https://github.com/hwang030915/Nuguri-project

프로젝트 보고서: **Nuguri** 게임

본 문서는 고급 자료구조 과목의 팀 프로젝트 결과를 정리한 보고서입니다.

1. 과제 정보

- 과목명 : 고급 자료구조
- 담당 교수님 : 차정원 교수님
- 제출일 :
- 팀명 : TEAM 8
- 팀원 및 역할 :

학번	이름	역할
20222161	황왕석	os별 전처리, github 협업관리
20243120	이윤정	플레이어 점프 충돌 처리 버그 수정, 맵 파일 수정
20223144	최영재	os별 비프음 재생 구현
20253161	박수영	생명 시스템 구현 / 게임 타이틀 출력

2. 프로젝트 개요

프로젝트 목적:
본 프로젝트는 제공된 초기 단계의 게임 소스 코드(nuguri.c)를 분석하여, 이를 완성된 형태의 게임으로 발전시키는 것을 목표로 한다.

- 크로스 플랫폼 프로그래밍 : 운영체제(Windows, Linux, macOS)에 종속적인 시스템 API(입출력 처리 등)를 이해하고, 전처리기(Pre-processer)를 활용하여 모든 환경에서 동작하는 이식성 높은 코드를 작성한다.

- 협업 도구 및 버전 관리 : GitHub를 활용하여 개발 진행 상황을 체계적으로 기록하고 관리하는 습관을 기른다.

- 자료구조 및 알고리즘 응용 : 2차원 배열 기반의 맵 데이터 처리, 충돌 감지 알고리즘 개선, 게임 루프 최적화를 수행한다.

필수 구현 기능

1. 크로스 플랫폼 지원(Cross-Platform Compatibility):
    - 현재 코드는 Windows에서 컴파일 및 실행이 불가능하다.

- #ifdef _WIN32 등이 전처리 지시어를 사용하여, Windows 환경에서는 <conio.h>등을, Linux/macOS에서는 <termios.h>를 사용하여 키보드 비동기 입력을 처리하도록 코드를 수정해야 한다.

- 화면 클리어(cls vs clear), 대기 함수(Sleep vs usleep) 등 OS 의존적인 기능들을 모두 호환되게 구현해야 한다.

2. 게임 완성도 향상:
   - 생명력 시스템 : 플레이어에게 3개의 생명(Heart)을 부여하고, 적과 충돌 시 생명이 감소하여 시작 지점으로 돌아가는 로직을 추가한다.(생명 0이 되면 Game Over).

   - 타이틀 및 엔딩 화면: 게임 시작 전 타이틀 화면과 게임 오버/클리어 시 나타나는 엔딩 화면을 별도로 구현한다.

3. GitHub를 이용한 버전 관리 (Version Control):
   - 프로젝트 시작과 동시에 GitHub Repository를 생성한다.(제출 시 접근 가능해야 함)

   - 기능 하나를구현할 때마다 커밋(Commit)을 수행해야 하며, 최소 10회 이상의 유의미한 커밋 로그가 남아야 한다.

   - 단순히 최종 결과물만 한 번에 업로드하는 행위는 인정하지 않는다.

선택 구현 기능(가산점)
- 사운드 효과 : 각 OS의 시스템 비프음 등을 활용한 간단한 효과음 구현.

3. 개발 일정

기간	내용
1주차	코드 수정, GitHub Repository관리, 통합 및 테스트
2주차	보고서 작성, 코드리뷰(발표 연습)

4. 개발 환경 및 기술 스택

- 개발 환경 : VSCode
- 실행 및 테스트 : cmd(Windows), WSL(Linux,macOS)
- 통합 및 협업 : GitHub

5. 기능 상세 설명

1) 크로스 플랫폼
기존 코드에서는 윈도우에서는 다음과 같은 문제로 인해 실행이 불가능하였다.

#ifdef, #else #endif 사용으로 os별 처리를 통해 실행가능하도록 하였다.

1. 윈도우에서는 사용할 수 없는 함수 사용

```
void delay(int ms){
#ifdef _WIN32
    Sleep(ms); // 단위 : 밀리초
#else
    usleep(ms * 1000); //단위 : 마이크로초, *1000을 해서 Sleep과 단위를 맞춤
#endif
}
```

usleep은 unisted.h 헤더에서 가져와서 시간을 지연 시키는 함수인데 윈도우에서는 사용할 수 없어 windows.h 헤더에서 Sleep()를 불러와서 사용하였다. delay() 함수를 사용해 windows에서는 Sleep(), Linux,mac에서는 usleep을 사용할 수 있도록 하였다.
usleep의 단위는 마이크로초이고 Sleep의 단위는 밀리초이기 때문에 같은 지연 시간을 위해 ulseep는 ms*1000으로 설정했다.

