

STL 과제2 보고서

2012180041 정다훈

< 목 차 >

1. 프로그램에 대한 설명

- 가. 프로그램 개요 3.
- 나. 프로그램 상세 설명 4.
- 다. 프로그램 조작법 8.

2. 과제 해결방법

- 가. 개발 범위 9.
- 나. Container의 선택 10.
- 다. 데이터의 저장 13.
- 라. FPS 15.
- 마. 리플레이 기본 기능 구현 20.
- 바. 리플레이 배속 기능 구현 24.
- 사. 리플레이 이동 기능 구현 27.

3. 과제를 마치고 느낀 점

- 가. 과제를 마치고 느낀 점 및 제안 30.

[프로그램에 대한 설명]

가. 프로그램 개요

과제를 받았을 때 무슨 프로그램을 사용할까? 고민 하던 중 미리 만들어 놓은 opengl 게임을 이용하는 것 보다는 한 번 더 공부 하는 느낌으로 콘솔 기반의 프레임워크를 직접 제작, 비행 슈팅 게임으로 목표를 결정 하였습니다.

(그림1) 프로그램 개요

목록	
환경	콘솔, 멀티바이트
사용언어	C,C++
프레임 워크	직접 제작

실제 프로그램에서 사용한 프레임워크는 게임 구현에 필요한 다음과 같은 아주 간단한 기능들만 구현하였습니다.

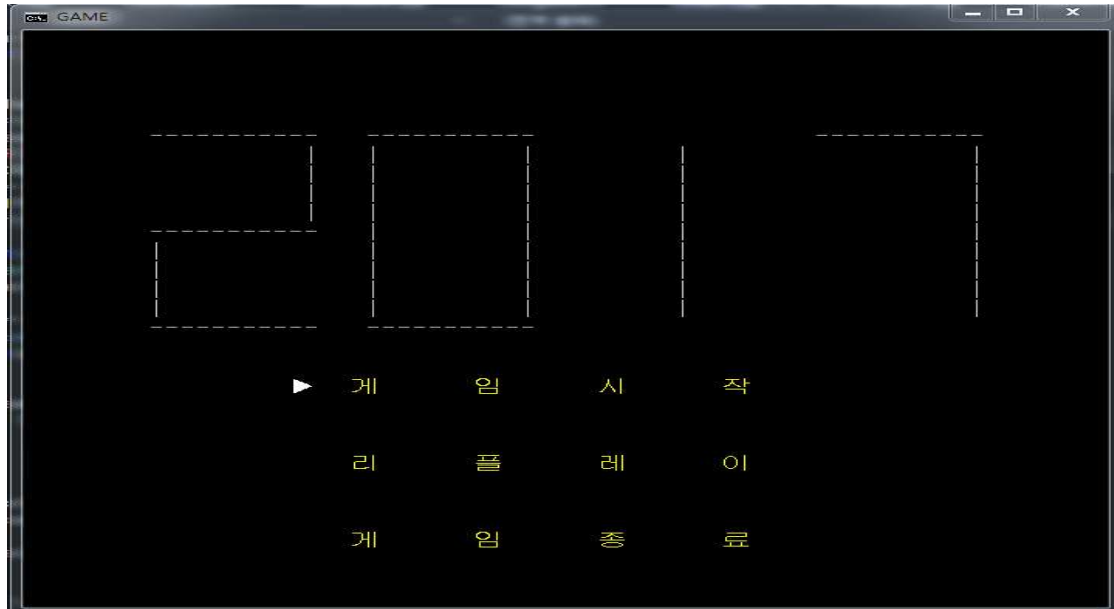
1. 콘솔 타이틀, 창 크기 조절 가능
2. STL 벡터로 관리되는 썬 및 싱글톤을 이용한 썬 전환
3. 게임에 필요한 키보드, 업데이트, 랜더링, 초기화, 제거 함수들
4. 콘솔 더블 버퍼링

이렇게 만들어진 프레임워크를 이용하여 본 과제에서 사용할 비행 슈팅게임을 제작하였습니다.

나. 프로그램 상세 설명

본 과제에서 사용한 프로그램은 콘솔 기반 슈팅 게임입니다.
게임의 각 화면 마다 설명 하도록 하겠습니다.

(그림2) 게임의 메뉴 화면



프로그램을 실행하면 가장 먼저 볼 수 있는 게임의 메뉴화면으로써
게임 시작, 리플레이, 게임 종료 총 3가지 메뉴를 선택 할 수 있습니다.
키보드 위, 아래 방향키로 메뉴 선택 커서를 움직일 수 있고
엔터키로 메뉴를 선택할 수 있습니다.
좀 더 자세한 키 설명은 (다) 조작법에서 자세히 설명 하도록 하겠습니다.

(그림3) 게임의 게임 진행 화면



빨간색 H는 플레이어입니다. M는 몬스터입니다. 별 모양은 총알입니다. 게임의 진행화면으로써 밑에 UI로 현재 히어로의 목숨, 필살기 등과 점수, 진행시간을 표시하였습니다. 필살기 같은 경우에는 현재 화면에 모든 몬스터들을 처치합니다. 히어로가 죽게 되면 목숨 한 개가 차감되면서 현재 화면에 모든 몬스터들을 처치 한 후 부활하게 됩니다. 몬스터는 두 가지 타입으로 연두색과 그림3에 보이는 진한 연두색으로 표시 하였으며 진한 연두색을 가지는 몬스터는 일정시간마다 총알을 발사합니다. 게임은 지속적으로 몬스터가 생성 된 후에 모든 몬스터(.txt 파일에 있는 모든 몬스터)가 없어진 후 보스 몬스터가 생성됩니다. 그 후 보스 몬스터가 처치한 거나, 히어로의 모든 목숨이 끝나면 다시 메뉴화면으로 돌아가면서 자동으로 리플레이 파일을 저장하도록 하였습니다. 리플레이 파일은 RePlay1,...2 순으로 저장되게 됩니다.

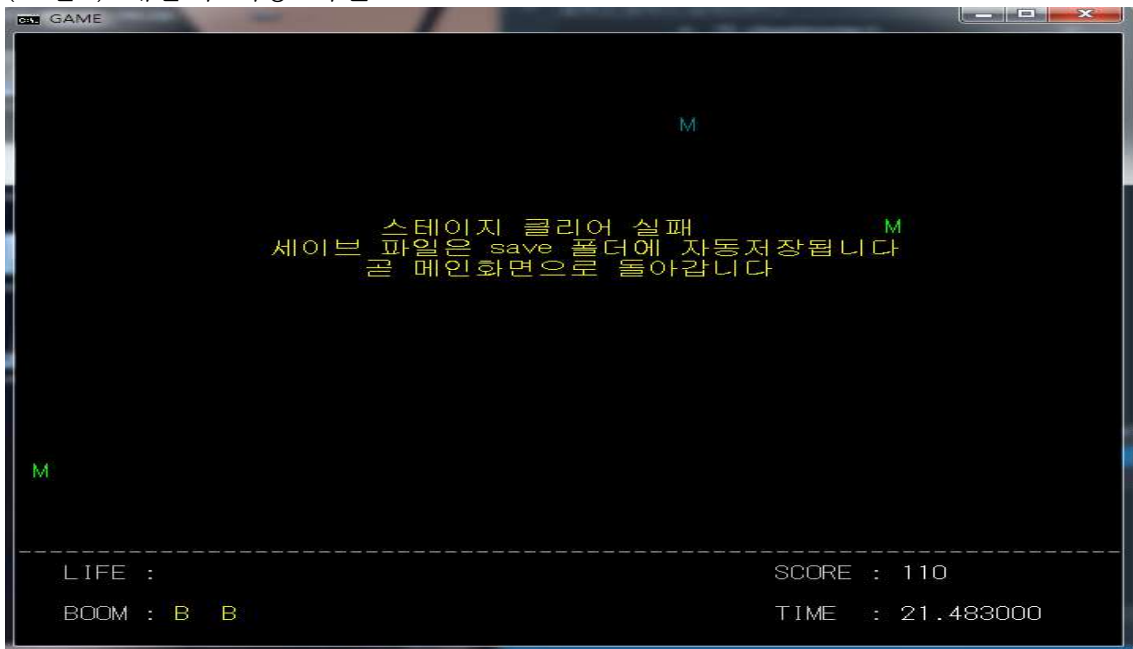
(그림4) 게임의 보스 진행 화면



모든 몬스터를 처치 한 후 등장하는 보스입니다.

