

Text Adventure Game Manager

프로젝트 소개

김가람(20201723)
백민우(20201737)



목차

01

프로젝트 소개

02

프로젝트 계획

03

주차별 개발 내용





01

프로젝트 소개



텍스트 어드벤처 게임 매니저 개발

텍스트를 기반으로 하는 **RPG** 게임으로
분기를 통해 재플레이성을 강화한 스토리 중심의 게임을 제작

플레이어 상태 관리:

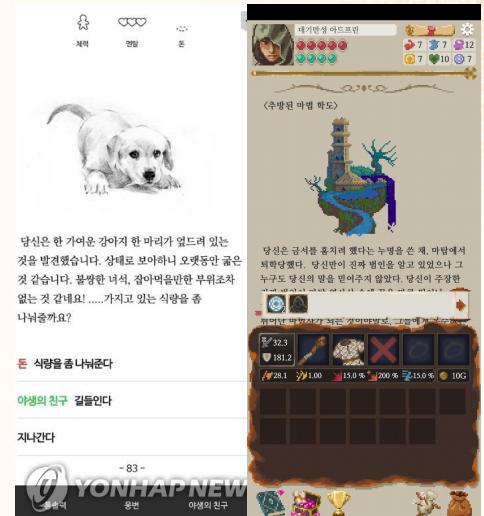
- 체력, 전투 체력, 정신력, 소유 자금으로 플레이어의 상태를 표시
- 전투시, 현재 체력을 반영한 전투 체력이 생성 되며, 해당 전투 체력으로 전투 이벤트를 진행

게임 흐름:

- World Map의 Room을 정의하여 플레이어가 해당 Room에 들어갈 경우 이벤트가 발생
- 이벤트의 발생 후 유저의 선택지에 따라 스토리의 흐름 및 분기를 결정
- 이벤트 안에서 NPC, 적대 Monster, Trap와의 상호작용

능력/아이템 시스템:

- 플레이어가 특정 이벤트에서 능력이나 아이템을 획득.
- 능력/아이템을 사용해 플레이어 status를 변경.



국내 텍스트 어드벤처 장르의 예시
작) 서울2033(반지하게임즈)
우) 모험가 이야기 (Studio Wheel)

해결하고자 하는 문제

재사용성

- Factory 패턴 기반 객체 생성 시스템 설계
- XML 데이터와 게임 로직 분리를 통한 콘텐츠 재사용
- 몬스터, NPC, 능력치 시스템의 모듈화

확장성

- 새로운 몬스터/NPC 추가가 용이하도록 프로토타입 패턴 적용
- XML 기반 맵으로 게임 월드 확장 하기

데이터 관리

- XML 파일을 통한 게임 데이터 외부 관리
 - 플레이어 상태 정보
 - 게임 맵 구조
 - 이벤트 데이터
- TinyXML2 라이브러리로 XML 파일 파싱
- 세이브/로드 시스템을 통한 게임 상태 보존

모듈화

- EventManager를 통한 이벤트 중심 게임 진행
- CombatManager를 통한 독립적인 전투 시스템
- InputManager를 통한 유연한 입력 처리



개발 환경



- 사용 언어: C++
- 빌드 도구: CMake
 - 크로스 플랫폼 빌드 시스템.
 - 대규모 프로젝트의 의존성 관리 및 자동화.
 - 다양한 컴파일러와 개발 환경 지원
- 라이브러리: TinyXML2.
 - Xml를 통한 데이터 관리를 위해 사용
 - 경량 라이브러리로, 빠르고 간단한 XML 읽기/쓰기 기능 제공.
 - 예: 게임의 월드맵(`resources/worldmap.xml`) 정보를 파싱하여 Room 객체로 변환 및 관리.





