설계과제2 최종보고서

4조 : 가상현실 드럼(VR DRUM)

목차

연구기간	2016.08.22~2017.05.31
제출일	2017.06.12
대표/행정/제작	권원표(20140931)
프로그래밍/개발	유정민(20130203)
디자인/애니메이션	이종건(20140707)

1. 개요

대부분의 사람들이 아파트에서 생활하는 우리나라에서는 가정에서 드럼 세트를 갖추기가 쉽지 않다. 기존 드럼 세트는 가격이 비싸고 부피가 클 뿐만 아니라 소리가 너무 커서 이웃 간 소음 문제를 발생시킬 수 있기 때문이다. 그래서 가정에서 사용할 수 있는 새로운 드럼이 필요했고, 우리는 가상현실 기술을 사용하여 소음과 부피 문제를 해결한 새로운 형태의 드럼을 만들었다.

본 설계과제는 '가정용 연습 환경을 위한 가상현실 드럼 제작'을 목표로 하여 연습과 연주가 모두 가능한 드럼을 가상현실에서 구현했다. 먼저 HTC VIVE 를 통해 가상 공간을 설정하고 3D 가상현실 영상을 사용자에게 보여주며, Unity 3D 프로그램에서 메인 프로그램을 제작하여 실행한다. 이때 가상현실 내에서 3D 이미지 모델링은 무료로 배포된 에셋(asset)과 Blender 라는 그래픽 툴을 이용하여 만들었으며, 드럼 소리는 가상 악기를 이용했다.

완성된 가상 드럼을 실행하면 연주자는 VIVE 컨트롤러와 아두이노 드럼 페달을 조작하여 드럼을 연주할 수 있으며, 실제 드럼처럼 타격 위치와 세기에 따라 다른 소리가 재생된다. 특히 시각적으로 심벌의 흔들림과 타격 애니메이션을 보여주어 현실과 이질감이 생기지 않도록 했으며, 컨트롤러를 통해 양손에 진동 반응을 전달한다. 연주자가 헤드폰을 사용할 경우 이 모든 내용은 연주자에게만 전달되며, 외부로 어떤 소리도 새어나가지 않아 소음이 없다. 헤드폰이 아닌 스피커와 연결하면 공개적으로 연주할 수도 있다.

본 가상 드럼은 현재도 연주용으로 사용할 수는 있지만, 본격적으로 사업화하기 위해서는 아직 개선해야 할 문제가 있으며, 특히 분리된 소프트웨어를 하나의 파일로 만들고, 전용 드럼 페달을 독립된 제품으로 만들어야 한다. 한편 다른 활용 방안으로는 같은 기술을 다른 악기에 적용하여 드럼처럼 일상에서 접하기 어려운 악기의 접근성을 향상시킬 수 있다. 이를 통해 악기 연주라는 취미 생활에 대한 대중의 접근성을 높여 문화생활 확대에 기여할 수 있을 것이다.

2. 연구 필요성 및 기대효과

A. 연구 필요성

(a) (b) (c)

그림 1 - 기존에 사용중인 여러 종류의 드럼

우리나라에서는 제대로 된 드럼 연습을 집에서 할 수 없다. 그림 1-(a)의 일반적인 드럼세트는 부피가 크고 소리도 시끄럽기 때문에 일반 가정집에서 사용하기는 어렵고, 소음 문제를 해결한 전자 드럼(그림 1-(b))도 악기의 부피는 똑같기 때문에 아파트가 많은 우리 나라에서 사용하기엔 여전히 어렵다. 한편, 드럼세트의 형태를 포기하고 부피를 대폭 줄인 패드 형태의 전자 드럼(그림 1-(c))도 등장했지만 실제 드럼과의 이질감이 커 연주 시 불편한 점이 많으며, 특히 실제와 형태가 다르므로 연습용으로 사용하기에는 적합하지 않다. 따라서 우리는 연습실과 비슷한 환경을 제공하면서 부피 문제와 소음 문제가 없는, 가정에서 사용할 수 있는 새로운 형태의 드럼이 필요하다.

따라서 본 설계과제는 드럼세트의 큰 부피와 소음을 해결하기 위하여 가상현실을 이용한 새로운 형태의 드럼을 고안했다. 가상현실 드럼은 부피가 거의 없으므로 휴대성과 공간 활용도를 대폭 늘릴 수 있고, VR 기기에서 헤드폰으로 소리 신호를 바로 출력할 수 있어 소음 문제도 해결할 수 있다.

B. 목적 및 기대효과

본 설계과제의 목적은 '가정용 연습 환경을 위한 가상현실 드럼 제작'이다. 따라서 초심자의 악기 체험 및 연습용으로 사용가능하면서 동시에 전문적인 연주도 가능한 드럼세트를 목표로 프로젝트를 진행했다.

가상현실 드럼은 VR 기기만 갖추어져 있다면 가격도 매우 저렴하며, 기존 드럼의 부피와 소음 문제를 모두 해결할 수 있다. 또한 기존 드럼에 비해 휴대가 간편하고, 장소의 제약이 적어 드럼이라는 악기에 대한 접근성을 크게 향상시킬 수 있다. 나아가 같은 기술을 일상에서 쉽게 접하기 힘든 팀파니나 그랜드 하프 같은 악기에 적용하여 VR로 만든다면 악기 연주라는 취미 생활에 대한 접근성을 높여 문화생활 확대에 크게 기여할 것이다.

C. 사업화 가능성

본 설계과제를 판매용으로 개발하여 사업화하기 위해서는 먼저 가상 드럼 전용 페달의 제품화가 필요하다. VIVE는 이미 상용화된 상품이지만 드럼 연주에 필요한 페달은 따로 구매해야 한다. 따라서 VIVE와 쉽게 연동 가능하도록 전용 페달 액세서리 제품을 만들어 판매해야 하며, 이를 위해 현재 외부로 노출되어 있는 센서와 아두이노 모듈을 소형화하여 페달 패키지 안에 숨겨야 한다. 기술적으로 어려운 일은 아니지만, 센서의 높은 인식률을 유지하면서 더 작게, 가격도 저렴하게 만드는 점이 어려울 것이다.

사업화의 다른 조건으로는 소프트웨어의 단일화가 필요하다. 현재는 Unity 3D로 만든 메인 프로그램에서 MIDI 신호를 출력하면 BFD3 가상 악기에서 이 신호를 받아 소리를 재생하는 구조로 되어 있다. 하지만 단독 실행 파일로 만들기 위해서는 가상 악기도 프로그램 안에 반드시 포함되어야 하며, 이를 위해서는 저작권을 침해하지 않는 자체 가상 악기와 DAW(Digital Audio Workstation; 디지털 오디오의 재생, 녹음 및 편집 등의 작업을 위한 워크스테이션[1])의 개발이 필요하다.

위 두 조건을 만족해서 단독 실행 가능한 어플리케이션 상품으로 만든다면, 그 다음엔 VR 시장이 커지고 VIVE 보급률이 충분히 높아졌을 때 이 어플리케이션을 팔아서 수익성을 기대할 수 있다. 하지만 아직은 VR 시장의 성장이 불확실하기 때문에 사업화하려면 시장의 동향을 면밀히 살펴볼 필요가 있다.

3. 용어 정리

앞으로의 내용에 대한 이해를 돕기 위해 드럼의 기본적인 용어 및 개념을 정리한다.

A. 각 북과 심벌의 명칭

그림 2 – 드럼세트의 구성과 명칭

일반적인 드럼세트에서 북과 심벌의 구성과 명칭은 그림 2와 같다. 본 보고서에서는 구별의 편의를 위해 톰톰(Tom-toms) 중 작은 것부터 차례로 하이톰, 미드톰, 로우톰이라고 하겠다.

B. 드럼 스틱

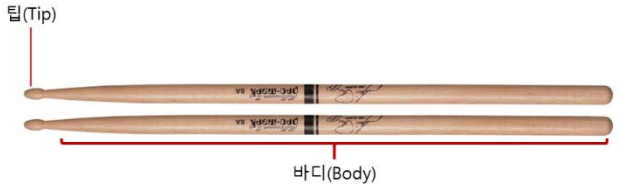

그림 3

본 설계과제에서는 드럼 스틱을 크게 두 부분으로 나누어 구분하고, 각 부분을 따로 3D 이미지로 모델링했다. 그 이유는 아래에서 설명할 북과 심벌의 여러가지 타격 방법을 구분하기 위해서다.

C. 북 타격 종류

일반적으로 북을 타격하는 방법은 크게 오픈 히트와 림샷으로 나눌 수 있다(그림 4). 오픈 히트란 스틱으로 북을 때리는 가장 기본적인 방법이며, 림샷은 오픈 히트를 하면서 동시에 림(북의 가장자

리)을 때려 더 크고 강한 소리를 내는 방법이다. 이외에도 스틱으로 림 만을 치는 방법 등도 있다.

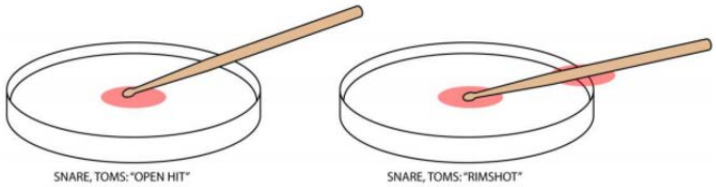

그림 4

D. 심벌 타격 종류

심벌의 소리를 내는 방법은 스틱의 어느 부분으로 심벌의 어느 부분을 치는가에 따라 크게 세 종류가 있다. 일반적으로 크래시 심벌은 스틱의 바디 부분으로 심벌의 가장자리를 치며, 하이햇과 라이드 심벌은 스틱의 팁 부분으로 심벌의 윗면을 때린다. 이중 라이드 심벌의 경우 특이하게 가운데 볼록하게 올라온 부분을 때리기도 한다. 이 경우 심벌에서 나는 소리가 완전히 다르다.

하이햇 심벌은 페달을 통해 두 심벌 사이의 거리를 조절할 수 있으며, 그 열린 정도에 따라 소리가 달라진다. 두 심벌의 거리에 따라 오픈, 세미오픈, 클로즈로 구분할 수 있으며 본 설계과제에서는 페달과 기울기 센서를 사용하여 5단계로 소리를 나누었다.