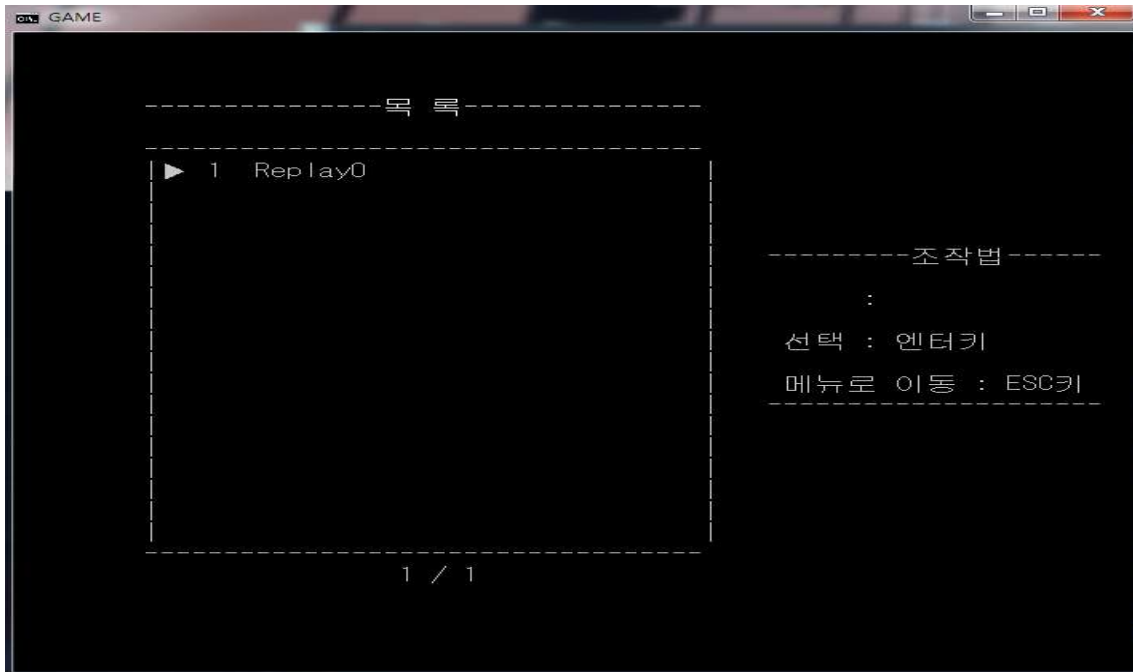
보스는 총 3가지 패턴을 바탕으로 공격하며 초기 체력은 250입니다.

(그림5) 게임의 최종 화면



모든 목숨을 다 하거나 보스를 클리어 하게 되면 다음과 같은 화면이 나오면서 동시에 리플레이 파일을 저장한 후 메인 화면으로 돌아가게 됩니다.

(그림6) 리플레이 선택 화면



리플레이 선택화면으로써 자동 저장된(RePlay0...) 파일을 커서 움직임으로 선택하여 선택한 리플레이를 볼 수 있습니다.

(그림7) 리플레이 진행 화면



리플레이 진행화면으로 밑에 시간, 현재 상태, 현재 배속 등을 볼 수 있습니다. 자세한 조작법은 바로 다음에 설명하겠습니다.

다. 프로그램 조작법

각 조작법은 각 씬마다 설명하겠습니다.

(그림8) 메뉴 화면에서의 조작법

조작키	기능	비고
키보드 상, 하	메뉴 커서 이동	
키보드 엔터키	커서 위치에 있는 기능 실행	

(그림9) 게임 화면에서의 조작법

조작키	기능	비고
키보드 상하좌우	히어로 이동	
키보드 스페이스바	총알 발사	
키보드 컨트롤 키	폭탄 발사	전 맵에 몬스터 처치
키보드 ESC	메뉴 화면으로	리플레이 파일 저장 안됨

(그림10) 리플레이 선택 화면에서의 조작법

조작키	기능	비고
키보드 상, 하	선택 커서 이동	
키보드 엔터키	커서 위치에 있는 리플레이 실행	
키보드 ESC	메뉴 화면으로	

(그림11) 리플레이 화면에서의 조작법

조작키	기능	비고
키보드 좌, 우	배속 증가, 감소	최대 8배속
키보드 P,p	일시정지, 재개	다시 입력하면 재개
키보드 M, m	입력 시간으로 이동, 재개	이동할 시간을 입력(1000이 1초), 다시 입력시 재개
키보드 ESC	메뉴 화면으로	

[과제 해결 방법]

가. 개발 범위

처음 과제를 받았을 때 제가 제일 먼저 고민한 것은 컨테이너의 선택, 어떻게 만들까도 아닌 과연 리플레이 기능을 어디까지 개발 할 것인가? 에 대한 개발 범위 고민 이였습니다.

이 개발 범위에 따라 컨테이너를 선택하는 것도 개발하는 방법도 많이 변한 거라고 생각했습니다.

그래서 저는 다음 한 개의 조건을 기반으로 리플레이 개발 범위를 설정하기로 하였습니다.

1. 본 프로그램은 오목 같은 게임이 아닌 실시간게임인 점.

제가 생각하기에 실시간 게임이기 때문에 리플레이 기능에서 한 단계 한 단계 진행 할 수 있는 기능은 만드는 것은 좋은 생각이 아니라고 판단했고 시간 기반으로 만드는 것이 낫다고 판단하였습니다. 때문에

1. 리플레이 도중 자유로운 시간 이동.

2. 배속 기능.

3. 일시정지 기능.

3가지 기능을 구현하고자 결정 하였습니다.

최종적으로, 리플레이 에서 시간 기반으로 시간 이동, 배속, 일시정지 기능을 구현하고자 결정하였습니다. 또한 여러 가지 고민거리를 던져줬으며 뒤에서 좀 더 자세히 설명 하도록 하겠습니다.

나. Container의 선택

개발범위도 설정하고 난 뒤 과제1을 했을 때 공부한 내용과 아래의 조건을 만족하는 컨테이너를 고민하는 도중

1. 끝을 알 수 없는 데이터의 숫자.
2. 시간 순으로 들어오는 데이터.
3. 많은 삽입과 많은 이동
4. 자유로운 접근

4가지 조건을 만족하는 Container중 제가 선택한 것은 vector였습니다.

처음 2번째 조건은 데이터를 컨테이너에 저장할 때 시간 순으로 저장할 것이기 때문에 자연스럽게 정렬 돼 있는 데이터가 저장될 것 이라고 판단하였습니다. 뿐만 아니라 4번의 조건 자유로운 접근은 리플레이 도중

자유로운 시간 이동을 위해 컨테이너의 자유로운 접근, 즉 랜덤 반복자가 필요하다는 말이고 이건 시퀀스 컨테이너가 필요하다고 판단하였고 자연스럽게 실험 할 필요도 없이 Map, Set를 제외시켰습니다.

그 후 3번째 조건인 많은 삽입으로 인해 list를 고민해보았습니다.

