

일반화 된 다중 입력 라인형

리듬 액션 게임 시스템



저자 1 201924464 문예강

저자 2 201424418 김세영

지도교수 조환규

목 차

1. 서론	1
1.1 연구 배경 및 기존 문제점	1
1.2 연구 목표	4
2. 게임 시스템	8
2.1 기본 데이터 구성	8
2.2 SCENE 구성	11
2.2.1 INTRO SCENE	11
2.2.2 MUSIC SELECT SCENE	12
2.2.3 IN GAME SCENE	15
2.2.4 SCORE SCENE	26
2.3 결과물 및 코드 재사용	36
3. 전용 컨트롤러 시스템 배경	41
3.1 기존 컨트롤러의 단점 및 전용 컨트롤러의 필요성	41
3.2 컨트롤러 설계	43
3.3 컨트롤러 확장성 구성	44
3.4 컨트롤러 버튼 구성	49
3.5 컨트롤러 펌웨어	50
3.6 컨트롤러하우징 및 기판 구성	53
4. 연구 결과 분석 및 평가	60
5. 결론 및 향후 연구 방향	61
6. 참고문헌	62

1. 서론

1.1 연구 배경 및 기존 문제점

그림 1-1)은 구글 리듬게임 검색 결과이다. 이미지에서 확인할 수 있듯이, “라인형 리듬 게임”은 리듬 게임의 한 갈래로, ‘비트매니아’ 시리즈를 바탕으로 시작되어 현재 리듬게임 시장의 주류로 자리잡고 있다.

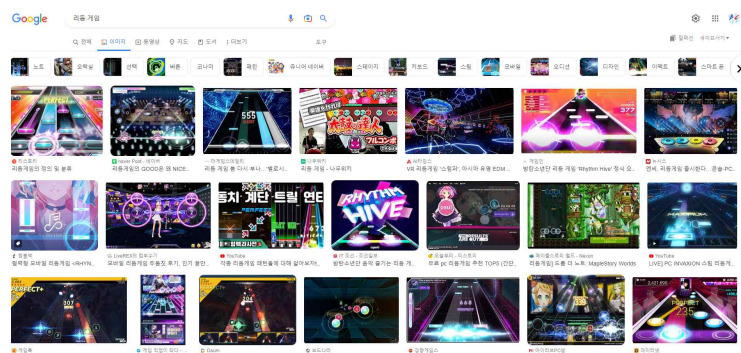


그림 1-1) 구글 리듬게임 검색 결과



그림 1-2) 리듬 게임의 예시(TAPSONIC BOLD)

그림 1-2) 는 고전적인 리듬 게임의 예시 이다. 리듬 게임은 각 라인으로 내려오는 객체(이하 “노트”로 칭함)를 노트가 판정선에 접근하는 타이밍에 맞추어 대응하는 버튼을 입력하는 게임이다. 이렇게 입력하는 것을 “노트를 처리한다” 라고 칭한다.

노트를 처리할 때, 일정 시간 이상 지속적으로 입력을 주어야 하는 경우 “롱 노트”로 칭하며, 일시적으로 입력을 주어야 하는 경우 “단타 노트”로 칭한다. 예를 들어, 롱 노트는 3초 이상 A를 눌러야 하고, 단타 노트는 A를 짧게 눌렀다 떼면 된다. 이러한 방식의, 고전적인 형태의 리듬 게임을 “건반형 리듬게임”이라고 칭한다. 또한, 노트가 내려오는 경로가 직선인 경우 “라인”이라고 칭한다. 이러한 “라인”을 기반으로 하는 리듬게임을 “라인형 리듬게임”이라고 칭한다. 노트들은 요구하는 입력 타이밍과 버튼을 서로 공유하여 동시에 처리해야하는 경우, 우리는 이

를 노트가 “겹친다”라고 칭하도록 한다. 겹친 노트는 여러 개의 버튼을 동시에 입력하여 처리하는 것이 바람직하다.

“DJMAX RESPECT V”, “EZ2ON REBOOT:R”, “Sixtar Gate: STARTRAIL” 등 많은 PC 리듬 게임이 이미 존재하나, 이들은 하나의 버튼에 하나의 노트만 대응한다. 또한, 노트의 폭 역시 일정하다.

“Nostalgia” 시리즈는 하나의 노트에 여러개의 버튼이 대응한다. 하지만 고정된 폭의 노트를 가지며, 두 개 이상의 노트가 겹쳐서 나오는 경우가 없다.

모바일 게임인 “프로젝트 세카이 컬러풀 스테이지”에서는 노트가 고정되지 않은 너비를 가지며, 두 개이상의 노트가 겹쳐서 나온다. 그러나 이들은 둘 이상의 단타 노트가 겹쳐서 나오지 않으며, 또한, 구현상의 한계로 2개 이상의 롱노트가 겹친 경우, 하나의 입력만 유지해도 처리가 된다. 이는 N개의 입력에는 N개의 입력을 주어야 한다는 직관과 어긋난다.

“vivid:stasis”의 경우 하나의 노트가 2개의 버튼에 대응하는 “범퍼 노트”가 존재한다. 이 때문에 2개의 단타 노트가 겹쳐서 등장할 수 있다. 그러나 롱 노트와 단타 노트 겹쳐서 등장하거나, 롱노트와 롱노트가 겹쳐서 등장하지 않는다.

“VOEZ”의 경우, 일반적인 패턴에서는 겹친 노트가 존재하지 않으나, 몇몇 특수 채보에서 겹친 노트가 등장한다. 이 때의 처리는 “프로젝트 세카이 컬러풀 스테이지”와 유사하게, 겹친 롱 노트를 하나의 입력만 유지해도 처리가 된다. 마찬가지로 이는 직관과 다르다.

“DEEMO”의 경우, 노트가 고정되지 않은 너비를 가진다. 그러나 노트가 겹쳐서 등장하지 않으며, 롱노트가 존재하지 않는다.

이외에도 많은 리듬게임이 존재하지만, 앞서 언급한 특징들에서 크게 벗어나지 않는다. 이 졸업 과제는 위와 같은 라인형 리듬 게임들의 핵심 시스템을 일반화하여 구현하는 것을 목표로 한다.

1.2 연구 목표

이 졸업 과제는 위와 같은 라인형 리듬 게임들의 핵심 시스템을 일반화한 시스템을 구현하는 것을 목표로 한다. 이를 위해서는, 앞서 언급한 요소들을 고려해야한다.

첫째, 하나의 노트는 여러개의 버튼에 대응해야한다.

둘째, 하나의 버튼은 여러개의 노트에 대응해야 한다.

셋째, N개의 서로 노트가 겹칠 수 있어야한다.

넷째, 겹치는 노트는 종류(롱 노트, 단타 노트)를 가리지 않아야한다.

다섯째, N개의 겹친 롱 노트를 처리할 때는, N개의 입력이 주어져야한다.

여섯째, 노트의 너비는 고정여야 아니어야한다.

기존 리듬 게임 시스템을 일반화한 이 조건들을 모두 만족시키는 시스템을 제안하고, 그러한 시스템을 구현하는 것을 목표로 한다.

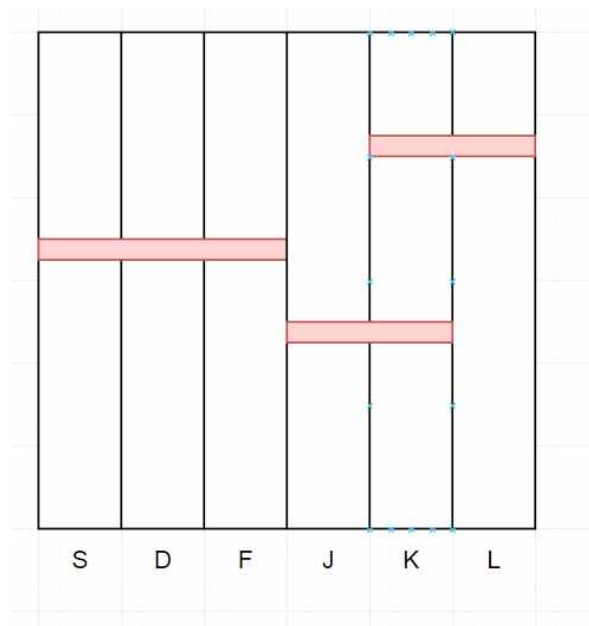


그림 1-4) 리듬 게임의 모식도

그림 1-4)은 그림 1-2)와 비슷하게 각 S,D,F,J,K,L 라인을 따라 노트가 내려오는 모습을 표현한 것이다. 그러나 그림 1-2)와 다르게, 하나의 노트가 여러 개의 버튼에 대응 할 수 있다. 위의 그림에서 세개의 노트가 각각 SDF, JK, KL 라인에 대응한다는 것을 확인할 수 있다. 즉, 이것은 "첫째, 하나의 노트는 여러개의 버튼에 대응해야한다." 라는 조건을 만족시킨다.

또한, 그림 1-4)에서 여러개의 너비의 노트가 등장할 수 있다는 것을 확인할 수 있는데, 이는 "여섯째, 노트의 너비는 고정 너비가 아니어야한다." 조건을 만족한다는 것을 뜻한다.

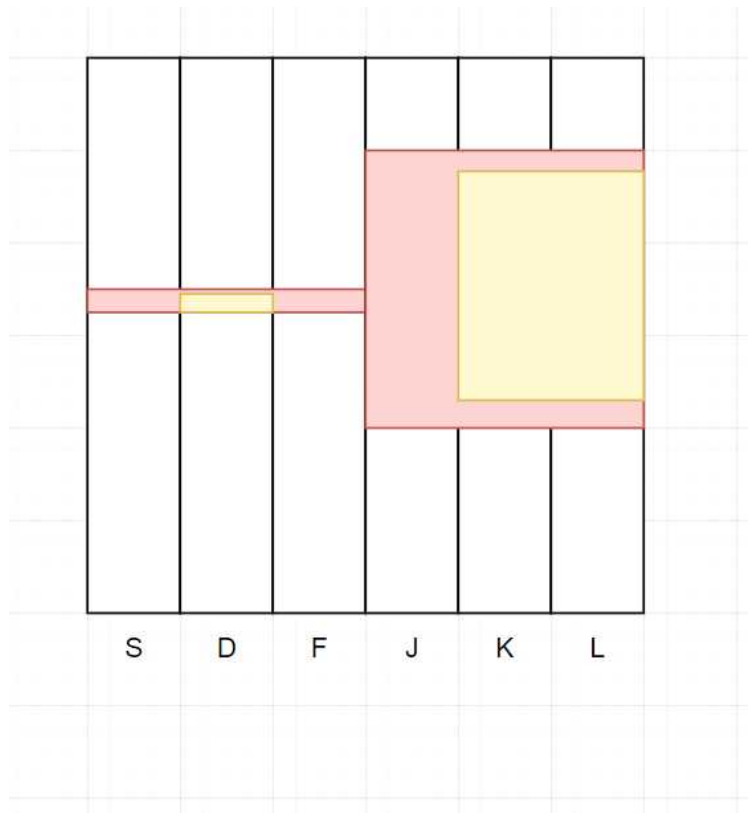


그림 1-5) 노트 겹침의 예

그림 1-5)에서는 왼쪽에 SDF에 걸치는 노트와, D에 걸친 노트가 겹쳐있다는 것을 확인 할 수 있다. 이는 D를 입력했을 때, SDF 노트와 D 노트에 입력이 모두 대응한다는 것을 알 수 있다. 따라서 이 시스템은 “둘째, 하나의 버튼은 여러개의 노트에 대응해야 한다.” 조건을 만족시킨다.

또한 그림 1-5)에서, SDF에 걸친 노트와 D에 걸친 노트가 겹쳐있으므로, 이 둘을 처리하기 위해서는 S와 D, 또는 D 또는 F 이렇게 2개의 입력이 주어져야한다. 즉, 직관과 동일하게 노트 수 만큼의 버튼 입력을 주어야 한다. 이는 시스템이 “다섯째, N개의 겹친 룡 노트를 처리할 때는, N개의 입력이 주어져야한다.” 조건을 만족시킨다는 것을 의미한다.

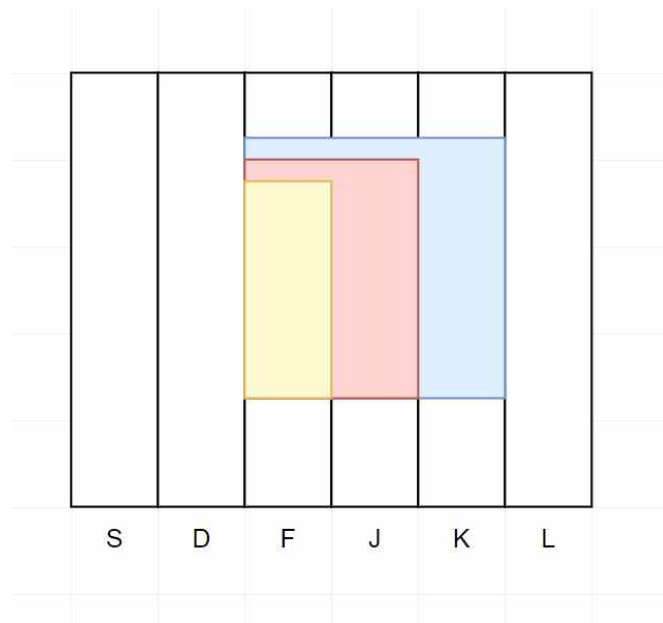


그림 1-6) N개의 겹친 노트의 예시

그림 1-6)에서는 FJK, FJ, F에 대응하는 세 개의 노트가 겹쳐 있음을 확인할 수 있다. 즉, 이는 시스템이 “셋째, N개의 서로 노트가 겹칠 수 있어야한다.” 조건을 만족하는 것이다.

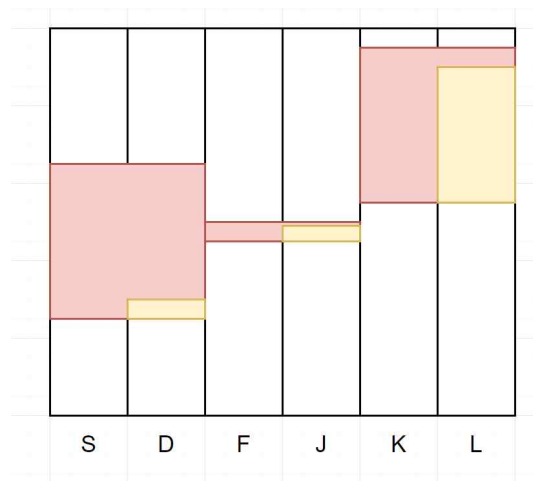


그림 1-7) 노트가 겹치는 세가지 케이스

그림 1-7)에서, SD 라인에는 롱 노트와 단타 노트가 겹치는 것을 확인할 수 있다. FJ 라인에는 단타 노트와 단타 노트가 겹치는 것을 확인할 수 있다. KL 라인에서는 롱노트와 롱노트가 겹치는 것을 확인할 수 있다. 따라서 이 시스템은 “넷째, 겹치는 노트는 종류(롱 노트, 단타 노트)를 가리지 않아야한다.”는 조건을 만족한다.

