Term Project – Squid Game

https://github.com/todayoneul/Squid-Game

22100757 이규한

시작하기 앞서…

구현할 것이 많아 코드가 너무 길어져서 캡쳐로 끊기가 어렵고, 보고서로 보는 게 오히려 가독성이 떨어질 수 있다는 생각이 들었습니다. GitHub repository 링크 첨부하겠습니다. (특히 5번 sample run은 한 번에 캡처 및 입력이 쉽지 않아 이로 대체합니다.)

1. bool PlayerRLGL::act() 구현

```
//내가 구현해야 할 곳
bool PlayerRLGL::act()
{
    //random distance 구현
    unsigned int randomDistance = rand_dis(random_engine);
    // fearlessness가 높으면 일정 확률 이상으로 bonus distance 얻을 기회가 주어짐
    if (fearlessness > 100 * possibility(random_engine)) {
        //75% 이상의 확률로 fearless_bonus_distance를 받고, 10%의 확률로 죽으며, 나머지 경우엔 둘 다 하지 않음
        auto RLGL = [] { // 임의의 확률로 0,1,2를 반환하는 lambda 함수
            float p = possibility(random_engine);
            // p에 0부터 100 사이의 임의의 정수가 생성되고, 이를 확률로 변환기켜주기 위해
            if (p < PlayerRLGL::fallDownRate) // 10퍼센트의 확률로 떨어져 죽음
                return 0;
            else if (p < 0.85f) // 75퍼센트 확률로 bonus distance 얻음
                return 1;
            else // 나머지 경우엔 아무것도 안함
                return 2;
        };

        if (RLGL() == 0) { // 10 퍼센트의 경우 떨어져 죽기 때문에 playing을 false로 바꿔주고 return false
            this->playing = false;
            return false;
        }
        else if (RLGL() == 1) { // 75퍼센트의 확률로 bonus distance 얻었을 경우 계산해서 현재 거리에 더해주기
            //fearless bounus distance 구현
            unsigned int fearless_bounus_distance = this->agility * (this->fearlessness * 0.01);

            current_distance += this->agility + randomDistance + fearless_bounus_distance;
        }
        else { // 나머지 경우엔 agility + randomdistance만 계산
            current_distance += this->agility + randomDistance;
        }

    }
    else {// 나머지 경우엔 agility + randomdistance만 계산
        current_distance += this->agility + randomDistance;
    }
    // 만약 경기장 크기보다 많이 움직이고, play중이라면
    if (current_distance >= RedLightGreenLight::distance && isPlaying()) {
        this->playing = false; // playing을 false로 바꾸고 탈출 메세지 출력
        printStatus();
        std::cout << " safely escaped from the ground." << std::endl;
    }

    return true; // 떨어져서 죽지 않는 한, true 반환
}
```

2. void RedLightGreenLight::play() 구현

```
void RedLightGreenLight::play()
{
    printGameName();
    // 내가 구현해야 할 곳
    for (int t = 0; t < turn; ++t) // 턴 수만큼 반복
    {
        std::cout << "[Trun #" << t + 1 << "]" << std::endl; // 몇번째 턴인지 출력

        auto player = players.begin(); // 처음부터

        while (player != players.end()) // 끝까지
        {
            if ((*player)->isPlaying() && !(*player)->act()) // players에 있는 player 객체가
                                                             //playing = ture 고, act() 이후 false를 반환한다면
            {
                (*player)->dyingMessage(); // 떨어져 죽었다는 말이니, dyingMessage() 출력
                delete (*player); // 그 player 객체 삭제
                player = players.erase(player); // 객체 삭제했으니 다음 객체를 가르키는 iteraotr 반환
            }
            else
            {
                ++player; // 죽지 않았다면 다음 player로 넘어가 act()
            }
        }

    }
```

3. Marble Game – 새로운 게임 구현

구슬게임 – 구슬게임 안에 2개의 게임이 존재

만약 상대방의 구슬을 모두 없앤다면 그 즉시 게임 종료 -> 구슬을 모두 잃은 참가자 탈락 (일정 턴 또는 기회 이후 승패가 갈리지 않을 경우엔 모두 탈락하므로 절반 이상이 탈락)

게임 1. 홀짝게임 (OddOrEven)