하지만 중간 삽입은 아니기 때문에 과제1를 했을 때 한 실험들을 베이스(그저 맨 뒤에 넣은 건 Vector가 빠르다는 결과)로 제외 시켰고 뿐만 아니라 랜덤 반복자를 제공해주지 않기 때문에 또 리플레이를 실행하기 위해 컨테이너 내에서 많은 이동이 있을 것이며 이는 탐색(이동)에 약점을 보이는 list는 적합하지 않다고 판단하였습니다. 이런 이유 때문에 list를 제외 시켰습니다.

이렇게 해서 남은 Vector와 deque와 가지고 고민을 많이 하였는데
 1번째 조건 데이터의 끝을 알 수 없다. 즉 플레이 시간에 따라 저장해야하는
 데이터는 늘어날 것이면 이는 끝을 알 수 없다고 판단하였습니다.
 때문에 벡터의 큰 장점인 reserve를 사용 할 수 없다는 점과 Vector와 달리 일정
 크기를 가지는 chunk 단위로 확장하는 deque이 훨씬 효율적이라고 판단하였습니다.
 그 후 각 컨테이너에 이만 개의 데이터를 넣는 간단한 실험을 해보았습니다.

(그림12) deque의 클래스 이만 개 넣기 - 114ms, Debug 모드

The screenshot shows a C++ IDE with a program that uses `deque<cPlayer>` to store 20,000 `cPlayer` objects. The program uses `chrono::system_clock` to measure the time taken to push back the objects. The output in the command prompt shows the number 114, representing the execution time in milliseconds.

```

_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_CHECK_ALWAYS_DF);
deque<cPlayer> d;
auto start = chrono::system_clock::now();
for (int i = 0; i < 20000; i++)
{
    d.push_back(move(cPlayer()));
}
auto end = chrono::system_clock::now();
cout << chrono::duration_cast<chrono::milliseconds>(end - start) << endl;
    
```

(그림13) Vector의 reserve 없이 이만 개 넣기 - 263ms , Debug 모드

The screenshot shows a C++ IDE with a program that uses `vector<cPlayer>` to store 20,000 `cPlayer` objects without calling `reserve`. The program uses `chrono::system_clock` to measure the time taken to push back the objects. The output in the command prompt shows the number 263, representing the execution time in milliseconds.

```

int main()
{
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_CHECK_ALWAYS_DF);
    vector<cPlayer> d;
    auto start = chrono::system_clock::now();
    for (int i = 0; i < 20000; i++)
    {
        d.push_back(move(cPlayer()));
    }
    auto end = chrono::system_clock::now();
    cout << chrono::duration_cast<chrono::milliseconds>(end - start) << endl;
}
    
```

예상대로 reserve없이는 deque이 미세하게나마 빨랐습니다.

하지만 여기서 한 가지 의문이 생긴 점은 게임의 평균적인 저장 횟수를 알 수
 있다면 Vector의 reserve를 평균적인 저장 개수만큼 잡아 놓고 사용한다면
 메모리의 낭비는 어느 정도 있겠지만 속도면 에서는 훨씬 월등하지 않을까?
 고민을 하였습니다(이는 실제 과제1을 할 때 알게 된 사실입니다). 그래서 게

임을 약 10번 실행하고 난 뒤 결과의 평균을 구해보니깐 약 오천 개라는 데이터가 평균적인 개수로 나타났습니다. 또한 deque은 vector와 달리 맨 앞 삽입이 빠르는데 프로그램 특성상 맨 앞 삽입하는 경우가 없고 또한 데이터를 맨 앞 인덱스부터 차례대로 접근하기는 하나 pop 같은 명령어를 실행하지 않는 점을 보았을 때 deque을 사용하는 건 효율적이지 않다는 생각을 하였습니다. 때문에 결과적으로 본 프로그램에서는 vector를 사용하는 것이 효율적이라고 판단하였습니다.

최종적으로 끝을 알 수 없는 데이터의 개수이기 때문에 Vector 보다 메모리면 에서 약간 유리하다는 점으로 deque을 선택하려고 하였으나, 본 프로그램 특성상 “스택” 기능이 필요하기 때문에 “큐” 기능을 가지고 있는 deque은 좋은 선택이 아니라고 생각했습니다. 또한 메모리면 에서 약간 유리한 점은 게임의 평균 저장 개수만큼 reserve를 사용하는 것으로 메모리 문제를 최대한 개선하려고 하였으며, reserve 사용으로 속도도 deque 보다 빠른 Vector을 선택하였습니다.

다. 데이터의 저장

개발 범위와 컨테이너를 선택 한 뒤 바로 고민 하였던 주제는 데이터의 저장이었습니다. 처음에는 각 시간 마다 현재 씬 정보를 저장하는 게 어떨까? 생각 하였습니다. 하지만 조금 만 더 생각해 봤을 때 현재의 정보는 히어로의 좌표, 총알, 몬스터들의 위치, 몬스터들의 총알, 등등 등 굉장히 많은 정보들을 저장해야하기 때문에 생각 보다 많은 용량이 들것 같았고 또 굉장히 비효율적이라고 생각이 들었습니다. 때문에 이런 저런 고민을 하는 도중 처음에는 히어로의 정보만 가지고 생각하고 이를 조금씩 확대하여 생각을 해보았습니다.

일단 히어로의 정보를 가지고 고민하는 도중 히어로가 입력 될 수 있는 조작키는 키보드 상하좌우, 스페이스 바 5개가 있습니다. 또한 항상 유저의 입력으로 움직이기 때문에 여기서 히어로는 유저의 입력에 인해서 다음 행동이 결정되는 결정론적 요소가 있다는 걸 발견했고, 히어로가 움직일 때 마다 좌표와 현재 히어로의 총알 등의 저장하는 게 아니라 히어로의 즉 플레이어의 키 입력을 저장하는 방법을 생각했습니다.

하지만 키 입력만 저장하면 히어로가 총알이 나갔다 라는건 알지만 그 총알의 지속적인 좌표는 알 수가 없기 때문에 고민을 하는 도중, 그냥 게임에서 쓰는 로직을 그대로 가지고 오면 즉 총알 생성은 데이터를 불러와서 하고 움직이고, 총알 같은 건 게임 로직에 맡기는 방법으로 해도 문제없다고 판단 이 방법을 사용해 과제를 해결하기로 결정 하였습니다.

다음으로 몬스터 같은 경우에는 생성 같은 경우 map파일에 있는 정보를 가지고와서 생성하고 움직임, 총알 모든 부분에서 랜덤적인 부분이 없고 게임 로직으로 돌아가기 때문에 따로 저장 할 데이터가 없다고 판단하여 저장하지 않았습니다.

보스 같은 경우에는 몬스터와 같이 게임 로직이 맡아서 실행시키지만 패턴 같은 경우에는 랜덤적인 요소이기 때문에 패턴은 실행할 때 마다 저장해야한다고 생각해 저장하였습니다.

그렇게 해서 <입력 값, 현재 시간>과 같은 데이터 형식을 가지게 되었습니다. 하지만 히어로의 입력 값인지 보스의 패턴 값인지 알 수가 없기 때문에 맨 앞에 타입을 추가하여 표현해주기로 결정 하였습니다. 저장 할 때는 save 폴더의 현재 파일 개수를 판단하여 자동으로 RePlay0,1,2 순으로 생성되게 하였고 리플레이 메뉴에서 선택하여 선택한 리플레이를 플레이 할 수 있도록 구현하였습니다.

(그림14) 최종적인 데이터 저장 형식

타입	입력 값	현재 시간
(보스, 히어로)	(입력 키, 패턴)	현재 시간(ms)

최종적으로 모든 부분을 게임 로직에 맡기고(충돌체크, 몬스터 생성, 위치변경)등 입력과 랜덤적인 요소로 바뀌는 부분을 <타입, 입력 값, 현재 시간> 으로 바이너리 파일로 저장하여 리플레이 메뉴에서 원하는 리플레이 파일을 진행 할 수 있도록 구현하였습니다.