정리하면, 이 제안한 시스템은 앞서 언급한 리듬게임들을 일반화한 6가지 조건을 만족한다. 따라서 시스템적으로 더 일반화되었다고 볼 수 있다. 이 졸업과제에서는 이 시스템을 사용한 리듬 게임을 직접 개발하여 실제로 상용화 가능성을 보이는 것을 목표로 한다. 위와 같은 조건을 모두 고려하면서 개발하는 것은 어려움이 있기에, 기존 리듬 게임들은 여러 제약을 추가하여 특수

한 시스템을 기반으로 게임을 개발하였다. 이 졸업과제에서는, 기존 게임들의 시스템을 모두 포함할 수 있는 일반화된 범용적인 리듬게임 시스템을 개발하여, 유사 게임 시스템 구현시 필요한 개발력을 최소화시킬 수 있는 게임 시스템을 개발하는 것이 목적이다 더 나아가, 게임 시스템을 포함한 주변기기까지 모두 제작하여 완전한 게임을 완성도 있게 제작하여 프로젝트의 완결성을 보여주는 것을 목표로 한다.

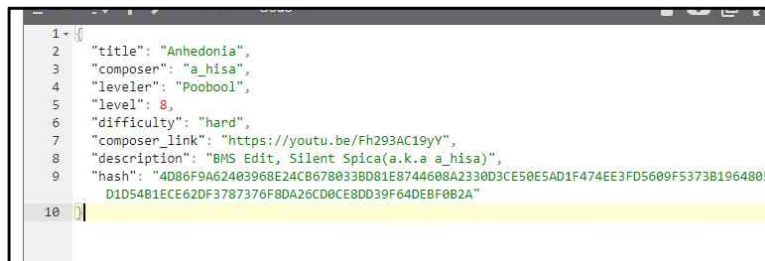
2. 게임 시스템

2.1) 기본 데이터 구성



이름	압축 크기	원본 크기
audio	3,254,240	3,341,147
icon.png	18,866	19,229
meta.json	253	329
Pallete.zip	75	127
preview	137,308	139,263
table.json	19,639	809,320

그림 2-1) pbrff 포맷 구성



```
1 {
2   "title": "Anhedonia",
3   "composer": "a_hisa",
4   "leveler": "Poobool",
5   "level": 8,
6   "difficulty": "hard",
7   "composer_link": "https://youtu.be/Fh293AC19yY",
8   "description": "BMS Edit, Silent Spica(a.k.a a_hisa)",
9   "hash": "4D86F9A62483968E24CB678033BD81E8744608A2330D3CE50E5AD1F474EE3FD5609F5373B1964805D1D5481ECE62DF3787376F8DA26CD0CE8DD39F64DEBF0B2A"
10 }
```

그림 2-2) meta.json 내용

이 프로젝트에서는 그림 2-1)에서 보이는 pbrff라는 자체 포맷을 사용한다. pbrff 포맷의 구성은 아래와 같다.

1. audio: 게임에 사용되는 음원 파일이다. 실제로는 vorbis로 압축된 ogg 포맷이나, xor encoding으로 음악 플레이어로 재생이 불가능하도록 제한한다. 이는 음원 상태로 재배포를 허가하지 않는 경우가 있었기 때문에 저작권 문제 때문에 이러한 조치를 취했다.
2. icon.png: 게임에 사용될 아이콘 이미지 파일이다. png파일로 저장된다. 고해상도 이미지의 경우 아이콘 용량이 과도하게 커지는 문제가 발생했기 때문에, 용량을 줄이기 위해 저장시에 1024*1024의 해상도로 다운스케일링 하도록 했다.
3. Pallete.zip: 게임에 사용되는 히트 사운드를 저장한다. wav파일은 무압축 파일 특성상 용량이 매우 커져 로딩 시간에 악영향을 주어, 각 히트 사운드는 10MB 이내가 되도록 제한했다.
4. preview: 미리듣기에서 사용되는 음원 파일이다. audio 파일과 마찬가지로 xor encoding 한다. 미리 듣기 파일을 따로 저장하는 이유는, 선곡 창에서 곡 전체에 해당하는 audio파일을 로딩하는 경우 시간이 약 5초정도로 로딩시간이 너무 길어졌다. 3분 분량의 음악 대신 10초 분량의 미리듣기 파일을 로딩하게 하여 로딩 시간을 단축했다.
5. table.json: 게임데이터의 전체데이터를 의미한다. 노트의 시각, 라인, 폭 등의 정보와, 게임 이벤트의 시각, 종류등의 정보를 저장한다.
6. meta.json: 게임데이터의 메타데이터를 의미한다. 곡 제목, 작곡가 이름, 레벨, 난이도등 간소화된 정보가 저장되어있으며, 암호화 되어있지 않다. 대략적인 구성은 그림 2-2)에서 확인할 수

있다. 처음에는 table.json에 meta.json을 포함하여 하나의 파일로 다루었는데, table.json의 길이가 너무 길어 선곡창에서 로딩 시간이 너무 길어지는 문제가 발생했다. 때문에 meta.json이라는 별도의 메타 데이터 파일로 분리하고, 선곡 화면에서는 이것만 로딩하는 방식으로 수정했다.

이 pbrff파일을 로딩하면 MusicEntry가 된다. 게임 데이터를 표현하는 객체인데, MusicEntry는 두가지로 나뉜다.

1. FullEntry: 실제 게임을 실행하기 위한 전체의 데이터 홀더이다. pbrff파일에서 audio, table.json을 포함한 모든 정보를 로딩한다
2. PreviewEntry: 선곡 화면에서 미리보기를 제공하기 위한 데이터 홀더이다. audio, table.json을 제외하고, 미리듣기를 위해 필요한 데이터만 로딩한다.

이렇게 MusicEntry를 두가지로 분리한 이유는, 선곡 화면에서의 로딩 속도를 줄이기 위해서다. 그림 2-1)에서 확인할 수 있듯이, audio파일과 table.json은 그 크기가 나머지 파일들을 합한것의 거의 30배에 달한다. 또한, table.json의 경우 매우 긴 json 파일 이기 때문에, 파싱하는데 상당한 시간이 필요했다. 이들을 직접 로딩하지 않고 preview, meta.json이라는 간략화된 메타데이터로 변경하여 파일을 로딩하는데 드는 시간을 획기적으로 줄일 수 있었다. 이것을 간단히 추상화 한 것이 FullEntry와 PreviewEntry이다.

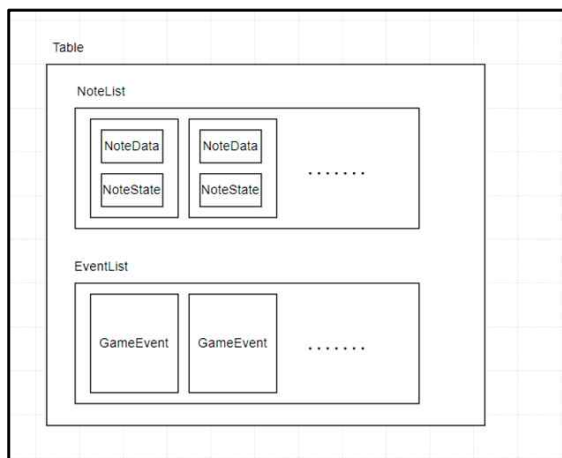


그림 2-3) Table 클래스 구조

게임에 사용되는 로드된 레벨 데이터의 기본 구성은 그림 2-3)과 같다.

각 노트들의 판정 시간, 길이 등 Load 후 변하지 않는 고정된 값을 가지는 NoteData, 그리고 노트 판정 모듈에 의해 변경되는 NoteState가 존재한다. 이 두가지를 묶어서 Note를 구성한다. GameEvent는 변속, 배경 효과 등의 노트와 무관한 시간에 따라 발생하는 사건들을 저장하는 데이터다. 따라서 기본적으로 Load후 변경되지 않는 고정된 값이다. 이 Note와 GameEvent들은 리스트를 이루며, NoteList와 EventList를 묶어 Table 클래스를 구성한다. Table 클래스는 이 두 리스트를 관리하는 기능을 수행한다. 이 클래스는 게임 데이터를 저장할 때, 그리고 게임 상태로 불러올 때 등, 기본적인 데이터 구조로 사용하기로 한다.

실제로 table.json에 저장될 때는 NoteList의 NoteData필드와 EventList만 저장된다. 이유는 NoteState는 게임 중 변경되는 상태를 표시하기 위한 객체이기 때문에, 저장해야할 데이터를 가지고 있지 않다. 이러한 불필요한 데이터가 저장될 필요는 없으므로, 이를 압축하기 위해 저장에 필요한 부분과 필요하지 않은 부분을 명백히 분리하기 위해 이러한 구조를 선택했다.

2.2 Scene 구성

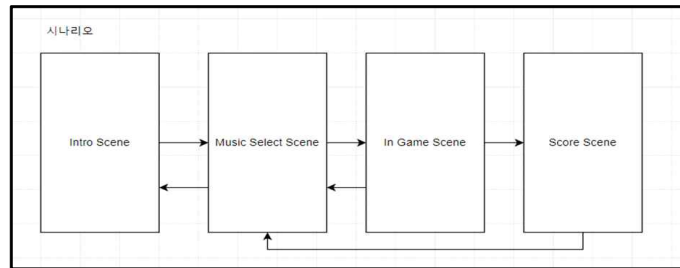


그림 2-4) Scene 구성도

그림 2-4)에서 볼 수 있듯이, 시스템은 크게 네가지 Scene으로 구성된다.

- Intro Scene: 빌드를 실행하면 처음 진입하는 화면으로, 사용자는 게임 설정을 조작할 수 있다.
- Music Select Scene: 플레이를 시작하기 전에 보여주는 화면으로, 사용자는 곡을 선택하고 In Game Scene으로 진입할 수 있다.
- In Game Scene: 실제 게임 플레이가 이루어지는 화면으로, 이 프로젝트의 핵심적인 부분이다.
- Score Scene: 게임 결과를 보여주고, Music Select Scene으로 되돌아가기 위한 화면이다.

2.2절에서는 이 Scene들의 실제 구현된 화면과, 구현을 위해 개발한 시스템에 대해 설명하도록 한다. 그리고 프로젝트의 핵심이 되는 In Game Scene 위주로 자세히 설명하도록 한다.

2.2.1) Intro scene



그림 2-5, 2-6) 구성된 인트로 화면의 UI

게임 첫 실행시 진입하는 화면으로, 시작시 실행 되어야하는 스크립트 실행 및 다른 Scene으로 진입하는 진입점 기능을 한다. 이 Scene의 UI의 기능은 아래와 같다.

1. 다른 Scene으로 이동
2. 게임 세팅

2.2.2) Music Select Scene

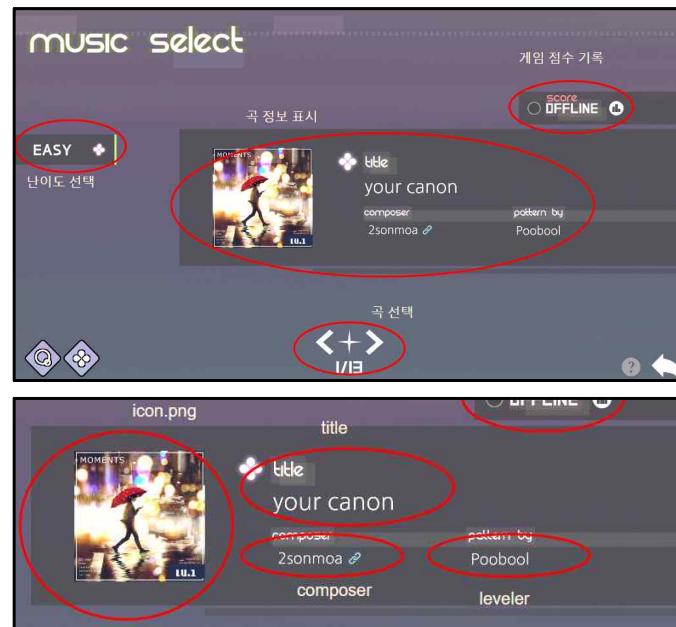


그림 2-7,2-8) 곡 정보 UI

그림 2-7)는 게임을 시작하기 전, UI를 통해 선곡을 하는 Scene의 모습을 보여주고 있다. 유저는 하단의 버튼을 마우스로 클릭하거나, 방향키 조작 등으로 곡을 선택할 수 있다. 그림 2-8)에서는 Pbrff의 데이터를 어떻게 표시하는지 보여주고 있다. 이는 meta.json에 title,composer,leveler등 있는 곡 정보에 대해 보여준다

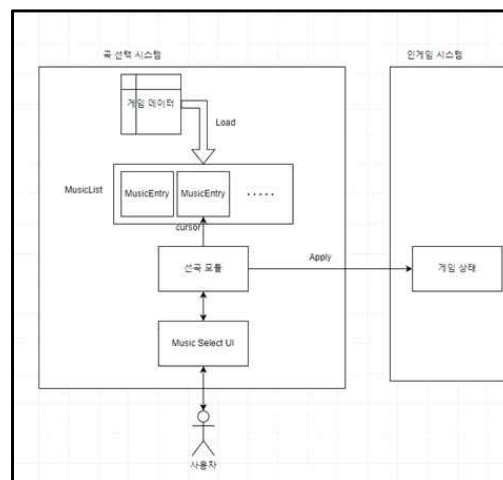


그림 2-9) Score Select Scene System

Music Select Scene에서 사용되는 시스템에 대한 개요다.

게임 데이터로부터 MusicEntry를 생성한다. (단, 이때 MusicEntry는 PreviewEntry만을 로딩하여 리소스를 절약한다.) 이 MusicEntry를 불러오고, 유저는 UI를 통해 MusicEntry 하나를 선택한다. 그리고 사용자가 해당 곡으로 게임 시작을 요청하면 해당 MusicEntry에서 실제 FullEntry를 추출하여 실제 게임에 로딩한다.

```

01:     public static void ApplyOnGame(PbrffExtractor.FullEntry data)
02:     {
03:         UnityEngine.Debug.Log("ApplyOnGame");
04:         NoteEditor.isLoaded = true;
05:         pbrffdata = data;
06:         mechanim = GameMech.Create(mechanim_type);
07:         modi.Modify(table);
08:         Game.pbrffdata.icon = data.icon;
09:         var pcm = data.audio.GetPCM();
10:         if (reverse) pcm.Reverse();
11:         MusicPlayer.SetMusic(pcm);
12:         NoteEditor.RefreshBPMS();
13:         NoteEditor.now_open_url = data.url;
...

```

코드 2-1) ApplyOnGame 메소드 일부

코드 2-1)은 선택한 음악의 PbrffExtractor.FullEntry를 게임 시작 전 로딩하는 기능을 한다. pbrffdata = data;에서 인수로 받은 FullEntry를 바인딩한다. 그리고 이를 기반으로 게임 시작 전 시스템을 초기화한다.

data.GetPCM()은 audio파일로부터 실제 음원의 PCM Sample을 추출하여 로딩하는 메서드이다. 여기서 몇가지 문제가 발생했는데, 처음엔 NAudio와 NVorbis를 이용해 Strategy Pattern으로 mp3, wav, ogg 세가지 포맷에서 PCM Sample을 추출하도록 구현했다. 이는 Thread safe하여 로딩을 병렬로 처리할 수 있어 장점이 있었다. 그러나 프로젝트를 중간에 유니티 버전을 2020에서 2022로 업그레이드를 하는 과정에서 라이브러리 호환 문제가 발생했고, NVorbis를 제거해야만했다.

