2020년 1학기

졸업논문

제목 : 센서 융합 상보 필터를 이용한 게임 컨트롤러 개발

Team Visible

강선민(2017103772)

김태연(2015104007)

정홍구(2014104045)

담당교수 : 서덕영교수님

목 차

제 1장 : 서론

1.1 논문 요약

센서를 통해 측정한 데이터에는 원치않은 정보인 노이즈가 포함되어 실제값과 오차를 보인다. 정확한 측정결과를 얻기 위해 필터를 이용하여 노이즈를 줄일 수 있다. 노이즈를 줄이는 과정에서 필터 계수에 따라 정확도의 증가와 처리 속도 감소가 발생해 이 두가지 요소의 절충이 필요하다. 이번 연구를 통해 산술평균 필터와 상보필터를 제작하여 가속도 센서와 자이로 센서가 내장된 MPU-6050 모듈에 적용하고 동작 속도와의 상관관계를 분석했다. 그리고 MPU-6050 모듈로 조종하는 유니티 게임을 제작하고 게임 플레이에 최적화된 필터를 설계함으로써 제작물의 성능을 평가했다.

1.2 연구의 배경

'게임 개발 배경'

IT 기술은 급속도로 발전하고 있지만 사회적 약자들을 위한 기술은 상대적으로 더디게 발전하고 있다. 예시로 기존의 게임들은 게임 화면을 통한 시각적 정보가 있어야만 플레이가 가능하기 때문에 기술 소외계층인 시각장애인들은 게임을 함께 할 수 없었다. 이를 인지하고 이번 연구를 통해 시각적 정보 없이도 할 수 있는 게임과 그에 맞는 컨트롤러를 개발하며 더 많은 사용자들이 함께 이용할 수 있도록 하드웨어와 소프트웨어를 만들었다. 2인이 참여하는 이 게임은 두 플레이어가 각각 시각과 촉각 중 하나에만 의존하며 상보적으로 목표를 달성하는 게임이다. 따라서 비장애인과 장애인이 함께 즐길 수 있다.

'유니티 선택 배경'

개발자에게 편리한 인터페이스와 다양한 에셋등을 가진 유니티를 이용해서 게임 환경을 간편하게 구현할 수 있었다. 카메라 시점의 조절과 움직임을 구현하는 C# 스크립트 개발 그리고 물리엔진을 통한 3D 게임을 구현하여 완성도를 높일 수 있었다.

'아두이노 선택 배경'

유니티 게임에 맞는 컨트롤러 제작을 위해서 다양한 오픈소스가 존재하고 많은 연산 속도가 빠른 아두이노 보드를 선택했다. 게임 플레이어의 기울기 값만 정하기 위해 아두이노 보드 중 간단하고 크기가 비교적 작은 UNO 보드 선택으로 력소모와 컨트롤러 크기를 줄였다.

‘센서 선택 배경’
아두이노에 연결되는 센서의 정확도가 높을 수록 가격이 높고, 저가의 센서의 경우 발생하는 오차를 필터를 통해 소프트웨어적으로 보정이 필요하다. 이번 연구에 적합한 컨트롤러를 제작하기 위해 저가의 센서를 선택했고, 게임 최적화를 위한 필터 성능에 대한 실험과 이에대한 분석과정에 연구의 높은 비중을 할당했다.

1.3 연구의 목적 및 방법

〔연구 목적〕
‘사용자와 게임 상호작용 구현과 필터 디자인’
센서를 통해 수집한 값에서 raw(원시) 값과 노이즈 성분을 분리하려면 필터가 필요하다. 필터마다 정확도와 연산에 의한 지연 시간의 대립이 존재하므로 개발 목적에 맞는 필터를 고안해야 한다. 상체의 움직임으로 조작하는 이 게임에 최적화된 필터의 종류와 계수를 실험을 통해 선택할 것이다. 게임 플레이에 있어 센서의 값이 아두이노에서 실시간으로 처리되므로 연산으로 발생하는 지연시간과 정확도의 최적화가 필요하다. 만일 정확도 향상을 위해 필터계수를 조정하면 연산량의 증가로 지연되어 게임과 플레이어의 움직임에 차이가 발생하고, 반대로 지연을 줄이기 위해 연산 양을 줄이면 정확도가 떨어질 것이다.

‘ 전공 이론 적용 및 확인’

- 확률 및 랜덤변수 :
필터를 통과한 가공된 수많은 데이터들의 분포를 분석하기 위해 확률적으로 접근을 할 것이다. 매우 불규칙한 noise 성분들은 서로 독립이고, 같은 확률 분포를 가지며 유한한 기댓값과 표준 편차를 가지므로 데이터의 수 N이 적당히 클 때 Central limit theorem에 의해 정규분포를 따른다.

〔Central limit theorem〕
확률변수 $X_1, X_2, \ldots, X_n$ 들이 서로 독립적이고, 같은 확률 분포를 가지고, 그 확률 분포의 기댓값 μ와 표준편차 σ가 유한하다면, 평균 $S_n = (X_1 + \cdots + X_n)/n$ 의 분포는 기댓값 μ, 표준편차 $\sigma/\sqrt{n}$ 인 정규분포 N($\mu, \sigma^2/n$)에 분포수렴한다.
따라서, $\sqrt{n}\left(\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) - \mu\right) \xrightarrow{d} \mathcal{N}(0, \sigma^2)$가 된다.
정확한 필터일 수록 표준편차가 작아 평균값에서 뾰족한 PDF를 보일것이고 이를 통해 필터의 성능을 평가한다.

- 디지털 신호 처리:
통신 시스템의 성능 평가지수인 SNR(signal to noise ratio)를 필터를 통과한 데이터의 그래프를 통해 확인할 수 있다. SNR이 클 수록 노이즈에 저항성을 가지므로 우수한 필터이다.

$SNR = \frac{P_s}{P_n}$, P_s와 P_n은 각각 신호의 전력, 노이즈의 전력이다. 이번 연구에서 SNR의 개념을 이용해 센서에 연결된 오실로스코프를 통해 측정한 값에서 신호의 크기에서 노이즈가 차지하는 비율로 설계한 센서의 성능을 평가할 것이다.

- 자동제어
실시간으로 생성되는 데이터에서 오차를 줄이기 위해 이전 값을 이용하는 피드백의 원리를 이용하는 필터를 이번 연구에서 제안한다. 실시간 움직임으로 얻은 값의 처리에 있어 이전 값을 이용하여 경향성을 파악하여 데이터가 노이즈에 영향을 덜 받으며 매끄럽게 측정될 수 있다.

〔그림 1.1 입력 출력 피드백〕

MPU-6050내에서 측정된 가속도 센서와 자이로센서 값을 이용하여 처리과정을 거친 후 그 결과를 처리과정의 입력단으로 넣는다. 그리고 입력된 값을 다시 처리하여 출력하는 방식으로 더욱 정밀하고 안정적인 측정값을 얻을 수 있다.

〔 연구 방법 〕
‘ 사용 센서 분석’
이번 연구에서 사용할 IMU 센서인 MPU- 6050에 대해 내장된 기능을 분석하여 정확도 향상을 위한 방법을 연구한다. 그리고 유니티와 연동을 위해 각도를 표현하는 좌표계를 설정한다. 그리고 측정된 센서값의 비교를 위해 측정값을 오일러 각으로 변환한다.
‘ 필터의 특성 연구’
이번 연구에서 사용될 필터들을 저주파 및 고주파 동작에서의 적용을통해 raw 값

을 기준으로 각 필터의 특성을 분석한다. 또한 필터 계수와 필터의 성능의 상관관계를 실험을 통해 분석한다.

‘ 게임에 최적화된 하드웨어 및 소프트웨어 개발’
유니티로 제작한 게임과 아두이노 컨트롤러와의 통신을 구현하고 위 실험결과를 통해 구한 상관관계로 게임내 원활한 동작을 위한 적합한 필터를 설계한다.

제 2장 가속도 센서 및 자이로 센서

2.1 MPU-6050의 특징

센서 사양	Gyroscope(MPU6050)	Accelerometer(MPU6050)
Interface	I2C interface	I2C interface
Startup	100 ms	100 ms
Max range	±2000 d/s	±16G
Min range	±250 d/s	±2G
Resolution(Max)	16.4 LSB/(degree/s)	2048 LSB/(degree/s)
Resolution(Min)	131 LSB/(degree/s)	16384 LSB/(degree/s)
Update rate	4 ~ 8000 hz	4 ~ 1000 hz

〔그림 2.1 MPU 6050 Data Sheet〕

〔그림2.1〕은 MPU 6050의 데이터시트 중 일부를 발췌한 것이다. 여기서 MPU6050은 I2C 의 interface를 사용한다는 것을 알 수 있다.
데이터 시트에서 자이로 센서의 Update rate는 4~8000 Hz의 높은 주파수로도 측정할 수 있으나, 가속도계의 Update rate는 4~1000 Hz로 상대적으로 낮은 주파수에서만 동작한다. 따라서 데이터 시트를 통해 가속도 센서는 낮은 주파수에서 강점을 가지고, 각속도계는 높은 주파수에서 강점을 가질 것으로 예상된다. 이를 4장과 5장의 실험을 통해 검증 할 것이다.
이번 연구에 사용되는 MPU-6050 (GY-521)의 경우 센서의 영점을 잡아주는 지자계, 기압계 등의 보조 센서를 가지고 있지 않는 모듈로 내장된 센서들만의 연산을 통해 값을 보정한다.

2.2 가속도 센서

2.2.1 가속도 센서의 원리

가속도 센서는 물체가 힘을 받아서 물체의 속도가 변하면 관성에 의해 센서 내부의 박막이나 긴 막대가 진동하거나 휘어지면서 압전소자나 가변저항등을 통하여 전압이 변하고 이 변화가 출력되는 원리로 가속도를 측정한다. 그리고 가속도 센서에 외력이 가해지지 않는 경우에서는 중력 가속도만 측정되어진다. 따라서 물체에 외력이 없는 상태일 경우에는 중력 가속도성분을 통하여 기울어진 각을 알 수 있다. 가속도 센서의 X축 기울기 각 는 다음 식 (2.1)에 의해 구해진다.

$$\Phi_x = \tan^{-1}\left(\frac{a_{ex}}{a_{ez}}\right)$$

Φ_x : x축 기울기 a_{ex} : 가속도 센서의 x축 가속도 a_{ez} : 가속도 센서의 z축 가속도 (식 2.1)

가속도 센서의 Y축 기울기 각 는 다음 식 (2.2)에 의해 구해진다.

$$\theta = \tan^{-1}\left(\frac{a_{ey}}{a_{ez}}\right)$$

θ : y축 기울기 a_{ey} : 가속도 센서의 y축 가속도 a_{ez} : 가속도 센서의 z축 가속도 (식 2.2)

2.2.2 가속도 센서의 특징

가속도 센서는 압전 소자로 부터 각도를 계산하는 과정에 있어 속도가 빠른 고주파 움직임에 더욱 큰 노이즈를 보이므로 움직임을 측정함에 있어 가속도 센서만을 사용하기에 부적합하다. 이는 이번 연구에서 사용되는 GY-521 센서의 특성으로, 데이터 시트에서 확인할 수 있다.