라. FPS

FPS 즉 초당 프레임 같은 경우에는 실제로 게임을 제작할 때뿐만 아니라 뒤에 배속, 시간 이동 등에서도 굉장히 중요한 부분이고 또한 본 과제를 하면서 제가 많은 고민을 한 부분이기도 합니다.

제 생각이지만 FPS 고려하지 않고 게임을 제작하게 된다면 현재 컴퓨터에서는 잘되나 모든 컴퓨터가 같은 성능 즉 같은 프레임워크 속도를 가지는 게 아니기 때문에 작게는 렌더링 소요시간부터 업데이트 시간까지 게임을 할 수 없는 상황이 발생할 수 도 있다고 생각했습니다.

제가 생각한 아주 간단한 상황은 이렇습니다.

(터무니없는 상황 일수도 있습니다.)

A컴퓨터에서는 렌더링 부분에서는 약 1초에 시간이 걸립니다.

```
while(1)
{
    update();
    render(); // 1초
}
```

그렇다면 1초에 이 while문이 돌아가는 횟수는 update에 따라 다르지만 초당 1번이 안될 수도 있습니다.

B컴퓨터에서는 렌더링 부분에서는 약 0.5초의 시간이 걸린다고 치면 이 컴퓨터에서는 1초에 이 while문이 돌아가는 횟수는 update에 따라 다르겠지만 초당 1~2번 사이가 될 것입니다.

만약 업데이트 부분에서 heroX++; 와 같은 아주 간단한 기능을 한다고 쳐도 A컴퓨터에서는 1초에 아예 안 움직이거나 최대 한 칸을 움직일 것입니다.

하지만 B컴퓨터에서는 1초에 한 칸 혹은 두 칸을 움직일 것입니다.

이런 문제는 제 생각에는 아주 큰 문제라고 판단하였고 이를 해결하기 위해 여러 가지 대안을 고민해보았습니다.

처음으로 생각난 대안은 바로 생각보다 자주 쓰이는 고정 FPS 방식을 생각해 봤습니다. FPS를 30으로 고정을 시켜두고 update와 render를 30번 실행시키면 괜찮지 않을까 생각을 했습니다.

하지만 이 방법에는 2가지 문제점이 있었는데

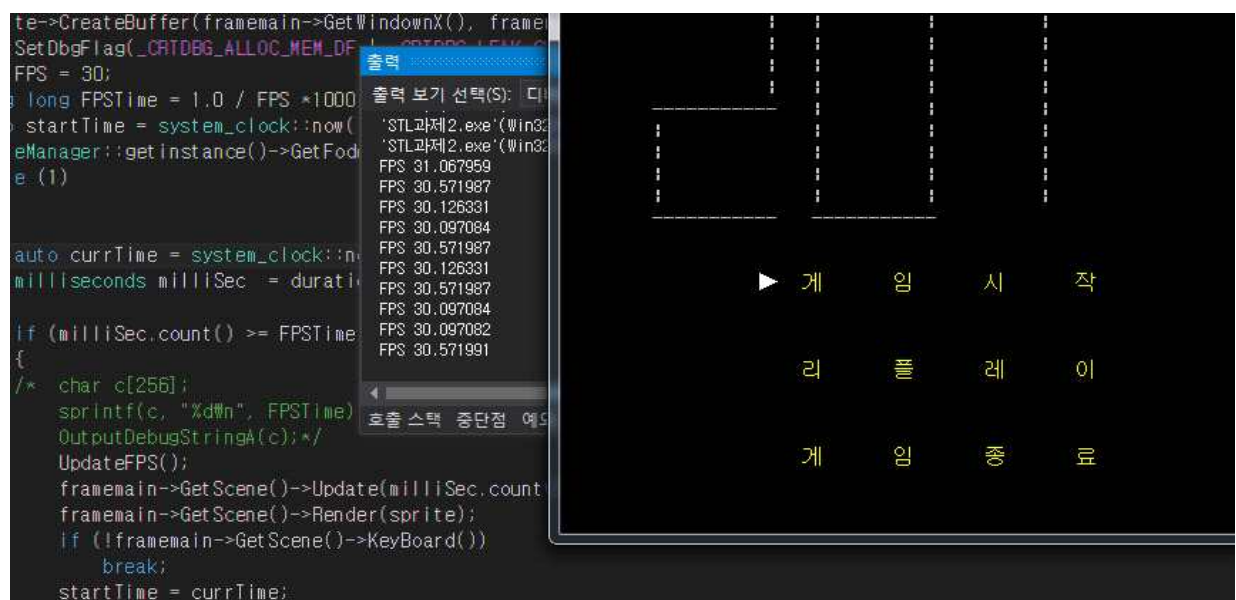
첫 번째 문제점은 빠른 하드웨어에서는 초당 30번 보다 더 많이 소화시킬 수 있는데 이를 억제한다는 점입니다. 하지만 이 문제점 같은 경우에는 게임 성능에는 크게 역할을 주지 않지만 빠른 하드웨어의 성능을 낭비합니다.

두 번째 문제점은 첫 번째 문제점보다 훨씬 더 심각한 문제입니다. 바로 FPS가 30도 나오지 않는 컴퓨터에서는 시간이 왜곡되어 제대로 된 게임을 진행조차도 못할 것입니다.

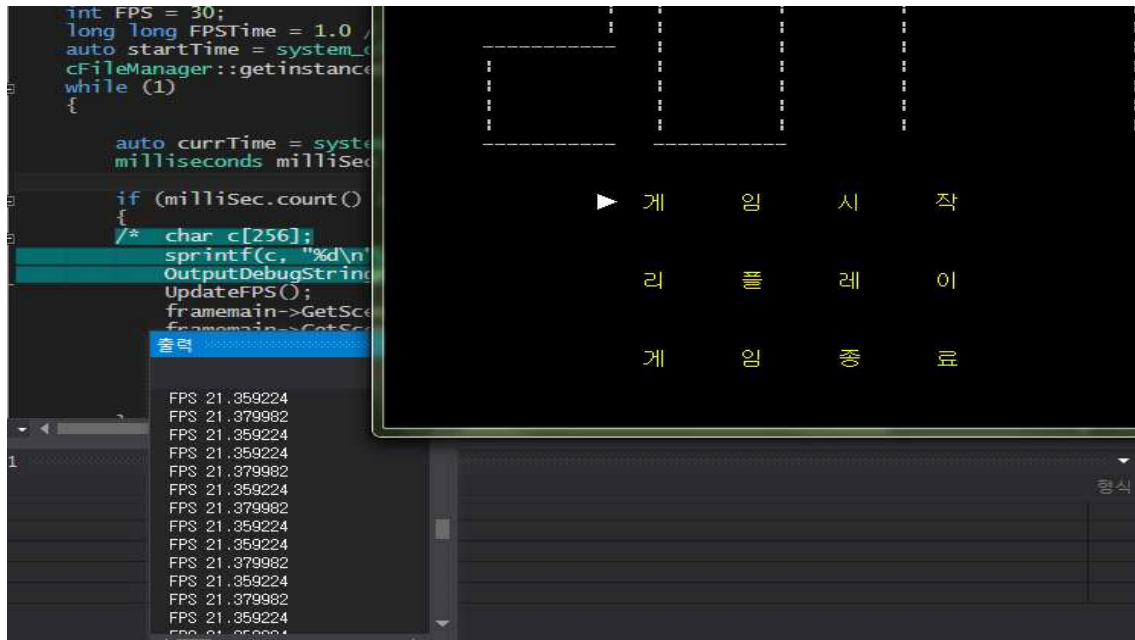
하지만 과연 FPS 30 이 나오지 않는 컴퓨터가 있을까? 고민하는 도중 생각보다 어렵지 않게 발견 할 수 있었습니다. 평상시에 데스크탑을 이용해 프로젝트를 진행하는데 테스트용으로 노트북에서 한번 실행해보았습니다.