게임은 단 2명씩 짝을 지어서, 턴은 10턴으로 제한 (인원이 홀수일 경우 맨 마지막 인원은 깍두기)
후공인 자는 가지고 있는 구슬의 개수 중 임의의 개수를 선택
선공은 선택한 구슬이 짝수인지 홀수인지 맞춰야 함
맞추면 일정량의 구슬을 상대방으로부터 가져옴/틀리면 일정량 상대방에게 헌납
모든 턴이 종료되고도 최종 승자가 결정되지 않으면 두 명 모두 탈락
player의 fearlessness가 낮을 수록 더욱 안전한 플레이 -> 구슬을 조금씩만 선택하고, 구슬의 예상 범위 작음

게임 2. 구슬 구멍에 넣기 (Hole_In_One)

구슬을 넣는 순간 게임 종료
1턴마다 각 플레이어들은 구슬을 구멍에 던질 수 있음.
구슬을 구멍에 넣는다면 그때까지 땅에 있는 모든 구슬을 가져옴.
모든 턴이 종료되고도 최종 승자가 결정되지 않으면 두명 모두 탈락
player의 agility가 클수록 구멍에 구슬을 넣을 확률 +

3.1. Game.h 에서의 구슬 게임

```
class MarbleGame : public Game
{

protected:
    static unsigned int turn; // 턴제를 구현하고 싶으면 turn 사용 가능
public:
    MarbleGame() : Game("Marble Game") {};
    ~MarbleGame() {};
    void join(Player* player) {};
    void play() {};
};
```

```
class OddOrEven : public MarbleGame
{
public:
    // game name을 새로 초기화 시켜주는 것 말고는 특별한 것은 없음
    OddOrEven() { gameName = "Odd or Even"; };
    OddOrEven(unsigned int t)
    {
        gameName = "Odd or Even";

    };
    ~OddOrEven() {};
    void join(Player* player);
    void play();
};
```

```
class Hole_In_One : public MarbleGame
{
public:
    // game name을 새로 초기화 시켜주는 것 말고는 특별한 것은 없음
    Hole_In_One() { gameName = "Hole In One!"; };
    Hole_In_One(unsigned int t)
    {
        gameName = "Hole In One!";
    };
    ~Hole_In_One() {};
    void join(Player* player);
    void play();
};
```

3.2. Player.h 에서의 구슬 게임

```
// 구슬 게임 참가자 class
class PlayerMarbleGame : public Player
{

public:
    PlayerMarbleGame(const Player& player) : Player(player) { playing = true; };
    virtual bool act2() { return false; }; // 플레이어 2명이서 하는 게임이라, player1, player2의 act를
                                           // 각각 정의해주어야 하는 경우가 있어 act2()함수를 가상함수로 만들어줌

    virtual void checkMarble() {}; // 각각의 게임에서 구슬 체크 함수 구현
    void dyingMessage(); // 죽음 메세지 출력 함수
};
```

```
//홀짝 게임 player class
class PlayerOddOrEven : public PlayerMarbleGame
{
    //구슬의 개수를 고르는, 그리고 구슬의 개수를 예상하는 random함수 2개 생성
    static std::uniform_int_distribution<int>randMarble_choose;
    static std::uniform_int_distribution<int>randMarble_expect;
    //홀짝 판별과, 개수 판별을 위한 bool 멤버 선언
    bool check_OddOrEven = false;
    bool check_num = false;
public:
    std::pair< int, int> player1_marble;  // player 1 의 구슬 저장 pair < 홀짝 구분 , 구슬 개수>
    std::pair< int, int> player2_marble; // player 1 의 구슬 저장 pair < 홀짝 구분 , 구슬 개수>
    void CompareOddorEven(const std::pair<int, int>& player1_marble, const std::pair<int, int>& player2_marble); // 홀짝 구분
    bool hasEnoughMarbles() { return this->current_marble >= 40; }; // 게임 통과 체크를 위한 함수
    int current_marble = 20; // 현재 구슬 개수
    PlayerOddOrEven(const Player& player) : PlayerMarbleGame(player) { playing = true; };
    bool act(); // player1 행동 함수
    bool act2(); // player2 행동 함수
    void checkMarble(); // 구슬 개수 확인 및 재분배 함수
    void dyingMessage(); // 죽음 메세지 출력 함수

};
```

```
//구슬 넣기 게임 player class
class PlayerHole_In_One : public PlayerMarbleGame
{
    static std::uniform_int_distribution<int>randHole_distance; // 구멍까지의 거리를 생성하는 random함수 생성
public:
    int current_marble = 10; // 구슬이 구멍에 들어가기만 하면 무조건 종료이므로, 구슬의 개수가 턴 수라고 볼 수 있음.
    PlayerHole_In_One(const Player& player) : PlayerMarbleGame(player) { playing = true; };
    bool hasEnoughMarbles() { return this->current_marble > 0; };
    bool act(); // 행동 함수
    void dyingMessage(); // 죽음 메세지 출력 함수

};
```