2.3 자이로 센서

2.3.1 자이로 센서의 원리

MPU-6050 모듈에는 3축 자이로 센서가 내장되어 있다. MPU-6050은 진동 MEMS 자이로센서로서 센서의 내부에 진동자가 진동하고 있다. 센서 본체가 회전할때 진동자에 발생하는 코리올리힘이 또 다른 진동을 발생시킨다. 발생된진동을 통하여 각속도를 출력하게 된다

코리올리 힘은 회전하는 계에서 느껴지는 관성력으로 그 힘은 식 (2.3)과 같다.

$$F_{Coriolis} = 2m(v \times \Omega)$$

m : 질량 $\vec{v}$: 회전계에서의 속도 $\vec{\Omega}$: 회전계에서의 각속도 (식 2.3)

자이로 센서에서 출력된 각속도를 마이크로프로세서와 같은 연산 처리 장치에 의해 적분하면 기울기 각도를 얻을 수 있다.

$$\theta = \int \vec{w} dt$$

θ : 기울기 각도 $\vec{w}$: 각속도 t : 샘플링 시간 (식 2.4)

2.3.2 자이로 센서의 특징

자이로센서는 각속도의 적분을 이용하여 각도를 측정하기 때문에 시간이 지날수록 오차들이 누적되어 오류가 점점 더 커지는 문제가 발생한다. 이러한 문제를 scale factor error 라고 하고 자이로 센서의 드리프트 현상이라 부른다. 이러한 취약점으로 인해 자이로센서를 바로 이용할 수 없고 가속도계의 측정값을 융합해야 정확한 측정이 가능하다.

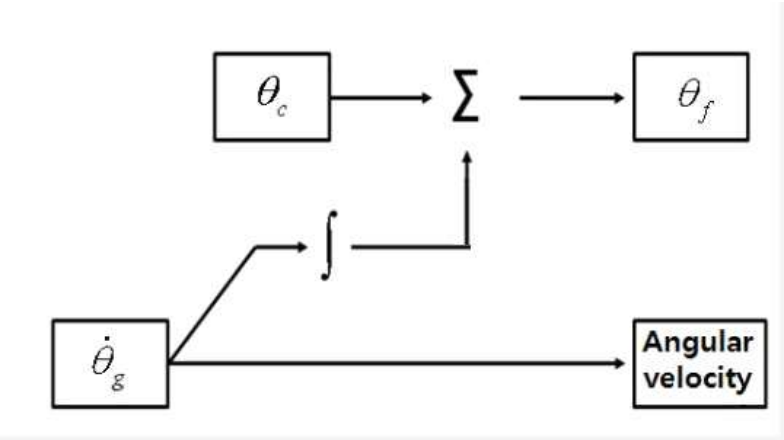
제 3장 제안하는 방법 : 센서 융합 상보 필터 적용

3.1 상보 필터의 정의

상보(相補, complementary)란 서로 모자란 부분을 보충 함을 뜻한다. 상보필터는 MPU-6050에서 측정한 가속도 값과 자이로 값 중 하나만 사용한 것이 아닌 두가지를 모두 이용한 필터이다. 앞서 언급했듯이, 가속도 센서는 이동과 진동에 취약하고, 자이로 센서는 초기 기준 값 결정에 취약하고 드리프트 현상이 존재한다. 각각의 센서로 회전한 각도를 구할 수 있지만 각자의 단점으로 인해 한 가지을 단독으로 사용하지않는다.

따라서 각 단점을 극복하고 장점만을 이용하기 위해 고안된 것이 상보필터이다. 상보필터는 자이로 센서의 고주파 영역에서의 장점과 가속도 센서의 저주파 영역에서의 장점을 융합한 필터이다.

3.2 상보 필터의 보정 과정



〔그림 3.1 상보필터 Diagram〕

상보필터의 각도는 다음과 같은 식으로 결정된다.

$$\theta_{complementary} = \alpha \cdot (\theta_{previous} + \theta_{gyro} \cdot (small\ value)) + (1-\alpha) \cdot \theta_{acc} \quad \text{(식 3.1)}$$

위의 α 는 각속도로 계산한 각도의 가중치를 결정하고, $(1-\alpha)$는 가속도계로 계산한 각도값의 가중치를 결정하게 된다.

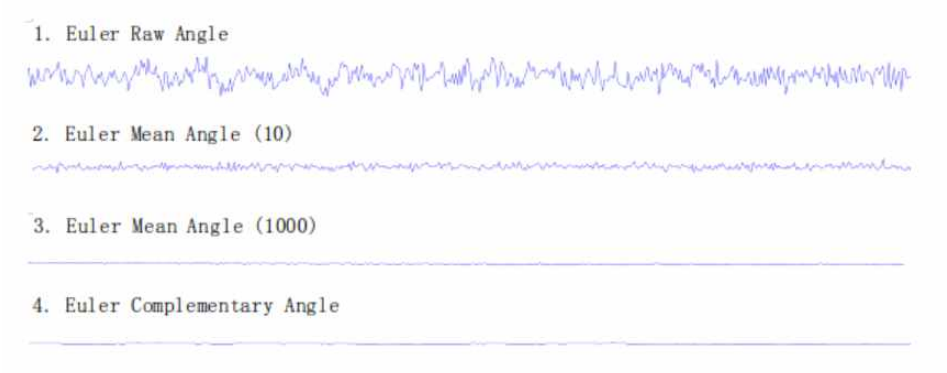
이때 가중치 뒤에 오는 자이로에 의해 유도되는 각에 작은 값을 곱하는 이유는 연구에 사용할 모델에서는 각속도에 의해 유도된 각도보다 가속도에 의해 유도된 각도에 더 큰 비중울 두기 위함이다. 만약 가속도계가 아닌 각속도에 의해 영향을 더 받는 경우인 고주파 움직임의 경우 필터에서 각속도와 가속도의 위치를 바꾸거나, α 값을 변화시키면서 더 나은 필터계수를 찾는것이 중요하다.

또한 α 값의 크기에 따라 우리가 설계한 상보필터의 rising time이 결정된다. 높은 값의 α 의 경우 새롭게 측정되는 각도에 의한 영향이 작기때문에 변화가 느리지만 그만큼 노이즈가 작다. 반대로 작은 α 의 경우 θ_{acc}에 의한 각도 변화량이 크기 때문에 빠른 변화를 정확히 측정 할 수 있는 대신에 노이즈가 커진다.

이러한 α 크기에 따른 노이즈는 이후 4, 5장의 실험에서 확인해보았다.

제 4장 저주파 데이터 특성 측정

센서를 고정한하고 노이즈에 의한 값의 변화를 아두이노의 serial plotter로 오일러 각을 측정한다.

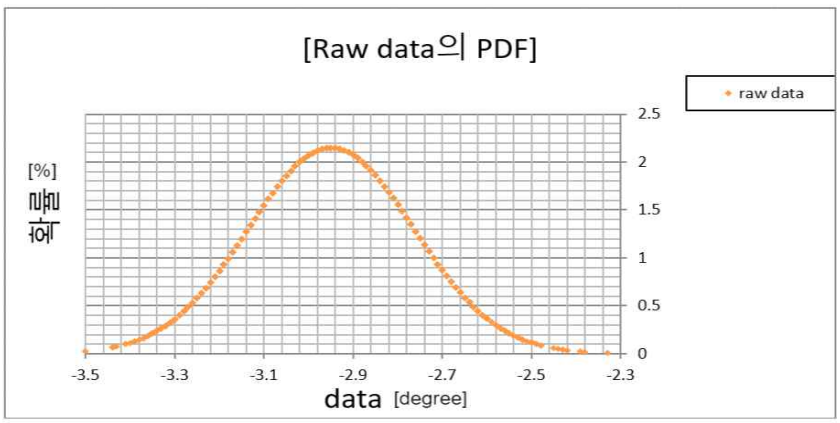


〔그림 4.1 시간에 따른 신호의 노이즈 측정〕

시간에 따라 노이즈가 포함된 측정된 신호의 크기를 4가지 조건으로 관찰했다. 〔그림 4.1〕에서 Euler Mean Angle은 산술평균 필터의 결과이며 괄호안의 숫자는 필터 계수(N)이다.

4.1 Raw 데이터 분석

필터 없이 측정된 데이터를 엑셀에 기록하고 평균과 표준편차를 계산해 정규분포를 그렸다.

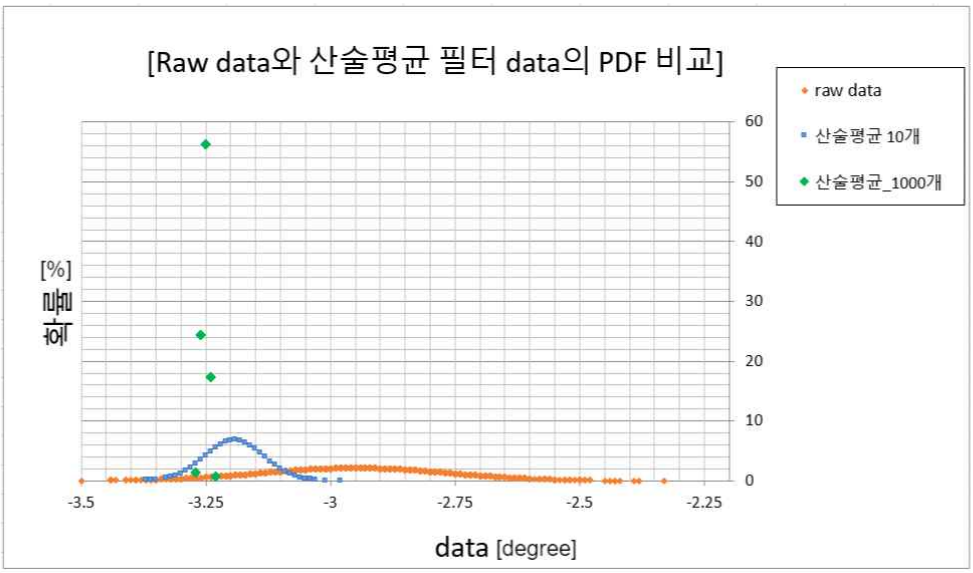


〔그림 4.2 Raw Data의 PDF〕

Raw 데이터는 -2.94의 평균과 0.185의 표준편차를 가지며 〔그림 4.1〕에서 Euler Raw Angle에 나타난 노이즈 성분이 〔그림 4.2〕의 정규분포에서 큰 표준편차로 표현되는 것을 해석할 수 있다. 아두이노의 데이터를 측정하는 Serial Oscilloscope를 이용하여 측정한 raw 데이터의 sample rate는 358 sample /sec 이다.

4.2 산술 평균 필터 분석

위에서 측정한 raw 데이터를 기준으로 산술 평균 필터의 계수에 따른 측정된 데이터의 분포를 필터계수(N)에 따라 분석했다.



〔그림 4.3 Raw Data와 Arithmetic Mean Filter Data PDF〕

'N = 10 일때'

-2.94의 평균값과 0.185의 표준편차를 가지며 최대 확률이 8%정도를 기록했다. 이 때 sample rate는 21 sample/sec이다.

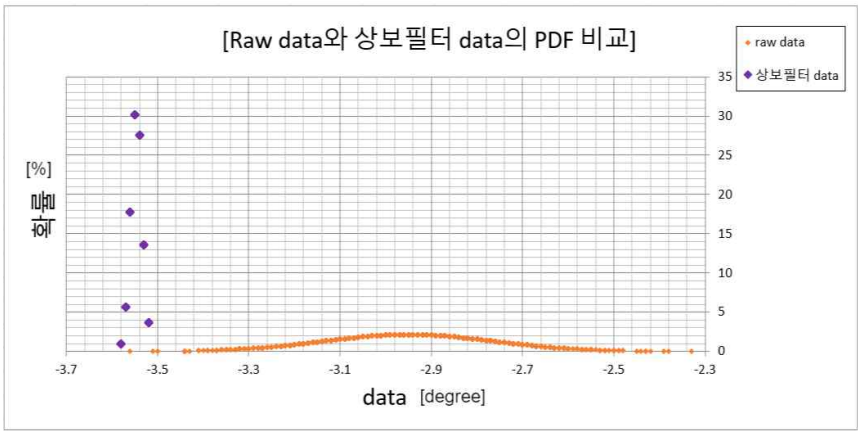
'N = 1000 일때'

-3.19의 평균값과 0.058의 표준편차를 가지며 최대 확률이 54%정도를 기록했다. 이 때 sample rate는 1 sample/sec이다.

이를 통해 필터계수 (N)이 증가할수록 정확도가 증가하나 sample rate가 감소하여 속도의 저하가 발생함을 알 수 있다.