2. clrscr()함수 os별 처리하여 화면 지우기 기능 사용

```
//화면 지우기
void clrscr() {
#ifdef WIN32
    system("cls");
#else
    system("clear");
#endif
}
```

system()은 운영체제 셸 명령을 실행하는 C 표준 라이브러리 함수이고,
윈도우에서는 cmd의 내장 명령 cls가 콘솔 화면 버퍼를 지우고 커서를 (0,0)으로 옮기고(windows console에서는 0,0이 커서 시작 좌표), Linux/mac 에서는 셸에 “clear”명령을 실행하여 clear 프로그램이 ANSI 시퀀스(\x1b[H\x1b[2J)을 보내서 화면을 지우고 커서를 ANSI규격 시작좌표 (1,1)로옮긴다.

3. 윈도우에서 방향키 입력시 게임 진행 방해 문제

```
char read_key() {  // 윈도우는 raw모드가 아니라서 getchar 사용시
                   // 엔터입력 전까지 대기가 계속되는 문제 발생
#ifdef _WIN32
    if (_kbhit()) { // 키보드 입력이 들어왔는지 체크
        int ch1 = _getch(); //getchar대신 _getch사용
                            //윈도우는 방향키 입력이 2바이트로 들어옴
        if (ch1 == 0xE0) { //prefix(특수키)
            int ch2 = _getch(); //2번째 바이트 실제 키 코드

            switch (ch2) {
            case 72: return 'w'; // up
            case 80: return 's'; // down
            case 75: return 'a'; // left
            case 77: return 'd'; // right
            }
            return '\0'; // 방향키가 아니면 무시
        }
        return (char)ch1; //입력한 키 반환
    }
    return '\0';//입력이 없으면 입력 없음 처리
```

```
#else // Linux/mac은 방향키 입력이 ESC 시퀀스 3바이트로 들어옴
    if (LM_kbhit()) { // 키보드 입력이 들어왔는지 체크
        int ch = getchar(); //첫번째 바이트

        if (ch == '\x1b') { //ESC
            int ch1 = getchar(); // 두번째 바이트
            int ch2 = getchar(); // 세번째 바이트

            if (ch1 == '[') {
                switch (ch2) {
                case 'A': return 'w'; // Up
                case 'B': return 's'; // Down
                case 'C': return 'd'; // Right
                case 'D': return 'a'; // Left
                }
            }
            return '\0';
        }
        return (char)ch;
    }
    return '\0';
#endif
}
```

윈도우에서 방향키를 입력하면 getchar로 문자를 받고 엔터를 입력 받을 때까지 게임이 멈추고 터미널에 입력한문자가 출력이 되어 실시간으로 움직이는 너구리 게임 흐름에 문제가 생겼다.
윈도우에서는 termios.h 사용이 불가하기 때문에 raw모드가 적용이 안 되었다. 또한 윈도우와 Linux/mac의 방향키 입력 형태가 달라서 처리를 달리해야 했다.
Linux/mac에서는 방향키 입력형태가 ESC 시퀀스 3바이트로 0x1B, '[', 'A','B','C','D'
윈도우에서는 2바이트로 0xE0(prefix), 2번째 바이트 (실제 키 코드) 로 들어온다.

윈도우에서는 conio.h에서 _getch를 불러와 getchar 대신 화면에 입력문자가 출력되지않고, 엔터 입력 전까지 대기가 없는 _getch를 사용

문제를 해결하기 위해 read_key()함수를 만들어 os별 입력 방식을 함수 안에서 처리해, 메인에서 win_getchar()을 호출하여 os별로 정상적인 입력을 받도록 하였다.

4. 프로그램 실행시 터미널에서 글자가 깨지는 문제

```
int main() {
#ifdef _WIN32
    SetConsoleOutputCP(CP_UTF8);     //출력되는 문자의 코드 페이지를 바꾸는 함수
    SetConsoleCP(CP_UTF8);           //입력되는 문자의 코드 페이지를 바꾸는 함수
#endif
```

Windows 콘솔은 기본적으로 CP949인코딩을 사용한다.
그러므로 Windows 콘솔은 UTF-8 문자를 제대로 처리하지 못해 깨진 문자가 생긴다.

콘솔의 문자 인코딩을 UTF-8로 맞춤으로써 출력,입력 인코딩을 정상적으로 가능하게 했다.

2) 생명력 시스템 기능구현

```
// 전역 변수
char map[MAX_STAGES][MAP_HEIGHT][MAP_WIDTH + 1];
int player_x, player_y;
int stage = 0;
int score = 0;
int life = 3; //플레이어 생명 3개 부여
int start_x, start_y;// 시작 위치
```

플레이어의 생명은 main() 밖에 있는 전역 변수 life 를 통해 관리된다. (게임의 흐름에서 생명 정보가 여럿의 함수에서 공통으로 쓰이기 때문)
게임 시작 전 life 값을 3으로 초기화 하여, 3번의 기회가 주어지도록 하였다.

```
// 게임 화면 그리기
void draw_game() {
    printf("\x1b[H");
    printf("Stage: %d | Score: %d\n", stage + 1, score);
    printf("조작: ← → (이동), ↑ ↓ (사다리), Space (점프), q (종료)\n");
    printf("Life : %d\n", life );
```

printf("Life : %d\n", life); 를 통하여 현재 플레이어의 생명 수를 출력한다.
현재 남은 목숨을 보여주는 인터페이스의 역할은 draw_game() 이 담당하고 있다.

```
if (player_x == enemies[i].x && player_y == enemies[i].y) {
    score = (score > 50) ? score - 50 : 0;

    life--;

    if (life <= 0){
          int retry = gameover();
          if (retry == 1){ //재도전 로직
            life = 3;
            score = 0;
            stage = 0;
            init_stage(); //1스테이지 맵 다시 로드
          }
          else{ //종료 로직
            disable_raw_mode();
            printf("\x1b[?25h");
            exit(0);

          }
    }
```