3.3. 홀짝 게임 구현 (Odd Or Even)

```
// 두 플레이어의 홀짝 예상과, 개수 예상을 확인하는 함수
void PlayerOddOrEven::CompareOddorEven(const std::pair<int,int>& p1, const std::pair<int,int>& p2)
{   //.first가 홀짝 판단, .second가 개수 판단
    if (p1.first == p2.first && p1.second == p2.second) // 사실 p1.second == p2.second 만 해도 되지만, 코드 읽기 쉽게 하기 위해
    {
        check_OddOrEven = true;
        check_num = true;
    }
    else if (p1.first == p2.first) // 홀짝만 맞췄을 경우
    {
        check_OddOrEven = true;
        check_num = false;
    }
    else { // 틀린 경우
        check_OddOrEven = false;
        check_num = false;
    }
}
```

```
//player가 구슬의 개수를 결정하고, 홀짝을 판별하는 함수 -> 구슬의 개수를 정하기만 하니, 항상 true 반환
bool PlayerOddOrEven::act()
{
    int marbleRange = 1 + (Player::fearlessness) / 15;
    std::uniform_int_distribution<unsigned int> randMarble_choose(1, marbleRange);
    int chooseMarble_num = randMarble_choose(random_engine);

    if (chooseMarble_num % 2 == 1)
    {
        player1_marble = std::make_pair(1, chooseMarble_num); //pair의 첫번째 원소에는 홀수면 1, 짝수면 2를,
                                                              //두번째 원소에는 걸은 원소의 개수를 집어넣는다.
    }
    else
    {
        player1_marble = std::make_pair(2, chooseMarble_num);
    }
    return true;

};
```

```cpp
bool PlayerOddOrEven::act2()
{
    const int known_marble = player1_marble.second;//known_marble을 가지고 player의 agility를 활용할 수 있는 방법

    //agility 크기에 따라 탐색 범위를 줄여주자
    int expect_MarbleRange_start = known_marble - (100 - agility) * 0.09; // 이후 비율 조절이 필요해 보임
    int expect_MarbleRange_end = known_marble + (100 - agility) * 0.09;
    //범위 내의 구슬 개수를 랜덤으로 고르게 랜덤 함수를 재정의 해줌

    //player 2가 구슬 개수를 예측
    randMarble_expect = std::uniform_int_distribution<int>(expect_MarbleRange_start, expect_MarbleRange_end);

    int expectedMarble_num = randMarble_expect(random_engine);
    if (expectedMarble_num % 2 == 1)
    {
        player2_marble = std::make_pair(1, expectedMarble_num); //pair의 첫번째 원소에는 홀수면 1, 짝수면 2를,
                                                                // 두번째 원소에는 예상한 원소의 개수를 집어넣는다.
    }
    else
    {
        player2_marble = std::make_pair(2, expectedMarble_num);
    }
    // plyaer1과 비교 및 구슬 재분배
    PlayerOddOrEven::CompareOddorEven(player1_marble, player2_marble);
    if (check_OddOrEven && check_num)
    {
        //홀짝 및 개수 예측까지 성공했을 시, 15개 획득
        this->current_marble += 15;
        //만약 구슬을 얻어 40개 이상이 되었다면
        if (current_marble >= 40 && isPlaying()) {
            this->playing = false; // playing = false로 바꾸고 탈출 출력
            printStatus();
            std::cout << " safely escaped from the Odd OR Even Game." << std::endl;
        }
        return true;
    }
    else if (check_OddOrEven && !check_num)
    {
        //홀짝만 맞췄을 경우엔 상대방 것 10개 획득
        this->current_marble += 10;
        //만약 구슬을 얻어 40개 이상이 되었다면
        if (current_marble >= 40 && isPlaying()) {
            this->playing = false;// playing = false로 바꾸고 탈출 출력
            printStatus();
            std::cout << " safely escaped from the Odd OR Even Game." << std::endl;
        }
        return true;
    }
    else
    {
        //예상이 틀렸다면, 상대방에게 5개 헌납
        this->current_marble -= 5;
        // 만약 구슬을 잃어 0개 이하가 되어 탈락했다면
        if (this->current_marble <= 0) {
            this->playing = false; // 일단 playing만 false로 바꿈 ->  return false로 처리해줄것임
            //this->dyingMessage();
        }
        return false;
    }
```

```cpp
//현재 객체(player1)에 대해 상대방이 홀짝을 맞췄는지 확인하고, 만약 current_marble이 40개가 넘어가거나 0보다 적으지면 playing false로 바꿈
void PlayerOddOrEven::checkMarble()
{
    // 이 함수는 예상하는 쪽이 아닌 홀짝 내기를 걸은 쪽이 사용하는 함수 -> 상대방이 맞추고 틀리고에 따른 구슬 재분배 함수라고 이해

    //만약 상대방이 맞췄다면 그만큼 뺏겨야 하며, 틀렸다면, 그만큼 얻어야 함
    if (check_OddOrEven && check_num) {
        this->current_marble -= 15; // 상대방이 가져간 만큼 잃음
    }
    else if (check_OddOrEven) {
        this->current_marble -= 10; // 상대방이 가져만 만큼 잃음
    }
    else {
        this->current_marble += 5; // 상대방의 예상이 틀려 구슬을 얻음
    }
    if (current_marble >= 40 && isPlaying()) { // 만약 구슬을 얻어서 탈출하면
        printStatus();// 탈출 출력 및 playing =false
        std::cout << " safely escaped from the Odd OR Even Game." << std::endl;
        this->playing = false;

    }