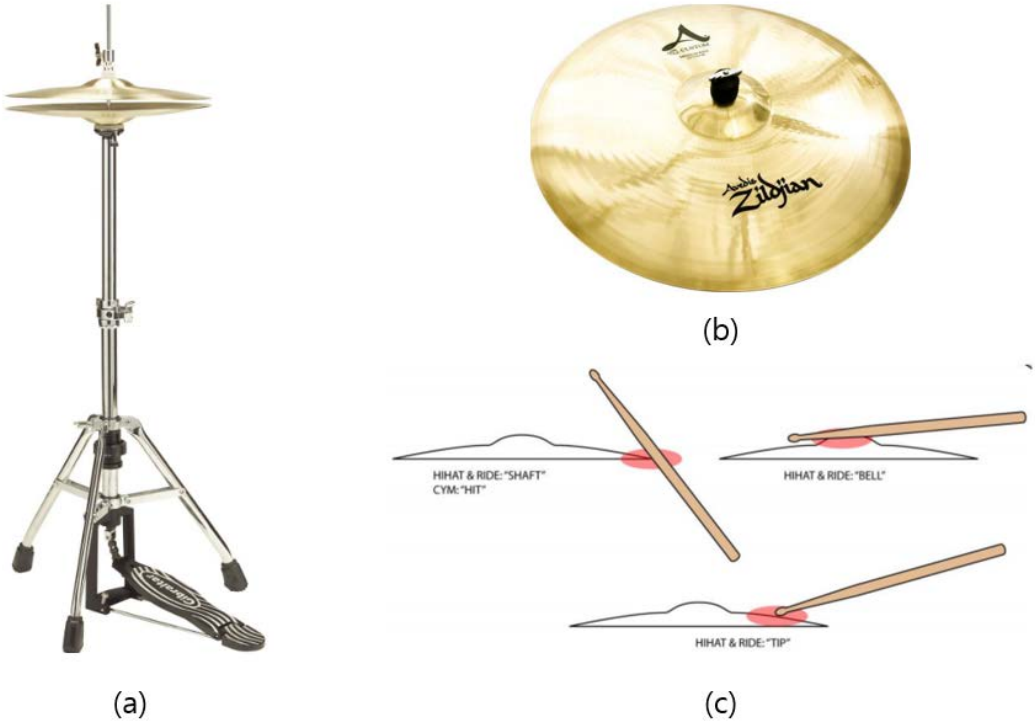

그림 5 – (a) 하이햇, (b) 라이드 심벌, (c) 심벌 타격의 종류

4. 전체 구조

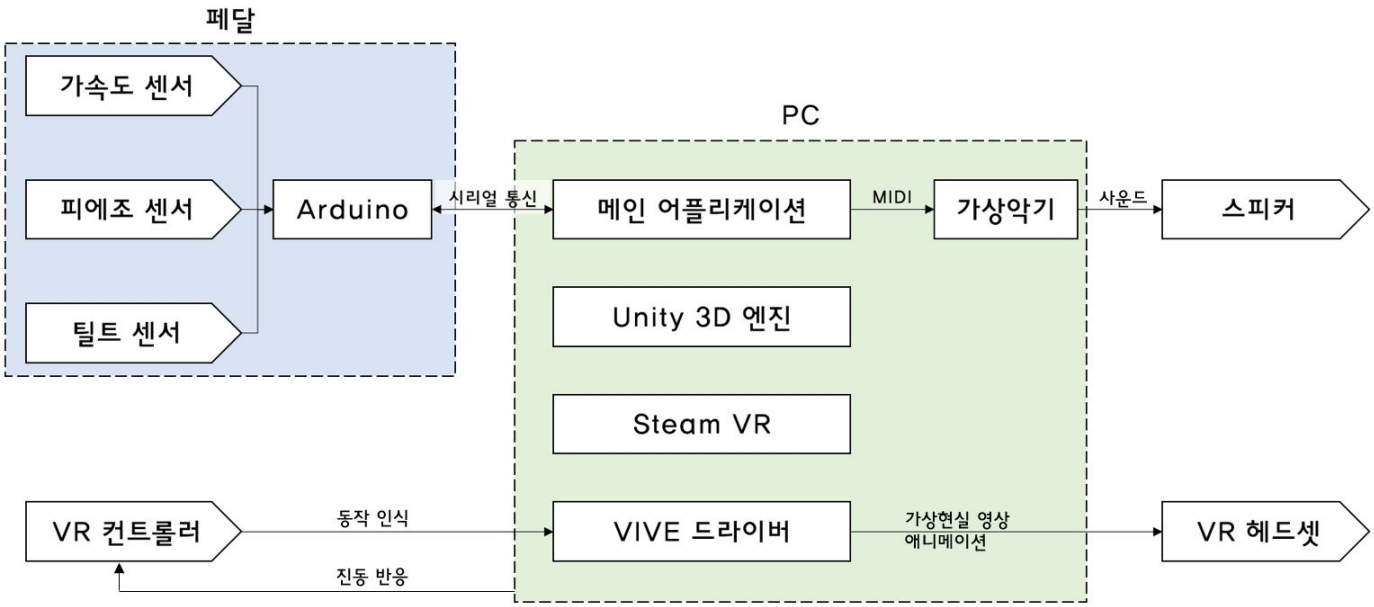

그림 6 – 전체 프로그램 구조

5. 하드웨어 및 소프트웨어

본 설계과제에 사용된 주요 하드웨어와 소프트웨어를 소개한다.

A. HTC VIVE

HTC VIVE는 2016년에 공개된 HTC와 Valve Corporation이 공동개발한 가상현실 헤드셋으로 패키지에 컨트롤러 한 쌍과 베이스 스테이션 한 쌍이 포함되어 있다. VIVE는 기존 가상현실 기기에는 없는 'room scale' 기능을 도입하여 베이스 스테이션의 센서를 통해 실제 공간을 3D 공간으로 설정할 수 있다.

그림 7 – HTC VIVE와 컨트롤러, 베이스 스테이션

VIVE는 베이스 스테이션이 헤드셋과 컨트롤러의 위치를 실시간으로 추적하며 인식이 빠르고 정확하다. 이를 통해 양 손의 위치를 항상 추적할 수 있다는 장점과 사용자에게 실제 물체를 잡는 느낌까지 동시

에 줄 수 있어 드럼 스틱의 역할을 하기에 가장 적합하다.

B. Unity3D

Unity 3D는 3D 비디오게임이나 건축 시각화, 실시간 3D 애니메이션 같은 기타 인터랙티브 콘텐츠를 제작하기 위한 통합 저작 도구로써,[2] 본 설계과제의 메인 프로그램을 Unity 3D 상에서 C# 언어로 프로그래밍했다.

Unity는 Steam VR 플러그인과 연동 가능하며, 이를 통해 Unity에서 제작한 가상현실 컨텐츠를 VIVE VR로 바로 볼 수 있다는 장점이 있어 가상 드럼을 만드는데 가장 적합한 툴이다. 그리고 Unity의 또 다른 장점은 엔진 자체에 라이트 맵핑이나 물리 엔진 등의 미들웨어를 탑재하고 있다는 점이다. 우리는 이 내장 물리 엔진을 이용해 드럼의 타격 흔들림을 구현했다.

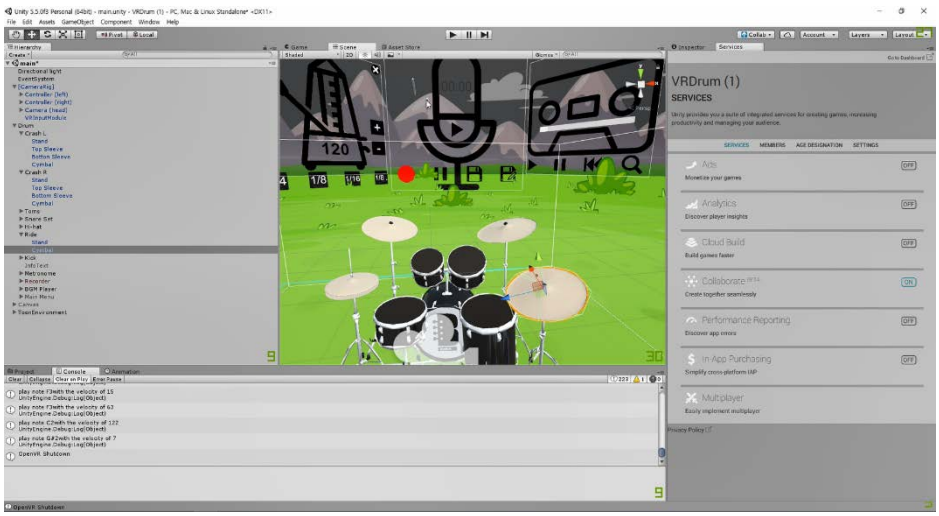

그림 8 - Unity 3D 실행 화면

C. Blender

Unity는 그래픽 툴이 아닌 게임 엔진이자 통합개발환경(Integrated Development Environment, IDE)이기 때문에 3D model 제작에는 적절치 않다. 따라서 외부 프로그램에서 3D model을 제작하여 Unity에 옮겨 사용하여야 하는데, 이를 위해 Blender를 사용했다. Blender는 무료로 제공되는 3D computer graphics software로, 무료임에도 불구하고 좋은 개발환경을 제공해주어 실제 상업용 애니메이션에도 활용된다. 우리는 blender를 통해 무료로 공개되어 있는 드럼의 3D 모델을 가져와 본 설계과제에 적합하게 수정하여 사용했다.

그림 9 – Blender 로고

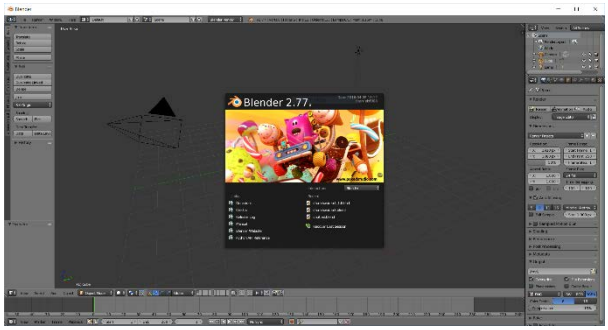

그림 10 – Blender 실행 화면

D. **BFD3**

BFD3는 fxpansion 사에서 제공하는 드럼 가상악기 소프트웨어다. 가상악기란 컴퓨터에서 실제 악기의 소리를 재생할 수 있도록 미리 프리셋을 녹음해 놓은 소프트웨어를 말하는 것으로, 본 설계과제에서는 가상 드럼의 소리를 재생하기 위해 메인 프로그램으로부터 MIDI 신호를 받아 정해진 소리를 재생할 때 사용한다.

그림 11 - BFD3 실행 화면

E. **아두이노와 드럼 페달**

일반 드럼세트에는 2개의 페달이 있다. 그중 왼쪽 페달은 하이햇 심벌즈를 열고 닫기 위해 사용하고, 오른쪽 페달은 베이스 드럼(큰 북)을 타격할 때 사용한다. 가상 드럼에서 두 손의 움직임은 컨트롤러를 통해 추적할 수 있지만, 이 페달을 밟는 두 발의 움직임은 추적할 수 없어 별도의 인식 센서가 필요하다. 그래서 우리는 실제로 사용하는 드럼 페달에 여러 센서를 부착하여 연주자가 페달을 밟는 동작을 인식하도록 했다. 이때 센서의 값을 읽고 PC로 전송하기 위해 아누이노 우노 모듈을 사용했다.

하이햇 심벌즈를 여닫는 왼쪽 페달에는 기울기(tilt) 센서를 부착하여 연주자가 페달을 얼마나 깊이 밟고 있는지 값을 읽는다. 페달이 가장 아래 바닥에 닿으면 심벌이 완전히 닫히고, 페달이 가장 위에 있을 때 심벌도 완전히 열리도록 하여, 그 사이를 균등한 간격으로 나누어 다섯 단계의 소리를 구분했다. 또한 심벌이 빠르게 닫히거나 열릴 때에도 소리가 나도록 프로그래밍했다.

베이스 드럼을 연주하는 오른쪽 페달은 가속도 센서와 피에조 센서를 부착하여 연주자가 어느 시점에 얼마나 빠르게 페달을 밟는지 확인한다. 가속도 센서가 밟는 세기를 측정하여 소리의 크기를 결정하고, 피에조 센서가 피크를 인식할 때 소리를 재생하는 신호를 PC로 전송한다.

아두이노는 오픈소스를 기반으로 한 단일 보드 마이크로컨트롤러로[3], 여기서는 위 세가지 센서로부터 값을 읽어 PC로 전송한다. 통신 방법은 시리얼 통신이며, USB포트를 사용한다. 소스 코드는 다음 아두이노 프로그램 설명에 있다.