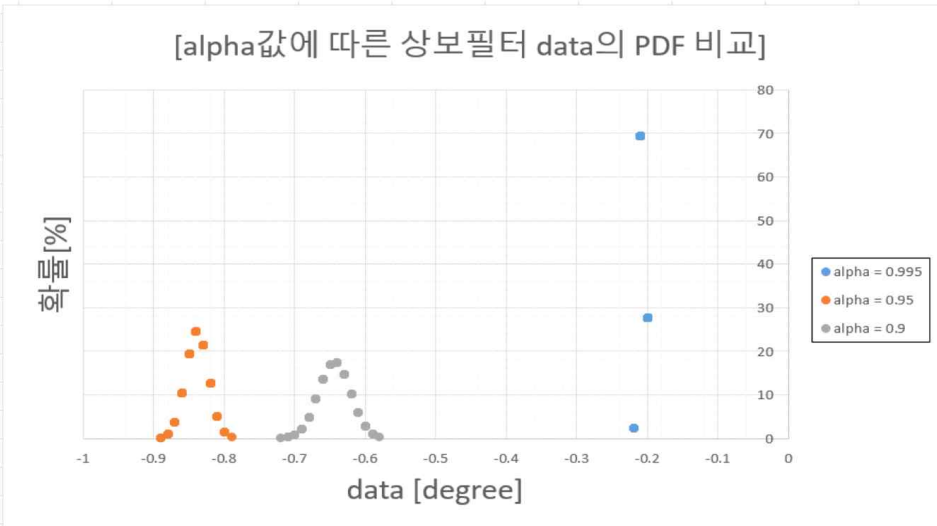
4.3 상보필터 분석

산술 평균 필터와 마찬가지로 raw 값을 기준으로 PDF와 sample rate를 비교했다.



〔그림 4.4 Raw Data와 Complementary Filter Data PDF〕

이 경우-3.54의 평균값과 0.012의 표준편차를 가지며 최대 확률이 31%정도를 기록했다. 이 때 sample rate는 335 sample/sec이다.

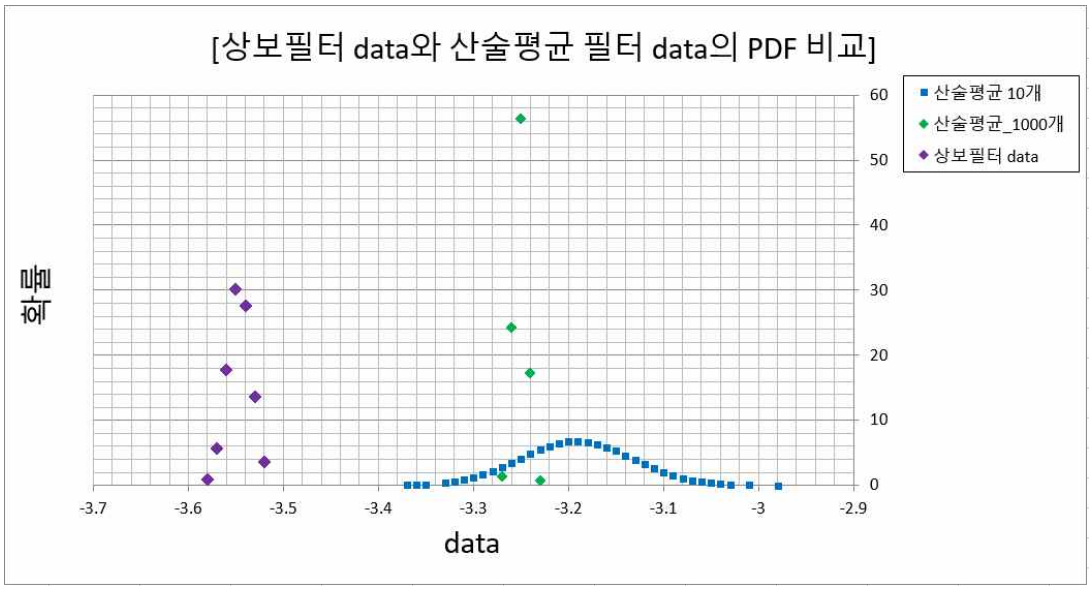


〔그림 4.5 α 에 따른 Complementary Filter Data PDF〕

상보필터의 계수인 α 를 0.995, 0.95, 0.9 세가지로 정확도를 측정했다. 그 결과 α 값이 높을 수록 산술필터의 계수 N의 변화와 같이 향상된 정확도를 가진다는 것을 알게되었다. 산술필터의 계수 N과 다른 점은 α 값은 연산시 현재값에 대한 가중치 계수로 연산 횟수와 무관하여 세가지 α 값 모두 sample rate가 동일하므로 속도의 변화는 없다.

4.4 상보필터와 산술평균 필터의 비교

정적 상태에서 최적으로 동작하는 필터를 찾기 위해 앞서 실험한 두 필터를 비교했다.



〔그림 4.6 Complementary Filter & Mean Arithmetic Filter data PDF〕

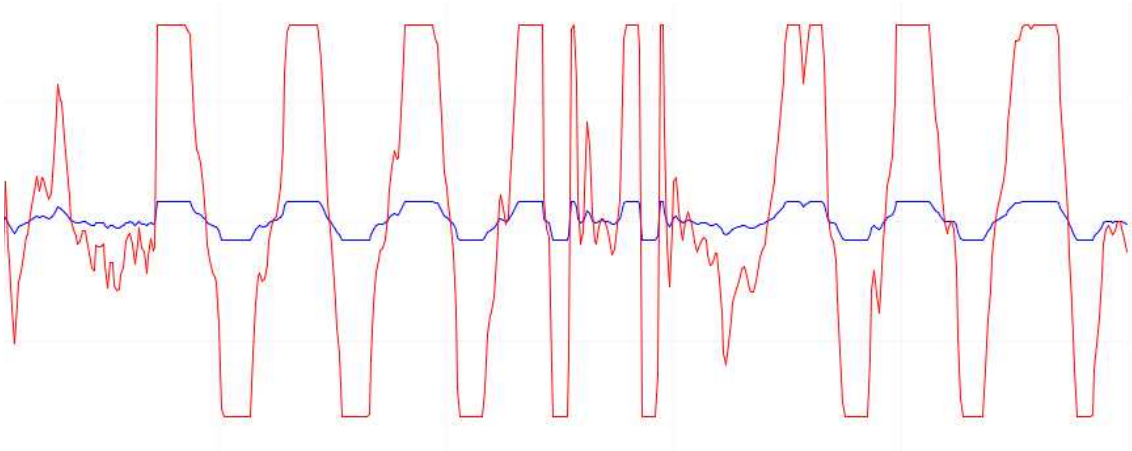
정확도와 sample rate를 고려하여 최적의 필터를 선택하기 위해 상보필터와 산술평균 필터를 비교했다.

필터 종류	표준편차	sampling rate [sample/sec]
상보 필터 (α = 0.995)	0.016	335
상보 필터 (α = 0.95)	4.93	335
상보필터 (α = 0.90)	6.90	335
산술 필터 (N=10)	0.185	21
산술 필터 (N=1000)	0.058	1

'실험 결과 ':
정확도가 제일 높은 N=1000인 산술 필터는 sample rate가 매우 낮다. 따라서 α 가 0.995 인 상보필터가 저주파에서 정확도와 속도의 최적의 균형을 이룰 수 있다는 것을 알 수 있다.

제 5장 고주파 데이터 특성 측정

5.1 산술 평균 필터 분석

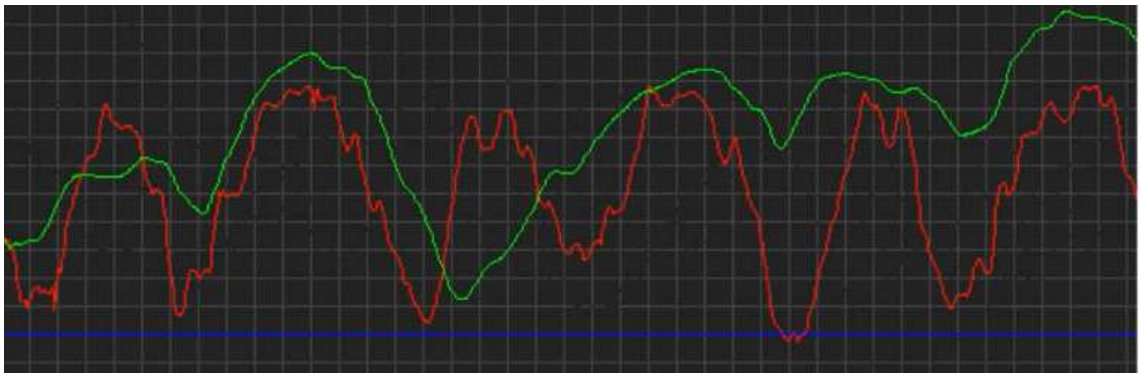


〔그림 5.1 Raw Data와 Arithmetic Mean Data의 비교〕

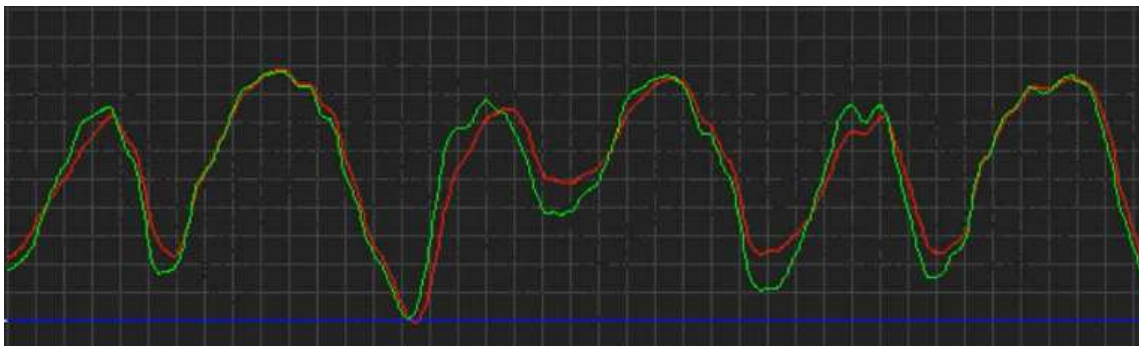
〔그림 5.1은 산술 평균 필터를 이용하여 고주파 동작에서의 분석결과이다. 필터계수 N = 1000인 경우 출력하는 것은 당장 앞의 저주파 분석만으로도 예측이 가능해 N = 10 인 경우를 이용하였다. 실험 결과 고주파에서는 실제 움직임을 따라가지 못하는 모습을 보였으며, 실제값보다 낮게 측정되었으며 큰 오차를 보였다.

위 결과의 이유는 결과값을 더해 출력하는 과정에 있어 값을 계산하는데 있어 큰 각도의 값과 작은 각도의 값이 더해져 평균을 구하기 때문이라 결론을 내렸다. 또한 기울여서 각도가 감소하는 경우 이전에 측정한 큰 값이 누적되어 계산되기 때문에 값이 감소하는 순간에도 완만한 경사를 보여 높은 주파수의 움직임을 제한하는 Low pass filtering을 함을 알 수 있었다.

5.2 상보 필터 분석



〔그림 5.2 빨간색 : RAW, 초록색 : α = 0.995〕



〔그림 5.3 빨간색 : α = 0.95, 초록색 : α = 0.9〕

세 가지 α 값에 대하여 동일한 시간과 동일 장비상에서 측정하고, 이를 분석하기 위해 Serial Oscilloscope 프로그램을 이용했다.