생각보다 빠르게 두 번째 문제점을 직접 발견, 확인 할 수 있었습니다.

(그림15) 저의 데스크탑에서의 FPS입니다 (고정 30 FPS에서 30프레임)



(그림16) 저의 노트북에서의 FPS입니다 (고정 30 FPS에서 21프레임)

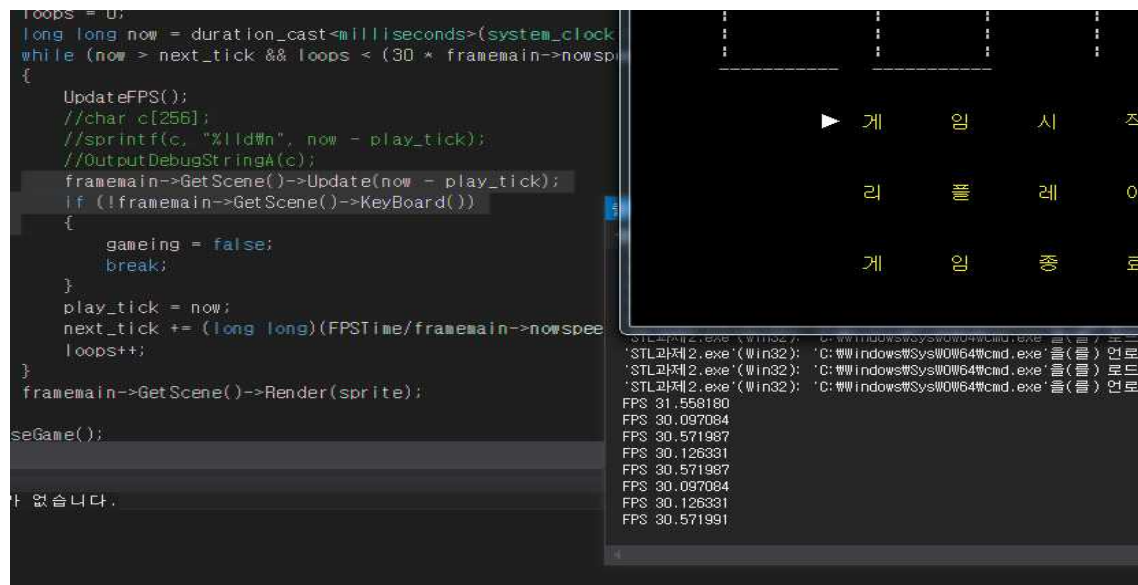


실제로도 데스크탑에서 클리어 한 리플레이가 노트북에서도 클리어 하지도 못하는 경우도 생겼습니다. 이는 30 FPS를 감당하지 못하는 하드웨어 에서는 이미 예상하고 있는 문제였습니다. 때문에 다른 대안이 필요했고 그 다음으로 생각했던 게 가변적인 FPS 이었습니다.

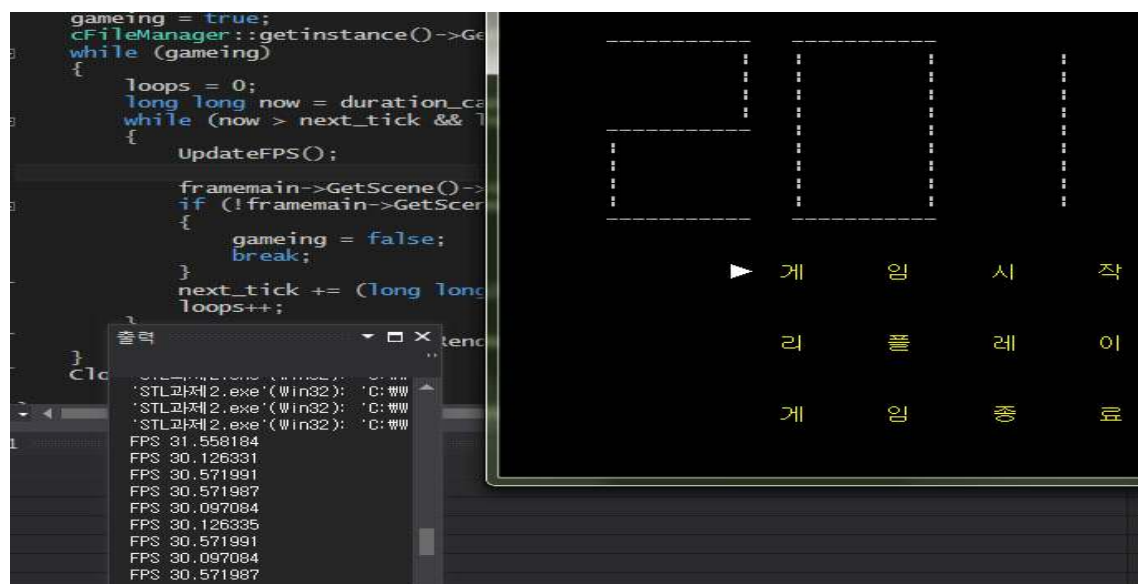
하지만 가변적인 FPS는 해볼 필요도 없이 느린 하드웨어에서는 고정 FPS에서 고정된 FPS 보다 낮게 나오는 하드웨어와 비슷한 현상이 발생하였고 빠른 하드웨어에서는 그만큼 로직이 빠르게 돌기 때문에 소수점 같은 계산을 할 때 컴퓨터의 부동소수점 특징으로 인해 0.1를 50번 더하는 코드와 0.1를 100번을 더하는 코드의 값은 5, 10 이 맞지만 컴퓨터에서는 약간의 오차가 발생합니다. 하지만 여기서 50, 100번이 가변적인 FPS에서 하드웨어 성능 차이에 따라 충분히 차이가 날 수 있는 부분입니다. 이는 각 하드웨어 성능 차이에 따라 계산 값이 미세하게나마 틀려질 수 있다는 이야기입니다. 때문에 이 방법은 적합하지 않다고 생각해 제외시켰습니다.

그다음으로 이런 저런 고민을 하는 도중 고정 FPS에서 렌더링 부분은 많이 호출 하여도 게임 플레이에는 큰 상관없다는 생각을 하였고 업데이트 부분과 렌더링 부분을 나누어서 렌더링 부분은 하드웨어에 따라 진행, 업데이트 부분은 고정 FPS와 같이 30 프레임을 고정하는 방법을 생각했습니다. 거기에 업데이트 부분에 30 프레임만큼 최대한 돌아가게 하면 업데이트 부분에서 오래 걸리게 되면 렌더링 부분의 호출하는 횟수가 줄어들기 때문에 게임에는 큰 지장이 없다고 생각하여 구현, 노트북과 데스크탑에서 실험해보았습니다.

(그림17) 제가 생각한 방법 - 데스크탑 (30 FPS 기준)



(그림18) 제가 생각한 방법 - 노트북(30 FPS 기준)



결과에 보시는 것처럼 30 FPS 양쪽 다 잘 나옵니다.

하지만 렌더링 부분이 하드웨어 성능에 따라 가기 때문에 더블 버퍼링을 했는데도 약간의 깜빡임은 존재하는 게 걸리지만 제 생각에는 이 방법이 그나마 제일 최선의 방법 이었고 이 방법을 최종적으로 결정하게 되었습니다.

최종적으로 FPS은 뒤에 배속, 시간 이동, 하드웨어에 따른 게임의 동일한 결과 보장은 중요한 내용이기 때문에 해결하기 위해 여러 가지 고민을 많이 하였고 고정 FPS, 가변 FPS 등은 앞서 설명 드린 것처럼 많은 문제점을 야기, 렌더링과 업데이트를 나누어서 하드웨어에 따라 렌더링은 최고의 성능을 보장하고자 하였으면 모든 컴퓨터에서 업데이트는 동일한 횟수가 호출되게끔 마지막 방법을 선택, 구현하였습니다.