6. 메인 프로그램 설명

A. Prerequisite: MIDI 신호 규약

MIDI(musical instrument Digital Interface)는 MIDI Manufacturer Association 과 AMEI 가 제정하고 공표한, 전자 악기의 연주 데이터를 전송하고 공유하기 위한 업계 표준 규격이다. 물리적인 수준의 인터페이스 회로 규격, 연주 데이터를 주고 받을 때 필요한 통신 프로토콜, 파일 형식 등에 대한 표준을 규정하고 있다.[4]

MIDI 로 송수신하는 데이터 규격은 16진수로 0x00VVNNSC 다. 여기서 VV 가 Velocity(음표의 연주 세기), NN 이 Note(음표), S 가 Status, C 가 Channel 을 의미한다. 이 중 Status 가 9이면 Note On 명령, 8이면 Note Off 하라는 명령이다. C 의 경우 0이 1번 채널을 나타내며, 본 가상 드럼은 1번 채널을 쓰고 있다.

B. Class Diagram

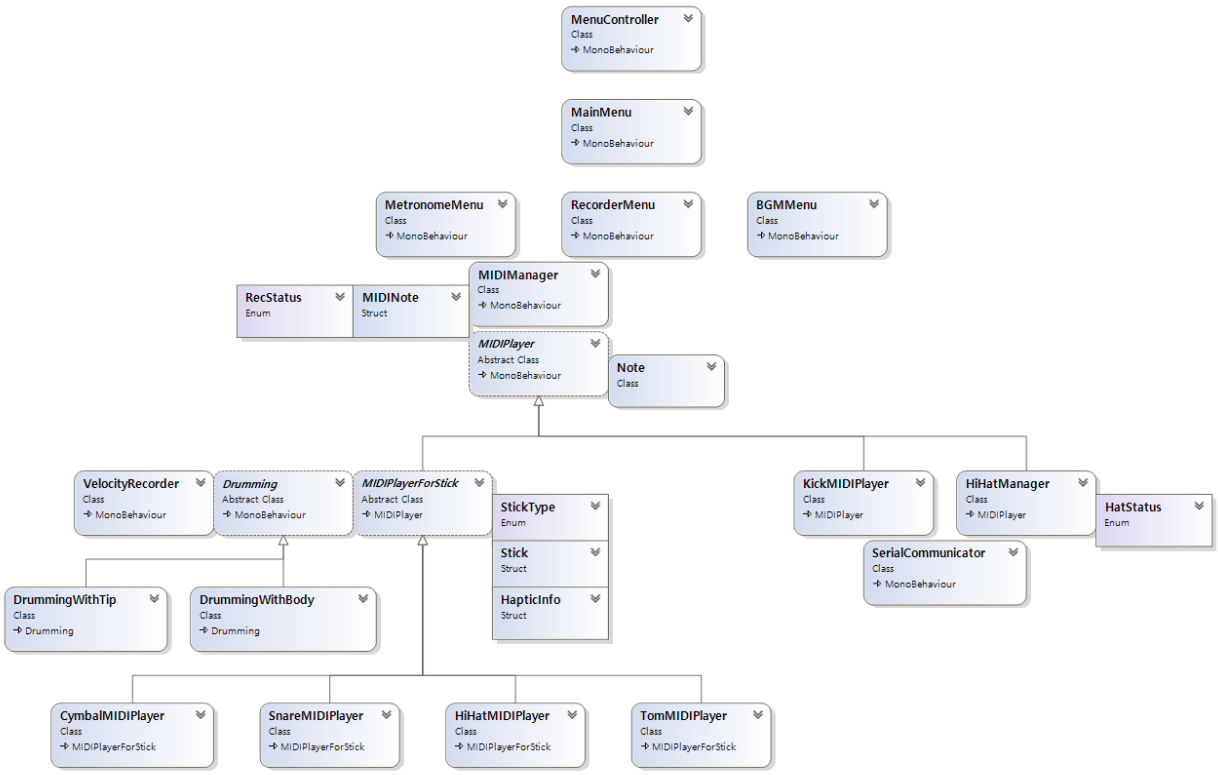

그림 12 – Class Diagram

- RecStatus 와 MIDINote, HatStatus, StickType, Stick, HapticInfo 은 Struct 혹은 Enum 타입으로, 데이터를 편하게 관리하기 위해 정의한 것이므로 생략한다.
- MenuController: Controller 에 Attach 되어 있으며, UI Menu 를 열고 닫는 동작과, 드럼 위치를 조정하는 동작을 Controller 로 할 수 있게 해준다.
- MainMenu: Metronome Menu 와 Recorder Menu, BGM Menu 를 열고 닫는 UI 버튼과 드럼 위치 조정을 시작하는 UI 버튼, 종료 UI 버튼의 기능을 구현하는 스크립트이다.
- MetronomeMenu: 메트로놈의 BPM(빠르기)과 간격(4분음표, 8분음표, 16분음표 및 점8분음표)을 설정하는 UI 버튼과 메트로놈을 켜고 끄는 UI 버튼의 기능을 구현하는 스크립트이다.
- RecorderMenu: 자신의 드럼 연주를 녹음/재생/정지 하는 UI 버튼의 기능을 구현하는 스크립트이다.
- BGMMenu: 배경음악(BGM) 파일을 선택하고 재생하는 기능을 구현하는 스크립트이다.
- MIDIManager: Drum 에 Attach 되어 있으며, 프로그램 시작 시 MIDI 통신 Port 를 열고, 종료 시 닫는다. MIDIPlayer 클래스가 호출할 수 있는 Play 함수가 있어 MIDI 신호를 실제로 송신하는 역할을 한다. 또한, RecorderMenu 의 드럼 연주 녹음 및 재생 기능, MetronomeMenu 의 메트로놈 재생 기능의 내부 구현을 담당한다.
- SerialCommunicator: Serial Port 를 열어 아두이노와 통신한다. 10ms 마다 가속도 센서와 피에조, 틸트 센서의 값을 받아서 Linked List 에 저장하여 KickMIDIPlayer 와 HiHatManager 에 알린다.
- MIDIPlayer: 추상 클래스로, Play 함수가 있어 MIDIManager 에게 재생할 note 의 정보를 보낸다. 또한, 컨트롤러의 속도 혹은 센서의 값을 MIDI velocity 로 변환하는 추상 메소드인 ConvertVelToMIDIVel 을 가지고 있어 자식 클래스에서 구현하도록 한다. 이 클래스의 자식 클래스는 드럼의 각 북과 심벌에 Attach 된다.
  - KickMIDIPlayer: SerialCommunicator 가 받아온 가속도 센서와 피에조 센서의 값을 이용해 Kick 드럼을 연주한다.
  - HiHatManager: SerialCommunicator 가 받아온 틸트 센서의 값을 이용해 Hi-Hat 이 열린 정도를 표현하고 순간적으로 열릴 때 나는 소리, 순간적으로 닫힐 때 나는 소리를 재생한다.
  - MIDIPlayerForStick: 컨트롤러의 속도를 MIDI velocity 로 변환하도록 ConvertVelToMIDIVel 함수를 구현한다. 이 클래스도 추상 클래스이고, MIDI velocity 를 Haptic(진동 피드백) 정보로 변환하는 추상 메소드인 HapticInformation 을 가지고 있어 자식 클래스에서 구현하도록 한다. 또한, Drumming 의 자식 클래스가 호출할 수 있는 PlayWithHaptic 함수는 부모의 Play 함수를 사용하는 동시에 HapticInformation 의 결과를 리턴한다. Stick 의 Tip 으로 연주할 때와 Body 로 연주할 때를 어떻게 구분해서 처리할 것인지에 대한 기능을 가지고 있다.

- CymbalMIDIPlayer: 심벌용 MIDIPlayerForStick 이다. Stick 속도를 MIDI Velocity 값으로 변환할 때 사용하는 상수를 심벌에 적절한 값으로 가지고 있다. 타격 위치를 판단하여 중심부이면 Bell, 중간이면 Bow, 가장자리이면 Edge 에 해당하는 note 를 재생하도록 한다.
- SnareMIDIPlayer: Snare 용 MIDIPlayerForStick 이다. Stick 의 Body 가 Rim 에 닿고 나서 Tip 이 닿았을 경우에는 RimShot 을 재생한다.
- HiHatMIDIPlayer: Hi-Hat 용 MIDIPlayerForStick 이다. 충돌 시 HiHatManager 의 상태를 참조해 Hi-Hat 이 열려 있는지 닫혀 있는지 확인하고 상태에 맞는 note 를 재생한다. Stick 속도를 MIDI Velocity 값으로 변환할 때 사용하는 상수를 Hi-Hat 에 적절한 값(심벌과 같은 값)으로 가지고 있다. 타격 위치가 중심부일 때는 Bell 에 해당하는 note 를 재생한다.
- TomMIDIPlayer: Tom 용 MIDIPlayerForStick 이다. Stick 의 Body 가 Rim 에 닿을 경우에는 RimClick 에 해당하는 note 를 재생한다.

- Drumming: 추상클래스로, Velocity 를 계산하는 추상 메소드인 GetVelocity 를 자식클래스가 구현하도록 한다. 자식클래스인 DrummingWithTip 과 DrummingWithBody 는 Stick 의 Tip 과 Body 에 각각 Attach 된다. MIDIPlayerForStick 을 가지고 있는 북이나 심벌과 충돌했을 때 스틱의 속도를 계산하고 MIDIPlayerForStick 의 PlayWithHaptic 함수를 호출하여 해당 note 를 재생하도록 한다. 또한 PlayWithHaptic 함수가 리턴한 Haptic 정보를 이용해 컨트롤러를 진동시키고, 충돌 애니메이션 효과를 재생한다.
  - DrummingWithTip: Stick 의 Tip 부분의 속도를 매 프레임마다 계산해놓고, 충돌 시에는 최근 10개의 속도 중 '북/심벌 표면에 수직인 방향 성분'이 가장 큰 것을 찾아 해당 방향 성분을 사용한다.
  - DrummingWithBody: Controller 중심의 속도와 각속도, 위치를 매 프레임마다 계산하고 충돌 시에는 해당 값을 이용하여 충돌한 부분의 속도 중 '북/심벌 표면에 수직인 방향 성분'을 구한다.
- VelocityRecorder: Controller 속도를 MIDI Velocity 에 매핑하는 적절한 함수를 찾기 위해 실험용으로 만든 스크립트이다. 완성된 프로그램에서는 사용하지 않는다.

C. 알고리즘

아래 ①~⑨의 모든 스크립트에서 공통적으로 사용되는 함수는 Start(), Update(), OnDestroy(), Mathf.Lerp(a,b,t), Mathf.RoundToInt(a)가 있다. Start() 함수는 프로그램이 시작할 때 호출되는 함수이며, Update() 함수는 매 프레임마다 호출되는 함수이다. 프로그램에 사용된 모든 스크립트의 Update 함수가 끝까지 처리되어야 다음 프레임으로 넘어간다. 그리고 OnDestroy() 함수는 '스크립트가 Attach되어 있는 오브젝트'가 사라질 때(프로그램이 종료될 때를 포함) 호출되는 함수이며, Mathf.Lerp(a,b,t) 함수는

a와 b 사이를 t:1-t 로 내분하는 값을 리턴하는 함수이다. 마지막으로 Mathf.RoundToInt(a)는 a를 정수로 반올림하는 함수이다.