이를 대신하기 위해 FMOD를 사용했다. FMOD는 기존에 사용하던 엔진이기 사용하기 용이하고, 적은 수의 라이브러리를 사용하는 것이 의존성 측면에서 좋다고 판단했기 때문이다. FMOD Sound파일을 생성하여 음원을 추출하는 방식으로 변경했는데, 단점은 FMOD가 멀티스레드를 지원하지 않기 때문에 비동기 로딩을 통해 최적화 할 수 있었던 부분을 동기화 하면서 시간상 손해를 봐야했다는 것이다. 때문에 로딩 시간이 상당히 느려지는 trade off가 있었지만, 이를 통해 NVIDIA Reflex를 지원할 수 있게 되었고, 기존 Strategy Pattern을 사용하던 코드를 제거하고, NAudio 및 NVorbis 없이 Pure FMOD로만 작동하게 되어 코드가 단순해졌다.

2.2.3) In Game Scene

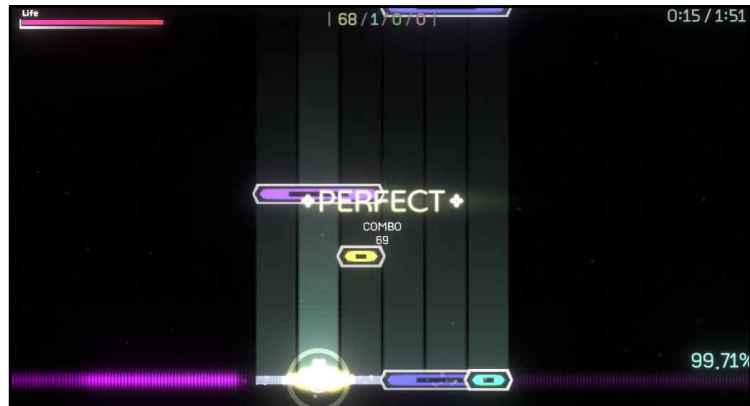


그림 2-10) 인게임 화면

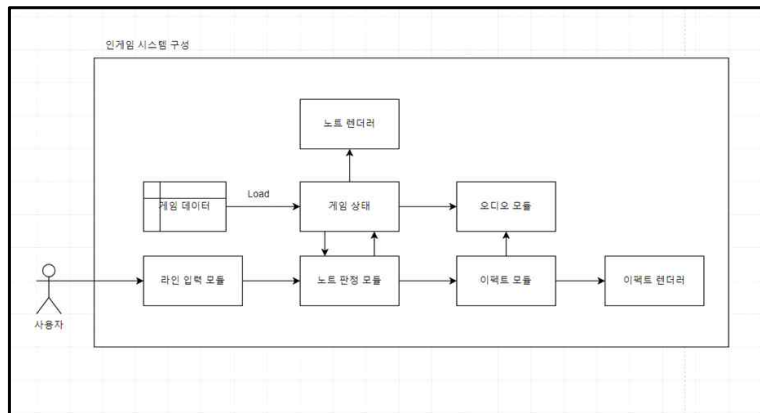


그림 2-11) 인게임 시스템 구성도

그림 2-10)은 실제로 게임이 동작하는 플레이 스크린샷이다. 그리고 그림 2-11)은 작동하는 게임의 시스템을 나타낸다. 사용자가 준 입력을 입력 모듈이 받고, 이것을 이용해 노트 판정 모듈은 게임 상태를 갱신하고 이펙트를 발생시킨다. 노트 렌더러는 현재 노트 상태를 기반으로 노트를 렌더링한다. 오디오 모듈은 게임 데이터에서 로딩된 PCM 샘플을 이용해 음악을 재생한다. 그림 2-11)은 이 과정을 간단하게 나타낸 것이다

1. 라인 입력 모듈

사용자의 키보드 입력을 추상화하여, 0~N번의 버튼 입력으로 추상화한다. 이 시스템에서는 하나의 버튼이 하나의 라인에 대응하므로, “라인 입력”이라는 용어를 사용했다.

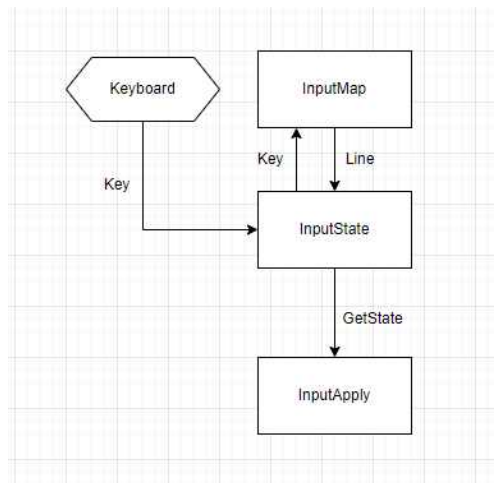


그림 2-12) 입력 모듈 모식도

그림 2-12)는 입력모듈의 구조이다. 키보드로부터 입력을 받아 InputState가 현재 모든 키 상태를 추적하고, 이를 InputMap로 라인 입력 상태로 변환한다. 이렇게 얻은 각 라인 입력 상태를 InputApply에 전달하면, 이 상태를 기반으로 InputApply는 노트 판정을 수행할 수 있다.

InputMap은 특정 키를 N번 라인의 입력으로 매핑해주는 기능을한다. 즉, 키 A는 1번, 키 S는 2번 라인에 대응한다 같은 정보를 관리하면서, A는 1로, S는 2로 변환해주는 일종의 함수 역할을 한다. 처음에는 이처럼 InputMap을 사용하지 않아 정해진 키로만 게임 플레이가 가능했으나, InputMap을 사용하여 키 입력을 추상화 하여, 이를 통해 사용자가 게임을 위한 키 세팅을 변경할 수 있게 되었다.

기본 unity 입력은 Update를 기반으로 키 입력을 확인하기 때문에, 1/fps초 이하의 정확도를 가질수 없다는 문제점이 있었다. 또한, 프레임 의존적이기 때문에 스테터링 등에 취약하다는 문제가 있었다. 따라서 이를 해결하기 위해 별도의 스레드를 통해 입력을 받도록 수정했다. 이 기능을 구현하는 AsyncInput 클래스를 작성하였는데, Unity 기본 입력을 사용할 수 없으므로 P/Invoke를 통해 User32.GetAsyncKeyState를 호출하는 방식으로 수정하였다.

AsyncInput에 Virtual Key를 등록하면, AsyncInput이 입력 스레드를 통해 계속 GetAsyncKeyState를 호출하면서 키 입력을 확인한다. 키 입력이 들어오면 Key가 Down인지 Up인지를 폴링한다. 이때 입력이 들어오는 경우, 해당 키의 입력 상태를 ConcurrentStack에 Push한다. 그 후 메인스레드에서 Unity의 Update가 호출되면, 이 ConcurrentStack에서 키 입력들을 받아와 한 프레임동안 누적된 입력들을 한꺼번에 처리한다. 이러한 방식으로 입력 스레드와 게임 스레드를 분리하였다. 이를 통해 보다 약 2ms정도의 정밀도를 갖는 입력 모듈을 만들 수 있었다. 이것을 래핑한 클래스가 그림 2-12)의 InputState다.

2. 노트 판정 모듈

현재 게임의 State와 라인 입력을 이용하여 State를 갱신하고 이펙트 모듈로 이펙트를 발생시킨다. 판정시, 앞서 언급했던 다중 입력을 처리할 수 있도록 판정 시스템을 구현한다.


```

1: while (input.NextState())
2: {
3:     CastBegin(notes, input);
4:     CastOn(notes, input);
5:     Complete(notes, input);
6: }
7: }

```

코드 2-2) 노트 판정 알고리즘

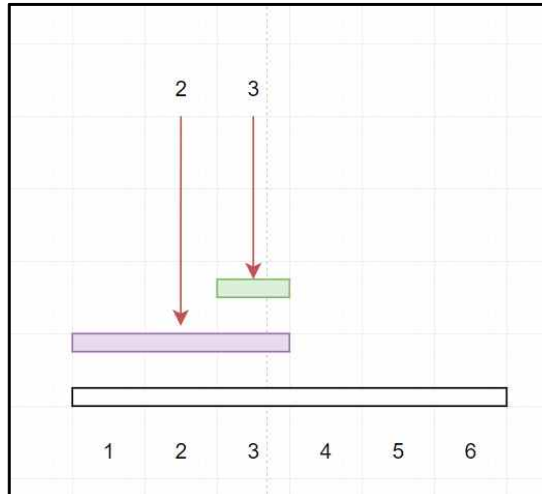


그림 2-13) Input Casting

코드 2-2)은 이 프로젝트의 핵심인 노트 판정 로직을 보여주고 있다. 크게 Begin-On-Complete 세단계를 거친다.

그림 2-13)은 1~6라인 노트, 1~3라인 노트, 3라인 노트 세가지가 겹쳐진 예시이다. 기본적인 아이디어는, 노트들을 순서대로 쌓아둔 후, 가장 위에 있는 노트가 처리 되는 것이 가장 직관적이라는 아이디어에서 시작했다. 예를들어 3번 라인 입력이 시작되면, 3번의 가장 위에 있는 노트가 먼저 처리되는 것이 맞을 것이다. 만약에 다른 노트가 처리 된다면, 나머지 1,2,4,5,6번 라인을 입력해도 초록색 노트는 처리할 수 없게 된다. 하지만 3번 노트가 처리된다면, 여전히 1,2,4,5,6라인을 이용하여 보라색 노트나 흰색 노트를 처리할 수 있다. 이러한 원리를 캐스팅(Casting)이라고 하도록 하자.

또한, 앞서 조건으로 단타 노트와 롱 노트 등 타입을 구분하지 않고 겹칠 수 있어야 한다고 조건을 걸었다. 이는 단타 노트를 길이가 0인 롱노트로 취급하여 생각하여 해결하기로 했다. 그렇게 되면 롱노트 끼리 겹친 경우 처리가 되면 자연스럽게 단타 노트도 구분하지 않고 정상적으로 처리가 될 것이라 판단했기 때문이다.

판정은 3단계로 이루어진다. 매 프레임마다, 각 노트는 Begin, On, Complete 세개의 판정을 진행한다. 그리고 노트의 상태 역시 세가지 단계를 갖는다.

- Begin 단계: 아직 Begin 판정이 일어나지 않았을 때
- On 단계: Begin 판정이 일어났으며, Complete 판정이 일어나지 않았을 때
- Complete 단계: Complete 판정이 일어났을 때

Begin 판정은 Begin 단계인 노트를, On 판정은 On 단계인 노트를 처리할때 일어난다.
단, Copmlete 판정은 Begin, On 단계의 노트에 일어난다.

이렇게 3단계로 구분한 근거는 다음과 같다. 입력은 기본적으로 키를 누른다-누른 상태로 유지한다-키를 뺀다 3단계로 이루어진다. 따라서 각각에 대응하는 Begin, On, Complete 세가지 판정으로 구분한 것이다. 그리고 현재 노트가 어떤 판정 중인지, 어떤 판정이 필요한지를 판정 상태로 표현한 것이다. 즉, Begin은 키를 눌렀을 때, On은 누르고 있을 때, Copmlete은 뺐을 때에 대응한다.

이제 다시 코드 2-2)에 대해 살펴보자.

1. CastBegin은 해당하는 라인을 그림과 같이 캐스팅하여 가장 먼저 닿는 Begin 단계의 노트를 선택하여 Begin 판정을 한다. 위 그림에서, 세개의 노트가 모두 Begin 단계라면, 초록색, 보라색 노트는 Begin 판정이 된다.

2. CastOn은 CastBegin과 마찬가지로 캐스팅하여 On 단계의 노트들을 선택하여 On 판정을 한다. 마찬가지로 그림에서 세 노트가 모두 On 단계라면, 초록색, 보라색 노트가 On 판정이 일어난다.

3. Complete은 세가지 경우 판정을 수행한다.

- 현재 On 단계인 노트들 중 롱노트가 완전히 처리되었을 때 (즉, 처리된 길이>노트 길이 일 때)
- On 단계의 노트가 판정 범위를 벗어났을 때
- Begin 단계의 노트가 판정범위를 벗어났을 때

첫 번째는 키 입력이 끝나서 판정이 완료되는 경우를 처리하기 위함이고,

두 번째, 세 번째는 판정이 불가능해져 Miss처리 되는 경우를 처리하기 위함이다.

정리하면 키를 눌렀을 때 판정을 시작해야하는 노트는 CastBegin으로 처리하고, 키를 계속 누르고 있어야하는 노트는 CastOn으로 골라내어 처리한다. 그리고 판정을 완료해야하는 노트는 처리가 끝나거나 불가능한 노트에 Complete 판정을 주어 처리한다.

```
01: public int OnCmp(Note x, Note y)
02:     {
03:
04:         if (isInclude(x, y))
05:         {
06:             return -1;
07:         }
08:         if (isInclude(y, x))
09:         {
10:             return 1;
11:         }
12:         if (Cmp(x.data.y, y.data.y) !=0)
13:         {
14:             return Cmp(x.data.y, y.data.y);
15:         }
16:         if (Cmp(x.data.x, y.data.x) !=0)
17:         {
18:             return Cmp(x.data.x, y.data.x);
19:         }
20:
21:
22:         if (Cmp(x.data.time, y.data.time) !=0)
23:         {
24:             return -Cmp(x.data.time, y.data.time);
25:         }
26:         if (Cmp(x.data.length, y.data.length) !=0)
27:         {
28:             return Cmp(x.data.length, y.data.length);
29:         }
30:
31:         return Cmp(x.data.dx, y.data.dx);
32:     }
```

코드 2-3) On 비교함수

```

01: public int BeginCmp(Note x, Note y)
02: {
03:     if (Cmp(x.data.time, y.data.time) !=0)
04:     {
05:         return Cmp(x.data.time, y.data.time);
06:     }
07:
08:
09:     if (isInclude(x, y))
10:     {
11:         return -1;
12:     }
13:     if (isInclude(y, x))
14:     {
15:         return 1;
16:     }
17:
18:     if (Cmp(x.data.y, y.data.y) !=0)
19:     {
20:         return Cmp(x.data.y, y.data.y);
21:     }
22:
23:     if (Cmp(x.data.x, y.data.x) !=0)
24:     {
25:         return Cmp(x.data.x, y.data.x);
26:     }
27:
28:
29:     if (Cmp(x.data.length, y.data.length) !=0)
30:     {
31:         return Cmp(x.data.length, y.data.length);
32:     }
33:
34:     return Cmp(x.data.dx, y.data.dx);
35: }

```

코드 2-4) Begin의 비교함수

그런데 CastBegin, CastOn에서 생략한 부분이 있는데, 세 개의 노트가 겹쳐져있고 그 위로 캐스팅을 수행하는데, 세 개의 노트를 어떤 순서로 쌓아야할지는 논의하지 않았다.

이 비교함수를 정확히 정의하는 것이 조금 까다로웠지만, 1~6번 라인 순서로 캐스팅을 수행한다는 점을 감안하여 비교함수를 코드 2-3) 2-4)와 같이 정의하였고, 정상적으로 작동하였다. 여기서, data.x는 노트 너비, data.y는 노트가 걸쳐있는 가장 왼쪽 라인을 의미한다. data.time은 노트가 시작되는 시각을 뜻하고, data.length는 룡 노트의 길이를 뜻한다. isInclude 함수는 위치상 노트 하나가 완전히 다른 하나에 포함이, 아니면 일부만 겹치는지를 의미한다. 추가로 data.dx는 더미 데이터로, 항상 같다.