마. 리플레이 기본 기능 구현

앞서 설명 드린 컨테이너의 선택에서는 vector, 데이터의 파일 저장(바이너리), FPS 구현 등을 완료 한 뒤에 리플레이 배속, 시간이동 등을 구현하기 전에 제일 먼저 리플레이의 기본 기능을 구현해야한다고 생각했습니다. 제가 생각한 기본 기능에는 다음과 같은 2가지입니다.

1. 1배속 리플레이 구현
2. 리플레이의 일시정지

아주 간단해 보이면서, 꼭 필요한 기능이지만 추후에 배속, 시간이동을 구현하기 위해서는 기본 기능이 정상적으로 진행이 되어 한다고 생각했습니다.

1. 1배속 리플레이 구현은 데이터가 파일로 저장돼있습니다. 뿐만 아니라 FPS 문제점을 해결했기 때문에 아주 심플하게 생각했을 때 파일에 있는 데이터를 vector에 넣어놓고 맨 앞부터 체크 하여 경과시간이 데이터의 저장돼있는 시간보다 크거나 같다면 실행 하는 식에 즉 스택과 같은 원리를 이용해 구현하고자 하였습니다. 또한 히어로, 보스 패턴만을 vector에서 사용하고 나머지 몬스터의 생성, 충돌 등은 FPS 문제점을 해결했기 때문에 게임 로직 그대로 사용해도 무방하다고 판단하였습니다.

하지만 이 구현에 있어 한 가지 문제점이 발생하였는데

문제점으로는 게임 특정 시간에 동시에 실행해야 하는 상황이 발생하는 문제점이 있습니다. 좀 더 이해를 돕기 위해 약간의 의사 코드를 보여드리면서 같이 설명 드리겠습니다.

```

now = datas.begin(); //vector의 iterator
if((*now).getTime() >= 경과시간)
{
    // 데이터의 맞는 형식 실행
    // now++;
}

```

부끄럽지만 처음에 테스트용으로 위와 비슷하게 코드를 작성하였습니다.

하지만 이 코드에는 치명적인 문제가 있는데 만약에

<히어로 왼쪽 이동, 1초>

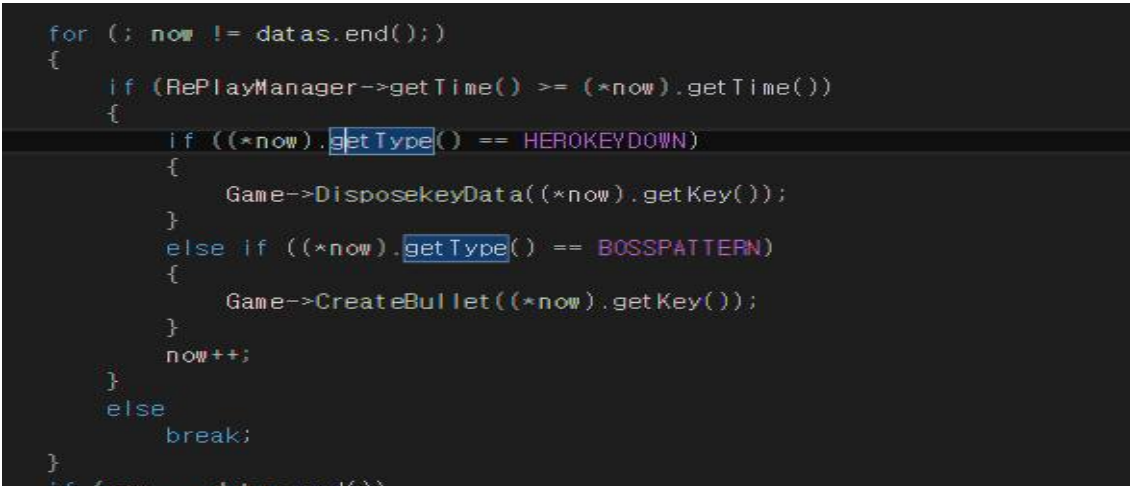
<몬스터 생성, 1초>

<총알 발사, 1초>

와 같은 데이터가 있다면 위에 코드로는 한 개씩 실행되고 그 다음 명령어은 다음 업데이트가 호출되는 시간만큼 지연 될 것입니다.

이런 저런 고민을 하는 도중 vector 안에 있는 모든 데이터를 순차적으로 접근하면서 경과 시간 보다 큰 시간에 있는 모든 명령어를 실행 되도록 구현하였습니다.

(그림19) 실제 구현한 데이터 디코딩 코드



```

for (; now != datas.end(); )
{
    if (RePlayManager->getTime() >= (*now).getTime())
    {
        if ((*now).getType() == HEROKEYDOWN)
        {
            Game->DisposekeyData((*now).getKey());
        }
        else if ((*now).getType() == BOSSPATTERN)
        {
            Game->CreateBullet((*now).getKey());
        }
        now++;
    }
    else
        break;
}
if (now == datas.end())

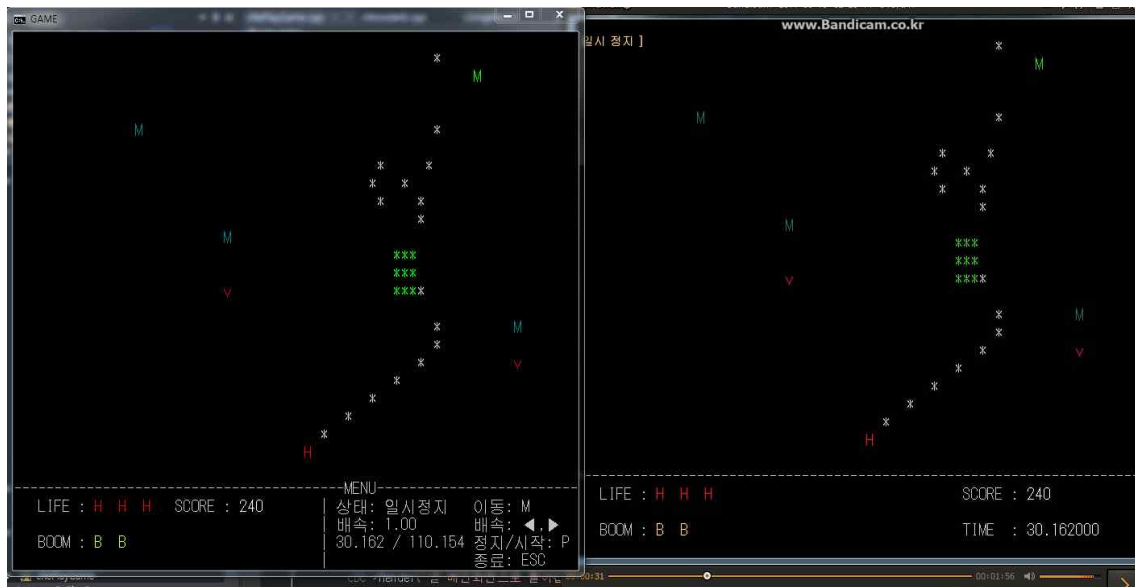
```

그림19와 같이 구현을 하였습니다.

그 후 리플레이 일시정지는 간단하게 일시정지가 되었을 때 update문 자체를 실행시키지 않는 방식으로 구현 완료하였습니다.

이렇게 해서 간단하게 리플레이 기본 기능들을 구현 완료하였으면
구현 보다 더 중요한 테스트를 해보았습니다. 테스트 방식은 실제 게임을 영상
으로 찍어놓고 노트북, 데스크탑에서 실행해 각 각의 시간에 대해서 같은 화면
을 보여주는지 확인 하는 식으로 테스트 하였습니다.