HTC Vive 컨트롤러와 관련된 부분인 MenuController를 제외하고 UI를 구현하기 위한 나머지 부분(MainMenu, BGMMenu, MetronomeMenu, RecorderMenu)은 가상 드럼의 핵심적인 알고리즘이 아니므로 생략한다. 마찬가지로 VelocityRecorder도 사용되지 않으므로 생략한다.

① MenuController

```
//Vive Controller를 위한 선언
private SteamVR_TrackedObject trackedObj;
private SteamVR_Controller.Device controller {
    get { return SteamVR_Controller.Input((int)trackedObj.index); }
}
private const Valve.VR.EVRButtonId gripButton
                                = Valve.VR.EVRButtonId.k_EButton_Grip;
private const Valve.VR.EVRButtonId menuButton
                                = Valve.VR.EVRButtonId.k_EButton_ApplicationMenu;
private const ulong padButton = SteamVR_Controller.ButtonMask.Touchpad;
private const Valve.VR.EVRButtonId padTouch
                                = Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad;

public MainMenu mainMenu;
public MIDIManager midiMgr;

private Wacki.ViveUILaserPointer pointer;

void Start () {
    trackedObj = GetComponent<SteamVR_TrackedObject>();
    pointer = GetComponent<Wacki.ViveUILaserPointer>();
}

void Update () {
    if(trackedObj == null) return;

    //메뉴버튼으로 메뉴와
    if(controller.GetPressDown(menuButton))
        mainMenu.gameObject.SetActive(!mainMenu.gameObject.activeSelf);
    //레이저포인터를 켜고 끔
    if(pointer.enabled != mainMenu.gameObject.activeSelf) {
        pointer.SetPointerActive(mainMenu.gameObject.activeSelf);
        pointer.enabled = mainMenu.gameObject.activeSelf;
    }
```

```
    //MainMenu 에서 드럼 움직이는 기능을 켰을 때
    //(이 때 드럼은 사용자의 헤드셋을 바라보며 따라간다.)
    if(mainMenu.MovingDrum) {
        //드럼과 사용자 간 거리를 패드버튼으로 조절하고
        if(controller.GetPressDown(padButton)) {
            if(controller.GetAxis(padTouch).y < 0.5)
                mainMenu.distance -= 0.2f;
            else
                mainMenu.distance += 0.2f;
        }
        //그립버튼을 누르면 현재 위치로 드럼을 고정시킴
        if(controller.GetPressDown(gripButton))
            mainMenu.StopMovingDrum();
    }
}
```

② **MIDIManager**

```
/*윈도우 내장 winmm.dll 파일에서 MIDI 관련된 함수 Import*/

//Recording 과 Metronome 이용 시 시간을 판단하기 위한 기준점(프로그램 시작 시간)
private DateTime programStartUpTime;

//현재 레코딩 중인지 리플레이 중인지 둘 다 아닌지를 저장
private RecStatus _status;
public RecStatus status { get { return _status; } }

private void Start() {
    /*통신 Port Open*/

    //시작 시각 저장
    programStartUpTime = DateTime.Now;

    _status = RecStatus.stopped;
}

private void PortOpen(string devName) {
    /*연결된 MIDI Device 중에서 원하는 Device 에 연결, 포트 개방*/
}

//MIDIPlayer 가 호출하는 Play 함수
public void Play(MIDINote midinote) {
    switch(_status) {
        case RecStatus.recording:
```

```
            /*records 에 midinote 와 ""현재 시각-시작 시각"" 정보 삽입*/
            break;
        case RecStatus.replaying:
            //리플레이 중에는 연주 불가
            return;
    }
    //새 Thread 에서 MIDI 데이터 전송
    new Thread(() => SendData(midinote)).Start();
}

private void SendData(MIDINote midinote) {
    //드럼은 Note On 명령 시 타격음이 끝까지 재생되므로
    //Note Off 명령은 임의의 시간이 지난 후 해주면 된다.
    /*
    "note 를 velocity 의 세기로 Note On" 메시지를 보냄
    Thread 를 10ms 동안 일시정지
    "note 를 velocity 의 세기로 Note Off" 메시지를 보냄
    */
}

//현재 Metronome 이 켜져있는지 아닌지 저장
private bool metronomeOn;

//Metronome 시작
public void StartMetronome(float bpm) {
    if(metronomeOn) metronomeOn = false;

    //새 Thread 에서 시작
    new Thread(() => {
        //Metronome 소리 간격
        float duration = 60 / bpm;
        //시작 시간
        TimeSpan start = DateTime.Now-programStartUpTime;

        metronomeOn = true;

        //반복
        while(metronomeOn) {
            //현재 시각
            TimeSpan now = DateTime.Now - programStartUpTime;
            if(/*"현재 시각-시작 시각"이 duration 의 배수일 때 */) {
                /*Metronome 소리에 해당하는 note 전송*/
            }
        }
```

```
    }).Start();
}

//Metronome 정지
public void StopMetronome() {
    metronomeOn = false;
}

//레코딩 시작 시간과 끝 시간을 저장
private TimeSpan startTime = new TimeSpan();
private TimeSpan lastTime = new TimeSpan();
public TimeSpan RecordLength { get { return lastTime; } }

//레코딩 시 어떤 음표를 어떤 시간에 연주했는지를 저장
private QueueOfRecords records = new QueueOfRecords();
//리플레이를 위한 백업
private QueueOfRecords records_Backup = new QueueOfRecords();

//레코딩 중이면 몇 분/몇 초 째 기록중인지를,
//리플레이 중이면 마지막 note 까지 얼마나 남았는지를 리턴
public TimeSpan GetTime() {
    switch(_status) {
        case RecStatus.recording:
            return DateTime.Now - programStartUpTime - startTime;
        case RecStatus.replaying:
            return lastTime - ( DateTime.Now - programStartUpTime - startTime );
        default:
            return new TimeSpan();
    }
}

//레코딩 시작
public void StartRecording() {
    if(_status == RecStatus.recording) return;

    //records 를 비우고 시작 시각 설정
    _status = RecStatus.recording;
    startTime = DateTime.Now - programStartUpTime;
    records.Clear();
    records_Backup.Clear();
    lastTime = new TimeSpan();
}

//레코딩 정지
```

```
public void StopRecording() {
    if(_status == RecStatus.recording) {
        _status = RecStatus.stopped;
        records_Backup = records;
    }
}

//리플레이 시작
public void ReplayRecord() {
    if(_status == RecStatus.replaying) return;
    _status = RecStatus.replaying;
    if(metronomeOn) metronomeOn = false;

    //이미 한번 리플레이해서 records 가 비었을 경우
    //backup 에서 가지고 옴
    if(records.Count == 0) records = records_Backup;

    //시작 시각 설정
    startTime = DateTime.Now - programStartUpTime;

    //새 Thread 에서 반복
    new Thread(()=>{
        while(_status==RecStatus.replaying) {
            if(records.Count == 0) break;
            if(/*records.Peek()의 시간보다 "현재 시각-시작 시각"이 클 경우*/) {
                //dequeue 하고 재생
                var item=records.Dequeue();
                new Thread(() => SendData(item.Value)).Start();
            }
        }
        _status = RecStatus.stopped;
    }).Start();
}

//리플레이 정지
public void StopReplaying() {
    if(_status == RecStatus.replaying) {
        _status = RecStatus.stopped;
        records.Clear();
    }
}

private void OnDestroy() {
    //레코딩/리플레이 모두 정지
```

```
    _status = RecStatus.stopped;
    metronomeOn = false;

    //포트 닫기
    midiOutClose(handleMIDI);
}
```

③ **SerialCommunicator**

```
//HiHatManager, KickMIDIPlayer와 상호작용
public HiHatManager hihat;
public KickMIDIPlayer kick;

//Serial 통신 간격은 10ms
private SerialPort sp;
public const float COMM_INTERVAL = 0.010f;

//센서 값을 저장하는 Linked List
private ListOfShort _accel;
public ListOfShort accel { get { return _accel; } }
private ListOfShort _piezo;
public ListOfShort piezo { get { return _piezo; } }
private ListOfShort _tilt;
public ListOfShort tilt  { get { return _tilt;  } }

private void Start() {
    OpenPort();
}

private void Update() {
    if(/*연결 끊김이 감지되면 재연결*/) {
        sp.Close();
        sp.Dispose();
        Start();
    }
}

private void OpenPort() {
    sp = new SerialPort(PortName, Baudrate, Parity.None, 8, StopBits.One);
    //새 Thread에서 연결 반복 시도
    new Thread(() => {
        while(!isOnline) {
            try {
                sp.ReadTimeout = 1000;
```

```
                sp.Open();

                /*데이터 초기화*/

                //값 받아오기 시작
                new Thread(Read).Start();

                break;
            }
            //위에서 에러 발생 시
            catch {
                //에러 메시지 표시하고 반복
                Debug.Log("Please Connect Serial Port");
                continue;
            }
        }
    }).Start();
}

private void Read() {
    /*아두이노에 수신 대기 메시지를 송신*/

    while(isOnline) {
        /*연결 끊기면 멈춤*/

        /*아두이노가 보낸 메시지 한 줄을 받아서 temp 에 대입*/

        //Tab 문자를 기준으로 분리
        string[] line = temp.Split('\t');

        //각 값을 List 에 넣음
        _accel.AddLast(Convert.ToInt16(line[0], 16));
        _piezo.AddLast(Convert.ToInt16(line[1], 16));
        _tilt.AddLast(Convert.ToSByte(line[2], 16));

        //List 가 너무 길면 가장 오래된 값을 지움
        if(_accel.Count > 100) _accel.RemoveFirst();
        if(_piezo.Count > 100) _piezo.RemoveFirst();
        if(_tilt.Count > 100) _tilt.RemoveFirst();

        //KickMIDIPlayer 와 HiHatManager 가 데이터를 처리하도록 함
        new Thread(() => kick.ProcessSignal()).Start();
        new Thread(() => hihat.ProcessData()).Start();
    }
```

```
}

private void OnDestroy() {
    //모든 while loop 정지
    needToReconnect = false;
    isOnline = true;
    isOnline = false;

    //통신 Port 닫기
    sp.Close();
    sp.Dispose();
}
```

④ **MIDIPlayer**

```
//Drumming이 호출하는 Play 함수
protected int Play(string note, float velocity) {
    //MIDI velocity로 변환하여 MIDIManager의 Play 함수 호출
    int midiVel = ConvertVelToMIDIVel(velocity);
    midiMgr.Play(new MIDINote(new Note(note).number, midiVel));
    return midiVel;
}

//velocity(컨트롤러 속도/센서 값 기반)를 MIDI velocity로 변환
//자식 클래스가 구현해야함
protected abstract int ConvertVelToMIDIVel(float velocity);
```