간단히 비교하자면, On의 비교함수는 Begin의 비교함수와 달리 노트의 시간이 우선순위가 낮

다. BeginCmp은 시간 / 포함관계 / 왼쪽우선 / 너비 / 길이 순서이고, OnCmp은 포함관계 / 왼쪽우선 / 너비 / 시간 / 길이 순서이다. On의 롱노트를 유지하는 판정이기 때문에, 언제 시작했는지가 판정에 큰 영향을 주지 못한다. 반면 비교함수가 동일한데, 비교함수를 이렇게 정의한 근거는 다음과 같다.

- 한 노트가 다른 하나를 포함하는 경우, 안쪽에 있는 노트가 먼저 처리되어야한다.
- 한 노트가 다른 하나를 포함하지 못하는 경우, 어떤 노트는 항상 다른 노트의 완전한 왼쪽에 위치한다. 캐스팅은 1번부터 6번 라인 순서로 이루어지므로, 왼쪽에 있는 노트가 기회가 더 적다. 따라서 왼쪽이 먼저 처리되어야 한다.
- Begin에서는 입력 타이밍이 더 빠른 노트가 판정되어야한다. 이는 리듬 게임의 문법이다.

이러한 근거로 pivot을 결정했고, 그 pivot을 근거로 비교함수를 정하였다

3. 노트 렌더러

현재 게임 상태를 읽어 그 상태에 맞게 노트를 화면에 표시한다.

```
01: notes.Sort(Cmp);
02: for (int i =0; i < amount; i++)
03:     objs[i].SetNote(null);
04: for (int i =0; i < notes.Count; i++)
05: {
06:
07:     Note note = notes[i];
08:     var obj = objs[i];
09:
10:     if (note.isInScreen())
11:     {
12:         obj.sortingOrder = amount - i;
13:         obj.SetNote(note);
14:     }
15:     else
16:     {
17:         obj.SetNote(null);
18:     }
19:
20: };
```

코드 2-5) 노트 렌더러 코드

코드 2-5)는 현재 화면 안에 보이는 노트들을 필터링하여, SetNote로 해당 노트를 렌더링해주고, sorting order를 지정해주는 Update 함수다. 이때, 일정 시간 범위 내에 있는 노트들을 1차적으로 걸러내고, 다시 거기에서 코헨 서더랜드 알고리즘을 응용하여 화면 밖에 있는 노트는 렌더링하지 않도록 했다. 이 두가지를 통해 NoteRenderer의 성능을 대폭 개선했다.

```

01: public bool isInScreen()
02: {
03:     // 코헨 서더랜드 알고리즘 변형
04:     Vector3 pos1 = GamePreprocessor.mainCam.WorldToViewportPoint(Position());
05:     Vector3 pos2 = GamePreprocessor.mainCam.WorldToViewportPoint(Position(data.length));
06:     int code1 =
07:         (pos1.x <0 ? 1 : 0) |
08:         (pos1.x >1 ? 2 : 0) |
09:         (pos1.y <0 ? 4 : 0) |
10:         (pos1.y >1 ? 8 : 0);
11:     int code2 =
12:         (pos2.x <0 ? 1 : 0) |
13:         (pos2.x >1 ? 2 : 0) |
14:         (pos2.y <0 ? 4 : 0) |
15:         (pos2.y >1 ? 8 : 0);
16:     return code1 ==0 || code2 ==0 || (code1 & code2) ==0;
17: }

```

코드 2-6) 코헨 서더랜드 알고리즘을 변형한 isInScreen함수

4. 오디오 모듈

요청되는 특정 소리를 재생한다. (배경음, 노트 키움)

처음에는 오디오 엔진으로 Unity 내장 오디오인 AudioSource를 사용했다. 그러나 AudioSource에 사용하는 AudioClip을 생성할 때 오버헤드가 크다는 점, 메인 스레드 외에 로딩 스레드에서는 사용이 불가능하다는 점, Scene 전환시 관리가 어렵다는 점, DSP버퍼 크기 변경이 불가능해 오디오 레이턴시가 너무 큰 점(약 90ms) 등의 문제가 있었다. 따라서 한단계 저수준 API인 FMOD를 사용하여 이 문제를 모두 해결하였다. FMODComponent, FMODWrapper, MusicPlayer, KeySoundPallette등 어댑터를 구현하여 사용하여 이를 해결했다.

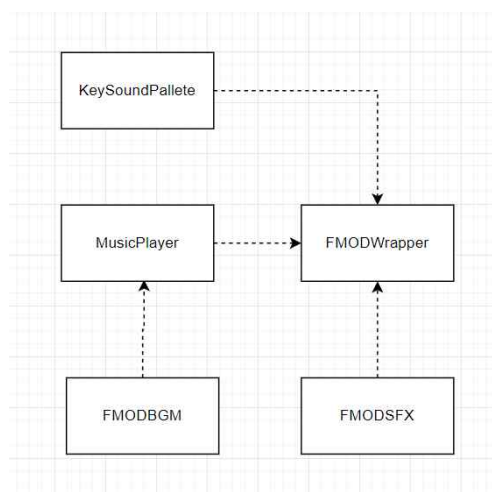


그림 2-14) 오디오 모듈

이렇게 구현한 어댑터들의 관계도는 그림 2-14)와 같다.

FMODBGM은 배경 음악을, FMODSFX는 효과음을, KeySoundPallete는 키음을 재생한다. FMODWrapper는 여기에 쓰이는 ChannelGroup과 System을 관리하는 역할을 한다. FMOD Sound 생성 및 관리, FMOD Sound 재생 등 FMOD API를 한단계 추상화 해준다. MusicPlayer는 FMODWrapper를 이용해 현재 재생되는 배경음을 조작할 수 있다. 현재 재생되는 음악의 channel과 sound를 관리하면서, Resume Pause등의 조작을 위한 인터페이스를 제공한다. KeySoundPallete는 FMODWrapper를 이용해 현재 로딩된 키음을 관리하고, 이를 재생 및 조작할 수 있는 인터페이스를 제공한다. FMODBGM과 FMODSFX는 조금 다른데, 정적 클래스인 KeySoundPallete와 달리 유니티 MonoBehaviour로, Scene에 컴포넌트를 추가하면 편리하게 해당 소리를 재생할 수 있는 유니티 컴포넌트다.

5. 이펙트 모듈, 이펙트 렌더러

판정에 따라 다른 모듈에 적절하게 이펙트를 만들고, Particle System을 이용해서 FX를 렌더링 한다.

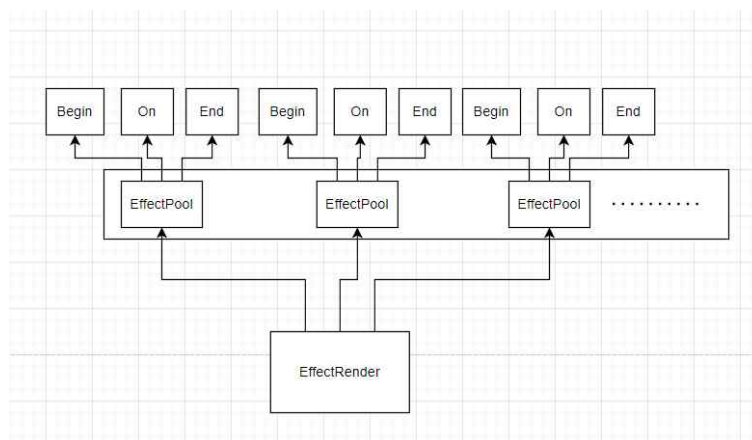


그림 2-15) 이펙트 렌더러

이펙트 모듈의 구조는 그림 2-15)와 같다. EffectRender의 정적 메소드인 makeNoteEffect를 호출하면 해당 판정에 맞는, 발생시켜야 하는 이펙트를 계산하여 해당하는 EffectPool을 선택하여 적절한 이펙트를 재생한다.

EffectPool는 Perfect, Good, Ok, Miss등 하나의 지정된 이펙트를 방출(Emit)하는 오브젝트 풀 객체이다. EffectPool은 Begin On End 세가지 단계에 해당하는 이펙트 모두 가지고 있어, 노트 처리 모듈에서 Begin On Complete 처리가 발생했을때, 각 단계에 대해 발생하는 이펙트를 요청하면 이를 풀링을 통해 재생해주는 역할을 한다.

예를들어, 노트 판정 모듈에 의해 Begin판정에 의해 Perfect처리가 되었다고 가정하자. 그러면 EffectRender는 Perfect 이펙트가 저장된 EffectPool을 가져온 후, EffectPool.Effect(BEGIN)을 호출하여 EffectPool이 가진 begin 이펙트를 재생하도록 시킨다.

이러한 방식을 취한 이유는 이펙트를 유연하게 변경할 수 있도록 하기 위해서이다. 만일 이런 방식을 취하지 않는다면, 모든 종류의 이펙트를 Begin, On, End 이펙트를 모두 따로 관리해주면서 코드를 수정해주어야 하는데, 이 방식에서는 정의된 EffectPool 객체에 해당하는 프리팹만 변경하면 되기 때문에 훨씬 간단하게 수정할 수 있다.

6. 게임 데이터, 게임상태

게임의 레벨 데이터가 기록된 파일로부터 읽어들이, 게임 상태에 사용할 수 있도록 불러온 데이터와 그 상태 뜻한다. Table, MusicPlayer, 그 외에 여러가지 세팅값들이 이에 해당한다.

```
01:
02: public class Table
03: {
04:     const int recent_version =5;
05:     public int version = recent_version;
06:     [JsonProperty]
07:     [SerializeField]
08:     NotesWindow NoteList =new NotesWindow();
09:     [JsonProperty]
10:     [SerializeField]
11:     List<EventData> EventList =new List<EventData>();
...

```

코드 2-7) Table 클래스

대표적으로 코드 2-17)의 Table 객체의 경우 불러온 일련의 노트와 이벤트들의 리스트를 가지고 있다,

이들은 Pbrff파일에서 로딩한 일련의 NoteData, EventData들을 가지고 있다.

Note의 경우 이렇게 불러온 NoteData 뿐 아니라, 노트의 처리 상태를 표현하기 위한 NoteState 필드를 같이 가지고있다. (게임의 진행은 이 NoteState를 수정하는 방식으로 이루어진다.) data필드는 Load된 게임 데이터 영역으로 게임 실행 도중 변경되지 않고, state 필드는 다른 모듈에 의해 변경되고 갱신될 수 있다.

2.2.4) Score Scene

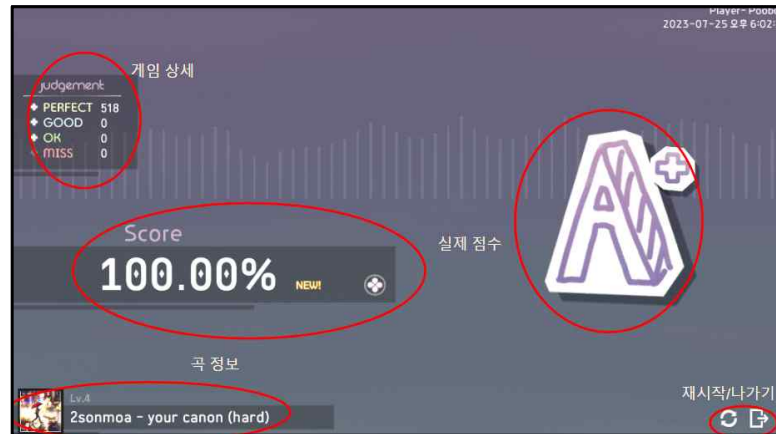


그림 2-16) Score Scene UI

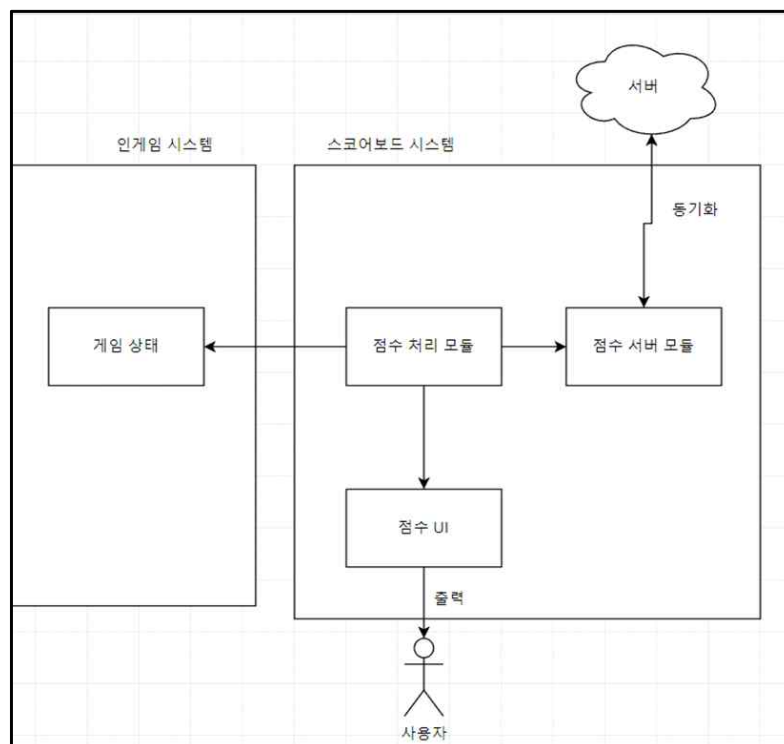


그림 2-17) Score Scene System 모식도

그림 2-17)의 스코어보드 시스템은 Score Scene을 위해 사용되는 시스템의 모식도다. 그 결과를 보여주는 UI는 그림 2-16)에 나타나 있다. Score Scene은 게임 중 노트 판정 모듈에 의해 판정된 결과들을 집계하여 게임 결과를 보여준다.

또한 앞서 바인딩된 MusicEntry로부터 곡 정보를 읽어 그림 2-16)의 좌측 하단에 표시해 준다. 인게임 플레이가 끝나고 결과를 보여주고 서버에 점수를 기록한다. 앞서 인게임 시스템에서의 게임상태를 받아 사용자에게 결과를 UI로 보여주고, 점수 서버 모듈을 이용해 동기화한다.

서버 구성

유저 간 기록을 비교하기 위한 리더보드와 이를 구현하기 위한 비동기 서버를 구현하였다. 클라이언트 단일 서버에 요청할 수 있는 정보는 아래와 같다.

1. Post Score: 현재 플레이 결과를 서버 리더보드에 업로드한다.
2. Get Leaderboard: 특정한 리더보드의 정보를 서버로부터 가져온다
3. Get User Records: 특정 유저의 최고 플레이 기록을 가져올 수 있다.

데이터베이스 스키마

```
Database changed
MariaDB [poobool]> show tables;
+-----+
| Tables_in_poobool |
+-----+
| Blacklist          |
| Checkhash          |
| Leaderboard        |
| Login              |
| fcap               |
+-----+
5 rows in set (0.00 sec)
```

그림 2-18) 데이터베이스를 구성하는 테이블

그림 2-18)은 서버의 DB를 구성하는 테이블을 보여준다. 각각에 대해서 후술하도록 한다.

```
MariaDB [poobool]> desc Leaderboard;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| userid | char(20) | YES | | NULL | |
| songcode | char(20) | YES | | NULL | |
| perfect | int(11) | YES | | NULL | |
| good | int(11) | YES | | NULL | |
| miss | int(11) | YES | | NULL | |
| maxcombo | int(11) | YES | | NULL | |
| score | int(11) | YES | | NULL | |
| date | timestamp | NO | | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
| life | float | NO | | NULL | |
| ok | int(11) | NO | | NULL | |
| modi | int(11) | NO | | NULL | |
+-----+-----+-----+-----+-----+-----+
11 rows in set (0.01 sec)

MariaDB [poobool]> █
```

그림 2-19) Leaderboard 테이블

그림 2-19)의 Leaderboard 테이블은 각 게임의 결과중 가장 우수한 것을 저장하고 집계하는 테이블이다.