플레이어가 적과 충돌이 일어나면 life- - 연산을 통하여 플레이어 생명을 1 감소시킨다.
이렇게 줄어든 생명 값이 0이하인지의 여부에 따라서 이후 흐름이 나뉜다.
만약 생명이 없다는 조건문에 걸리면 생명이 모두 소진된 경우를 처리한다.

생명이 0이하가 되면 게임 오버 시 출력되는 타이틀을 가진 gameover() 함수를 호출하게 되고, 플레이어는 재도전의 여부를 선택하게 된다. gameover() 의 반환값이 1이면 재도전이 되므로, 점수와 스테이지를 초기화 시키고 생명값 3을 부여해 다시 시작할 수 있는 로직을 만들어주었다. 그리고는 init_stage()를 호출시켜 첫 번째 스테이지를 다시 진행하게 된다.

그렇지 않을 시(재도전을 하지 않을 시) 게임을 종료하는 로직이 실행된다.
disable_raw_mode( ) 를 호출해 비정규 입력 모드를 원래대로 되돌리고 exit(0)를 통해 프로그램이 종료되도록 구현하였다.

```
        else {
        // 생명 남아 있으면 시작점으로 이동
        player_x = start_x;
        player_y = start_y;

        // 점프랑 속도 값 초기화
        is_jumping = 0;
        velocity_y = 0;
    }
     return;
    }
}
```

생명이 감소하였지만 0이하가 되지 않았을 경우 게임이 즉시 종료되지 않고,
플레이어를 시작 지점으로 되돌리는 방식으로 진행이 된다.
플레이어의 좌표를 start_x, start_y로 돌려 초기 위치로 이동 시킨다.
또, 점프와 속도 관련 상태 변수들도 함께 초기화 시켜줬다.

3) 타이틀 및 엔딩 화면 구현

```
void title(); //타이틀, 게임오버, 게임클리어 함수 선언
int gameover();
int gameclear();
```

title() -> 게임 시작 화면 출력
gameover() -> 게임 오버 화면 출력
gameclear() -> 게임 클리어 화면 출력

필요한 함수들을 정의해두었다.

```c
void title() {
    clrscr();
    printf("\n\n");
    printf("=====================================================\n");
    printf("==                                                 ==\n");
    printf("==            N U G U R I   G A M E                ==\n");
    printf("==                                                 ==\n");
    printf("==-------------------------------------------------==\n");
    printf("==      영차영차 오늘도 모험을 떠나는 너구리       ==\n");
    printf("==      하지만 위험한 뭔가가 돌아다니고 있어       ==\n");
    printf("==-------------------------------------------------==\n");
    printf("==       아무키나 눌러 게임을 시작해보세요!        ==\n");
    printf("=====================================================\n");

    while (1) {
        char key = read_key();
        if (key != '\0') { //입력된 키가 있으면 루프 탈출 후 게임시작
            break;
        }
    }
    clrscr(); //게임 시작 전 타이틀 화면 지우기
}
```

title() 함수는 게임 시작 시 처음으로 보여지는 화면을 구성한다. printf로 UI를 구현하고, while 루프를 이용해 키 입력이 들어올 때까지 대기하도록 구성하였다.

```c
int gameover() {
    clrscr();
    printf("\n\n");
    printf("=====================================================\n");
    printf("==                                                 ==\n");
    printf("==            G A M E   O V E R                    ==\n");
    printf("==                                                 ==\n");
    printf("==-------------------------------------------------==\n");
    printf("==        앗! 목숨 3개를 모두 잃었어 -ㅅ-          ==\n");
    printf("==-------------------------------------------------==\n");
    printf("==            최종 점수 : %-5d                   ==\n", score);
    printf("==                                                 ==\n");
    printf("==            다시 도전해볼까? (r)               ==\n");
    printf("==                나가기 (q)                       ==\n");
    printf("=====================================================\n");

    while (1) {
        char c = read_key();

        if ( c == '\0') continue; //입력 없는 경우 무시

        if (c == 'r' || c == 'R') {
            clrscr();
            return 1; //재도전 1 반환
        }
        else if (c == 'q' || c == 'Q') {
            clrscr();
            return 0;
        }
        else {
            printf("\n!!키를 잘못 입력했습니다!! (r 또는 q를 눌러주세요)\n");
        }
    }
    return 0;
}
```

gameover() 함수는 플레이어의 생명이 모두 소진되었을 때 호출되는 화면을 띄운다. 마찬가지로 printf를 이용해 화면을 출력시키고, 플레이어의 최종 스코어도 확인할 수 있게 해주었다.
또한 조건문을 이용해 재도전 여부를 선택할 수 있게 하였고, r을 누르면 1을 반환하여 다시 게임을 시작할 수 있게 해주었다. 반대로 q를 누르면 0을 반환해 프로그램이 종료가 된다.
예외처리로, 사용자가 키를 잘못 입력하면 경고문을 출력이 된다