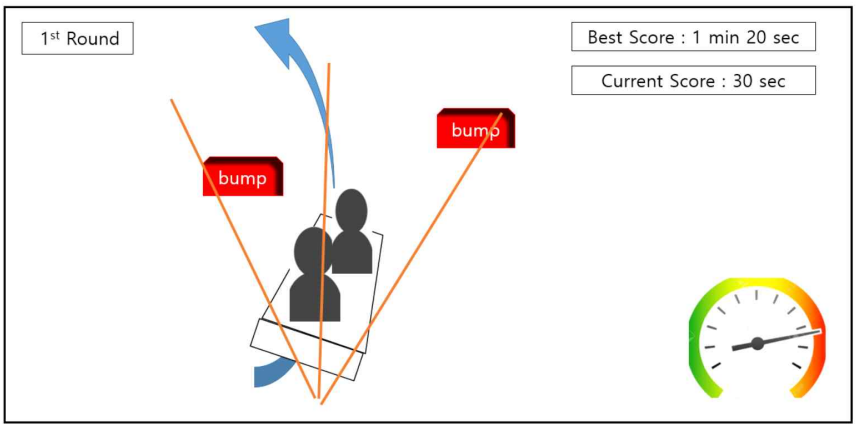
〔그림 5.2〕와 〔그림 5.3〕을 통해 고주파에서 α 의 크기에 따른 노이즈와 실제 값의 변화 추세와 유사도의 ttrade-off가 일어난다는 것을 알 수 있다.

〔그림 5.2〕에서 보이듯이 고주파 움직임에서 α 값이 큰 경우, 변화하는 값의 기울기가 완만해서 실제 움직임과 다른 모습을 보인다.

〔그림 5.3〕에서는 α 가 0.95일 때 노이즈가 적고 변화의 기울기가 유사하나 짧은 순간의 변화 구간에서 실제 움직임 값을 반영할 수 없었다. 반면에 α 가 0.9일 때는 변화의 기울기가 유사하나 측정값의 노이즈가 큰 것을 볼 수있다.

제 6장 하드웨어 및 소프트웨어 적용

6.1 게임 소개



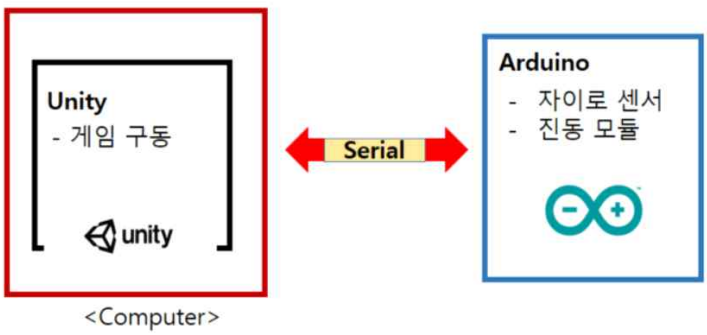
〔그림 6.1 게임 컨셉〕

유니티로 제작한 게임은 2명의 플레이어가 롤러코스터를 타고 코스를 주행하며 진로를 방해하는 장애물을 몸을 기울여 피한다. 코스를 최단 시간내에 완주하는 것이 목표이며 장애물과 충돌 할 때마다 속도가 줄어든다.

게임을 시작 후 머리를 앞으로 기울이면 속도가 증가한다. 기울일 경우 전방의 장애물을 볼 수 없지만 기록을 단축시킬 수 있다. 또한 전방에 장애물이 근접하면 진동으로 장애물의 위치를 알려주는 기능이 있어 화면을 보지않고도 게임을 할 수 있다.

6.2 하드웨어 제작

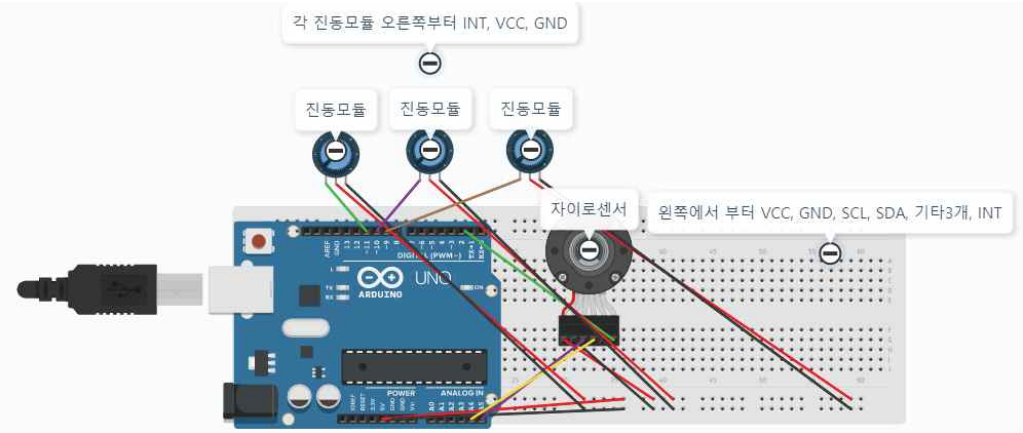
'시스템 모델'



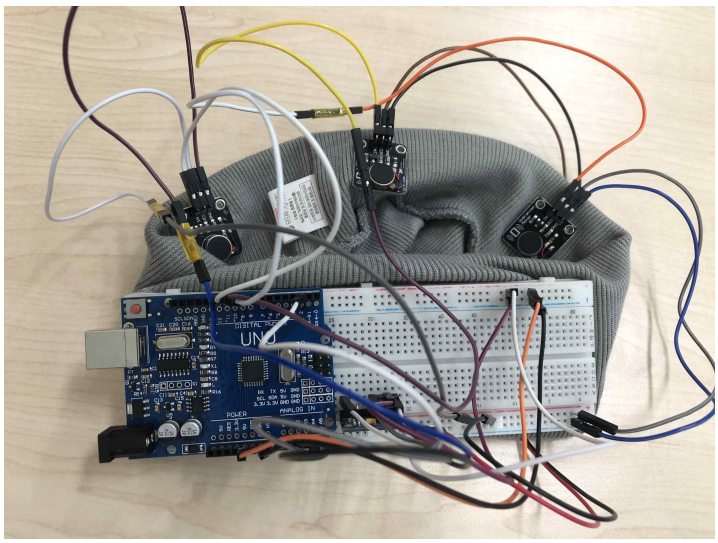
〔그림 6.2 아두이노 유니티 시스템〕

유니티와 아두이노를 serial 통신으로 아두이노에서 MPU-6050으로 측정한 오일러

각을 유니티로 전송하여 게임 내 객체를 조종한다. 게임 플레이중 장애물이 감지되면 유니티에서 장애물의 위치를 3개의 진동센서를 이용해 아두이노에서 표현한다.



〔그림 6.3 컨트롤러 회로도〕



〔그림 6.4 컨트롤러〕

컨트롤러를 머리에 착용하고 진동을 사용자가 인식할 수 있게 헤어밴드에 브레드보드와 아두이노를 결합했다.

6.3 소프트웨어 제작

6.3.1 아두이노 필터 설계

아두이노에 업로드될 필터에 관한 내용은 Appendix [1] 아두이노에 기록되어있다.

6.3.2 아두이노 - 유니티 통신

'아두이노에서 유니티로 정보 전송'

1] 아두이노에서 아두이노와 유니티간 정보 교환은 8bit 데이터를 이용한다. 따라서 0~255 범위의 정수를 이용하여 좌우와 전후의 기울임 정보를 전송했다. 최종 정보는 int형 변수 ang가 전송되며 ang은 좌우와 관련한 int형 변수 rl과 전후와 관련된 int형 변수 fb의 합으로 정해진다.

2] 아두이노에서 자이로센서의 정보를 계속해서 받는다. 아두이노 코드에서 앞으로의 기울임 판단 정보를 0~7중 7번째 bit에 담았다. 따라서 q.y(전후의 기울임 각도)값이 50도 보다 크면 fb=128, 그렇지 않다면 fb=0으로 결정되게 하였다.

3] 좌우의 경우 기울임 정도가 그대로 반영되어야 한다. q.x(좌우의 기울임 각도)값의 범위는 -30~30로 설정했다. 이 정보를 8bit 중 남은 7bit에 담아야 했고, 이 범위는 0~127이다. 그래서 (-30 + 30)*2 = 0 ~ (30+30)*2=120, 결과적으로 rl의 범위를 0~120으로 잡았다. 그리고 rl+fb값을 결정하여 그 값을 유니티로 전송한다.

4] 객체 arduino의 getarddetail.cs에서 전송받은 ang값에서 128로 나눈다. 몫은 플레이어를 앞으로 기울이는데에, 나머지는 플레이어를 좌우로 기울이는데 사용한다. 이 때 몫이 1이면 float형 변수 headdown을 1로 설정하고 0이라면 headdown을 0으로 설정한다. 나머지의 경우, 나머지/2.0 - 30.0을 하여 움직일 각도의 범위를 -30~30으로 만든다. 이 각도를 float형 변수 lr에 저장하고, 이 변수 lr은 객체core의 move.cs 스크립트에서 사용한다.

'유니티에서 아두이노로 정보 전송'

1] 객체 Detect의 detect.cs에서 특정 pre~에 부딪혔을 때 해당 flag들을 0에서 1로 바꾸어준다. 객체 arduino의 getarddetail.cs에서 preL,preM,preR의 flag가 0보다 클 때 아두이노로 각각 A,B,C라는 문자를 전송한다.

2] 아두이노에서 유니티에서 보낸 문자를 확인하고 조건문을 사용하여 A일때는 왼쪽 진동센서, B일때는 중간진동센서, C일때는 오른쪽 진동센서가 진동한다.

6.3.3 유니티에서 객체 조작 - 사용된 함수들 rotate 등

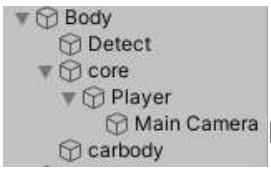
6.3.3.1 주요 object

1) path(트랙)

'Path Creator Asset' 를 사용 하여 길을 생성했다. player는 생성된 path를 따라서 등속운동을 하게 된다.

2) player(플레이어)

'Path Creator Asset' 의 경우 단일 객체가 레일을 이동하며 tilting이 어려웠고 플레이어가 중심이 아닌 객체의 바닥을 기준으로 좌우로 기울여야 하기 때문에 tilting중심을 이동시키기 위해 상속된 empty object를 생성 후에 pivot을 이동 시킨 뒤 역상속 하는 방법을 사용했다. 따라서 path위를 달리는 player object의 경우 단일 객체가 아니라 empty object를 이용한 상속기능으로 병합 객체를 만들었다

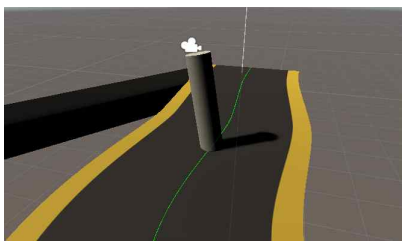


〔그림 6.5 Unity에서의 Player Object 목록〕

- Body(empty object) : 생성한 path에 올리기 위한 오브젝트

- core(empty object) : 회전을 위해 중심을 이동시켰고, BoxCollider를 kinematic의 성질을 가지고 있다. 그래서 위에서의 StartLine과 FinishLine을 인식하는 역할 또한 한다. 그리고 tilting을 위한 스크립트도 컴포넌트로 가지고 있다.

- Player : 게임상에서 움직이는 실린더, 혹은 캡슐 모양의 객체이다. core에 상속되어 있어서 core가 좌우로 기울어지면 같이 기울어진다. path asset의 Road Mesh기능으로 path의 윤곽을 만들었을때 오브젝트가 누워서 가는 것으로 보이는 오류가있어 그것을 조정하기위해 zRotation을 90으로 설정했다. 또한 player의 시점으로 게임이 진행되기 때문에 Main Camera를 상속 시켰다.