⑤ **KickMIDIPlayer**

```
//SerialCommunicator가 저장한 센서 값을 받아와야함
public SerialCommunicator serialcomm;

//Kick Note
public string hitNote;

//Kick 연주 시 애니메이션 재생을 위한 변수
private Animation move;
private bool needToPlay = false;

//페달을 건드리지 않을 때 가속도 센서 값 저장
private short offset = 0;
private const int OFFSET_DECISION_T = 10;

//피에조 센서 값이 0가 되면 true
```

```
private bool piezoIsZero = true;

//SerialCommunicator가 준비되고 offset이 계산된 후에
//킥의 모습이 나오도록 하기 위한 변수
private bool kickNeedsToBeActive = false;

//피에조 센서값과 MIDI Velocity 매핑을 위한 상수
private const short VEL_SCALE = 380;

protected override void Start() {
    base.Start();

    //처음엔 킥의 모습을 숨김
    kickBody.SetActive(false);
    kickBody = transform.Find("Body").gameObject;
    move = kickBody.transform.Find("Plane").gameObject.GetComponent<Animation>();
}

private void Update() {
    //SerialCommunicator가 준비되고 offset이 계산되면
    if(kickNeedsToBeActive) {
        //킥의 모습이 나옴
        kickBody.SetActive(true);
        kickNeedsToBeActive = false;
    }
    //킥 소리가 재생됐는데 아직 애니메이션이 재생 안됐을 경우
    if(needToPlay) {
        move.Play();
        needToPlay = false;
    }
}

//ProcessSignal 함수가 킥 사운드 재생이 필요할 경우 호출
public void Play(short velocity) {
    //MIDIPlayer의 Play 함수
    Play(hitNote, (float)velocity);
    needToPlay = true;
}

protected override int ConvertVelToMIDIVel(float velocity) {
    //피에조센서 값과 선형 매핑
    return Mathf.RoundToInt(Mathf.Lerp(0, 127, velocity / VEL_SCALE));
}
```

```
//Serial 통신이 끊겼다가 다시 연결되면 SerialCommunicator 가 호출하여
//offset 값을 재설정한다.
public void InitOffset() {
    offset = 0;
}

//센서 값 처리. SerialCommunicator 가 새로운 Thread 에서 호출한다.
public void ProcessSignal() {
    //offset 을 결정하기 위한 데이터가 모일 때까지 기다림
    if(serialcomm.accel.Count < OFFSET_DECISION_T) return;

    //offset 을 결정할 수 있을 때
    if(offset == 0 && serialcomm.accel.Count >= OFFSET_DECISION_T) {
        offset=/*serialcomm.accel 의 값들의 평균*/;
        kickNeedsToBeActive = true;
    }

    //offset 이 결정된 후일 때
    var current = serialcomm.accel.Last.Previous;
    var currentPiezo = serialcomm.piezo.Last.Previous;

    if(/*이전에 피에조 센서 값이 0 이었지만 현재(최근값 직전) 값은 0 보다 크고,
       더 커지고 있으며, 가속도 센서의 값이 현재 피크이거나 피크를 바로 지났을 경우*/)
    {
       piezoIsZero = false;

        //명확히 페달을 아래로 찍어내리는 가속도가 있었는지 판별하기 위해
        //가속도 센서의 값 목록을 시간을 거슬러 올라감
        bool downPeakStart = false;
        var pivot = current.Previous;
        int sum = 0;

        while(true) {
            //가속도 센서 값이 offset 보다 아래로 내려가기 시작한 순간 판단
            if(!downPeakStart && pivot.Value <= offset && pivot.Previous.Value < offset)
                downPeakStart = true;

            //offset 보다 아래로 내려가기 시작한 이후부터 더함
            if(downPeakStart) {
                sum += ( offset - pivot.Value );

                //다시 offset 보다 커지는 순간 더하기를 멈춤
                if(pivot.Value > offset) break;
            }
```

```
            pivot = pivot.Previous;
            if(pivot == null) break;
        }
        //직전에 offset 보다 작은 센서 값이 존재해서 위에서 더해졌다면
        //명확히 페달을 아래로 찍어내렸다고 판단, 피에조 센서 값을 기준으로 소리 재생
        if(sum > 0) Play(currentPiezo.Value);
    }
    else if(/*피에조 센서 값이 0 보다 크지만 작아지고 있을 경우*/) piezoIsZero = false;
    else if(/*피에조 센서 값이 0 이면*/) piezoIsZero = true;
}
```

⑥ HiHatManager

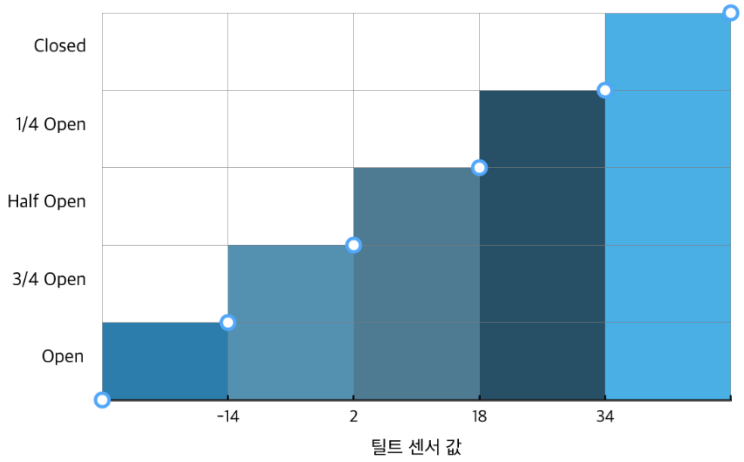

그림 13 – 틸트 센서 값과 Hi-Hat note 매핑

틸트 센서의 값을 5구간으로 나누어 각 구간마다 다른 소리가 나도록 했다. Hi-Hat 오브젝트의 겉모습은 틸트 센서의 값에 따라 선형적으로 열리고 닫히는 것처럼 보인다.

```
//SerialCommunicator 가 저장한 센서 값을 받아와야함
public SerialCommunicator serialcomm;

//순간적으로 여닫을 때 나는 소리에 해당하는 note
public string openingNote;
public string closingNote;

//하이햇 여닫는 구간 설정
public const short TH_OPEN_3_4 = -14;
public const short TH_3_4_HALF = 2;
public const short TH_HALF_1_4 = 18;
public const short TH_1_4_CLOSED = 34;

//열리고 닫히는 소리가 나는 기점
```

```
public const short TH_OPENING = -10;
public const short TH_CLOSING = 24;

//열리고 닫힐 때 센서 값 변화속도와 MIDI velocity 매핑을 위한 상수
private const short VEL_SCALE_CLOSING = 3800;
private const short VEL_SCALE_OPENING = 1900;

//TH_OPENING 보다 작아지는 순간 true, 다시 커지는 순간 false
private bool hatopening = true;
//TH_CLOSING 보다 커지는 순간 true, 다시 작아지는 순간 false
private bool hatclosing = false;

private short statusValue;
//구간에 따라 다른 값
public HatStatus status {
    get {
        if(statusValue < TH_OPEN_3_4)
            return HatStatus.open;
        else if(TH_OPEN_3_4 <= statusValue && statusValue < TH_3_4_HALF)
            return HatStatus.three_quarter;
        else if(TH_3_4_HALF <= statusValue && statusValue < TH_HALF_1_4)
            return HatStatus.half;
        else if(TH_HALF_1_4 <= statusValue && statusValue < TH_1_4_CLOSED)
            return HatStatus.one_quarter;
        else
            return HatStatus.closed;
    }
}

private void Update() {
    /*센서 값에 따라 Hi-Hat 의 Top Cymbal 위치 위아래로 이동*/
}

//ProcessData 함수가 호출함
public void Play(float velocity) {
    Play(( /*닫히는중인지 열리는 중인지 판단*/ ) ? closingNote : openingNote, velocity);
}

protected override int ConvertVelToMIDIVel(float velocity) {
    //센서값 변화 속도(velocity)와 MIDI velocity 선형 매핑
    if(/*닫히는 소리면*/)
        return Mathf.RoundToInt(Mathf.Lerp(0, 127, velocity / VEL_SCALE_CLOSING));
    else
        return Mathf.RoundToInt(Mathf.Lerp(0, 127, (-velocity) / VEL_SCALE_OPENING));
```

```
}

public void ProcessData() {
    if(serialcomm.tilt.Count <= 1) return;

    //틸트 센서의 현재 값과 "(현재값-이전값)/10ms" 받아옴
    var value = serialcomm.tilt.Last.Value;
    var velocity = ( serialcomm.tilt.Last.Value - serialcomm.tilt.Last.Previous.Value )
                    / SerialCommunicator.COMM_INTERVAL;

    statusValue=value;

    //hatopening/hatclosing 이 false 였는데 threshold 를 넘은 경우
    //hatopening/hatclosing 을 true 로 만들고 여닫는 소리 재생
    if(!hatopening && value < TH_OPEN_3_4) {
        hatopening = true;
        Play(velocity);
    }
    else if(!hatclosing && TH_1_4_CLOSED <= value) {
        hatclosing = true;
        Play(velocity);
    }

    //threshold 안쪽으로 다시 돌아온 경우 hatopening/hatclosing 을 false 로
    if(TH_OPENING <= value && hatopening)
        hatopening = false;
    if(value < TH_CLOSING && hatclosing)
        hatclosing = false;
}
```