- userid: 사용자의 Steam Id로 유저를 식별한다.
- songcode: 유저가 플레이한 레벨을 식별하는 code 값이다.
- perfect, good, miss, maxcombo, score, life, ok : 게임 결과를 의미하는 속성이다.
- date: 게임이 플레이 된 날짜를 의미한다.
- modi: 게임모드를 의미하는 속성값이다.

```
MariaDB [poobool]> desc Checkhash;
```

Field	Type	Null	Key	Default	Extra
userid	char(20)	NO	PRI	NULL	
songcode	char(20)	NO	PRI	NULL	
hash	char(128)	YES		NULL	

3 rows in set (0.07 sec)

그림 2-20) Leaderboard 테이블

그림 2-20)의 Leaderboard 테이블은 레벨 데이터의 변조 여부를 확인하기 위해 올바른 해시값을 저장하는 테이블이다

- userid: 사용자의 Steam Id로, 해시 코드를 업로드한 유저를 식별한다.
- songcode: 유저가 플레이한 레벨을 식별하는 code 값이다.
- hash: 서버에 미리 등록된 정상적인 레벨 데이터의 해시값이다. 이 값이 (userid,songcode)에 해당하는 파일의 해시값과 일치하지 않는 경우, 그 값은 위조된 것으로 간주할 수 있다.

모든 유저는 userid가 자신의 Steam Id에 해당하는 경우, 해시의 삽입, 제거, 갱신이 가능하다.

만일 A가 songcode가 0인 레벨의 해시를 등록하면, 튜플 (A의 Steam id,0, 해시값)이 등록될 것이다. 이 경우 A는 이를 수정할 수 있지만, 제 3자인 B는 이 값을 수정할 수 없다. B가 songcode가 0인 레벨의 해시를 등록하려고 시도하면, (B의 Steam Id, 0, 해시 값)라는 새로운 튜플이 생성된다. 그러나 A의 튜플을 수정할 수는 없다.

따라서 만약 레벨 데이터의 제작자가 누구인지 주어진다면, 그 레벨 데이터가 올바른 데이터인지 검증할 수 있다. 만약 (A의 Steam Id, 0, 해시 값1),(B의 Steam Id, 0, 해시 값2)가 테이블에 등록되어 있다고 가정하자. 만약 파일 데이터의 제작자가 A라면 해시 값 1이 올바른 해시값이 될 것이고, B라면 해시 값 2가 올바른 해시값이 될 것이다.

정리하면, (userid,songcode)를 키로 사용하는 이유는 사칭을 방지하기 위해서다. 업로드하는 유저의 Steam Id는 Steamworks Authentication을 통해 알아내므로, Steam Id를 위조하는것은 불가능하다. 따라서 각 튜플은 Steam Id가 userid인 유저가 등록한 해시 값임을 보장할 수 있고, 이를 통해 레벨에 대한 무결성을 검증할 수 있는 것이다.

```
MariaDB [poobool]> desc Blacklist;
```

Field	Type	Null	Key	Default	Extra
userid	char(20)	YES		NULL	

1 row in set (0.00 sec)

```
MariaDB [poobool]> █
```

그림 2-21) Blacklist 테이블

그림 2-21)의 Blacklist테이블은 사용 제한(Ban)된유저의 id를 저장하는 테이블이다.

- userid: 사용자의 Steam Id로, 유저를 식별한다. 핵, 어뷰징 등 부적절한 플레이를 하는 유저의 사용을 제한하기 위해 사용된다.

```

MariaDB [poobool]> desc fcap;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| userid | char(20) | NO | PRI | NULL | |
| songcode | char(20) | NO | PRI | NULL | |
| modi | int(11) | NO | PRI | NULL | |
| fcap | int(11) | YES | | NULL | |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)

MariaDB [poobool]>

```

그림 2-22) fcap 테이블

그림 2-22)의 fcap 테이블은 Leaderboard와 유사하나 fcap라는 결과값을 저장한다.

- userid: 사용자의 Steam Id로 유저를 식별한다.
- songcode: 유저가 플레이한 레벨을 식별하는 code 값이다.
- modi: 게임모드를 의미하는 속성값이다.
- fcap: Fail, Clear, Full Combo, Good Play, All Perfect로 구분되는 게임의 달성도를 나타낸다.

Leaderboard와 fcap 테이블을 분리하여 기록하는 이유는 두 가지의 최고 기록 기준이 다르기 때문이다. Leaderboard의 경우, score가 가장 높은 게임의 결과를 저장한다. 반면 fcap는 달성도가 가장 높은 게임의 결과를 저장한다. 따라서, 두 가지 테이블은 서로 별개의 게임을 저장하기 때문에, 이를 논리적으로 구분해주는 것이 더 직관적이다.

```

MariaDB [poobool]> desc Login;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| userid | char(20) | NO | PRI | NULL | |
| name | text | YES | | NULL | |
| count | int(11) | YES | | NULL | |
| recent | timestamp | NO | | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

MariaDB [poobool]>

```

그림 2-23) Login 테이블

그림 2-23)의 Login 테이블은 로그인 횟수를 집계하기 위한 테이블이다.

- userid: 사용자의 Steam Id로 유저를 식별한다.
- name: 사용자의 이름을 의미한다.
- count: 로그인 횟수를 의미한다.
- recent: 가장 최근 접속일을 의미한다.

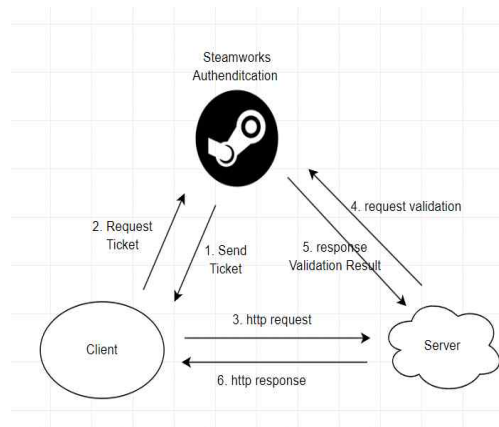


그림 2-24) 서버-클라이언트 모식도

서버 단은 node.js+typescript+MariaDB로 개발했다. 서버의 구조는 그림 2-24)와 같다. 이때, 부인 방지를 위해 Steamworks Authentication을 사용한다. Steamworks에 Ticket을 요청하고, Post Score를 수행하기 위해서는 이 Ticket을 통해 유저가 자신임을 증명해야만 Server가 게임 결과를 DB에 기록한다. 이는 핵 유저를 감지하기 위한 수단이다. 그 외에 Get Leaderboard와 Get User Records는 Write 명령어가 아니므로, 별도의 Authentication을 실행하지 않는다.

- Post Score

1. 클라이언트는 Steamworks Authentication Ticket을 발급받는다.

인증을 위한 API는 Facepunch의 Steamworks 라이브러리를 사용했다. 코드 2-8은 이 과정에 해당하는 실제 코드이다.

```

1: public async void PostScore(ScoreEntry score)
2: {
3:     isDone =false;
4:     var ticket =await SteamUser.GetAuthSessionTicketAsync();
5:     StartCoroutine(UploadRecord(score, ticket));
6: }
  
```

코드 2-8) 클라이언트 Ticket 발급 코드

2. 서버로 Post Score를 요청한다.

이때, Ticket과 게임 결과를 함께 전송한다. 이에 해당하는 코드는 2-9)와 같다.

```

01: IEnumerator UploadRecord(ScoreEntry score, AuthTicket ticket)
02:     {
03:         Debug.Log("Ticket Upload...");
04:         var data = ticket.Data;
05:         Dictionary<string, string> dic = GetRecordDictionary(score);
06:         dic.Add("token", BitConverter.ToString(data).Replace("-", ""));
07:         dic.Add("client_id", "" + SteamClient.SteamId);
08:         dic.Add("name", SteamClient.Name);
09:         dic.Add("music", Game.table.title);
10:         dic.Add("composer", Game.table.composer);
11:         dic.Add("difficulty", Game.table.difficulty);
12:         dic.Add("hash", Game.pbrffdata.hash);
13:         dic.Add("ownerId", "" + Game.content_entry.ownerID);
14:         dic.Add("modi", "" + (int)Game.modi.code);
15:         Debug.Log(Game.modi.code);
16:         UnityWebRequest www
= UnityWebRequest.Post(server_url + "/api/auth/record", dic);
17:         yield return www.SendWebRequest();
18:         Debug.Log(www.result);
19:         if (www.isNetworkError || www.isHttpError)
20:         {
21:             Debug.Log(www.error);
22:         }
23:         else
24:         {
25:             Debug.Log("Send Ticket Done");
26:         }
27:         isDone = true;
28:         GetUserRecords(SteamClient.SteamId);
29:     }

```

코드 2-9) Post Score의 http request

3. 클라이언트 인증을 진행한다.

서버가 http request를 수신하면, 미들웨어는 request에 포함된 티켓, 앱 Id, 앱 인증 키를 Steam에 전송한다. 이후 Steam으로 부터 반환된 티켓의 Steam Id와 클라이언트의 Steam Id가 일치하는지를 확인한다. 신원이 일치하는 경우엔 다음 미들웨어로 넘겨주며, 일치하지 않는 경우 401 오류를 Response한다. 코드 2-10)은 이 과정을 실행한다.

```

01: const authMiddleware = async (ctx: any, next: any) =>{
02:   const token = ctx.request.body.token;
03:   logger.info("Authentication...");
04:   try{
05:     const response = await axios({
06:       method: "get",
07:       url: "https://partner.steam-api.com/ISteamUserAuth/AuthenticateUserTicket/v1/",
08:       params: {
09:         key: "8B7D5FD05DAB6A30F53A832EA4C7AFCB",
10:         appid: 1735670,
11:         ticket: token,
12:       },
13:       responseType: "text",
14:     });
15:     logger.info(
16:       "Auth Valid?" +
17:       (response.data.response.params.steamid == ctx.request.body.client_id)
18:     );
19:     ctx.request.body.userid = response.data.response.params.steamid;
20:     await next();
21:   } catch(e) {
22:     console.log(e);
23:     ctx.status = 401; // 401 Unauth
24:   }
25: };

```

코드 2-10) 서버에서 Ticket의 SteamId를 확인하여 검증하는 코드

3. DB에 점수를 기록한다.(코드 2-11, 12행~17행)

이때, 플레이한 레벨 데이터의 해시값을 서버에 기록된 이때 해시 값을 확인한다. (코드 2-11, 4~7행) 서버에 미리 등록된 레벨 데이터의 해시값과 다른 경우, 레벨 데이터가 변조된 것으로 간주한다. 변조된 레벨 데이터를 플레이 한 경우, 핵 유저로 판단하여 점수를 기록하지 않는다. 또한, 이러한 리더보드 등록이 제한된 유저의 경우 역시 점수를 기록하지 않는다 (코드 2-11, 8~11행)

```

01: async upsert(new_record: ScoreRequest) {
02:   await this.getConnection(async (connection) =>{
03:     const { userid, songcode, ownerId, hash } = new_record;
04:     if(!(await this.hashCheck(songcode, ownerId, hash))) {
05:       logger.info("hash check failed");
06:       return;
07:     }
08:     if(!(await this.banCheck(new_record.userid))) {
09:       logger.info("Banned user:" + new_record.userid);
10:       return;
11:     }
12:     await connection.beginTransaction();
13:     logger.info("Post Record");
14:     await this.FcapUpsert(connection, new_record);
15:     await this.leaderboardUpsert(connection, new_record);
16:
17:     await connection.commit();
18:   });
19: }
20:

```

코드 2-11) Post된 기록을 DB에 저장하는 코드

- Get Leaderboard

특정 레벨 데이터에 대한 모든 유저의 기록을 서버에 요청한다.

1. 클라이언트에서 서버로 특정 songcode에 대한 Get request를 전송한다.
즉, 서버에 `$/api/leaderboard?songcode={songcode}`에 대해 Get Request를 한다.
2. 서버는 코드 2-12)의 sql 쿼리를 수행하고 결과값을 http response로 반환한다.

```
SELECT *FROM Leaderboard lb
WHERE lb.songcode = ?
AND NOT EXISTS (select *from Blacklist bl
where lb.userid = bl.userid)
```

코드 2-12) Get Leaderboard SQL

Rank	Player	Accuracy	Medal	Max Combo	Perfect	Good	OK	Miss
2nd	MrTouchNEO	98.69%	Gold	487	996	32	3	6
3rd	Fluffybunny	98.19%	Silver	522	959	71	4	3
4th	DMusic	97.72%	Gold	300	966	48	15	8
5th	cici1973	96.66%	Gold	245	925	91	6	14
6th	엑파네버	96.05%	Gold	435	890	122	14	11
7th	Poobool	95.68%	Gold	532	878	114	32	9
8th	天燕子	93.30%	Gold	203	896	77	15	48

그림 2-25 리더보드 UI

그림 2-25)은 Get Leaderboard로 서버에서 받은 데이터를 이용해 순위표를 표시하는 것을 구현한 것이다.

리더보드를 구성하는 과정에서 약 1000개의 항목을 로딩하는 경우, 성능 문제로 프리징이 발생하였다. 이를 해결하기 위해 현재 스크롤 뷰의 뷰포트에 보이는 구간의 엔트리만 렌더링하도록 수정하였다. 이때, 오브젝트 풀링을 사용하였다. 모든 해상도에서 엔트리가 12개 이상 렌더링 되는 경우는 없었으므로, 12개의 미리 생성된 오브젝트 중, 나머지 연산으로 사용할 오브젝트를 결정하여 사용하는 방식을 택했다. 코드 2-13은 이 오브젝트 풀링을 수행하는 코드를 나타낸다.


```

01: public void Render(List<ScoreEntry>scores)
02:     {
03:         //범위 내에 있으면 렌더링
04:         foreach(var i in pool)
05:         {
06:             i.SetActive(false);
07:         }
08:         for(int i=0; i<scores.Count; i++)
09:         {
10:             if(IsVisible(i))
11:             {
12:                 /*i번째 위치에 리스트의 i번째 원소를 표시*/
13:                 var obj = pool[i % 12];
14:                 obj.GetComponent<LeaderBoardEntry>().SetEntry(i+1, scores[i]);
15:                 obj.transform.localPosition = Vector3.down **52;
16:                 obj.gameObject.SetActive(true);
17:             }
18:         }
19:     }

```

코드 2-13) 오브젝트 풀링 코드

- Get User Records

특정 유저의 모든 기록을 서버에 요청한다.

1. 클라이언트에서 서버로 특정 songcode에 대한 Get request를 전송한다.

즉, \$"/api/userRecords?userid={userid}"을 요청한다.

2. 서버는 코드 2-14)의 쿼리를 수행하고 결과값을 http response로 반환한다.

```
select * from Leaderboard natural join fcap where userid =?;
```

코드 2-14) Get User Records SQL

클라이언트는 이 Get User Records 요청으로 받은 데이터를 이용해 그림 2-26)과 같이 각 곡에 대한 플레이어의 최대 기록을 표시할 수 있다.