(참고)→

```c
if (life <= 0){
      int retry = gameover();
      if (retry == 1){ //재도전 로직
        life = 3;
        score = 0;
        stage = 0;
        init_stage(); //1스테이지 맵 다시 로드
    }
    else { //종료 로직
        disable_raw_mode();
        printf("\x1b[?25h");
        exit(0);

    }
}
```

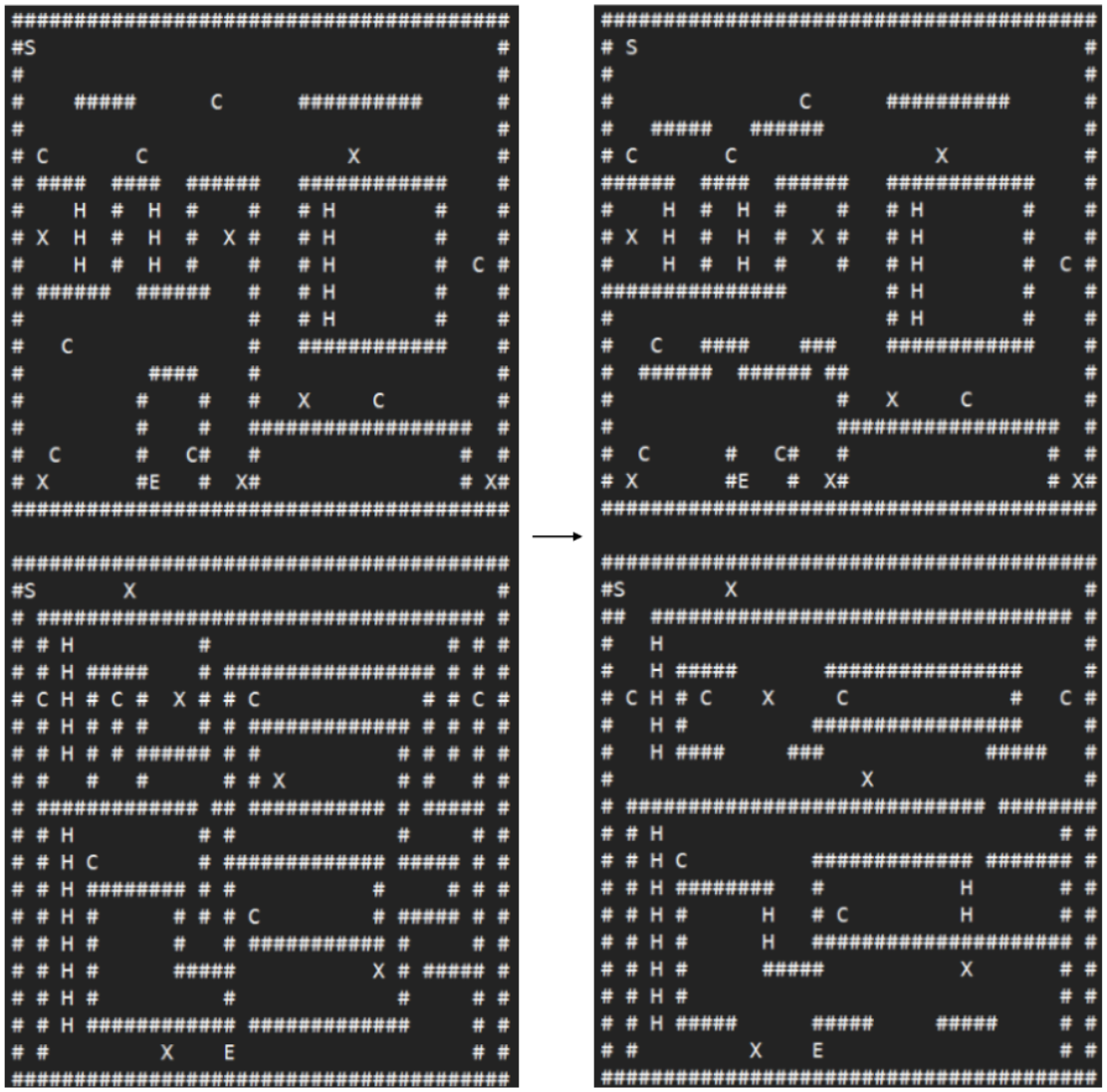
```c
int gameclear() {
    clrscr();
    printf("\n\n");
    printf("=========================================================\n");
    printf("==                                                     ==\n");
    printf("==            G A M E    C L E A R                     ==\n");
    printf("==                                                     ==\n");
    printf("==-----------------------------------------------------==\n");
    printf("==         너구리가 무사히 도착했어 !                  ==\n");
    printf("==     대단해 모든 스테이지 클리어 성공~!              ==\n");
    printf("==-----------------------------------------------------==\n");
    printf("==              최종 점수 : %-5d                       ==\n", score);
    printf("==                                                     ==\n");
    printf("==         타이틀로 가기 (t) | 나가기 (q)              ==\n");
    printf("=========================================================\n");

    while (1) {
        char c = read_key();

        if ( c == '\0') continue; //입력 없는 경우 무시

        //r 타이틀로 돌아가기
        if (c == 't' || c == 'T') {
            clrscr();
            return 1;
        }
        //q 완전 종료
        else if (c == 'q' || c == 'Q') {
            clrscr();
            return 0;

        }
        else {
            printf("\n!!키를 잘못 입력했습니다!! (t 또는 q를 눌러주세요)\n");
        }
    }
    return 0;
}
```