⑦ MIDIPlayerForStick

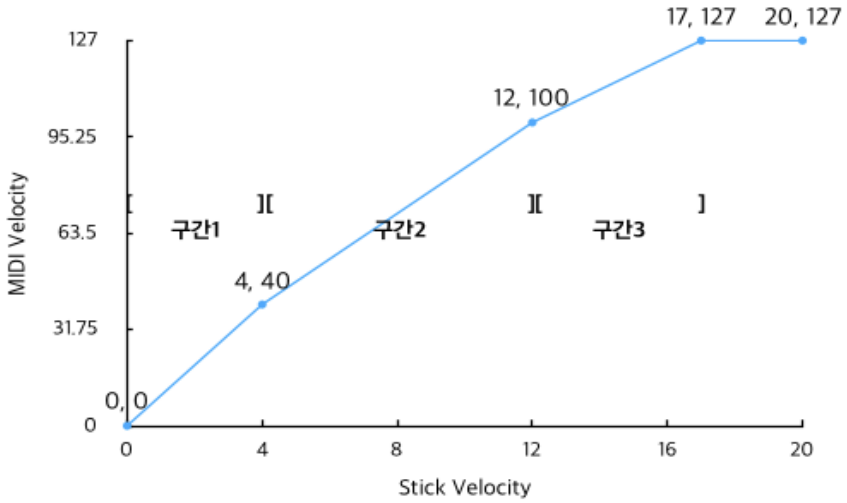

그림 14 – Stick의 Velocity와 MIDI velocity 매핑

MIDI Velocity 는 0부터 127까지의 값을 가지는데, 본 프로젝트에서 사용한 가상악기의 경우 0~40, 40~100, 100~127의 세 구간에서 각각 서로 다른 특징을 가진 소리가 재생되는 것을 확인했다. 그 중 무난한 소리가 나는 40~100 구간을 측정하기 위해 MIDI Velocity 가 40, 50, 60, 70, 80, 90, 100 인 가상악기 스네어 소리를 드러머에게 들려준 후, 실제 스틱과 컨트롤러를 한 손에 동시에 쥔 채로 들려준 가상악기 소리와 비슷한 소리가 나도록 실제 스네어를 연주하게 했다. 그 결과 컨트롤러로 측정된 스틱의 피크 속도가 위 그래프(구간 2)와 같이 선형적으로 나타났다. 또한, 드러머가 최대한 세게 연주할 때의 스틱 속도가 17로 나타났다. 이에 따라 구간1과 구간3은 자동으로 결정되었다. 구간1과 구간3도 선형으로 매핑한 이유는 소리나 주법 자체가 구간2와 크게 다르지는 않고, 곡선으로 매핑하지 않아도 충분히 실제 연주에 근사할 것이라 생각했기 때문이다. 심벌의 경우도 마찬가지로 구간을 나누지만, 구간을 나누는 값이 다르다.

```
//Velocity-MIDIVelocity 매핑 상수
protected readonly int TH_MIDI1 = 40;
protected readonly int TH_MIDI2 = 100;
protected readonly int MAX_MIDI = 127;
protected readonly int TH_VEL1 = 4;
protected readonly int TH_VEL2 = 12;
protected readonly int MAX_VEL = 17;

//생성자를 이용하여 자식 클래스에서 위 상수를 다른 값으로 교체 가능
public MIDIPlayerForStick():base() { }
public MIDIPlayerForStick(int _th_midi1, int _th_midi2,
                          int _th_vel1 , int _th_vel2 , int _max_vel):base() {
```

```
    TH_MIDI1 = _th_midi1;
    TH_MIDI2 = _th_midi2;
    TH_VEL1 = _th_vel1;
    TH_VEL2 = _th_vel2;
    MAX_VEL = _max_vel;
}

/*Stick 의 Body 와 Tip 중 어느 것이 먼저 충돌했는지 저장해두는 기능들*/

//북 혹은 심벌의 충돌면 반지름 저장
private float _extents;
protected float extents {
    get { return _extents; }
}

protected override void Start() {
    base.Start();

    //북 혹은 심벌의 충돌면 반지름 저장
    var collider = GetComponent<Collider>();
    _extents = collider.bounds.size.x + collider.bounds.size.z;
    _extents /= 2;
}

//MIDIPlayer 의 Play 함수로 재생하고 컨트롤러 햅틱 반응(진동) 정보 리턴
protected HapticInfo PlayWithHaptic(string note, float velocity) {
    return HapticInformation(Play(note,velocity));
}

//Drumming 의 자식클래스가 호출할 함수.
//스틱의 어느 부위로 어느 위치에 어느 속도로 충돌했는지를 받아 적절한 note 를 재생
//자식클래스에서 구현해야 함
public abstract HapticInfo PlayWithHaptic(Stick stick, Collider col,
                                          Vector3 hitpoint, float velocity);

//MIDI Velocity 를 햅틱 반응 정보로 변환
private HapticInfo HapticInformation(int midiVel) {
    float hapticTime, hapticStrength;

    //진동 시간
    if(midiVel < TH_MIDI1)      hapticTime = 0.05f; //구간 1
    else if(midiVel < TH_MIDI2)  hapticTime = 0.10f; //구간 2
    else                        hapticTime = 0.15f; //구간 3
```

```
    //진동 강도를 midiVel 에 따라 선형 매핑
    hapticStrength = Mathf.Lerp(0.1f, 1.0f, midiVel / MAX_MIDI);
    return new HapticInfo(hapticTime, hapticStrength);
}

//컨트롤러 속도를 MIDI Velocity 로 변환
protected override int ConvertVelToMIDIVel(float velocity) {
    //Map Controller Velocity to MIDI Velocity
    int midiVel=0;

    if(velocity < TH_VEL1)
        /*구간 1 내에서 선형 매핑*/
    else if(velocity < TH_VEL2)
        /*구간 2 내에서 선형 매핑*/
    else if(velocity < MAX_VEL)
        /*구간 3 내에서 선형 매핑*/
    else
        midiVel = MAX_MIDI;

    return midiVel;
}
```

⑧ CymbalMIDIPlayer

```
//타격 위치에 따라 다른 note
public string bellNote;
public string bowNote;
public string edgeNote;

//Bell 부분의 반지름과 Edge 부분의 두께
private const float SIZE_BELL=0.25f;
private const float SIZE_EDGE = 0.05f;

//심벌 류에 맞는 Velocity-MIDIVelocity 매핑 상수를 설정
public CymbalMIDIPlayer():base(40,100,1,6,11) {}

public override HapticInfo PlayWithHaptic(Stick stick, Collider col,
                                          Vector3 hitpoint, float velocity) {
    /*Stick 의 body 가 이미 닿았는데 tip 이 또 닿은 경우, 혹은 그 반대의 경우를 제외*/

    //타격 위치에 따라 note 재생
    if(hitpoint.magnitude <= SIZE_BELL)
        return PlayWithHaptic(bellNote, velocity);
    else if(hitpoint.magnitude > extents - SIZE_EDGE)
        return PlayWithHaptic(edgeNote, velocity);
```

```
    else
        return PlayWithHaptic(bowNote, velocity);
}
```

⑨ **SnareMIDIPlayer**

```
//일반 Hit note와 Rim Shot note 구분
public string hitNote;
public string rimShotNote;

public override HapticInfo PlayWithHaptic(Stick stick, Collider col, Vector3 hitpoint,
    float velocity) {
    /*Body로 Rim을 친 것으로 확인되면 Keep해두고
      Body로 Rim이 아닌 다른 곳을 쳤으면 무시*/

    if(/*Tip으로 쳤는데 그 전에 Keep해둔 Body-Rim 충돌이 있으면*/)
        return PlayWithHaptic(rimShotNote, velocity);
    else
        return PlayWithHaptic(hitNote, velocity);
}
```

⑩ **HiHatMIDIPlayer**

```
//5개의 구간마다 Tip/Body로 연주할 때의 note를 각각 구분하고
//Bell 연주에 해당하는 note를 구분함
public string Note_Tip;     public string Note_Body;
public string _3_4_Tip;     public string _3_4_Body;
public string _Half_Tip;    public string _Half_Body;
public string _1_4_Tip;     public string _1_4_Body;
public string Closed_Tip;   public string Closed_Body;
public string bellNote;

//Bell의 반지름
private const float SIZE_BELL = 0.2f;

//HiHatManager 에서 열고 닫힌 정도를 받아와야 함
private HiHatManager manager;

//심벌 류에 맞는 Velocity-MIDIVelocity 매핑 상수 설정
public HiHatMIDIPlayer():base(40,100,1,6,11) { }

public override HapticInfo PlayWithHaptic(Stick stick, Collider col,
                                          Vector3 hitpoint, float velocity) {
    /*Stick의 body가 이미 닿았는데 tip이 또 닿은 경우, 혹은 그 반대의 경우를 제외*/
```

```
    //Bell 을 타격했으면 Bell note 재생
    if(hitpoint.magnitude <= SIZE_BELL)
        return PlayWithHaptic(bellNote, velocity);

    //stick.type 과 하이햇 열고 닫힌 정도(manager.status)에 따라 note 재생
    return PlayWithHaptic(notes[stick.type.ToString() + manager.status], velocity);
}
```

⑪ TomMIDIPlayer

```
//Hit note 와 Rim Click 에 해당하는 note 구분
public string hitNote;
public string rimNote;

//피격 시 애니메이션 재생
private Animation move;

public override HapticInfo PlayWithHaptic(Stick stick, Collider col, Vector3 hitpoint,
    float velocity) {
    if(/*Rim 이 아닌 곳을 Tip 으로 타격 시*/) {
        //애니메이션 재생하고 해당 note 재생
        if(move!=null) move.Play();
        return PlayWithHaptic(hitNote, velocity);
    }
    else if(/*Rim 을 Body 로 타격 시*/) {
        //애니메이션 재생하고 해당 note 재생
        if(move != null) move.Play();
        return PlayWithHaptic(rimNote, velocity);
    }
    else
        return new HapticInfo(0, 0);
}
```

⑫ Drumming

```
//타격 애니메이션 효과
public GameObject hitEffect;

//스틱의 속도를 계산하는 함수. 자식클래스에서 구현해야 함.
public abstract float GetVelocity(Collision col);

//충돌하여 겹치기 시작할 때 호출되는 함수
void OnCollisionEnter(Collision col) {
    /*충돌한 object 의 MIDIPlayer 에 충돌했다고 알림
     (이 클래스가 attach 되어있는 object 가 body 인지 tip 인지도 알림)*/
```

```
    /*스틱이 아래 방향으로 이동 중에 충돌이 났을 때에만 아래를 처리*/

    //GetVelocity 함수로 Stick 속도를 얻은 후
    float velocity = GetVelocity(col);

    //MIDIPlayer의 PlayWithHaptic 함수로 소리 재생 및 Haptic 정보를 얻음
    var haptic = player.PlayWithHaptic(id, col.collider, col.contacts[0].point, velocity);

    //소리를 재생하면 안되는 경우를 제외하고
    if(haptic != new HapticInfo(0, 0)) {
        //컨트롤러 진동
        StartHapticVibration(controller, haptic.time, haptic.strength);

        /*hitEffect 애니메이션을 타격 위치에서 velocity에 따라 0.3초 동안 동안 재생*/

    }
}

//충돌이 끝나 따로 떨어지게 되었을 때 호출되는 함수
void OnCollisionExit(Collision col) {
    //충돌한 object의 MIDIPlayer에 충돌이 끝났다고 알림
}
```

⑬ **DrummingWithTip**

```
//속도를 저장하는 Linked List
private LinkedList<Vector3> velocities;

//이전 프레임에서 object의 위치
private Vector3 previousPos;

private void Update() {
    //"(현재 위치-이전 위치)/프레임 간 시간 간격" 저장
    velocities.AddFirst((gameObject.transform.position - previousPos)/Time.deltaTime);

    //10개가 넘어가면 지움
    if(velocities.Count > 10) velocities.RemoveLast();

    //이전 위치 업데이트
    previousPos = gameObject.transform.position;
}

public override float GetVelocity(Collision col) {
```

```
    //보관하고 있는 velocity 들의 북(혹은 심벌) 표면에 수직인 방향 성분을 계산
    //그 성분들 중 가장 최근의 peak 를 구하여 return
    float peakVel = float.MaxValue;

    foreach(Vector3 vel in velocities) {
        float normalVel = Vector3.Dot(vel, col.transform.up.normalized);

        if(peakVel >= normalVel)
            peakVel = normalVel;
        else
            break;
    }
    return Mathf.Abs(peakVel);
}
```

⑭ **DrummingWithBody**

```
//매 프레임마다 업데이트 되는 컨트롤러 중심의 위치, 속도, 각속도
private Kinetics kinController;

private void FixedUpdate() {
    //매 프레임마다 업데이트
    kinController = new Kinetics(controller.transform.pos, controller.velocity,
    controller.angularVelocity);
}

public override float GetVelocity(Collision col) {
    //컨트롤러의 중심에서 타격 위치 사이의 변위
    var displacement = col.contacts[0].point - kinController.pos;

    //컨트롤러 중심의 속도와 가속도를 이용해 타격 위치의 속도를 구하여 리턴
    var velocity = kinController.vel + Vector3.Cross(kinController.angVel, displacement);
    return Mathf.Abs(2*Vector3.Dot(velocity, col.contacts[0].normal));
}
```