02

프로젝트 계획



요구사항 항목

요구사항 #1: 엔티티 시스템

- Factory 패턴으로 게임 객체 생성
- MonsterFactory: Dragon, Goblin 등 몬스터 생성 관리
- NPCFactory: DialogNPC NPC 등 생성 관리
- TripFactory: PoisonGasTrap 등 생성 관리
- AbilityFactory: 능력치 시스템 생성 관리

요구사항 #2: 상호작용

- 전투, NPC 대화, 함정 등 게임 내 상호작용
- CombatManger: 전투 시스템
- BaseNPC, BaseTrap, BasedMonster, BaseAbility: 로 상호작용

요구사항 #3: 데이터 관리 시스템

- XML 기반 게임 데이터 관리
- 플레이어 상태 정보 (PlayerStatus.xml)
- 게임 맵 구조 (worldmap.xml)
- 이벤트 데이터 (events.xml)

요구사항 #4: 입력 처리 시스템

- Decorator 패턴으로 입력 확장
- InputManager InputDecorator



03

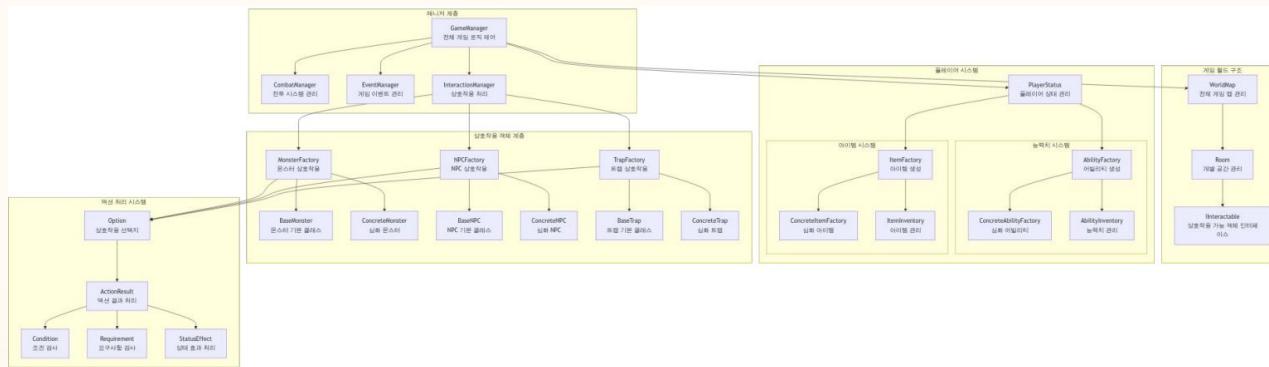
주차별 개발 내용



주차별 개발 내용 (1/4)

1. 프로젝트 구조 설계

- 전반적인 시스템 구조 작성
 - 클래스 다이어그램을 통한 데이터 흐름 및 주요 모듈 간 관계 정의



주차별 개발 내용 (1/4)

1. 매니저 계층 설계

- **GameManager**를 싱글톤 패턴을 통해 설계
 - 게임 전체의 상태를 관리

```
class GameManager {  
public:  
    static GameManager& getInstance();  
    void initializeGame();  
    void startGame(); // 게임 시작  
    void gameLoop(); // 게임 메인 루프  
    void displayOptions(const std::vector<std::string>& options); // 옵션 표시  
    void startCombat(BaseMonster& enemy); // 전투 시작  
    void saveGame(const std::string& saveFileName);  
    void loadGame(const std::string& loadFileName);  
    Event _loadEvent(const std::string& eventId);  
    void displayMessage(const std::string& message);  
    void displayPlayerStatus(); // 플레이어 스테이터스 표시  
    bool deleteSaveFile(const std::string& saveFileName);  
    void displaySaveFiles();  
    void useAbility();  
    void managePlayerAbilityUsage();
```

- **CombatManager** 설계
 - 전투 시스템 동작 구현

```
class CombatManager {  
public:  
    CombatManager(Player& player, BaseMonster* monster);  
  
    void startCombat();  
  
private:  
    Player& player;  
    BaseMonster* monster;  
  
    void combatLoop();  
    void playerAttack();  
    void monsterAttack();  
    void displayCombatStatus() const;  
};
```

```
Combat with Goblin started!  
You attack the Goblin for 20 damage.  
Goblin attacks you for 10 damage.  
Player Combat Health: 90 | Monster Combat Health: 30  
You attack the Goblin for 20 damage.  
Goblin attacks you for 10 damage.  
Player Combat Health: 80 | Monster Combat Health: 10  
You attack the Goblin for 20 damage.  
You have defeated the Goblin!
```