(그림20) 데스크탑에서의 왼쪽 실제 리플레이, 오른쪽 녹음 영상



(그림21) 노트북에서의 왼쪽 실제 리플레이, 오른쪽 녹음 영상

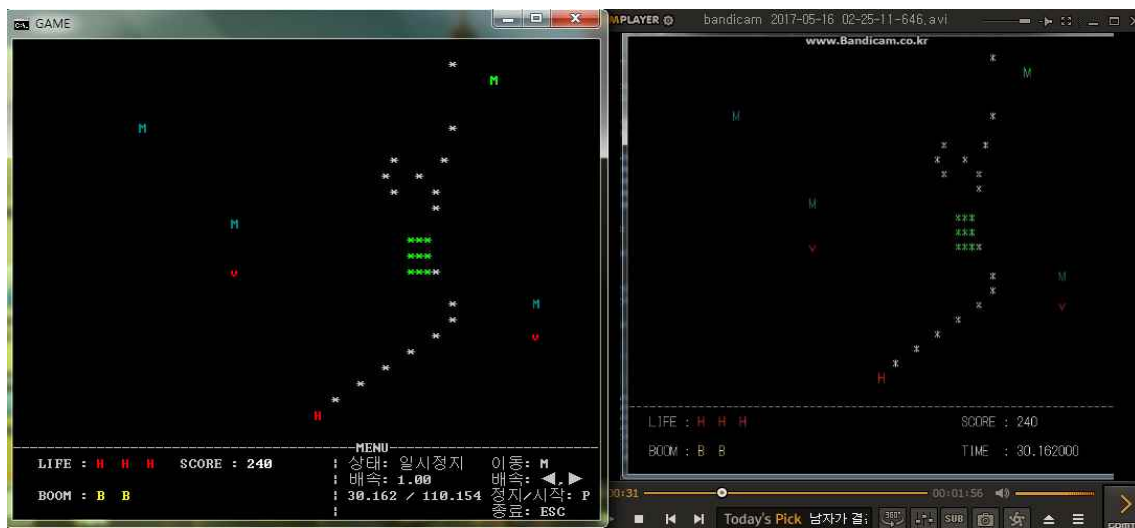


그림20, 그림21을 보시는 것처럼 노트북, 데스크탑에서 같은 시간에서 같은
화면을 보이고 있습니다. 때문에 정확하게 일치한다는 걸 실험 결과로 알았고

이는 정상적으로 작동한다고 판단, 구현을 완료 하였습니다.

최종적으로 앞서 데이터 저장, FPS 문제점을 해결한 내용을 바탕으로 리플레이 기본 기능을 구현하였습니다. 또한 추후에 시간 이동을 위해 인덱스만 이동해야하기 때문에 vector의 반복자를 이용하여 구현하였고 게임 특정 시간에 여러 개의 명령어(다중 키 입력 상황)를 해결하기 위해 vector의 현재 경과 시간 보다 같거나 큰 시간들을 가지는 모든 데이터를 진행하는 식으로 구현하였습니다.

바. 리플레이 배속 기능 구현

기본적인 기능을 구현을 완료하고 처음 개발 범위에서 선정한 기능 중 배속 기능을 구현하기로 결정, 어떻게 배속 기능을 구현해야 할까 고민을 많이 하였습니다. 처음 생각한 방법은 생성 시간, 총알의 발사 시간 등 즉 게임에 쓰이면서 리플레이와 연관된 모든 시간을 (나누기 배속)을 하는 방법 이었습니다. 간단하게 예를 들자면 기본 1배속에서는 60ms 뒤에 총알이 생성됩니다. 만약에 2배속 이라면 30ms 4배속 이라면 15ms 등 이런 식으로 계산하는 방법입니다. 하지만 이 방법에는 아주 큰 문제가 존재했습니다. 게임 자체가 30FPS 기반입니다. 이는 while문 한번에 33ms 만큼 시간이 걸립니다. 즉 60ms을 실제 2프레임에서 작동을 하게 되고 30ms은 실제 1프레임에서 작동됩니다. 또한 4배속으로 했을 때 시간은 15ms이고 이는 실제 1프레임에서 작동되게 됩니다. 때문에 2, 4배속을 했을 때 같은 시간에 생성되는 것처럼 보일 것입니다. 이를 간단하게 표현해보면 60ms에 총알이 생성돼 히어로에게 다가 오고 있습니다. 히어로는 30ms에 왼쪽으로 이동시키면 죽지 않습니다. 즉 2프레임에 총알이 생성, 1프레임일 때 히어로가 이동을 해야 죽지 않습니다. 하지만 4배속으로 올려버리기 되면 총알도 1프레임에 생성, 히어로도 1프레임에 움직이게 됩니다(60/4, 30/4 두 개의 시간이 33ms (한 프레임) 시간 보다 작기 때문에) 때문에 죽지 않아야하는 히어로가 죽는 상황이 발생 할 수 도 있고 이는 실제로 구현 도중 발생한 문제였습니다.

(그림22) 나누기 배속을 하는 방법에 대한 본인이 생각한 문제점.

60ms	30ms	15ms	7.5ms
2 프레임	1프레임	1프레임	1프레임
기본	x2 배속	x4 배속	x8배속

배속의 증가에 따라 같아 지는 프레임

이런 한 문제점을 겪으면서 저는 시간 단위로 배속하는 방법보다는 프레임 단위로 배속을 하는 게 낫겠다고 생각하였습니다. 여러 가지 고민을 하는 도중 생각해본 방법은 (물론 좋은 방법은 아닌 거 같습니다.) 업데이트 횟수를 늘려주는 방법입니다. 업데이트 횟수를 30번에서 60번으로 늘려주면서 updqte 함수 인자로 초기 30FPS 의 한 프레임 시간 즉 33ms를 넘기면 되지 않을까 생각했습니다. 이런 방법을 취하면

초당 30번 업데이트 호출, 인자 값으로 33ms => 업데이트를 5번 호출하는데 $(5 \times 33\text{ms}) = 165\text{ms}$, 인자 값으로 넘어오는 시간의 합 = 165ms

초당 60번 업데이트 호출, 인자 값으로 33ms => 업데이트를 5번 호출하는데 $(5 \times 16\text{ms}) = 80\text{ms}$, 인자 값으로 넘어오는 시간의 합 = 165ms

즉 업데이트 횟수 자체가 많아지기 때문에 실제로 2배속, 4배속 같은 효과를 낼 수 있겠다고 판단, 실제 이 방법을 채택하여 구현해보았습니다.

그 후 실험을 하였는데 실험 방법은 노트북과 데스크탑에서 배속으로 플레이 후 랜덤으로 일시 정지 후 동영상 녹음 파일의 위치를 리플레이 실행 시간과 똑같이 설정 한 후 같은지 확인하였습니다.

(그림23) 데스크탑에서의 왼쪽 실제 리플레이. 오른쪽 녹음 영상(배속)



(그림24) 노트북에서의 왼쪽 실제 리플레이. 오른쪽 녹음 영상(배속)



그림23, 24에서 보시는 것처럼 동일한 결과를 나타내고 있고 이는 문제 없이 잘 돌아간다고 판단하여 구현을 완료하였습니다.

최종적으로 초당 update 횟수 조절과 시간은 33ms(30FPS)을 고정 시키는 방법을 사용하여 2, 4, 8배속을 구현하였습니다.

사. 리플레이 이동 기능 구현

마지막으로 구현한 내용은 리플레이 상에서 시간 이동 이였습니다.

처음 시간 이동을 구현 할 때는 가장 큰 고민거리는 입력 받은 시간으로 이동 하면서 게임 내용도 그 시간대와 일치하는 화면을 보여줘야 한다는 점이였습니다. 간단하게 시간을 입력받고 그 시간대로 이동하면서 그 시간 전에 해야 하는 모든 작업을 실행시키면 된다고 생각했습니다. 하지만 이 생각에는 큰 문제점이 있었습니다. 그 문제는 앞서 설명 드린 것처럼 히어로의 입력키, 게임의 랜덤적인 요소(보스 패턴)등을 제외하고는 게임 로직에 맡기는 방식입니다. 때문에 유저가 입력한 시간과 현재 시간 사이에 몬스터의 생성, 충돌, 움직임 등은 실제로 프레임을 돌려봐야 결과를 알 수 있을 것 이라고 생각했습니다. 또한 현재시간보다 전으로 돌아가는 경우에는 몬스터의 좌표, 생성, 충돌을 역으로 되돌려야 하기 때문에 많은 생각을 했습니다.

