb_db_y);
    retcode = SQLFetch(hstmt);
    SQLCloseCursor(hstmt);
    if (retcode != SQL_SUCCESS && retcode != SQL_SUCCESS_WITH_INFO) {   // Create New ID
        sqlcommand = L"EXEC Insert_New_User ";

        strcpy_s(_name, p->name);
        _x = (rand() % 500) + 750;
        _y = (rand() % 500) + 750;

        _max_hp = 100; _hp = 100;
        _level = 1; _exp = 0;
        _need_exp = 100 * pow(2, _level - 1);
        _attack = 30;
```

DB에 존재하면 DB에서 받은 정보로 SESSION을 생성하고 없다면 새로 만든 유저 정보로 SESSION을 생성한다.

이후 과정은 avatar_info를 보내주고 주변 유저에게 enter 패킷을 보내도록 한다.

- 로그아웃

로그아웃 시 DB에 현재 정보를 저장해야한다. disconnect()를 이용해 저장한다.

```
std::wstring sqlcommand{ L"EXEC save_user_data " };
std::string sName{ _name };
std::wstring wName{ sName.begin(), sName.end() };
sqlcommand = sqlcommand + std::to_wstring(_hp) + L", " + std::to_wstring(_level) + L", " +
    std::to_wstring(_exp) + L", " + std::to_wstring(_x) + L", " + std::to_wstring(_y) +
    L", " + wName;
retcode = SQLExecDirect(hstmt, (SQLWCHAR*)sqlcommand.data(), SQL_NTS);
SQLCloseCursor(hstmt);
```

disconnect() 사용 시 SESSION을 지우기 전 stored procedure를 이용해 DB에 저장하도록 한다.

- sector 최적화

시야 최적화를 위해 sector를 이용한다.

이 프로젝트에서 sector 하나의 크기는 20x20이며 총 10000개의 Sector가 존재한다.

```
std::mutex g_sl;
std::array<std::array<std::unordered_set<long long>, MAP_WIDTH / SECTOR_SIZE>,
    MAP_HEIGHT / SECTOR_SIZE> g_sector;
```

sector는 다음과 같이 array와 unordered_set를 사용했고 datarace를 방지하기 위해 mutex와 함께 전역공간에 선언하였다.

Npc나 플레이어가 추가될 때 그 객체가 존재하는 sector에 id가 추가되며 이동이나 종료할 때 sector를 최신화 하도록 한다.

```
p->_x = x; p->_y = y;
p->_ix = x; p->_iy = y;

p->_sector_coord[0] = p->_y / SECTOR_SIZE;
p->_sector_coord[1] = p->_x / SECTOR_SIZE;

g_sector[p->_sector_coord[0]][p->_sector_coord[1]].insert(p->_id);
```

(npc나 플레이어 추가 시 해당 좌표를 이용해 sector를 결정하고 추가한다.)

```
_x = x; _y = y;
_move_time = p->move_time;
short sx = _y / SECTOR_SIZE;    // row
short sy = _x / SECTOR_SIZE;    // col
if (_sector_coord[0] != sx || _sector_coord[1] != sy) {
    g_sl.lock();
    g_sector[_sector_coord[0]][_sector_coord[1]].erase(_id);
    g_sector[sx][sy].insert(_id);
    g_sl.unlock();
    _sector_coord[0] = sx;
    _sector_coord[1] = sy;
}
```

(sector 업데이트 시 반드시 lock을 걸고 하도록 한다.)

이제 시야 처리에 sector를 이용하도록 한다.

예를 들어 플레이어 이동시 near_list를 만들기 전 먼저 나와 같거나 인접한 sector에 있는 객체들을 뽑는다.

```
std::unordered_set<long long> su_list;
g_sl.lock();
for (short i = sx - 1; i <= sx + 1; ++i) {
    if (i < 0 || i >= MAP_HEIGHT / SECTOR_SIZE) continue;
    for (short j = sy - 1; j <= sy + 1; ++j) {
        if (j < 0 || j >= MAP_WIDTH / SECTOR_SIZE) continue;
        for (long long id : g_sector[i][j])
            su_list.insert(id);
    }
}
g_sl.unlock();
```

(주위 8개의 sector를 포함한 9개의 sector에서 id를 가져온다.)

이제 이 su_list를 이용해 진짜 내 시야에 들어오는 객체들을 뽑도록 한다.