그림 2-26) 플레이어 기록 UI

Get User Records 요청을 추가한 이유는 Get Leaderboard 요청으로는 한계가 있기 때문이다. 만일 GetLeaderboard를 사용하는 경우, 특정 유저에 대한 정보만 가져올 수 없어 모든 songcode에 대해 실행을 해야 하며, 이는 Leaderboard 테이블 전체를 요청하는 것과 같아 매우 비효율적이다. 따라서 특정 유저의 기록만 가져오는 Get User Records 요청을 추가했다.

2.3) 결과물 및 코드 재사용

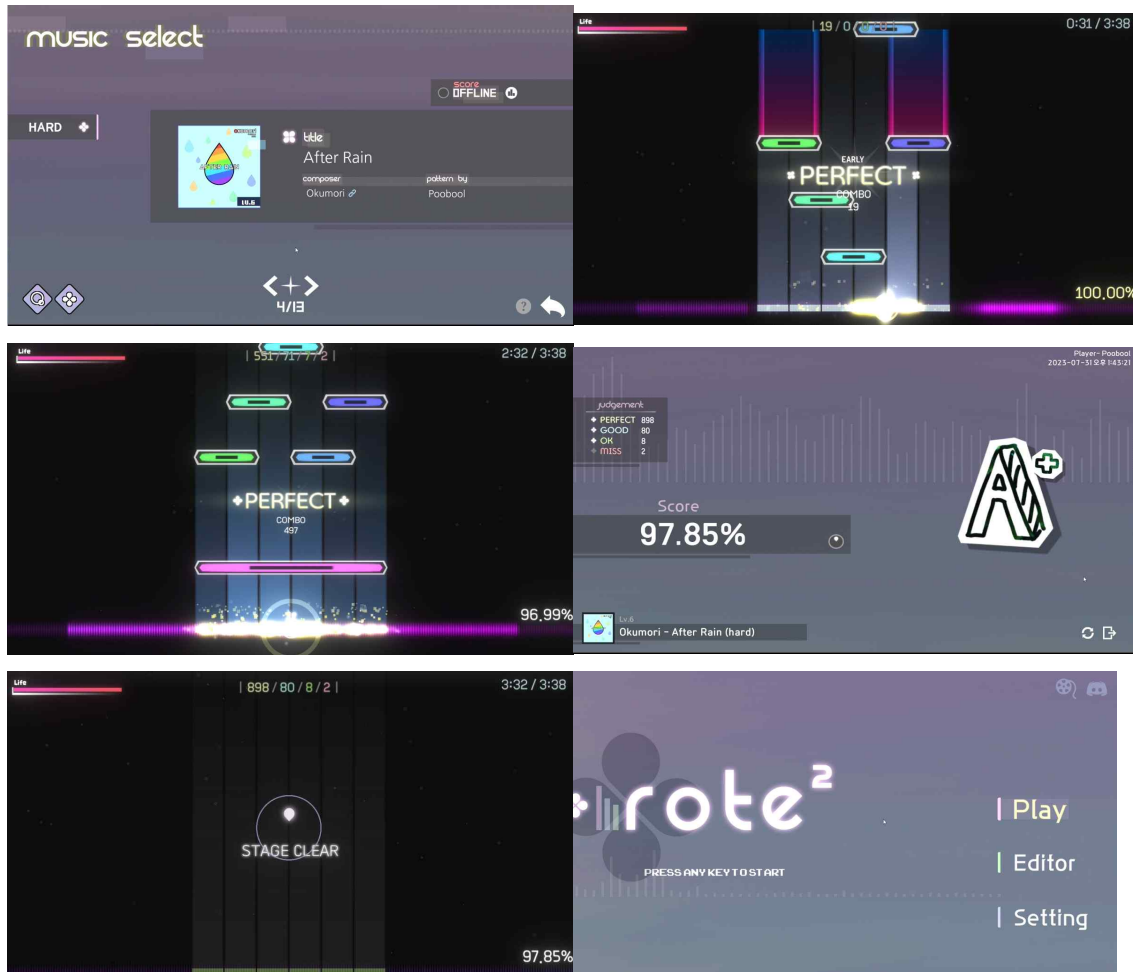


그림 2-27) 게임 실행 화면

그림 2-27은 실제 게임의 실행 화면이다. 실제 시연 영상은 아래의 링크를 통해 확인할 수 있다

https://drive.google.com/file/d/1vuwl4lMUfX6pBzmBBH2QGUmKT4P_FxF/view?usp=sharing

앞에서 구현한 시스템은 상용 리듬 게임들의 일반화 된 형태라고 언급했다. 때문에 이 코드베이스를 이용하면, 코드를 재사용하여 다른 리듬 게임을 제작하는 것이 가능하다. 이것을 가능하게 하는 구조는 NoteType이다. NoteType은 게임 메커니즘에 대한 Strategy이다. 추상 클래스인 NoteType을 상속하는 것으로 구체적인 NoteType을 정의하고, 이를 통해 게임을 간단히 구현할 수 있다.

```

01:  abstract public bool Condition(GameState state, Note data, int input_line); // 판정 조건을 정의
02:
03:
04:  abstract public Vector3 Position(GameState state, Note note, float offset);
05:  abstract public Vector3 Scale(GameState state, Note note);
06:  abstract public Quaternion Rotation(GameState state, Note note);
07:
08:  abstract public Color NoteColor(GameState state, Note note);
09:  abstract public Color LongNoteColor(GameState state, Note note, float t);
10:
11:
12:  abstract public LinePath.Point[] LongNotePositions(GameState state, Note note);

```

코드 2-15) NoteType 추상 클래스의 추상 메서드

NoteType은 게임 메커니즘을 위한 다양한 메서드를 정의한다. 코드)는 NoteType에서 선언하는 메서드 중 일부이다. 이 절에서는 해당 메서드들을 다르게 정의하여 실제로 메커니즘이 다른 별개의 게임을 제작할 수 있음을 보일 것이다.

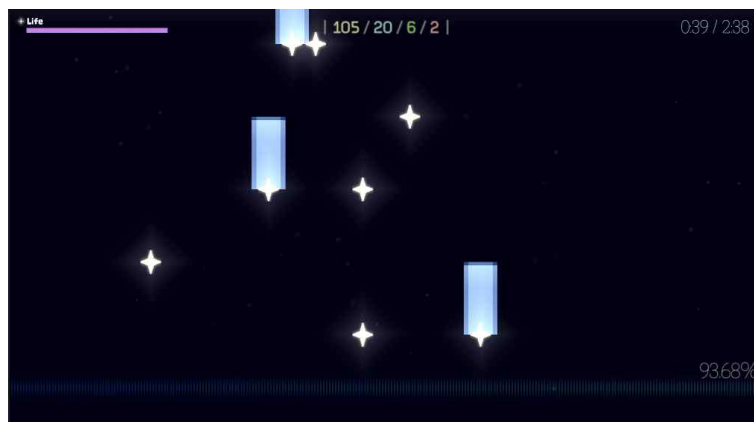


그림 2-28) 변경된 게임 화면

구현하려는 게임은 그림 2-28)과 같다.

1. 노트의 크기가 일정하다.
2. 라인 구분이 없다.
3. 노트가 판정선에 닿는 타이밍에 맞추어 누르면 처리된다.
4. 노트 N개가 동시에 닿는 경우 동시에 여러 키를 눌러야 한다
5. 롱노트가 존재한다.

```

1:  public override bool Condition(GameState state, Note note, int input_line)
2:  {
3:      return true;
4:  }

```

코드 2-16) Condition 메서드

코드 2-16의 Condition 메서드는 시간 조건을 제외한 노트 판정 조건을 의미한다. 예를 들어 1 키에 1라인이 대응하는 기존의 라인형 리듬게임은 `input_line == note.data.y` 같은 방식으로 구현할 수 있다. 이처럼 주어지는 키 입력의 종류를 `input_line`로 판별할 수 있으나, 여기서는 라인 구분이 없으므로 항상 `true`를 반환하면 된다.

```

1: public override Vector3 Position(GameState state, Note note, float offset)
2: {
3:     float distance = GetDistance(note.data.time + offset) * note.data.vx;
4:     var path = NotePath(state, note);
5:     var point = path.Get(distance);
6:     if (point.t >= path.maxTime) return new Vector3(990429, 990429);
7:     else return point.ToVector3();
8: }

```

코드 2-17) Position 메서드

코드 2-17의 Position 메서드는 노트의 위치를 반환하는 함수이다. LinePath라는 구조체를 사용하여 노트가 내려오는 경로를 추상화한다. 주어진 시각에 적절한 지점을 LinePath에서 얻어내어 반환한다.

```

01: private LinePath NotePath(GameState state, Note note) // time = 노트 시간일때 기준으로 거리
02: {
03:     float offset = (note.data.y / Game.lineCount -0.5F)
04:         *2 * Camera.main.orthographicSize * Camera.main.aspect;
05:     LinePath path =new LinePath(
06:         new LinePath.Point[]
07:         { new LinePath.Point(0, 0 + offset, 0)
08:           , new LinePath.Point(1, 0 + offset, 10) });
09:     return path;
10: }
11:

```

코드 2-18) NotePath 메서드

코드 2-18의 NotePath는 LinePath를 반환하는 함수로, 여기서는 (t,x,y)가 (0,0+offset,0), (1,0+offset,10)인 LinePath를 사용한다. 이는 판정선에서 시작하여 위쪽으로 멀어지는 수직방향의 경로를 의미한다. offset은 노트의 수평방향 위치로, `note.data.y`에 의해 표현된다.

```

1: public override LinePath.Point[] LongNotePositions(GameState state, Note note)
2: {
3:     var headTime = GetDistance(note.data.time) * note.data.vx;
4:     var tailTime = GetDistance(note.data.time + note.data.length) * note.data.vx;
5:     return NotePath(state, note).Slice(headTime, tailTime).GetPoints();
6: }

```

코드 2-19) LongNotePositions 메서드

코드 2-19의 LongNotePositions는 롱노트의 시작점부터 끝점까지를 잇는 경로의 모든 점을 반환한다. 이는 LinePath의 일부를 잘라 그 구간을 이루는 점들을 반환하는 Slice 메서드를 이용한다.

```

1: public override Vector3 Scale(GameState state, Note note)
2: {
3:     return Vector3.one / 2F;
4: }

```

코드 2-20) Scale 메서드

코드 2-20의 Scale은 노트 시작 부분의 크기를 정의한다.
여기서 노트의 크기는 일정하므로 고정된 상수를 반환한다.

```

1: public override Quaternion Rotation(GameState state, Note note)
2: {
3:     return Quaternion.identity;
4: }

```

코드 2-21) Rotation 메서드

코드 2-21의 Rotation 메서드는 노트의 회전 상태를 정의한다. Scale과 마찬가지로 고정된 상수를 반환한다.

```

1: public override Color NoteColor(GameState state, Note note)
2: {
3:     return Color.white;
4: }

```

코드 2-22) NoteColor 메서드

```

1: public override Color LongNoteColor(GameState state, Note note, float t)
2: {
3:     return Color.white;
4: }

```

코드 2-23) LongNoteColor

코드 2-22의 NoteColor 메서드는 노트의 머리 부분의 색을, 코드 2-23의 LongNoteColor 메서드는 꼬리부분의 색을 정의한다. 판정중인 노트의 색을 바꾼다거나, 위치에 따라 색을 다르게 할 수 있다. 편의상 여기선 흰색으로 고정했다.

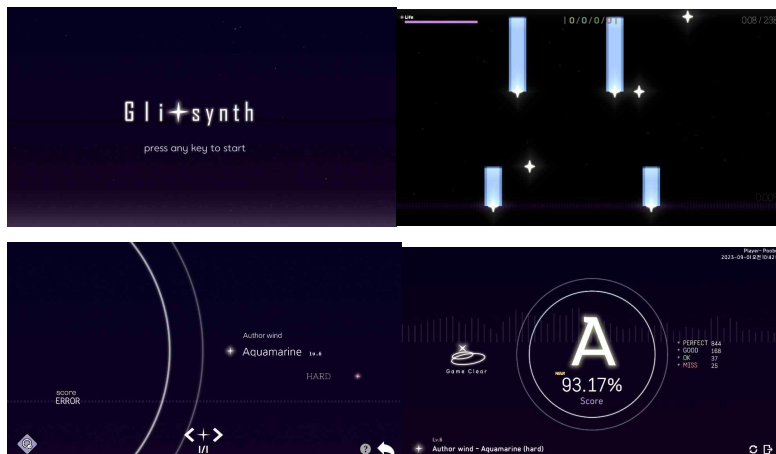


그림 2-29) 수정된 게임 결과물

그림 2-29)는 앞서 서술한 방식 대로 메커니즘을 수정하고, 게임 그래픽을 교체하여 제작된 별개의 게임이다. 이와 같이 메커니즘을 수정하는 것만으로도 이처럼 별도의 게임을 만들 수 있었다. 이는 앞서 언급했듯이, 이 졸업과제의 목표대로, 리듬 게임을 개발하는 재사용할 수 있는 일반화된 코드베이스가 될 수 있음을 보인 것이다.

위의 기능을 구현하기 위하여 고려했던 부분은 기존에 나와있는 커스텀 키보드와 컨트롤러의 융합이다. 커스텀 키보드의 자유로운 커스터마이징과 자가 수리가 가능하다는 장점을 그대로 컨트롤러에 구현할 수 있다면 이상적인 전용 컨트롤러가 될 수 있으리라 생각한다.

3.2) 전용 컨트롤러 설계

전용 컨트롤러는 크게 2개의 완성된 부품의 형태로 구성할 예정이며 각각의 부품은 왼손용 키보드와 오른손용 키보드이다. 왼손용 키보드와 오른손용 키보드를 따로 구별한 이유는 일반적인 컨트롤러의 경우에는 장기간 사용 시에 어깨가 움츠러들고, 높이가 있어 손목에 지속적인 통증을 발생시킨다. 이는 손목 터널 증후군 혹은 어깨 결림에 영향을 줄수 있기에 취지에 맞는 인체공학적인 컨트롤러를 설계하였다.

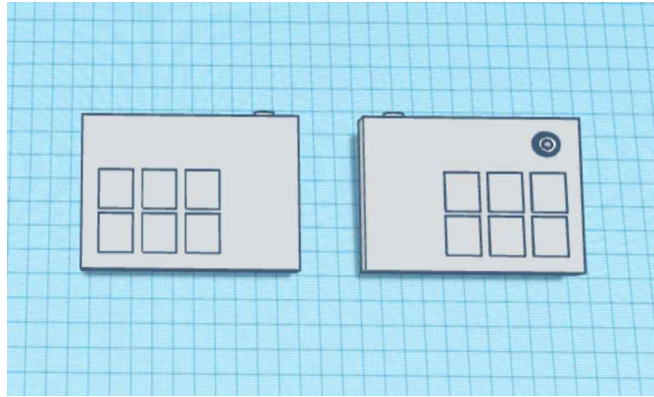


그림 3-2) 왼손용 컨트롤러, 오른손용 컨트롤러 CAD도면

위의 사진에서 확인할 수 있듯이 왼손의 키보드에는 단순히 6개의 물리적 버튼과 오른손 전용 키보드에는 6개의 물리적 버튼과 1개의 노브로 구성이 되어있다. 우선 오른손용 키보드는 한 손 인터페이스를 염두해 두고 설계를 하였다. 전용 컨트롤러의 펌웨어를 통해 개인이 원하는 키로 키매핑을 할 수 있게 설계되었으며 노브를 통해 게임 중간 볼륨 조절을 용이하게 하였다. 각각의 컨트롤러는 USB-C타입의 단자를 가지고 있으며 C타입 분배기를 통해 2개의 컨트롤러를 동시에 운영할 수 있다. 전용 컨트롤러를 사용할 경우에 일반적인 컨트롤러와 달리 어깨를 움츠리지 않고 펴고 사용할 수 있으며, 입력이 발생하는 높이를 낮추어 손목에 부담이 덜 하도록 설계하였다. 또한 사람이 가장 편안하게 타이핑 할 수 있다는 6°의 경사를 컨트롤러에 담아 냈다.