gameclear() 함수는 플레이어가 모든 스테이지를 통과했을 시 호출이 된다.
마찬가지로 printf문을 이용해 타이틀을 출력해주었으며 최종 스코어도 띄워주었다.
while 루프로 키가 들어올때 까지 대기를하며, 조건문을 이용해 흐름을 나누었다.

```c
int go_title = gameclear();

if (go_title == 1){
    score = 0;
    life = 3;
    stage =0;

    title();
    init_stage();
    game_over = 0;
}
```
<-- (참고)
키보드 t를 누르면 1을 반환하여 타이틀 화면으로 돌아가게 되며,
q를 누르면 프로그램을 종료시키는 흐름의 로직을 구성하였다.

4) 맵 오류로 인한 **map.txt** 수정

탈출이 불가능하거나 게임 진행이 어려운 맵 디자인을 수정하였다.



원본 맵의 1스테이지에서 도착지인 E가 벽으로 둘러싸여 있어 클리어를 할 수 없음

이를 도착지 주위의 벽을 제거함으로써 해결함

원본 맵의 2스테이지에서 플레이어의 아래에 바닥이 없어서 게임이 시작되자마자 낙하하여 갇히는 현상 발생.
이를 해결하기 위해 플레이어의 아래에 바닥을 설치하여 원활한 게임 플레이가 가능하도록함

원본 맵의 1스테이지, 2스테이지에서 플레이어가 게임을 진행하는것에 장애가 있도록
바닥이 설계됨.
이를 해결하기 위해바닥을 뚫고 맵의 벽과 바닥 구조를 조정하여 게임 플레이에 장애가
없도록 설계함

5) 각 **OS**별 시스템 비프음을 활용한 효과음 구현

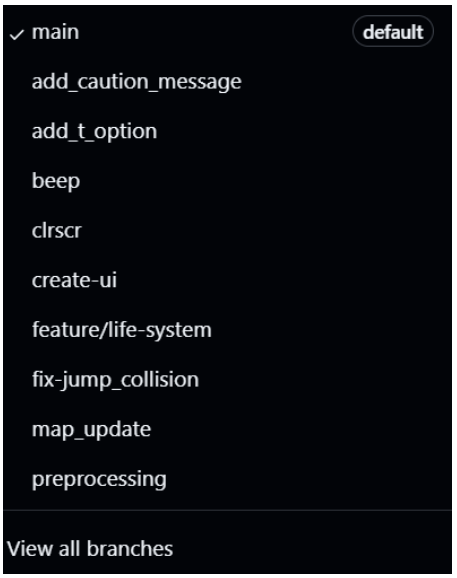
```
// 비프음 함수 정의 (os별)
#ifdef _WIN32
    // 1. 코인 획득 소리: 높은 소리
void play_coin_sound() {
    Beep(1000, 70);
}

// 2. 스테이지 클리어 소리 : 2중 음
void play_clear_sound() {
    Beep(700, 100);
    delay(50);
    Beep(1200, 150);
}

// 3. 충돌/데미지 소리: 낮은 소리
void play_damage_sound() {
    Beep(300, 150);
}
```

#ifdef _WIN32 를 통해 윈도우에서 코인 획득, 스테이지 클리어, 충돌 시 소리를 구분하였다. **windows.h** 에 있는 함수 **Beep**을 사용했으며 인수는 주파수수와 지속시간이다.
클리어나 코인 획득은 긍정적 상호작용이기 때문에 높은 주파수의 비프음으로 설정했으며, 충돌은 경고의 의미로 낮은 주파수의 비프음으로 설정하였다.