주차별 개발 내용 (1/4)

2. 상호작용 계층 설계

- Monster를 팩토리 패턴으로 설계

```
class BaseMonster {
public:
    virtual ~BaseMonster() = default;

    BaseMonster(const std::string& name, int health, int attackPower);
    virtual void attack(Player& player) = 0;
    virtual std::unique_ptr<BaseMonster> clone() const = 0;

    // Getter 메서드
    const std::string& getName() const;
    int getHealth() const;
    int getAttackPower() const;

    void setHealth(int newHealth);
    bool isCombatDefeated() const;      // 체력이 0 이하인지 여부를 반환
    void takeCombatDamage(int damage); // 데미지를 받고 체력을 감소시킴

    void setAttributes(const std::string& name, int health, int attackPower);

    void displayStatus() const;
}
```

- Concrete Monster인 Goblin

```
class Goblin : public BaseMonster {
public:
    Goblin(const std::string& name, int health, int attackPower)
        : BaseMonster("Goblin", health, attackPower) {}

    void attack(Player& player) override {
        // 고블린의 공격 로직 구현
        std::cout << "Goblin strikes the player with a dagger!" << std::endl;
        player.takeDamage(attackPower);
    }

    // Goblin의 복사본을 생성하는 clone 메서드 구현
    std::unique_ptr<BaseMonster> clone() const override {
        return std::make_unique<Goblin>(*this);
    }
};
```

주차별 개발 내용 (1/4)

3. 플레이어 객체

- 이름, 체력, 공격력 등 관리

```
6  ~ Player::Player(const std::string& name, int health, int mentalStrength, int attackPower, int money)
7    : name(name), health(health), mentalStrength(mentalStrength), attackPower(attackPower), money(money),
8      combatHealth(health), itemInventory(), abilityInventory() {}
9
10 ~ void Player::setName(const std::string& name) {
11   this->name = name;
12 }
13
14 ~ std::string Player::getName() const {
15   return name;
16 }
17 }
18 ~ void Player::setHealth(int health) {
19   this->health = health;
20 }
21
22 ~ int Player::getHealth() const {
23   return health;
24 }
25 }
26
27 ~ void Player::setMentalStrength(int mentalStrength) {
28   this->mentalStrength = mentalStrength;
29 }
30 }
```

4. WorldMap, room

- 이벤트가 발생하는 Room과 관리하는 World Map을

```
class WorldMap {
public:
  void addRoom(const Room& room);
  Room* getRoom(const std::string& roomId);
  std::vector<Room*> getAdjacentRooms(const std::string& roomId);
  std::vector<Room*> getAllRooms();

private:
  std::unordered_map<std::string, Room> rooms;
};

class Room {
public:
  Room();
  Room(const std::string& id, const std::string& name,
       const std::string& description, const std::string& eventId = "");
  void enter();
  void displayDescription() const;
  bool isCleared() const;
  void setCleared(bool cleared);
  bool hasEvent() const;
  std::string getRoomId() const;
  std::string getRoomName() const;
  std::string getEventId() const;
};
```



주차별 개발 내용 (1/4)

5. 리소스 관리

- Events

```
<Events>
  <Event id="event1">
    <name>Entrance Hall</name>
    <description><![CDATA[
A group of goblins attacks you at the entrance. You need to overcome this situation.
]]></description>
    <Monster type="Goblin" name="Goblin Leader" health="50" attackPower="10"/>
    <Choices>
      <Choice id="run" description="Flee to the misty room" nextEventId="eventPosionTrap"/>
      <Choice id="fight" description="Fight" nextEventId="eventPosionTrap"/>
    </Choices>
  </Event>
```