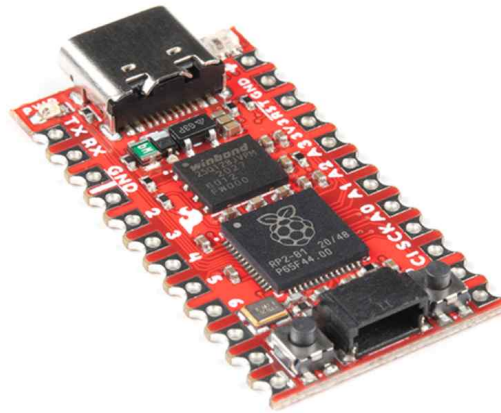


그림 3-3) 라즈베리파이 RP2040

컨트롤러의 기반이 될 MCU로는 RP2040을 사용하였다. RP2040을 사용한 첫 번째 이유는 비교적 저렴한 단점과 컨트롤러의 특성상 많은 기능이 필요 없기 때문이다. 또한 주변 장치와의 호환성이 좋아 자유도가 높고 컨트롤러의 펌웨어의 기반으로 사용할 QMK에서 RP2040을 정식으로 지원하기 때문이다. 또한 확장성에 있어 무선연결 모듈을 지원하기에 이를 채택하였다.

3.3) 무선 연결 구성

첫 번째, 컨트롤러와 PC간의 무선 연결과 유선연결을 하기 위한 기능의 구성 단계이다. 우선 RP2040 MUC와 호환되는 블루투스 모듈중 가장 저렴하고 교체가 용이한 부품을 찾았다.

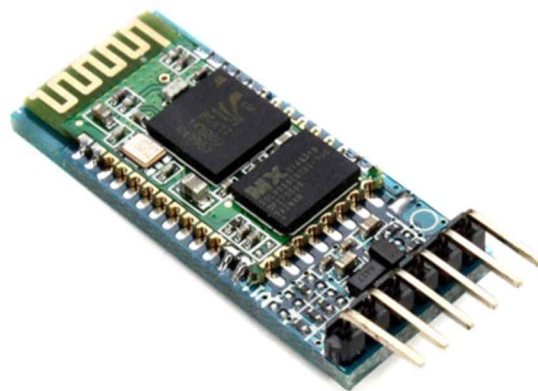


그림 3-3) HC-05 블루투스 모듈

위의 사진은 RP2040과 호환되는 블루투스 모듈로써 RP2040과 UART로 통신한다. 블루투스의 버전이 낮기에 대부분의 기기에서 호환이 가능하다.



그림 3-6) 소형 LCD

이후 이와 같은 소형 LCD를 이용하여 각 통신 연결시 연결 실패 혹은 연결 상태를 출력할 수 있도록 한다. 라즈베리파이의 IC2 인터페이스 모듈을 사용하여 화면에 바로 출력할 수 있도록 한다.

```

01: def Wifi_Mode(ssid, password):
02:     wifi = pywifi.PyWiFi()
03:     iface = wifi.interfaces()[0]
04:     iface.disconnect()
05:     time.sleep(1)
06:     profile = pywifi.Profile()
07:     profile.ssid = ssid
08:     profile.auth = pywifi.const.AUTH_ALG_OPEN
09:     profile.akm.append(pywifi.const.AKM_TYPE_WPA2PSK)
10:     profile.cipher = pywifi.const.CIPHER_TYPE_CCMP
11:     profile.key = password
12:     iface.remove_all_network_profiles()
13:     tmp_profile = iface.add_network_profile(profile)
14:     iface.connect(tmp_profile)
15:     time.sleep(5)
17:     return iface.status() == pywifi.const.IFACE_CONNECTED
...

```

코드 3-1) 와이파이 모듈

```

01: def Bluetooth_Mode():
02:     def setup_raspberry_pi_as_bluetooth_device():
03:         local_name = "Controler"
04:         local_address = "11:11:11:11:11:11"
05:         service_id = bluetooth.SERIAL_PORT_CLASS
06:         port = 11
07:         bluetooth.advertise_service(name=local_name, service_id=service_id,
08:         service_classes=[service_id,
09:         bluetooth.SERIAL_PORT_PROFILE],
10:         profiles=[bluetooth.SERIAL_PORT_PROFILE],
11:         address=local_address, port=port)
...

```

코드 3-2) 블루투스 모듈

각 모드에 대한 호출 함수는 위와 같다. 와이파이 연결의 경우 오픈소스를 참고하여 작성하였다. 블루투스 모드 같은 경우에는 라즈베리파이 자체를 pc에서 검색하여 연결할수 있도록 작성하였다. hiconfig를 통해 해당 라즈베리파이의 블루투스 ID를 확인한 이후 교체할 계획이다. 포트같은 경우에는 임의의 포트로 할당해 주었다.

```

01: while True:
02:     STATE_WIFIMODE = GPIO.input(selector_pin1)
03:     STATE_BLUETOOTHMODE = GPIO.input(selector_pin2)
04:     STATE_WIREDCONNECT = GPIO.input(selector_pin3)
05:     if STATE_WIFIMODE == GPIO.LOW:
06:         if Wifi_Mode(wifi_ssid, wifi_password):
07:             lcd_string("Wi-Fi", LCD_LINE_1)
08:             lcd_string("connected", LCD_LINE_2)
09:         else:
10:             lcd_string("Wi-Fi connection", LCD_LINE_1)
11:             lcd_string("is failed", LCD_LINE_2)
12:     elif STATE_BLUETOOTHMODE == GPIO.LOW:
13:         devices = Bluetooth_Mode()
14:         if len(devices) > 0:
15:             lcd_string("Bluetooth", LCD_LINE_1)
16:             lcd_string("connected", LCD_LINE_2)
17:         else:
18:             lcd_string("No Bluetooth", LCD_LINE_1)
19:             lcd_string("devices found", LCD_LINE_2)
20:     elif STATE_WIREDCONNECT == GPIO.LOW:
21:         lcd_string("Wired", LCD_LINE_1)
22:         lcd_string("connected", LCD_LINE_2)
23:
24:     time.sleep(0.1)
25:
...

```

코드 3-3) 스위칭 모듈

무선 연결을 위한 소스코드의 내용이다. 셀렉터 스위치의 상태에 따라 각 연결을 High, low의 값으로 입력 받는다. 이후 상태에 따라 wifi, bluetooth, wired connect로 전환할수 있도록 함수를 호출한다. 각 함수 호출시에 연결된 상태에 따라 현재 연결 상태를 보여줄수 있도록 LCD에 문구를 표시한다.

```

01: def lcd_byte(bits, mode):
02:     bits_high = mode | (bits & 0xF0) | LCD_BACKLIGHT
03:     bits_low = mode | ((bits << 4) & 0xF0) | LCD_BACKLIGHT
04:     bus.write_byte(I2C_ADDR, bits_high)
05:     lcd_toggle_enable(bits_high)
06:     bus.write_byte(I2C_ADDR, bits_low)
07:     lcd_toggle_enable(bits_low)
08: def lcd_toggle_enable(bits):
09:     time.sleep(E_DELAY)
10:     bus.write_byte(I2C_ADDR, (bits | ENABLE))
11:     time.sleep(E_PULSE)
12:     bus.write_byte(I2C_ADDR, (bits & ~ENABLE))
13:     time.sleep(E_DELAY)
14: def lcd_string(message, line):
15:     message = message.ljust(LCD_WIDTH, " ")
16:     lcd_byte(line, LCD_CMD)
17:     for i in range(LCD_WIDTH):
18:         lcd_byte(ord(message[i]), LCD_CHR)
19: bus = smbus2.SMBus(1)
...

```

코드 3-4) LCD 모듈

LCD 모듈의 구성 함수는 위와 같다.

초기에 I2C 주소와 LCD 의 크기를 설정해 준 이후, 문자열을 출력하도록 해 주었다.

초기에 I2C 주소와 LCD 의 크기를 설정해 준 이후, 문자열을 출력하도록 해 주었다.

3.4) 컨트롤러 버튼 구성

컨트롤러 버튼은 크게 6개의 입력 스위치와 1개의 노브로 구성된다. 입력 스위치는 각각 6개의 노드를 입력하기 위함이고 노브는 볼륨을 조절하기 위한 용도로 사용하였다. 전용 컨트롤러의 유지 보수성을 높이기 위해 스위치는 일반적인 기계식 키보드에 사용되는 Cherry MX기반의 3핀 스위치를 사용하되 개인의 자유도를 높이기 위해 5핀의 스위치도 호환이 되도록 5개의 홀타이트를 사용하였다.

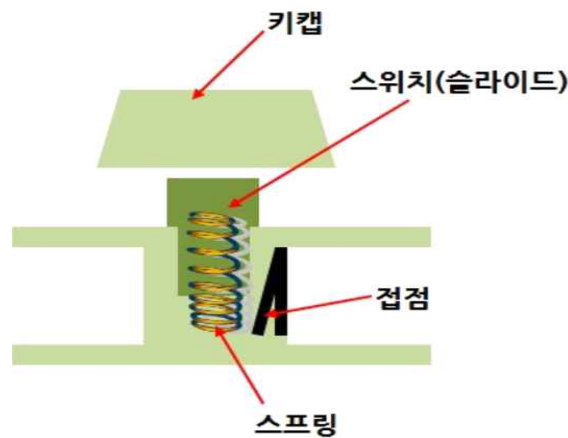


그림 3-7) 스위치 구성

스위치의 구성은 위와 같다. 키캡, 스템(슬라이드), 스프링, 하우징으로 구성이 되어 있으며, 스템의 모양과 하우징의 구성 물질에 따라 다양한 타건감과 타건음을 가지고 있기 때문에 개인의 니즈를 확실하게 충족할 수 있다고 판단하였다. 스위치의 작동 원리는 스위치를 눌리면 스프링이 압축되면서 스템이 안으로 들어가 접점부를 누르게 되고, 이때 전류가 흐르면서 키 입력을 인식하는 식이다. 스위치는 각각 하나의 GPIO포트와 접지를 위해 10kΩ을 연결하여 GND에 같이 연결하도록 한다. 여기서 주의할 점은 RP2040은 5V를 지원하지 않기 때문에 모든 포트를 3.3V에 연결해야 한다. 일반적인 기계식 키보드를 사용할 것이고, 여러 가지 입력압과 걸림을 가진 스위치들을 교체가 가능하도록 만든다.

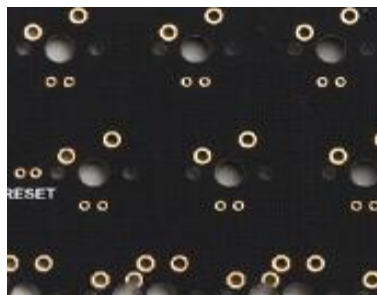


그림 3-8) 키보드 PCB 기판



그림 3-9) 카일 핫스왑 소켓

그림 3-8)은 흔히 볼수 있는 키보드의 PCB 기판이다. 해당 기판은 기계식 스위치를 장착하기 위한 PCB기판으로 기계식 스위치를 하나하나 납땜을 해주어야 한다. 이로 인해 사용자들의 접근성과 유지보수성을 해한다고 생각했다. 그림 3-9)는 전용 컨트롤러의 유지보수성과 커스터마이징에 제일 중요한 역할을 하는 카일사의 핫스왑 소켓이다. 해당 부품은 일반적으로 납땜 방식을 사용하는 스위치들과는 달리, 풀러로 스위치를 장착하고 분해할 수가 있다.

3.5) 전용 컨트롤러 펌웨어

전용 컨트롤러의 펌웨어는 가장 대중적으로 쓰이는 키매핑 프로그램인 QMK를 기반으로 작성되었다. 특히나 QMK에서는 RP2040을 정식으로 지원하기 때문에, 개발 단계에 있어서 접근성이 용이하여 채택하였다. 기본적인 펌웨어의 작성은 QMK에서 공식으로 배포한 <https://docs.qmk.fm/#>를 따라서 진행하였다.[1]

```
01:
02: qmk setup
...
```

코드 3-5) 펌웨어 업로더 세팅


```

'C:\WINDOWS\system32\drivers\etc\hosts' -> '/etc/hosts'
Welcome to QMK MSYS!
* Documentation:  https://docs.qmk.fm
* Support:       https://discord.gg/Uq7gcHh

If you have not already done so,
run qmk setup to get started.
run qmk compile -kb <keyboard> -km default to start building.
run qmk config user.hide_welcome=True to hide this message.
[김세영@DESKTOP-E7QBLHK ~]$ qmk setup
❏ Could not find qmk_firmware!
Would you like to clone qmk/qmk_firmware to 'C:/Users/김세영/qmk_firmware'? [y/n] y
'/c/Users/김세영/qmk_firmware'에 복제합니다...
Updating files:  0% (298/36990)
Updating files:  1% (370/36990)
Updating files:  1% (615/36990)
Updating files:  2% (740/36990)

```

그림 3-10) QMK Firmware Uploader 실행화면

위의 소스코드를 QMK Firmware Uploader에 입력하면 그림 10-9)와 같은 창이 나타나면서 펌웨어의 세팅을 시작한다.

```

01:
02: qmk config user.keyboard=controler
03: qmk new-keymap -kb <controler>
...

```

코드 3-6) 컨트롤러 설정

```

[김세영@DESKTOP-E7QBLHK ~]$ qmk new-keymap
# Generating a new keymap

Name Your Keymap
Used for maintainer, copyright, etc

Your GitHub Username? error302-pnu-cse5

```

그림 3-11) QMK Firmware Uploader 실행화면

이후 파일에 컨트롤러의 키맵 파일이 생성되면, 원하는 키에 맞게 키맵을 수정하여 준다.

```
01:
02:  [_BL] = LAYOUT(
03:      KC_KB_VOLUME_UP, KC_KB_VOLUME_DOWN
04:      KC_S,   KC_D,   KC_F,
05:      KC_J,   KC_K,   KC_L,
06:  ),
...
```

코드 3-7) 컨트롤러 keymap.c 소스코드 일부

```
01:
02:  [{
03:      "x": 5
04:  },
05:  "Knob"
06:  ],
07:  [{
08:      "y": 1
09:      "M1", "M2", "M3"
10:  },
11:  [
12:      "J", "K", "L"
13:  ]]
...
```

코드 3-8) 컨트롤러 layout.json 소스코드 일부

위와 같이 국제표준으로 정의된 키보드의 키 값들을 레이아웃 맞춰서 수정해주었다.

3.6) 컨트롤러 하우징 구성

하우징은 3D프린터로 출력을 할 수 있도록 CAD를 사용하여 3D프린트로 출력을 할 수 있도록 제작하였다. 우선 오른손용 컨트롤러의 하우징을 살펴보자.