〔그림 6.6 Unity내에서 Play 중 Player의 모습〕

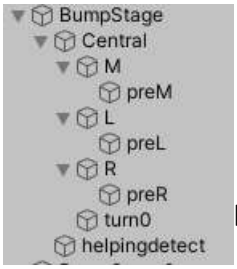
- Detect : 플레이어에게 곧 다가올 장애물의 방향을 인식해주는 역할로 빈 오브젝트에 box콜라이더로만 구성되어있다.

- carbody : Camera의 시점에서 몸체가 기울어졌는지 육안으로 보이게 해주는 단순한 박스 모양의 객체이다.

3) bump(장애물)

BumpStage라는 empty object안에 장애물이 위치하는 공간인 Central과 플레이어에게 장애물의 위치를 알려주는 구간인 helpindetect가 존재한다.

- M,L,R : 실질적인 장애물 객체 로 각각 중간, 왼쪽, 오른쪽에 위치한 장애물이다.
- preM, preL, preR : 각각 M,L,R에 상속된 BoxCollider를 가지는 empty object이다.helpingdetect에 위치하며 플레이어에게 전방의 장애물의 존재 여부를 특정 flag를 통해 알려준다.
- turn0 : preM, preL, preR가 바꾼 flag들을 reset시키는 empty object이다.



〔그림 6.7 Unity에서의 Bumpstage Object 목록〕

4) arduino(empty object)

아두이노에서 전송한 정보들을 이용하여 전후의 tilting, 좌우의 tilting 정보를 결정해주고 그 정보를 다른 스크립트에서 사용가능하게 만들어주는 getarddetail.cs 스크립트가 포함된 empty object이다.

6.3.3.2 주요 동작과 원리

1) 완주시간 측정

Player의 core가 StartLine의 BoxCollider와 부딪히면 시간측정을 시작한다. 그리고 FinishLine의 BoxCollider와 부딪히면 시간측정을 멈추고 속도를 줄인다. core의 move.cs에서 core가 태그가 StartLine인 객체와 부딪히면 시간 측정을 시작하도록 했다. 이는 FinishLine에서도 같은 원리로 적용된다.

2) 장애물 생성과 인식

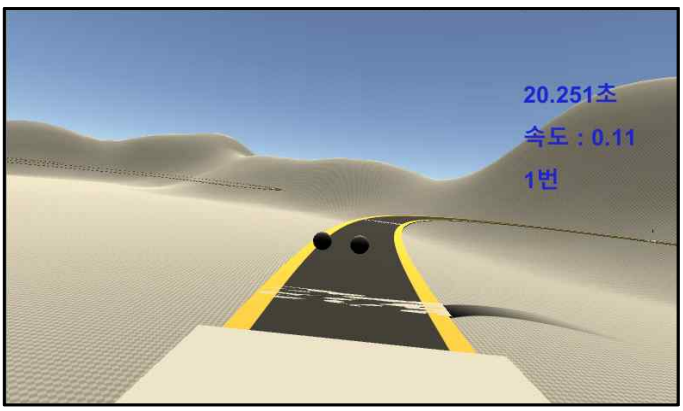
Central의 MakingBump.cs에서 게임이 시작하면 특정 변수 set이 -1,0,1중 하나로 랜덤하게 결정된다. -1일때는 L. 0일때는 M, 1일때는 R을 Destroy시킨다. 그러면 helpingdetect위에는 두개의 pre~만 남게 되고 객체 Detect가 어떤 pre~와 부딪혔는지 정보를 알려준다.객체 Detect가 장애물들을 지나고 뒤의 객체 turn0를 만나면 pre~와 관련된 모든 flag들을 다시 0으로 바꿔준다.

3) 플레이어 조작

- 객체 core의 move.cs에서 플레이어의 좌우 기울임을 결정한다. getarddetail.cs에서의 좌우 각도 정보 lr을 받아서 사용한다. 시변하는 lr값을 객체 core의 Rotation.Z값에 더해주는 방식으로 플레이어의 좌우 기울임을 조절해준다.

- 객체 core의 tiltfront.cs에서 플레이어의 고개의 숙임을 결정시켜준다. getarddetail.cs에서의 숙임 정보 headdown을 받아서 사용한다. tiltfront.cs 스크립트에서 headdown이 0보다 클 때 객체 core의 Rotation.X값에 40을 더해 앞으로 숙이게 하고 다시 headdown이 0이 되면 Rotation.X값을 원래 값인 3으로 돌아오게 한다.

- 객체 body에 포함된 Folloewer.cs는 플레이어의 속도를 결정해준다. 속도에 해당 하는 변수는 float형변수 speed로 public으로 선언되어 있다. tiltfront.cs에서 headdown이 0보다 커져서 앞으로 숙여졌을 때 flag를 받아서 숙여졌을 때 플레이어의 속도를 증가시킨다. 그리고 move.cs에서 플레이어가 장애물에 부딪혔을 때 Folloewer의 public형 변수 .speed에 0.6을 곱해 속도가 감소되도록 설계했다.



〔그림 6.8 완성된 게임 플레이 화면〕

6.4 게임 플레이를 통한 필터 선택

6.4.1 sampling rate 분석

4장의 실험결과를 통해 각 필터의 sampling rate를 측정했다. 따라서 sampling rate가 높을 수록 같은 시간 동안 더 많은 데이터를 처리 할 수 있었고 이를 통해 필터로서 성능이 좋다고 판단했다. 하지만 게임에서 sampling rate가 높은 경우 컨트롤러의 실제 움직임과 게임상에서 크게 차이가 났다. 따라서 sampling rate를 변경해가며 유니티의 데이터 처리능력에 알맞은 sampling rate를 찾는 실험을 했다.

'Sampling Rate 비교 실험'

아두이노 코드상에서 측정된 센서값을 유니티에 전송하는 빈도수를 결정지음으로써 Sampling rate를 조절할 수 있다. 따라서 전달 빈도수를 1번마다, 10번마다, 100번마다 세 가지 경우에서 게임 플레이에 어떻게 영향을 미칠지 실험을 통해 확인했다. 게임 내에서 움직이지 않는 정적인 상황, 좌우로 움직이는 동적인 상황 두 가지 상황에서 sampling rate와 게임 내 동작의 관계를 분석했다.

sampling rate가 350 samples/sec 였던 α =0.985 인 상보필터는 전달 빈도수를 1번, 10번, 100번으로 조절함에 따라 sampling rate는 350, 35, 3으로 변화한다. 따라서 변화된 sampling rate를 기준으로 다음 3가지 실험을 진행했다.

이때 아두이노에서 유니티로 정보를 보낼 때 시리얼 통신을 이용하였는데, 이 시리얼 통신은 8bit 데이터를 보낸다. 이 8bit 내로 보내면 가장 빠른 속도로 데이터 전송이 가능해, 아두이노측에서 얻은 데이터를 8bit 내로 표현하였다.

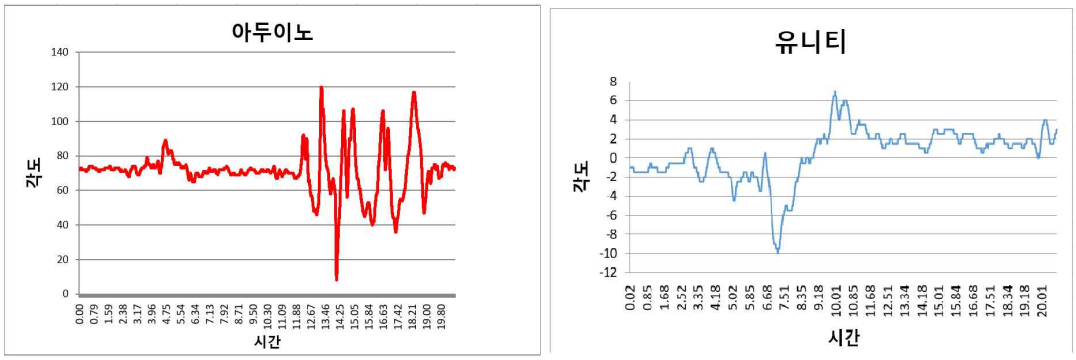
우선 게임에서 숙이는 정보를 1bit로 표현하기 위해 일정 각도를 넘어가면 128, 못 넘으면 0을 더하였다. 유니티에서는 -30도에서 30도 까지만 각도가 변화하기 때문에, 보내는 쪽에서도 그 사이의 각도만 받아 변환을 거쳐주었다.

우선 -30도에서 0도까지는 음수이고, 음수값은 8bit 표현시 보수 표현에 의해 추가 bit가 필요해지므로 모두 양수로 바꾸기 위해 30도를 더해주었다. 그렇게 아두이노에서 정리되는 값은 0~60 이고, 이 0~60을 2배로 한 뒤, 0~120의 값을 강제로 int 형 변환을 통해 Serial Port로 전송하게 된다.

즉 아두이노 상의 60인 경우가 유니티 각도가 0이고, 아두이노 값이 0인 경우가 유니티 각도 -30이며, 아두이노 값이 120인 경우가 유니티 각도 30이 되는 것이다.

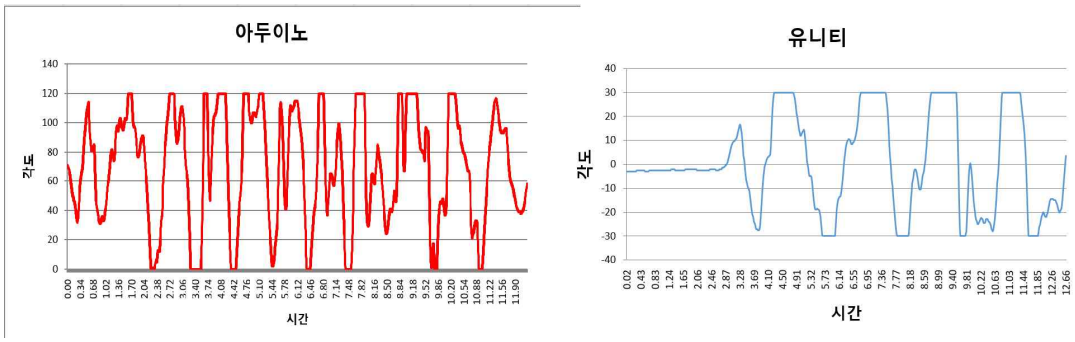
경우 1. Sampling Rate : 350 samples/sec (α =0.985 일 경우)

'정적인 움직임 측정'



〔그림 6.9 정적인 움직임 측정 (sampling rate = 350)〕

'동적인 움직임 측정'



〔그림 6.10 동적인 움직임 측정 (sampling rate = 350)〕

'결과 분석'

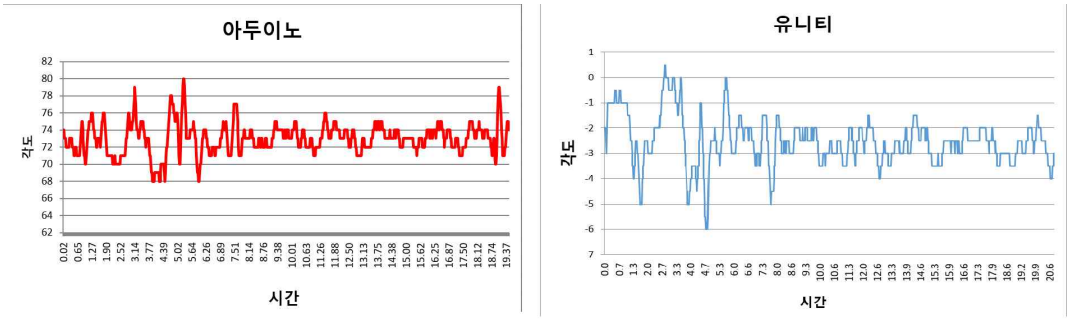
Sampling Rate가 350으로 큰 경우, 유니티와 아두이노에서 표시되는 각도의 변화가 서로 다른 것을 확인하기 위해, 첫 번째로 정적상태에서 확인했다.