    if (this->current_marble <= 0) { // 구슬을 다 잃으면
        this->playing = false; // playing = false
        //this->dyingMessage();
    }
    else
        this->playing = true; // 탈출하거나 탈락하지 않으면 계속 playing
}
```

```
void OddOrEven::play() {
    printGameName();
    //두명이 짝을 이뤄 진행하는 게임이므로 두개의 iterator 선언
    auto player1 = players.begin();// 처음 참가자
    auto player2 = std::next(player1); // 바로 그 다음 참가자
    auto lastPlayer = std::prev(players.end()); // 끝 참자가

    std::vector<std::list<Player*>::iterator> playersToRemove; // 삭제할 플레이어를 가르키는 포인터 리스트에 iterator를 vector에 저장

    while (player2 != lastPlayer) {
        bool notPlaying_butAlive = false; // break로 나왔는지 판별하기위해(중간에 나오게되면 playing = false지만, 살아있으므로 x로 처리해줘야 함)
        for (int t = 0; t < 10; ++t) { // 10 턴동안 진행 -> 턴제로 진행할 경우 turn 넣어서 수정
            std::cout << "[Trun #" << t + 1 << "]" << std::endl;
            if ((*player1)->hasEnoughMarbles() || (*player2)->hasEnoughMarbles()) { // 만약 누군가가 이미 40개 이상의 구슬을 획득했다면
                notPlaying_butAlive = true; // 즉시 게임 종료
                break;
            }
            if ((t + 1) % 2 == 1) { // 홀수 턴은 player1이 홀짝 고르기 시작

                if ((*player1)->act() && (*player2)->act2()) { //player1이 임의의 구슬을 정하고 player2가 홀짝을 맞춘 경우(이미 player 2의 구슬은 재정비 됨)

                    if ((*player1)->hasEnoughMarbles() || (*player2)->hasEnoughMarbles()) { // 누군가 40개 이상의 구슬이 있다면
                        notPlaying_butAlive = true; // 게임 종료
                        break;
                    }
                    (*player1)->checkMarble();//player1의 구슬 개수 재정비 및 개수 확인
                }
                else { //act2()가 false라면 예상이 틀린 것이니 구슬을 잃을 수밖에 없음

                    (*player2)->checkMarble(); // player2의 구슬 개수 재정비 및 개수 확인
                }
                if (!(*player1)->isPlaying() || !(*player2)->isPlaying()) { // 만약 checkMarble()함수로 인해 구슬 재정비로 누군가가 탈락하게 된다면
                    notPlaying_butAlive = true; // 게임 종료
                    break;
                }
            }
            else {// 짝수 턴에는 player2가 홀짝 고르기

                if ((*player2)->act() && (*player1)->act2()) { // 위와 마찬가지지만 순서만 다름
                    if ((*player1)->hasEnoughMarbles() || (*player2)->hasEnoughMarbles()) {
                        notPlaying_butAlive = true;
                        break;
                    }
                    (*player2)->checkMarble();
                }
                else { // 짝수턴에 player1의 act2()가 false라면 예상이 틀린 것이고 구슬을 잃을 수밖에 없음
                    (*player1)->checkMarble(); // player1의 구슬 개수 재정비 및 개수 확인
                }
                if (!(*player1)->isPlaying() || !(*player2)->isPlaying()) {
                    notPlaying_butAlive = true;
                    break;
                }
            }
        }

        // 10번의 턴이 끝나고 난 뒤
        std::cout << "[[Game Over]]" << std::endl;
        //10번의 턴 모두 사용하였고 player1과 player2 모두 서로의 구슬을 모두 빼앗지 못하면 죽음
        if ((((*player1)->isPlaying() && (*player2)->isPlaying())) && !notPlaying_butAlive) {
            (*player1)->dyingMessage();
            (*player2)->dyingMessage();
            playersToRemove.push_back(player1);
            playersToRemove.push_back(player2);
        }
        //break문으로 밖으로 나왔을 경우
        else if (notPlaying_butAlive && (*player1)->hasEnoughMarbles()) { //player1이 구슬 40개 이상으로 통과

            (*player2)->dyingMessage();
            playersToRemove.push_back(player2);
        }
        else if (notPlaying_butAlive && (*player2)->hasEnoughMarbles()) {// player2가 구슬 40개 이상으로 통과
            (*player1)->dyingMessage();
            playersToRemove.push_back(player1);
        }


        //다음 두명의 player들로 초기화
        player1 = std::next(player2);
        if (player1 == lastPlayer)// 만약 player1의 다음 player가 존재하지 않는다면 꼭두기로 부전승 시켜줌
            break;
        else
            player2 = std::next(player1);

    }

    if (!playersToRemove.empty()) {//게임에서 탈락한 사람이 존재하면
        for (auto& it : playersToRemove) {
            delete* it; // playersToRemove에는 Player객체를 가르키는 포인터가 존재하니, 그 포인터의 원본 삭제
            it = players.erase(it);
        }
    }

    printAlivePlayers();

    std::cout << players.size() << " players are alive After Odd or Even Game." << std::endl << std::endl;
```