이렇게 고민을 하는 도중 “실제로 프레임을 돌린 다음에 결과를 보여주면 되지 않을까?” 생각을 하였고 조금 더 확대하여 계산을 해보았습니다.

일단 유저가 만약에 5300ms(5.3초)을 입력을 했다면

$5300\text{ms}/33\text{ms} = 160$ (한 프레임당 33ms)

$5300\text{ms}\%33\text{ms} = 20\text{ms}$ (안 떨어지는 시간)

즉 현재 시간을 처음으로 (0ms) 초기화 하고 160번의 업데이트와 20ms 만큼 간 뒤에 업데이트를 한 번 더 호출하면 5.3초로 실제 리플레이가 갔을 때와 입력해서 이동했을 때가 똑같은 거라고 생각을 했고 실제 구현, 테스트하였습니다. 테스트 방식은 전과 동일하게 노트북, 데스크탑에서 실제 녹음영상과 시간 이동을 하고 난 뒤에 결과를 보았습니다.

(그림25) M/m 명령어를 눌러 시간 이동 메뉴 표시



(그림26) 데스크탑에서 영상과 시간 이동 후 비교(왼쪽 프로그램, 오른쪽 영상)



(그림27) 노트북에서 영상과 시간 이동 후 비교원쪽 프로그램, 오른쪽 영상)



그림 25, 26, 27에서 보시는 것처럼 정상 작동한다는 걸 확인하였고 이 방법으로 결정, 구현을 완료하였습니다.

최종적으로 입력 받은 시간을 몇 번의 프레임에 돌아가는지 확인 한 후 임의적으로 update 함수를 확인 한 프레임 횟수만큼 실행시킨 뒤 결과 화면을 보여주는 것으로 최종 구현 하였습니다.

[과제를 마치고 느낀 점]

가. 과제를 마치고 느낀 점 및 제안

처음 과제를 받았을 때 게임공학과에 진학한 이상 메모장, 그림판 같은 프로그램을 리플레이 할 경우 보다는 실제 게임을 리플레이 할 경우가 많다고 판단하여 실제 게임을 제작하는 것부터 시작했습니다. (물론 기본적인 구현 베이스를 갖겠지만 실제 게임을 리플레이 하는 것이 더 도움이 될 거 라고 생각했습니다.) 그 다음에 리플레이 기능에 대해서 생각을 해보았습니다.

리플레이 기능은 게임을 만들 때 현재 제작하고 있는 게임이 스타크래프트, 워크래프트, 리그오브레전드등 전략적인 요소가 중요한 게임에서는 자신이 했던 경기를 다시 보면서 전략을 연구하고 시간적으로 타이밍을 계산하거나 하는 면에서 리플레이는 선택이 아닌 필수라고 생각하고 있었습니다. 때문에 처음 과제를 받았을 때는 여기서 끝나는 과제가 아닌 추후에 굉장히 도움이 많이 되고 또 재밌는 과제라는 생각에 흥분을 숨길 수 없었습니다. 하지만 첫 번째의 과제와 같이 제가 선택한 컨테이너가 옳은 선택 이었는지 또 구현에 있어 제가 선택한 방법과 만든 프로그램에 발생 할 수 있는 문제점이 없는지를 고민하고 선택하는데 큰 어려움이 있었습니다. 하지만 이런 저런 고민들을 해결하고 구현하면서 개인적으로 첫 번째 과제는 STL에 대한 기본적인 이해와 실력 향상에 큰 도움이 되었다면 이번 두 번째 과제는 C++ 능력, 전체적인 문제 해결 능력에 있어 큰 도움이 되었습니다.

하지만 이 글을 쓰고 있는 지금 까지도 게임 상의 시간을 직접 조작하는 방법이 옳은 방법인가? 즉 게임의 프레임을 자연적으로 흘러가게 하는 게 아니라 임의적으로 조작하여 배속, 시간 이동을 구현하는 방법이 합당한 방법인가? 에 대한 정확하게 대답을 못할 거 같습니다. 이 부분에 있어서는 굉장히 아쉽고 또 과제가 기간이 지난 뒤에도 좀 더 생각을 해 봐야 할 거 같습니다.

또한 히어로 입력과 랜덤적인 요소를 제외하고 전부 게임 로직에 맡기는 현재 프로그램의 시스템 상 어쩔 수 없고 또 알지만 감당한 문제점이 한 개 존재합니다. 그것은 게임의 패치(업데이트, 게임의 밸런스 패치 예를 들어 총알의 속도) 진행 되었을 때 리플레이 기능이 제대로 작동 하지 않는다는 문제입니다. 이 문제 같은 경우에는 게임의 패치 같은 기능이 없기 때문에 따로 구현하지는 않았지만 조금 생각해봤을 때

1. 버전이 다른 리플레이를 작동하지 못하게 한다.

-> 즉 패치 된 이후에 전 패치 버전의 리플레이를 아예 작동하지 못하게 하는 방법입니다.

2. 몬스터 , 총알, 파워 등등을 저장 해놓고 이 데이터들이 변했을 때 그에 맞게 동기화 작업을 하는 방법을 생각했습니다.

첫 번째 과제는 어떤 상황에서는 어떤 컨테이너를 사용해야하는지? 이것이 성능에 직결되는지 등을 중요하게 생각하고 과제에 임했다면

이번 과제는 실제 구현에 있어 어떻게 구현해야 정상적으로 작동이 되는지? 등에 초점을 맞춰 과제에 임했습니다.

또한 실제 게임에서는 어떤 식으로 리플레이를 구현하고 처리 했는지 굉장히 궁금해졌습니다. 때문에 과제를 낸 후에도 좀 더 공부를 해야겠다! 라고 생각 하였습니다.

결론적으로 이번 과제는 추후에 게임의 리플레이 기능을 구현할 때(물론 저의 방식이 좋지 않는 방식이라는 걸 공부를 통해 알게 된 것만으로도) 큰 도움이 될 것이고 과제를 하는 동안 재밌었습니다.

제안 이라는 항목이 과제에 대해 이런 식에 구현도 과제에 내보면 좋겠다고 생각이 들어 교수님께 제안 하는 항목이라고 생각하여 첫 번째 과제, 두 번째 과제를 하면서 생각난 과제가 있어 감히 제안 란에 글을 쓰기로 하였습니다.

물론 리플레이, 랭킹 시스템에 대한 제안은 아니지만

게임 인벤토리를 관리하는 프로그램을 만들면 어떨까 생각을 해보았습니다.

게임 인벤토리를 관리한다는 말이

- 게임의 현재 유저 데이터 관리
- 게임의 아이템 데이터 관리

- map 이라는 컨테이너의 사용과 이해

등 과 같은 요소들을 이해 할 수 있지 않을까 생각이 들었습니다.

거기다가 인벤토리에 재료 아이템, 장비 아이템, 소모성 아이템을 인벤토리 창을 나누어 (즉 첫 번째 과제에 5개의 아레나 있는 거와 비슷합니다.)

각 아이템에 개수에 따른 정렬

아이템 성능에 대해 정렬

인벤토리 내의 아이템 검색

등을 기본적으로 실행 되어야 하는 필수적인 요소로 넣고 하면 괜찮지 않을까 해서 감히 글을 써보았습니다.

마지막으로 과제를 내주신 교수님께 감사하다는 말을 드리고 싶습니다.

이상으로 마치겠습니다. 두서없는 긴 글 읽어주셔서 감사합니다.