- PlayerStatus

```
<PlayerStatus>
  <Player name="Hero" health="100" mentalStrength="50" attackPower="20" money="100" />
</PlayerStatus>
```

- WorldMap

```
<WorldMap>
  <Rooms>
    <Room id="room1" name="Entrance Hall" description="room1" eventId="event1" />
    <Room id="room2" name="Library" description="room2" eventId="event2" />
```



주차별 개발 내용 (2/4)

1. 게임 매니저에서 이벤트 매니저 분리 분리

```
11 class EventManager {
12 public:
13     static EventManager& getInstance();
14
15     // 이벤트를 로드 (이벤트 XML 파일을 모두 로드하여 캐싱 가능)
16     void loadEvents(const std::string& filePath);
17
18     // 특정 이벤트를 가져오기
19     Event* getEvent(const std::string& eventId);
20
21     // 특정 이벤트를 처리 (플레이어와 이벤트 처리)
22     std::string processEvent(const std::string& eventId, Player& player);
23
24     // 플레이어의 선택을 실행
25     void executeChoice(const std::string& choiceId, Event* currentEvent, Player& player);
26 }
```

2. Event 내부에 선택지 구현

```
class Event {
public:
    // 생성자
    Event(const std::string& eventId, const std::string& eventName, const std::string& eventDescription)
        : id(eventId), name(eventName), description(eventDescription), completed(false), hasAbilityRewardFlag(false) {}

    // 선택지 추가
    void addChoice(Choice&& choice) {
        choices.push_back(std::move(choice));
    }

    // 선택 목록
    const std::vector<Choice>& getChoices() const {
        return choices;
    }

    // 이벤트 실행 함수
    void execute(Player& player);

    // 선택지 화면 표시 함수
    void displayChoices() const;
}
```

3. Action 시스템 제작

- 기본 액션 인터페이스

```
class BaseAction {
public:
    virtual void execute(Player& player) = 0;
    virtual ~BaseAction() = default;
};
```

- Choice를 통해 선택된 Action 사용

```
class Choice {
private:
    std::string id;
    std::string description;
    std::unique_ptr<BaseAction> action;
};
```

주차별 개발 내용 (3/4)

1. 게임매니저에서 인풋매니저 분리

```
class InputManager {  
public:  
    static InputManager* getInstance() {  
        static InputManager instance;  
        return &instance;  
    }  
  
    // 기본 입력  
    virtual std::string getUserInput();  
  
    // Choice 입력 (Event에서 사용)  
    std::string getChoiceInput(const std::vector<Choice>& choices);  
  
    // Yes/No 입력  
    std::string getYesNoInput();  
};
```

2. 매크로를 통한 게임에 객체 자동 등록

```
// 몬스터를 팩토리에 자동 등록하는 매크로 (프로토타입 등록)  
#define REGISTER_MONSTER(TYPE) \  
    namespace {} \  
    struct TYPE##Registration { \  
        TYPE##Registration() {} \  
        auto prototype = std::make_unique<TYPE>("Prototype " #TYPE, 100, 20); \  
        MonsterFactory::registerPrototype(#TYPE, std::move(prototype)); \  
    }; \  
    static TYPE##Registration global_##TYPE##Registration; \  
} \  
#endif // MONSTER_REGISTRATION_H
```

3. 팩토리 구조 개선

- clone()을 통한 객체 생성 방식으로 수정

```
std::unique_ptr<BaseMonster> MonsterFactory::createMonster  
(const std::string& type, const std::string& name, int health, int attackPower) {  
    auto it = prototypes.find(type);  
    if (it != prototypes.end()) {  
        auto monster = it->second->clone(); // 프로토타입을 복사  
        monster->setAttributes(name, health, attackPower); // XML에서 읽은 값을 설정  
        return monster;  
    }  
    return nullptr;
```

주차별 개발 내용 (3/4)

4. InputManager의 입력 표준화 시스템

- 선택지 입력 피드백을 통해 기존 전체 명령어를 쳐야하는 방식에서 약자도 입력 가능한 아래같은 방식으로 수정

1. 기본 입력 처리

```
std::string InputDecorator::getUserInput() {
    std::string input;

    // InputManager 객체를 통해 기본 사용자 입력을 받음
    input = inputManager->getUserInput();
```