또한 유니티상에서 객체가 진동하는 것을 보고, 마지막 구간에서 연결상태를 확인하기 위해 아두이노를 움직임으로써 제대로 연결되었음을 확인했다. 하지만 이 움직임은 유니티에서 정확히 표현되지 않았다.

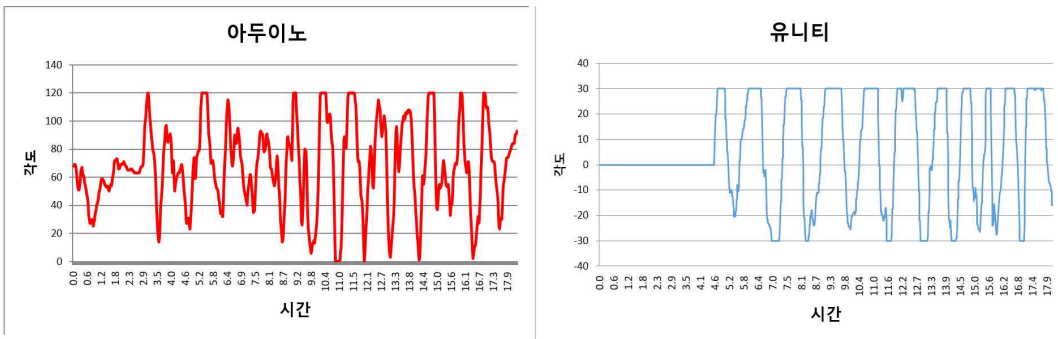
동적인 상태에서 아두이노와 유니티의 각도를 비교했다. 첫 구간부터 아두이노에 고주파의 움직임을 가했으나 유니티에서는 제대로 진동되는 모습이 보이지 않는다. 그리고 일정 시간 이후 유니티에서 입력된 움직임을 따라 값을 출력하는 모습이 보였지만 실제 움직임과 매우 다른 파형을 출력하는 것을 유니티에서 확인했다. 〔그림 6.9〕을 보면 아두이노에서는 약 10번의 진동이 가해졌지만, 유니티상에서는 약 4번만 진동한다,

이 실험을 통해 Sampling Rate가 350으로 너무 높은 경우 유니티와 아두이노 사이의 Serial 통신에 문제가 있음을 알 수 있다.

경우 2. Sampling Rate : 35 samples/sec (α =0.985)



〔그림 6.11 정적인 움직임 측정 (Sampling Rate = 35)〕



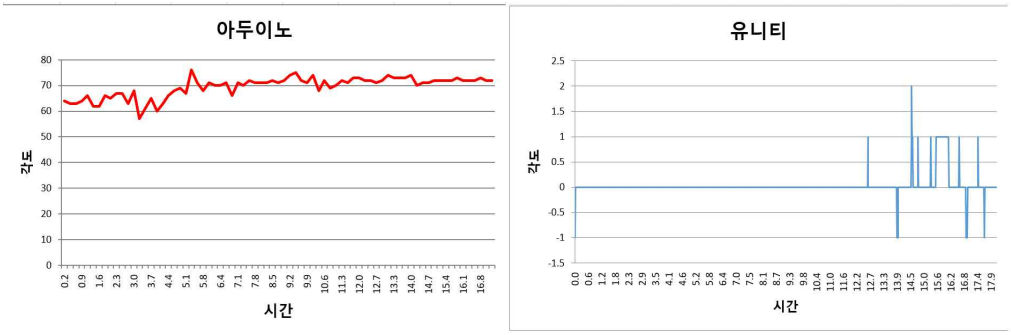
〔그림 6.12 동적인 움직임 측정 (Sampling Rate = 35)〕

'결과 분석'

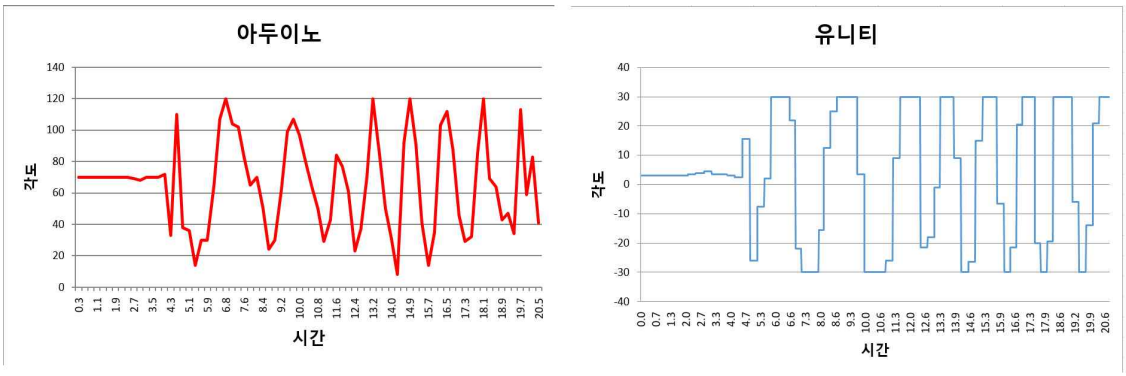
정적인 움직임 측정의 경우에 컨트롤러를 머리에 착용한 상황을 설정하기 위해 센서에 미세한 손떨림을 가했다. 〔그림 6.11〕을 통해 아두이노에 가해진 미세 진동이 유니티에서 유사한 파형으로 출력되는 것을 확인했다. Sampling Rate=350일 때 정적 상황에서, 유니티내 각도의 진동 범위가 -10~6이었는데, Sampling Rate=35로 줄였을 때 각도의 진동 범위가 -6~0으로 이전보다 진동이 줄어든 것을 확인했다.

동적인 움직임 측정의 경우에 게임 플레이 상황을 구현하기 위해 센서의 각도를 급격히 변화시켜 고주파 파형을 유니티와 아두이노의 출력 결과를 비교했다. 이 경우에 유니티에 출력되는 빈도수는 양호해졌으나 값이 정확하지 않아 최적의 Sampling Rate를 찾기 위해 한번 더 실험을 진행했다.

경우 3. Sampling Rate : 3 samples/sec (α =0.985)



〔그림6.13 정적인 움직임 측정 (sampling rate = 3)〕



〔그림 6.14 동적인 움직임 측정 (sampling rate = 3)〕

'결과 분석'

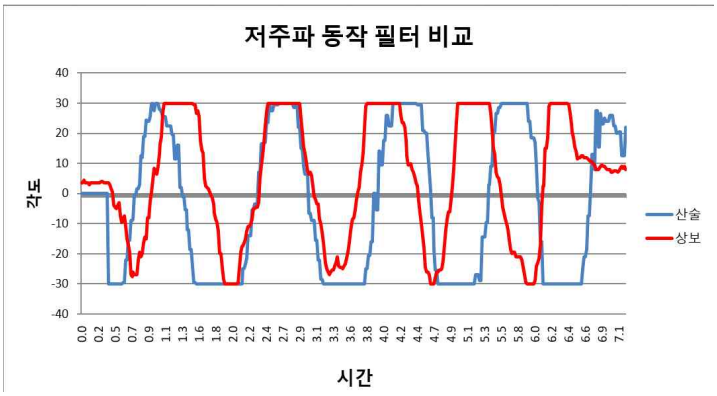
Sampling Rate를 약 3으로 낮춘 경우, 정적인 움직임에서 〔그림 6.13〕 처럼 아두이노에서는 노이즈가 발생했다. 그리고 유니티에서는 아두이노로부터 받은 이 값을 100개당 1회씩 출력하여 전 구간동안 0에 가까운 일정한 값을 출력했다.
유니티에서 각도가 매우 각지게 출력을 확인 할 수 있다. 이는 값이 연속적으로 나오는 것이 아닌, 일정 값을 출력하고 그 값을 유지하는 것이 원인이라 생각한다. 따라서 유니티와 아두이노의 값의 차이를 보이고, 실제로는 아두이노에서 측정된 값들이 코드상의 딜레이에 의해 일정 시간동안 유지된 후 유니티에 출력된다.

'최종 결론'

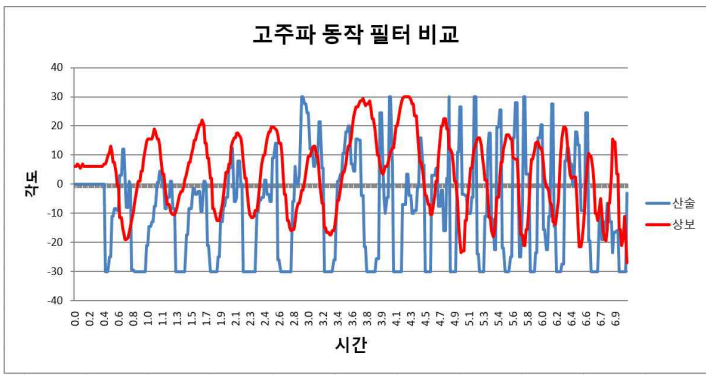
게임상에서 움직임이 없는 정적인 동작에서는 sampling rate가 35와 3일 때 객체가 떨림이 적어 양호한 결과를 보였다. 그리고 동적인 동작에서는 sampling rate가 35 일 때 변하는 움직임을 유니티에서 정확하게 표현했다. 따라서 실험 결과를 바탕으로 게임 플레이에 있어 정적, 동적 움직임에서 양호한 결과를 보인 sampling rate가 35일 때를 적용했다.

6.4.2 산술 필터와 상보 필터 비교

동일한 기울기 측정을 위해 하나의 브레드 보드에 두개의 센서를 부착하고 두 개의 아두이노를 이용했다. 첫번째 아두이노에는 상보필터, 다른 아두이노에는 산술필터를 업로드 하여 같은 입력되는 기울기값에 대해 유니티상에서 객체가 움직임을 표현하는 양상을 측정했다.



〔그림 6.15 상보필터와 산술평균 필터 저주파 동작 비교〕



〔그림 6.16 상보필터와 산술평균 필터 고주파 동작 비교〕

‘결과 분석’

게임을 실행하고 컨트롤러 〔그림 6.15〕와 같이 산술평균 필터는 동적인 움직임에서 상보필터보다 실제 움직임 값을 제대로 표현하지 못했다. 움직이는 속도가 증가할 수록 두 필터로부터 출력된 결과값의 차이는 증가했다. 따라서 산술평균 필터로 게임을 하는 경우 급격히 장애물을 피하려는 경우에 객체가 피크에서 피크로 움직이지 못해 충돌 횟수가 증가했다.

‘최종 결론’

고주파 움직임을 크게 제한하는 산술평균 필터는 노이즈 감소에 있어 우수하다. 하지만 급격한 움직임을 불가능하게 하는 Low pass filtering 특성은 게임 플레이에 부적합함을 알 수 있어 상보필터를 게임에 적용하기로 했다.

6.4.3 상보 필터 계수 설정

4장과 5장의 동작의 주파수에 따른 특성 실험에서 Serial Oscilloscope를 통해 상보필터 계수 α 값이 저주파와 고주파 동작에 미치는 영향을 알았다. 실험 결과를 정리하면 다음 표와 같다.