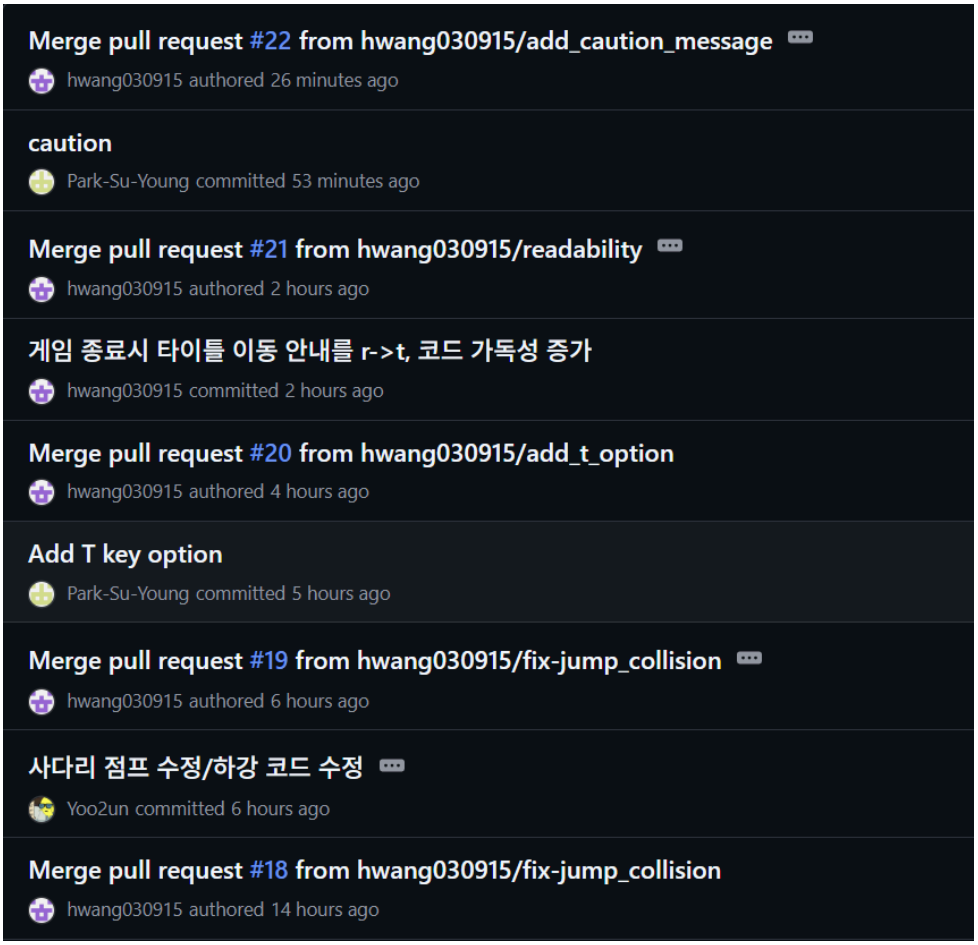
```
#else // Linux/macOS 환경
    // POSIX 환경: 터미널 벨 문자 (₩a) 사용
#include <unistd.h>

// 1. 코인 획득
void play_coin_sound() {
    printf("₩a");
    fflush(stdout);
}

// 2. 스테이지 클리어
void play_clear_sound() {
    printf("₩a");
    fflush(stdout);
}

// 3. 충돌/데미지
void play_damage_sound() {
    printf("₩a");
    fflush(stdout);
}
#endif
```

리눅스에서는 비프음 소리를 따로 설정할수 없어서 동일한 소리로 설정해 상호작용을 알리는 용도로만 비프음을 사용하였다. 리눅스에서는 윈도우에서와 다르게 Beep함수가 없기 때문에 ASCII 벨 문자인 \a를 사용해 소리가 나게 했다. printf("\a")를 호출하면 stdout(표준출력)으로 문자를 내보내고 프로그램을 실행하고있는 터미널로 전달된다. 터미널에서 이 문자를 받으면 사용자에게 시각적인 문자를 표시하는 대신 운영체제의 사운드 시스템에 알림소리를 재생하라는 명령을 보내고 알림소리가 울리게 된다.

6) GitHub 기능 별 commit

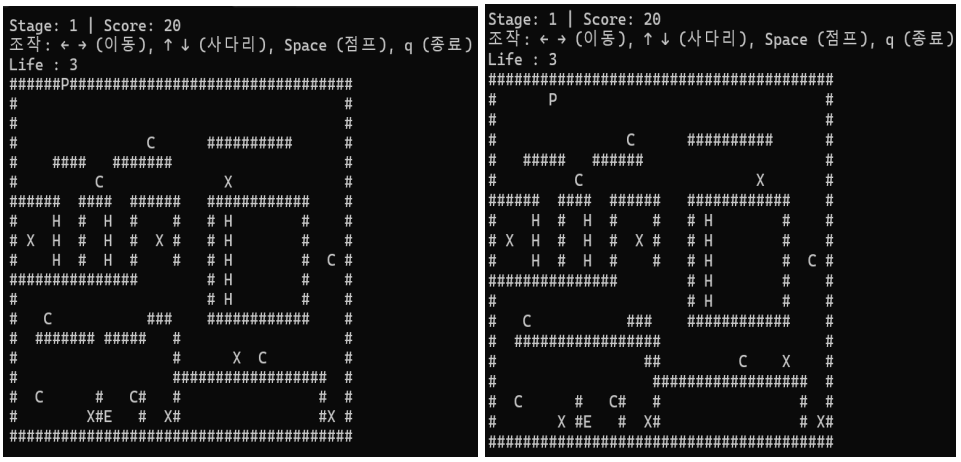


협업을 진행하기 전 각자 구현할 기능을 분업하여 기능별로 브랜치를 생성하였다. 그 후 브랜치에서 작업한 후 PR을 요청하면 조장이 코드를 분석한 후 코드 충돌(conflict)이 생기면 수정한 후 수락 후 merge하였다. 또한 코드 테스트를 해본 후 실행이 되지 않거나, 버그가 생기면 PR을 거절하는 방식으로 Github repository를 관리하였다.