7. 아두이노 프로그램 설명

다음은 아두이노에 사용된 소스코드 전문이다. 아두이노 코드는 전용 IDE 프로그램에서 작성 후 USB 포트를 통해 아두이노 보드의 플래시 메모리에 저장하여 사용한다.

```
#include <AcceleroMMA7361.h> //가속도 센서를 위한 라이브러리
#include <MMA_7455.h> //tilt 센서를 위한 라이브러리
```

```
#include <Wire.h>

#define N 5 //tilt 센서 값의 Moving Average 계산에 이용할 데이터 개수

AcceleroMMA7361 accelSensor;
MMA_7455 tiltSensor = MMA_7455();

char tiltZ; //tilt 센서 Z-axis 값
int acZ; //가속도 센서 Z-axis 값
int piezo; //피에조 센서 값

void setup()
{
  Serial.begin(9600);
  delay(100);
  pinMode(A3,INPUT);
  tiltSensor.initSensitivity(2); //tilt tpstj 감도 설정
  accelSensor.begin(13, 12, 11, 10, A0, A1, A2); //가속도 센서를 연결한 pin 번호 설정
  accelSensor.setARefVoltage(5); //Reference Voltage 설정
  accelSensor.setSensitivity(LOW); //가속도 센서 감도 설정
  accelSensor.calibrate(); //가속도 센서 calibration
  Serial.print("\n");
}

//Moving Average 계산
//새 값이 들어올 때마다 backup 에 저장해두고
//N 개의 값을 유지하면서 그 값들의 평균을 리턴
char MovingAverage(char a) {
    static char backup[N]={0};
    static int pivot=0;
    static bool first=true;
    int i=0;
    int sum=0;
    backup[pivot++]=a;
    if(pivot==N) {
        if(first) first=false;
        pivot=0;
    }
    for(i=0; i<((first)?pivot:N); i++) {
        sum+=backup[i];
    }
    return (char)(sum/((first)?pivot:N));
}
```

```
void loop()
{
    char buf[3];
    //송신 버퍼가 넘치지 않도록 관리
    if(Serial.availableForWrite()<10) Serial.flush();

    //센서 값 읽어옴
    acZ = accelSensor.getZRaw(); //가속도 센서
    piezo=analogRead(A3); //피에조 센서
    tiltZ = MovingAverage((char)tiltSensor.readAxis('z')); //틸트 센서

    //해당 값의 HEX 표현 문자열을 Serial Port 로 송신
    //값 끼리 Tab 문자로 구분하고 끝에 줄바꿈
    if(acZ<0) acZ=0;
    Serial.print(acZ,HEX);
    Serial.print('\t');

    if(piezo>255) piezo=255;
    Serial.print(piezo,HEX);
    Serial.print('\t');

    sprintf(buf, "%02x", (unsigned char)tiltZ);
    Serial.print(buf);
    Serial.print('\n');

    //10ms 간격으로 송신
    delay(10);
 }
```

8. 타격 애니메이션

가상현실에서는 연주자가 현실과의 이질감을 느끼지 않도록 타격 시 드럼이 흔들리는 모습을 시각적으로 구현해야한다. 이를 구현하는 방법으로는 물리 엔진(physics engine)과 애니메이션(Animation)이 있다. 물리 엔진은 타격 시 충돌 세기에 따라 흔들림이 크게 변하는 경우에 사용하며, 반대로 애니메이션은 흔들림이 적어 타격 시 충돌 세기와 거의 무관한 경우에 사용한다. 특히 애니메이션은 미리 저장된 모션을 재생시키는 것이기 때문에 컴퓨터의 연산 과정이 물리 엔진보다 적어 latency를 최소화 시킬 수 있다.

A. 물리 엔진(physics engine)

충돌에 Unity 물리 엔진을 사용하기 위해서는 먼저, 'Rigidbody'와 'Collider'를 설정해 주어야한다.

- Rigidbody: Object가 물리 법칙 안에서 동작하게 된다. 따라서 힘과 토크를 받아 현실적으로 오

브젝트를 움직일 수 있게 된다.

- Collider: Object간의 충돌 감지를 위해 object를 감싸는 경계를 설정하는 것이다. Object의 모양이나 충돌 특징에 따라 Box collider, Sphere collider, Mesh collider 중에서 한 가지를 선택하여 사용한다. 복잡한 model인 경우 부분별로 나누어 적용하는 경우도 있다.

이후 실제 드럼과 같이 각 부분을 서로 연결해야 한다. 이 연결을 Unity에서는 'Joint'라 부른다. Joint에는 여러 종류가 있으면 이번 프로젝트에서는 Fixed Joint와 Hinge Joint를 주로 사용하였다.

- Fixed Joint: 명칭 그대로 두 Object를 고정하는 Joint이다. Tom과 Kick을 연결하거나 세분화된 Stand를 연결하는데 사용한다. 한 rigidbody에 fixed joint를 add하고 connected body에 다른 rigidbody를 선택하여 두 rigidbodies를 연결할 수 있다. Break Force와 Break Torque를 설정하여 joint가 해제되는 조건을 부여할 수 있으나 이번 프로젝트에서는 사용하지 않았다.

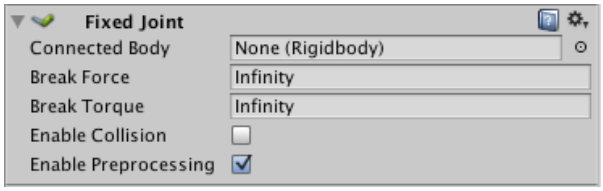

그림 15 – Fixed Joint

- Hinge Joint: Hinge Joint는 여닫이 문의 경첩 부분과 같다. 그렇기 때문에 보통 문에 적합하지만 설정을 통해 사슬이나 진자와 같은 model에도 사용할 수 있다. 한 rigidbody에 hinge joint를 add하고 connected body에 다른 rigidbody를 선택하여 두 rigidbodies를 연결할 수 있다. Joint가 움직이는 방향을 Aixs component를 통해 설정할 수 있다. 또한 추가적인 설정을 통해 joint에 spring과 같은 탄성을 가진 운동을 하도록 설정할 수 있고, Limits를 설정하여 joint의 운동 범위를 제한할 수 있다. 이번 프로젝트에서는 실제 드럼에서의 탄성과 연결부분에 의해 각도가 제한되는 것을 확인하고 Hinge Joint의 Spring과 Limits를 사용하였다. Crash cymbal의 흔들림을 구현하기 위해 Hinge Joint를 사용하였다.

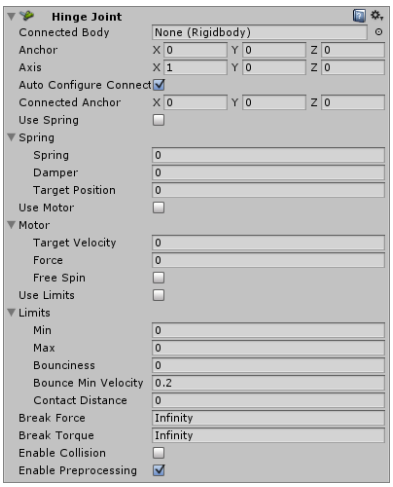

그림 16 – Hinge Joint

B. 애니메이션

애니메이션은 Object 를 선택 한 후에 rotation 값을 조절하여 설정한다. 이후 Collision 이 되었을 때 애니메이션이 재생되도록 하는 script 를 attach 한다. 메인 프로그램에서 애니메이션이 사용된 object 는 High Tom 과 Low Tom 이다.

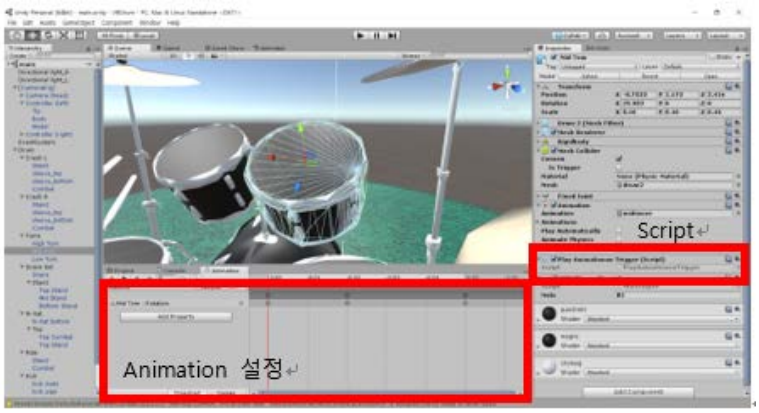

그림 17 – 애니메이션 설정

9. 주요 오브젝트

A. Crash L

그림 18 – Crash L

하위 object	Stand, Cymbal, Top Sleeve, Bottom Sleeve
Attached Scripts	Cymbal MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였으며, Hinge Joint를 사용하여 흔들림을 구현하였다. 또한 Smoothing을 하여 실물과 좀 더 비슷하도록 object 외형을 수정하였다.

B. Crash R

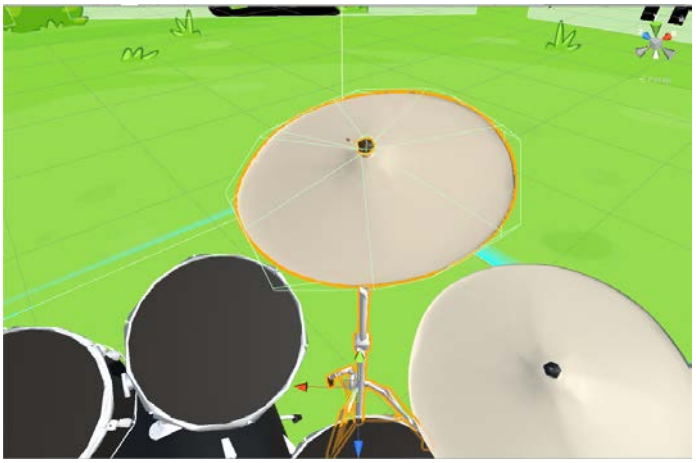

그림 19 – Crash R

하위 object	Stand, Cymbal, Top Sleeve, Bottom Sleeve
Attached Scripts	Cymbal MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였으며, Hinge Joint를 사용하여 흔들림을 구현하였다. 또한 Smoothing을 하여 실물과 좀 더 비슷하도록 object 외형을 수정하였다.

C. High Tom

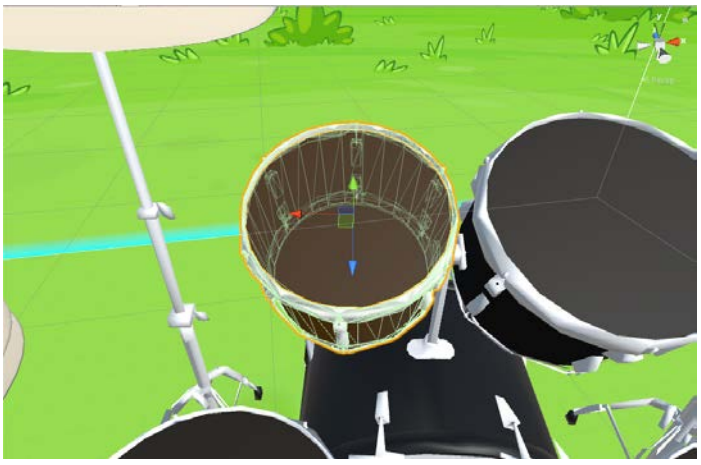

그림 20 – High Tom

하위 object	Plane, Rim
Attached Scripts	Tom MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였으며, Fixed Joint를 사용하여 Kick과 연결하였다. 또한 애니메이션을 사용하여 흔들림을 구현하였다. Tom을 Rim과 Plane를 구별하여 림샷을 구현하였다.