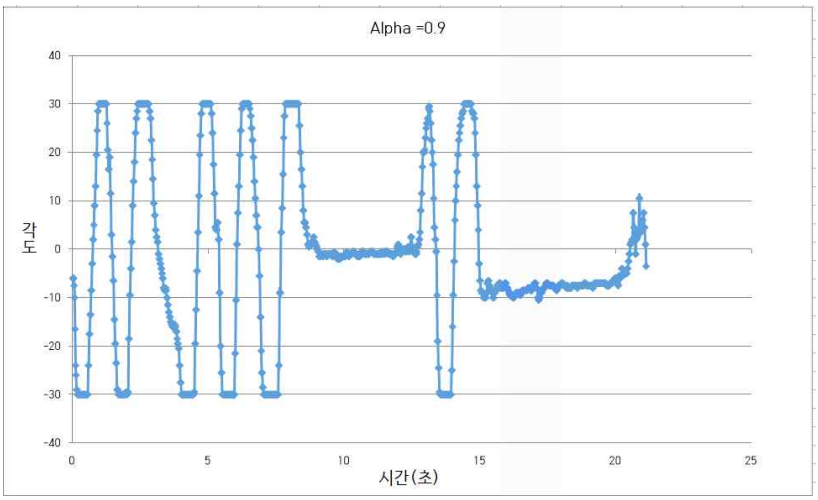
α 값	저주파 동작	고주파 동작
증가 (0.995)	상대적 높은 정확도	Low pass filtering이 강해서 raw값의 고주파 성분 왜곡
감소 (0.90)	상대적 낮은 정확도	Raw 값의 변화 추이를 유사하게 표현함

6.4.1의 sampling rate 비교 분석실험을 통해 유니티로 전송된 아두이노의 데이터는 sampling rate에 영향을 받는다는 것을 확인했다. 따라서 이번 실험에서는 아두이노로 부터 받은 값을 α 값을 변경하며 유니티에서 측정했다.

‘α 값 비교실험’

아두이노 코드에서 α 값은, 상보필터에서 현재와 이전 값의 종합적 계산과정에서 현재값의 비중을 나타낸다. 이 α 값을 0.9, 0.95, 0.995 세가지 경우에서 게임플레이에 어떤 영향을 가져올지 확인했다. α 값에 따른 정적 상황 분석은 4장에서 진행하였기에 게임 중 동적으로 움직임의 영향 중심으로 확인했다.

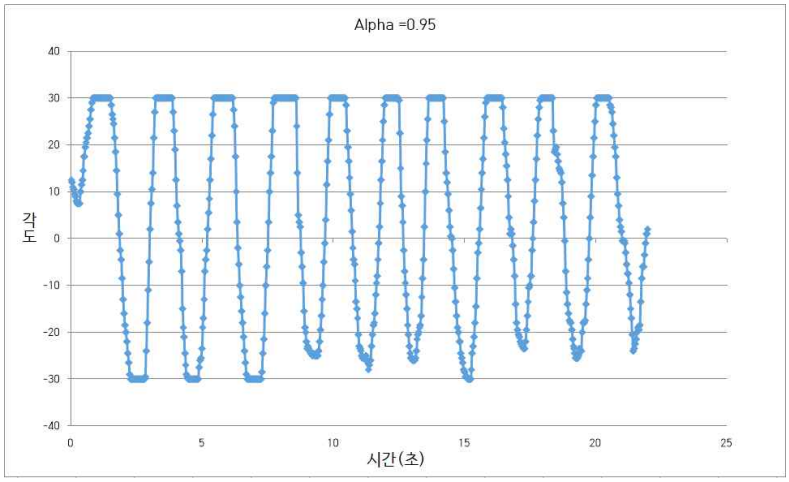
경우 1. α =0.9 일 경우 (처리 data : 1221개, 총 시간 : 21.12초)



〔그림 6.17 Alpha = 0.9〕

α = 0.9인 경우 변하는 입력 값에 대해서 정확히 출력하지만 정지 동작의 경우 미세한 진동을 보였고 〔그림 6.17〕의 중간 구간에서 값이 흔들려 게임 내 객체가 진동했다

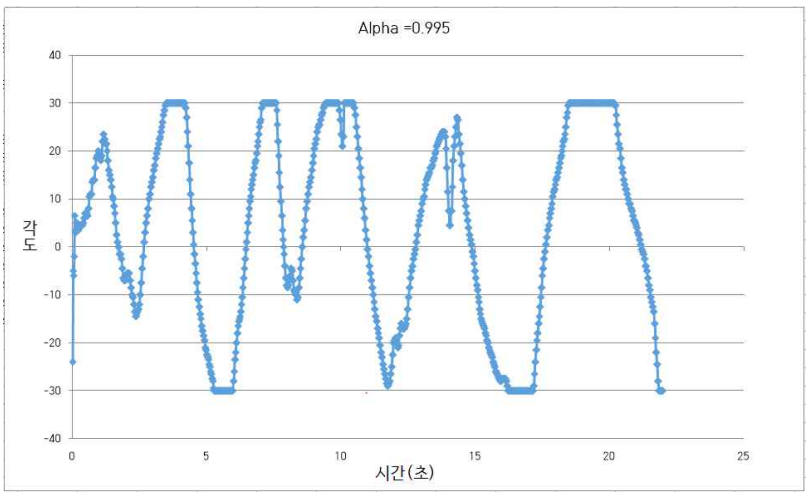
경우 2. α =0.95 인 경우 (처리된 data : 1221개, 총 시간 : 21.71초)



〔그림 6.18 Alpha = 0.95〕

α = 0.95인 경우 진동에 대하여 잘 움직이고, 전체 구간에서 의도치 않은 피크값이 없이, 깔끔한 모습을 보였다.

경우 3. α =0.995 인 경우 (처리된 data : 1221개, 총 시간 : 22.07초)



〔그림 6.19 Alpha = 0.995〕

α = 0.995인 경우 파형이 매우 불안정하게 출력되었다. 이는 α =0.995인 경우에 수신 값을 처리해서 각도가 변화하기에 시각적인 딜레이가 약 0.5s정도로 느리기 때문에 이러한 모습을 보인 것으로 추정된다.

	Peak to peak 처리 데이터 개수 (소요 시간)
α =0.9	25개(=0.432초)
α =0.95	43(=0.764초)
α =0.995	61(=1.102초)

위 표를 통해 α 가 클 수록 최고점에서 최저점 까지 갈 때 까지의 처리 가능한 데이터 개수가 증가하는 것을 알 수 있다. 실험과정에서 하나의 data가 나타나는 시간은 동일하기 때문에 이는 최고점에서 최저점까지 이동하는데에 걸린 시간이 길어졌다는 것을 의미한다.

실제로 α =0.995일 때 게임상 플레이어의 진동이 전혀 없이 부드럽게 움직였지만 좌우를 이동하는 속도가 1초 정도로 느려서 장애물을 제때 피할 수 없어 누적 충돌 횟수가 증가했다.

반대로 α =0.9일 때, 좌우 이동이 컨트롤러의 움직임을 유사하게 반영하여 움직였지만 객체의 진동이 너무 심해서 가만히 있을 때에도 좌우로 격렬히 움직이는 모습을 보였다.

제 7장 결론

7.1 필터 선택의 이유

1) 본 논문에서는 센서 융합 상보 필터를 이용한 게임 컨트롤러 개발 및 성능을 개선하였다. 우선 제작한 컨트롤러의 분석을 위해 게임을 고안했다. 게임의 경우 센서로 측정하는 속도가 계속 바뀌기 때문에 조건에 맞는 필터를 설계 하기 위해 실험을 진행했다. 4,5,6장의 실험을 통해 최적의 필터를 설계하기 위해서 필터의 계수도 중요하지만 기기의 통신에 있어서 데이터를 이용하는 sampling rate 또한 중요한 것을 알았다. 최적의 필터라 결정한 상보필터에서 α 값과 정확도의 상관관계를 측정했고 sampling rate와 유니티 객체의 딜레이를 측정했다. 이를 통해 최적의 계수를 구할 수 있었다.

2) 각속도만을 이용하였을 때 비해 상보 필터를 적용하여 측정하는 속도가 느려지는 현상이 발생한다. 따라서 적용 분야에 맞춰 속도가 중요시 된다면, 상보 필터를 적용하지 않고 단순 센서 값만을 사용하여도 될 것이다. 하지만 속도 지연 문제를 제외한다면 상보 필터를 사용하는 것이 효과적인 방법임이 분명하다.

7.2 연구의 한계점

1) 초기 게임의 목적은 시각장애인과 청각장애인 상관없이 즐겁게 즐길 수 있게 하드웨어와 소프트웨어를 제작하는 것이였다. 더 나아가 시각 장애인과 청각 장애인이 같이 협동하는 환경을 준비중이었다. 연구 초기에 2개의 아두이노를 블루투스로 연결하여 두 자이로 값을 동시에 계산하여 최적화된 값을 이용하려 하였다.

2) 안드로이드 디바이스와 아두이노의 블루투스를 이용하여 값을 전송해 VR로 구현하는 것이 목표였다. 하지만 유니티 코드를 이용하여 아두이노와 블루투스 연결이 원활하지 못했고, 아두이노 두개를 이용해 한 유니티에 연결하는 것 또한 제대로 작동하지 못했다. 이에 따라 VR 또한 불가피하게 포기하였다.

3) MPU6050 센서에 칼만필터를 이용한 결과와 상보필터를 이용한 결과를 갖고 성공적인 필터링이 되는지 확인해보려 하였으나, 칼만필터의 경우 설계에 난항을 겪었다. 그래서 상보필터를 이용한 결과와 실제 결과 및 산술필터를 이용한 결과를 비교하는 쪽으로 변경하였다.

4) 아두이노 자체적으로 전압이 부족해 자이로센서와 진동모듈을 동시에 적용시키기에 문제가 있었고, 그리하여 불가피하게 delay를 사용하여 자이로센서를 잠시동안 멈추고 진동을 입력하는 방식을 사용하였다. 또한 한 개의 센서를 이용하여 두 개의 컴퓨터에서 결과를 확인하려고 하였으나, 하드웨어 구조적 문제로 인해 불가능하여 두개의 센서를 같은 브레드보드 위에 올리고 두 컴퓨터에서 측정하였다.

5) 자이로 센서의 scale factor error를 예상하였으나, 실험결과 실제로는 해당 문제를 발견하지 못하였다. 이는 센서 보정과정에서 효과적으로 누적 적분 오차를 제거했음을 알 수 있었다.

7.3 추가 제안

1) 칼만필터 : 향상된 센서 융합 필터

칼만 필터는 상보필터의 필터계수 α 를 실시간으로 변화하여 정확도와 속도가 우수하다. 시간이 더 있었다면 이를 적용해 볼 수 있을 것이다.

본 프로젝트에서는 센서값의 보정을 위해 상보필터를 선정했다. 상보필터의 α 값 선정과정에서 α 값을 작게 하면 반응속도가 빠르고 실제 값과 가까웠지만 노이즈 필터링의 역할을 제대로 하지 못했다. 반대로 α 값을 1에 가깝게 할수록 노이즈가 제거 되고, 정적상태에서의 진동 현상을 완화 했지만 반응속도가 느려져서 게임의 진행이 힘든 모습을 보였다. 그래서 어느정도 적절한 α 값을 선정하기 위해 여러 가지 시행 착오를 했었다.

칼만필터의 경우 수식은 상보필터와 유사하지만 가장 큰 차이점은 α 값이 시간에 따라 바뀌는 점이다. 상황에 따라 α 값을 변화시키며 게임을 진행 한다면, 노이즈 감쇠도 잘되면서 반응속도 까지 빠른 게임 컨트롤러를 제작할 수 있을 것이다.

2) 본 프로젝트에서는 MPU-6050(GY-521)을 사용하였다. GY-521은 자이로센서와 가속도 센서만을 사용하여 각도를 계산할 수 있게 해준다. 이번 연구는 가속도 센서와 자이로 센서 두개만을 이용해서 값을 보정했다. DMP를 이용하면 이 두 센서 말고 지자계 센서나 기압 센서를 이용해 정확하고 연산량이 적게 값을 보정 할 수 있다.