3.4. 구슬 넣기 게임 (Hole In One)

```
bool PlayerHole_In_One::act() // 구슬 굴리기
{
    this->current_marble -= 1; // 구슬을 굴릴때마다 그 객체의 구슬 개수 1개씩 줄어듦
    int holeDistance = randHole_distance(random_engine);
    int throw_distance_start = holeDistance - (100 - agility) * 10; // 이후 비율 조절이 필요해 보임
    int throw_distance_end = holeDistance + (100 - agility) * 10;
    randHole_distance = std::uniform_int_distribution<int>(throw_distance_start, throw_distance_end);

    int player_throw_distance = randHole_distance(random_engine);
    if (player_throw_distance >= (holeDistance - 25) && player_throw_distance <= (holeDistance + 25)) { // 일정 구멍 크기 안에 들어가면 성공
        return true;
    }
    if (this->current_marble == 0) { // 구슬을 모두 다 사용하면
        return false;
    }
    else {
        return false;
    }
}


void PlayerHole_In_One::dyingMessage()
{
    if (isPlaying())
    {
        printStatus();
        std::cout << " failure to insert marbles in hole and died." << std::endl;
    }

    else
    {
        printStatus();
        std::cout << " lost all marbles and died." << std::endl;
    }
}
    if (!playersToRemove.empty()) {//게임에서 탈락한 사람이 없지 않다면
        for (auto& it : playersToRemove) {
            delete* it; // 그 객체 삭제
            it = players.erase(it);
        }
    }

    printAlivePlayers();

    std::cout << players.size() << " players are alive After Hole In One game." << std::endl << std::endl;
}
```

```cpp
void Hole_In_One::play()
{
    printGameName();
    //두명이 짝을 이뤄 진행하는 게임이므로 두개의 iterator 선언
    auto player1 = players.begin();
    auto player2 = std::next(player1);
    auto lastPlayer = std::prev(players.end());

    std::vector<std::list<Player*>::iterator> playersToRemove; // 삭제할 플레이어의 iterator 저장

    while (player2 != lastPlayer) {
        bool check_marble_empty = false;
        static bool check; // 성공해 살아남은 사람 check
        std::cout << "\n[Throw Marble]" << std::endl;

        while (true) { // 구슬을 다 굴리거나 구슬을 구멍 안에 넣을 때까지 반복
            bool check_player1_success = (*player1)->act();
            bool check_player2_success = (*player2)->act();

            if (check_player1_success) { // 만약 선공인 player1 이 성공한다면
                check = false; // 성공한 사람 check
                (*player1)->printStatus(); // player 1 성공 메세지 출력
                std::cout << " safely escaped from the Hole In One Game." << std::endl;
                break;
            }
            else if (check_player2_success) { // player 2가 성공한다면
                check = true; // 성공한 사람 check
                (*player2)->printStatus(); //player 2 성공 메세지 출력
                std::cout << " safely escaped from the Hole In One Game." << std::endl;
                break;
            }
            else if (!(*player1)->hasEnoughMarbles() || !(*player2)->hasEnoughMarbles()) {
                check_marble_empty = true;
                break;
            }

        }
        std::cout << "[Round Over]" << std::endl;
        if (check_marble_empty) {
            (*player1)->dyingMessage();
            (*player2)->dyingMessage();
            playersToRemove.push_back(player1);
            playersToRemove.push_back(player2);
        }
        else if (!check) {
            (*player2)->dyingMessage();
            playersToRemove.push_back(player2);

        }
        else if (check) {
            (*player1)->dyingMessage();
            playersToRemove.push_back(player1);

        }

        //다음 두명의 player들로 초기화
        player1 = std::next(player2);
        if (player1 == lastPlayer)
            break;
        else
            player2 = std::next(player1);

    }
    if (!playersToRemove.empty()) {//게임에서 탈락한 사람이 없지 않다면
        for (auto& it : playersToRemove) {
            delete* it; // 그 객체 삭제
            it = players.erase(it);
        }
    }

    printAlivePlayers();

    std::cout << players.size() << " players are alive After Hole In One game." << std::endl << std::endl;
}
```