다음과 같이 main 브랜치에 merge하여 최종 코드를 업데이트 하였다.

7) 기타 오류 해결



벽과 충돌할 때 종종 겹치거나 벽을 뚫어버리는 버그를 수정하였다.

6. 컴파일 및 실행방법

Windows 환경일 경우:

1) cmd에서 디렉토리 이동
   cd “nuguri.c 파일이 위치한 디렉토리 주소”

2) 컴파일 시작
   gcc nuguri.c -o nuguri.exe

3) 프로그램 실행
   nuguri.exe

Linux/macOS 환경일 경우:

1) shell에서 디렉토리 이동
   cd “nuguri.c 파일이 위치한 디렉토리 주소”

2) 컴파일 시작
   gcc nuguri.c -o nuguri

3) 프로그램 실행
   ./nuguri

7. 문제 발생 및 해결

문제	원인	해결방법
맵을 뚫어서 통과하는 문제	기존 코드에서는 플레이어 점프 시 map에서 next_y인 부분만 검사하여 # 충돌을 판정했기 때문에, 플레이어와 next_y사이 #있을 때 충돌 판정이 없어도 계산된 next_y 위치로 이동하는 구조여서, 천장이 있어도 점프가 진행되었습니다. 또한 기존 코드에서는 velocity_y++ 연산 이후 if (velocity_y < 0 && map[next_y] == '#') 조건을 검사하기 때문에, 천장이 있어도 velocity_y가 0 이상으로 바뀌면 충돌 감지가 실패하여 천장을 뚫고 이동하는 문제가 생겼습니다.	상승과 하강을 명확히 나누고, 플레이어의 y를 기준으로 velocity_y만큼 한 칸씩 천장과 바닥을 검사하도록 수정했습니다. 중간에 충돌이 발생하면 velocity_y와 is_jumping을 0으로 설정하여 플레이어가 천장이나 바닥과 겹치지 않도록 하였습니다. 다만, 여전히 계산된 next_y 위치까지 점프가 진행되어, 천장이 있어도 점프가 되는 것처럼 보이는 현상은 존재했고, 컴파일 시 렉이 생기는는 문제가 발생하였습니다.
for문 방식의 문제	계산된 next_y 위치까지 한 번에 이동하는 구조여서,	새 코드에서는 next_y = player_y ± 1로 한 칸씩

	천장이 있어도 점프하여 천장을 넘는게 가능하였고, 컴파일시 렉이 발생하는 문제가 있었습니다.	이동하면서 # 충돌 여부를 확인하도록 하였고, 천장이 있는 경우 플레이어의 player_y를 업데이트하지 않음으로써 천장을 통과하는 문제를 방지했습니다. 불필요한 for문과 계산은 지우고 while문 하나로 렉 발생을 최소화하려고 하였습니다.
#에 p가 겹쳐지는 문제	하강 중일 때, 바로 옆의 # 위치로 이동하면, 플레이어가 #과 겹쳐 표시되는 문제가 있었습니다.	기존에는 x값을 처리하고 점프하는 부분이 나와서 겹침이 발생했습니다.점프/하강 처리가 완료된 후에 x좌표 이동을 처리하도록 순서를 변경하여, 플레이어가 #과 겹치지 않도록 수정했습니다.
리눅스 비프음 실행시 소리가 한번만 나는 문제	printf("\a\a\a")로 충돌 시 세번 비프음이 나게 초안을 작성했는데 실행해보니 한번만 나는 문제	리눅스에서는 여러개의 \a를 받으면 이를 경고로 인식해 뒤따르는 벨 문자 \a를 억제해서 하나로 통일 적정한 delay 이용해서 클리어, 충돌, 코인 소리를 구분하려 했으나 하나로 통일했을 때보다 부자연스러워 상호작용했음을 알리기 위한 용도로 하나로 통일했음
게임에서 생명을 모두 소진하고 다시하기를 하였을 때 전에 게임 출력이 남아있는 문제	전에 게임을 끝나고 다시하기를 하면 다시 맵을 그리게 되는데 전에 게임 맵을 지우지 않아 출력이 남아있었습니다.	다시하기, 종료 시 화면을 지우는 clrscr 함수를 호출하여 게임 출력이 남아있지 않도록 수정하였습니다.

9. 결과 및 회고

배운 점 :

20222161 황왕석 - 이번 프로젝트를 통해 게임의 기능을 구현하면서, 다양한 실무 능력을 기를 수 있었던 경험이었다. 특히 windows 와 Linux/macOS 환경에서 모두 실행되도록 크로스 플랫폼 프로그래밍 능력을 확보한 점이 큰 성과였다. 2차원 배열 기반 맵, 플레이어와 코인, 적에 대한 충돌 처리, 중력, 점프, 사다리 이동 등 다양한 게임 메커니즘을

배열, 조건문, 구조체 등 자료구조 개념을 실제 게임 로직에 적용하여 자료구조 활용 능력과 문제 해결 능력을 더욱 발전시킬 수 있었다. 또한 GitHub을 활용하여 팀원들과 브랜치를 나누어 개발하고 PR 기반으로 코드를 병합하는 협업 방식을 실제로 수행하면서, 버전 관리와 협업에 대한 실전 감각을 쌓을 수 있었다. 충돌 해결, 코드 리뷰 등 협업 개발의 주요 요소를 경험할 수 있는 좋은 기회가 되었다.