D. **Mid Tom**

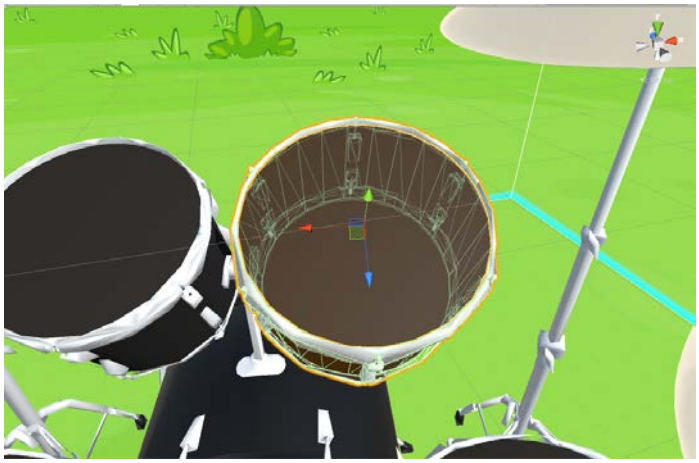

그림 21 – Mid Tom

하위 object	Plane, Rim
Attached Scripts	Tom MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였으며, Fixed Joint를 사용하여 Kick과 연결하였다. 또한 애니메이션을 사용하여 흔들림을 구현하였다. Tom을 Rim과 Plane를 구별하여 림샷을 구현하였다.

E. **Low Tom**

그림 22 – Low Tom

하위 object	Plane, Rim
Attached Scripts	Tom MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였다. Tom을 Rim과 Plane를 구별하여 림샷을 구현하였다.

F. Hi-Hat

그림 23 – Hi-Hat

하위 object	Top(Top Cymbal, Top Stand), Bottom(bottom crash, stand)
Attached Scripts	Hi Hat MIDI Player

페달을 밟는 정도에 따라 심벌이 움직이는 것을 구현하기 위해 Top과 Bottom으로 나누었다. Top Cymbal에 Mesh Collider를 사용하여 충돌 판정을 하였다.

G. Snare Set

그림 24 – Snare Set

하위 object	Snare(Plane, Rim), Stand(Top stand, Mid Stand, Bottom Stand)
Attached Scripts	Snare MIDI Player

Model이 복잡하여 전체를 하나의 Collider로 설정할 수 없으며, snare의 높이 조절을 위해 Stand를 여러 단계로 분리하였다. Snare를 Rim과 Plane를 구별하여 림샷을 구현하였다.

H. **Kick**

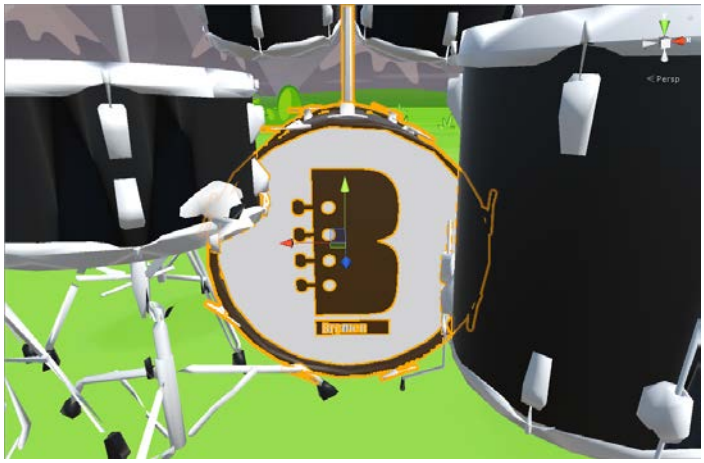

그림 25 - Kick

하위 object	Pipe, Body(Plane(Logo), Rim)
Attached Scripts	Kick MIDI Player

Pipe는 High, Mid Tom과 Fixed Joint로 고정되었다. Body는 Plane과 Rim으로 나누어 페달을 밟았을 때 Plane이 흔들리는 애니메이션을 구현하였다.

I. **Ride**

그림 26 - Ride

하위 object	Stand, Cymbal
Attached Scripts	Cymbal MIDI Player

Mesh Collider를 사용하여 충돌 판정을 하였으며, Hinge Joint를 사용하여 흔들림을 구현하였다.

J. Controller(Stick)

그림 27 - Stick

하위 object	Tip, Body, Model
Attached Scripts	Drumming With Tip, Drumming With Body

실제 드럼을 치는 경우 Stick의 body보다는 tip으로 타격을 한다. 따라서 Tip과 Body를 분리하여 좀 더 정확한 충돌 판정이 가능하도록 하였다. Tip은 Sphere Collider, Body는 Box Collider를 사용하였다. Mesh Collider는 또다른 Mesh Collider와 충돌 판정이 불가능 하기 때문에 Body는 Box Collider를 사용하였다.

K. 추가기능 UI

- Main Menu

그림 28 – 메인 메뉴

Controller에서 System menu 을 누르면 UI가 나타나도록 설정했다. Menu UI를 추가하여 부가 기능들을 사용하고 관리하기 편리하도록 했다. 왼쪽 위부터 순서대로 Metronome, Recorder, BGM Player UI를 여는 버튼이 있고, 자체적으로 드럼 이동과 프로그램 Off 하는 버튼이 있다. 드럼 이동을 하고 controller의 grip 버튼을 누르면 그 상태로 드럼의 위치가 고정된다. Off 버튼을 누르면 프로그램이 종료된다.

- Metronome

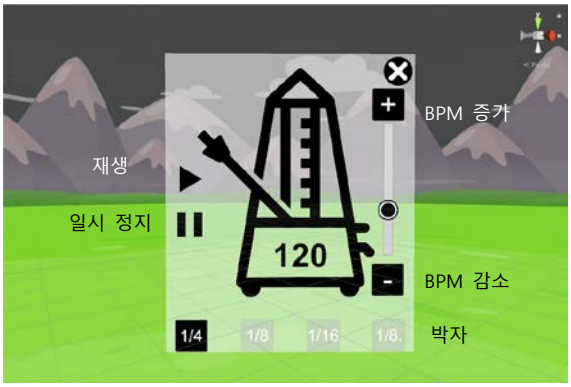

그림 29 – 메트로놈

왼쪽에 있는 재생과 일시 정지 버튼으로 기능을 활성화 또는 비활성화 시킬 수 있다. 오른쪽에 있는 +, - 버튼으로는 bpm을 조절할 수 있다. 또한 두 버튼 사이에 있는 스크롤로도 조절 할 수 있다. 아래 있는 4개의 버튼으로 metronome의 박자를 변경할 수 있다. 기본적으로 1/4박자로 설정되어 있으며 추가적으로 1/8, 1/6, 점 1/16 박자가 있다. 오른쪽 위 X버튼으로 창을 닫을 수 있다.

- Recorder

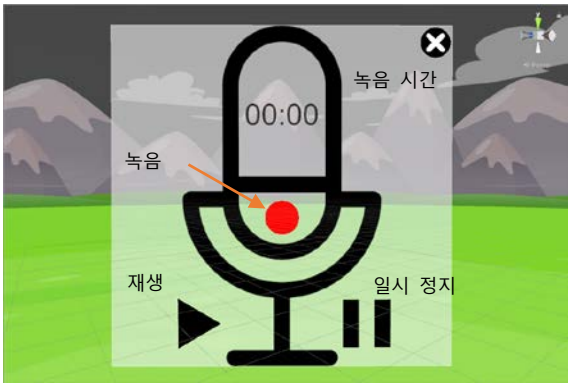

그림 30 – 녹음 기능

빨간 record 버튼을 누르면 시간이 올라가면서 녹음이 된다. 오른쪽에 있는 일시정지 버튼으로 녹음을 중지 시키고, 왼쪽에 있는 재생 버튼으로 녹음한 것을 들을 수 있다. 오른쪽 위 X버튼으로 창을 닫을 수 있다.

- BGM Player

그림 31 – 배경음악 재생

아래에 4개의 버튼이 있으며 왼쪽부터 재생, 일시정지, 처음으로, 불러오기 기능을 한다. 중간에 2개의 버튼은 한번 재생인지, 반복 재생인지를 선택한다. 음원을 불러오면 파일의 이름이 표시된다. 스크롤로 재생 위치를 임의로 선택할 수 있다. 프로그램 특성상 WAV 형식의 파일만 재생이 가능하다. 오른쪽 위 X버튼으로 창을 닫을 수 있다.

10. 결과 및 보완해야 할 점

완성된 가상현실 드럼은 움직임과 소리 재생에 지연시간이 거의 없으며, 컨트롤러와 페달을 이용한 타격 모두 대체로 인식이 잘된다. 하지만 가끔 타격 인식이 되지 않는 경우가 있었다. 컨트롤러를 이용해 연주하는 경우 간헐적으로 심벌 타격 인식이 안 될 때가 있는데, 이는 프레임 업데이트 사이에 스틱이 심벌을 지나치기 때문으로 추측된다. 충돌 판정이 프레임 단위로 업데이트되기 때문에 두 프레임 사이에 스틱이 빠르게 지나가면 충돌이 일어나지 않는 것이다. VIVE는 초당 90프레임, 즉 두 프레임 사이의 간격이 약 11.1ms이므로 이 시간 사이에 스틱과 심벌의 충돌을 감지할 방법이 필요하다.

한편 페달 인식의 경우 오른쪽 페달을 약하게 밟을 때 간헐적으로 소리가 나지 않는데, 이는 오른쪽 페달 구조의 영향이 크다. 현재는 피에조 센서가 상대적으로 면적이 작고, 페달 아랫면에 센서를 누를 수 있게 튀어나온 부분이 좁아 이런 문제가 발생하게 된다. 이 문제는 인터넷에서 구매할 수 있는 페달 기성품으로는 해결하기 힘들기 때문에 3D 프린터 등을 이용한 페달 자체 개발이 필요하다.

추가로 공개 시연에서 받은 피드백 중에는 실제 타격의 느낌이 있도록 컨트롤러로 때릴 수 있는 고무판이 있으면 좋겠다는 의견이 있었다. 하지만 고무판을 두고 때릴 시에는 컨트롤러가 손상될 수 있고, 처음 제작 목표와 부합하지 않기 때문에 진동 반응을 좀 더 강하게 하는 방법으로 타격감을 확보하면 좋을 것이다. 이외에는 타격 부분별로 소리가 정확하게 재생되므로, 사업화 가능성에서 언급했던 문제와 타격 인식 문제만 해결한다면 어플리케이션을 마켓에 등록할 수 있을 것이다.

11. 참고 자료 및 사진 출처

- 참고 자료

1) DAW : https://ko.wikipedia.org/wiki/디지털_오디오_워크스테이션

2) Unity3D : https://ko.wikipedia.org/wiki/%EC%9C%A0%EB%8B%88%ED%8B%B0_(%EA%B2%8C%EC%9E%84_%EC%97%94%EC%A7%84)

3) 아두이노 : https://ko.wikipedia.org/wiki/%EC%95%84%EB%91%90%EC%9D%B4%EB%85%B8

4) MIDI : https://namu.wiki/w/MIDI

- 사진 출처

그림 1-(b) : https://namu.wiki/w/

그림 1-(c) : https://www.linio.com.mx/p/worlde-easypad-12-mini-usb-porta-til-12-drum-pad-midi-controlador-ylc4qd

그림 2 : https://creativedrumming.wordpress.com/2013/04/07/knowing-your-drum-set-part-i/

그림 3 : http://audioboxed.blogspot.com/2016/04/bagian-bagian-stik-drum.html

그림 4, 그림 5-(c) : :http://blog.cakewalk.com/ad2-sounds-real/

그림 5-(a) : http://acehmusician.org/sejarah-lahirnya-hi-hat/

그림 5-(b) : http://www.ebay.com/gds/Ride-Cymbal-Buying-Guide-/10000000177330374/g.html

그림 7 : https://namu.wiki/w/HTC%20Vive

그림 9 : https://opentutorials.org/course/2221