```
for (long long id : su_list) {
    std::shared_ptr<SESSION> ply = g_users.at(id);
    if (nullptr == ply)
        continue;
    if (ply->_id == _id) continue;
    if (ply->_state != ST_INGAME) continue;
    if (can_see(_x, _y, ply->_x, ply->_y))
        near_list.insert(ply->_id);
}
```

이런식으로 시야처리를 하면 맵에 있는 인원 전체를 검색하지 않기 때문에 계산 시간이 단축된다.

- NPC AI

먼저 NPC 종류 먼저 기억하고 가자

1. Peace Fix(PF): 후공 고정 몹 초기 상태(0)
2. Peace Roming(PR): 후공 로밍 몹 초기 상태(2)
3. Agro Fix(AF): 선공 고정 몹 초기 상태(0)
4. Agro Roming(AR): 선공 로밍 몹 초기 상태(2)

Npc는 상태를 가진다. 상태는 3종류가 있으며 다음과 같다.

0: 평화 상태, 고정 상태

2: 로밍 상태

3: 추적 상태

해당 상태들을 생각해 lua 스크립트를 다음과 같이 작성했다.

```
myid = 99999;
target_id = 99999;
myState = 0;


function set_myid(x)
    myid = x;
end

function set_target(x)
    target_id = x;
end

function set_state(x)
    myState = x
end


function npcAI()
    if(myState == 0 or myState == 1) then
        myState = API_CheckUser(myid);
    elseif(myState == 2) then
        myState = API_Roming(myid);
    else
        myState = API_Chase_target(myid, target_id);
    end
end
```

본래 상태 1은 죽은 상태로 사용하려 했으나 사용하지 않기로 했다.

이제 각각의 상태에 따른 행동을 보도록 한다. 먼저 0상태일 때 사용하는 CheckUser먼저 살펴보자

```
int API_CheckUser(lua_State* L)
{
    long long uid = (long long)lua_tointeger(L, -1);
    lua_pop(L, 2);
    std::shared_ptr<SESSION> p = g_users.at(uid);
    if (nullptr == p) return 1;
    int ret = p->do_check_near_user();

    if (ret == -1)
        lua_pushnumber(L, 0);
    else {
        lua_getglobal(L, "set_target");
        lua_pushnumber(L, ret);
        lua_pcall(L, 1, 0, 0);
        lua_pushnumber(L, 3);
    }

    return 1;
}
```

해당 함수는 npc의 do_check_near_user()를 호출하며 do_check_near_user()는 내 npc_type이 AF일 때 근처에 플레이어가 있으면 해당 플레이어의 id를 return 한다. 그리고 그 플레이어를 추적하도록 npc를 3번(추적) 상태로 변경한다.

찾지 못하거나 npc_type이 틀릴 경우 -1을 return 하고 npc의 상태는 0으로 변하지 않는다.

두번째로 API_Roming이다.

```
int API_Roming(lua_State* L)
{
    long long uid = (long long)lua_tointeger(L, -1);
    lua_pop(L, 2);
    std::shared_ptr<SESSION> p = g_users.at(uid);
    if (nullptr == p) return 1;
    int nextstate = p->do_npc_move();

    lua_pushnumber(L, nextstate);
    return 1;
}
```

이 함수는 do_npc_move를 이용해 랜덤한 방향으로 이동 시킨다.

이동 시킨 후에 내 npc_type이 AR이고 근처에 플레이어가 있으면 해당 플레이어를 타겟으로 잡고 nextstate로 3를 return 한다. 이렇게 되면 npc의 상태는 3번(추적) 상태로 변경되고 타겟 추적을 시작한다.

만약 AR이 아니거나 타겟을 찾지 못했다면 2번 상태를 유지하도록 한다.

마지막으로 추적 상태때 사용하는 API_Chase_taget이다.

```
int API_Chase_target(lua_State* L)
{
    long long target_id = (long long)lua_tointeger(L, -1);
    long long uid = (long long)lua_tointeger(L, -2);
    lua_pop(L, 3);

    std::shared_ptr<SESSION> p = g_users.at(uid);
    if (nullptr == p) return 1;
    int nextstate = p->do_npc_chase(target_id);

    lua_pushnumber(L, 3);
    return 1;
}
```

3번 상태일 땐 do_npc_chase를 이용해 타겟을 추적한다. do_npc_chase를 상펴보자.

```
std::shared_ptr<SESSION> target = g_users.at(id);
if (target == nullptr) { ... }
```

```
auto t = std::chrono::high_resolution_clock::now();
if (_path_trace_term < t) {
    std::priority_queue<NODE*> openq;
    std::vector<std::vector<bool>> opencheck(2000, std::vector<bool>(2000, false));
    std::vector<std::vector<bool>> closelist(2000, std::vector<bool>(2000, false));
    std::list<NODE*> AllNode;

    short goal_x = target->_x;
    short goal_y = target->_y;
```

do_npc_chase에선 A*를 이용해 경로를 찾고 타겟을 추적한다.

하지만 매번 A*를 계산하는 건 너무 큰 작업이다.

그래서 _path_trace_term를 주어 3초마다 A*를 실행하도록 설정하였다.

평가함수에 필요한 h는 타겟과 노드 사이의 거리, g는 시작 노드와 현재 노드 사이의 깊이를 가지도록 했다.