20223144 최영재 - 전처리기를 통해 리눅스나 윈도우같이 각기 다른 환경에서 동일한 코드가 작동하도록 전처리기 지시어를 사용해 os별 api를 분리하고 통합하는 법을 배울 수 있었다.
GIthub를 활용한 팀 프로젝트는 개발 과정의 체계성을 높여주는 경험이었다. 지속적인 커밋을 수행하고 개발 진행 상황을 체계적으로 기록하고 관리하는 습관을 형성할 수 있었다.

20243120 이윤정 - 이번 과제를 통해 행열 좌표를 이용해 플레이어의 위치를 관리하고 점프와 낙하를 구현하는 방법을 직접 이해할 수 있었다. 또한 기존에 작성된 코드의 구조가 복잡했는데, 하나씩 따라가면서 어떤 부분에서 버그가 발생하는지 분석하는 과정에서 코드 흐름을 읽는 능력이 조금이나마 향상되었다고 느꼈다. 그리고 개인 브랜치에서 작업한 뒤 PR를 생성하여 코드를 병합하는 과정을 처음으로 경험하여 개발에서 협업이 어떻게 이루어지는지도 간접적으로나마 알 수 있었다.

20253161 박수영 - 프로그램에 소리를 입히는 코드를 처음 접함으로써 생동감 있고, 프로그램의 퀄리티를 높일 수 있는 방법의 다양함을 실감했다. 게임 구현 시 단순한 출력이나 이동만이 아닌
kbhit 나 getchar와 같은 입력 처리 문제도 함께 고려해야 하는 것을 깨달았다. 코딩의 기초가 되는 조건문들을 작성하며 게임을 완성시켜 나갔지만, 간단한 게임이라도 함수 여러개가 맞물려 있어 생각보다 더 깊은 세세함과 디테일에 신경써야함을 배웠다. 아직은 미숙하지만 git을 사용해보며 어떻게 코드가 원격저장소에 올라가고 팀원들의 코드가 하나의 브랜치로 병합이 되는지의 흐름을 이해할 수 있었다.

아쉬운 점 :

20222161 황왕석 - Windows 환경에서 raw mode가 아닌 점을 생각하지 않고, Linux/macOS 환경과 같이 getchar를 사용했다가 엔터 입력 전까지 대기, 화면에 입력 키가 출력되는 문제가 생겼다. git을 사용한 협업 프로젝트가 처음이라 commit을 개인 브랜치에서 하지 않고 main에서 하는 실수가 빈번하게 일어났다. Github 협업관리를 기능별 브랜치를 만들어 커밋을 하는 방식을 사용했는데, 이 방식보다 개인 브랜치를 각각 만들어서 구현기능을 커밋로그로 남기는 방식이 더 깔끔하고 누가 어떤 기능을 구현했는지 시각적으로 알아보기 편한 것 같다. 다음 협업 때부터는 후자 방식을 선택해서 협업관리를 해보고 싶다 그리고 PR로그를 생성하지 않도록 설정을 했더라면 커밋로그가 더 체계적이고 예쁘게 보였을텐데 그 부분이 아쉽다.

20223144 최영재 - 리눅스 비프음 재생 부분에서 윈도우처럼 소리들을 구분할 수 있게 하고 싶었는데 최선의 방법을 찾지 못한 것에 아쉬움이 남는다.

20243120 이윤정 - for문에서 렉이 발생하여 while문 하나로 변경했지만, velocity가 커지면 여전히 렉이 발생할 가능성이 있어 아쉬웠다. 또한 깃에 코드를 올릴 때 각 기능별로 계속 수정하면서 커밋 설명과 기록이 깔끔하지 못했다. 다음부터는 코드를 꼼꼼히 확인하고, 오류 없는 상태에서 정리하여 올리도록 하고 싶다.

20253161 박수영 **-** git을 활용한 협업방식이 익숙하지가 않아 브랜치 생성을 빼먹거나, 클론을 안하고 커밋을 해 코드 전체가 새로 추가된 것 처럼 보이는 등 코드를 올리는 과정에서 비교적 많은 시간이 소요된 점이 다소 아쉽다. 중간중간에 반복되는 부분이 있어 코드 유지보수에 있어서 아쉬움이 느껴졌으며 다음은 초기에 함수와 코드 흐름을 명확하게 잡아 구조적인 완성도도 높이고싶다. 타이틀 출력에서 문자들로 그림을 그려 조금 더 생동감 넘치는 UI를 구현하고 싶었지만 생각보다 문자열 배치가 어려워서 간단한 박스모양으로 처리한 것이 아쉬웠다.