4. Project.cpp의 변화

```cpp
//Marble game에서 홀짝 게임을 진행할지, 구슬 넣기 게임을 진행할지 랜덤으로 선택해주는 함수 다른 class에서도 이용할 수 있도록 추상 class인 Game에 구현
int Game::getGameType()
{
    std::default_random_engine random_engine(time(nullptr));
    std::uniform_int_distribution<unsigned int> Choose_game(0, 1);
    unsigned int gameNum = Choose_game(random_engine);
    return gameNum;
};
```

```cpp
for (auto game : games)
{
    if (game->getGameName() == "Marble Game")
    {
        if (game->getGameType() == 0) // 게임 선택 매커니즘을 만들었음
        {
            delete game;
            game = new OddOrEven();
            for (auto player : players)
            {
                game->join(player);
            }
            game->play();
        }
        else
        {
            delete game;
            game = new Hole_In_One();
            for (auto player : players)
            {
                game->join(player);
            }
            game->play();
            break;
        }

        auto alivePlayers = game->getAlivePlayers();
        players.clear();
        for (auto player : alivePlayers)
        {
            players.push_back(new Player(*player));
        }



    }
    else
    {
        for (auto player : players)
            game->join(player);

        game->play();

        auto alivePlayers = game->getAlivePlayers();
        players.clear();
        for (auto player : alivePlayers)
        {
            players.push_back(new Player(*player));
        }
    }

}
```

5. Sample Run

https://github.com/todayoneul/Squid-Game의 Sample Run 파일을 참고해주세요.

6. 아쉬운 점/개선할 점

6.1. 구슬 게임 class 밑에 더 다양한 구슬치기, 구슬 멀리 보내기, 등의 자식 클래스들을 구현해보지 못해 아쉬움.

6.2. 구슬 게임을 구현하려고 하면서 코드가 약간 스파게티 코드가 된 감이 있음.

6.3. 시간의 여유가 있다면 지난 학기에 console창을 제어해 snake-game을 구현한 것처럼 console에 ascii code를 이용해 그림을 그려 게임을 표현한다던가, player가 1명 남을 때까지 게임을 진행해 우승자에게 축하 메시지를 띄워주는 것을 해보고 싶었음.

6.4. 마찬가지로, 설탕뽑기(달고나 게임)도 구현하고 싶었지만 못해 아쉬우며, 징검다리 건너기, 줄다리기 등의 게임도 어렵지 않게 구현할 수 있을 것 같음.