2. 입력 전처리

```
// 입력 문자 전처리
std::string InputManager::preprocessInput(const std::string& input) {
    std::string processed = input;
    std::transform(processed.begin(), processed.end(),
                  processed.begin(), ::tolower);
    return processed;
```

3. 선택지 입력 처리

```
std::string input = preprocessInput(getUserInput());

auto it = std::find_if(choices.begin(), choices.end(),
                      [&](const Choice& choice) {
    std::string fullId = preprocessInput(choice.getId());
    return fullId == input ||
           (input.length() == 1 && fullId[0] == input[0]);
});
```

```
while (true) {
    std::string input = preprocessInput(getUserInput());
    if (input == "y" || input == "yes") return "yes";
    if (input == "n" || input == "no") return "no";
    std::cout << "Please enter y/n: ";
}
```

주차별 개발 내용 (4/4)

1. 어빌리티

- 어빌리티 인벤토리 구현

```
class AbilityInventory {
public:
    // 어빌리티 추가 및 제거 메서드
    void addAbility(std::unique_ptr<Ability> ability);
    void removeAbility(const std::string& abilityId);
    bool hasAbility(const std::string& abilityId) const;
    const Ability* getAbility(const std::string& abilityId) const;

    // 어빌리티 목록을 출력하는 메서드
    void displayAbilities() const;
    const std::vector<std::unique_ptr<Ability>>& getAbilities() const {
        return abilities;
    }

    bool isEmpty() const {
        return abilities.empty();
    }
private: // 어빌리티들을 저장
    std::vector<std::unique_ptr<Ability>> abilities;
};
```

- 어빌리티 팩토리 구현 및 매크로로 자동 등록

```
class AbilityFactory {
public:
    static AbilityFactory& getInstance();

    // 어빌리티 프로토타입 등록 메서드
    void registerAbility(const std::string& abilityName, std::unique_ptr<Ability> prototype);

    // 어빌리티 생성 메서드 (프로토타입 복제)
    std::unique_ptr<Ability> createAbility(const std::string& abilityName);

private:
    AbilityFactory() = default;
    std::unordered_map<std::string, std::unique_ptr<Ability>> prototypes;
};
```

2. NPC

- NPC 팩토리 구현

```
class NPCFactory {
public:
    static std::unique_ptr<BaseNPC> createNPC(const std::string& type,
                                                const std::string& name, const std::string& dialog);
    static void registerPrototype(const std::string& type,
                                 std::unique_ptr<BaseNPC> prototype);

private:
    static std::unordered_map<std::string, std::unique_ptr<BaseNPC>> prototypes;
};

#endif
```

3. Trap

- Trap 팩토리 구현

```
class TrapFactory {
public:
    static std::unique_ptr<BaseTrap> createTrap(const std::string& type,
                                                const std::string& name, int damage);
    static void registerPrototype(const std::string& type,
                                 std::unique_ptr<BaseTrap> prototype);

private:
    static std::unordered_map<std::string, std::unique_ptr<BaseTrap>> prototypes;
};
```

주차별 개발 내용 (4/4)

4. Save/Load 기능 구현

- 플레이어 status, worldmap 상에서의 위치 저장

```
// 저장 기능
displayText("Would you like to save your progress? (yes/no): ");
std::string saveChoice = InputManager::getInstance()->getYesNoInput(); // getYesNoInput 사용
if (saveChoice == "yes") {
    displayMessage("Enter save file name: ");
    std::string saveFileName = InputManager::getInstance()->getUserInput();
    saveGame(saveFileName);
}
```

- 게임을 시작할시 , 새로운 게임을 시작 혹은 기존 게임을 Load 할지 선택

자료 출처

사진 자료

- ◆ <https://m.thisisgame.com/hs/nboard/212/?n=101439>

참고 자료

- ◆ <https://www.studiowheel.net/>

THANKS!



CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)