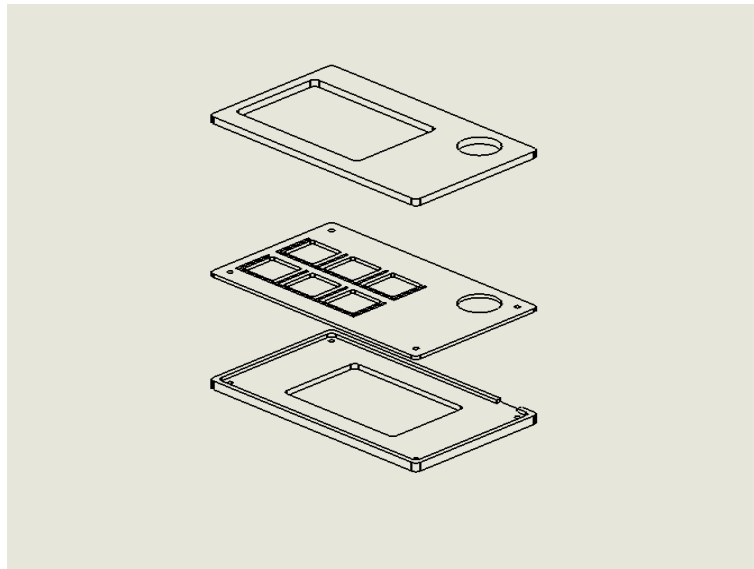


그림 3-12 컨트롤러 조감도

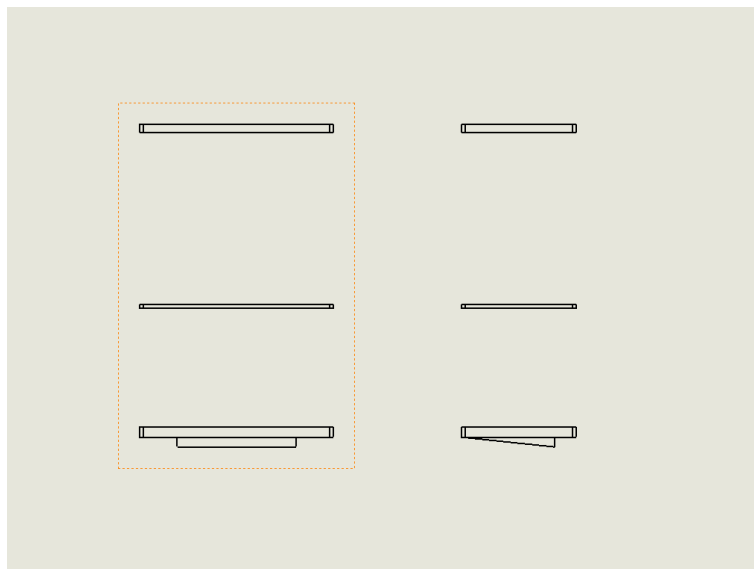


그림 3-13) 컨트롤러 측면 사진

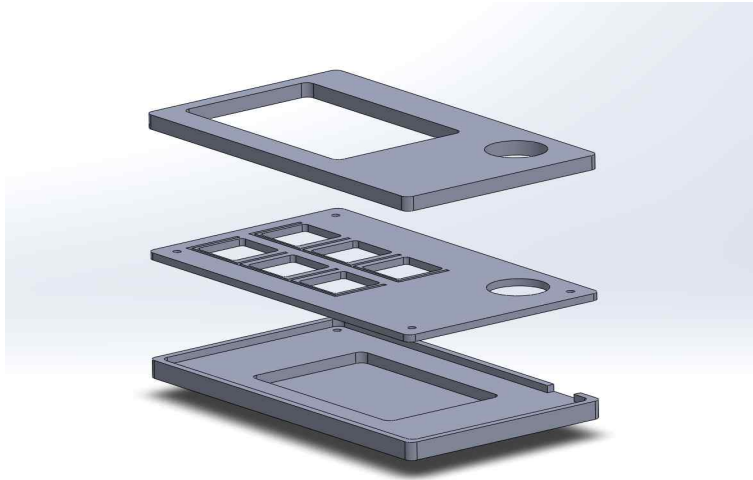


그림 3-14) 컨트롤러 3D 조감도

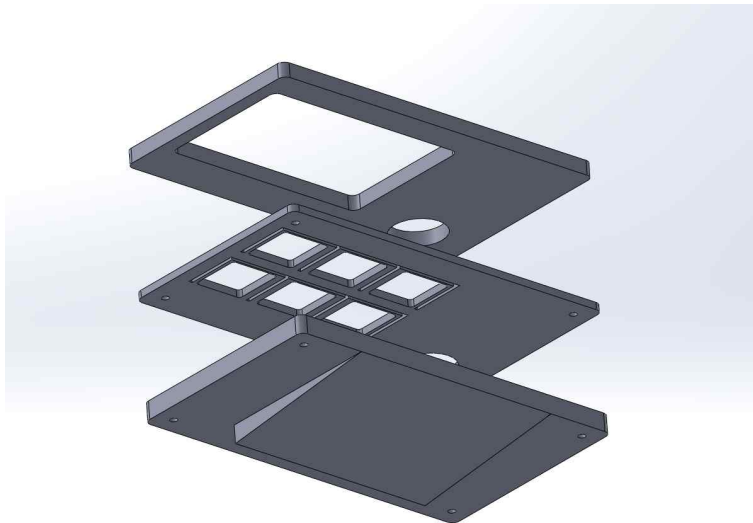


그림 3-15) 컨트롤러 3D 조감도

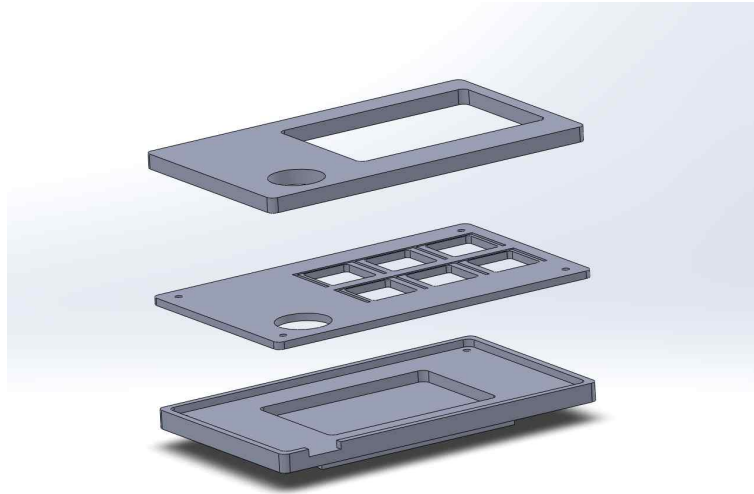


그림 3-16) 컨트롤러 3D조감도

위의 그림을 참고하면 전체적인 모습을 확인할 수 있다. 컨트롤러의 중요한 특징으로는 첫 번째, 볼륨 조절을 위한 노브를 지원하고 두 번째, 보강판을 플렉스 컷으로 제작하여 조금 더 부드러운 타건감을 느낄 수 있도록 하였다. 세 번째, 6도의 경사를 바닥면에 주어 장기간 이용시에도 손목의 부담을 덜도록 하여 사용자의 건강을 고려하였다. 네 번째, 주위에서 흔히 구할 수 있는 규격인 H2.0규격의 나사 홀을 사용했기 때문에 추가적인 부품의 구입 없이도 손쉽게 주변에서 여분의 부품을 구할 수 있도록 하였다. 또한 바닥면을 평평하게 만들었기 때문에, 개인이 직접 범폰 등을 구매하여 미끄럼 방지 등의 기능을 추가할 수 있도록 확장성을 넓혀 주었다.

추가적인 특징으로는 PCB기판이 들어갈 자리와, 각 레이어 사이에 단차를 만들어 소음에 민감한 사용자인 경우 흡음재를 추가할 수 있도록 각각 2T정도의 여유공간을 제공한다.

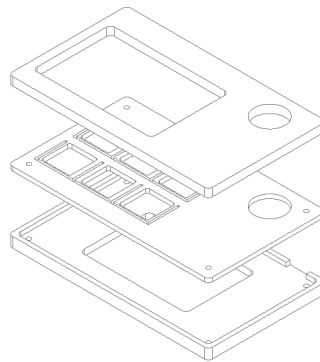


그림 3-20) 카드 설계도

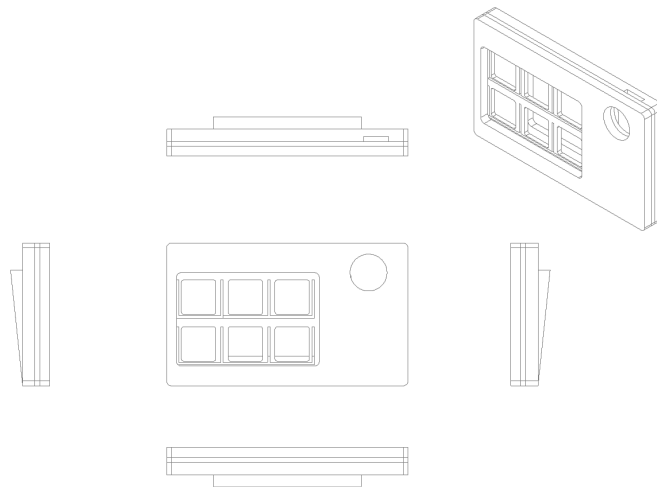


그림 3-21) 카드 설계도

위의 그림들과 같이 DWG파일로 작성하여 3D프린트에서 출력이 가능하도록 만들어 전용 컨트롤러의 가격 경쟁력을 높일 수 있으리라 생각한다. 왼손용 컨트롤러는 해당 컨트롤러에서 노브만 없앤 후에, 좌우 반전을 하여 제작하였으며, 2개의 컨트롤러는 2포트 C타입 분배기를 통해 컴퓨터에 직접 연결된다.

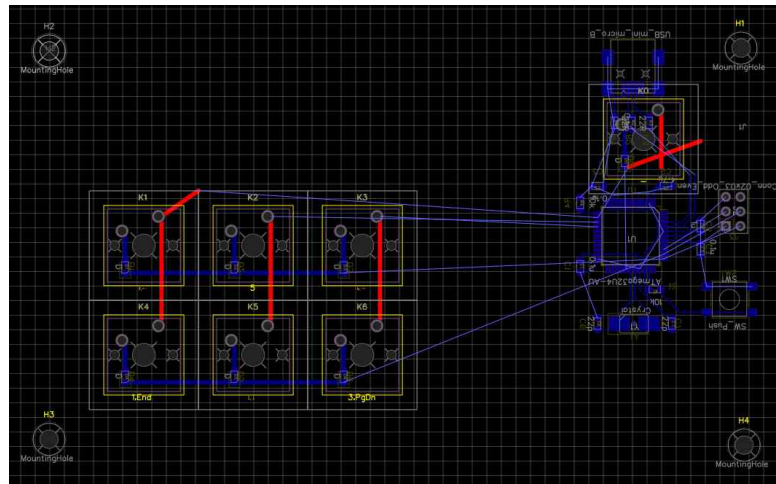


그림 3-22) 전용 컨트롤러 pcb 설계도

그림 22) 는 전용 컨트롤러와 호환되도록 제작한 컨트롤러이다. 현재는 Atmogo32ua 칩셋이 장착되어 있지만, rp2040으로 업그레이드를 진행할 예정이다.

4. 연구 결과 분석 및 평가

게임 시스템

과제 착수 시 계획한 사양대로 게임이 성공적으로 구현되었다. 제시한 일반화된 형태의 리듬게임을 실제로 구현하였고, 이 코드 베이스를 이용하여 실제로 다른 게임으로 변형한 예시를 보여 코드 재사용 측면에서 가치가 있음을 보였다. 그 외 리더보드, UI 등 게임을 위해 필수적인 게임 기능들을 구현하였고, 이 역시도 재사용 가능하여 다른 게임을 만들 때의 생산성을 높여줌을 확인했다.

컨트롤러

시중에 유통되는 컨트롤러와는 달리 전용 컨트롤러는 이용자의 건강과 접근성에 초점을 맞춘 제품이다. 모든 재료들을 주변에서 구할수 있거나 인터넷으로 간편하게 주문할 수 있는 재료들을 사용하였으며, 하우스 설계의 특징으로 이용자의 건강을 고려했고 3D프린터로 출력을 하기 때문에 개인이 하우스의 재료(PC, FR4, ALU, BRASS)를 선택할 수 있도록 하였다, 또한 기계식 스위치를 사용했기 때문에 본인이 원하는 스위치를 직접 사용할 수 있다. 단언컨대 현재 유통되는 컨트롤러 중에 가장 높은 자유도를 가진 컨트롤러라고 할 수 있다.

5. 결론 및 향후 연구 방향

게임 시스템

비동기 입력을 구현할 때, DirectInput이나 RawInput등의 API를 사용하지 않고 GetAsyncKeyState를 사용하였다. 이것이 원인인지는 불명이나, 간헐적으로 키 입력이 무시되는 현상이 나타났다. 현재 Unity의 입력 시스템을 사용하면 이러한 현상이 사라지나, 게임 프레임과 동기화되어 발생하는 정밀도 문제가 발생한다. 이를 앞서 언급한 입력 API를 사용하여 개선한다면 보다 품질을 높일 수 있을 것이다.

현재 시스템은 멀티플레이어를 고려하지 않고 제작되었다. 몇몇 리듬 게임의 경우 룸 시스템을 이용한 멀티플레이어를 지원하는데, 이러한 부분 역시 고려하여 시스템을 확장한다면, 현재보다 더 넓은 범위의 게임 제작을 지원할 수 있을 것이다.

컨트롤러

직접 하우징을 설계하기 위해 3D Cad와 Kicad를 공부하느라 생각보다 너무 많은 시간이 소요되었고, 국내 PCB제작 업체들의 가격이 너무 비싸 해외에서 주문제작을 하느라 기간이 오래 소요되어 아직 PCB기판을 받지 못하였다. 또한 앞에서 설명한 계획과 같이 HC-05 칩셋을 RP2040에 부착하여 무선 연결이 가능하도록 기현을 구현하고 싶었지만, 컨트롤러의 기반이 된 펌웨어인 QMK에서 자꾸 HC-05 칩셋을 인식하지 못하는 에러가 계속해서 발생하여 현재 수정 중에 있지만 아직 배움이 모잘라 많은 어려움이 있다. 추후 확장성을 염두해두어 하우징 내부에 빈 공간을 추가적으로 더 설계 하였으며 바닥면 같은 경우에는 500mA의 리튬이온 전지가 들어갈 수 있도록 홀을 만들어 두었다.

해당 기판들이 정상적으로 호환 되지 않으면 최후의 방법으로 졸업작품 전시회에는 rp2040이 아닌 라즈베리파이4를 사용하여 스위치를 라즈베리파이피코에 직접 장착한 후 시연할 예정이다. 추후 연구 방향으로는 첫 번째, 완벽한 무선 연결 지원이다. 두 번째로는 왼손용 컨트롤러에 현재 상태를 나타내는 LCD를 추가하여 리듬 게임 전용 양손 컨트롤러를 제작하여 직접 아마존이나 이베이 등의 오픈마켓에서 판매해 볼 계획을 가지고 있다.

참고문헌

- [1] [QMK Firmware Guide](https://docs.qmk.fm/), Availble : <https://docs.qmk.fm/>(downloaded 2023, 10. 18)
- [2] Facepunch Steamworks Guide, Available : <https://wiki.facepunch.com/steamworks/>
(2023,10. 17)
- [3] FMOD API, Available: <https://www.fmod.com/docs/2.00/api>(2023,10. 17)
- [4] Unity Scripting Refrence, Available: <https://docs.unity3d.com/ScriptReference/>(2023,10. 17)