```
while (!openq.empty()) {
    NODE* node = openq.top();
    openq.pop();
    if (node->_x == goal_x && node->_y == goal_y) { // 탐색 성공
        for (NODE* p = node->_parent; p != nullptr; p = p->_parent) {
            _path.push(NODE{ p->_x, p->_y, p->_h, p->_g, nullptr });
        }
        delete node;
        while (!openq.empty()) {
            NODE* p = openq.top();
            openq.pop();
            if (p) delete p;
        }
        for (NODE* p : AllNode)
            if (p) delete p;
        break;
    }
    else {
        short x = node->_x; short y = node->_y;
        switch (_in_section) { ... }
        opencheck[node->_y][node->_x] = false;
        closelist[node->_y][node->_x] = true;
        AllNode.push_back(node);
    }
}
```

탐색에 성공하면 _path에 경로가 저장된다.

```
if (!_path.empty()) {
    auto& n = _path.top();
    _path.pop();

    std::unordered_set<long long> su_list;
    g_sl.lock();
    for (short i = _sector_coord[0] - 1; i <= _sector_coord[0] + 1; ++i) {
        if (i < 0 || i >= MAP_HEIGHT / SECTOR_SIZE) continue;
        for (short j = _sector_coord[1] - 1; j <= _sector_coord[1] + 1; ++j) {
            if (j < 0 || j >= MAP_WIDTH / SECTOR_SIZE) continue;
            for (long long id : g_sector[i][j])
                su_list.insert(id);
        }
    }
    g_sl.unlock();

    std::unordered_set<long long> old_vl;
    for (auto& pid : su_list) {
```

그 경로를 이용해 이동하도록 한다.

```
std::unordered_set<long long> new_vl;
for (auto& pid : su_list) {
    std::shared_ptr<SESSION> p_ob = g_users.at(pid);
    if (p_ob == nullptr) continue;
    if (ST_INGAME != p_ob->_state) continue;
    if (false == is_pc(p_ob->_id)) continue;
    if (true == can_see(_x, _y, p_ob->_x, p_ob->_y)) {
        new_vl.insert(p_ob->_id);
        if (_x == p_ob->_x && _y == p_ob->_y) {
            p_ob->_hp -= _attack;
            std::string syschat;
            syschat = "You " + std::to_string(_attack) + " damaged from " + _name;
            p_ob->do_send_chat_packet(syschat.data());
            p_ob->do_send_stat();
            if (p_ob->_hp <= 0) {
                p_ob->_hp = p_ob->_max_hp;
                p_ob->_exp /= 2;
                p_ob->_x = (rand() % 500) + 750;
                p_ob->_y = (rand() % 500) + 750;
                p_ob->do_send_move();
            }
        }
    }
}
```

이동 중 플레이어와 충돌하면 해당 플레이어의 체력을 감소시키도록 한다.
그 플레이어의 체력이 0이 됐을 때 처리도 하도록 한다.

마지막으로 추적대상을 놓치면 FIX 몬스터는 다시 0번 상태로, Roming 몬스터는 2번 상태로 상태를 변경해주도록 한다.

정리해보면 Npc는 주로 이런식으로 작동한다.

타입에 따른 개인 행동 -> 추적 -> 타입에 따른 개인 행동 -> 추적

이제 npc가 공격을 받았을 때 행동을 보도록 하자.

Npc가 플레이어에게 공격을 받았을 땐 특별한거 없이 npc의 체력을 감소시키고 처치했을 시 플레이어에게 경험치를 주면 되는 간단한 것이다.

그런데 여기서 하나를 더 추가하도록 한다.

바로 Peace 몬스터의 경우인데 Peace 몬스터의 경우 공격받을 경우 공격한 플레이어를 추적하도록 해야한다.

```
if (attack_check(_x, _y, o->_x, o->_y)) {
    o->_hp -= _attack;
    std::string pname{ o->_name };
    std::string message;
    message = "You gave " + std::to_string(_attack) + "damage to " + pname;
    do_send_chat_packet(message.data());
    if (o->_hp <= 0) {
        o->_cl.lock();
        o->_x = o->_ix; o->_y = o->_iy;
        o->_cl.unlock();

        std::unordered_set<long long> npc_su_list;
        g_sl.lock();
        for (short i = o->_sector_coord[0] - 1; i <= o->_sector_coord[0] + 1; ++
        g_sl.unlock();
```

공격에 성공 시 시스템 채팅을 보내고

```
else {
    if (o->_npc_type == NPC_PEACE_FIX || o->_npc_type == NPC_PEACE_ROMING) {
        o->_ll.lock();
        lua_getglobal(o->_lua_machine, "set_state");
        lua_pushnumber(o->_lua_machine, 3);
        lua_pcall(o->_lua_machine, 1, 0, 0);

        lua_getglobal(o->_lua_machine, "set_target");
        lua_pushnumber(o->_lua_machine, _id);
        lua_pcall(o->_lua_machine, 1, 0, 0);
        o->_ll.unlock();
    }
}
```

해당 npc가 Peace 몬스터면 추적상태로 변경한다.