참고문헌

- 상보필터 관련 - 가속도, 자이로 센서 원리
- 칼만필터 관련

〔1〕 김성원(2014), “센서 융합 칼만 필터를 이용한 휴대용 오십견 치료 기기 개발”, 석사 학위 논문, 부경대학교

〔2〕 Pierre-Jean Bristeau, François Callou, David Vissière, Nicolas Petit(2011), “The Navigation and Control technology inside the AR.Drone micro UAV”, 18th IFAC World Congress

Appendix

[1] 아두이노 스크립트

```
#include <Wire.h>
const int MPU_addr = 0x68;
int r1, fb, ang;
int16_t ax, ay, az;
int16_t gx, gy, gz;
float p,q,r;
float pi, ce, wo;
float ppi, pce, pwo;
int count;
void setup() {
  Wire.begin();
  Wire.beginTransmission(MPU_addr);
  Wire.write(0x6B);
  Wire.write(0);
  Wire.endTransmission(true);
  Serial.begin(115200);

ax=0;ay=0;az=0;gx=0;gy=0;gz=0;p=0;q=0;r=0;pi=0;ce=0;wo=0;ppi=0;pce=0;pwo=0;r1=0;f
b=0;ang=60;
  count=0;
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(11, OUTPUT);
}

void loop() {
  GetGyro(ax, ay, az, gx, gy, gz);
  CalW(gx,gy,gz,p,q,r);
  CalAngle(ax, ay, az, pi, ce, wo);
  CalCom(pi, ce, p, q, r,ppi,pce);
  if(pce>-30 && pce<30){    r1 = (pce+30)*2;  }
  else if(pce<-30){    r1 = 0;  }
  else if(pce > 30){    r1 = 120;  }
  if(ppi < -50){    fb = 128;  }
  else{    fb = 0;  }
  ang = r1 + fb;
  if(count>=10){    count=0;    Serial.write(ang);    Serial.flush();  }
  count+=1;
  int a = Serial.read();
  if(a=='A')   analogWrite(9, 255);
  else  analogWrite(9, 0);
  if(a=='B')   analogWrite(10, 255);
  else  analogWrite(10, 0);
  if(a=='C')   analogWrite(11, 255);
  else  analogWrite(11, 0);
  //delay(15);            //딜레이를 주면 진동의 세기가 더 커짐, 그 이유는 딜레이 동안
입력을 받지않아 오롯이 그 행동에만 사용 가능하기 때문
}
//----------------------Get_Gyro_Raw-------------------------//

void GetGyro(int16_t &ax1, int16_t &ay1, int16_t &az1, int16_t &gx1, int16_t &gy1,
int16_t &gz1){
  int16_t tmp1;
  Wire.beginTransmission(MPU_addr);
  Wire.write(0x3B);
```

```
  Wire.endTransmission(false);
  Wire.requestFrom(MPU_addr,14,true);
  ax1=Wire.read()<<8|Wire.read();
  ay1=Wire.read()<<8|Wire.read();
  az1=Wire.read()<<8|Wire.read();
  tmp1=Wire.read()<<8|Wire.read();
  gx1=Wire.read()<<8|Wire.read();
  gy1=Wire.read()<<8|Wire.read();
  gz1=Wire.read()<<8|Wire.read();
}

//-----------------------------------------------------------//

//--------------Calculating Angular Velocity-----------------//

void CalW(int16_t gx1, int16_t gy1, int16_t gz1, float &p1, float &q1, float &r1){
  p1 = (float)(gx1)/131.0;
  q1 = (float)(gy1)/131.0;
  r1 = (float)(gz1)/131.0;
}

//-----------------------------------------------------------//

//---------------------Calculating Angle---------------------//

void CalAngle(int16_t ax1, int16_t ay1, int16_t az1, float &pi1, float &ce1, float
&wo1){
  pi1 = atan(ax/(sqrt(pow(ay,2)+pow(az,2))))*180/3.141592;
  ce1 = atan(ay/(sqrt(pow(ax,2)+pow(az,2))))*180/3.141592; ce1+=20;
  wo1 = atan((sqrt(pow(ax,2)+pow(ay,2)))/az)*180/3.141592;
}

//-----------------------------------------------------------//

//---------------Calculating Complementary Value-------------//

void CalCom(float pi1, float ce1, float p1, float q1, float r1, float &ppi1, float
&pce1){
  ppi1 = 0.96*(ppi1 + p1/80.0) + 0.04*pi1;
  pce1 = 0.985*(pce1 + q1/80.0) + 0.015*ce1;
}

//-----------------------------------------------------------//
```

〔2〕 Unity 스크립트

〔2-1〕 Folloewer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using PathCreation;

public class Folloewer : MonoBehaviour
{
    public PathCreator pathCreator;
    public float speed = 0;      //플레이어의 속도
    float distanceTravlled;

    float maxspeed = 1.5f;
    int limiting = 0;
    int addVelo = 0;
    int stopbody = 0;

    GameObject Veloc;            //UI객체
    GameObject limit;            //객체 core의 스크립트들을 사용하기 위해 선언

    void Start()
    {
        this.Veloc = GameObject.Find("Velo");
        this.limit = GameObject.Find("core");
    }
    void Update()
    {
        addVelo = this.limit.GetComponent<tiltfront>().plus;
        //플레이어의 고개숙임 flag

        stopbody = this.limit.GetComponent<move>().stop;
        //결승선 통과 여부 flag

        distanceTravlled += speed;
        transform.position = pathCreator.path.GetPointAtDistance(distanceTravlled);
        transform.rotation = pathCreator.path.GetRotationAtDistance(distanceTravlled);

        if(stopbody >0)          //플레이어가 결승선을 통과했을 때
        {
            speed *= 0.99f;
        }

        if(addVelo>0 && speed < maxspeed)    //플레이어의 머리가 숙여졌을 때
        {
            speed += 0.0008f;
        }

        //속도를 UI에 표시
        this.Veloc.GetComponent<Text>().text = "속도 : " + speed.ToString("F2");
    }
}
```

〔2-2〕 move.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class move : MonoBehaviour
{
     //객체들
    public Rigidbody player;
    public Rigidbody core;
    public Rigidbody body;

    //이동속도
    float runspeed = 1.0f;
    float maxspeed = 20.0f;

    //태그 인식
    public int count;
    float time = 0;
    public int stop;
    float bumping = 0;

    //기록시간 & 현속도 UI
    GameObject Record;
    GameObject Score;

    //bump후 스피드 저하
    GameObject center;

    //ard송수신
    GameObject commu;

    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "StartLine") //출발선 통과했을 때
        {
            this.count = -this.count;     //기록시간 측정 시작
            this.stop = 0;
            this.time = 0;
            this.time += Time.deltaTime;
        }
        else if (other.gameObject.tag == "FinishLine")    //결승선 통과했을 때
        {
            this.count = 1;               //기록시간 측정 종료
            this.stop = 1;
            this.time += 0;
        }
        else if (other.gameObject.tag == "BumpM")      //장애물 M과 부딪혔을 때
        {
            this.center.GetComponent<Folloewer>().speed *= 0.6f;
            this.bumping += 1;
        }
        else if (other.gameObject.tag == "BumpL")      //장애물 L과 부딪혔을 때
        {
            this.center.GetComponent<Folloewer>().speed *= 0.6f;
            this.bumping += 1;
```

```
        }
        else if (other.gameObject.tag == "BumpR")       //장애물 R과 부딪혔을 때
        {
            this.center.GetComponent<Folloewer>().speed *= 0.6f;
            this.bumping += 1;
        }
    }

    // Start is called before the first frame update
    void Start()
    {
        this.Record = GameObject.Find("TimeRecord");
        this.Score = GameObject.Find("bump");
        this.center = GameObject.Find("Body");
        this.commu = GameObject.Find("arduino");
        this.count = 1;
        this.stop = 0;
    }

    // Update is called once per frame
    void Update()
    {
        if (this.count < 0)
        {
            this.time += Time.deltaTime;          //기록시간 측정
        }

        //좌우Control
        float direc = this.commu.GetComponent<getarddetail>().lr;

        Vector3 now = this.body.transform.rotation.eulerAngles;
        transform.rotation = Quaternion.Euler(3.0f, now.y, direc);

        //기록UI
        this.Record.GetComponent<Text>().text = time.ToString("F3") + "초" ;
        this.Score.GetComponent<Text>().text = bumping.ToString("F0") + "번";

        //정지 후 다시시작
        if (stop > 0 && Input.GetMouseButtonDown(0))
        {
            SceneManager.LoadScene("GameScene");
        }
    }
}
```

[2-3] tiltfront.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class tiltfront : MonoBehaviour
{
    GameObject commu;
    public Rigidbody core;
    public int plus;

    // Start is called before the first frame update
    void Start()
    {
```

```
        this.commu = GameObject.Find("arduino");
        this.plus = 0;
    }

    // Update is called once per frame
    void Update()
    {
        Vector3 ang = this.core.transform.rotation.eulerAngles;
        float down = this.commu.GetComponent<getarddetail>().headdown;

        if (down > 0)
        {
            if (transform.rotation.eulerAngles.x < 50)
            {
                transform.rotation = Quaternion.Euler(ang.x + 40, ang.y, ang.z);
                this.plus = 1;    //속도 증가 flag
            }
        }
        else
        {
            transform.rotation = Quaternion.Euler(3.0f, ang.y, ang.z);
            this.plus = 0;
        }
    }
}
```

[2-4] getarddetail.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO.Ports;

public class getarddetail : MonoBehaviour
{
    // Start is called before the first frame update
    public float lr;
    public float headdown;

    SerialPort sp = new SerialPort("COM5", 115200);

    public Rigidbody core;
    public Rigidbody body;

    GameObject warning;

    int start = 0;
    void Start()
    {
        this.warning = GameObject.Find("Detect");
        this.lr = 0;
        this.headdown = 0;
        sp.Open();
        sp.ReadTimeout = 1;
    }

    // Update is called once per frame
    void Update()
    {
```

```
        float signm = this.warning.GetComponent<detect>().m;
        float signl = this.warning.GetComponent<detect>().l;
        float signr = this.warning.GetComponent<detect>().r;
        if (Input.GetKey("g"))            sp.Write("a");
        if (signl > 0)                sp.Write("A");
        if (signm > 0)                 sp.Write("B");
        if (signr > 0)                sp.Write("C");

        if (sp.IsOpen)
        {
            try
            {
                MoveObject(sp.ReadByte());
            }
            catch (System.Exception)
            {
                throw;
            }
        }

    }

    void MoveObject(int Direaction)
    {
        int fbflag = Direction/128;
        if (fbflag == 1)  //전
        {
            headdown = 1;
        }
        else headdown = 0;  //후


       int rlflag = Direaction % 128;
       this.lr = ((float)rlflag /2.0f-30.0f) * (-1);
    }

}
```

[2-5] Makingbump.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Makingbump : MonoBehaviour
{
    public int set;
    private void Awake()
    {
        set = Random.Range(-1, 2);
    }
}
```

[2-6] makingL.cs // makingM.cs // makingR.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class makingL : MonoBehaviour
{
```

```
    public GameObject L;
    public GameObject center;

    void Start()
    {
        if (this.center.GetComponent<Makingbump>().set < 0)
        {
            Destroy(gameObject);
        }
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class makingM : MonoBehaviour
{
    public GameObject M;
    public GameObject center;

    // Start is called before the first frame update
    void Start()
    {

        if(this.center.GetComponent<Makingbump>().set ==0 )
        {
            Destroy(gameObject);
        }
    }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class makingR : MonoBehaviour
{
    public GameObject R;
    public GameObject center;

    // Start is called before the first frame update
    void Start()
    {
        if (this.center.GetComponent<Makingbump>().set > 0)
        {
            Destroy(gameObject);
        }
    }
}
```

[2-7] detect.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class detect : MonoBehaviour
{
    public float m;
    public float l;
    public float r;

    void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "preBumpM")
        {
            //Debug.Log("M!!!");
            this.m = 1;
        }
        else if (other.gameObject.tag == "preBumpL")
        {
            //Debug.Log("L!!!");
            this.l = 1;
        }
        else if (other.gameObject.tag == "preBumpR")
        {
            //Debug.Log("R!!!");
            this.r = 1;
        }
        else if (other.gameObject.tag == "Reset")
        {
            this.m = 0;
            this.l = 0;
            this.r = 0;
        }
    }

    // Start is called before the first frame update
    void Start()
    {
        this.m = 0;
        this.r = 0;
        this.l = 0;
    }


}
```